02 Aug 2018

Algorithms CS2500

Simone Silvestri
Missouri University of Science and Technology, silvestris@mst.edu

Ken Goss

Zhishan Guo
Missouri University of Science and Technology, guozh@mst.edu

Ashikahmed Bhuiyan

Algorithms CS2500

Simone Silvestri, Ken Goss, Zhishan Guo, Ashikahmed Bhuiyan

August 2, 2018

Contents

1 Introduction to Algorithms **4**
- 1.1 Algorithms design & analysis, why do we care? . . . 4
- 1.2 Pseudo-code . . . 5
- 1.3 Introduction to Complexity Analysis . . . 5
  - 1.3.1 Insertion sort . . . 6
  - 1.3.2 Example of Execution . . . 6
  - 1.3.3 Complexity Analysis . . . 7
- 1.4 Asymptotic Complexity Analysis . . . 9
  - 1.4.1 Big O Notation . . . 9
  - 1.4.2 Big Ω Notation . . . 10
  - 1.4.3 Big Θ Notation . . . 11
  - 1.4.4 Algebra of Asymptotic Notation . . . 12
- 1.5 The Search Problem: Binary search . . . 13
  - 1.5.1 Example . . . 13
- 1.6 Recurrence Equations . . . 14
  - 1.6.1 Recursive Tree Method . . . 14
  - 1.6.2 A More Complex Example . . . 16
  - 1.6.3 Another Example . . . 17
  - 1.6.4 The Master Theorem . . . 18
- 1.7 Exercises . . . 19

2 Divide and Conquer **21**
- 2.1 Merge Sort . . . 21
  - 2.1.1 Pseudo-code . . . 22
  - 2.1.2 Complexity . . . 24
- 2.2 Maximum Sub-array Problem . . . 25
  - 2.2.1 Divide and Conquer Solution . . . 26
  - 2.2.2 Complexity . . . 28
- 2.3 QuickSort . . . 29
  - 2.3.1 Pseudo-code . . . 29
  - 2.3.2 Complexity . . . 30
  - 2.3.3 Selection in Worst-Case Linear Time . . . 32
- 2.4 Counting Sort - Sorting in Linear Time . . . 33

2.4.1 Pseudo-code . . . 33
2.4.2 Complexity . . . 33
2.5 Exercises . . . 35

3 Greedy Approach to Algorithm Design **38**
3.1 The Activity Selection Problem . . . 38
3.1.1 An Example . . . 39
3.1.2 Potential Greedy Solutions . . . 39
3.1.3 Proving Greedy Algorithm Correctness . . . 41
3.1.4 More Detailed Psedo-code for Earliest Termination . . . 43
3.2 The Cashier Problem . . . 43
3.2.1 Examples in the US System . . . 44
3.2.2 Greedy Solution . . . 44
3.3 The Knapsack 0-1 Problem . . . 44
3.3.1 Greedy Solution . . . 45
3.3.2 Counter Examples and NP-Completeness . . . 45
3.4 Exercise . . . 46

4 Dynamic Programming **47**
4.1 General Dynamic Programming Approach . . . 47
4.2 Max Sub-array Problem . . . 47
4.2.1 Pseudo-code . . . 48
4.2.2 Example . . . 49
4.2.3 Pseudo-code to Find the Actual Solution . . . 49
4.3 Longest Common Subsequence . . . 50
4.3.1 Dynamic Programming Solution . . . 50
4.3.2 LCS Pseudo-code . . . 51
4.3.3 Example . . . 51
4.3.4 PrintLCS Pseudo Code and Execution . . . 53
4.3.5 Example - Actual LCS . . . 53
4.4 Dynamic Programming Solution for the knapsack problem . . . 54
4.4.1 Pseudo-code . . . 55
4.4.2 Example . . . 55
4.4.3 Calculate the Actual Solution . . . 56
4.4.4 A Discussion on Computational Complexity . . . 57
4.5 Exercise . . . 57

5 Graphs **60**
5.1 Definitions . . . 60
5.2 Representation of Graphs . . . 61
5.2.1 Adjacency Lists . . . 61
5.2.2 Adjacency Matrix . . . 62
5.2.3 Complexity Comparison of Common Operations . . . 63
5.3 Depth First Search - DFS . . . 63

5.3.1 Pseudo-code . . . 64
5.3.2 Example . . . 65
5.4 Directed Acyclic Graphs (DAG) and Topological sort . . . 65
5.4.1 Pseudo-code . . . 66
5.4.2 Example . . . 67
5.5 Breadth First Search (BFS) . . . 69
5.5.1 Pseudo-code . . . 70
5.5.2 Example . . . 70
5.6 Strongly Connected Components . . . 71
5.6.1 SCC Example . . . 72
5.7 Red-Black Trees . . . 73
5.7.1 Properties of a Red-Black Tree . . . 73
5.7.2 Rotation . . . 74
5.7.3 Deletion . . . 76
5.8 Minimum Spanning Trees . . . 78
5.8.1 Kruskal's Algorithm . . . 80
5.8.2 Efficient Implementation of Kruskal's Algorithm . . . 82
5.8.3 Prim's Algorithm . . . 82
5.9 Single Source Shortest Paths . . . 92
5.9.1 Optimality Principle . . . 92
5.9.2 Dijkstra's Algorithm . . . 93
5.9.3 Bellman-Ford: Shortest Path with Negative Weights . . . 98
5.10 Floyd-Warshall Algorithm: All Pairs Shortest Path . . . 102
5.10.1 Pseudo-code . . . 104
5.10.2 Example . . . 104
5.10.3 Constructing the Shortest Path . . . 105
5.11 Maximum Flow . . . 108
5.11.1 Ford-Fulkerson Algorithm . . . 110
5.11.2 Example . . . 110
5.11.3 MaxFlow-MinCut Theorem . . . 112
5.12 Bipartite Matching Problem . . . 113
5.12.1 Algorithm for Maximum Matching . . . 113
5.13 Exercise . . . 115

Chapter 1

Introduction to Algorithms

This course provides the basics for the design and analysis of algorithms. Algorithms are generally designed to solve specific problems automatically, i.e. executable by a computer. This requires the definition of an algorithm as a set of *instructions*, which constitute the *code* of the algorithm. Instructions are executed on the *input*, in order to provide the desired *output*, possible with the use of supporting *data structures*.

A typical problem solved by an algorithm is *sorting*, which takes a sequence of numbers and returns the same numbers sorted in increasing (or decreasing) order. In the following section we use sorting to motivate the need of designing efficient algorithms and necessity of tools to analyze their complexity.

1.1 Algorithms design & analysis, why do we care?

Let us formally define the sorting problem.

Definition 1.1.1 (Sorting problem)**.** *Given n numbers $< a_1, \ a_n >$ find a permutation of these numbers such that the numbers are in increasing order.*

As we will see in the next sections, there are, among others, two solutions to the sorting problem, named *Insertion Sort* and *Merge Sort*. Let n be the length of the sequence being sorted, these algorithms have the following *time complexity*:

- Insertion Sort: $c_1 n^2$
- Merge Sort: $c_2 n \log n$

where c_1 and c_2 are constants, such that usually $c_1 < c_2$. This means that the time required to execute Insertion Sort on a sequence grows quadratically as a function of the sequence length n, while for Merge sort the growth is $c_2 n \log n$. The constants depend on the specific implementation and computer on which the code is compiled and executed.

Let us now compare the two algorithms in terms of execution time by using two computers, one, very recent and powerful, and one that I used to play with when I was 4 years old.

- **Computer A** IntelCore i7 (2015), $\approx 10^{11}$ instructions per second, executes Insertion sort
- **Computer B** Intel 386 (1985) $\approx 10^7$ instructions per second, executes Merge sort
- We also assume that the young and cool owner of Computer A is a better programmer than the nostalgic owner of Computer B, thus $c_1 < c_2$, and in particular $c_1 = 2$ and $c_2 = 50$.
- We consider a sequence of $n = 10^8$ elements

The time to execute Insertion sort on Computer A is:

$$T_A = \left(\frac{2(10^8)^2}{10^{11}}\right) = 2 \cdot 10^5 s \approx 5.5 hours$$

While the time to execute Merge sort on Computer B is:

$$T_B = \left(\frac{50 \cdot 10^8 \cdot log_2(10^8)}{10^7}\right) = 500 \cdot log_2(10^8) \approx 1.2 hours$$

This example shows that no matter the technological advancements, designing efficient algorithms is the key to achieving good and satisfactory performance. Additionally, we need mathematical tools to be able to quantify the efficiency of an algorithm, given its code.

1.2 Pseudo-code

The code of an algorithm may be defined in several ways, ranging from lower level assembly language, or even binary code, to high level languages such as C, C++, Java, Python, etc. In order to analyze algorithms it is often convenient to express the code in a form that abstracts from specific details of a programming language and focuses on the core idea of the algorithm. We refer to this way of writing as *Pseudo-code*.

The pseudo-code can include high level abstract language, is unconcerned with implementation details such as variable types and pointers, and does not need definitions or instantiations. However, when studying sorting, we cannot just summarize an algorithm just writing "*sort the sequence*". Conversely, this will be possible when our focus will be on different problems, and sorting will be only a subroutine of our algorithm.

In general, we should find the right level of abstraction to express clearly and without ambiguity the core idea of the algorithm we are designing.

1.3 Introduction to Complexity Analysis

We now start introducing some basic ideas to study algorithm complexity. We will take Insertion sort as an example, provide its pseudo-code, and analyze the complexity in the best and worst case scenario.

1.3.1 Insertion sort

Algorithm idea[1]: Given an array A of length n containing the sequence of numbers to be sorted, at the generic iteration j, the elements in $A[1, \ldots, j-1]$ are already sorted. We want to find the correct position for the element $A[j]$. To this purpose, we use and index i, initially set to $j-1$, and compare $A[i]$ to $A[j]$. We shift the elements to the right and decrease i until we find an element smaller than $A[j]$. The index of i is the correct index of $A[j]$ in the sorted subarray $A[1, \ldots, j]$. Figure 1.1 summarizes the idea.

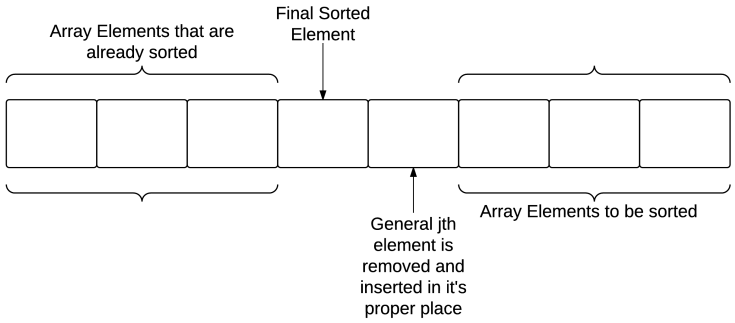


Figure 1.1: Insertion sort idea.

We now provide the pseudo-code of the algorithm.

```
1  InsertionSort(A, n)begin
2      for j=2 to n do
3          key=A[j];
4          i=j-1;
5          while A[i] > key and i > 0 do
6              A[i+1] = A[i];
7              i = i-1;
8          end
9          A[i+1] = key;
10     end
11 end
```

Algorithm 1: Insertion Sort

1.3.2 Example of Execution

In the following, we provide an example of the execution of Insertion sort on the array A: $< 5, 2, 1, 4, 3 >$.

[1]Exams and homework often require you to describe the algorithm idea. This is an example of the required level of detail.

First for iteration: j=2 key=2 i=1
A[i] > key $\rightarrow <5,5,1,4,3>$
i $\not> 0 \rightarrow$ exit while loop
A[i] = key $<2,5,1,4,3>$

Second for iteration: j=3 key=1 i=2
A[i] >key $\rightarrow <2,5,5,4,3>$
i >0 and A[i]> key $\rightarrow <2,2,5,4,3>$
i $\not>$0 exit while$\rightarrow <1,2,5,4,3>$

Third for iteration j=4 key=4 i=3
A[i] > key $\rightarrow <1,2,5,5,3>$
A[i] $\not>$key exit while $\rightarrow <1,2,4,5,3>$

Fourth and final for iteration j=5 key=3 i=4
A[i] > key $\rightarrow <1,2,4,5,5>$
i > 0 and A[i] > key $\rightarrow <1,2,4,4,5>$
i > 0 but A[i] $\not>$key exit while $\rightarrow <1,2,3,4,5>$

1.3.3 Complexity Analysis

We want to understand the growth of the algorithm runtime with respect to the size of the array i.e. the number of items contained in the array to be sorted, n. Note that, other complexity measures may be of interest, for example the *memory complexity*, i.e. how much memory is used as a function of the array size. In this course we will focus only on the execution time, also called runtime, of the algorithms.

Line Number	**Pseudocode**	**Runtime**	**Number of Executions**
	InsertionSort(A, n) {		
1	for j=2 to n {	c_1	n
2	key=A[j]	c_2	n-1
3	i=j-1	c_3	n-1
4	while(i>0 and A[i]>key){	c_4	$\sum_{j=2}^{n} t_j$
5	A[i+1]=A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6	i=i-1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	}		
8	A[i+1]=key	c_7	n-1
9	}		

Table 1.1: Algorithm Complexity Analysis

The total runtime is the sum of the executions of each of the instructions in the code. We assume each line (i) of code has a runtime c_i which is a known constant. We denote by t_j the number of times that the while loop is executed at iteration j of the for loop. Table 1.1 summarizes the complexity of the algorithm instructions.

We can now calculate the expression for the algorithm runtime $T(n)$:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

The best case occurs when the array is already sorted. In this case, the while loop is not executed, therefore $t_j = 1$ for each $j = 1, \ldots, n$. We can simply the expression as follows:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (1 - 1) + c_6 \sum_{j=2}^{n} (1 - 1) + c_7(n-1)$$

$$= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 n + c_6 n + c_7(n-1)$$

$$= an + b$$

where a and b are appropriate positive constants. We call this type of complexity, *linear time* since the runtime $T(n)$ grows linearly with the input size n.

The worst case occurs instead when the array is sorted in reverse order. This implies that $t_j = j$, and leading to the following expression for $T(n)$:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j - 1) + c_6 \sum_{j=2}^{n} (j - 1) + c_7(n-1)$$

$$= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left(\frac{n(n-1)}{2} - 1 \right) + c_5 \left(\frac{n(n-1)}{2} \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7(n-1)$$

$$= an^2 + bn + c$$

where a, b and c are appropriate positive constants.

We should now ask ourselves if we really need this level of precision in analyzing the complexity, i.e. if we really need to specify for each line a different constant. In any case, these constants would depend on the specific implementation and computer on which the algorithm is executed. Additionally, at the end we summarized these constants in other constants a, b and c.

This should give us the idea that what really matters is the *order of growth* of the runtime function, rather than the individual constants. This intuition will lead us to study the *asymptotic complexity*.

1.4 Asymptotic Complexity Analysis

We want to define the mathematical foundation to compare the performance (runtime) of different algorithms, neglecting implementation details. To this purpose, we look at input sizes which are *sufficiently large* to ensure that the constants become insignificant and the runtime depends only on the leading term of the complexity expression, i.e. the term with the highest power of n. Considering large inputs gives this analysis its name, i.e. the asymptotic analysis. We now introduce the basic notations of asymptotic analysis.

1.4.1 Big O Notation

The big O notation provides an asymptotic upper bound of a function.

Definition 1.4.1. *Given a function $g(n)$, and $n \in \mathbb{N}$ we say that $O(g(n)) = \{f(n)$ s.t. $\exists c > 0$ and $n_0 > 0$ s.t. $0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\ \}$. Therefore $O(g(n))$ is a set of functions.*

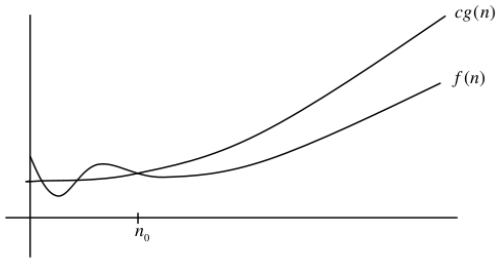


Figure 1.2: Big O relationship graph

We generally use the definition to show that a function $f(n)$ is a "big O" of another function (usually with a simpler form) $g(n)$. To this purpose, given $f(n)$ and $g(n)$ we look for the constants c and n_0 that satisfy the definition. If we are able to find such constants, then we say that $f(n)$ "belongs" or "is" a big O of $g(n)$. We should write, to be mathematically correct, $f(n) \in O(g(n))$, this is often written as $f(n) = O(g(n))$.

A visualization is provided in Figure 1.2. As the figure shows, if the input is sufficiently large, $g(n)$, multiplied by a constant, is always bigger than $f(n)$, therefore $f(n)$ is upper-bounded by $g(n)$. In terms of runtime, this means that if an algorithm has complexity $f(n)$, it cannot do worse than $g(n)$ when the input is sufficiently large.

Example 1: Consider $f(n) = 3n + 3$, prove that $f(n) = O(n)$
We need to find values for c and n_0 that satisfy the necessary conditions.

$$3n + 3 \leq cn \quad \forall n \geq n_0$$

divide by n

$$3 + \frac{3}{n} \leq c$$

if $n > 3$ then $\frac{3}{n} \ll 1$, therefore we can pick $c = 4$ (do not make it more complicated than necessary).
We now solve the remaining inequality for n

$$3n + 3 \leq 4n$$

$$3 \leq n$$

We find that $n_0 = 3$ and $c = 4$ satisfies the conditions sought to be proven. Therefore, $f(n) = O(n)$.

Example 2: Given $f(n) = 3n + 3$, prove that $f(n) = O(n^2)$

$$3n + 3 \leq cn^2$$

$$\frac{3}{n} + \frac{3}{n^2} \leq c \rightarrow c = 2$$

$$3n + 3 \leq 2n^2$$

$$2n^2 - 3n - 3 \geq 0$$

$$n_0 = \left\lceil \frac{3 \pm \sqrt{9 - 4 \cdot 2 \cdot -3}}{4} \right\rceil$$

Choose maximum (ceiling) value, so $n_0 = 5$. This example shows that although $f(n)$ is a linear function, it is upperbounded by a quadratic function. This is mathematically correct, but we say that the bound is not tight. We generally look for tight bounds that better characterize the complexity.

1.4.2 Big Ω Notation

The big Ω notation identifies an asymptotic lower bound.

Definition 1.4.2. *Given a function $g(n)$, and $n \in \mathbb{N}$ it is said that $\Omega(g(n)) = \{f(n)$ s.t. $\exists$ $c > 0$ and $n_0 > 0$ s.t. $0 \leq cg(n) \leq f(n)$ $\forall n \geq n_0\}$*

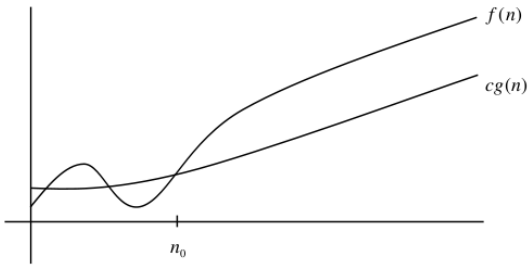


Figure 1.3: Big Ω relationship graph

If an algorithm has complexity $f(n)$ and we can prove that $f(n) = \Omega(g(n))$, then this implies that when the input is sufficiently large, the algorithm runtime cannot be better than $g(n)$. Figure 1.3 show a representation of the big Ω notation.

Example 3: Given $f(n) = 2n^2 + 3$, prove that $f(n) = \Omega(n)$.

$$2n^2 + 3 \geq cn \rightarrow c = 1$$

$$2n^2 + 3 \geq n \text{ which is true } \forall n \geq 0$$

Example 4: Given $f(n) = 2n^2 + 3$, prove that $f(n) = \Omega(n^2)$

$$2n^2 + 3 \geq cn^2 \rightarrow c = 1$$

$$2n^2 + 3 \geq n^2 \text{ which is true } \forall n \geq 0$$

1.4.3 Big Θ Notation

The big Θ notation identifies an asymptotic tight bound. Intuitively, $f(n)$ and $g(n)$ have the same asymptotic growth.

Definition 1.4.3. *Given a function $g(n)$, and $n \in \mathbb{N}$ it is said that $\Theta(g(n)) = \{f(n)$ s.t. $\exists c_1, c_2, n_0 > 0$ s.t. $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$*

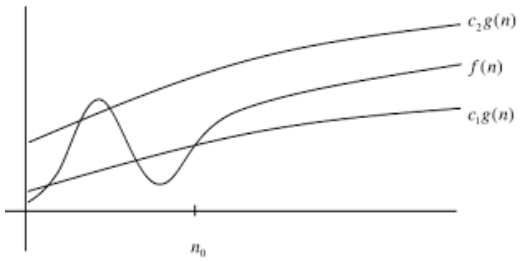


Figure 1.4: Big Θ relationship graph

Example 5: Given $f(n) = \frac{n^2}{2} - 3n$, prove that $f(n) = \Theta(n^2)$

$$c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

$$\forall n \geq 12 \rightarrow c_1 = \frac{1}{4}, c_2 = 1$$

$$\frac{1}{4} n^2 \leq \frac{n^2}{2} - 3n \leq n^2$$

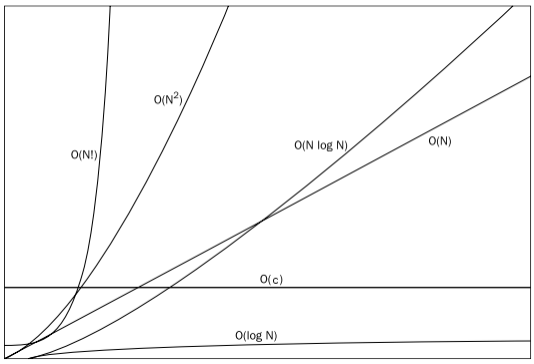


Figure 1.5: Common complexities graphed concurrently

The inequality above is true $\forall n \geq 12$, therefore we can pick $n_0 = 12, c_1 = \frac{1}{4}, c_2 = 1$

Figure 1.5 summarizes some common complexity functions. The following theorem, of which we omit the proof, describes the relation between the O, Ω and Θ.

Theorem 1. *Given $f(n)$ and $g(n)$, $f(n) = \Theta(g(n)$ if and only if $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$.*

1.4.4 Algebra of Asymptotic Notation

We now introduce some rules, the algebra of asymptotic notation, that enable the determination of the asymptotic complexity much quicker than the examples in the previous section.

Constants: $\forall$ k $\in \mathbb{R}^+$, if $f(n) = O(g(n))$ then $kf(n) = O(g(n))$. Similar rule applies to Ω and Θ.

Sum: if $f(n) = O(g(n))$ AND $d(n) = O(h(n))$, then $O(f(n) + d(n)) = O(g(n) + h(n)) = O(max[g(n), h(n)])$. Similar rule applies to Ω and Θ.

Multiplication: if $f(n) = O(g(n))$ AND $d(n) = O(h(n))$, then $f(n) \cdot d(n) = O(g(n)) \cdot h(n))$. Similar rule applies to Ω and Θ.

Transitivity: if $f(n) = O(g(n))$ AND $g(n) = O(h(n))$, then $f(n) = O(h(n))$. Similar rule applies to Ω and Θ.

Reflexivity: $f(n) = O(f(n))$. Similar rule applies to Ω and Θ.
Simmetry: $f(n) = O(f(n))$. Similar rule applies to Ω and Θ.

- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
- $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$

1.5 The Search Problem: Binary search

We now focus on the search problem, which can be defined as follows. Given an ordered sequence $< a_1, a_2, ...a_n >$ and a value x, determine if x is in the sequence or not. A trivial solution for this problem is to iterate through the whole sequence. This would clearly incur a cost of $O(n)$. We consider a smarter approach, called *binary search*, which exploits the fact that numbers are sorted.

Algorithm idea: Binary search is a recursive approach. Recursion is performed on the length of the array. In the base case the array is of length 1. If the single element is equal to x we return True, otherwise False. In the recursive step, we look at the element y in the middle of the current subarray, if $y < x$ we recursively search on the left half of the array, otherwise on the right half.

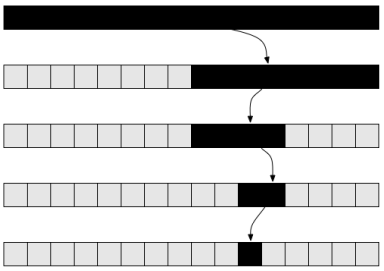


Figure 1.6: Binary Search Execution

```
1  BinarySearch(A, start, end)begin
2  |  //(A, 1, n) for first call
3  |  if start==end then
4  |  |  if A[start]==x then
5  |  |  |  return True
6  |  |  end
7  |  |  return False
8  |  end
9  |  m=⌈(start+end)/2⌉
10 |  if A[m] > x then
11 |  |  return BinarySearch(A, start, m-1)
12 |  end
13 |  return BinarySearch(A, m, end)
14 end
```

Algorithm 2: Binary Search

1.5.1 Example

Given the array $< 1, 3, 7, 9, 11 >$ we execute binary search for the value $x = 9$.

First call: start=1 end =5
start $\neq$ end $\rightarrow$ m=$\lceil \frac{5+1}{2} \rceil = 3 \rightarrow$ A[3]=7$\not>$x
recursive call BinarySearch(A, m, end)

Second call: start=3 end=5
start $\neq$ end $\rightarrow$ m=$\lceil \frac{5+3}{2} \rceil = 4 \rightarrow$ A[4]=9$\not>$x
recursive call BinarySearch(A, m, end)

Third call: start=4 end=5
start $\neq$ end $\rightarrow$ m=$\lceil \frac{5+4}{2} \rceil = 5 \rightarrow$ A[5]=11>x
recursive call BinarySearch(A, start, m-1)

Fourth call: start=4 end=4
start = end $\rightarrow$ A[4]=9=x
return True

1.6 Recurrence Equations

We use the binary search algorithm in the previous section to describe a general methodology to analyze the complexity of recursive algorithms. This methodology is based on recurrence equations. The goal is to calculate the complexity $T(n)$ of the algorithm, which is initially defined recursively. In the case of the binary search it is defined as follows.

$$T(n) = \begin{cases} \Theta(1) & n = 1 \text{ base case} \\ T(\frac{n}{2}) + \Theta(1) & n > 1 \text{ recursive step} \end{cases}$$

The base case occurs when the array has length 1, and in this case the complexity is constant, therefore $\Theta(1)$. The recursive case occurs when $n > 1$. The complexity of a recursive step has two components, the cost of the current iteration, which in the case of binary search is constant since there are no loops, and the cost of the other recursive call(s). In this case, we recursively call the algorithm on an input which is half of the previous one, so the recursive cost is $T(\frac{n}{2})$.

There are several methods to solve recurrence equations such as the one above, for example substitution method, iteration method, recursive tree method, and master theorem. In this course we focus on the last two approaches.

1.6.1 Recursive Tree Method

The recursive tree method is based on building a tree, with the following meaning.

- Nodes represent the cost of a single sub-problem at that level of recursion
- Each level of the tree is a level of recursion

- Intermediate levels (between the root and leaves) are intermediate recursive calls
- Leaves are the base cases
- We use the constant c to indicate a constant cost
- For each level of the tree we keep track of the input size, and the overall cost (sum of all nodes at that level)

Let's now see an example of the recursive tree method applied to the recurrence equation for binary search.

Tree	Level	Input Size	Cost
c	0	n	c
c	1	$\frac{n}{2}$	c
c	2	$\frac{n}{4}$	c
..	..	..	..
c	i	$\frac{n}{2^i}$	c
..	..	..	..
c	k (leaf or base case)	$\frac{n}{2^k} = 1$	c

Figure 1.7: Recursion Tree table basic example

We start with the root node, which is at level 0, has input size n, and since the cost of each recursive call of binary search is constant, we have a cost of c. We then write at least a couple of additional levels to understand how things change as we go deeper in the tree. We can then generally see a pattern, and we are able to write the generic level i. In this case in the generic level i we have an input size of $\frac{n}{2^i}$, this is due to the fact that we always divide the input in a half with binary search.

We stop when we reach the base case, at level k. This occurs when the input cannot be further divided, i.e. we are considering a subarray of length 1. In other words, when $\frac{n}{2^k} = 1$. Solving for k:

$$1 = \frac{n}{2^k}$$

$$2^k = n$$

$$k = log(n)$$

Note that when we perform asymptotic analysis, we usually omit the base of the logarithm. This is due to the logarithm property for which changing the base translates in multiplying by a constant, and, as we know here, constants do not matter.

From the above equations we know that binary search performs a logarithmic number of steps. To calculate the overall cost we sum the total cost of all levels. We split this in two parts, the sum of the costs from 0 to $k-1$ and the sum of the costs of the leaves. In this

case the costs from 0 to $k-1$ is $\sum_{i=0}^{k-1} c$ and there is only one leaf therefore the cost is c. In summary:

$$\sum_{i=0}^{k-1} c + c = \sum_{i=0}^{log(n)-1} c + c = c \cdot log(n) + c = \Theta(log(n))$$

We can hence conclude that binary search has a complexity which is logarithmic in the length of the input, which is better than the trivial linear approach.

1.6.2 A More Complex Example

Let's consider a more complex example given by the following recurrence equation.

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n^2) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

Let's first make clear what this means. An algorithm having this recurrence equation has a constant base case when $n = 1$. The recurrence step has two recursive calls, each of which works on an input half the size of the previous input, therefore the term $2T(\frac{n}{2})$. Additionally, the cost of each recursive call is quadratic with the size of the input, $\Theta(n^2)$. Be sure to have understood where each part of the equation comes from, which is usually a source of confusion.

Tree	Level	Input Size	Cost
cn^2	0	n	cn^2
$c(\frac{n}{2})^2 \quad c(\frac{n}{2})^2$	1	$\frac{n}{2}$	$c\frac{n^2}{2}$
$c(\frac{n}{4})^2 \quad c(\frac{n}{4})^2 \quad c(\frac{n}{4})^2 \quad c(\frac{n}{4})^2$	2	$\frac{n}{4}$	$c\frac{n^2}{4}$
...	..	...	..
$c(\frac{n}{2^i})^2 \quad ... \quad c(\frac{n}{2^i})^2$	i	$\frac{n}{2^i}$	$2^i \frac{n^2}{2^{2i}} = c\frac{n^2}{2^i}$
...	..	...	..
$\Theta(1) \quad ... \quad \Theta(1)$	k (leaf or base case)	$\frac{n}{2^k} = 1 \rightarrow k = \log(n)$	$2^k\Theta(1)$

Table 1.2: Recursion Tree table example

We can now sum the cost of all levels. The cost of the levels from 0 to $k-1$ is $\sum_{i=0}^{log(n)-1} c\frac{n^2}{2^i}$. The cost of the leaves is $2^k\Theta(1)$, but since $k = \log(n)$ we get $2^{\log n}\Theta(1) = n\Theta(1) = nc$. Therefore:

$$\sum_{i=0}^{log(n)-1} c\frac{n^2}{2^i} + cn = cn^2 \sum_{i=0}^{log(n)-1} (\frac{1}{2})^i + cn$$

We can now use the following summation expression property for geometric series. This formula is used often in solving recurrence equations, so be sure to remember it. $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad , if x < 1$. In in our case:

$$cn^2 \sum_{i=0}^{log(n)-1} (\frac{1}{2})^i + cn = 2cn^2 + cn = \Theta(n^2)$$

1.6.3 Another Example

$$T(n) = \begin{cases} 16T(\frac{n}{4}) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

Let's again be clear about the meaning of this equation. The algorithm has 16 recursive calls, each of which works on an input which is one fourth of the previous call. The cost of each recursive call is linear.

Tree	Level	Input Size	Cost
cn	0	n	cn
$\overbrace{\frac{cn}{4} \cdots \frac{cn}{4}}^{16}$	1	$\frac{n}{4}$	$c16\frac{n}{4} = 4cn$
$\overbrace{\frac{cn}{16} \cdots \frac{cn}{16}}^{16^2}$	2	$\frac{n}{16}$	$c16^2\frac{n}{16} = 16cn$
...	...	...	...
$\overbrace{\frac{cn}{4^i} \cdots \frac{cn}{4^i}}^{16^i}$	i	$\frac{n}{2^i}$	$c16^i\frac{n}{4^i} = 4^icn$
...	...	...	...
$\overbrace{\Theta(1) \cdots \Theta(1)}^{16^k}$	k	$1 = \frac{n}{4^k} \quad k = log_4(n)$	$c16^k = c16^{log_4(n)} = cn^{log_4(16)} = cn^2$

Table 1.3: Complexity Analysis: Recursion Tree Method

We can now sum all levels between 0 and $k-1$, plus the cost of the leaves.

$$\sum_{i=0}^{log(n)-1} 4^icn + c16^k = cn \sum_{i=0}^{log(n)-1} 4^i + cn^2$$

To solve summation recall:

$$\sum_{i=0}^{k} z^i = \frac{z^{k+1}-1}{z-1}, \text{ apply here:}$$

$$cn \sum_{i=0}^{log(n)-1} 4^i = cn\frac{4^{log_4(n)-1+1}-1}{4-1} = cn4^{log_4(n)} - cn = cnn^{log_4(4)} - cn = cn^2 - cn$$

Above we absorbed the denominator $(4-1)$ into the constant c (this is the magic of asymptotic analysis). Concluding, we can rewrite the complexity as:

$$cn^2 - cn + cn^2 = \Theta(n^2)$$

1.6.4 The Master Theorem

The master theorem provides a simple way to solve most recurrence equations.

Theorem 2. *Let $a \geq 1$ and $b \geq 1$ be constants, $f(n)$ be a function, and $T(n) = aT(\frac{n}{b}) + f(n)$ then:*

1. *if $f(n) = O(n^{log_b(a)-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{log_b(a)})$*
2. *if $f(n) = \Theta(n^{log_b(a)})$, then $T(n) = \Theta(n^{log_b(a)} log(n))$*
3. *if $f(n) = \Omega(n^{log_b(a)+\varepsilon})$ for some $\varepsilon > 0$, and if $af(\frac{n}{b}) \leq cf(n)$ for $c < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$*

Example case 1

$$T(n) = 9T(\frac{n}{3}) + n \rightarrow \quad a = 9 \quad b = 3 \quad f(n) = n$$

$$n^{log_b(a)} = n^{log_3(9)} = n^2 \rightarrow f(n) = O(n^{2-\varepsilon}) \quad \forall \quad 0 < \varepsilon \leq 1$$

Therefore this example falls into case 1 and

$$T(n) = \Theta(n^{log_b(a)}) = \Theta(n^{log_3(9)}) = \Theta(n^2)$$

Example case 2

$$T(n) = 9T(\frac{2n}{3}) + 1 \rightarrow \quad a = 1 \quad b = \frac{3}{2} \quad f(n) = 1$$

$$n^{log_b(a)} = n^{log_{\frac{3}{2}}(1)} = n^0 = 1$$

$$f(n) = \Theta(n^0) = \Theta(1)$$

Therefore this example falls into case 2 and

$$T(n) = \Theta(n^0 log(n)) = \Theta(log(n))$$

Example case 3

$$T(n) = 3T(\frac{n}{4}) + nlog(n) \rightarrow \quad a = 3 \quad b = 4 \quad f(n) = nlog(n)$$

$$n^{log_b(a)} = n^{log_4(3)} = n^{0.7925} \rightarrow f(n) = \Omega(n^{0.7925+\varepsilon}) \quad for \quad \varepsilon \approx 0.2$$

In order to apply case three, we also need to prove the following that $af(\frac{n}{b}) \leq cf(n)$.

$$af(\frac{n}{b}) = 3f(\frac{n}{4}) = 3\frac{n}{4}log(\frac{n}{4}) = \frac{3}{4}n \log n - \frac{3}{4}n \log 4 \leq \frac{3}{4}n \log n$$

We can select $c = \frac{3}{4}$, and satisfy the condition for case 3. Therefore,

$$T(n) = \Theta(nlog(n))$$

1.7 Exercises

1. Let, $P(n) = \sum_{i=0}^{d} a_i n^i$, where, $a_d > 0$, be a degree-d polynomial in n,and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties:

If $k \geq d$, then $p(n) = O(n^k)$

2. Indicate, for each pair of expressions (A, B) in the table below, whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\varepsilon > 0$, and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

A	B	O	o	Ω	ω	Θ
$lg^k n$	n^{ε}					
n^k	c^n					
$\sqrt{n}$	$n^{\sin n}$					
2^n	$2^{n/2}$					
$n^{\lg c}$	$c^{\lg n}$					
$\lg(n!)$	$\lg(n^n)$					

3. Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

a. $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

b. $f(n) + g(n))$ implies $\Theta(min(g(n), f(n)))$.

c. $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.

d. $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.

e. $f(n) = O((f(n))^2)$.

f. $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

4. Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 2T(n/2) + n^4$

b. $T(n) = T(7n/10) + n$

c. $T(n) = 16T(n/4) + n^2$

d. $T(n) = 7T(n/3) + n^2$

e. $T(n) = 7T(n/2) + n^2$

f. $T(n) = 2T(n/4) + \sqrt{n}$

g. $T(n) = T(n-3) + n^2$

(**Hint:** Applying the Master Theorem whenever possible would be less painful most of the time when you are asked to "justify your answers" make sure you justified which case are you referring to.)—

5. Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for sufficiently small n. Make your bounds as tight as possible, and justify your answers.
a. $T(n) = 2T(n/2) + n/\lg n$
b. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

6. Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \ldots, g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$lg(lg^* n)$	$2^{lg^* n}$	$\sqrt{2}^{lg\ n}$	n^2	$n!$	$(lgn)!$
$(\frac{3}{2})^n$	n^3	$lg^2\ n$	(lgn)!	2^{2^n}	$n^{\frac{1}{lgn}}$
$ln\ ln\ n$	$lg^* n$	$n.2^n$	$n^{lg\ lg\ n}$	$ln\ n$	1
2^{lgn}	$(lgn)^{lgn}$	e^n	$4^{lg\ n}$	$(n+1)!$	$\sqrt{lg\ n}$
$lg^*(lg\ n)$	$2^{\sqrt{2lg\ n}}$	n	2^n	$n\ lg\ n$	$2^{2^{n+1}}$

Chapter 2

Divide and Conquer

The Roman Emperor Julius Cesar, and probably Philip II of Macedon before him, involuntarily defined a concept that became today a cornerstone of algorithm design. As the old Romans realized, when you are facing a big problem, it is more effective to decompose it in a set of smaller problems, the solution of which will easily lead to the solution of the main problem. This is known as the *Divide et Impera* approach, in Latin, or *Divide and Conquer*, in English.

With less belligerent intentions than the Roman Emperor, we can identify the main components of this approach to solve a given problem (e.g. sort an array of numbers).

1. **Divide**: Identify a number of sub-problems that are smaller instances of the same problem.
2. **Conquer**: If the size of the problem is small enough, then solve it in a straightforward way (base case), otherwise solve the sub-problems recursively (recursive case).
3. **Combine**: Put together the solutions of the sub-problems in a way that yields the solution to the main problem.

In this chapter we discuss some examples of this technique applied to different problems.

2.1 Merge Sort

Merge sort is a recursive sorting algorithm designed according to the Divide and Conquer approach. Different from most iterative approaches, (e.g. Insertion sort, Bubble sort, Selection sort) which have a quadratic complexity $O(n^2)$, Merge sort complexity is $\Theta(n \log(n))$.

The previous high level scheme of Divide and conquer is applied as follows. Here we are sorting an array of numbers of length n.

1. **Divide**: Break up the sequence of n elements into two smaller sub-sequences, each of length $\frac{n}{2}$.

2. **Conquer**: If sub-sequences have length one, then they are already sorted (base case), otherwise sort them recursively.

3. **Combine**: Combine the sorted sub-sequences to obtain the original array sorted.

2.1.1 Pseudo-code

The algorithm is composed by two main functions. The first function `MergeSort(A, start, end)` takes as input the array A to be sorted, and the index of the first and last element in A. When the function is called the first time, start is set to 1 and end to 0. However, these values will change during the subsequent recursive calls. In general, start and end identify the beginning and the end of the sub-array considered by the current recursive call.

The base case is implicit, and occurs when `start == end`, which implies that A has length 1. In the recursive case, the function calculates the mid-point m of the array, recursively calls itself on the left sub-array `MergeSort(A, start, m)` and on the right sub-array `MergeSort(A, m+1, end)`, and finally combines the two sorted sub-arrays calling the function `Merge(A, start, m, end)`.

```
1 MergeSort(A, start, end)begin
2 |  if start<end then
3 |  |  m=⌊(start+end)/2⌋
4 |  |  MergeSort(A, start, m)
5 |  |  MergeSort(A, m+1, end)
6 |  |  Merge(A, start, m, end)
7 |  end
8 end
```

Algorithm 3: Merge Sort

The function merge takes as input the array A, the current start and end indices, and the mid-point index m. It knows that `A[start,..., m]` and `A[m+1,..., end]` are sorted, and combines them. This is relatively a trivial task, the only challenge is to be sure that the resulting complexity is linear, i.e. $\Theta(n)$. This will be crucial for the complexity of the overall algorithm.

In order to ensure the linear complexity, the function Merge makes use of an additional array B, of length end-start+1, i.e. the same length of the combined sub-arrays we are considering. We use an index i to iterate on the first sub-array, and j to iterate on the second, while we use k to iterate on B.

The function iteratively compares the values in $A[i]$ and $A[j]$, and put the smaller one in $B[k]$. When i reaches the end of the first sub-array, or j reaches the end of the second, remaining elements, if present, are copied in B. Finally, B is copied in A. Note that this is not the smartest implementation of the function, for example we can easily get rid of B, saving memory. I prefer this implementation as it is very clear and simple to understand, while providing the same asymptotic complexity.

```
1  Merge(A, start, m, end)begin
2      //Linear Compleity i.e. Θ(n)
3      i=start
4      j=m+1
5      k=1
6      create an array B of length end-start+1
7      while i≤m and j≤end do
8          if A[i]<A[j] then
9              B[k]=A[i]
10             k++
11             i++
12         end
13         else
14             B[k]=A[j]
15             k++
16             j++
17         end
18     end
19     while i≤m do
20         B[k]=A[i]
21         i++
22         k++
23     end
24     while j≤end do
25         B[k]=A[j]
26         j++
27         k++
28     end
29     Copy B in A
30 end
```

Algorithm 4: Merge

An example of the execution of merge sort is in Figure 2.1.

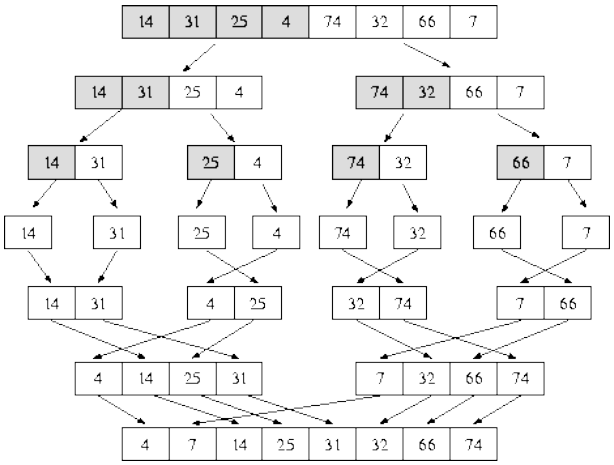


Figure 2.1: Merge Sort Execution

2.1.2 Complexity

To analyze the complexity of Merge sort, let's first look at the function `MergeSort()`. It splits the input in two parts, each of which of size $n/2$, and performs a recursive call on each of these. Therefore the recurrence equation $T(n)$ will have a component $2T(\frac{n}{2})$. Let's now focus on the function `Merge()`. Lines 1-3 are constant, $\Theta(1)$. The while loop on line 7-18 is executed at most n times, since i can go from start to m, and j from $m+1$ to end. The remaining whiles may also do at most a linear number of iterations. Therefore, the overall complexity of `Merge()` is $\Theta(n)$. This meas that each recursive call has a linear complexity. Given that the base case occurs when $n = 1$, we can write the recurrence equation as follows.

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

Recursion Tree Method:

Tree	Level	Input Size	Cost
cn	0	n	cn
$\frac{cn}{2}$ $\frac{cn}{2}$	1	$\frac{n}{2}$	cn
$\frac{cn}{4}$ $\frac{cn}{4}$ $\frac{cn}{4}$ $\frac{cn}{4}$	2	$\frac{n}{4}$	cn
...	...	...	...
$\frac{cn}{2^i}$ $\cdots$ $\cdots$ $\frac{cn}{2^i}$	i	$\frac{n}{2^i}$	cn
...	...	...	...
$\Theta(1)$ $\cdots$ $\cdots$ $\Theta(1)$	k	$1 = \frac{n}{2^k}$ $k = log_2(n)$	2^k

Table 2.1: Merge Sort Complexity Analysis: Recursion Tree Method

$$\sum_{i=0}^{log(n)-1} cn + c2^k = cn \sum_{i=0}^{log(n)-1} 1 + c2^{log_2(n)} = cnlog(n) + cn^{log_2(2)} = \Theta(nlog(n))$$

Master Theorem:

$$a = 2 \quad b = 2 \quad f(n) = \Theta(n)$$
$$n^{log_b(a)} = n^{log_2(2)} = n^1 = n$$

Since $\Theta(n^{log_2(2)}) = \Theta(n) = f(n)$, we are on Case II, therefore $T(n) = \Theta(nlog(n))$.

2.2 Maximum Sub-array Problem

The maximum sub-array problem can be defined as follows. Given an array of size n containing positive and negative numbers, find the sub-array with the maximum sum of elements. A sub-array is a set of consecutive positions of the original array, therefore we are looking for the indices that identify the beginning and the end of the sub-array with maximum sum of elements. Figure 2.2 shows an example.

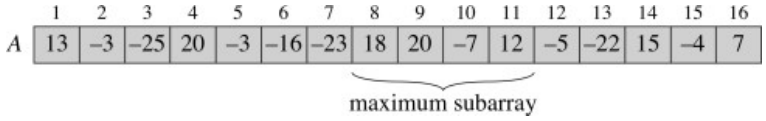


Figure 2.2: Example of maximum sub-array

Several possible solutions may easily come to mind for this problem, which can however incur a very different complexity. A straightforward solution is to consider each element of the array, and all possible sub-arrays starting at that element. For each sub-array we calculate a new sum of elements. This solution is very inefficient, and results in a complexity of $\Theta(n^3)$. This could be improved to $\Theta(n^2)$ if we take into account the fact that the sum

could be updated, instead of calculated from scratch for every sub-array starting at a given element.

A more efficient solution is based on the Divide and Conquer approach, and allows to solve the problem in $\Theta(nlog(n))$. In Chapter 4 we will see a solution based on Dynamic programming that allows to solve the problem in linear time.

2.2.1 Divide and Conquer Solution

The main idea of the Divide and Conquer approach is the following. Consider Figure 2.3. Let `start` and `end` be the first and last index of the array, and m be the mid-point. Additionally, let i and j be the beginning and the end of the sub-array with max sum that we are looking for. We can have three cases:

1. The max sub-array completely lies on the left side, hence both i and j are on the left, $start \leq i \leq j \leq m \leq end$
2. The max sub-array completely lies on the right side, hence both i and j are on the right, $start \leq m+1 \leq i \leq j \leq end$
3. The max sub-array crosses m, hence i is on the left of m and j is on the right, $start \leq i \leq m < j \leq end$

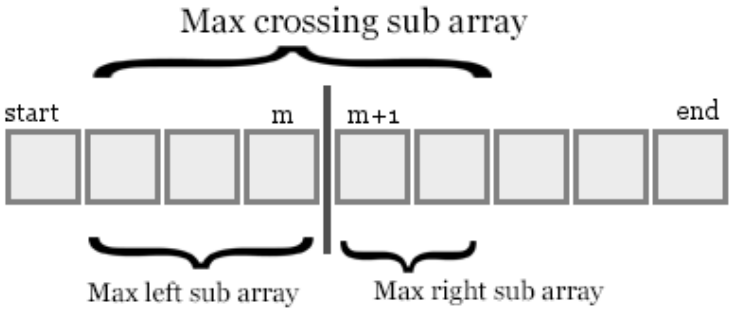


Figure 2.3: Maximum Sub-array Algorithm Idea

Exploiting these cases, we first solve the problem for the case in which the sub-array crosses m with the function `FindMaxCrossingSubarray()`. The main idea here is that if it crosses m, then both the elements in $A[m]$ and $A[m+1]$ belong to the sub-array (otherwise it would not cross m and we would be in one of the other cases). We exploit this fact to look for the max sub-array on the left of m ending at $A[m]$, and for the max sub-array on the right of m starting at $A[m+1]$. The concatenation of these two sub-arrays gives the maximum sub array that crosses m.

The function `FindMaxCrossingSubarray()` first looks for the maximum sub-array ending at $A[m]$ (lines 2-11). It keeps track of the maximum sum of elements encountered L_{sum} and

```
1  FindMaxCrossingSubarray(A, start, m, end) begin
2  |  L_sum = −∞ //sum of max left sub-array ending at A[m]
3  |  max_l //index of the first element of the max left sub-array
4  |  sum=0
5  |  for  i=m down to start do
6  |  |  sum = sum+A[i]
7  |  |  if L_sum < sum then
8  |  |  |  L_sum=sum
9  |  |  |  max_l=i
10 |  |  end
11 |  end
12 |  R_sum = −∞
13 |  max_r
14 |  sum=0
15 |  for  j=m+1 end do
16 |  |  sum = sum+A[j]
17 |  |  if R_sum < sum then
18 |  |  |  R_sum=sum
19 |  |  |  max_r=i
20 |  |  end
21 |  end
22 |  return (max_l, max_r, L_sum + R_sum)
23 end
```

Algorithm 5: FindMaxCrossingSubarray

the index max_l of the first element of the sub-array that generated such maximum sum. Subsequently, it looks for the maximum sub-array starting at $A[m+1]$, and similarly it keeps track of the maximum sum R_{sum} and the index max_r. The algorithm outputs the starting index max_l and ending index max_r of the maximum sub-array that crosses m, having sum of elements $L_{sum} + R_{sum}$.

```
1  FindMaxSubarray(A, start, end){
2  if start==end then
3  |  return (start, end, A[start])
4  end
5  m=⌊(start+end)/2⌋
6  (L_start, L_end, L_sum)=FindMaxSubarray(A, start, m)
7  (R_start, R_end, R_sum)=FindMaxSubarray(A, m+1, end)
8  (C_start, C_end, C_sum) = FindMaxCrossingSubarray(A, start, m, end)
9  if L_sum ≥ R_sum and L_sum ≥ C_sum then
10 |  return(L_start, L_end, L_sum
11 end
12 if R_sum ≥ L_sum and R_sum ≥ C_sum then
13 |  return(R_start, R_end, R_sum
14 end
15 return(C_start, C_end, C_sum)
16 }
```

Algorithm 6: FindMaxSubarray

Given the above procedure, we can now introduce the main function `FindMaxSubarray()` that solves the problem recursively. The base case occurs when the array has a single element (`start == end`). In this case the maximum sub-array is the element itself. Otherwise, we calculate the mid-point m and find recursively the maximum sub-array that lies on the left of m, on the right of m, and use the function `FindMaxCrossingSubarray()` to find the maximum sub array crossing m. The sub-array which has maximum sum between these three is the sub-array we are looking for.

2.2.2 Complexity

Let's start from the function `FindMaxCrossingSubarray()`. This function iterates over the left half of the array, and then on the right half. There are no nested loops, thus the complexity is linear $\Theta(n)$. The main algorithm function `FindMaxSubarray()` has two recursive calls, each of which operates on an input which is half of the current input. Additionally, each call invokes the function `FindMaxCrossingSubarray()` which has complexity $\Theta(n)$. Therefore we can write the following recursive equation, and solve it with the Master Theorem.

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

$$a = 2 \quad b = 2 \quad f(n) = n$$
$$n^{log_2(2)} = n \quad f(n) = n = \Theta(n)$$

We are therefore in case II and we can conclude that $T(n) = \Theta(nlog(n))$.

2.3 QuickSort

Quicksort applies the divide and conquer approach to the sorting problem. It has a worst case complexity of $O(n^2)$, but it can be shown that on average the complexity is $O(n \log(n))$. Additionally, the hidden constants, i.e. the constants that disappear in the asymptotic analysis, are small, making Quicksort one of the best sorting algorithms in practice.

Algorithm idea. Quick sort is a recursive algorithm, at each recursive call it selects a pivot value x and places it the correct position, i.e. in the position that x should have in the sorted array. It then recursively sorts the sub-arrays at the left and right side of x.

The algorithm applies the divide and conquer approach as follows.

- **Divide:** given the array A[start..end] choose a *pivot value* x, find the correct position p for x such that A[p]=x and both of the following statements hold:
  - $\forall i \in [start...p-1], \quad A[i] \leq x$
  - $\forall i \in [p+1...end] \quad A[i] > x$
- **Conquer:** Recursively sort the sub-arrays A[start...p-1] and A[p+1...end]
- **Combine:** Nothing left to do. The array is already sorted.

2.3.1 Pseudo-code

```
1 QuickSort(A, start, end)begin
2 |  if start<end then
3 |  |  p=Partition(A, start, end)
4 |  |  QuickSort(A, start, p-1)
5 |  |  QuickSort(A, p+1, end)
6 |  end
7 end
```

Algorithm 7: QuickSort

The algorithm has two functions. The main function `QuickSort()` takes as input the array A, and the start and end indices. It invokes the function `Partition()` that returns the position p of the pivot. The function `QuickSort()` then recursively sorts the sub-arrays on the left and right side of p.

```
1  Partition(A, start, end)begin
2  |  x=A[end]
3  |  i=start-1
4  |  for j=start to end do
5  |  |  if A[j]≤x then
6  |  |  |  i++
7  |  |  |  swap(A[i], A[j])
8  |  |  end
9  |  end
10 |  swap(A[i+1], A[end])
11 |  return i+1
12 end
```

Algorithm 8: Partition

The function `Partition` picks the last element of the current array, i.e. A[end], as pivot value x. It then uses an index j to iterate over the array, and an index i. The idea is to separate the array so that, at the j-th iteration all elements in $A[start, \ldots, i]$ are less than or equal to x, all elements in $A[i+1, \ldots, j]$ are greater than or equal to x, while the elements in $A[j, \ldots, end]$ have not been analyzed yet. In order to keep this property, initially j is set to start and i to $start - 1$. As j increases and an element is found which is smaller than the pivot value x, i is increased, and $A[i]$ (which is greater than x) and $A[j]$ are swapped, restoring the desired property. When j reaches end, the function performs a last swap to put x in its correct position, which is $i + 1$. This location is also returned as pivot to the main function.

2.3.2 Complexity

The complexity of QuickSort depends on the choice of the pivot value, and how this divides the array in the subsequent recursive calls. In general, we can say that the array is split in two parts, one of length k and the other one of length $n - k - 1$. The complexity of each recursive call is given by the complexity of the function `Partition()`. This function has a single for loop, which iterates on the entire array, therefore its complexity is $\Theta(n)$. We can write the following general recurrence equation.

$$T(n) = \begin{cases} T(k) + T(n-k-1) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

We can identify the worst and best case for this algorithm. The worst case occurs when the pivot value chosen at each iteration is either the largest or the smallest element of the array. As a result, $k = 0$, or $k = n - 1$, and one recursive call will work on an empty array, but the other one on $n - 1$ elements. The resulting recurrence equation is:

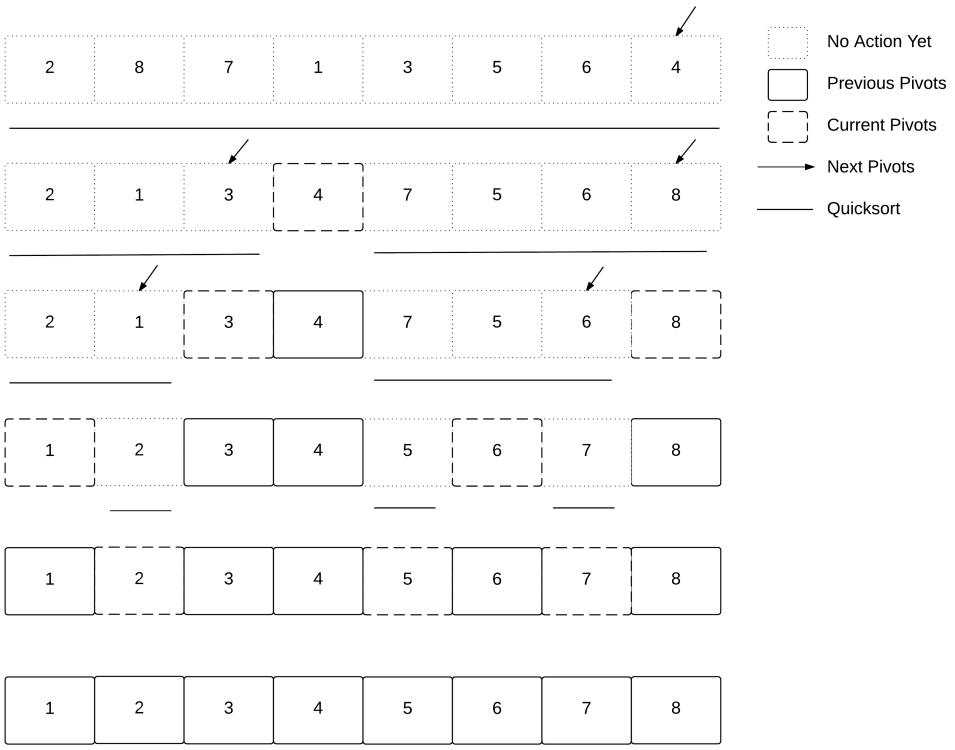


Figure 2.4: Execution of QuickSort on an array

$$T(n) = \begin{cases} \cancel{T(0)} + T(n-1) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

In terms of recursion tree, this degenerates in a tree where each level has a complexity linear in the input size. Each level has a single child and works on an input equal to the previous input decreased by a unit. As a result, the complexity is $\sum_{i=0}^{n} c(n-i)$, since at level i we work on an input of size $n-i$ and the complexity is linear. By calling $q = n - i$ we can transform the summation as

$$\sum_{i=0}^{n} c(n-i) = \sum_{q=1}^{n} cq = \frac{n(n=1)}{2} = O(n^2).$$

Therefore in the worst case Quicksort has quadratic complexity. The best case, instead, occurs when the position of the pivot value is in the middle of the array, i.e. $k \approx n/2$. We can rewrite the general equation as follows.

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

We have seen this equation already, as an example for the MergeSort algorithm, and we know leads to a complexity of $\Theta(nlog(n))$.

2.3.3 Selection in Worst-Case Linear Time

We now discuss a selection algorithm $SELECT$ that finds the desired element by recursively partitioning the input array. Also, algorithm $SELECT$ is deterministic because it uses the deterministic $PARTITION$ algorithm from quicksort (refer to Algorithm 8). However, it is a slightly modified version of the of the deterministic Partition routine, where the partition element is an input to this algorithm. In the worst case, algorithm $SELECT$ can achieve linear-time complexity. However, it needs to guarantee that the split is always *good* at each Partition. Now, we build a solution so a good split will always be guaranteed.

Pseudo-code:

```
1  SELECT()begin
2  |  /* index denotes the index of x that Partition() returns.*/
3  |  index = PARTITION(n)
4  |  if index = i then
5  |  |  return x.
6  |  end
7  |  if i<index then
8  |  |  apply SELECT recursively to A[1...index − 1] to find the i^th smallest
   |  |   element.
9  |  end
10 |  else
11 |  |  apply SELECT recursively to A[index + 1...n] to find the (i − k)^th smallest
   |  |   element.
12 |  end
13 end
```

Algorithm 9: SELECT

```
1  PARTITION(n)begin
2  |  Divide the n elements into M groups where, M = ⌈n/5⌉
3  |  Determine the median of each of the groups.
4  |  recursively find the median x of all the medians found in previous step.
5  end
```

Algorithm 10: $PARTITION$ for Algorithms $SELECT$

Worst-case Split: There are $\lceil \lfloor \frac{n}{5} \rfloor /2 \rceil$ groups with 3 elements (out of 5) is greater than or equals to x. Note that, the last group is ignored if it has fewer than 5 elements. So, the number of elements which are greater (resp. smaller) than x is at least $3(n-4)/10$. Hence, $SELECT$ is called recursively on at most $(7n+12)/10$ elements in the worst case.
Now, we define the recurrence relation for worst-case running time:
T(Select)$\leq$ T(Median-of-medians) + T(Partition) + T(recursive call to select)

$T(n) \leq O(n) + T(\lfloor n/5 \rfloor) + O(n) + T((7n+12)/10)$
Where, first two terms stands for T(Median-of-medians), third terms is for T(Partition), and the last term is for T(recursive call to select). From this relation we get:
$T(n) \leq T(\lfloor n/5 \rfloor) + O(n) + T((7n/10)) + 1.2)$
Solving the Recurrence:
Base: For all $n \leq 24$, $T(n) \leq 24n$
For $n > 24$, $T(n) \leq an + T(n/5) + T((7n/10)) + 1.2)$
We want to find a constant c, where $c > 0$ and for all $n > 0$ $T(n) \leq cn$. Now,

$$\begin{aligned} T(n) &\leq an + T(n/5) + T((7n/10)) + 1.2) \\ &\leq an + cn/5 + 7nc/10 + 1.2c \\ &= cn - (cn/10 - an - 1.2c) \\ &= cn - ((c/20 - a)n + (n/20 - 1.2)c) \\ &\leq cn; \text{as long as } c \geq 20a. \end{aligned}$$

Hence, in the worst-case, algorithm $SELECT$ has a linear running time.

2.4 Counting Sort - Sorting in Linear Time

It can be shown that *any* sorting algorithm based on comparisons has a complexity $\Omega(n \log(n))$. This implies, for example, that MergeSort is asymptotically optimal. However, we can improve this bound by designing algorithms not based on comparisons, Counting Sort is an example of these algorithms.

Algorithm idea. Counting sort assumes that the maximum value k in the array A is a $\Theta(n)$. The algorithm creates an array $C[1, \ldots, k]$. It then iterates over the array A and keeps track in C of how many times a given number occurs in A. It then iterates on C, and writes on A the numbers sorted.

2.4.1 Pseudo-code

2.4.2 Complexity

The pseudo-code of counting sort has several independent loops. The for loop (lines 3-7) has complexity $\Theta(n)$. The for loop (lines 9-11) has complexity $\Theta(k)$. The for loop (lines 12-14) has complexity $\Theta(n)$. The while loop needs a slightly more deep analysis, since j is not increased at every iteration. We should ask ourselves how many times the condition of the `If` can be true. This can happen only $\Theta(n)$ times, since it occurs only once for each element of A. When the `If` condition is not true, the else is executed and j increased. Hence the complexity of the while loop is $\Theta(n + k)$, since the `If` occurs n times and the else k.

Combining the above complexities we obtain $\Theta(n + k + n + n + k)$. Since we assumed that $k = \Theta(n)$, the overall complexity is $\Theta(n)$.

```
1  CountingSort(A, n)begin
2      k = A[1];
3      for i = 2 to n do
4          if A[i] > max then
5              k = A[i]
6          end
7      end
8      Create a new array C[0...k];
9      for j=0 to k do
10         C[i]=0 ;
11     end
12     for i=1 to n do
13         C[A[i]]=C[A[i]]+1;
14     end
15     i = 1;
16     j = 1;
17     while j ≤ k do
18         if C[0] ≠ 0 then
19             A[i] = j;
20             C[j]–;
21             i++;
22         end
23         else
24             j++;
25         end
26     end
27 end
```

Algorithm 11: CountingSort

2.5 Exercises

1. *The Carnival Coin Game:* You are running a game booth at your local village carnival. In your game you lay out an array of n ($n > 0$) coins on a table. In this game, you and your customer alternately pick coins from the table, either 1 or 2 at a time. If your customer can make you pick up the last coin, he wins and walks away with all the coins. You graciously allow your customer to go first. Being an enterprising sort, you want to arrange the game so that you will always win.
a) What constraint(s) for n do you need to guarantee a win every time?
b) Describe a correct algorithm (strategy).
c) What is the loop invariant for the algorithm?

2. *The Max Sub-vector Problem:* Given an array $a[1 \dots n]$ of numeric values (can be positive, zero and negative) determine the maximum value of sums to all sub-vectors $a[i \dots j](1 \leq i \leq j \leq n)$. Show that: Maximum Sub-vector is of $\Omega(n)$. (**Hint:** How to treat a subvector with negative sum during the search process?)

3. Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.
a. Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, selection sort, counting sort, radix sort.
b. Show that quicksort is not stable.
c. Modify quicksort and make it stable. (**Hint:** You need new comparison rule for two elements with the same value)

4.

a. Demonstrate the operation of $HOARE-PARTITION$ on the array $A = [13; 19; 9; 5; 12; 8; 7; 4; 11; 2; 6; 21]$, showing the values of the array and auxiliary values after each iteration of the **while** loop in lines 5-18. The next three questions ask you to give a careful argument that the procedure $HOARE - PARTITION$ is correct. Assuming that the subarray $A[p \dots r]$ contains at least two elements, prove the following:
b. The indices i and j are such that we never access an element of A outside the subarray $A[p \dots r]$.
c. When $HOARE - PARTITION$ terminates, it returns a value j such that $p \leq j < r$.
d. Every element of $A[p \dots j]$ is less than or equal to every element of $A[j + 1 \dots r]$ when $HOARE - PARTITION$ terminates.
e. Rewrite the $QUICKSORT$ procedure to use $HOARE - PARTITION$.
(Note that, the $HOARE - PARTITION$ procedure, always places the pivot value (originally in $A[p]$) into one of the two partitions $A[p \dots j]$ and $A[j + 1 \dots r]$. Since $p \leq j < r$, this split is always nontrivial.)

```
1  HOARE-PARTITION(A, p, r)begin
2  |  x = A[p];
3  |  i = p-1;
4  |  j = r+1;
5  |  while TRUE do
6  |  |  while A[j] ≤ x do
7  |  |  |  j = j-1;
8  |  |  end
9  |  |  while A[j] ≥ x do
10 |  |  |  i = i+1;
11 |  |  end
12 |  |  if i < j then
13 |  |  |  exchange A[i] with A[j];
14 |  |  end
15 |  |  else
16 |  |  |  return j
17 |  |  end
18 |  end
19 end
```

Algorithm 12: HOARE-PARTITION

5. a. If all element values are equal, then What would be randomized quicksorts running time?
b. The $PARTITION$ procedure returns an index q such that each element of $A[p \dots q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1 \dots r]$ is greater than $A[q]$. Modify the $PARTITION$ procedure to produce a procedure $PARTITION'(A, p, r)$, which permutes the elements of $A[p \dots r]$ and returns two indices q and t, where $p \leq q \leq t \leq r$, such that

- all elements of $A[q \dots t]$ are equal,
- each element of $A[p \dots q-1]$ is less than $A[q]$, and
- each element of $A[t+1 \dots r]$ is greater than $A[q]$

Like $PARTITION$, your $PARTITION'$ procedure should take $\Theta(r-p)$ time.
c. Modify the $RANDOMIZED-QUICKSORT$ procedure to call $PARTITION'$, and name the new procedure $RANDOMIZED{-}QUICKSORT'$. Then modify the $QUICKSORT$ procedure to produce a procedure $QUICKSORT'(p, r)$ that calls $RANDOMIZED{-}PARTITION'$ and recurses only on partitions of elements not known to be equal to each other.

6. Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

7. Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the

best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.
a. Sort the numbers, and list the i largest.
b. Build a max-priority queue from the numbers, and call $EXTRACT - MAX$ i times.
c. Use an order-statistic algorithm to find the i th largest number, partition around that number, and sort the i largest numbers.

8. Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

9.Show that we can use a depth-first search of an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v are in the same connected component.

10. Show that quicksorts best-case running time is $\Omega(nlgn)$.

Chapter 3

Greedy Approach to Algorithm Design

Greedy Algorithms are useful in a wide array of cases, especially dealing with optimization problems defined in a particular domain. These are commonly problems of optimizing (maximizing/minimizing) over a given input (e.g. array of numbers, sets, graph, etc.).

Greedy algorithms are generally iterative, and at each iteration the algorithm makes the "best" currently available decision according to its selection criteria. These algorithms build a partial solution which is extended at each iteration. Every time the current best element is added to the solution, and it is not removed in the subsequent iterations.

Greedy Algorithms do not always find the optimal solution to a problem. The theory of Matroids defines the necessary and sufficient condition under which a greedy algorithm returns an optimal solution. If a greedy algorithm does not solve a problem optimally, often it provides a solution with provable approximation bounds.

3.1 The Activity Selection Problem

The activity selection problem consist in scheduling activities (e.g. classes) that should take place in a classroom. Obviously, in the schedule activities cannot overlap, and our interest is to maximize the number of activities in the schedule. More formally,

- Consider n activities $A = [a_1 \cdots a_n]$, from which a subset should be selected to occur in one classroom
- The activity a_i has a start time s_i and a finish time f_i, such that $0 \leq s_i < f_i < \infty$.
- When an activity is scheduled, it takes place in the open interval $[s_i, f_i)$
- Two activities $a_1 = [s_i, f_i)$ and $a_2 = [s_j, f_j)$ are *compatible* if they do not overlap, i.e. if $s_i \geq f_j$ or $s_j \geq f_i$.

Problem: Given the activities in A, select the maximum number of compatible activities.

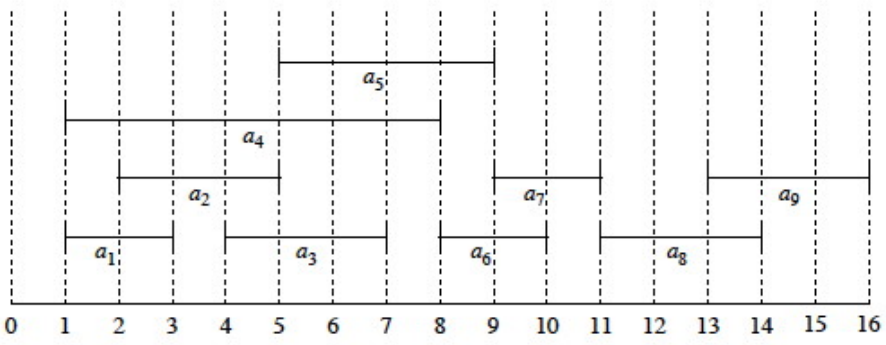


Figure 3.1: An example of input for the activity selection problem.

It is interesting to note that if instead of maximizing the number of classes to be scheduled, we want to maximize the amount of class time allocated, the problem becomes NP-Hard, and in fact cannot be solved in polynomial time.

3.1.1 An Example

Figure 3.1 shows an example of input for the activity selection problem. In this example, one optimal, but not the only optimal, solution would consist of the activities $\{a_1, a_3, a_6, a_8\}$. In any feasible solution, given this set of activities, the maximal number of activities which may be compatible is 4. Therefore any feasible solution which consists of exactly 4 compatible activities is optimal.

3.1.2 Potential Greedy Solutions

According to the greedy philosophy, we build a solution set S of non-overlapping activities. We extend S at each iteration of the algorithm, and once an activity is included in S, we never remove it from the solution. However, we have several options to pick the activity to include in S at each iteration. Possible examples are the following.

1. Earliest start time
2. Least duration
3. Earliest termination time

Earliest start time

Algorithm 13 shows the pseudo code for the earliest termination criteria. We begin with an empty solution set S. At each iteration of the while loop, we calculate the set D of the activities that are compatible with the activities in S, and we pick the activity a_k with the minimum start time. We add a_k in S and proceed to the next iteration. We keep iterating until there are activities that can be added to S without creating overlaps.

```
1 EarliestStart(A)begin
2 |  S=∅
3 |  while ∃ a ∈ A : S ∪{a} does not create overlaps do
4 |  |  Let D ⊆ A \ S be the set of activities that do not overlap with activities in S
5 |  |  a_k = arg min_{a_i ∈ D} s_i
6 |  |  S=S∪{a_k}
7 |  end
8 |  return S
9 end
```

Algorithm 13: Earliest Start time as greedy criteria

a_1	[1,10)
a_2	[2,3)
a_3	[4,5)

Table 3.1: Activity set to serve as counterexample to EarliestStart

We can easily find a *counter example* to show that the earliest start time criteria is not optimal, i.e. it does not always find an optimal solution. Finding a counter example is a general way of proving that an algorithm is not optimal, the idea is to find an input under which the algorithm provides an output that is clearly not optimal. In this case, we can consider the input in Table 3.1. With the greedy criteria being earliest start, a_1 alone is chosen, since it conflicts with both of the other two potential activities. We can instead select a_2 and a_3 and provide a solution with two activities. This selection criteria therefore does not lead to an optimal solution, as proven by the counter example.

Least duration

```
1 LeastDuration(A)begin
2 |  S=∅
3 |  while ∃ a ∈ A : S ∪{a} does not create overlaps do
4 |  |  Let D ⊆ A \ S be the set of activities that do not overlap with activities in S
5 |  |  a_k = arg min_{a_i ∈ D} (f_i − s_i)
6 |  |  S=S∪{a_k}
7 |  end
8 |  return S
9 end
```

Algorithm 14: Least duration as greedy criteria

Another valid option is to look first for activities which have a short duration. The idea

a_1	[5,7)
a_2	[3,6)
a_3	[6,9)

Table 3.2: Activity set to serve as counterexample to LeastDuration

is to select small activities first, so more should fit in the schedule. The pseudo code is in Algorithm 14.

However, not even this approach is optimal. Let us consider the input in Table 3.2. In this case, the algorithm would pick a_1 alone, since it conflicts with both of the other two potential activities. The optimal solution is instead a_2, a_3. This selection criteria therefore does not lead to an optimal solution, as proven by the counter example.

Earliest termination

```
1 EarliestTermination(A)begin
2 |  S=∅
3 |  while ∃ a ∈ A : S ∪{a} does not create overlaps do
4 |  |  Let D ⊆ A \ S be the set of activities that do not overlap with activities in S
5 |  |  a_k = arg min_{a_i ∈ D} f_i
6 |  |  S=S∪{a_k}
7 |  end
8 |  return S
9 end
```

Algorithm 15: Earliest Finish time as greedy criteria

Another option is to consider the earliest finish time. The intuition behind it is to select activities that leave more room for others. The pseudo-code is in Algorithm 15. We can easily see that this approach solves both the previous counter examples. However, can we find another counter example? If not, can we conclude that it is optimal?

3.1.3 Proving Greedy Algorithm Correctness

In general, proving the correctness of an algorithm requires creativity, ingenuity, and also a fair amount of luck. Although this is true for greedy algorithms too, we can provide some general guidelines on how the proof can be structured.

1. Prove that for any input, the algorithms terminates in finite number of steps
2. Prove that the partial solution produced at every iteration of the algorithm is always included in an optimal solution, i.e. there exist S^* optimal solution, such that for any iteration we can prove that $S \subseteq S^*$. This is normally achieved through a proof by induction.

3. Prove that the final solution is optimal, i.e. it cannot be further extended.

Correctness of the Earliest Termination solution

We structure the proof according to the general guidelines.

1) At each iteration of the while loop we add a new activity to S. This can be repeated at most for all activities in A, i.e. for at most n iterations.

2) Let S_h be the solution at the h-th iteration. Let m be the total number of iterations of the while loop, hence $h = 0, \ldots, m$. We need to prove that $\forall h = 0, \ldots, m \; \exists$ an optimal solution S^* s.t. $S_h \subseteq S^*$. We prove this by induction.

Base case, $h = 0$: the statement is true because $S_0 = \emptyset$, and $\emptyset$ is a subset of any set, and thus $S_0 \subseteq S^*$ for any optimal solution S^*.

Inductive hypothesis: The statement is true at the h-th iteration, i.e. $\exists$ an optimal solution S^* s.t. $S_h \subseteq S^*$.

Inductive step: We prove that the statement is true at the iteration $h + 1$. We know that $S_h \subseteq S^*$. Let a_k be the activity selected at the $h + 1$ iteration.

If $a_k \in S^*$ the statement is true, since S^* is the optimal solution that includes S_{h+1}.

If $a_k \notin S^*$ we need to find another optimal solution that contains a_k and all the activities in S_h. In particular, we look for an activity in S^* that can be substituted with a_k. Such activity exists because $|S_{h+1}| \leq |S^*|$. Let a_j be the activity in $S^* \setminus S_h$ with the earliest termination and let $S^{\#} = (S^* \setminus \{a_j\}) \cup \{a_k\}$

We need to prove that $S^{\#}$ is feasible (i.e. does not contain overlaps) and it is also optimal. Since $a_j \in S^*$ and $a_j \notin S_h$ a_j could have been selected by our algorithm at the $h+1$ iteration, but the algorithm selected a_k instead. This implies that the termination time of a_k is less than or equal to the termination time of a_j, i.e. $f_k \leq f_j$. As a result, a_k does not overlap with the activities in S^* and hence $S^{\#}$ is feasible.

To build $S^{\#}$ we removed the activity a_j from S^* and added the activity a_k. As a result, we have $|S^{\#}| = |S^*|$, hence $S^{\#}$ is optimal because it maximizes the number of selected activities.

3) We need to prove that S_m, i.e. the solution produced at the last iteration of the algorithm, is indeed optimal. We know that exists an optimal solution S^* s.t. $S_m \subseteq S^*$. We proceed by contradiction. Let us assume that $|S_m| < |S^*$, i.e. S_m is not optimal. Hence, there exists $a \in S^*$ s.t. $a \notin S_m$.

Since a does not overlap with any activity in S^*, it does not neither overlap with the activities in S_m, because $S_m \subset S^*$. As a result, the condition of the while loop would have been true at the $m+1$ iteration, since there exist an activity in A that does not overlap with the activities already selected in S_m. The algorithm would have then selected a and added it to S_m, hence we get a contradiction since S_m cannot be the final solution of the algorithm.

This implies that $|S_m| = |S^*|$, hence S_m is optimal.

```
1  EarliestTermination(A)begin
2  |  S=∅;
3  |  Sort A in a′_1, ..., a′_n, s.t. f_{a′_1} ≤ f_{a′_2} ≤ ··· ≤ f_{a′_n};
4  |  t = 0;
5  |  for j = 1 to n do
6  |  |  if s_{a′_j} ≥ t then
7  |  |  |  S = S ∪ {a′_j};
8  |  |  |  t = f_{a′_j};
9  |  |  end
10 |  end
11 |  return S;
12 end
```

Algorithm 16: Earliest Finish detailed pseudo-code.

3.1.4 More Detailed Psedo-code for Earliest Termination

The previous pseudo-code is more high-level, which is typically useful to prove the correctness of an algorithm, where we focus on the algorithm idea rather than on the implementation details. However, this does not allow us to evaluate the complexity, since many operations are not specified in sufficient detail. As an example, we do not clarify how we actually pick the next activity with earliest termination time, or how we check that the activity that we pick does not overlap with the activities already in S. Therefore, there is another step to do, which is not trivial, to make these aspects explicit and evaluate the algorithm complexity.

Algorithm 16 shows the more detailed pseudo-code. The key idea is to sort the activities by finish time before starting the selection. This can be done in $\Theta(n \log(n))$ using one of the sorting algorithms we already discussed. Sorting by finish time allows us to iterate over the sorted set, and know that an activity has a finish time smaller than all the subsequent ones.

We need however to solve also the problem of checking compatibility efficiently. Going through all the already selected activities and verify if the potential new activity is compatible is very inefficient, and would cost $O(n)$ at each iteration. Since this cost would be incurred at every algorithm iteration, we would end up with a overall cost of $O(n \log(n) + n^2) = O(n^2)$.

We can however do something smarter, and consider an timestamp t which corresponds to the finish time of the last selected activity. This way, to asses if the new activity under consideration is compatible with all the other ones in S, we can just check if its starting time is larger or equal to t. This way, we can verify the compatibility in constant time $O(1)$, and the while loop has complexity $\Theta(n)$.

The overall complexity of the algorithm is then $\Theta(n \log(n) + n) = \Theta(n \log(n))$.

3.2 The Cashier Problem

A Cashier has to give W cents (integer) in change for a transaction and she wants to use the least number of coins. The available coin types belong to the set $C = \{c_1, \ldots, c_n\}$ and the

element c_i has value v_i. Formally, we want to solve this optimization problem.

$$S^* = \arg\min_{S \subseteq C} |S|$$
$$\text{s.t.} \sum_{c_i \in S} v_i = W$$

3.2.1 Examples in the US System

In the US system the set of values is $\{1, 5, 10, 25\}$. For $W = 30$ the optimal solution is $S^* = \{25, 5\}$, while for $W = 67$ is $S^* = \{25, 25, 10, 5, 1, 1\}$.

3.2.2 Greedy Solution

The idea behind the greedy approach is to pick the largest value coin with value less than or equal to W, then update remaining value of W. The pseudo-code is shown in Algorithm 17.

```
1 Change(W)begin
2 |  S=∅
3 |  while W > 0 do
4 |  |  c_k = arg max_{c_i ∈ C : v_i ≤ W} v_i
5 |  |  S = S ∪ {c_k}
6 |  |  W = W − v_k
7 |  end
8 |  return S
9 end
```

Algorithm 17: Change Algorithm

The algorithm yields an optimal solution for the US coinage system. However, this is not guaranteed to do so for all systems. As an example, consider the US coinage system without the nickel, i.e. $\{1, 10, 25\}$. We want to give a change for an amount $W = 30$ cents. The algorithm returns a solution $\{25, 1, 1, 1, 1, 1\}$ which is clearly not optimal, since we can use just three coins $\{10, 10, 10\}$.

3.3 The Knapsack 0-1 Problem

The Knapsack problem is a well-known problem in Computer Science, and many variants exist in its formulation. The variant that we consider here is the so called Knapsack 0-1 problem, which can be formulated as follows.

A thief has a sack which can hold a limited amount of weight W. He enters in a shop at night and has the option to pick the items to steal. There are a set of $A = \{a_1, a_2, \ldots, a_n\}$ items, each item $a_i \in A$ has a value v_i and a weight w_i. The thief has to find the best set

of items S^* that maximizes the overall value, and has an overall weight less than or equal to W. Formally the problem can be formalized as follows.

$$S^* = \arg\max_{S \subseteq A} \sum_{a_i \in S} w_i :\leq W$$

The name Knapsack 0-1 derives from the fact that the thief can pick an item or not, other variants for example enable to pick multiple copies of the same item.

3.3.1 Greedy Solution

Following the greedy strategy, we can approach the knapsack problem as follows. We start with an empty knapsack, $S = \emptyset$. We iteratively pick the "best" element, following an appropriate criteria, until the knapsack is full. Several criteria can be formulated, as an example we may pick the item a_i (i) with max value v_i, (ii) with minimum weight w_i, (iii) with maximum ratio $\frac{v_i}{w_i}$. Algorithm 18 summarizes the greedy strategy with these criteria.

```
1  GreedyKnapsack(A)begin
2      S=∅
3      R=W //R is the residual weight capacity of the knapsack
4      D=A
5      while D ≠ 0 do
6          a_k = arg max_{a_i∈D} v_i | arg max_{a_i∈D} v_i/w_i | arg min_{a_i∈D} w_i
7          if w_k ≤ R then
8              S=S∪{a_k}
9              R = R − w_k
10         end
11         D = D \ {a_k}
12     end
13     return S
14 end
```

Algorithm 18: Greedy Knapsack

3.3.2 Counter Examples and NP-Completeness

All the three criteria mentioned above are not optimal, i.e. for each of them there exist an input in which the greedy algorithm does not give the optimal solution. In order to show that an algorithm is not optimal, it is sufficient to provide such an input and show the resulting output compared to the optimal solution. This is in general much easier that showing the optimality. Obviously, if you find a counter example then showing the optimality is impossible, while if you show the optimality then you cannot find any counter example.

v_1	v_2	v_3	w_1	w_2	w_3	max v_i	min w_i	max $\frac{v_i}{w_i}$	Optimal
10	9	9	50	25	25	$\{a_1\}$, 10	$\{a_2, a_3\}$, 18	$\{a_2, a_3\}$, 18	$\{a_2, a_3\}$, 18
10	1	1	50	25	25	$\{a_1\}$, 10	$\{a_2, a_3\}$, 2	$\{a_1\}$, 10	$\{a_1\}$, 10
60	100	120	10	20	30	$\{a_3, a_2\}$, 120	$\{a_3, a_2\}$, 120	$\{a_1, a_2\}$, 160	$\{a_3, a_2\}$, 120

Table 3.3: Counter examples for knapsack greedy criteria.

We provide in table 3.3 some counter examples for each criteria. We consider three items and a knapsack with capacity $W = 50$.

It is possible to show that the Knapsack problem belongs to a class of problems known as NP-Complete. This means that there cannot exist a polynomial time algorithm that optimally solves the problem. Therefore, our greedy algorithms which are clearly polynomial, have no chance to be optimal. It is however possible to show that the greedy criteria based on max $\frac{v_i}{w_i}$, with some minor modification, yields a solution which is at least half of the optimal solution. We say that such a greedy approach has a provable *performance bound*.

3.4 Exercise

1. Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the $i-$th element of set A,and let b_i be the $i-$th element of set B. You then receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

2. Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

3. Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

4. Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

Chapter 4

Dynamic Programming

Dynamic programming solves optimization problems by combining solutions of sub-problems, but differently from divide and conquer, such subproblems are not disjoint, therefore individual sub-problems may be common components to multiple higher level sub-problems. To avoid computing the solution to the same sub-problem multiple times, which would be disastrous in terms of complexity, tables are generally used to keep track of the already calculated solutions. A bottom up approach is followed, where we start from simple problems (base cases) and use a recursive relation to solve higher level sub-problems combining the solutions of the lower level sub-problems stored in the tables.

4.1 General Dynamic Programming Approach

When an algorithm is designed using a dynamic programming approach, generally three steps are involved:

1. Identify sub-problems whose solution can lead to the solution of the bigger problem.
    - The number of sub-problems are polynomial with respect to the original problem.
    - Polynomial complexity to combine the solutions to obtain the solution to the original problem.
2. Define a recurrence relation that relates the solutions of smaller problems to the solutions of bigger problems.
3. First focus on calculating the value of the solution then on calculating the actual solution.

4.2 Max Sub-array Problem

We discussed this problem in Section 2.2. We briefly recall its definition here. Given an array A with positive and negative numbers, find the sub-array with maximum sum. We showed

that we can solve the problem in $\Theta(nlogn)$ using a divide and conquer approach. Dynamic programming can go do even better, and achieve a complexity of $\Theta(n)$.

We first define the table, and the sub-problems, that will be used by the algorithm. It is advised to fully understand the definition .

Definition 4.2.1. *Let T be a one dimensional table of length n. $\forall\ i = 1, \ldots, n$ T[i] is the sum of the elements in the max sub-array that ends at i.*

We can therefore re-define our problem compactly as $\max_{i=1,\ldots,n} T[i]$. We now need to define the recursive relation to calculate the solution $T[i]$ given the solutions $T[1], \ldots, T[i-1]$. Let's start with the base case, i = 1. The sum of the elements of the max sub-array that ends at 1 is clearly only $A[1]$. Now assume that you calculated all solutions up to $i-1$, we can have two cases. Either the max sub-array that ends at i is composed by the single element $A[i]$, so $T[i] = A[i]$, or it is composed by the previously max sub-array to which we append $A[i]$, that is $T[i] = T[i-1] + A[i]$. We can thus say that $T[i] = \max(T[i-1] + A[i], A[i])$. We thus summarize the discussion in following recursive relation.

$$T[i] = \begin{cases} A[1] & i = 1 \\ max(A[i], A[i] + T[i-1]) & i > 1 \end{cases}$$

Note that the only reason for which we may not decide to append the current item to the previous maximum sub-array is when $T[i-1] < 0$, this is exploited in the pseudo-code.

```
1  MaxSubarrayDP(A) begin
2  |  T[1]=A[1]
3  |  max = T[1]
4  |  for 1=2 to n do
5  |  |  T[i]=A[i]
6  |  |  if T[i − 1] > 0 then
7  |  |  |  T[i]=A[i]+T[i-1]
8  |  |  end
9  |  |  if max < T[i] then
10 |  |  |  max = T[i]
11 |  |  end
12 |  end
13 |  return max
14 end
```

Algorithm 19: Dynamic Programming MaxSubarray Algorithm

4.2.1 Pseudo-code

The hard work in defining dynamic programming algorithms is to identify the sub-problems and the recursive relation, while the pseudo-codes are pretty straight forward. Generally we

just iterate on the table, and in this case the table is just a one-dimensional array. Algorithm 19 shows the pseudo-code.

4.2.2 Example

Find the Maximum Subarray within $A =< -1, 2, 10, -13, 5, -10, 1, -2, -4 >$

A	-1	2	10	-13	5	-10	1	-2	-4
T	-1	2	12	-1	5	-5	1	-1	-4
max	-1	2	12	12	12	12	12	12	12
i	1	2	3	4	5	6	7	8	9

Figure 4.1: Step by step execution on the given array

```
1  MaxSubarrayDP(A) begin
2  |  T[1]=A[1]
3  |  max = T[1]
4  |  b=1
5  |  for 1=2 to n do
6  |  |  T[i]=A[i]
7  |  |  if T[i − 1] > 0 then
8  |  |  |  T[i]=A[i]+T[i-1]
9  |  |  end
10 |  |  if max < T[i] then
11 |  |  |  max = T[i]
12 |  |  |  b=i //this is the new end of the max sub-array
13 |  |  end
14 |  end
15 |  a=b
16 |  while max − A[a] ≠ 0 do
17 |  |  max=max-A[a]
18 |  |  a- -
19 |  end
20 |  return (a, b)
21 end
```

Algorithm 20: Dynamic Programming MaxSubarray Algorithm

4.2.3 Pseudo-code to Find the Actual Solution

We should now realize that the table T only gives us the value of the solution, i.e. the sum of the elements in the maximum sub-array, but it does not tell us what the actual sub array is.

We can now easily extend the algorithm to find the solution. The idea is to use to indices a and b, such that $0 \leq a \leq b \leq n$. The index a represents the start of the maximum sub-array while b the end. The idea is to first calculate the table T and b, then move backwards in the array starting at $A[b]$ to find a. The pseudo-code is shown in Algorithm 20.

4.3 Longest Common Subsequence

The Longest Common Sub-sequence (LCS) problem is a typical example that shows the power of Dynamic programming. It solves a problem apparently hard in polynomial time. Let's first define the concept of sequence and sub-sequence.

Definition 4.3.1 (Sequence and Sub-sequence)**.** *A sequence $X =< X_1, X_2, \ldots X_n >$ is an ordered list of symbols (e.g. numbers, characters, etc.). Given a sequence $X =< X_1, X_2, \ldots X_n >$, we say that $Z =< Z_1, Z_2, \ldots Z_k >$ is a sub-sequence of X if there exists a strictly increasing list of indices $< i_1, \ldots, i_k >$ such that $\forall\ j \in [1 \ldots k]\ X_{i_j} = Z_j$.*

As an example, given $X =< 9, 15, 3, 6, 4, 2, 5, 10, 3 >$, $Z =< 15, 6, 2 >$ is a subsequence of X, while $Y =< 4, 3, 10 >$ is not. The LCS problem definition is pretty straight forward.

Definition 4.3.2 (LCS problem definition)**.** *Given two sequences $X =< X_1, \ldots, X_n >$ and $Y =< Y_1, \ldots, Y_m >$ find the longest common sub-sequence, i.e.the longest sequence Z that is a sub-sequence for both X and Y.*

Example Consider the following sequences.

$$X =< 9, 15, 3, 6, 4, 2, 5, 10, 3 >$$

$$Y =< 8, 15, 6, 7, 9, 2, 11, 3, 1 >$$

The LCS is $< 15, 6, 2, 3 >$, since:

$$X =< 9, \underline{15}, 3, \underline{6}, 4, \underline{2}, 5, 10, \underline{3} > \text{ indices } < 2, 4, 6, 9 >$$

$$Y =< 8, \underline{15}, \underline{6}, 7, 9, \underline{2}, 11, \underline{3}, 1 > \text{ indices } < 2, 3, 6, 8 >$$

4.3.1 Dynamic Programming Solution

In order to describe the approach to solve the problem, we need to provide some definitions.

Definition 4.3.3 (Prefix X_i)**.** *Given a sequence $X =< X_1 \ldots X_n >$ its prefix X_i is defined as $< X_1, \ldots X_i >$, $\forall i \in [1, \ldots, n]$*

We now introduce the table T that will keep track of the already solved sub-problems during the algorithm execution. I encourage you to spend sometime to fully understand the meaning of this table and its definition before moving forward with the rest of the discussion.

Definition 4.3.4 (Table T)**.** *Given two sub-sequences X and Y, of length n and m, respectively. We define a table $T : n \times m$ where $T[i, j]$ is the length of the longest common sub-sequence of X_i and Y_j, $\forall i \in [1, \ldots, n]$ and $j \in [1, \ldots, m]$.*

As the definition of the table suggests, the sub-problems consist in finding the LCS for the prefix X_i and Y_j, progressively increasing i and j, until $i = n$ and $j = m$. At that point, $T[n, m]$ will contain the answer to the main problem.

We now need to identify a recursive relation to calculate the solution of a given sub-problem, given the solutions of the smaller sub-problems. In other words, we want to be able to calculate the value of $T[i, j]$, given that the table has been filled for all elements with indices less than i and j.

Every good recursive relation starts with a base case. Here the base case occurs when at least one of the two sequences is empty, in that case it is straight forward that the LCS is also empty. Formally, $T[i, 0] = 0$, $\forall i = 0, \ldots, n$, and $T[0, j] = 0$, $\forall j = 0, \ldots, m$.

When both i and j are greater than zero, there are two cases that may occur, depending on the symbols in $X[i]$ and $Y[j]$.

1. If $X[i] == Y[j]$ then these two elements are in common and the LCS between X_i and Y_j includes this symbol. Therefore, the length of such LCS is equal to the length of the LCS between X_{i-1} and Y_{i-1} plus one, i.e. $T[i, j] = T[i - 1, j - 1] + 1$.

2. If otherwise $X[i] \neq Y[j]$ then these symbols do not belong to the LCS. Therefore the LCS of X_i and Y_j is the longest between the LCS of X_i and Y_{j-1}, and the LCS of X_{i-1} and Y_j. That is, $T[i, j] = max(T[i - 1, j], T[i, j - 1])$.

We can summarize the recursive relation as follows.

$$T[i,j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ T[i-1, j-1] + 1 & \text{If } i, j > 0 \wedge X[i] = Y[j] \\ max(T[i-1, j], T[i, j-1]) & \text{If } i, j > 0 \wedge X[i] \neq Y[j] \end{cases}$$

4.3.2 LCS Pseudo-code

As usual with dynamic programming approaches, the bigger effort is to identify the sub-problems and the recursive relation that combines their solutions. Once this is done, the pseudo-code is relatively straight forward. The pseudo-code is show in Algorithm 21. The algorithm initially fills the table considering the base cases. Then, it iterates over all elements of the table T and fills the value of $T[i, j]$ applying the recursive relation accordingly.

4.3.3 Example

Find the LCS in the two sequences following:

$$X =< 9, 15, 3, 6, 4, 2, 5, 10, 3 >$$

$$Y =< 8, 15, 6, 7, 9, 2, 11, 3, 1 >$$

```
1  LCS(X,Y) begin
2  |  for i=0 to n do
3  |  |  T[i,0]=0
4  |  end
5  |  for j=0 to m do
6  |  |  T[0,j]=0
7  |  end
8  |  for i=1 to n do
9  |  |  for j=1 to m do
10 |  |  |  if x[i]==y[j] then
11 |  |  |  |  T[i,j]=T[i-1,j-1]+1
12 |  |  |  end
13 |  |  |  else
14 |  |  |  |  T[i,j]=max(T[i-1,j],T[i,j-1])
15 |  |  |  end
16 |  |  end
17 |  end
18 |  return T[n,m]
19 end
```

Algorithm 21: Dynamic Programming LCS Algorithm

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1
2	0	0	1	1	1	1	1	1	1	1
3	0	0	1	1	1	1	1	1	2	2
4	0	0	1	2	2	2	2	2	2	2
5	0	0	1	2	2	2	2	2	2	2
6	0	0	1	2	2	2	3	3	3	3
7	0	0	1	2	2	2	3	3	3	3
8	0	0	1	2	2	2	3	3	3	3
9	0	0	1	2	2	2	3	3	4	4

Table 4.1: Example table T filled by the LCS algorithm

In this example $n = m = 9$. The Table 4.1 shows the table T filled with values. We know that $T[9, 9]$ contains the length of the LCS, that is 4. However, we are not still able to provide the actual symbols that compose the LCS.

4.3.4 PrintLCS Pseudo Code and Execution

In order to calculate the actual LCS, we use an additional recursive function `PrintLCS()`. The function starts from the bottom right cell of the table T and backtracks the decisions of the algorithms in order to reconstruct the LCS. Specifically, at each recursive call the function considers the table T and the positions i and j, initially set to n and m. It compares $X[i]$ and $Y[j]$. If they are equal, then the function prints the symbol and calls itself recursively on the prefixes of length $i-1$ and $j-1$. If otherwise they are different, the symbol may not be in the LCS, and the function calls itself on the maximum between $T[i-1,j]$ and $T[i,j-1]$. The pseudo-code is shown in Algorithm 22.

```
1  PrintLCS(T,i,j)begin
2  |  if  i > 0 and j > 0 then
3  |  |  if X[i] == Y[j] then
4  |  |  |  PrintLCS(T, i-1,j-1)
5  |  |  |  Print(X[i])
6  |  |  end
7  |  |  else
8  |  |  |  if T[i − 1, j] ≥ T[i, j − 1] then
9  |  |  |  |  PrintLCS(T,i-1,j]
10 |  |  |  end
11 |  |  |  else
12 |  |  |  |  PrintLCS(T,i,j-1)
13 |  |  |  end
14 |  |  end
15 |  end
16 end
```

Algorithm 22: Printing the elements of the LCS

4.3.5 Example - Actual LCS

We now extend the previous example to reconstruct the LCS by executing `PrintLCS()`. We highlighted in bold the path followed by the function. Additionally, we underlined the recursive calls for which $X[i] == Y[j]$, and thus those symbols will be part of the LCS. As a result, the LCS is $< 15, 6, 2, 3 >$, which has length 4 as expected.

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	***0***	0	0	0	1	1	1	1	1
2	0	0	**1**	1	1	1	1	1	1	1
3	0	0	**1**	1	1	1	1	1	2	2
4	0	0	1	**2**	**2**	**2**	**2**	2	2	2
5	0	0	1	2	2	2	**2**	2	2	2
6	0	0	1	2	2	2	**3**	3	3	3
7	0	0	1	2	2	2	**3**	3	3	3
8	0	0	1	2	2	2	3	**3**	3	3
9	0	0	1	2	2	2	3	3	**4**	**4**

Figure 4.2: Traceback through T of the LCS in the PrintLCS algorithm

4.4 Dynamic Programming Solution for the knapsack problem

We now consider again the knapsack problem and give a solution based on dynamic programming. Surprisingly, this approach yields an optimal solution. To understand how an apparently polynomial algorithm can solve a Np-Hard problem, we need to dig a bit deeper into the complexity analysis of the solution. Let's first recall the knapsack problem.

Problem Description: Consider a set of items $A = \{a_1 \dots a_n\}$ with values $\{v_1 \dots v_n\}$ and weights $\{w_1 \dots w_n\}$. Given a knapsack of capacity W, find a set of items $S \subseteq A$ that maximize the sum of values, and their cumulative weight is less than or equal to W.

The idea of the dynamic programming approach is to decompose the main problem in subproblems which have less items and smaller capacity of the knapsack. Gradually increasing the available elements and the size of the knapsack we get to the actual solution of the problem. As usual, we make use of a table T defined as follows.

Definition 4.4.1. *Given a set of items $A = \{a_1 \dots a_n\}$ and a knapsack of capacity W, we define a table $T : n \times W$. The elemeng $T[i, j]$ is the value of the solution of the knapsack problem considering only the first $\{a_1 \dots a_n\}$ items, and a knapsack of capacity j.*

In order to define the recursive relation, let's first consider the base cases. The problem is straight forward to solve if there are no items to select ($i = 0$) or if the knapsack has zero capacity ($j = 0$). Let's consider the generic case with $i, j > 0$, and define the recursive relation for $T[i, j]$. It may occur that the item a_i has a weight w_i that exceeds j, in this case, the solution to the problem cannot include a_i, therefore it is the same as the solution with the elements $a_1, \dots, a_{i-1}$ and same capacity j. In other words, in this case $T[i, j] = T[i - 1, j]$. If otherwise $w_i \leq j$ we should decide weather to pick the item a_i or not. If we pick it, we are left with the items $a_1, \dots, a_{i-1}$ and a residual capacity $j - w_i$, but we gained a value v_i

by including a_i in the solution. If we do not pick a_i, instead, we are still left with the items $a_1, \ldots, a_{i-1}$, the knapsack capacity j is unaltered, but we gain no value. To optimal solution should be the best among these two, therefore $T[i,j] = max(v_i + T[i-1, j-w_i], T[i-1,j])$. We can summarize the above discussion as follows.

$$T[i,j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ T[i-1,j] & w_i > j \\ max(v_i + T[i-1, j-w_i], T[i-1,j]) & i > 0 \wedge j > 0 \wedge w_i \leq j \end{cases}$$

4.4.1 Pseudo-code

Once again, once the recursive relation is unveiled, it is simple to write the pseudo-code. The algorithm simple initializes the table T for the base cases, and then has two nested loops to iterate on every element of the table applying the recursive formula with a bottom up approach. The algorithm is shown in Figure 23.

```
1  KnapsackDP(A, W)begin
2  |  for i=1 to n do
3  |  |  T[i,0]=0
4  |  end
5  |  for j=i to W do
6  |  |  T[0,j]=0
7  |  end
8  |  for i=1 to n do
9  |  |  for j=1 to W do
10 |  |  |  if w_i > j then
11 |  |  |  |  T[i,j]=T[i-1,j]
12 |  |  |  end
13 |  |  |  else
14 |  |  |  |  T[i,j]=max(T[i-1,j], v_i + T[i-1, j-w_i])
15 |  |  |  end
16 |  |  end
17 |  end
18 |  return T[n,W]
19 end
```

Algorithm 23: KnapsackDP Pseudo Code

4.4.2 Example

Consider the set of items with their associated values and weights in Table 4.3. The execution of the Dynamic programming solution to the knapsack problem is given in Table 4.4.

Item	Value	Weight
a_1	1	1
a_2	6	2
a_3	18	5
a_4	22	6
a_5	28	7

Figure 4.3: Set of Items A upon which to execute the KnapsackDP algorithm

$A \setminus W$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

Figure 4.4: The table T after the execution of the algorithm

4.4.3 Calculate the Actual Solution

Similarly to the LCS problem, we have until now only calculated the value of the problem solution, but not the actual items in the knapsack. Also in this case, we can recursively backtrack the algorithm actions from the table, starting in the position $T[n, W]$. For each position, if $T[i, j] == v_i + T[i-1, j-w_i]$ then the algorithm picked a_i, and we can print it, otherwise it did not, and we can call recursively on $T[i-1, j]$. Algorithm 24 shows the pseudo-code.

```
1  PrintKnapsack(A, T, i, j)begin
2  |  if i ≤ 0 ∨ j ≤ 0 then
3  |  |  return
4  |  end
5  |  if T[i, j] == v_i + T[i − 1, j − w_i] then
6  |  |  print a_i
7  |  |  PrintKnapsack(A, T, i − 1, j − w_i)
8  |  end
9  |  else
10 |  |  PrintKnapsack(A, T, i − 1, j)
11 |  end
12 end
```

Algorithm 24: PrintKnapsack Pseudo Code

We represent in bold the position of the table where the function `PrintKnapsack` traces

back, while we underline the items that have been picked to be part of the solution. Specifically, if $T[i, j]$ is underlined than the algorithm picked the item a_i.

$A \setminus W$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	***0***	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	**18**	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	**40**
5	0	1	6	7	7	18	22	28	29	34	35	**40**

Figure 4.5: Traceback through T of KnapsackDP in the PrintKnapsack algorithm

4.4.4 A Discussion on Computational Complexity

Looking at Algorithm 23 it is straight forward to conclude that the complexity is $\Theta(nW)$. However, we discussed that this problem is NP-Complete, thus the complexity to find the optimal solution should be exponential. To understand this apparent contradiction, we need to think at the meaning of "size of the input". It is straight forward that n is the size of the set A, however W is a number and the size of this input is proportional to $\log(W)$, since this is the number of bits necessary to represent W. As a result, the complexity is indeed exponential in the size of the input, since it depends on W although the input is of size $\log(W)$. The knapsack problem is called weakly NP-complete for this reason.

4.5 Exercise

1. In order to transform one source string of text $x[1 \ldots m]$ to a target string $y[1 \ldots n]$, we can perform various transformation operations. Our goal is, given x and y, to produce a series of transformations that change x to y. We use an array $\mathcal{Z}$ assumed to be large enough to hold all the characters it will need-to hold the intermediate results. Initially, $\mathcal{Z}$ is empty, and at termination, we should have $\mathcal{Z}[j] = y[j]$ for $j = 1, 2, \ldots, n$. We maintain current indices i into x and j into $\mathcal{Z}$, and the operations are allowed to alter $\mathcal{Z}$ and these indices. Initially, $i = j = 1$. We are required to examine every character in x during the transformation, which means that at the end of the sequence of transformation operations, we must have $i = m + 1$. We may choose from among six transformation operations:
Copy a character from x to $\mathcal{Z}$ by setting $\mathcal{Z}[j] = x[i]$ and then incrementing both i and j. This operation examines $x[i]$.
Replace a character from x by another character $\mathcal{C}$ by setting $\mathcal{Z}[j] = \mathcal{C}$ and then incrementing both i and j. This operation examines $x[i]$.
Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$.
Insert the character $\mathcal{C}$ into $\mathcal{Z}$ by setting $\mathcal{Z}[j] = \mathcal{C}$ and then incrementing j, but leaving i

alone. This operation examines no characters of x.
Twiddle(i.e., exchange) the next two characters by copying them from x to $\mathcal{Z}$ but in the opposite order; we do so by setting $\mathcal{Z}[j] = x[i+1]$ and $\mathcal{Z}[j+1] = x[i]$ and then setting $i = i+2$ and $j = j+2$. This operation examines $x[i]$ and $x[i+1]$.
Kill the remainder of x by setting $i = m+1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation.

As an example, one way to transform the source string *algorithm* to the target string *altruisticis* to use the following sequence of operations, where the underlined characters are $x[i]$ and $\mathcal{Z}[j]$ after the operation:

Operation	x	$\mathcal{Z}$
initial strings	$\underline{a}lgorithm$	$\_$
copy	$a\underline{l}gorithm$	$a\_$
copy	$al\underline{g}orithm$	$al\_$
replace by t	$alg\underline{o}rithm$	$alt\_$
delete	$algo\underline{r}ithm$	$alt\_$
copy	$algor\underline{i}thm$	$altr\_$
insert u	$algor\underline{i}thm$	$altru\_$
insert i	$algor\underline{i}thm$	$altrui\_$
insert s	$algor\underline{i}thm$	$altruis\_$
Twiddle	$algori\underline{t}hm$	$altruisti\_$
insert c	$algori\underline{t}hm$	$altruistic\_$
kill	$algorithm\_$	$altruistic\_$

Note that there are several other sequences of transformation operations that transform *algorithm* to *altruistic*.

Each of the transformation operations has an associated cost. The cost of an operation depends on the specific application, but we assume that each operation's cost is a constant that is known to us. We also assume that the individual costs of the copy and replace operations are less than the combined costs of the delete and insert operations; otherwise, the copy and replace operations would not be used. The cost of a given sequence of transformation operations is the sum of the costs of the individual operations in the sequence. For the sequence above, the cost of transforming *algorithm* to *altruistic* is
(3.cost(copy)) + cost(replace) + cost(delete) + 4.cost(insert) + + cost(twiddle) + cost(kill).
a. Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$ and set of transformation-operation costs, the **edit distance** from x to y is the cost of the least expensive operation sequence that transforms x to y. Describe a dynamic-programming algorithm that finds the edit distance from $x[1 \ldots m]$ to $y[1 \ldots n]$ and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

The edit-distance problem generalizes the problem of aligning two DNA sequences. There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences x and y consists of inserting spaces at arbitrary

locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ a space). Then we assign a '*score*' to each position. Position j receives a score as follows:

- +1 $x'[j] = y'[j]$ and neither is a space,
- -1 $x'[j] \neq y'[j]$ and neither is a space,
- -2 if either $x'[j]$ or $y'[j]$ is a space.

The score for the alignment is the sum of the scores of the individual positions. For example, given the sequences $x = GATCGGCAT$ and $y = CAATGTGAATC$, one alignment is
$G\ ATCG\ GCAT$
$CAAT\ GTGAATC$
-*++*+*+-++*
A + under a position indicates a score of +1 for that position, a − indicates a score of 1, and a $*$ indicates a score of 2, so that this alignment has a total score of $6.1 - 2.1 - 4.2 = -4$.
b. Explain how to cast the problem of finding an optimal alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

2. We are given a color picture consisting of an $m \times n$ array $A[1 \ldots m, 1 \ldots n]$ of pixels, where each pixel specifies a triple of red, green, and blue (**RGB**) intensities. Suppose that we wish to compress this picture slightly. Specifically, we wish to remove one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid disturbing visual effects, however, we require that the pixels removed in two adjacent rows be in the same or adjacent columns; the pixels removed form a '*seam*' from the top row to the bottom row where successive pixels in the seam are adjacent vertically or diagonally.
a. Show that the number of such possible seams grows at least exponentially in m, assuming that $n > 1$.
b. Suppose now that along with each pixel $A[i, j]$, we have calculated a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixels disruption measure, the more similar the pixel is to its neighbors. Suppose further that we define the disruption measure of a seam to be the sum of the disruption measures of its pixels.
Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

Chapter 5

Graphs

Graphs are a fundamental mathematical structure with an uncountable number of applications in Computer Science. When you access the Internet and your packets are router to a server, when you use your GPS navigator, and when you use a search engine, there are graphs used in the underlying algorithms. In this course we will review the basic graph notions and algorithms, that compose part of the basic knowledge of any Computer Scientist.

5.1 Definitions

Definition 5.1.1 (Graph)**.**
A graph is a pair of sets $G = (V, E)$ in which V is the set of nodes or vertices, and E the set of edges, also called links, such that $E \subseteq V \times V$.

Definition 5.1.2 (Undirected Graphs)**.**
A graph $G = (V, E)$ is undirected if its edges do not have a direction. As a consequence, an edge $(u, v) \in E$ is the same as (v, u).

Definition 5.1.3 (Directed Graphs)**.**
A graph $G = (V, E)$ is directed if its edges have a direction, i.e. they go from a node to another node. As a consequence, an edge $(u, v) \in E$ is different than (v, u).

Definition 5.1.4 (Degree - undirected graph)**.**
Given an undirected graph $G = (V, E)$ the degree of a node $v \in V$ is the number of edges incident to v, that is:

$$deg(v) = |\{(v, u) s.t. (v, u) \in E\}|$$

Definition 5.1.5 (Degree - directed graph)**.**
Given a directed graph $G = (V, E)$, and a vertex $v \in E$, we define the in-degree $deg^-(v)$ as the number of incoming edges of v, that is:

$$deg^-(v) = |\{(v, u) s.t. (u, v) \in E\}|$$

We define the out-degree of $deg^+(v)$ as the number of outgoing edges from v, that is:

$$deg^+(v) = |\{(v, u) s.t. (v, u) \in E\}|$$

Definition 5.1.6 (Walk)**.**
Given a graph $G = (V, E)$ a walk is a list of vertices $(v_1, \ldots, v_k)$ such that $(v_i, v_{i+1}) \in E$ $\forall i = 1, \ldots, k-1$.

Definition 5.1.7 (Path)**.**
Given a graph $G = (V, E)$ a path is a walk such that each node, except the first and last, are distinct. The first and the last node can instead coincide.

Definition 5.1.8 (Cycle)**.**
A cycle is a path $p = (v_1, \ldots, v_k)$ such that $v_1 = v_k$.

Definition 5.1.9 (Connected Nodes)**.**
Two nodes are connected if there exists a path $(v_1 \ldots v_k)$ such that $u = v_1$ and $v = v_k$.

Definition 5.1.10 (Connected Component)**.**
A connected component is a maximal set $C \subseteq V$ such that, for each $u, v \in C$ there exist a path from u to v.

Definition 5.1.11 (Connected Graph)**.**
A graph is connected if it has a single connected component.

Definition 5.1.12 (Tree)**.**
An undirected graph is a tree if it is both connected, and contains no cycles.

Definition 5.1.13 (Complete graph (Clique))**.**
A graph is complete, or is a clique, if it contains all possible edges between its nodes, that is $\forall u, v \in V \ \exists (u, v) \in E$.

5.2 Representation of Graphs

The mathematical representation of a graph as a pair $G = (V, E)$ is useful for abstract reasoning, for example while designing an algorithm. However, how a graph is represented in the memory of a computer is very different. Several option are possible, which differ in the resulting complexity of common operations. In general, the best data structure should be the one that optimizes the complexity of the most common operations that an algorithm is supposed to perform. In the following, we assume that nodes are numbered from 1 to $|V|$, and $|V| = n$. Figure 5.1 shows an example graph that is used to show the different representations.

5.2.1 Adjacency Lists

The adjacency list consists of a mono-dimensional array *Adj* of size $|V|$. Each element of the array is a pointer to a list of elements. Specifically, for a node v, $Adj[v]$ points to a list that contains an element for each node adjacent to v, i.e. such that $(v, u) \in E$. Figure 5.2 shows the adjacency list for the graph in Figure 5.1.

If the graph is directed, there will be $|E|$ elements in the list. If the graph is undirected, there will be $2|E|$ elements in the list. The memory complexity is therefore $\Theta(|V| + |E|)$.

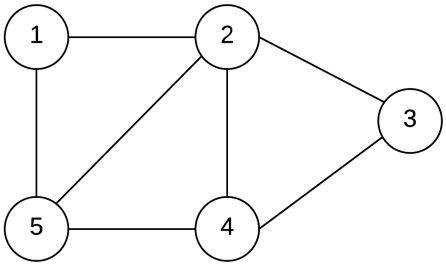


Figure 5.1: Example undirected graph

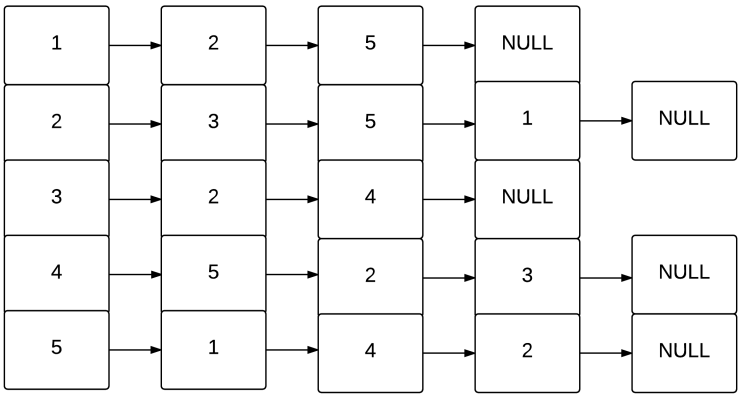


Figure 5.2: Example of an Adjacency List for the above graph

5.2.2 Adjacency Matrix

The adjacency matrix is a matrix $M = |V| \times |V|$ such that:

$$M[i,j] = \begin{cases} 1 & (i,j) \in E \\ 0 & else \end{cases}$$

If the graph is undirected, the resulting associated matrix is symmetric, i.e $M[i,j] = M[j,i]\ \forall i,j = 1,\ldots,n$. If the graph is directed, this is not true in general. The memory complexity of an adjacency matrix is $\Theta(|V|^2)$, since there is no dependence on the number of edges in the graph. Table 5.1 shows the adjacency matrix of the graph in Figure 5.1.

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Table 5.1: Example of an Adjacency Matrix for the above graph

	Adjacency List	Adjacency Matrix
$\exists(u,v) \in E$	$\Theta(deg(v))$	$\Theta(1)$
Degree of a node v	$\Theta(deg(v))$	$\Theta(\|V\|)$

Table 5.2: Complexity of Basic Operations.

5.2.3 Complexity Comparison of Common Operations

We now consider two common operations, specifically (i) to check if an edge exists between (i, j), and (ii) calculate the degree of a node v. Table 5.2 summarizes the discussion.

Existence of an edge (i, j)

Using the adjacency list we need to iterate over the entire list of node i, pointed by $Adj[i]$, since the element j may be the last. This requires a number of iteration proportional to the degree of node i, so the complexity is $\Theta(deg(v))$. Conversely, with the adjacency matrix we should just check the value of $M[i, j]$, which can be done in constant time, that is $\Theta(1)$.

Degree of a node v

Using the adjacency list we need to determine the length of the list if pointed by $Adj[i]$. This can be done in $\Theta(deg(v))$, however if we extend the structure to keep track of the number of elements in each list, it can be reduced to $\Theta(1)$. Differently, with an adjacency matrix we need to iterate over the entire i-th row of the matrix M, so the complexity is $\Theta(n)$

5.3 Depth First Search - DFS

Graph visits are algorithms that explore all nodes of a graph, generally starting from a root node. From such node, these algorithms follow the edges in the graph to reach nodes that have not been visited before, according to a criteria that defines the visit itself. As an example, the Depth First Search (DFS) goes deeper in the graph. A new root node may be selected if not all nodes can be visited from the previous root node. The process ends as soon as all nodes are visited. Visit algorithms generally produce a forest that spans all nodes in the graph.

We can summarize the DFS visit as follows.

- The visit goes deeper and deeper in the graph until it can't find any non-visited nodes then it backtracks to visit other nodes.
- Keep track of the predecessor of each visited node.
- The predecessor sub-graph G_π is a set of trees, called DSF forest.
- For each node u we consider 2 attributes:

- $u.\pi$ - the predecessor of the node u
- u.visited - a boolean variable which is false (F) if the algorithm has not visited u yet, and true (T) otherwise.

5.3.1 Pseudo-code

The DFS visit has two main functions. The first function `DFS()` first initializes the attributes for each node, and then picks a root node that has not been visited and starts the visit from there. The pseudo code is shown in Algorithm 25.

```
1  DFS(G) begin
2  |  for u ∈ V do
3  |  |  u.π =NULL
4  |  |  u.v =false
5  |  end
6  |  for u ∈ V : u.v == false do
7  |  |  //for a connected graph this is called once
8  |  |  DFSVisit(G,u)
9  |  end
10 end
```

Algorithm 25: Depth First Search Pseudo Code

The second function `DFSVisit()` performs the visit starting from the root node. It is a recursive function. It first sets the visit attribute of the current node u to true, then it explores the list of adjacent nodes, and as soon as it finds an adjacent node v that is not visited, it recursively calls itself on v and sets the parent of v as the current node u. The algorithm is shown in Algorithm 26.

```
1  DFSVisit(G,u) begin
2  |  u.v=true
3  |  for v ∈ Adj(u) do
4  |  |  if v.v==false then
5  |  |  |  v.π=u
6  |  |  |  DFSVisit(G,v)
7  |  |  end
8  |  end
9  end
```

Algorithm 26: Depth First Search Visit Pseudo Code

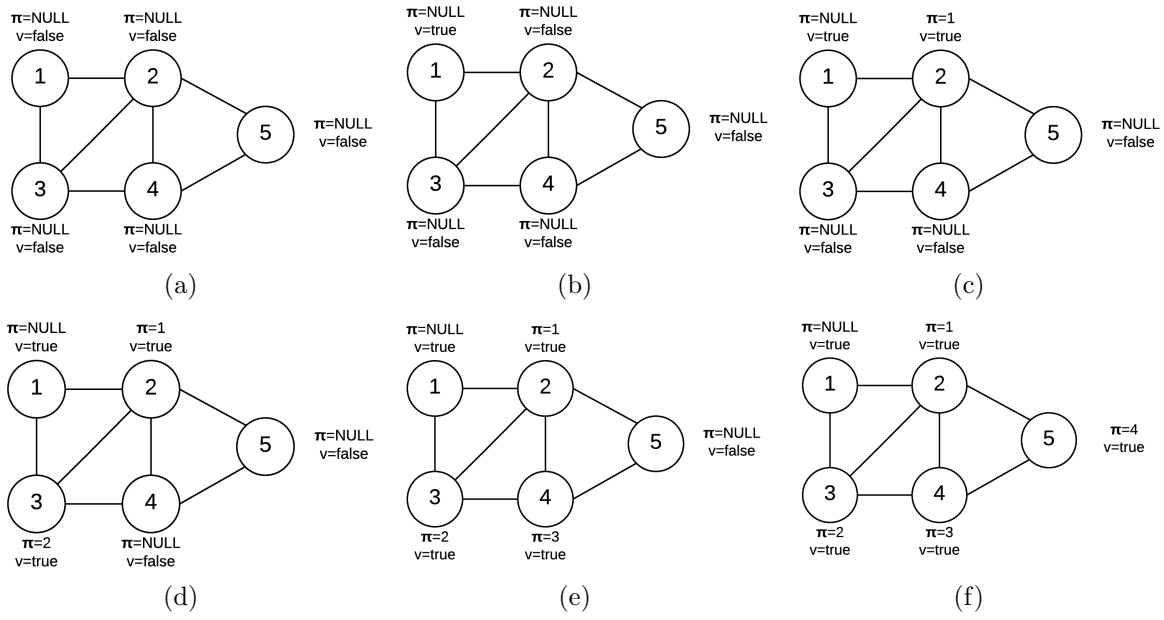


Figure 5.3: Step by step execution of the DFS Algorithm

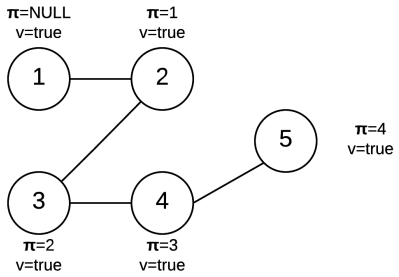


Figure 5.4: Graph pf the G_π Tree

5.3.2 Example

Figure 5.3 shows an example of the execution of the DFS visit. Figure 5.4 shows the resulting DSF tree.

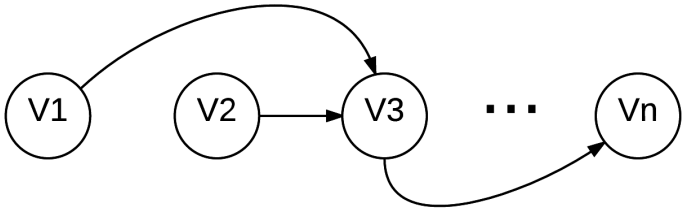
5.4 Directed Acyclic Graphs (DAG) and Topological sort

Graphs and graph algorithms have multiple application in practical settings. In this section we give an example of how we can use a slightly modified DSF visit to determine a scheduling of jobs that have dependencies between each other. Let us consider the following setting.

- Nodes in the graph represent jobs or activities

- There exists an edge from node u to v if v cannot be executed until u has terminated (e.g., job v depends on the output of job u)
- These graphs cannot have cycles, therefore there must exist at least one activity with indegree 0

Our problem consist in finding a scheduling of jobs $v'_1, v'_2, \ldots v'_n$, such that if the jobs are executed in the prescribed order, when an activity is reached, all the activities on which it depends have already been completed. Graphically, this is equivalent to say that if we put the nodes in a linear order following the scheduling, all edges would go from left to right. This order is called *topological sort*. An example is shown in Figure 5.4.



5.4.1 Pseudo-code

In order to find a topological sort, we modify the DFS algorithm. We first add two additional attributes to each node u.

- $u.s$: (start) - the time at which the visit of u begins.
- $u.e$: (end) - the time at which the visit of DFS from u ends.

```
1  TopologicalDFS(G)begin
2  |  t=0
3  |  for u ∈ V do
4  |  |  u.π=NULL
5  |  |  u.v=false
6  |  |  u.start=0
7  |  |  u.end=0
8  |  end
9  |  for u ∈ V : u.v == false do
10 |  |  TDFSVisit(G,u)
11 |  end
12 end
```

Algorithm 27: TopologicalDFS Pseudo Code

```
1  TDFSVisit(G,u)begin
2  |  t=t+1
3  |  u.start=t
4  |  u.v=true
5  |  for v ∈ adj(u) do
6  |  |  if v.v==false then
7  |  |  |  v.π=u
8  |  |  |  TDFSVisit(G,v)
9  |  |  end
10 |  end
11 |  t=t+1
12 |  u.end=t
13 end
```

Algorithm 28: TDFSVisit Pseudo Code

We then keep track of a timer t which is increased every time a visit starts and ends from a node. The topological sort is then obtained by sorting the nodes in descending order by end time. The modified algorithms are shown in Algorithms 28 and 28.

5.4.2 Example

Figure 5.5 shows an example of the execution of the topological sort algorithm. Figure 5.6 shows the resulting topological sort.

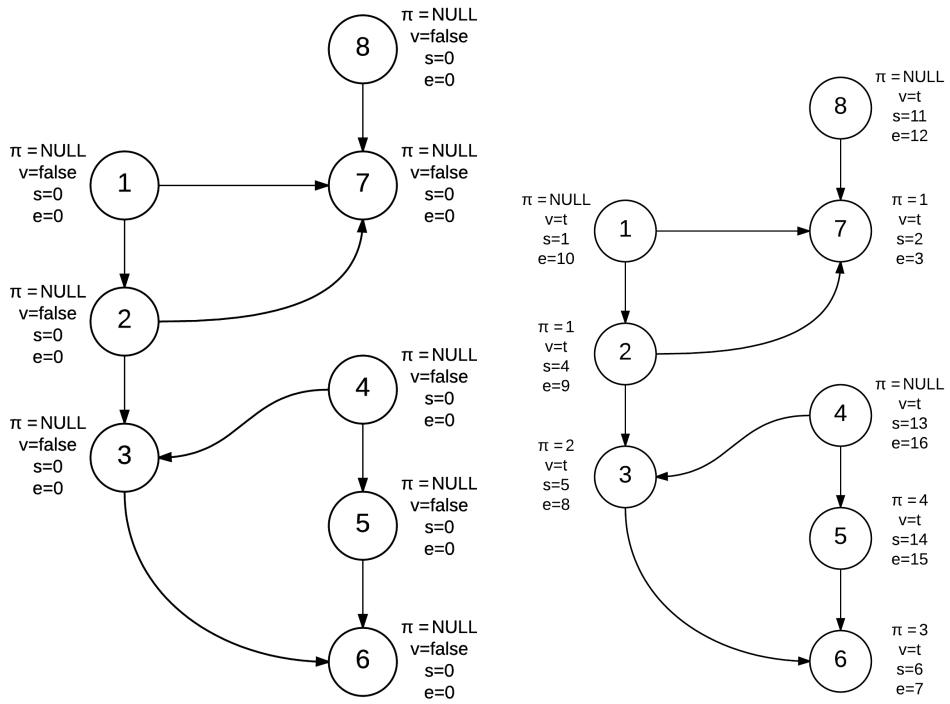


Figure 5.5: Initial and final states for the DAG on which the Topological Sorting algorithm is being executed

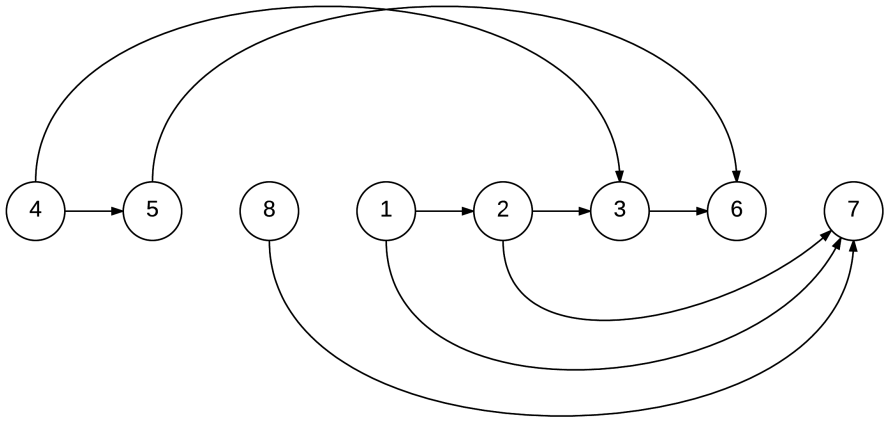


Figure 5.6: Final sorted order of the activities according to the Topological Sorting algorithm's execution

5.5 Breadth First Search (BFS)

Breadth first search is also a visit algorithm, but differently from DFS, it first visits all nodes adjacent to the current node before moving to the next node and visit its neighbors. Given the source node s, the algorithm computes the distance between s and each node in the graph. Additionally, it outputs a BFS-Tree containing all the vertices reachable from s and s.t. the path from s to any node v in the tree is the *shortest path* in the graph G. The algorithm uses the following attributes for every node u in G.

- $u.v$ - a boolean variable which is false (F) if the algorithm has not visited u yet, and true (T) otherwise.
- $u.\pi$ - the predecessor of the node u
- $u.d$ - the distance from the source s to u

```
1  BFS(G,s)begin
2      for u ∈ V \ {s} do
3          u.v=false
4          u.d=∞
5          u.π=NULL
6      end
7      s.v=true
8      s.d=0
9      s.π=NULL
10     Q={}
11     Enqueue(Q,s)
12     while Q ≠ {} do
13         u=Dequeue(Q)
14         for v ∈ adj(u) do
15             if v.v==false then
16                 v.v=true
17                 v.d=u.d+1
18                 v.π=u
19                 Enqueue(Q,v)
20             end
21         end
22     end
23 end
```

Algorithm 29: Breadth First Search Pseudocode

5.5.1 Pseudo-code

The algorithm makes use of a queue Q which is used in First In First Out (FIFO). The queue contains the nodes from which the visit should be extended. The algorithm initially enqueues only the source s, then at each iteration it dequeues a node from Q, visits all its unvisited neighbors and enqueues them in Q. The procedure is repeated until th Q is empty. The Algorithm 29 shows the pseudo-code of the BFS visit.

Complexity Each node may be enqueued only once, therefore the while loop can perform at most $|V|$ iterations. For each node u, we need to scan its list of neighbors, which can be done in $\Theta(deg(u))$ using an adjacency list. Finally, Enqueue and Dequeue operations have constant complexity $\Theta(1)$. As a result, the overall complexity is $\Theta(|V|+\sum_{u\in V} deg(u)) = \Theta(|V|+|E|)$.

5.5.2 Example

Figure 5.7 shows an example of the execution of the BFS algorithm. Figure 5.8 shows the resulting BFS-Tree.

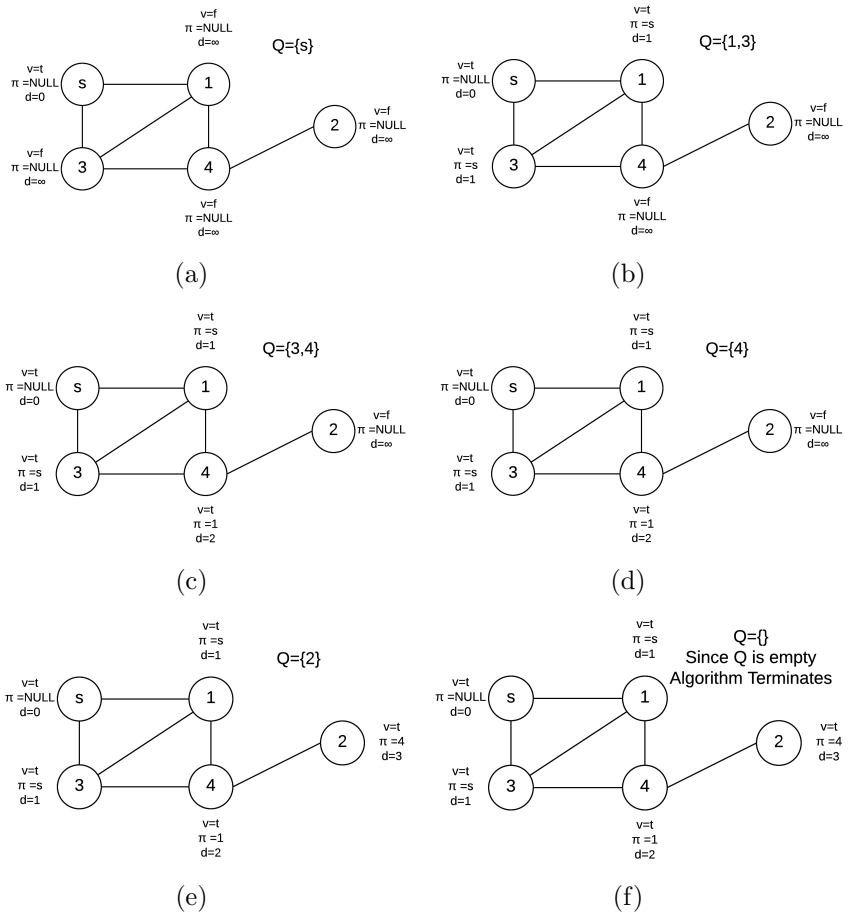


Figure 5.7: Step by step execution of the BFS Algorithm

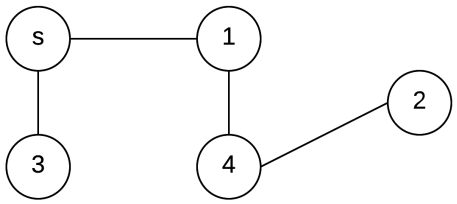


Figure 5.8: BFS Tree following execution of the algorithm

5.6 Strongly Connected Components

In this section we describe an extension of the DFS algorithm that allows to identify the Strongly Connected Components (SCC) of a directed graph.

Definition 5.6.1 (Strongly Connected Component)**.**
Given a directed graph $G = (V, E)$, a strongly connected component $C \subseteq V$ is a maximal set of vertices such that for each pair of nodes $u, v \in V$, there exists a path from u to v and from v to u.

A graph may have multiple SCCs, as shown in Figure 5.9. If a graph G has a single SCC we say that G is strongly connected.

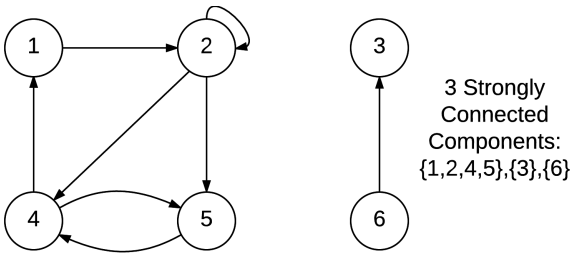
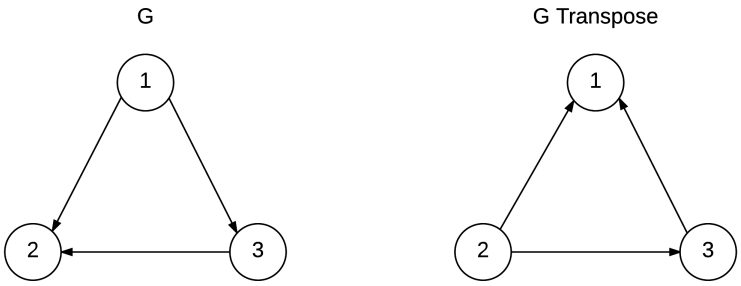


Figure 5.9: Example of a graph and its constituent strongly connected components

The algorithm to identify SCCs makes use of the *transpose* of a graph. Specifically, given a graph $G = (V, E)$ the transpose $G^T = (V, E^T)$ is a graph with the same set of nodes and where $E^T = \{(u, v) : (v, u) \in E\}$. Intuitively, we are inverting the directions of the edges in G. Figure 5.6 shows an example of the transpose of a graph.

The algorithm to find the SCCs of a graph can be summarized by the following steps.

1. Execute the DFS algorithm and compute the end time $u.e$ for each $u \in V$
2. Compute G^T.
3. Execute DFS on G^T where in the main loop we consider vertices in decreasing order of end time.
4. Output the vertices of each tree in the DFS forest calculated in step 3 as a separate SCC.



5.6.1 SCC Example

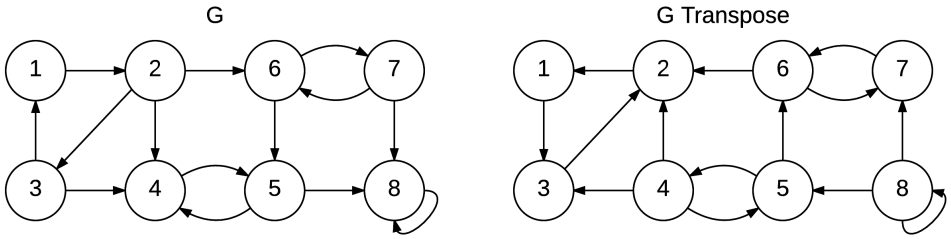


Figure 5.10: Graph and its transpose

1. Execute the DFS algorithm and compute the end time $u.e$ for each $u \in V$

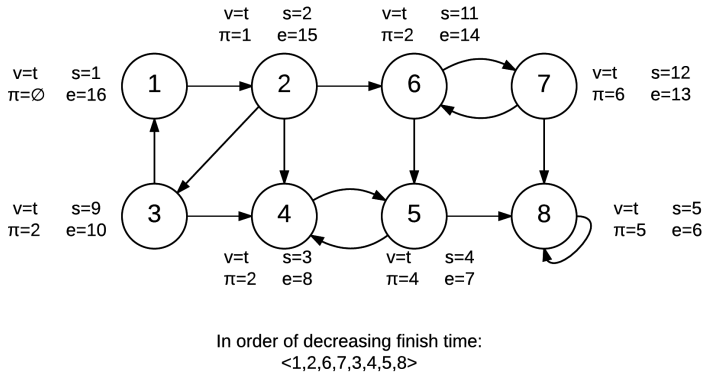


Figure 5.11: Execution of the modified DFS algorithm on G

2. Compute G^T. See Figure 5.10.

3. Execute DFS on G^T such that the nodes are considered in decreasing order of finish time computed in step 1.

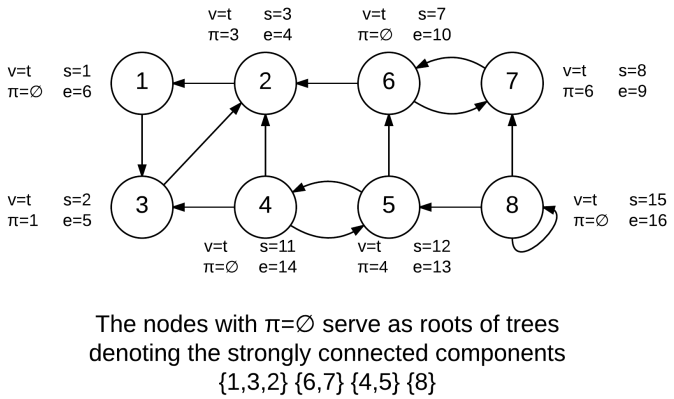


Figure 5.12: Execution of the modified DFS algorithm on G^T

4. Output each tree in the DFS forest calculated in step 3 as a SCC. Each tree in the following forest represents a strongly connected component in the original graph.

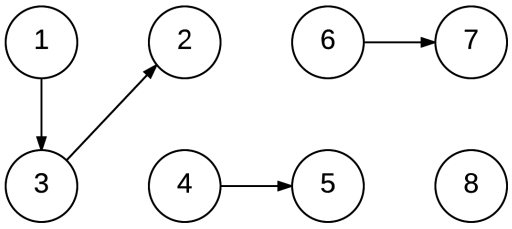


Figure 5.13: DFS forest resulting from the execution of the SCC finding algorithm

There are 4 strongly connected components in the original graph: {1,3,2}, {6,7}, {4,5}, and {8}.

5.7 Red-Black Trees

A **red-black tree** is a special form of a binary search tree with one extra property for node: its *color*, which is either RED or BLACK. In order to make the tree approximately balanced, a red-black tree ensures no path exists (from the root to a leaf) which is more than twice as long as any other. Each node of a red-black tree contains five attributes, namely: *color*, *key*, *left*, *right*, and *p*. The corresponding pointer attribute of a node is *NIL* if its child or the parent does not exist. Now we formally describe the properties of a red-black tree.

5.7.1 Properties of a Red-Black Tree

A legal red-black tree satisfies the following properties:

1. Every node is either red or black.
2. The root is black.
3. All the leafs (often marked as NIL) are black.
4. Both the children of a red node are black.
5. For each node, the number of black nodes in all paths from this node to descendant leaves is same.

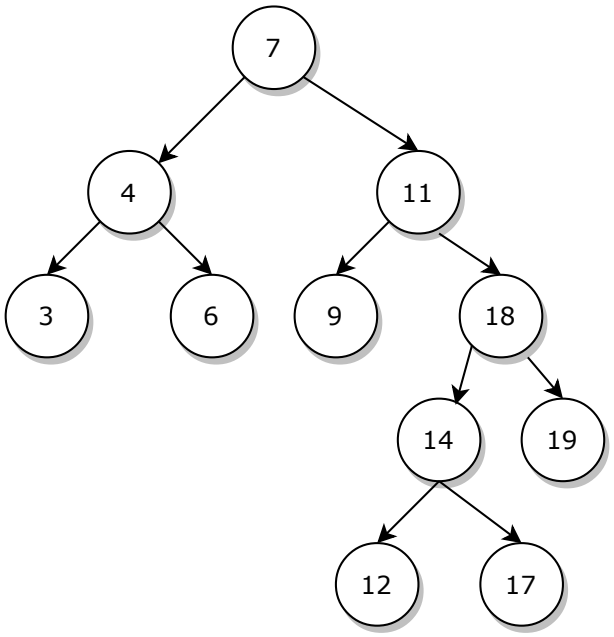


Figure 5.14: A red-black tree (color is not shown in this diagram).

5.7.2 Rotation

Search-tree operations such as INSERT and DELETE (when running with n keys, take $O(lgn)$ time) modify the tree structure. Hence, the resultant tree structure may violate the red-black properties enumerated in the previous subsection. In order to restore these properties, colors of some of the nodes in the tree needed to be changed and also some changes required in the pointer structure. These changes can be made by *rotating* the tree structure, which is a local operation in a search tree that preserves the binary-search-tree property. In this subsection, we describe two kinds of rotations: *left rotations* and *right rotations*.

Pseudo-code for *left rotations*: The pseudo-code for the left rotation is presented in Algorithm 30. Left rotation operation is explained in Figure 5.14 and 5.15. The tree in Figure 5.14 are left rotated with respect to node 18. So, node 18 takes the place of node 11 and left child of 18 become the right child of 11.

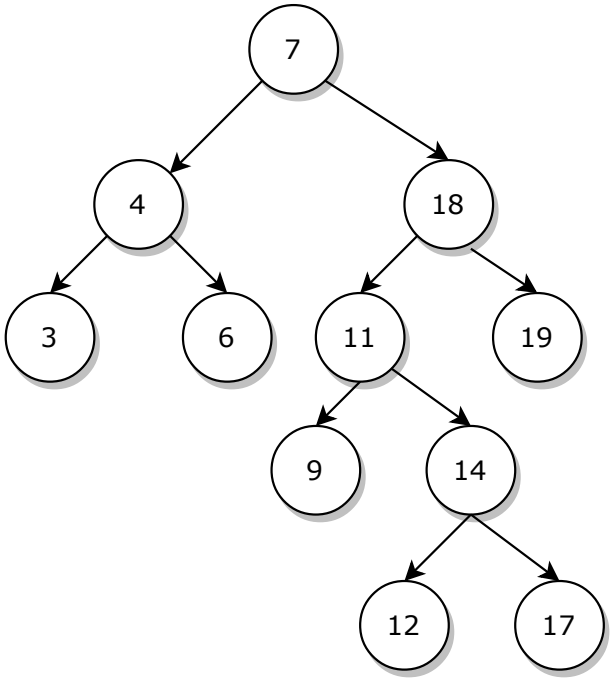
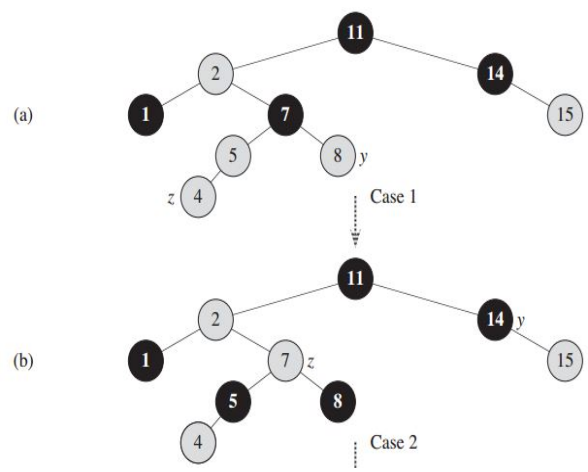
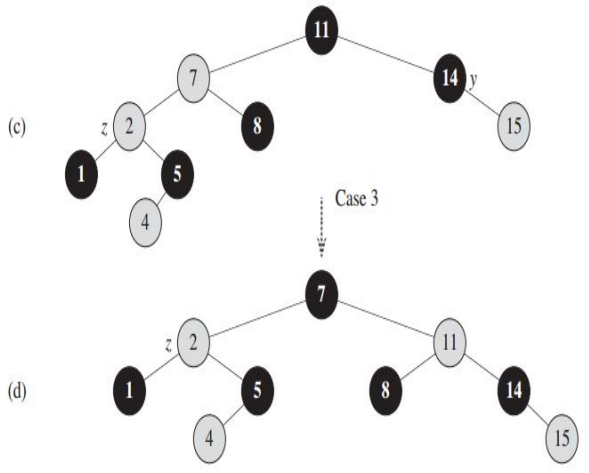


Figure 5.15: A red-black tree after performing *leftrotate*(*root*, 18) in Figure 5.14 (color is not shown in this diagram).

The code for **right rotation** is exactly same as the **left rotation**, except for the pointers which are changed by a rotation (i.e., all the $left$ will become $right$ in Algorithm 30 and vice verca).



(a) Transition between first two cases after inserting a node in a red-black tree.



(b) Transition between last case and the finish state after inserting a node in a red-black tree.

Figure 5.16: INSERT operation in a red-black tree. This figure is adopted from [1]

A node can be inserted into an n-node red-black tree in $O(lgn)$ time. In this subsection, we discuss how to insert a node z into the tree T (then we color it as red). In order to

```
1  leftrotate(node root, node x)begin
2  |  node y = x.right;
3  |  x.right = y.left;
4  |  if y.left ≠ NIL then
5  |  |  y.left.parent = x;
6  |  end
7  |  if x.parent = NIL then
8  |  |  root = y;
9  |  end
10 |  else if x.parent.left = x then
11 |  |  /* x is left child*/
12 |  |  x.parent.left = y;
13 |  end
14 |  else
15 |  |  x.parent.right = y;
16 |  end
17 |  y.left = x;
18 |  y.parent = x.parent;
19 |  x.parent = y;
20 end
```

Algorithm 30: Left rotation in a red-black tree

guarantee that the red-black properties (discussed in this section previously) are not violated, an auxiliary procedure RB-INSERT-FIXUP is invoked which performs the recoloring of the nodes and some required rotations. Pseudo-code for the insert operation presented in Algorithm 31. The operation for inserting a node in a red-black tree is explained in Figure 5.16. In this figure dark nodes are the black nodes. In Figure 5.16 (a), the tree is shown after a node z is inserted. A violation of property 4 occurs as both z and its parent are red. Now, case 1 applies because of the uncle of z (y in this figure) is red. So the nodes are recolored and move the node z up, resulting tree are shown in Figure 5.16(b). Now, z's uncle y is black but z and its parent are both reds. Case-2 applies in this case Since z is the right child of its parent. So a left rotation is performed, and the tree that results is shown in Figure 5.16(c). Now, case 3 applies and z becomes the left child. We perform recoloring and right rotation which yields the legal red-black tree in 5.16(d). A diagram that shows the transition between different cases are shown in Figure 5.17.

5.7.3 Deletion

Deletion of a node from an n-node red-black tree, takes $O(lgn)$ time. Compared to the insertion of a node in a red-black tree, deletion is a bit more complicated. Pseudo-code for deleting a node from the red-black tree is given in Algorithm 35. Similar to the insertion

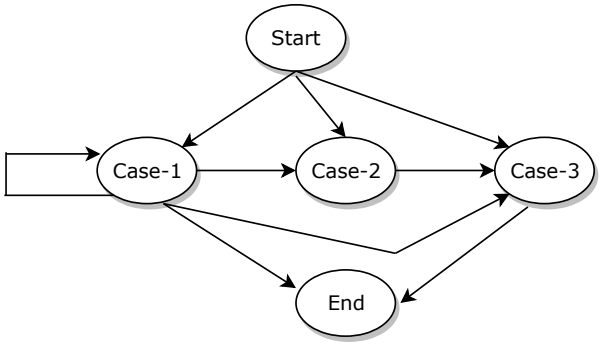
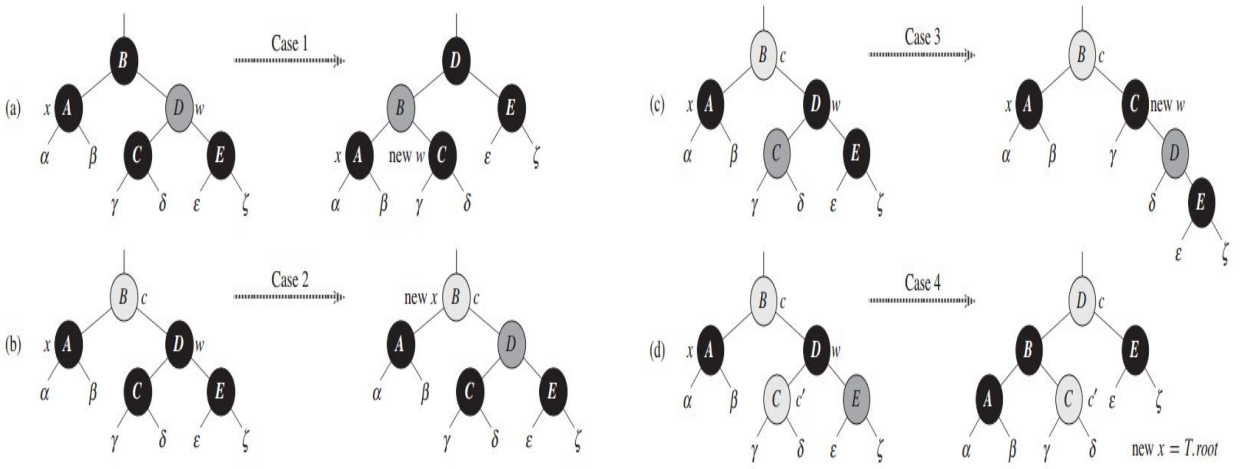


Figure 5.17: Transition between different cases after inserting a node in a red-black tree.



(a) Transition between first two cases after deleting a node from a red-black tree.

(b) Transition between last two cases after deleting a node from a red-black tree.

Figure 5.18: DELETE operation in a red-black tree. This figure is adopted from [1]

operation, deleting a node causes the violation in the red-black tree properties. So, an auxiliary procedure RB-DELETE-FIXUP is invoked which performs the recoloring of the nodes and some required rotations. A diagram that shows the transition between different cases are shown in Figure 5.19.

All the cases in the while loop of the procedure RB-DELETE-FIXUP are explained in Figure 5.18. All the Dark nodes are BLACK in color, shaded nodes have are RED in color. Let us assume that all the lightly shaded nodes have color attributes which are represented by c and c (either RED or BLACK). In Figure 5.18, arbitrary subtrees are represented by $\alpha, \beta, \ldots, \zeta$. In each case, tree on the left side is transformed into the configuration on the right. This is done by recoloring some nodes and/or by performing required rotations (left or right). If any node is pointed by x, then it has an extra BLACK color attribute which is either doubly BLACK or RED-and- BLACK. Note that, repetition of the loop causes only by case 2. In Figure 5.18 (a), by exchanging the colors of nodes (B and D in this figure) and performing a left rotation case 1 is transformed to case 2, 3, or 4. Case 2 is represented in Figure 5.18 (b), where the extra BLACK color is represented by the pointer x. It moves up the tree (by

```
1  insert(node root, int data)begin
2  |  node y = NIL;
3  |  node x = root;
4  |  while x ≠ NIL do
5  |  |  y = x;
6  |  |  if data < x.data then
7  |  |  |  x = x.left;
8  |  |  end
9  |  |  else
10 |  |  |  x = x.right;
11 |  |  end
12 |  end
13 |  if y = NIL then
14 |  |  root = z;
15 |  end
16 |  else if z.data < y.data then
17 |  |  y.left = z;
18 |  end
19 |  else
20 |  |  y.right = z;
21 |  end
22 |  z.parent = y
23 |  fixup(root, z);
24 end
```

Algorithm 31: Insert operation in a red-black tree

coloring node D in RED) and setting x to point to node B. If case 2 is entered through case 1, the termination of the while loop happens because of the new node x, which is red-and-black. It denotes the value c of its color attribute is RED. In Figure 5.18 (c), it is shown how case 3 is transformed to case 4 by exchanging the color of some nodes (C and D in this figure) and performing a right rotation. Figure 5.18 (d), shows how the extra BLACK represented by x removed in case 4 by changing some colors and upon performing a left rotation. Note that, the left rotation is performed in such a way so that the red-black properties are not violated. Finally, the loop terminates after this operation.

5.8 Minimum Spanning Trees

In this section we consider another fundamental problem in graph algorithms, named the *Minimum Spanning Tree* (MST) problem. The goal is to find a tree that spans all nodes in a weighted graph and has minimum weight. We will show that greedy algorithms can solve this problem optimally. First, to motivate the MST problem consider the following application.

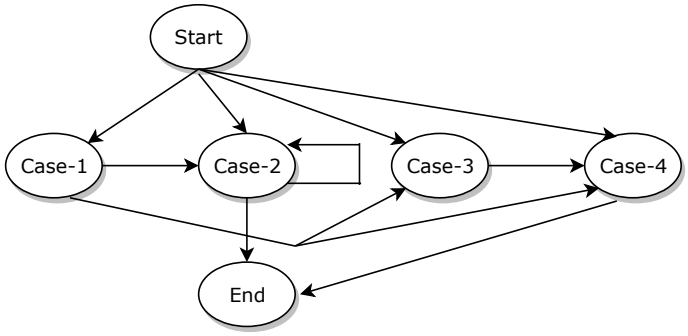
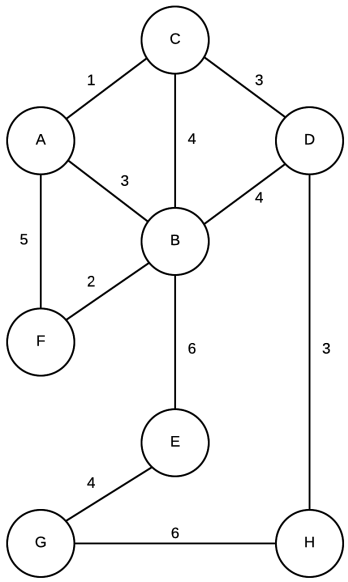


Figure 5.19: Transition between different cases after deleting a node in a red-black tree.

Data center design. We have a data center that contains a set of servers. These servers need to be connected with cables either directly or indirectly i.e. through multiple nodes (called hops in networking jargon). Cables have different costs, as an example as a function of their length. We want to find the set of cables which has minimal cost and allows to connect the set of servers. An example of this problem is shown in Figure **??**, where nodes are servers and the numbers on the edges represent their costs.



As the image suggests, we can describe the problem using a graph $G = (V, E)$ in which each vertex represents a server and the edges are the set of all possible cables which may connect servers. The graph is weighted, specifically edges are weighted, and we represent this with a *weight function* $w : E \to \mathbb{R}^+$, that returns for each edge a weight greater than zero. We can formalize the problem as follows.

Definition 5.8.1 (Minimum Spanning Tree)**.** *Given a connected weighted graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}^+$, find the set of edges $T^* \subseteq E$ such that in the induced*

graph $G^* = (V, T^*)$ $\forall u, v \in V$ $\exists$ *a path from* u *to* v*, and* $\sum_{e \in T^*} w(e)$ *is minimal.* $T*$ *is a Minimum Spanning Tree of* G*.*

In the above definition, the induced graph $G^* = (V, T^*)$ is just a graph with the same nodes as G, but considering only the edges in T^*. Before we proceed, it is useful to reason on the following question. The answer is left to the reader.

Question 1. *Consider a simplified version of the problem, in which all edges have the same weight, i.e.* $w(e) = c$ $\forall e \in E$*. Which of the algorithms we have already studied can be used to solve the problem?*

5.8.1 Kruskal's Algorithm

The Kruskal's algorithm is a greedy algorithm for the MST problem. The idea of the algorithm is the followin.
Algorithm idea Kruskal's Algorithm is greedy algorithm. It iteratively builds a a set S of edges, which is initially empty and is expanded at each iteration. Specifically, at each iteration the algorithm adds to S the edge with minimum cost in $E \setminus S$ that does not create a cycle when added to S.

Pseudo-code and example

The pseudo of Kruskal's Algorithm is shown in Algorithm 36.

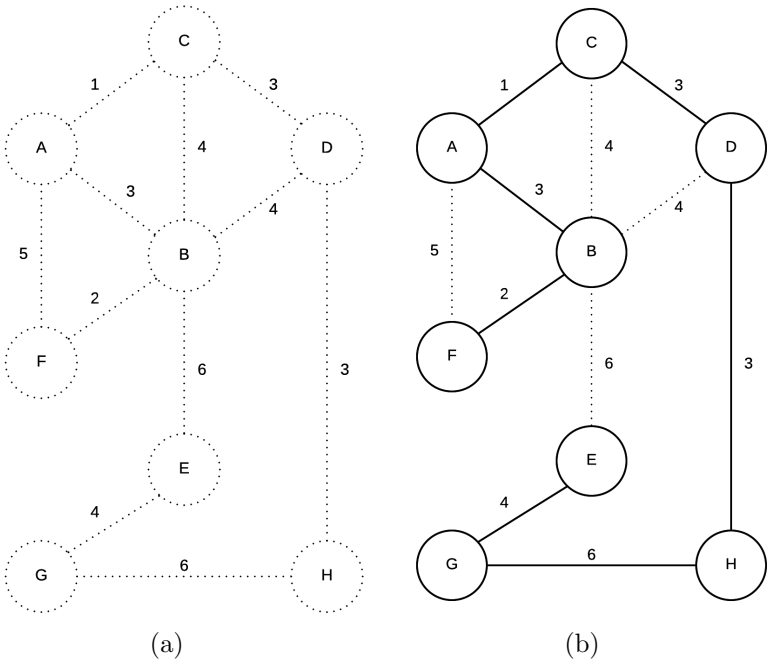


Figure 5.20: An example of Kruskal's Algorithm

Figure 5.20 shows an example of the execution of Kruskal's algorithm. Figure 5.20 (a) represent the initial graph G, while in Figure 5.20 (b) we highlighted in bold the edges picked by the algorithm in the final solution. Such edges are chosen in the following order $S = \{(A, C), (F, B), (A, B), (C, D), (D, H), (G, E), (G, H)\}$.

Correctness of Kruskal's Algorithm

How can we be sure that the Kruskal's Algorithm actually finds an optimal solution, i.e. an MST? We need to prove that this is true in general, that is for any graph, and not for just an example. We prove the algorithm correctness using the standard scheme for greedy algorithms we introduced in Section 3.1.3.

Termination. At each iteration, an edge is added to the solution. The loop terminates when adding any additional edge would create a cycle. Therefore, the final solution is a tree and at most, $|V| - 1$ iterations can be performed.

Optimality of intermediate solutions. Let S_h be the solution at the h^{th} iteration. We want to prove that $\forall h \; \exists$ an optimal solution S^* s.t. $S_h \subseteq S^*$.

- Base case: $S_0 = \emptyset \rightarrow S_0 \subseteq S^*$
- Inductive hypothesis: $\exists S^* : S_h \subseteq S^*$
- Inductive step: We want to prove that given the inductive hypothesis, S_{h+1} is a subset of some optimal solution. Let (u, v) be the edge selected at the $h+1$ iteration. If $(u, v) \in S^*$ then $S_{h+1} \subseteq S^*$ and the we are done. Otherwise, we can agree that $S^* \cup \{(u, v)\}$ definitely contains a cycle, since S^* is a tree. Look now at Figure 5.21, which represents the solution S^* to which we add the edge (u, v). Since the algorithm picked (u, v), then

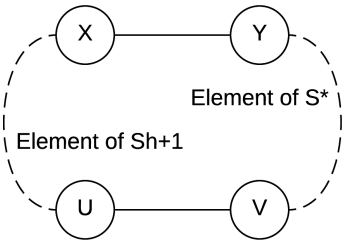


Figure 5.21: Optimal solution S^* to which we add the edge (u, v).

this edge does not create a cycle when added to S_h, however it does create a cycle when added to S^*. Therefore there exists at least one edge (x, y) in S^* which is not in S_h. This edge (x, y) does no create a cycle in S^*, thus it also does not in S_h, since $S_h \subseteq S^*$. Therefore (x, y) is an edge which could have been selected by the algorithm at the $h + 1$ iteration, but the algorithm instead picked (u, v). This implies that $w((u, v)) \leq w((x, y))$. Let's now build another solution $S^{\#} = (S^* \setminus \{(x, y)\}) \cup \{(u, v)\}$,

clearly $S_{h+1} \subseteq S^{\#}$, thus our objective is to show that $S^{\#}$ is optimal. It indeed is, since $S^{\#}$ has $|V| - 1$ edges, and $w(S^{\#}) \leq w(S^*)$ which follows from $w((u, v)) \leq w((x, y))$. Therefore, $S^{\#}$ is optimal and S_{h+1} is a subset of an optimal solution.

The final solution is optimal. We know that the algorithm performs at most $n-1$ iterations, where $n = |V|$. Let S_{n-1} be the final solution, we know that $S_{n-1} \subseteq S^*$, with S^* optimal solution. S_{n-1} is a tree, additionally $w(S_{n-1}) \leq w(S^*)$, thus S_{n-1} is optimal.

5.8.2 Efficient Implementation of Kruskal's Algorithm

Similar to the activity selection algorithm, the high level pseudo-code is useful to reason about the algorithm and prove its correctness. However, it is too abstract to be of practical use, that is to translate it in an actual implementation. Two main problem arise: (i) pick the edge with minimum weight may be more or less efficient depending on the implementation, (ii) it is not straight forward to check if an edge does not create a cycle. In the following we discuss an efficient implementation of the algorithm.

We use an array E to store the information about the edges, where E[i].u and E[i].v are the endpoints of the edge i, and E[i].w is the weight. This array is sorted in ascending order by weight. This will allow us to pick efficiently the next edge to consider. To solve the second problem, i.e. check if an edge creates a cycle, the key point is to realize that an intermediate solution S identifies a set of connected components. As more edges are added, these components merge and eventually become a single component (the MST) at the end of the algorithm. The idea of the efficient implementation is to keep track for each node of the component of that node. When we pick an edge, this creates a cycle in the current solution if it connects nodes from the same component, while it does not create a cycle if it connects nodes in different components. We use an array CC[i. . . n], where CC[i] is the ID of the connected component of node i. Initially, $CC[i] = i \ \forall i \in [1 \dots n]$. When a new edge is added and two components merge, the ID of one component overwrites the ID of the other.

The efficient implementation allows us to study the complexity. Sorting the edges costs $\Theta(m \log m)$. The external for loop is executed m times, however the internal for loop is executed only at most $n-1$ times, that is once for every edge added in the solution. Therefore, the complexity of these loops is $\Theta(m + n^2)$. The overall complexity is $\Theta(m \log m + n^2)$.

5.8.3 Prim's Algorithm

Prim's algorithm is also a greedy approach for the MST problem. It follows a different approach compared to Kruskal's algorithm, as summarized by the following algorithm idea.

Algorithm idea. Prim's algorithm is an iterative greedy algorithm to find the MST of a graph. It starts from a node and at each iteration it extends the tree rooted at that node by adding the edge with minimum weight that connects a node in the tree to a node not in the tree. The algorithm terminates when the tree includes all nodes in the graph.

Pseudo-code

The algorithm iteratively builds a solution S, which contains the edges of the resulting MST. The algorithm also keeps track of the nodes currently in the tree using the set C that initially contains only the root r. The pseudo-code of the algorithm is shown in Algorithm 11.

```
1  fixup(node root, node z)begin
2      while z.parent and z.parent.color == RED do
3          if z.parent = z.parent.parent.left then
4              y = z.parent.parent.right
5              if y.color == RED then
6                  /*CASE-1*/
7                  z.parent.color = BLACK
8                  y.color = BLACK
9                  z.parent.parent.color = RED
10                 z = z.parent.parent
11             end
12             else
13                 if z.parent.right = z then
14                     /*case 2*/
15                     z = z.parent
16                     leftrotate(root, z);
17                 end
18                 /*case 3*/
19                 z.parent.parent.color = RED
20                 z.parent.color = BLACK
21                 rightrotate(root, z.parent.parent);
22             end
23         end
24         else
25             Symmetric w.r.t the if condition above with right and left exchanged;
26         end
27     end
28     root.color = BLACK;
29 end
```

Algorithm 32: Fixup operations to restore the red-black tree properties after inserting a node into it.

```
1  Delete(T, node z)begin
2      node x, y = z;
3      y.originalColor = y.color;
4      if z.left = T.NIL then
5          x = z.right;
6          RB-TRANSPLANT(T, z, z.right);
7      end
8      else if z.right = T.NIL then
9          x = z.left;
10         RB-TRANSPLANT(T, z, z.left);
11     end
12     else
13         y. = TREE-MIN(z.right);
14         y.originalColor = y.color;
15         x = y.right;
16         if y.parents = z then
17             x.parents = y;
18         end
19         else
20             RB-TRANSPLANT(T, y, y.right);
21             y.right=z.right;
22             y.right.parents=y;
23         end
24         RB-TRANSPLANT(T, z, y);
25         y.left=z.left;
26         y.left.parents=y;
27         y.color=z.color;
28     end
29     if y.originalColor = BLACK then
30         RB-DELETE-FIXUP(T, x)
31     end
32 end
```

Algorithm 33: Delete operation in a red-black tree

```
1  RB-TRANSPLANT(T, node u, v)begin
2  |  if u.parents = T.NIL then
3  |  |  T.root=v;
4  |  end
5  |  else if u = u.parents.left then
6  |  |  u.parents.left= v;
7  |  end
8  |  else
9  |  |  u.parents.right= v;
10 |  end
11 |  v.parents= u.parents;
12 end
```

Algorithm 34: TRANSPLANT operation in a red-black tree

```
1  RB-DELETE-FIXUP(T, node x)begin
2      while x.color=BLACK and x ≠ T.root do
3          if x = x.parents.left then
4              node w = x.parents.right;
5              if w.color=RED then
6                  /* CASE - 1*/
7                  w.color=BLACK;
8                  x.parents.color=RED;
9                  LEFT-ROTATE(T, x.parents);
10                 w = x.parents.right;
11             end
12             if w.left.color= w.right.color=BALCK then
13                 /* CASE - 2*/
14                 w.color=RED;
15                 x = x.parents;
16             end
17             else
18                 if w.right.color=BALCK then
19                     /* CASE - 3*/
20                     w.left.color=BALCK
21                     w.color=RED;
22                     RIGHT-ROTATE(T, w);
23                     w = x.parents.right;
24                 end
25                 /* CASE - 4*/
26                 w.color=x.parents.color;
27                 x.parents.color=BLACK;
28                 w.right.color=BALCK;
29                 LEFT-ROTATE(T, x.parents);
30                 x = T.root;
31             end
32         end
33         else
34             Symmetric w.r.t the if condition above with right and left exchanged;
35         end
36     end
37     x.color=BLACK;
38 end
```

Algorithm 35: Fixup operations to restore the red-black tree properties after deleting a node from it.

```
1  Kruskal(G,w)begin
2  |  S = ∅;
3  |  while ∃(u, v) ∈ E \ S s.t. S ∪ {(u, v)} does not contain a cycle do
4  |  |  let D ⊆ E \ S be the set of edges that do not create a cycle when added to S;
5  |  |  (u, v) = arg min_{(p,q)∈D} w(p, q);
6  |  |  S = S ∪ {(u, v)};
7  |  end
8  |  return S;
9  end
```

Algorithm 36: Kruskal's Algorithm

```
1  Kruskal(G,w)begin
2  |  S=∅
3  |  m=|E|
4  |  Sort E in ascending order by weight
5  |  for i=1 to n do
6  |  |  CC[i]=i
7  |  end
8  |  for j=1 to m do
9  |  |  (u.v)=(E[j].u,E[j].v)
10 |  |  // The edge does not create a cycle
11 |  |  if CC[u]≠CC[v] then
12 |  |  |  S=S∪{(u,v)}
13 |  |  |  cID=CC[v]
14 |  |  |  // Update component IDs of the nodes in component CC[v]
15 |  |  |  for p∈V do
16 |  |  |  |  if CC[p]==cID then
17 |  |  |  |  |  CC[p]=CC[u]
18 |  |  |  |  end
19 |  |  |  end
20 |  |  end
21 |  end
22 |  return S
23 end
```

```
1  Prim(G,w)begin
2  |  S=∅
3  |  Select a node r ∈ V as the root
4  |  C={r}//nodes currently in the tree
5  |  while C≠V do
6  |  |  (u, v) = arg min  w(x, y)
   |  |                (x,y)∈E:
   |  |                x∈C∧y∉C
7  |  |  S=S∪{(u,v)}
8  |  |  C=C∪{v}
9  |  end
10 |  return S
11 end
```

It is suggested to execute Kruskal's and Prim's algorithm on the same graph to understand the differences in their logical execution.

Efficient Implementation of Prim's Algorithm

The main problem to address to have an efficient implementation of Prim's algorithm is how to select the edge with minimum weight that connects a node in the tree to a node not in the tree. Here we propose an efficient implementation that makes use of a queue Q. Q contains the nodes not yet in the tree (unvisited nodes). We also make use of two attributes for a node v, $v.\pi$ that represents the parent of v in the tree, and $v.d$ that represents the distance of v from the current tree. Intuitively, we will look at each iteration for the node v in Q with minimum distance. The efficient implementation is shown in Algorithm 37.

```
1  Prim(G,W,r)begin
2      for v ∈ V do
3          v.π=NULL
4          v.d=∞
5      end
6      r.d=0
7      Q=V //set of unvisited nodes
8      while Q≠ ∅ do
9          v=Q.ExtractMin()
10         for u∈ adj(v) do
11             if u ∈ Q ∧ u.d > (v, u) then
12                 u.π=v
13                 u.d=w(v,u)
14             end
15         end
16     end
17 end
```

Algorithm 37: Pseudo Code for an efficient implementation of Prim's Algorithm

The complexity of the algorithm is dominated by the while loop. This loop clearly does $O(n)$ iterations. In a straightforward implementation, the function `Q.ExtractMin()` takes $O(n)$. To better define the complexity of the internal for loop, we should do an *aggregate analysis*. For each node u, we iterate over its set of neighbors, so the complexity is $O(deg(u))$. Considering all nodes, the overall complexity of the for loop is the sum of the degrees, i.e. $O(sum_{u \in V} deg(u) = O(|E|) = O(m)$. As a result, the complexity of the while loop, and of the algorithm, is $O(n^2 + m) = O(n^2)$.

Example

We now show the execution of the efficient implementation on the graph in Figure 5.20 (a). The execution is in Table 5.3. Node A is selected as the root node. Each row of the table is an iteration and shows the distances of each node. Additionally, we bar the value if the element is visited.

	Q=	A	B	C	D	E	F	G	H
	init	0	∞	∞	∞	∞	∞	∞	∞
1	Select A	~~0~~	3	1	∞	∞	5	∞	∞
2	Select C	~~0~~	3	~~1~~	3	∞	5	∞	∞
3	Select B	~~0~~	~~3~~	~~1~~	3	6	2	∞	∞
4	Select F	~~0~~	~~3~~	~~1~~	3	6	~~2~~	∞	∞
5	Select D	~~0~~	~~3~~	~~1~~	~~3~~	6	~~2~~	∞	3
6	Select H	~~0~~	~~3~~	~~1~~	~~3~~	6	~~2~~	6	~~3~~
7	Select G	~~0~~	~~3~~	~~1~~	~~3~~	4	~~2~~	~~6~~	~~3~~
8	Select E	~~0~~	~~3~~	~~1~~	~~3~~	~~4~~	~~2~~	~~6~~	~~3~~

Table 5.3: Iterative Execution of Prim's Algorithm given in a table

Note that in step 7 the algorithm had a choice, since two edges were available of equal weight. In this case, one was chosen arbitrarily, but if the other edge was selected, it too would have yielded a different, but still optimal solution. The resulting graph of the minimum spanning tree, which is the same as the one yielded for this graph when Kruskal's algorithm was applied, is given in Figure 5.22.

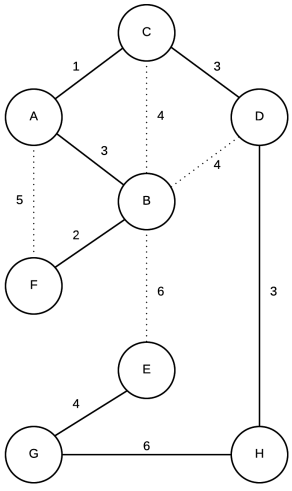


Figure 5.22: Graph of the Minimum Spanning Tree generated from Prim's Algorithm

5.9 Single Source Shortest Paths

We now consider another classical problem of graph theory, finding the shortest path from a node to all other nodes in the graph. This emerges in many application, including the one in the following.

Internet routing. The Internet is composed by a network of routers interconnected through fiber links. Packets sent from one router need to reach another router generally through multiple hops. The process of determining which path to follow is called routing. Links are not all equal, they generally have different delays depending on their capacity, on the traffic that goes through them and on the processing power of the routers at the endpoints of that link. The routing problem consist in finding the path with minimum delay (minimum sum of delays of the links in the path) from the source to the destination.

We can model the problem using a weighted graph $G = (V, E)$ such that V represents the set of routers and E the set of links between them. We introduce a weight function on the edges to model the delay, $w : E \to \mathbb{R}^+$, where $w(u, v)$ is the delay in the link $(u, v) \in E$. Given a path $p = (v_1, v_2, \ldots, v_k)$ the weight of p is $w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$. The shortest path weight between two nodes u and v is defined as $\delta(u, v)$ defined as follows.

$$\delta(u,v) = \begin{cases} \arg\min_{p:u\to v} w(p) & \text{if u and v are connected} \\ \infty & otherwise \end{cases}$$

A path p between u and v is a shortest path if $w(p) = \delta(u, v)$.

Definition 5.9.1 (Single source shortest path problem)**.** *Given a weighted graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$ and a source $s \in V$, find the shortest path from s to all other nodes in V, i.e. $\forall v \in V$ find p_v^* s.t. $w(p_v^*) = \delta(s, v)$.*

5.9.1 Optimality Principle

Consider a graph $G = (V, E)$ and a source s, assume that you calculated the shortest paths from s to all the other nodes. Let's now consider the graph that contains all nodes in V but only the edges that appear in at least one shortest path. How does this graph look like? The optimality principle helps answering this question, and gives a powerful tool to design efficient algorithms to solve the shortest path problem.

Theorem 3 (Optimality principle)**.** *Given a weighted graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, two nodes u and v and the shortest path p between them, if a node q is in p, then the path p' between u and q is the shortest path for these two nodes.*

Proof. We proceed by contradiction. Assume that p' is **not** the shortest path between nodes u and q, this implies that there exists another shorter path p'' such that $w(p'') < w(p')$. Let p''' be the path from q to v, the weight of the shortest path p between u and v is $w(p) =$

$(w(p') + w(p'''))$. However, we can build another path between u and v by concatenating p'' and p''', the weight of such path is less than $w(p)$ since $w(p'') < w(p')$. Formally, $w(p) = (w(p') + w(p''')) > w(p'') + w(p''')$. This leads to a contradiction because p is assumed to be the shortest path from u to v. □

Corollary 5.9.1 (Shortest path tree)**.** *The union of all shortest path from a source to all destination forms a tree, which is called the* Shortest path tree*.*

5.9.2 Dijkstra's Algorithm

The Dijkstra's algorithm is a greedy algorithm for the single source shortest path problem. It exploit the optimality principle as summarized by the following algorithm idea.

Algorihtm idea. The algorithm iteratively builds the shortest path tree from a source node s, which is initially composed by the sole node s. Let R be the set of nodes in the tree at the current iteration. The algorithm picks the edge that minimizes the distance from s to reach a node not in the tree from any other node in the tree. Formally, it picks edge (u, v) such that $u \in R$, $v \notin R$ and minimizes $u.d + w(u, v)$, where $u.d$ is the shortest path distance of u from s.

Peudo-code

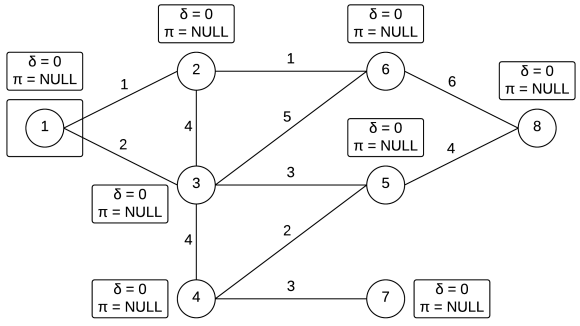
We will use the following attributes for a node u.

- $u.d$: distance from the source in the shortest paths tree, for the source $s.d = 0$.
- $u.d = \delta(s, u)$: weight of the shortest path from s to u.
- $u.\pi$: parent of u in the shortest paths tree.

Example

Figure 5.24 shows the graph considered in the example. Figure 5.24 shows the execution of the Dijkstra's algorithms. The source node is node 1, we highlight in bold the edges selected by the algorithm.

```
1  DIJKSTRA(G,w,s)begin
2  |  s.d=0
3  |  s.π=NULL
4  |  // R is the set of nodes currently in the tree
5  |  R={s}
6  |  while ∃(u,v) ∈ E : u ∈ R ∧ v ∋ R do
7  |  |  (u,v) = arg min (x.d + w(x,y))
   |  |            (x,y)∈E:
   |  |              x∈R
   |  |              y∋R
8  |  |  v.d=u.d+w(u,v)
9  |  |  v.π=u
10 |  |  R=R∪{v}
11 |  end
12 end
```



(a)

Figure 5.23: Example Dijkstra's algorithms: initial graph

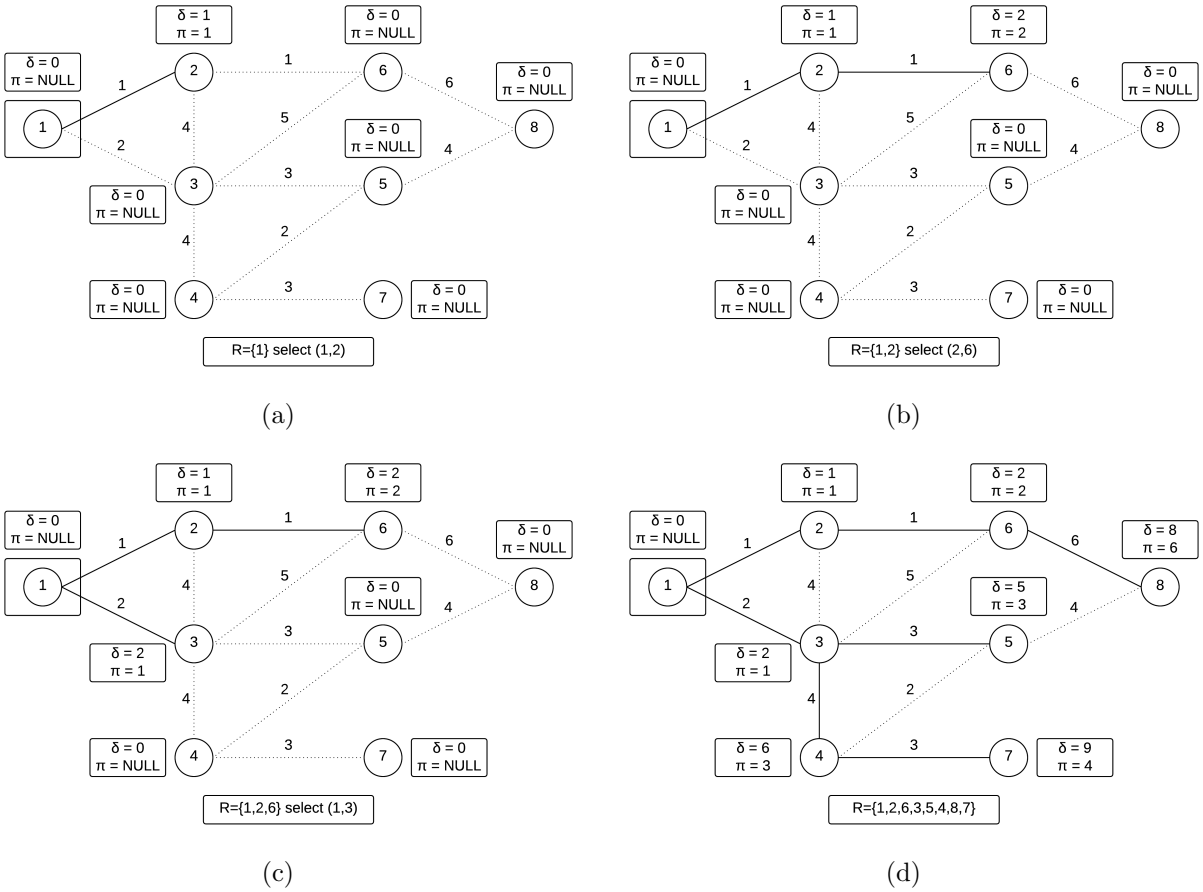


Figure 5.24: Example Dijkstra's algorithms: algorithm execution

Efficient Implementation

In the efficient implementation of Dijkstra's algorithm we make use of some additional variable and data structures.

- Boolean array $v[1...n]$: $visited[i] = T$ if node i is in the tree, F otherwise.
- Integer k: number of visited nodes.
- Array $\pi[1...n]$: parent vector.
- Integer array $dist[1..n]$:

$$dist[i] = \begin{cases} 0 & i = source \\ \delta(s,v) & visited[i] \\ \text{current shortest path distance from s} & otherwise \end{cases}$$

The key point to understand here is the role of the array dist[]. The position dist[i] contains the shortest path distance from s only if i is already in the tree, that is the algorithm has already taken its decision on that node regarding the shortest path from s to i. On the contrary, if i is not in the tree, dist[i] contains the best known distance up to now. This is updated as more nodes are added to the tree. Basically, we keep track of the distance from s of all nodes which are one hop away from any node already in the tree, and include in the tree at each iteration the node which is at minimum distance. Similarly, $\pi[i]$ is the actual parent of i if i is part of the tree, otherwise is the potential father, i.e. the best node through which I can reach i considering the nodes currently in the tree. Potential fathers, as the best known distances, can be updated as we add more nodes into the tree.

```
1  DIJKSTRA(G,w,s)begin
2      dist[s]=0
3      π[s]=NULL
4      v[s]=T
5      k=1
6      for q ∈ V \ {s} do
7          v[q]=False
8          if q ∈ Adj(s) then
9              dist[q]=w(s,q)
10             π[q]=s
11         end
12         else
13             dist[q]=∞
14             π[q]=NULL
15         end
16     end
17     // Find the best node to add
18     while k<n do
19         minD=∞
20         minV=NULL
21         for i=1 to n do
22             if v[i]==F∧dist[i]<minD then
23                 minD=dist[i]
24                 minV=i
25             end
26         end
27         v[minV]=T
28         k+=1
29         // Update the distances for the unvisited nodes adjacent to the
               newly added node
30         for q∈ Adj(minV) do
31             if !v[q]∧dist[q]>dist[minV]+w(minV,q) then
32                 dist[q]=dist[minV]+w9minV,q)
33                 π[q]=minV
34             end
35         end
36     end
37     return dist,π
38 end
```

The while and for loop perform each n iterations. The for loop over the set of adjacent nodes of the newly added node $minV$ performs a number of iterations equal to $\Theta(Adj(minV))$. We can perform again an aggregate analysis and conclude that overall this loop has a complexity of $\Theta(m)$. As a result the complexity is $\Theta(n^2 + m) = \Theta(n^2)$ since $m = O(n^2)$.

5.9.3 Bellman-Ford: Shortest Path with Negative Weights

The Dijkstra's algorithm provides an efficient way of calculating shortest path from a single source, however this algorithm works only if the edge weights are positive. Edges with negative weight may happen, for example, in applications in which going from a node to another may provide a gain, instead of a cost. In these cases, the Dijkstra's algorithm may not work, as shown by the example in Figure 5.25.

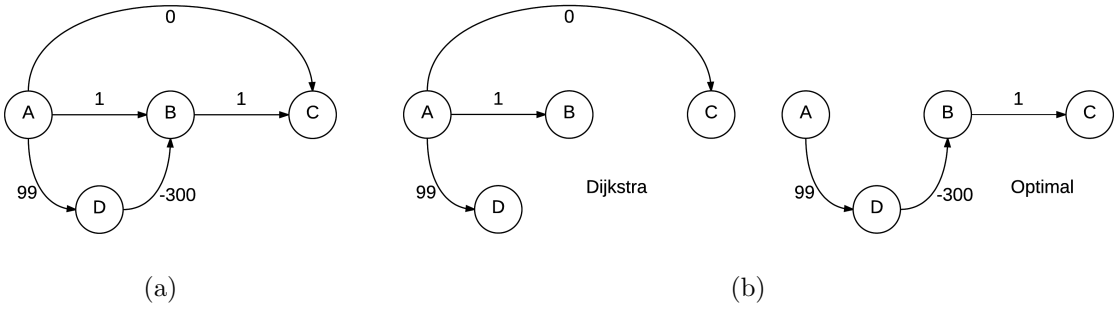


Figure 5.25: Graph with negative weights (a), output of Dijkstra (b), optimal shortest path tree (c).

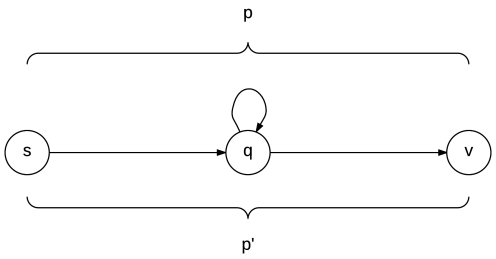
The Dijkstra's algorithm fails since, as a good greedy algorithm, is based on the idea that what is the best solution now, it is actually the best solution that will eventually bring to the optimum. It has, in other words, a stubborn attitude an never changes its mind. In addition, it also has a limited horizon, since it only considers the nodes currently reachable by the nodes in the tree. Being stubborn and having a limited horizon rarely brings you somewhere, and in this example it does not allow the algorithm to see that node B can be reached much more efficiently through D than through A, but it is stubborn and does not change its mind.

The problem of finding the single source shortest path needs however to be better defined to make sense. Specifically, the problem makes sense only if the graph has no negative cycles. A path $p = (v_1, \ldots, v_k)$ is a negative cycle if $v_1 = v_k$ and $\sum_{i=1}^{k-1} w(v_i, v_{i+1}) < 0$. Intuitively, if we have a negative cycle we can arbitrarily decrease the distance between two nodes by going through the cycle as many times as we want. Therefore, in this cases the distance between two nodes is not defined. We must consider only graphs with no negative cycles for the problem to be well defined.

As a consequence of the above discussion, the graph also has to be directed. Indeed, undirected edges with negative weights can be seen as negative cycles of length 1, and would make our problem not well defined. In this context, the following theorem is at the basis of the Bellman-Ford algorithm.

Theorem 4. *Consider a directed graph $G = (V, E)$ with positive and negative weights, but no negative cycles, and a source s. If a node v is reachable from s then the length (nr. of hops) of the path is less than n.*

Proof. We prove the theorem by contradiction. Assume that there is a shortest path p with length $\geq$ n between s and a node v. This implies that there must exist a cycle in p. The cycle cannot have a positive weight, since this would imply that p is not the shortest path. Therefore the cycle must have a negative weight, which violates the assumptions of the theorem. Figure 5.9.3 summarizes the proof. □



Bellman-Ford algorithm

The Bellman-Ford algorithm is based on dynamic programming, which allows more flexibility with respect to greedy solutions. As usual with a dynamic programming algorithms, we first define the sub-problems, and then the recursive relation between their solution. We will also make use of a table M to keep track of the problems already solved.

The sub-problems are defined by limiting the maximum length of shortest paths. We know by Theorem 4 that a path cannot be longer than $n - 1$. The algorithm considers the sub-problems of finding the shortest path from s to all other nodes with length at most k. The value of k is increased from 0 (no edge in the path) to $n - 1$, the maximum length.

We define the table $M : (n+1) \times n$. M[k,v] is the weight of the shortest path of length at most k between the source s and v. If such path does not exist, then M[k,v] equals infinity. We can now define the recursive relation between problems.
Base case. The base case occurs when $k = 0$, i.e. no edge in the path. This can be trivially solved by setting $M[0, s] = 0$, and $M[0, v] = \infty$ for any other node $v \neq s$.
Inductive case. Assume that we know the shortest path from s to v with length at most k-1, and we want to calculate M[k,v]. We have two cases. First case, the shortest path of at

most length k is the same, that is M[k,v] = M[k-1,v]. This happens when adding an extra hop does not improve the length of the shortest path, so the previous one we found is still good. The second case occurs when there exists a path p of length k with less weight between s and v. Such path p is composed by a path p_u from s to a node u, plus an edge (u, v). The path p_u has length $k-1$, thus its weight is M[k-1,u], and it has been already calculated. We summarize the inductive case with the equation below.

$$M[k, v] = \min\{M[k-1, v], \min\{M[k-1, u] + w(u, v) \text{ s.t. } (u, v) \in E\}\}$$

Note that we can use the above formula for $k \geq n$, this is useful to detect negative cycles. Specifically, it is possible to prove that if and only if the graph has a negative cycle, there exists v s.t. $M[n, v] \neq M[n-1, v]$.

Pseudo-code

```
1  BellmanFord(G,w,s)begin
2      M: table of size (n + 1) × n
3      π[n]=new parent vector
4      ∀v ∈ V, M[0, v] = ∞
5      M[0,s]=0
6      π[s]=NULL
7      for k=1 to n do
8          for v in V do
9              M[k,v]=M[k-1,v]
10             for (u,v)∈E do
11                 if M[k-1,v]+w(u,v)<M[k,v] then
12                     M[k,v]=M[k-1,v]+w(u,v)
13                     π[v]=u
14                 end
15             end
16             if (k==n∧M[k,v]≠M[k-1,v] then
17                 return "G has a negative cycle"
18             end
19         end
20     end
21     return M, π
22 end
```

Algorithm 38: Bellman-Ford algorithm

The algorithm is shown in Algorithm 38. It initially applies the base case, it then performs two nested loops to apply the recursive formula. The vector π contains the parents of the nodes in the shortest path tree.

The algorithm has two nested for loops, each of which performs n iterations. The complexity of the internal loop that iterates on the edges can be again calculated using the aggregate analysis. The complexity is therefore $\Theta(n(n+m)) = \Theta(n^2)$.

Examples

Table 5.4 shows the execution of the Bellman-Ford algorithm on the graph in Figure 5.25 (a), for which Dijkstra's algorithm yielded a non-optimal solution.

	A	B	C	D
0	0	∞	∞	∞
1	0	1	0	99
2	0	-201	0	99
3	0	-201	-200	99
4	0	-201	-200	99

Table 5.4: Example of Bellman-Ford on the graph in Figure 5.25 (a).

Consider now the simple graph in Figure 22 with a negative cycle. Table 22 shows the execution of the algorithm. The last line of the table shows a change, as expected, which is due to the presence of the negative cycle.

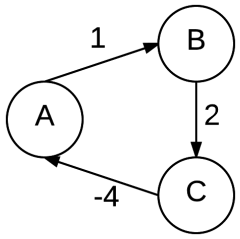


Figure 5.26: Example of Bellman-Ford on a graph with negative cycle.

Figure 22 shows a more complex graph, while Table 22 the execution of the algorithm. Note starred rows in the preceding example. These rows show no changes, this indicates that paths with length 3 are better than any path of length 4. In general, when two consecutive

	A	B	C
0	0	∞	∞
1	0	1	∞
2	0	1	3
3	**-1**		

Table 5.5: Example of Bellman-Ford on the graph in Figure 22.

rows are equal for every element, we can terminate the algorithm, since all the subsequent rows will be the same. This will save you time during exams.

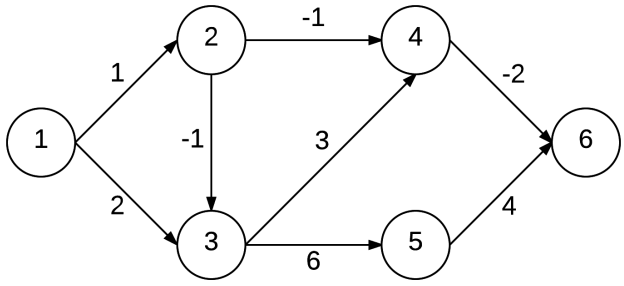


Figure 5.27: Example of Bellman-Ford on a more complex graph.

	1	2	3	4	5	6	
0	0	∞	∞	∞	∞	∞	
1	0	1	2	∞	∞	∞	
2	0	1	0	0	8	∞	
3	0	1	0	0	6	-2	$*$
4	0	1	0	0	6	-2	$*$
5	0	1	0	0	6	-2	
6	0	1	0	0	6	-2	

π	1	2	3	4	5	6
	NULL	1	2	2	3	4

Table 5.6: Example of Bellman-Ford on the graph in Figure 22.

5.10 Floyd-Warshall Algorithm: All Pairs Shortest Path

Dijkstra's and Bellman-Ford's algorithms are designed to calculate the shortest path from a single source to all other nodes in the network. In theory, we may run these algorithms on each node and have the shortest paths from any node to any other node. This however would be inefficient. In this section we introduce the Floyd-Warshall algorithm that specifically calculates the shortest path for each pair of nodes in the network.

This is a dynamic programming approach that works with graphs having edges with negative weights, however it assumes that there are no negative cycles. Nodes are numbered

from 1 to n. The graph is represented by an adjacency matrix $W : n \times n$ defined as follows.

$$W[i,j] = \begin{cases} 0 & i = j \\ w(i,j) & i \neq j \wedge (i,j) \in E \\ \infty & otherwise \end{cases}$$

The algorithm is based on the notion of *intermediate vertex* which is used to define the sub-problems.

Definition 5.10.1 (Intermediate vertex)**.** *Given a path $p =< v_1, \ldots, v_l >$, an intermediate vertex is any vertex v_i s.t. $i = 2, \ldots, l-1$, that is any vertex except v_1 and v_l.*

The sub-problems are defined by progressively expanding the nodes that can appear in a shortest path as intermediate vertices, until all nodes are considered. In particular, let's consider the subset of vertices $\{1, \ldots, k\}$. **For any pair $i, j \in V$, let p be the path with minimum weight of all paths between i and j whose intermediate vertices are in $\{1, \ldots, k\}$.** We can have two cases:

- *k IS NOT an intermediate vertex of p.* Then all intermediate vertices of p are $\{1, \ldots, k-1\}$, thus the sortest path from i to j with intermediate vertices in $\{1, \ldots, k\}$ is also the shortest path from i to j with intermediate vertices in $\{1, \ldots, k-1\}$.
- *k IS an intermediate vertex of p.* Then we have a sub-path p_1 from i to k, and then a sub-path p_2 from k to j. The concatenation of p_1 and p_2 is hence equal to p. Since p is a shortest path, k can only appear once in p. Therefore, k cannot be an intermediate vertex of p_1 or p_2. In addition, thanks to the optimality principle, p_1 is the shortest path between i and k, and p_2 is the shortest path between k and j, both with intermediate vertices in $\{1, \ldots, k-1\}$.

Let $d_{ij}^{(k)}$ be the weight of the shortest path between i and j with intermediate nodes $\{1, \ldots, k\}$. We can use the above discussion to define the recursive relation between the solutions of sub-problems.

Base case, $\mathbf{k = 0}$. There are no intermediate vertices in the path, hence $d_{ij}^{(0)} = W[i,j]$, for each $i, j \in V$.

Inductive case, $\mathbf{k > 0}$. We can pick the minimum between using k as an intermediate vertex and not doing so, that is:

$$d_{ij}^{(k)} = min\left(d_{ij}^{(k-1)}, (d_{ik}^{(k-1)} + d_{kj}^{(k-1)})\right)$$

when $k = n$ we have the solution, that is the length of the shortest path for each pair of vertices.

5.10.1 Pseudo-code

The pseudo-code of the algorithm uses a series of matrices $D^{(k)}$, for $k = 0, \ldots, n$. $D^{(k)}$ contains the elements $d_{ij}^{(k)}$, that is the shortest path distance between i and j using only the intermediate vertices in $\{1, \ldots, k\}$. The code first initializes the matrix $D^{(0)}$ to W, then applies the formula to calculate $D_{ij}^{(k)}$ given $D_{ij}^{(k-1)}$.

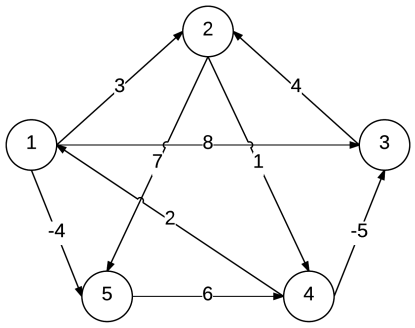
The algorithm complexity is straight forward, since we have three nested loops each performing n iteration. Hence the complexity is $\Theta(n^3)$.

```
1  FloydWorshall(W,n)begin
2  |  D^(0) = W
3  |  for k=1 to n do
4  |  |  Let D^(k) be a new n×n matrix
5  |  |  for i=1 to n do
6  |  |  |  for j=1 to n do
7  |  |  |  |  D_ij^(k) = min(D_ij^(k-1), (D_ik^(k-1) + D_kj^(k-1)))
8  |  |  |  end
9  |  |  end
10 |  end
11 |  return D^(n)
12 end
```

Algorithm 39: Floyd-Warshall algorithm

5.10.2 Example

We now show the execution of the algorithm on the graph in Figure 5.10.2.



$D^{(0)}$	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

(a)

$D^{(1)}$	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

(b)

$D^{(2)}$	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

(c)

$D^{(3)}$	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

(d)

$D^{(4)}$	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

(e)

$D^{(5)}$	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

(f)

Figure 5.28: Execution of Floyd-Warshall algorithm

5.10.3 Constructing the Shortest Path

We are now able to calculate the shortest path distance, but we need to be able to translate this into the actual shortest path between a pair of nodes. We can easily extend the previous algorithm. We make use of the predecessor matrix $\Pi : n \times n$ as follows.

$$\Pi[i,j] = \begin{cases} NIL & i = j \\ NIL & \nexists \text{ a path from i to j} \\ \text{predecessor of j in the shortest path to i} & \text{otherwise} \end{cases}$$

Given Π we can print the shortest path between i and j as shown in Algorithm 40.

```
1  Print-All-Pairs-Shortest-Path(Π,i,j)begin
2      if i=j then
3          print i;
4          return;
5      end
6      if Π[i, j] = NIL then
7          print "No path between i and j";
8          return;
9      end
10     Print-All-Pairs-Shortest-Path(Π,i,Π[i, j]);
11     print j;
12     return;
13 end
```

Algorithm 40: Print shortest path from predecessor matrix

We define the matrix $\Pi^{(k)} : n \times n$ such that $\Pi^{(k)}[i, j]$ is the predecessor of j in the shortest path to i using intermediate vertices in $\{1, \ldots, k\}$. We can now follow a similar idea of the main algorithm to update $\Pi^{(k)}$.

Base case, k = 0.

$$\Pi^{(0)}[i, j] = \begin{cases} NIL & i = j \vee W[i, j] = \infty \\ i & i \neq j \wedge W[i, j] < \infty \end{cases}$$

Inductive case, k > 0.

- If the shortest path does not contain k, then $\Pi^{(k)}[i, j] = \Pi^{(k-1)}[i, j]$
- If the shortest path contains k, then $\Pi^{(k)}[i, j] = \Pi^{(k-1)}[k, j]$

The pseudo-code is in Algorithm 41.

```
1  FloydWorshall(W,n)begin
2      D^(0) = W
3      for i = 1 to n do
4          for j = 1 to n do
5              if i = j OR W[i,j] = ∞ then
6                  Π^(0)[i, j] = NIL
7              end
8              else
9                  Π^(0)[i, j] = W[i, j]
10             end
11         end
12     end
13     for k=1 to n do
14         Let D^(k) be a new n×n matrix
15         for i=1 to n do
16             for j=1 to n do
17                 if D_ij^(k−1) < (D_ik^(k−1) + D_kj^(k−1)) then
18                     D_ij^(k) = D_ik^(k−1)
19                     Π^(k)[i, j] = Π^(k−1)[i, j]
20                 end
21                 else
22                     D_ij^(k) = (D_ik^(k−1) + D_kj^(k−1))
23                     Π^(k)[i, j] = Π^(k−1)[k, j]
24                 end
25             end
26         end
27     end
28     return D^(n), Π^(n)
29 end
```

Algorithm 41: Floyd-Warshall algorithm with predecessor matrix

5.11 Maximum Flow

The maximum flow problem is another example of widely used approach from graph theory. It finds applications in communication networks, road traffic management, resource management, and operating systems, just to mention a few. In the general formulation, we consider a graph, where a node s, called *source* produces data, while a node t, called *destination*, collects the data coming from the source. The graph is directed, where the direction defines the allowed flow direction of data. Edges are weighted, and the weight of an edge represents its capacity, i.e. the amount of data flow that can go through that edge.

Intermediate nodes (not s or t) just forward the data, with the constraint that all incoming data in a node equals all the outgoing data from that node. This is often called the flow conservation constraint, which allows only the source to generate data and only the destination to consume it.

Our objective is to calculate the maximum rate at which s can send data to t without violating the link capacities. As an example, consider the graph in Figure 5.29. The maximum flow from s to t is 5. In this case it is straight forward to calculate the solution, since the paths from s to t are disjoint, however it gets more complicated when we have an arbitrary graph, and we need to define an algorithm that returns the MaxFlow. Let's first introduce some definitions.

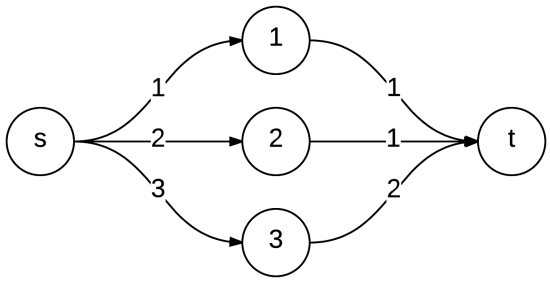


Figure 5.29: Example of MaxFlow problem, the maxflow is 5.

Definition 5.11.1 (Flow Network)**.** *A Flow Network is a directed graph G=(V,E) s.t.* $\forall (u, v) \in E$ *the capacity* $c(u, v) \geq 0$*. In addition, if* $(u, v) \in E$ *then* $(v, u) \notin E$ *and G does not contain self-loops. There are two special nodes* $s, t \in V$*,* $s \neq t$*. Finally,* $\forall\ v \in V \setminus \{s, t\}$ $\exists$ *a path* $\{s, \ldots, v, \ldots, t\}$*.*

Definition 5.11.2 (Flow)**.** *A flow in G is a function* $f : V \times V \to \mathbb{R}$ *such that:*

- *Capacity constraint:* $\forall (u, v) \in V$*,* $0 \leq f(u, v) \leq c(u, v)$
- *Flow conservation constraint:* $\forall u \in V \setminus \{s, t\}$ $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$

Definition 5.11.3 (Flow value)**.** *Given a flow of f, the value $|f|$ is the total amount of flow leaving the source minus the total amount of flow entering the source, that is:*

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Definition 5.11.4 (Residual network)**.** *Given a flow network G and a flow f the residual network G_f consists of edges with capacities that represent how we can change the flow on G. Formally, G_f is defined as follows.*

- *It has same set of nodes as G.*
- *$\forall (u, v) \in E$:*
  - *Residual capacity is $c_f(u, v) = c(u, v) - f(u, v)$. This represents how much we can increase the flow in (u, v).*
  - *We consider the edge (v, u) and set its capacity to $c_f(v, u) = f(u, v)$. This represents how much we can decrease the flow in (u, v).*
  - *In summary:*

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & (u, v) \in E \\ f(u, v) & (v, u) \in E \\ 0 & otherwise \end{cases}$$

- *We add to G_f the all the edges $E_f = \{(u, v) \in V \times V \text{ s.t. } c_f(u, v) > 0\}$*

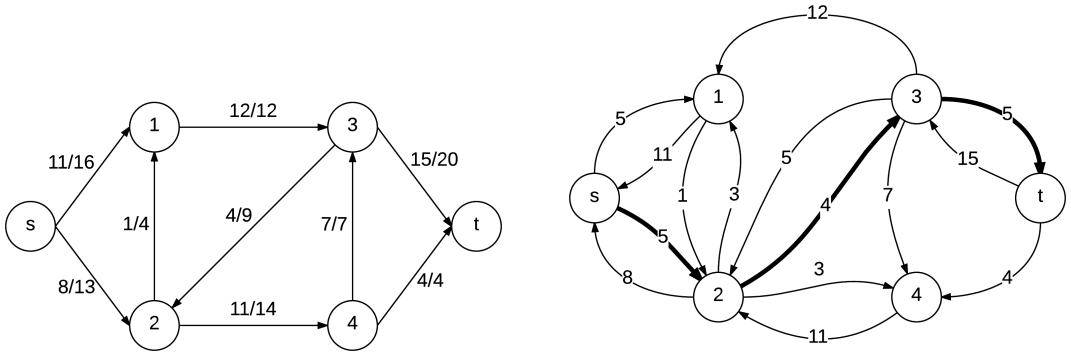


Figure 5.30: Example of flow network and corresponding residual network. The augmenting path is highlighted in bold.

Definition 5.11.5 (Augmenting Path)**.** *Given a flow network $G = (V, E)$ and a flow f, an augmenting path is a simple path from s to t in G_f. The maximum amount of flow that can be sent on an augmenting path p is the residual capacity $c_f(p) = min\{c_f(u, v) \text{ s.t. } (u, v) \in p\}$.*

An example of flow network and corresponding residual network is shown in Figure 5.30. Figure 5.30 (b) also shows an augmenting path with residual capacity 4, which is due to the capacity of the edge $(2, 3)$.

Given an augmenting path p with residual capacity p, we can calculate a new *augmenting flow* f_p s.t.

$$f_p(u, v) = \begin{cases} c_f(p) & (u, v) \in p \\ 0 & otherwise \end{cases}$$

The augmenting flow f_p can be used to augment the flow f in G as follows:

$$f'(u, v) = \begin{cases} f'(u, v) + f_p(u, v) - f_p(v, u) & (u, v) \in E \\ 0 & otherwise \end{cases}$$

5.11.1 Ford-Fulkerson Algorithm

The Ford-Fulkerson exploits the concepts defined previously. The main idea is to start from a flow network G with an empty flow f. We build the residual network G_f and find an augmenting path p, and update the flow f by the augmenting flow f_p. The procedure is iterated until no more augmenting path can be found. The value of the flow in G is then the maximum flow.

The pseudo-code of the Ford-Fulkerson algorithm is shown in Algorithm 42. If the capacities are all integer values, it can be shown that the complexity is $O(E \times |F^*|)$.

5.11.2 Example

```
1  Ford-Fulkerson(G,s,t)begin
2  |  // The initial flow on each edge is empty
3  |  for (u, v) ∈ E do
4  |  |  (u, v).f = 0
5  |  end
6  |  calculate residual network G_f
7  |  while ∃p from s to t in G_f do
8  |  |  select an augmenting path p in G_f
9  |  |  c_f(p) = min{c_f(u, v) : (u, v) ∈ p}
10 |  |  for (u, v) ∈ p do
11 |  |  |  if (u, v) ∈ E then
12 |  |  |  |  (u, v).f = (u, v).f + c_f(p)
13 |  |  |  end
14 |  |  |  else
15 |  |  |  |  (v, u).f = (v, u).f − c_f(p)
16 |  |  |  end
17 |  |  end
18 |  |  calculate G_f
19 |  end
20 |  return Σ_{v∈V} f(s, v) − Σ_{v∈V} f(v, s)
21 end
```

Algorithm 42: Pseudo-code Ford-Fulkerson algorithm

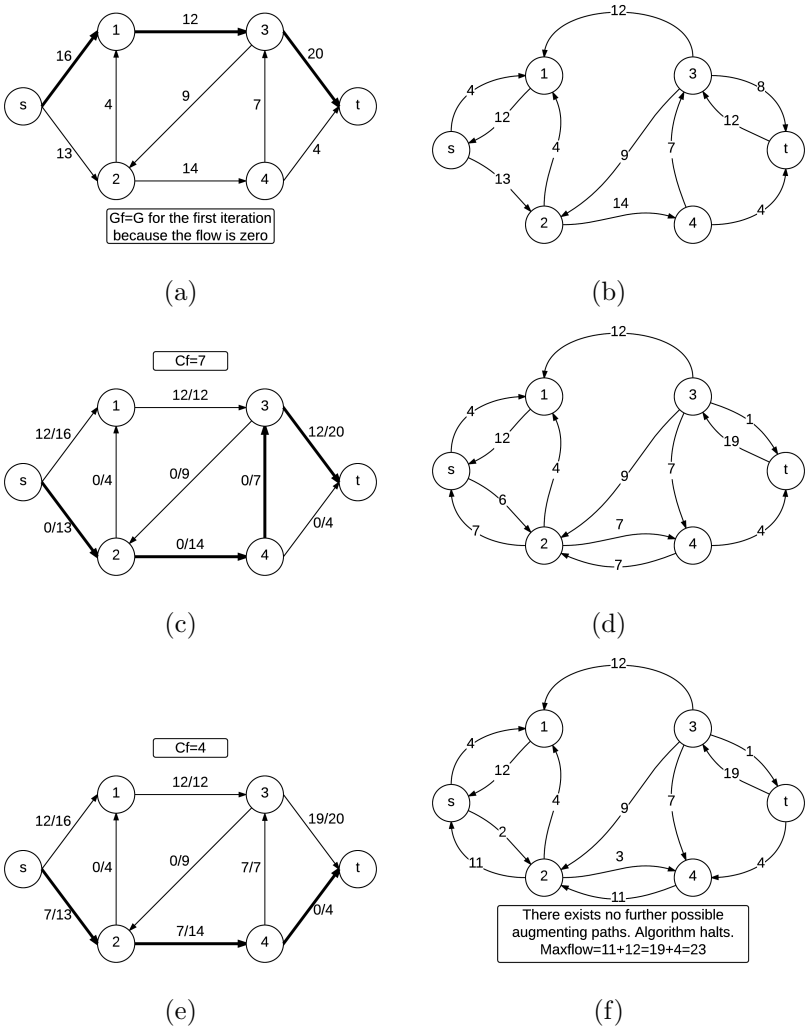


Figure 5.31: Example of Ford-Fulkerson algorithm.

5.11.3 MaxFlow-MinCut Theorem

Definition 5.11.6 (Cuts)**.** *A cut in a flow netwrok G=(V,E) is a partition of V into two sets S and T=V\S, such that $s \in S$ and $t \in T$.*

Definition 5.11.7 (Cut Capacity)**.** *The capacity of a cut (S,T) is:*

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

The minimum cut is a cut with minimum capacity.

Definition 5.11.8 (Maxflow Min Cut Theorem)**.** *Given a flow network G=(V,E) the value of the maximum flow for G is equal to the capacity of the minimum cut.*

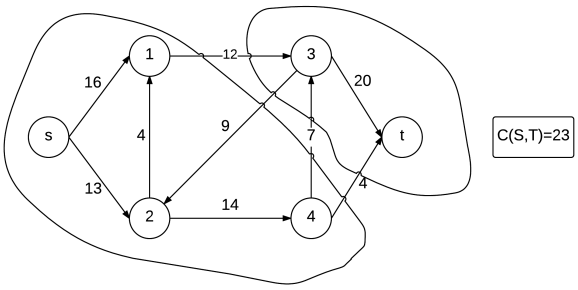


Figure 5.32: Cut in a graph illustrating the maxflow-min cut principle

5.12 Bipartite Matching Problem

Consider the following problem. We have n_L employees to be assigned to n_R tasks. Each employee has a set of skills which allow him/her to perform only some of the tasks. A task needs only a single employee to be completed and an employee can execute at most one task. Our problem is to assign, or match, employees with tasks so that we maximize the number of completed tasks. We can model this problem using a bipartite graph, defined as follows.

Definition 5.12.1 (Bipartite Graph)**.** *A Bipartite graph is a graph $G = (L \cup R, E)$ in which the set of nodes is partitioned into two disjoint subsets, L and R. This means $V = L \cup R$ and $L \cap R = \emptyset$. Edges are only between a node in L and a node in R, that is $E \subseteq L \times R$ which is equivalent to say that $\forall\ (u, v) \in E$, $u \in L \wedge v \in R$.*

In the worker assigned problem, L is the set of employees, R is the set of tasks, and there is an edge between and employee $u \in L$ and a task $v \in R$ if u can perform v.

Definition 5.12.2 (Matching)**.** *A matching M is a subset of E such that for each node $v \in L \cup R$, there is at most one edge in the matching which is incident to v.*

Definition 5.12.3 (Maximal Matching)**.** *A matching M is maximal if it cannot be further extended, that is adding any other edge would make M not a matching anymore.*

Definition 5.12.4 (Maximum Matching)**.** *A maximum matching is a matching with maximum cardinality.*

Note that, a maximum matching is a maximal matching, but the viceversa is not necessarily true. Figure 5.33 provides an example.

5.12.1 Algorithm for Maximum Matching

We can use the Ford Fulkerson Algorithm to solve the maximum matching problem. Specifically, given the bipartite graph $G = (L \cup R, E)$ we build a flow network as follows.

- Add a node for each node in L and R.

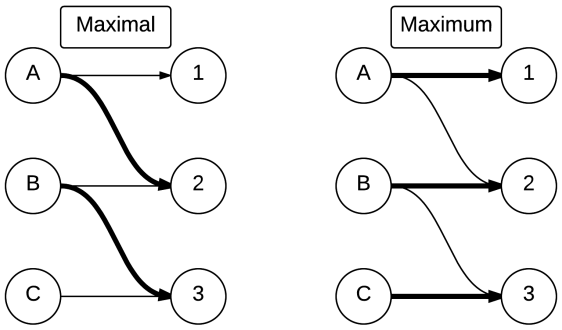


Figure 5.33: Maximal vs. Maximum matching.

- Edges are the same as in G, but directed from nodes in L to nodes in R.
- We add two special nodes s and t.
- We add an edge (s, v) for each $v \in L$.
- We add an edge (v, t) for each $v \in R$.
- All edges have capacity 1.

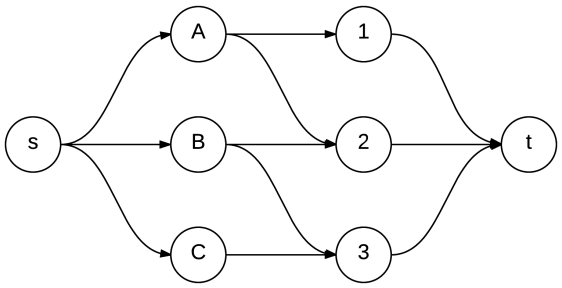


Figure 5.34: Flow network for the bipartite graph in Figure 5.33.

Figure 5.34 shows the flow network for the bipartite graph in Figure 5.33. We can now calculate the maximum flow from s to t, the edges (u, v), from L to R, s.t. $(u, v,).f = 1$ correspond to the maximum matching.

In order to have an intuition of why this works, we should realize that the Ford Fulkerson algorithm saturates an augmenting path when it selects it. In this case, since all edges have the same capacity, it saturate all edges in the path. As a result, if a flow traverses a node v through an edge, v cannot be traversed by the flow of any other edge. This implies that the output of the Ford Fulkerson Algorithm is a matching. Since we find the maximum flow, and again edges have unit capacity, we also maximize the number of nodes that are part of the flow (each node can appear at most once), thus we are also calculating the maximum matching.

5.13 Exercise

1. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

2. Show the d and π values that result from running breadth-first search on the directed graph of Figure 5.1, using vertex 3 as the source.

3. Show how depth-first search works on the graph of Figure 5.35. Assume the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.

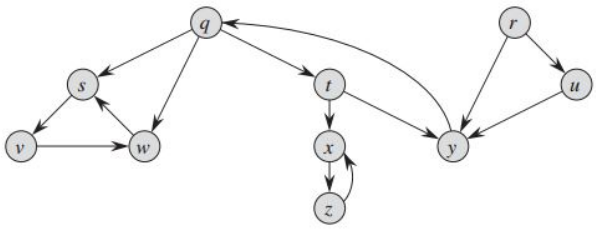


Figure 5.35: A directed (not acyclic) graph

4. A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.

a. Prove that in a *breadth-first search* of an undirected graph, the following properties hold:

1. There are no back edges and no forward edges.
2. For each tree edge (u, v), we have $v.d = u.d + 1$.
3. For each cross edge (u, v), we have $v.d = u.d$ or $v.d = u.d + 1$.

b. Prove that in a *breadth-first search* of a directed graph, the following properties hold:

1. There are no forward edges.
2. For each tree edge (u, v), we have $v.d = u.d + 1$.
3. For each cross edge (u, v), we have $v.d \leq u.d + 1$.
4. For each back edge (u, v), we have $0 \leq v.d \leq u.d$.

5. The **edge connectivity** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

6. A **path cover** of a directed graph $G = (V, E)$ is a set P of vertex-disjoint paths such that every vertex in V is included in exactly one path in P. Paths may start and end anywhere, and they may be of any length, including0. A **minimum path cover** of G is a path cover containing the fewest possible paths.
a. Give an efficient algorithm to find a minimum path cover of a directed acyclic graph $G = (V, E)$. (Hint: Assuming that $V = \{1, 2, \ldots n\}$, construct the graph $G' = (V', E')$ where $V' = \{x_0, x_1, \ldots x_n\} \cup \{y_0, y_1, \ldots y_n\}$
$E' = \{(x_0, x_1) : i \in V\} \cup \{(y_0, y_1) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\}$, and run a maximum-flow algorithm.)
b. Does your algorithm work for directed graphs that contain cycles? Explain.

7. Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

8. Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $49 \times 2 \times 0.0107 = 1.0486$ U.S. dollars, thus turning a profit of 4.86 percent. Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table Rof exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j.
a. Give an efficient algorithm to determine whether or not there exists a sequence of currencies $(c_{i1}, c_{i2}, \ldots c_{ik})$ such that
$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] > 1$.
Analyze the running time of your algorithm.
b. Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

9. a. Given a DAG with non-negative weights, briefly provide an $O(V^2)$ algorithm.
b. Can you improve the complexity?
c. What happens if there are negative weights?

10. In this problem, we give pseudo-code for three different algorithms (refer to the Algorithm 43, Algorithm 44 and Algorithm 45). Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not a minimum spanning tree. Also describe

the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree.

```
1  MAYBE-MST-A(G, w)begin
2  |  sort the edges into non-increasing order of edge weights w;
3  |  T = E;
4  |  for each edge e, taken in non-increasing order by weight do
5  |  |  if T − {e} is a connected graph then
6  |  |  |  T = T − {e};
7  |  |  end
8  |  end
9  |  return T
10 end
```

Algorithm 43: MAYBE-MST-A

```
1  MAYBE-MST-B(G, w)begin
2  |  T = ∅;
3  |  for each edge e, taken in arbitrary order do
4  |  |  if T ∪ {e} has no cycles then
5  |  |  |  T = T ∪ {e};
6  |  |  end
7  |  end
8  |  return T
9  end
```

Algorithm 44: MAYBE-MST-B

11. We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

12. Suppose that we wish to maintain the transitive closure of a directed graph $G = (V, E)$ as we insert edges into E. That is, after each edge has been inserted, we want to update the transitive closure of the edges inserted so far. Assume that the graph G has no edges initially and that we represent the transitive closure as a boolean matrix.
a. Show how to update the transitive closure $G^* = (V, E^*)$ of a graph $G = (V, E)$ in $O(V^2)$ time when a new edge is added to G.
b. Give an example of a graph G and an edge e such that $\Omega(V^2)$ time is required to update the transitive closure after the insertion of e into G, no matter what algorithm is used.

```
1  MAYBE-MST-C(G, w)begin
2  |  T = ∅;
3  |  for each edge e, taken in arbitrary order do
4  |  |  T = T ∪ {e};
5  |  |  if T has a cycle c then
6  |  |  |  let e' be a maximum-weight edge on c T = T − {e'};
7  |  |  end
8  |  end
9  |  return T
10 end
```

Algorithm 45: MAYBE-MST-C

c. Describe an efficient algorithm for updating the transitive closure as edges are inserted into the graph. For any sequence of n insertions, your algorithm should run in total time $\sum_{i=1}^{n} t_i = O(V^3)$, where t_i is the time to update the transitive closure upon inserting the $i-$th edge. Prove that your algorithm attains this time bound.

13. Let us define a **relaxed red-black tree** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5 (refer to Subsection 5.7.1). In other words, the root may be either red or black. Consider a relaxed red-black tree T whose root is red. If we color the root of T black but make no other changes to T, is the resulting tree a red-black tree?

14. Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

15. What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

16. Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$rotations. (Hint: First show that at most $n-1$ right rotations suffice to transform the tree into a right-going chain.)

17. We say that a binary search tree T_1 can be **right-converted** to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to $RIGHT-ROTATE$. Give an example of two trees T_1 and T_2 such that T_1 cannot be *right-converted* to T_2. Then, show that if a tree T_1 can be right-converted to T_2, it can be right-converted using $O(n^2)$ calls to $RIGHT-ROTATE$.

18. Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

19. Professor Teach is concerned that *RB-INSERT-FIXUP* might set *T.nil.color* to RED, in

which case the test in *while loop* would not cause the loop to terminate when $\mathcal{Z}$ is the root. Show that the professors concern is unfounded by arguing that *RB-INSERT-FIXUP* never sets $T.nil.color$ to RED.

20. Suppose that a node x is inserted into a red-black tree with *RB-INSERT* and then is immediately deleted with *RB-DELETE*. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

21. In each of the cases of Figure 5.18, give the count of black nodes from the root of the subtree shown to each of the subtrees $\alpha, \beta, \ldots, \zeta$, and verify that each count remains the same after the transformation. When a node has a color attribute c or c', use the notation count(c) or count(c') symbolically in your count.

22. *Join operation on red-black trees:* The **join** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \leq x.key \leq x_2.key$. It returns a set $S = S_1 \bigcup \{x\} \bigcup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

A. Given a red-black tree T, let us store its black-height as the new attribute $T.bh$. Argue that *RB-INSERT* and *RB-DELETE* can maintain the bh attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show that while descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited.

We wish to implement the operation $RB - JOIN(T_1, x, T_2)$, which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \bigcup \{x\} \bigcup T_2$. Let n be the total number of nodes in T_1 and T_2.

1. Assume that $T_1.bh \geq T_2.bh$. Describe an $O(lgn)$ time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose blackheight is $T_2.bh$.

2. Let T_y be the subtree rooted at y. Describe how $T_y \bigcup \{x\} \bigcup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

3. What color should we makexso that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(lgn)$ time.

4. Argue that no generality is lost by making the assumption in part (1). Describe the symmetric situation that arises when $T_1.bh \leq T_2.bh$.

5. Argue that the running time of *RB-JOIN* is $O(lgn)$.

Bibliography

[1] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. MIT press, 2009.