Algorithmic Methods

Based on the course by Prof. Gianmaria Silvello

Francesco Ferretto, Riccardo Mazzieri,
Gianmarco Cracco [1] and Alessandro Manente [2]

University of Padua, Department of Mathematics
Master Degree in Data Science

February 25, 2021

[1] Acknowledgements goes for reviewing and contributing for the first draft of the notes.
[2] Acknowledgements goes for improving the aesthetics for the first draft of the notes, for some minor corrections and being BRV.

Contents

Notation

In this elaborate, in order to keep track of important things to the eye, we used coloured boxes.
In particular, gray boxes are used for *theoretical* parts (like Theorems, Definitions, Lemmas,...):

Computational Problem

A computational problem $\mathbf{\Pi}$ is a mathematical relation between a set I of possible instances and a set S of possible solutions:

$\mathbf{\Pi} \subseteq I \times S$ such that $\forall i \in I$ there exists ($\exists$) at least one solution s $\in$ S such that $(i, s) \in \mathbf{\Pi}$

An algorithm A solves the problem $\mathbf{\Pi} \subseteq I \times S$, if $\forall i \in I$:

$$\mathrm{A(i)} \rightarrow \mathrm{s}$$

In other words an algorithm is a *function* that take instances i as input from I and gives back a solution s as output.

Blue boxes are used for *conventions* that will be used for rest of the notes:

Assumption of the cost per type of command

So even if we know that the cost depends on the type of statement[a] (and it's possible to obtain it, for precise estimates), indeed in the following we **assume the unitary cost for all elementary operations**:

$$c_1 = c_2 = c_5 = \cdots = c_8 = 1 \quad (*)$$

and $c_3 = c_4 = 0$ (for the **comment**).

[a] These costs in practice depend on the kind of machine

Yellow for *key aspects*:

Asymptotic Notation in Seven Words

$$\text{suppress } \underbrace{\text{constant factors}}_{\text{too system-dependent}} \text{ and } \underbrace{\text{lower-order terms}}_{\text{irrelevant for large inputs}}$$

Red for *formulas to keep in mind*:

Formula to keep in mind

In general, for any polynomial $p(n) = \sum_{i=0}^{d} a_i n^i$, where the a_i are constants and $a_d > 0$, we have $p(n) = \Theta\left(n^d\right)$.

Green for *solving schemes*:

How to solve Recurrencies

1. Understand the pattern of the cost as the level increases;
2. Understand the number of needed passages to get unitary input, i.e. compute the height of the tree;
3. Complexity computation

$$T(n) \in \underbrace{\sum_{i=0}^{height-=\log_b(n)-1} \text{cost per level}}_{InternalLevels'Cost} + \underbrace{\sum_{i=1}^{\#leaves=a^{\log_b(n)}} \Theta(1)}_{ExternalLevel'sCost}$$

4. Eventually check the solution with the *substitution method*;

Note: floors $\lfloor \cdot \rfloor$ and ceilings $\lceil \cdot \rceil$ *usually* do not matter when solving recurrences.

Disclaimer: In these notes we have tried to unify all the material given by Prof. Silvello, trying to create attractive diagrams and to take care of the elements to be recovered during the demonstrations. They may contain errors, so use them responsibly. We recommend that, in case you'll chose to use these notes, to do that in conjunction with the course textbooks. We apologise in advance for the presence of possible errors and we invite you to send us an email to correct them (and also to give suggestions) at francesco.ferretto@studenti.unipd.it with the subject line "`AMnotes [specification]`".

These notes were not supervised by Professor Silvello.

Chapter 1

Introduction

Computational Thinking

Computational Thinking - TC

CT is the though process involved in ***formulating a problem*** *and expressing its* ***solution****(s) in such a way that a computer - human or machine - can effectively carry out.*

In other words is a process of the mind, that consist into formulating a problem and a solution in such a way that can be understood by someone or something that does not understand anything about the solution or the problem.
This way of formulating problems is not meant just for machines, but also for people that executes a sequence of steps in a mechanical way.
Another definition is the following:

> *[CT is] the mental activity for* ***abstracting problems*** *and formulating* ***solutions*** *that can be* ***automated****.*

Here we see a more in depth meaning.
But after these definition what get the point is that CT is the abstraction of a problem using every available tool from mathematics and CS in order to formalize and algorithmically implement a code that can be executed by anyone (or anything), without any cognitive load, in fact:

CT is the process of recognising aspects of computations in the world that surrounds us, and applying tools and techniques from Computer Science to understand and reason about both natural and artificial systems and processes.

So finally, we see that the junction of *Algorithmic Thinking* and *Computer Science* leads to a process of abstraction of a problem in order to express it in mathematical way that could be implemented in a sequence of finite steps executable by a Turing Machine (TM), and so by every agent who is capable of execute the same set of operations that a TM does.

Let's get deep into Algorithmic Thinking:

Algorithmic Thinking

Solutions are Algorithms? 1024, 32, TRUE, ...
These may be solutions of a problem, but they are not algorithms.

Algorithms are Solutions?

Algorithms are sequences of basic steps that a non-intelligent being can blindly follow to solve a problem

Let's focus on the features of algorithms:

- Sequences of *basic* step, every operation is minimal in terms of effort;
- *Ordered*;
- A *protocol*, a precise set or rules to follow in order to solve the problem;

But, what is exactly an Algorithm?

What's an Algorithm

Basically an algorithm is a sequence of instructions to describe the solution of a problem, a well-defined computational procedure that takes some value(s) as input and produces some value(s) as output.

Algorithms give you a language to talk about problems and solutions. Let's see an example:

Euclid's algorithm Given two positive numbers n and m find their greatest common divisor.

E1. [Find remainder] Divide m by n and let r be the remainder ($0 \leq r < n$);

E2. [Is it zero?] If $r = 0$, the algorithm ends; n is the answer.

E3. [Reduce] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step E1.

From this example we may ask, are there specific characteristics that an algorithm should respect in order to be defined such? Knuth replied to this, with 5 rules.

Knuth's Five Rules

1. *Finiteness*: It must always terminate after a finite number of steps;
2. *Definiteness*: Each step must be precisely defined;
3. *Input*:It has zero or more inputs;
4. *Output*: It has one or more outputs;
5. *Effectiveness*: Its operations must all be sufficiently basic that they can in principle be done exactly in a **finite amount** of time by someone using pencil and paper.

An algorithm without finiteness is a *computational method*.

A famous problem in algorithmics: The Sorting problem

- Input: a sequence of n numbers $< a_1, a_2, \ldots, a_n >$
- Output: a permutation (or rearrangement) $< a'_1, a'_2, \ldots, a'_n >$ of the input such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

In this case an algorithm is the sequence of operations to obtain the sorted list of numbers starting from the unordered one. We can define the algorithm by using *pseudo-code* or any available programming language (e.g. Java, Python,...). For example $< a_1, a_2, \ldots, a_n > = < 34, 2, 1, 45, 565 >$, with $n = 5$, this is called an *instance* of the sorting problem, i.e. a *specific* sequence of numbers that we are considering. A sorting algorithm returns the output sequence $< 1, 2, 34, 45, 565 >$.

An algorithm is *correct* if it halts with the correct output, so we say that an algorithm is correct if it solves the given computational problem.

Computational Problem

A computational problem $\mathbf{\Pi}$ is a mathematical relation between a set I of possible instances and a set S of possible solutions:

$\mathbf{\Pi} \subseteq I \times S$ such that $\forall i \in I$ there exists ($\exists$) at least one solution s $\in$ S such that $(i, s) \in \mathbf{\Pi}$. An algorithm A solves the problem $\mathbf{\Pi} \subseteq I \times S$, if $\forall i \in I$:

$$\text{A(i)} \rightarrow \text{s}$$

In other words an algorithm is a *function* that take instances i as input from I and gives back a solution s as output.

The study of Performance

In our first attempt of defining computational thinking, we encounter algorithms that for expressing words through eye-blink may get better using different protocols, and so having different performances.
Nowadays performance is a metric of evaluation of algorithms, but we should be aware that some other aspects may be evaluated in order to choose an algorithm that better fits to an application in a specific domain, so...

What's more important than performance?

- Correctness: if it's not correct in providing a solution, then it's not useful;
- Cost: maintenance cost, developing cost, HR,...
- Maintainability: explanation of the code and the reasons of our choices;
- Stability and robustness: It must work for any instance of the input, and so the behaviour must be predicted and the running time must remain the same across time, so robust to change and to complex inputs.
- Having a wide range of features: at a parity of the previous conditions is preferable to choose an algorithm that have more features.
- Modularity: it's a key characteristic. Consist in to decomposing the problem in different and independent parts in order to create scalable algorithms and, consequently, software solutions in a dynamic world.
- Security
- User-friendliness: the comprehension of algorithm is a fundamental aspect.

But why focusing on performance? Often performances define the line between *feasible* and *unfeasible*:

- Sometimes if it's not real-time then it's not useful;
- If it requires too much *time* or *space* then it is not usable;

So in other words, looking at the more important aspects is not enough in certain scenarios, we need for computationally less expensive algorithms (in terms of space (memory) and time (velocity)).

But how we choose a measure of evaluation of an algorithm, in order to compare performances of different algorithms with the same duty?

Analysis of Algorithms

Analysis of Algorithms is the study of computer-program performance and resource usage.

It's aim is to determine how *good* an algorithm is, focusing particularly at performance than other characteristics (some of are taken as granted and secondly studied).
The idea is to take an algorithm and to determine its *quantitative behaviour*, so we're not evaluating the correctness of the algorithm, but the overall performance in terms of time a space required. So we may ask:

- How many operations are the algorithm doing to solve a problem?
- Given two algorithms A and B both solving the problem p, do we prefer A or B? Why?

The core is to determine the performance characteristics of an algorithm.

Efficiency of an Algorithm Even though computers are getting faster and faster, *we should still care about algorithmic efficiency and memory use, Fig.1.1*

Expressing efficiency of an algorithm

Algorithmic ***efficiency is often measured in terms of the input size*** n (n is the size of the input).

For example, `Insertion sort` requires $c_1 n^2$ time to sort a sequence of n numbers, meanwhile `MERGE sort` requires $c_2 \cdot n \cdot log_2(n)$ time to sort a sequence of n numbers. We see how a common measure of performance can be based on the order of magnitude of the number of operations to be done in order to solve a problem (in function of n).

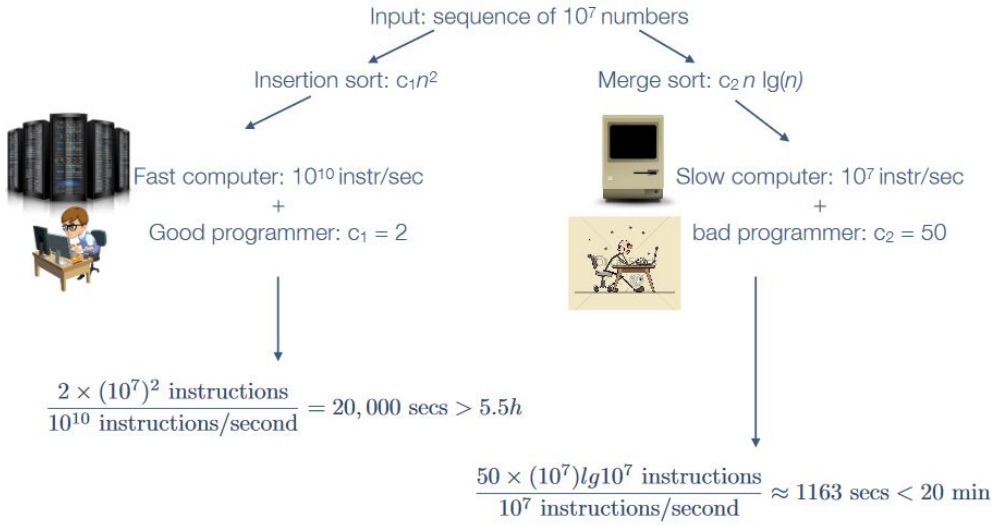


Figure 1.1: Comparison of Insertion Sort and MergeSort on different machines.

We see how measuring performance in terms of running time (that depends on n) allows to compare performance across different machines, and this emphasises the fact the performance of an algorithm is crucial, because the computational aspects of an algorithm will always predominate on the computational power of the machine onto which it's executed.

Part I

Fundamentals

Chapter 2

Sorting Algorithms

To introduce the world of algorithms we start from those that face the *sorting* problem, in the meanwhile we present the canonical way in which iterative algorithms are studied in terms of their performance.

2.1 Insertion Sort - $O(n^2)$

Intuition: at step j we want to insert the element $A[j]$ in its right place into the ordered list $A[1, \ldots, j-1]$. In this way, the sublist $A[1, \ldots, j]$ will be always ordered after each iteration of the `for` cycle.

Algorithm 1 INSERTION SORT

```
1: for j = 2 to length(A) do
2:     key=A[j]
3:     i = j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] = A[i]
6:         i = i − 1
7:     A[i + 1] = key
```

Worst Case Complexity: $O(n^2)$.

Intuition: the worst case is the one where the list is in reversed order. For this reason, for each iteration of the for loop, the whole list will be scanned again $\longrightarrow n^2$ complexity

2.2 Selection Sort - $O(n^2)$

Intuition: the idea of the Selection Sort algorithm is to find the smallest element of the list and bringing it at the beginning of the list. More specifically, the concept is, at a given iteration j, to move the smallest element on the sub-array on the right (`A[j+1,...,n]`) in the current location `A[j]` (*loop invariant's property of the cycle*).

Worst Case Complexity: $O(n^2)$.

Intuition: the worst case is the same of the best case, since the whole list needs to be scanned anyway. We have one full scan to check all the elements (outer for loop), and $\sum_{j=1}^{n-1} j = \frac{(n-1)n}{2}$ [1] scans to find the minimum (inner for loop) $\longrightarrow n^2$ complexity.

[1] $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$

Algorithm 2 SELECTION SORT

```
1: for j = 1 to length(A)−1 do
2:     smallest = j
3:     for i = j + 1 to length(A) do
4:         if A[i] < A[smallest] then
5:             smallest = i
6:     exchange A[j] with A[smallest]
```

2.3 Bubble Sort - $O(n^2)$

Intuition: Bubble Sort iteratively compares every pair of adjacent elements and swaps them if they are in the wrong order. The pass is repeated until no swaps are needed.

Algorithm 3 BUBBLE SORT

```
1: flag = true
2: while flag do
3:     flag = false
4:     for j = 1 to length(A)−1 do
5:         e1 = A[j]
6:         e2 = A[j + 1]
7:         if e1 > e2 then
8:             A[j] = e2
9:             A[j + 1] = e1
10:            flag = true
```

Worst Case Complexity: $O(n^2)$

Intuition: worst case is that of a reversed order list. In such a case, we need to perform a swap at each step: this will be two nested for loops! $\longrightarrow n^2$ complexity.

2.4 Loop Invariant Technique

A **Loop Invariant** is a property of a loop that is true before (and after) each iteration. We can verify this property with a technique made of three main steps.

To understand it better, we make the example of the Insertion Sort algorithm: the loop invariant in that case is that the sublist $A[1, \ldots, j-1]$ is always ordered. We will return to this aspect in future sections.

In general, the three steps are the following:

Loop Invariant Technique

1. **Initialization**: the loop invariant condition is true **prior** to the first iteration of the loop;
2. **Maintanance**: If the invariant is true before an iteration of the loop, it should be true also after the iteration.
3. **Termination**: when the loop terminates, the loop invariant is still satisfied (in the case of Insertion Sort, we know that the whole list is ordered at the end of the loop, therefore the loop invariant is still satisfied).

When the first two properties hold, the loop invariant is true prior to every iteration of the loop (Of course, we are free to use established facts other than the loop invariant itself to prove that the loop invariant remains true before each iteration). Note the

similarity to *mathematical induction*, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step. The third property is perhaps the most important one, since we are using the loop invariant to show correctness. Typically, we use the loop invariant along with the condition that caused the loop to terminate. The termination property differs from how we usually use mathematical induction, in which we apply the inductive step infinitely; here, we stop the "induction" when the loop terminates.

Chapter 3

Asymptotic Analysis

In some cases we can define the exact running time of an algorithm, but in reality the extra precision is almost never useful: *constants do not have a real impact on running times*. We are really interested in **asymptotic behavior** of the algorithms (i.e. when $n \to \infty$, where n is the size of the input[1]), thus we can get rid of multiplicative factors and lower order terms.

Asymptotic Notation in Seven Words

$$\text{suppress } \underbrace{\text{constant factors}}_{\text{too system-dependent}} \text{ and } \underbrace{\text{lower-order terms}}_{\text{irrelevant for large inputs}}$$

We now define three different notations that will be useful to compare the asymptotic behavior of different functions (i.e. algorithms' running time). Below we assume that $f(n)$ and $g(n)$ are real valued functions.

3.1 Big-Theta - Θ

Big-Theta

$$\Theta(g(n)) := \{f(n) : \ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0, \quad \exists c_1, c_2 > 0, n_0 \geq 0\}$$

alternatively:

$$f(n) \in \Theta(g(n)) \iff \exists c, \quad 0 < c < +\infty \quad s.t. \lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

With a little **abuse of notation**[2] we can write $f(n) = \Theta(g(n))$ instead of $f(n) \in \Theta(g(n))$. We say that $g(n)$ is an asymptotically **tight bound** for $f(n)$ (in algo's lingo we say that $f(n)$ is sandwiched between $c_1 g(n)$ and $c_2 g(n)$). An algorithm having this property (and a low order of growth) is desirable. Now we're going to see 2 examples regarding how to use the definition in order to establish whether or not a function $f(n)$ has a tight bound described by a specific function $g(n)$.

AAoN

Example:

[1] We're going to consider a specific case of input type of length n, which will be the most complex input (i.e. the input of length n that takes the maximum amount of time to be "solved") and defines the *worst-case analysis* of an algorithm

[2] An allowed abuse of notation (AAoN) which we'll use from this moment on

Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.
We have to show that

$$\exists c_1, c_2 \in \mathbb{R}^+ \setminus + \{\infty\} \quad s.t. \quad 0 \leq c_1 g(n) \leq \frac{1}{2}n^2 - 3n \leq c_2 g(n) \quad \text{for} \quad n \geq n_0$$

.

Solution:

$$\Rightarrow (0 \leq) c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

$$\Rightarrow (0 \leq) \frac{c_1 n^2}{n^2} \leq \frac{\frac{1}{2}n^2 - 3n}{n^2} \leq \frac{c_2 n^2}{n^2}$$

$$\Rightarrow (0 \leq) c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Now, we look for the minimum quantity n_0 that satisfy both the conditions. We operate by looking at the inequalities individually, starting from the **RHS**[3]:

RHS $\frac{1}{2} - \frac{3}{n} \leq c_2$ for $n \to +\infty$ we have that $f(n) \to \frac{1}{2}$ which implies that $c_2 \geq \frac{1}{2}$. We set $c_2 = \frac{1}{2}$ and find the minimum n_0 that satisfy all inequalities. To do so, we focus on the inequality between brackets (given by definition): $0 \leq \frac{1}{2} - \frac{3}{n} \Rightarrow \frac{1}{2} \geq \frac{3}{n} \Rightarrow 2 \leq \frac{n}{3} \Rightarrow n \geq 6$. We could select $n_0 = 6$ but it would invalidate the condition $c_1 > 0$, so we select $n_0 = 7$. Fixed $n_0 = 7$ we see if this value allows us to find a c_1 which satisfies all the conditions of the definition of $\Theta(g(n))$.

LHS $n_0 = 7 \Rightarrow c_1 \leq \frac{1}{2} - \frac{3}{7} = \frac{7-6}{14} \Rightarrow c_1 = \frac{1}{14}$

In case the n_0 isn't sufficiently high to find c_1 we repeat all the passages until all conditions holds. Note that these constants depend on the function $\frac{1}{2}n^2 - 3n$; a different function belonging to $\Theta(n^2)$ would usually require different constants.

Example:

We can also use the formal definition to verify that $6n^3 \neq \Theta\left(n^2\right)$. Suppose for the purpose of contradiction that c_2 and n_0 exist such that $6n^3 \leq c_2 n^2$ for all $n \geq n_0$. But then dividing by n^2 yields $n \leq c_2/6$, which cannot possibly hold for arbitrarily large n, since c_2 is constant (and finite).

Intuitively, the lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large n. When n is large, even a tiny fraction of the highest-order term suffices to dominate the lower-order terms. Thus, setting c_1 to a value that is slightly smaller than the coefficient of the highest-order term and setting c_2 to a value that is slightly larger permits the inequalities in the definition of Θ -notation to be satisfied. The coefficient of the highest-order term can likewise be ignored, since it only changes c_1 and c_2 by a constant factor equal to the coefficient.

[3] **R**ight **H**and **S**ide

Rule of Thumb for polynomials Consider any quadratic function $f(n) = an^2 + bn + c$, where a, b, and c are constants and $a > 0$. Throwing away the lower-order terms and ignoring the constant yields $f(n) = \Theta\left(n^2\right)$. Formally, to show the same thing, we take the constants $c_1 = a/4, c_2 = 7a/4$, and $n_0 = 2 \cdot \max(|b|/a, \sqrt{|c|/a})$. You may verify that $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ for all $n \geq n_0$. This is an example that can be embodied in the general formula:

Formula to keep in mind

In general, for any polynomial $p(n) = \sum_{i=0}^{d} a_i n^i$, where the a_i are constants and $a_d > 0$, we have $p(n) = \Theta\left(n^d\right)$.

Another little abuse of notation Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta\left(n^0\right)$, or $\Theta(1)$. This latter notation is a minor abuse, however, because the expression *does not indicate what variable is tending to infinity*. We shall often use the notation $\Theta(1)$ to mean either a constant or a constant function with respect to some variable. AAoN

3.2 Big-Oh - O

Big-Oh

$$O(g(n)) := \{f(n) : \ 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0, \quad \exists c > 0, n_0 \geq 0\}$$

or equivalently:

$$f(n) \in O(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

If $f(n)$ and $g(n)$ are two algorithms' complexities and $f(n) = O(g(n))$, this means that the algorithm f **can't go slower** than algorithm g (i.e. we use Big-O for **worst case** analisys). In algorithmics, however, when we write $f(n) = O(g(n))$, we are merely claiming that some constant multiple of $g(n)$ is an asymptotic upper bound on $f(n)$ with no claim about how tight an upper bound it is. Distinguishing asymptotic upper bounds from asymptotically tight bounds is standard in the algorithms literature. Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure

3.3 Big-Omega - Ω

Big-Omega

$$\Omega(g(n)) := \{f(n) : \ 0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0, \quad \exists c > 0, n_0 \geq 0\}$$

or equivalently:

$$f(n) \in \Omega(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

If $f(n)$ and $g(n)$ are two algorithms' complexities and $f(n) = O(g(n))$, this means that the algorithm f **can't go faster** than algorithm g (i.e. we use Big-Omega for **best case** analisys).
An illustration of the latter three concepts is present for the figure 2.1.

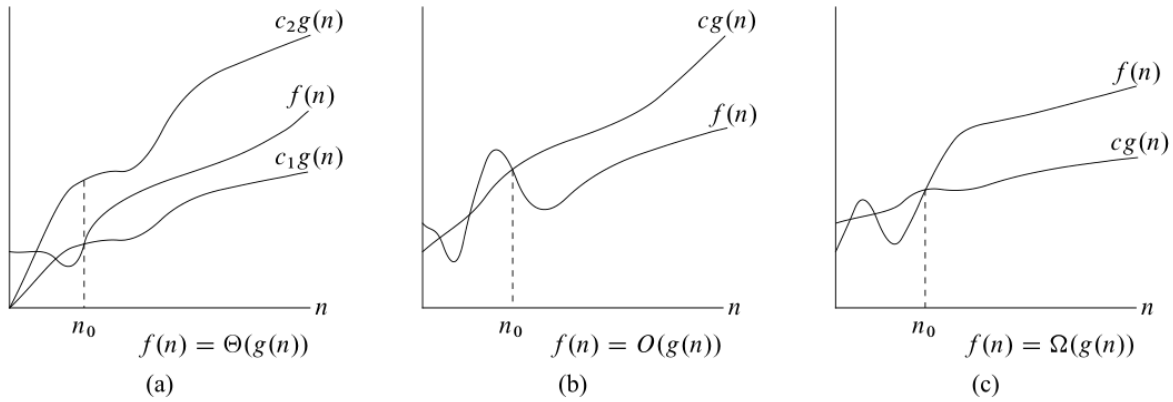


Figure 3.1: Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. (a) Θ -notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. (b) O -notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. (c) Ω -notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

3.4 In-depth analysis of Insertion Sort

This section is thought to be a concise analysis of Insertion Sort using all the theoretical ingredients previously introduced.

A little example of how it works on arrays:

Index $1, 2, 3, 4, 5, 6$
$A = \langle 5, 2, 4, 6, 1, 3 \rangle$
$A = \langle 5, 2, 4, 6, 1, 3 \rangle$
$A = \langle 2, 5, 4, 6, 1, 3 \rangle$
$A = \langle 2, 4, 5, 6, 1, 3 \rangle$
$A = \langle 2, 4, 5, 6, 1, 3 \rangle$
$A = \langle 1, 2, 4, 5, 6, 3 \rangle$
$A = \langle 1, 2, 3, 4, 5, 6 \rangle$
Note that:

1. The leftmost element w.r.t. the current index is always ordered.
2. When we move an element from j to $i < j$ we have to make room for it by shifting all the elements from i to $j - 1$

The pseudo-code is:
Seeing how it works in here: Python Live Programming Mode[4]

Loop Invariant Methodology Is a methodology used in order to ensure the correctness of an *algorithm's loop*[5]:

[4] http://pythontutor.com/live.html#mode=edit for the people printing it
[5] This is done in order to asses whether the loop has or not the Loop Invariant Property

Algorithm 4 INSERTION SORT

```
1: for j = 2 to length(A) do
2:     key=A[j]
3:     // insert A[j] into the
4:     // sorted sequence A[1..j − 1]
5:     i = j − 1
6:     while i > 0 and A[i] > key do
7:         A[i + 1] = A[i]
8:         i = i − 1
9:     A[i + 1] = key
```

1. for every instance, that is consistent with the input specification of the algorithm;
2. for every input of size n;
3. at each iteration of it;

It consists into state a *condition* C (proposition) on a loop (and its corrispondent piece of code, or, part of the algorithm) that has to be consistent and correct for any input size and for any kind of instance.

In order to do that we might execute a procedure that is analogue to the *inductive principle*.

This operation is made through 3 main steps:

1. *Initialization*: C is true prior to the first iteration of the loop.
2. *Maintenance*: If C is true before an iteration of the loop, it remains true before the next iteration.
3. *Termination*: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

In the case of `insertion sort` we need to ensure that $C =$ *"the left sub-array* `A[1,...,j-1]` *is always ordered"*. This invariant property of the external `for` will be assessed for every step wrt the loop.

> We'll refer to the condition with the letter C, we specify it in order to avoid misunderstandings later on.[6]

In our case then the three steps are:

1. *Initialization*: *C is true prior to the first iteration of the loop.*

   Since that at the beginning `j=2`[7], the sub-array `A[1,...,j-1] = A[1]` is *ordered*, because for the ordering definition, a sub-array of dimension k is ordered $\iff$ `A[1]` $\leq$ `A[2]` $\leq \ldots \leq$ `A[k]`, and having one element does not violate the condition.

2. *Maintenance*: *If it is true before an iteration of the loop, it remains true before the next iteration.*

[6] The Cormen, in the condition, mentions also the fact that the left sub-array elements `A[1,...,j]` are also the same at each iteration (so for any `j`). The text: "At the start of each iteration of the for loop of lines 1–8, the subarray `A[1,...,j-1]` consists of the elements originally in `A[1,...,j-1]`, but in sorted order."

[7] In a 1-based indexing framework

Given that in every iteration we consider the element `A[j]` and the we look for positioning it inside the left sub-array `A[1,...,j-1]`, potentially moving every element inside the latter one step on the right. So, in this case the elements in `A[1,...,j]` are the same elements that we had before the loop (even if potentially disposed in an unordered manner, meaning that the value of the `j` element could be less that the `j-1` element[8]), because all the elements (in the worst case) are shifted in the `A[2,...,j]` sub-array, and the considered element (that we saved in `key`) is placed in `A[1]`. So the elements in `A[1,...,j]` are the same and after the loop are ordered.

3. *Termination*: *When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.*

   Finally, we examine what happens when the loop terminates. The condition causing the for loop to terminate is that $j >$ `length(A)` $= n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the sub-array `A[1, ..., n]` consists of the elements originally in `A[1, ..., n]`, but in sorted order. Observing that the sub-array `A[1, ..., n]` is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

Analysis of the Running Time: T(n) Here we execute the analysis of the running time.

We firstly introduce the following elements in order to evaluate the running time of the algorithm:

- COST: it represent the cost of the single operation (it depends on the type of it);
- TIMES: it represent the number of times that a specific operation is repeated;

Let's assume that each single operation in this pseudo-code has a different cost c_i.

INSERTION-SORT(A)	COST	TIMES
1. for j = 2 to length(A) do	c_1	n
2. key = A[j]	c_2	$n-1$
3. // insert A[j] into the	c_2	$n-1$
4. // sorted sequence A[1..j-1]	c_3	$n-1$
5. i = j - 1	c_4	$n-1$
6. while i>0 and A[i]>key do	c_5	$\sum_{j=2}^{n} t_j$
7. A[i+1] = A[i]	c_6	$(\sum_{j=2}^{n} t_j) - 1$
8. i = i - 1	c_7	$(\sum_{j=2}^{n} t_j) - 1$
9. A[i+1] = key	c_8	$n-1$

MOTIVATIONS:

1. Since the `for` cycle is repeated $(n-2)+1 = n-1$ times from 2 to n, plus 1 time due to the checking of the condition of the header (when `j`$= n+1$ and the condition is not true anymore). So $(n-1)+1 = n$
2. This operation is done $n-1$ times, 'cause we enter in the for loop $n-1$ times.

[8] Notice that this is not important, we are trying to understand if the maintenance property of the loop is granted or not, so we don't care about the ordering.

3. It's executed $n-1$ times, but since is not read by the interpreter (i.e. not executable code), its cost is c_3 =0.

4. It's executed $n-1$ times, but since is not read by the interpreter (i.e. not executable code), its cost is c_4 =0.

5. $n-1$, since we're still inside the for code, every piece of code in the same nest is executed the same number of times.

6. $n-1$

7. it depends on the instance,

$$\sum_{j=2}^{n} t_j$$

where t_j is the number of times that the test of the while is reproduced, notice that the quantity t_j embodies already the condition checking at the end if the while is still `true`.

We say that depends on the instance because we may get, at a certain iteration j that[9]:

- the sub-array `A[1,...,j]` is already ordered (*best-case scenario* - `A[i]=A[j-1]<key`), so $t_j = 1$ (just checking the while condition, that is false);
- the `A[i]>key` $\forall i = 1, \dots, j-1$, then $t_j = j$;

8. $(\sum_{j=2}^{n} t_j) - 1$

9. $(\sum_{j=2}^{n} t_j) - 1$

10. $n-1$

Now, if we evaluate the overall cost of a statement i we compute $c_i \cdot$(`# of times it's executed`).

Assumption of the cost per type of command

So even if we know that the cost depends on the type of statement[a] (and it's possible to obtain it, for precise estimates), indeed in the following we **assume the unitary cost for all elementary operations**:

$$c_1 = c_2 = c_5 = \cdots = c_8 = 1 \quad (*)$$

and $c_3 = c_4 = 0$ (for the **comment**).

[a] These costs in practice depend on the kind of machine

Running Time Now we recall that, from the previous lecture:

- we said that every step in an algorithm is **basic**, being a basic step means that they have minimum cost.
- also that the running time depends on the instance, and the input size. In particular t_j varies accordingly to the input, as we already showed.

The latter point suggest that an algorithm may behave differently on the basis of the input type which is exposed to.
How do we evaluate the performance of an algorithm, if it varies accordingly to the type of input?

[9] Just looking at the extremal cases

Running Time Scenarios

The analyses are based upon three scenarios that we may encounter in reality:

- Worst-Case Analysis: In this scenario the input is completely inverted, i.e. $a_n \leq a_{n-1} \leq \cdots \leq a_1$ a.k.a. *inverse sequence*:

  $$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_5 \cdot (n-1) + c_6 \cdot \sum_{j=2}^{n} t_j + c_7 \cdot [(\sum_{j=2}^{n} t_j) - 1] + c_8 \cdot [(\sum_{j=2}^{n} t_j) - 1] + c_9 \cdot (n-1)$$

  and since we enter every time in the while loop $c_6 \cdot \sum_{j=2}^{n} t_j$:

  - At 1^{st} iteration: $1 + 1 = 2$ (`cost_of_first_element_swapping + condition_of_exit`)
  - At 2^{nd} iteration: $2+1 = 3$ (`cost_of_first_and_second_elements_swapping + condition_of_exit`)
  - $\vdots$
  - At last iteration: $n-1+1 = n$ (`cost_of_first_n-1_elements_swapping`[10] `+ condition_of_exit`)

  By this we can state that

  $$c_6 \cdot \sum_{j=2}^{n} t_j = c_6 \cdot \sum_{j=2}^{n} j = 2 + 3 + 4 + \cdots + (n-1) + n = c_6 \cdot [\frac{(n \cdot (n+1))}{2} - 1]$$

  also:

  $$c_7 \cdot [(\sum_{j=2}^{n} t_j) - 1] = c_7 \cdot \sum_{j=2}^{n-1} j = c_7 \cdot [\frac{(n \cdot (n-1))}{2} - 1]$$

  and:

  $$c_8 \cdot [(\sum_{j=2}^{n} t_j) - 1] = c_8 \cdot \sum_{j=2}^{n-1} j = c_8 \cdot [\frac{(n \cdot (n-1))}{2} - 1]$$

  $$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_5 \cdot (n-1) + c_6 \cdot \sum_{j=2}^{n} t_j + c_7 \cdot [(\sum_{j=2}^{n} t_j) - 1] + c_8 \cdot [(\sum_{j=2}^{n} t_j) - 1] + c_9 \cdot (n-1)$$

  $$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_5 \cdot (n-1) + c_6 \cdot [\frac{(n \cdot (n+1))}{2} - 1] + c_7 \cdot [\frac{(n \cdot (n-1))}{2} - 1] +$$
  $$c_8 \cdot [\frac{(n \cdot (n-1))}{2} - 1] + c_9 \cdot (n-1) = \frac{3}{2}n^2 + \frac{5}{2}n - 4 \in O(n^2)$$

- Expected-Case Analysis: We must compute a probability distribution of the input's complexity (not part of the course).

- Best-Case Analysis:

  The scenario is of having an already ordered sequence,

  $$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_5 \cdot (n-1) + c_6 \cdot \sum_{j=2}^{n} t_j + c_7 \cdot [(\sum_{j=2}^{n} t_j) - 1] + c_8 \cdot [(\sum_{j=2}^{n} t_j) - 1] + c_9 \cdot (n-1)$$

  and since we never enter in the while loop, $c_6 \cdot \sum_{j=2}^{n} t_j = c_6 \sum_{j=2}^{n} 1 = c_6(n-2+1) = c_6 \cdot (n-1)$ and $c_7 \cdot [(\sum_{j=2}^{n} t_j) - 1] + c_8 \cdot [(\sum_{j=2}^{n} t_j) - 1] = 0$. Getting:

  $$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_5 \cdot (n-1) + c_6 \cdot (n-1) + c_9 \cdot (n-1)$$

  and by $(*)$:

  $$T(n) = n + n - 1 + n - 1 + n - 1 + n - 1 = 5n - 4$$

[10] Remember that the while loop starts from the $n - 1^{th}$ element of the array

3.4.1 Common orders of growth

To provide a more exhaustive number of examples (with the associated theory) we recap here the most common orders of growth:

NAME	NOTATION	EXAMPLE	CODE FRAGMENT
Constant	$O(1)$	array access arithmetic operation function call	`op();`
Logarithmic	$O(\log n)$	binary search in a sorted array insert in a binary heap search in a red–black tree	`for (int i = 1; i <= n; i = 2*i)` `  op();`
Linear	$O(n)$	sequential search grade-school addition BFPRT median finding	`for (int i = 0; i < n; i++)` `  op();`

Linearithmic	$O(n \log n)$	mergesort
		heapsort
		fast Fourier transform
			`for (int i = 1; i <= n; i++)`
			`  for (int j = i; j <= n; j = 2*j)`
			`  op();`

Quadratic	$O(n^2)$	enumerate all pairs
		insertion sort
		grade-school multiplication
			`for (int i = 0; i < n; i++)`
			`  for (int j = i+1; j < n; j++)`
			`  op();`

Cubic	$O(n^3)$	enumerate all triples
		Floyd–Warshall
		grade-school matrix multiplication
			`for (int i = 0; i < n; i++)`
			`  for (int j = i+1; j < n; j++)`
			`  for (int k = j+1; k < n; k++)`
			`  op();`

Part II

Algorithmic Paradigms

Chapter 4

Recursion

A recursive sequence, from the mathematical point of view, is a sequence of numbers in which early terms are *explicitly* specified and the following ones are expressed as a function of the predecessors. For example:

$$\begin{cases} T_1 = 1 \\ T_{n+1} = 2T_n + 1, \forall n \in \mathbb{N} \end{cases}$$

This mathematical concept can be implemented into the design of algorithms via expressing the output of a sub-problem T_{n+1} as an expression of the T_n's output which, in turn, will depend on the predecessor's output, an so on and so forth.
An algorithm is **recursive** if it calls itself one or more times to solve a problem via storing the intermediate information inside an historical *stack* (an array based on the LIFO principle[1]).

Recursiveness of an algorithm

The recursiveness of an algorithm is based onto the possibility that the solution of a problem can be spontaneously obtained via the solution of the same problem formulated on subsets of data and on the recombination of the results of those to build the result of the original problem.

An example of recursive algorithm to compute a factorial is the following:

Algorithm 5 FACT(x)

```
1: if x == 0 then
2:     return 1
3: else
4:     return x * (FACT(x − 1))
```

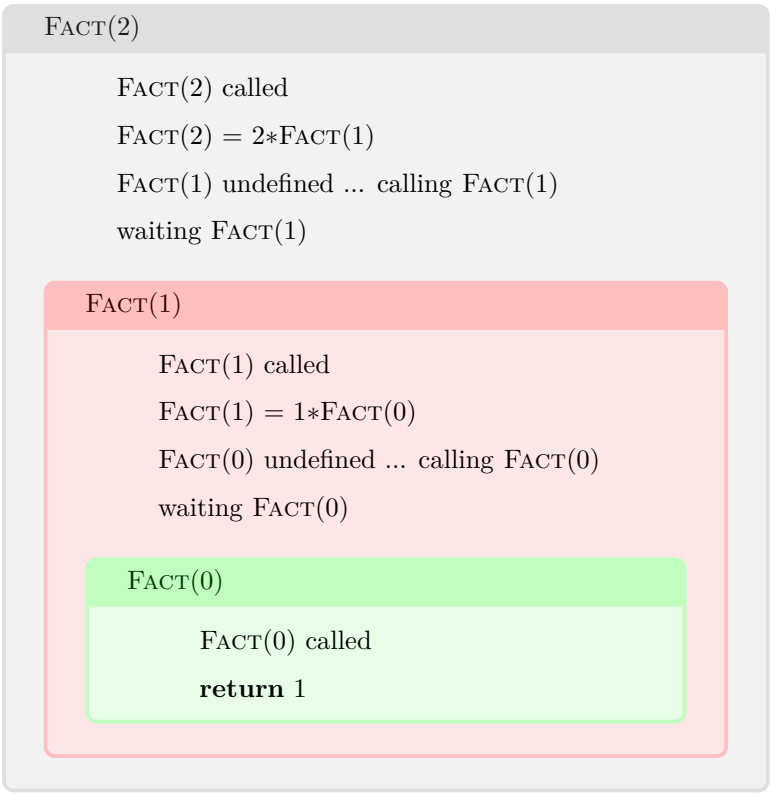
From the operative point of view, when we execute a recursive algorithm, the algorithm calls itself until it reaches the *exit condition* (or *base case*), a value or condition from where the algorithm will starting producing the final output according to the nested intermediate computations.

[1] It's a **ADT**, *Abstract Data Type*. A mathematical model of a data structure that it specifies the type of the data stored, the supported operations, and the parameters type

Here's what an recursive algorithm does when it calls itself in a *nested* manner:

1. The *current execution* is paused
2. The *execution context* associated with the current algorithm execution is stored in a stack (LIFO structure)
3. The *nested call executes*
4. After it ends, the previous call is retrieved from the stack and the execution is resumed from where it stopped

Here's the way in which the algorithm works for the input $x = 2$:



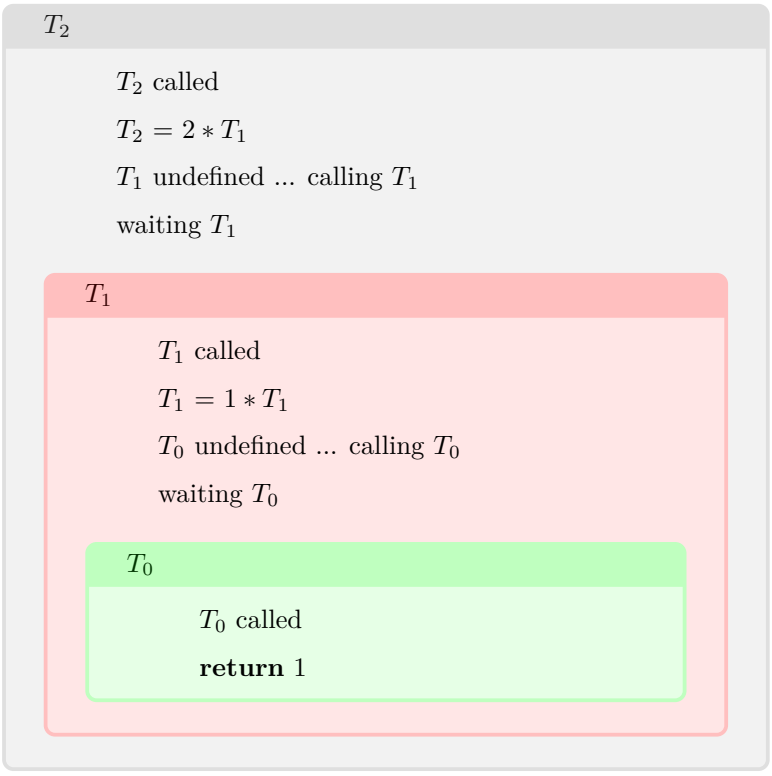
Once the algorithm reaches the *exit condition* "**return** 1" it starts to generate the output by feeding backwards the intermediate instructions of the previous calls until the first call, all saved in the *contextual stack*. Notice the analogy with the mathematical recursion.

$$\begin{cases} T_0 = 1 \\ T_n = n * T_{n-1}, \forall n \in \mathbb{N} \end{cases}$$

So:

$$T_2 = 2 * T_1 = 2 * \underbrace{(1 * T_0)}_{T_1} = 2 * 1 * 1$$

Here's a comparison, in mathematical terms, of what a recursive algorithm does:



An important aspect of recursive algorithms is that they tend to occupy a lot of space wrt their iterative counterpart and, since every recursive algorithm has an iterative version (sometimes hard to extract or derive), you can choose a preferred version on the basis of the operative context.

4.1 Divide and Conquer Paradigm

In the algorithmic world, Recursion is mostly powerful when is used on a problem that can be split into equal, but smaller problems[2]. Those algorithms typically follow a **divide-and-conquer** strategy:

- **Divide** the problem into smaller sub-tasks until reached the minimum solvable size of a sub-problem for the algorithm (*decomposition* in CT[3]);
- **Conquer** the sub-problems by solving them one at a time;
- **Combine** all the solutions of the sub-problems into the solution of the general problem (*composition* in CT).

A classical algorithm employing the divide-and-conquer strategy is the very famous **MergeSort**.

[2]With this we mean that it's more efficient on those kind of problems, but, as a principle, Divide Combine can be applied also to problems that are splitted in unequal size sub-problems

[3]Computationa Thinking

4.1.1 MergeSort - $\Theta(n \log(n))$

Intuition: the MergeSort algorithm follows the following logic:

Assume to have an un-ordered list of n elements;

- *Divide* the sequence into two sequences of length $n/2$;
- *Conquer*: sort the two sequences calling MergeSort recursively;
- *Combine*: merge the two ordered sub-sequences into the ordered output sequence.

Clearly, the key (and less trivial) passage of this algorithm is the combine step (i.e. the Merge algorithm).

Algorithm 6 MergeSort(A, p, r)

```
1: if p < r then
2:     q = ⌊(p + r)/2⌋
3:     MergeSort(A, p, q)
4:     MergeSort(A, q + 1, r)
5:     Merge(A, p, q, r)
```

Algorithm 7 Merge(A, p, q, r)

```
1: n_1 = q − p + 1
2: n_2 = r − q
3: let L[1, ..., n_1 + 1 and R[1, ..., n_2 + 1] be new arrays
4: for i = 1 to n_1 do
5:     L[i] = A[p + i − 1]
6: for j = 1 to n_2 do
7:     R[j] = A[q + j]
8: L[n_1 + 1] = ∞
9: R[n_2 + 1] = ∞
10: i = 1
11: j = 1
12: for k = p to r do
13:     if L[i] ≤ R[j] then
14:         A[k] = L[i]
15:         i = i + 1
16:     else
17:         A[k] = R[j]
18:         j = j + 1
```

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.
The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure Merge(A, p, q, r), where A is an array and p, q, and r are indices into the array such that $p \leq q < r$. The procedure assumes that the subarrays $A[p \ldots q]$ and $A[q + 1 \ldots r]$ are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray $A[p \ldots r]$
Our Merge procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged.

Example of how MergeSort works on $A = [1, 3, 2]$:

MergeSort$(A = [1, 3, 2], p = 1, r = 3)$ - MergeSort$(A, 1, 3)$

if $p < r$ **then** (**True**)

$q = \lfloor (p + r)/2 \rfloor \quad (= 2)$

MergeSort$(A = [1, 3, 2], p = 1, q = 2)^*$ - MergeSort$(A, 1, 2)$

if $p < r$ **then** (**True**)

$q = \lfloor (p + r)/2 \rfloor \quad (= 1)$

MergeSort$(A = [1, 3, 2], p = 1, q = 1)$ - MergeSort$(A, 1, 1)$

if $p < r$ **then** (**False**)

exit

MergeSort$(A = [1, 3, 2], q + 1 = 2, r = 2)$ - MergeSort$(A, 2, 2)$

if $p < r$ **then** (**False**)

exit

Merge$(A, 1, 1, 2)$

$\rightarrow A = [1, 3]$

MergeSort$(A = [1, 3, 2], q + 1 = 3, r = 3)$ - MergeSort$(A, 3, 3)$

if $p < r$ **then** (**False**)

exit

Merge$(A, 2, 3)$

$\rightarrow A = [1, 2, 3]$

$*$ Here we put q to ease the comprehension of the link between the initial recursive passages. Just keep in mind that the q that we wrote here is like an r for the next recursive calls, reason why we used r in the condition below, to maintain the notation of the book. Our suggestion is to do this exercise on paper to get straight to the point ($\approx$ 2 minutes).

The Merge algorithm has $O(n)$ complexity! But how can we compute the overall complexity of MergeSort? We can't compute the complexity of recursive algorithms with the technique we used up to now. We will need **recurrences**.

4.2 Recurrences

Recurrences are one way to express the running time of recursive algorithms, in particular of those using the divide and conquer strategy. Recurrences describes the complexity of the algorithm in terms of its smaller subcalls. The general form of a recurrence is the following:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c, \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Where:

- a is the number of parts in which we divide the input at every recursive call;
- b is the factor by which the size of the subproblem is reduced;
- $D(n)$ is the cost of the *divide* step;
- $C(n)$ is the cost of the *combine* step.

So, for example, in the case of the MergeSort algorithm we have the following recurrence:

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

Where we removed $D(n)$ since it's constant. Note that, we can also have more peculiar cases where we don't split the input in a symmetric way. For example we can have $T(n) = T(\frac{2}{3}n) + T(\frac{n}{3}) + f(n)$

4.3 Ways to solve Recurrences

We can solve recurrences in two[4] main ways: with **Recurrence Trees** and with the **Master Theorem** (we'll not consider the substitution method, but we state the concept behind it):

- In the **substitution method**, we guess a bound and then use mathematical induction to prove our guess correct.
- The **recursion-tree method** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- The **master method** provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$, and $f(n)$ is a given function. Such recurrences arise frequently. A recurrence of this characterizes a divide-and-conquer algorithm that creates a sub-problems, each of which is $1/b$ the size of the original problem, and in which the divide and combine steps together take $f(n)$ time.

[4] Actually there are another 2 methods that we did not face here. The *substitution* method and the *Plug Chug* one. If you want to know more please look at the Cormen [1] for the substitution method and to "*Mathematics for Computer Science*" by *Eric Lehman* for the PC one

4.3.1 Substitution Method

Here's how it works briefly with an example.

The substitution method for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values; hence the name "substitution method." This method is powerful, but we must be able to guess the form of the answer in order to apply it. We can use the substitution method to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

which is similar to recurrences (4.3) and (4.4). We guess that the solution is $T(n) = O(n \lg n)$. The substitution method requires us to prove that $T(n) \leq cn \lg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \end{aligned}$$

How to apply the Substitution Method

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

The **trick** is to take the argument of the recurrence term (e.g. $\lfloor n/2 \rfloor$) and substitute it with the n term(s) in the proposed solution (e.g. if the proposed solution is $n \log(n)$ given $T(n) = 3T(\lfloor n/2 \rfloor) + \Theta(n^2)$ then $\rightarrow T(n) \leq 3d(\lfloor n/2 \rfloor) \log(\lfloor n/2 \rfloor) + cn^2$, where $d, c > 0$

4.3.2 Recurrence Trees

We will see recurrence trees in action to solve the cost of the MergeSort algorithm. We firstly recall the MergeSort recurrence:

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

Let's substitute some terms:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.[5]
For convenience, we assume that n *is an exact power of* 2. Part (a) of the figure 4.1 shows $T(n)$, which we expand in part (b) into an equivalent tree representing the recurrence. The cn term is the root (the cost incurred at the top level of recursion), and the two subtrees of the root are the two smaller recurrences $T(n/2)$. Part (c) shows this process carried one step further by expanding $T(n/2)$. The cost incurred at each of the two subnodes at the second level of recursion is $cn/2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of c. Part (d) shows the resulting recursion tree.
Next, we add the costs across each level of the tree. The top level has total cost cn, the next level down has total cost $c(n/2) + c(n/2) = cn$, the level after that has total cost $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, and so on. In general, the level i below the top has 2^i nodes, each contributing a cost of $c\left(n/2^i\right)$, so that the i th level below the top has total cost $2^i c\left(n/2^i\right) = cn$. The bottom level has n nodes, each contributing a cost of c, for a total cost of cn.
The total number of levels of the recursion tree in Figure 2.5 is $\lg n + 1$, where n is the number of leaves, corresponding to the input size. An informal inductive argument justifies this claim. The base case occurs when $n = 1$, in which case the tree has only one level. Since $\lg 1 = 0$, we have that $\lg n + 1$ gives the correct number of levels. Now assume as an inductive hypothesis that the number of levels of a recursion tree with 2^i leaves is $\lg 2^i + 1 = i + 1$ (since for any value of i, we have that $\lg 2^i = i$). Because we are assuming that the input size is a power of 2, the next input size to consider is 2^{i+1}. A tree with $n = 2^{i+1}$ leaves has one more level than a tree with 2^i leaves, and so the total number of levels is $(i + 1) + 1 = \lg 2^{i+1} + 1$ To compute the total cost represented by the recurrence (2.2), we simply add up the costs of all the levels. The recursion tree has $\lg n + 1$ levels, each costing cn for a total cost of $cn(\lg n + 1) = cn \lg n + cn$. Ignoring the low-order term and the constant c gives the desired result of $\Theta(n \lg n)$.

Formulas to keep in mind: Tree's Features

Given a recurrence of the following type:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c, \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{otherwise} \end{cases}$$

we can compute the following quantities easily:

- $\#Levels = \log_b(n) + 1$, the first $\log_b(n)$ levels (from $level = 0, \ldots, \log_b(n) - 1$) define the number of levels of the *internal part of the tree*, i.e. the number of levels that exclude the leaves;
- *Tree height* $= \log_b(n)$;
- $\#Leaves = a^{\log_b(n)}$, which determines the cost for the last level of the tree;

A tree with one node (**root**) has 1 level and 0 height.

Note that this is the fastest sorting algorithm we found!

[5] It is unlikely that the same constant exactly represents both the time to solve problems of size 1 and the time per array element of the divide and combine steps. We can get around this problem by letting c be the larger of these times and understanding that our recurrence gives an upper bound on the running time, or by letting c be the lesser of these times and understanding that our recurrence gives a lower bound on the running time. Both bounds are on the order of $n \lg n$ and, taken together, give a $\Theta(n \lg n)$ running time.

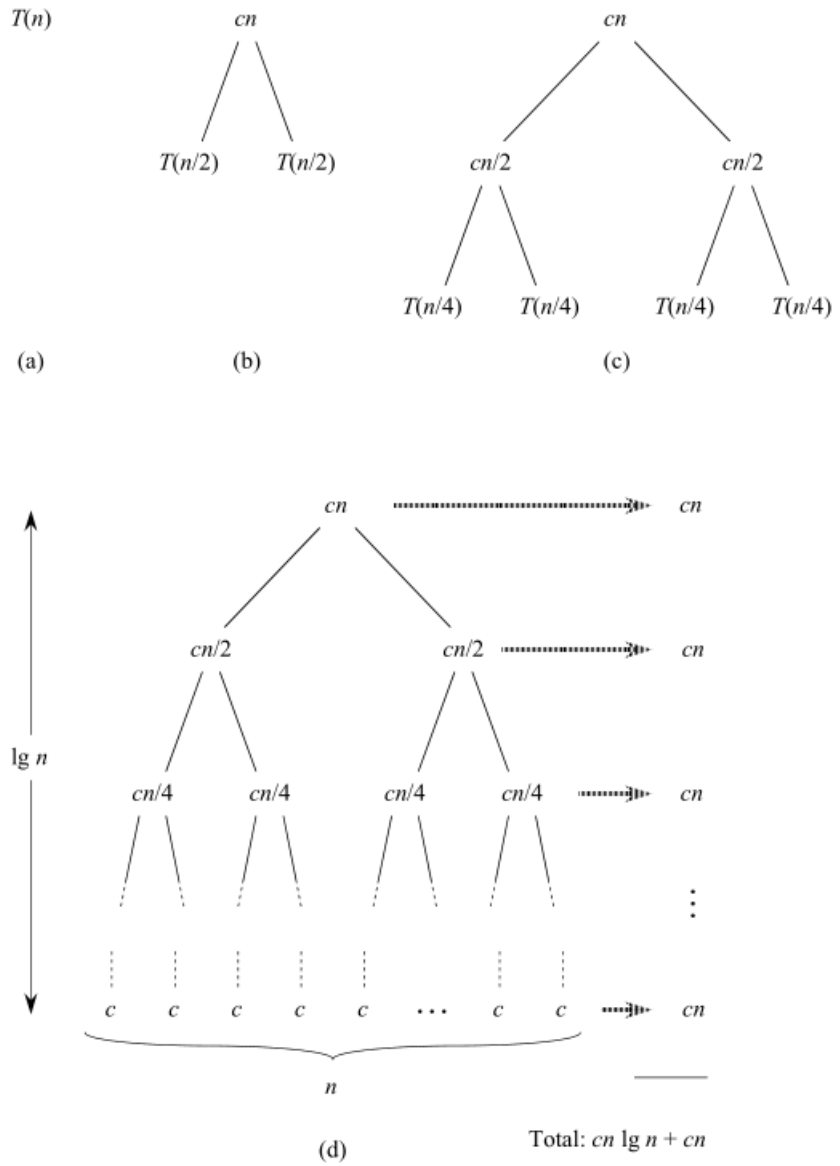


Figure 4.1: Recurrence tree for the MERGE Sort algorithm. Part (a) shows $T(n)$, which progressively expands in (b)-(d) to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn.

Example

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta\left(n^2\right).$$

We start by focusing on finding an *upper bound* for the solution. Because we know that floors and ceilings usually do not matter when solving recurrences (here's an example of sloppiness that we can tolerate), we create a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$, having written out the implied constant coefficient $c > 0$.

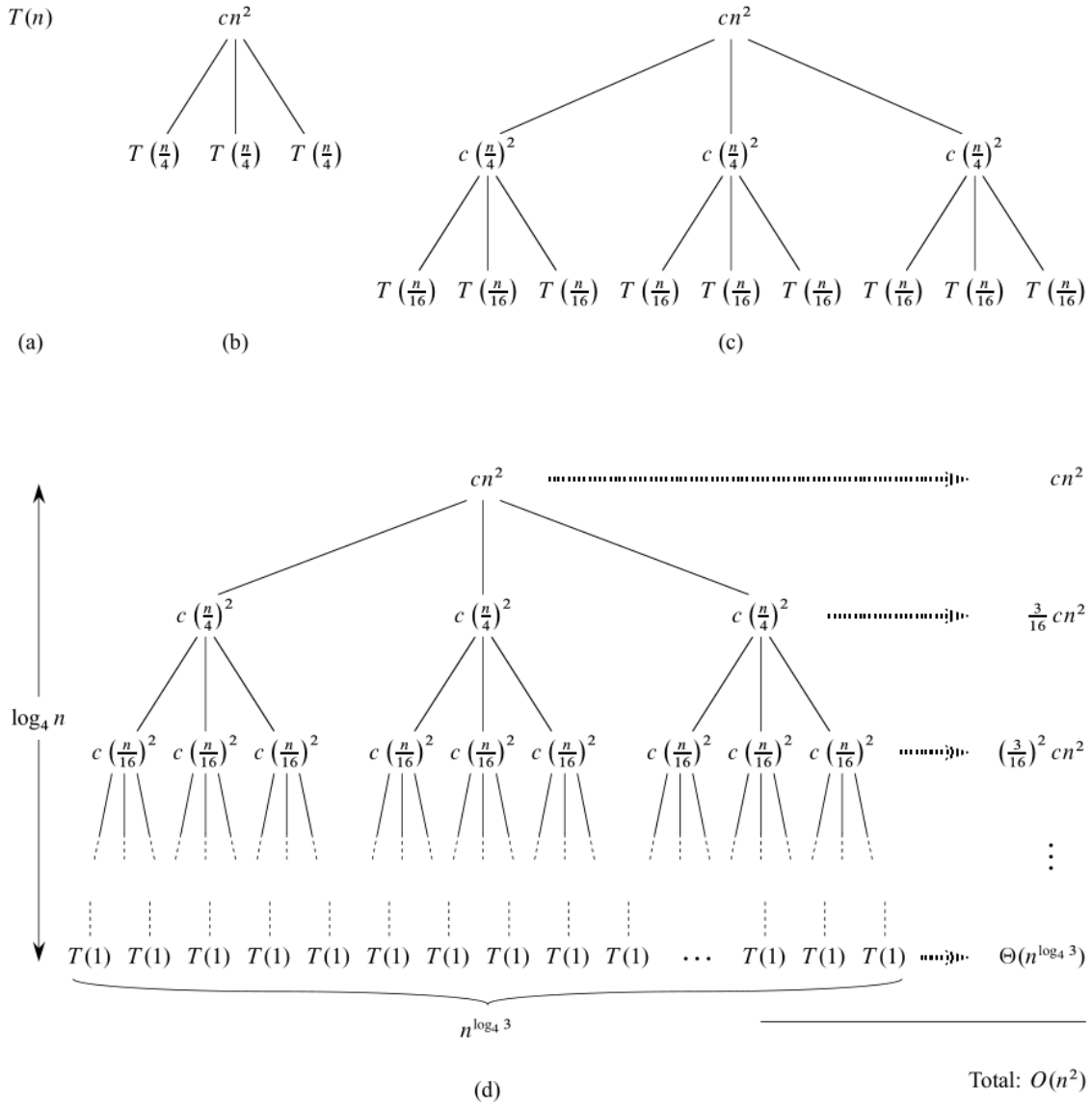


Figure 4.2: Recursion tree of $T(n) = 3T(n/4) + \Theta\left(n^2\right)$.

Figure 4.2 shows how we derive the recursion tree for $T(n) = 3T(n/4) + cn^2$. For convenience, we assume that n is an exact power of 4 (another example of tolerable sloppiness) so that all subproblem sizes are integers. Part (a) of the figure shows $T(n)$, which we expand in part (b) into an equivalent tree representing the recurrence. The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the subproblems of size $n/4$. Part (c) shows this process carried one step further by expanding each node with cost $T(n/4)$ from part (b). The cost for each of the three children of the root is $c(n/4)^2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.

Because subproblem sizes decrease by a factor of 4 each time we go down one level, we eventually must reach a **boundary condition**.

How far from the root do we reach one?

The subproblem size for a node at depth i is $n/4^i$. Thus, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$).

Next we determine the cost at each level of the tree.

Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i. Because subproblem sizes reduce by a factor of 4 for each level we go down from the root, each node at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, has a cost of $c\left(n/4^i\right)^2$. Multiplying, we see that the total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, is $3^i c\left(n/4^i\right)^2 = (3/16)^i cn^2$. The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3} T(1)$, which is $\Theta\left(n^{\log_4 3}\right)$, since we assume that $T(1)$ is a constant.

Now we add up the costs over all levels to determine the cost for the entire tree:

$$
\begin{aligned}
T(n) &= \underbrace{cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2}_{\text{internal levels: from } 0, \ldots, log_4(n) - 1} + \underbrace{\Theta\left(n^{\log_4 3}\right)}_{\text{leaves}} \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta\left(n^{\log_4 3}\right) \qquad \text{(Using Geometric Series' Properties - Equation 3.2)}
\end{aligned}
$$

This last formula looks somewhat messy until we realize that we can again take advantage of small amounts of sloppiness and use an infinite decreasing geometric series as an upper bound. Backing up one step and applying equation (A.6), we have

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) \\
&= \frac{1}{1 - (3/16)} cn^2 + \Theta\left(n^{\log_4 3}\right) \\
&= \frac{16}{13} cn^2 + \Theta\left(n^{\log_4 3}\right) \\
&= O\left(n^2\right)
\end{aligned}
$$

Thus, we have derived a guess of $T(n) = O\left(n^2\right)$ for our original recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta\left(n^2\right)$. In this example, the coefficients of cn^2 form a decreasing geometric series and, by equation (3.3), the sum of these coefficients is bounded from above by the constant $16/13$. Since the root's contribution to the total cost is cn^2, the root contributes a constant fraction of the total cost. In other words, the cost of the root dominates the total cost of the tree.

In fact, if $O\left(n^2\right)$ is indeed an upper bound for the recurrence (as we shall verify in a moment), then it must be a tight bound. Why? The first recursive call contributes a cost of $\Theta\left(n^2\right)$, and so $\Omega\left(n^2\right)$ must be a lower bound for the recurrence.

Now we can use the substitution method to verify that our guess was correct[6], that is, $T(n) = O\left(n^2\right)$ is an upper bound for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta\left(n^2\right)$. We want to show that $T(n) \leq dn^2$ for some constant $d > 0$ Using the same constant $c > 0$ as before, we have

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2 \end{aligned}$$

where the last step holds as long as $d \geq (16/13)c$.

Formulas to keep in mind: Geometric series

For real $x \neq 1$, the summation

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n \tag{4.1}$$

is a geometric or exponential series and has the value

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \tag{4.2}$$

When the summation is *infinite* and $|x| < 1$, we have the infinite decreasing geometric series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \tag{4.3}$$

Because we assume that $0^0 = 1$, these formulas apply even when $x = 0$.

[6] In reality we're checking whether our approximations led to a correct solution

How to solve Recurrencies

1. **Understand the pattern of the cost as the level increases**. A way to understand the pattern is to expand the recurrence in mathematical terms, e.g.

$$T(n) = 3T(n/4) + \Theta\left(n^2\right) = cn^2 + 3\underbrace{(c(n/4)^2 + 3T(n/16))}_{T(n/4)}$$

$$= cn^2 + 3\underbrace{(c(n/4)^2 + 3\overbrace{(c(n/16)^2 + 3T(n/64))}^{T(n/16)})}_{T(n/4)}$$

$$= cn^2 + 3\underbrace{(c(n/4)^2 + 3\overbrace{(c(n/16)^2 + 3\underbrace{(c(n/64)^2 + 3T(n/256))}_{T(n/64)})}^{T(n/16)})}_{T(n/4)}$$

   we express the b at each recursive call as a power of 4:

$$= cn^2 + 3\underbrace{(c(n/4)^2 + 3\overbrace{(c(n/4^2)^2 + 3\underbrace{(c(n/4^3)^2 + 3T(n/4^4))}_{T(n/64)})}^{T(n/16)})}_{T(n/4)}$$

$$= cn^2 + 3c(n/4)^2 + 9c(n/4^2)^2 + 27c(n/4^3)^2 + 81T(n/4^4)$$

$$= cn^2 + 3c(n/4)^2 + 3^2c(n/4^2)^2 + 3^3c(n/4^3)^2 + 3^4T(n/4^4) = \sum_{i=0}^{3} 3^i c(\frac{n}{4^i})^2 + 3^4T(n/4^4)$$

   From this latter computation it's reasonable to assume that each i^{th} level ($i = 0, \ldots, \log_b(n) - 1$) has cost $3^i c(\frac{n}{4^i})^2$. [a]

2. **Understand the number of needed passages to get unitary input**, i.e. compute the height of the tree. To do so it is common to assume (and also it's a tolerable approximation) to hypothesise that the input of the recurrence is a power of b (the number of new sub-problems per node at each level) and compute $i : \frac{n}{b^i} = 1 \iff i = \log_b(n)$.

3. **Compute the complexity**

$$T(n) \in \underbrace{\sum_{i=0}^{height-1=\log_b(n)-1} \text{cost per level}}_{\text{Internal Levels' Cost}} + \underbrace{\sum_{j=1}^{\#leaves=a^{\log_b(n)}} \Theta(1)}_{ExternalLevel'sCost}$$

   A little note: be careful when specifying the indices of the part of the summation associated with the internal levels of the tree (the ones that start from level 0 to the penultimate, here above), they're a cause for error, usually.

4. Check the solution with the *substitution method*.

Note that:

1. floors $\lfloor \cdot \rfloor$ and ceilings $\lceil \cdot \rceil$ *usually* do not matter when solving recurrences.
2. when a recursion is expressed as a function of 2 or more recursive calls, do compute the complexity with respect the longest possible path from root to leaf.

[a] We've assumed constants $c = 1$. In this case we didn't prove this by *induction*, and we may pay consequences if our summation isn't correct (redoing the exercise by computing more levels or checking for errors, in the worst case we'll should bring out the mathematician inside us). But don't worry, to see if our solution is fine it'll suffice to apply the substitution method.

Example

In another, more intricate, example, Figure 4.3 shows the recursion tree for

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

(Again, we omit floor and ceiling functions for simplicity). As before, we let c represent the constant factor in the $O(n)$ term.

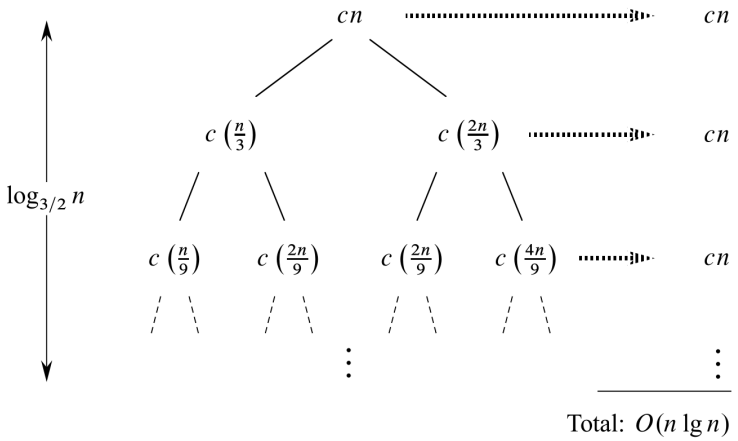


Figure 4.3: Recursion tree of $T(n) = T(n/3) + T(2n/3) + O(n)$

When we add the values across the levels of the recursion tree shown in the figure, we get a value of cn for every level. The longest simple path from the root to a leaf is $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Since $(2/3)^k n = 1$ when $k = \log_{3/2} n$, the height of the tree is $\log_{3/2} n$.

Intuitively, we expect the solution to the recurrence to be at most the number of levels times the cost of each level, or $O\left(cn \log_{3/2} n\right) = O(n \lg n)$. Figure 4.3 shows only the top levels of the recursion tree, however, and not every level in the tree contributes a cost of cn. Consider the cost of the leaves. If this recursion tree were a complete binary tree of height $\log_{3/2} n$, there would be $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ leaves. Since the cost of each leaf is a constant, the total cost of all leaves would then be $\Theta\left(n^{\log_{3/2} 2}\right)$ which, since $\log_{3/2} 2$ is a constant strictly greater than 1 is $\omega(n \lg n)$. *This recursion tree is* ***not*** *a complete binary tree*, however, and so it has fewer than $n^{\log_{3/2} 2}$ leaves.

Moreover, as we go down from the root, more and more internal nodes are absent. Consequently, levels toward the bottom of the recursion tree contribute less than cn to the total cost. We could work out an accurate accounting of all costs, but remember that we are just trying to come up with a guess to use in the substitution method. Let us tolerate the sloppiness and attempt to show that a guess of $O(n \lg n)$ for the upper bound is correct.

Indeed, we can use the substitution method to verify that $O(n \lg n)$ is an upper bound for the solution to the recurrence. We show that $T(n) \leq dn \lg n$, where d is a

suitable positive constant. We have

$$\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + cn \\
&\leq d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&+ (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\leq dn\lg n
\end{aligned}$$

as long as $d \geq c/(\lg 3 - (2/3))$. Thus, we did not need to perform a more accurate accounting of costs in the recursion tree.

4.3.3 Master Theorem

The master method is a cookbook method for solving recurrences. Although it cannot solve all recurrences, it is nevertheless very handy for dealing with many recurrences seen in practice. Suppose you have a recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

where a and b are arbitrary constants and f is some function of n. This recurrence would arise in the analysis of a recursive algorithm that for large inputs of size n breaks the input up into a subproblems each of size n/b, recursively solves the subproblems, then recombines the results. The work to split the problem into subproblems and recombine the results is $f(n)$

We can visualize this as a recurrence tree, where the nodes in the tree have a branching factor of a. The top node has work $f(n)$ associated with it, the next level has work $f(n/b)$ associated with each of a nodes, the next level has work $f\left(n/b^2\right)$ associated with each of a^2 nodes, and so on. At the leaves are the base case corresponding to some $1 \leq n < b$. The tree has $\log_b(n)$ levels, so the total number of leaves is $a^{\log_b n} = n^{\log_b a}$.

The total time taken is just the sum of the time taken at each level. The time taken at the i-th level is $a^i f\left(n/b^i\right)$, and the total time is the sum of this quantity as i ranges from 0 to $\log_b n - 1$, plus the time taken at the leaves, which is constant for each leaf times the number of leaves, or $O\left(n^{\log_b a}\right)$. Thus

$$T(n) = \Sigma_{0 \leq i < \log_b(n)} a^i f\left(n/b^i\right) + O\left(n^{log_b a}\right)$$

What this sum looks like depends on how the asymptotic growth of $f(n)$ compares to the asymptotic growth of the number of leaves. There are three cases:

- **Case 1**: $f(n)$ is $\mathrm{O}\left(n^{\log_b a - \epsilon}\right)$. Since the leaves grow faster than f, asymptotically all of the work is done at the leaves, so $T(n)$ is $\Theta\left(\mathrm{n}^{\log_b a}\right)$.

- **Case 2**: $f(n)$ is $\Theta\left(n^{\log_b a}\right)$. The leaves grow at the same rate as f, so the same order of work is done at every level of the tree. The tree has $O(\log n)$ levels, times the work done on one level, yielding $T(n)$ is $\Theta\left(\mathrm{n}^{\log_b a}\log n\right)$

- **Case 3**: $f(n)$ is $\Omega\left(n^{\log_b a + \epsilon}\right)$. In this case f grows faster than the number of leaves, which means that asymptotically the total amount of work is dominated

by the work done at the root node. For the upper bound, we also need an extra smoothness condition on f in this case, namely that $af(n/b) \leq cf(n)$ for some constant $c < 1$ and large n. In this case $T(n)$ is $\Theta(f(n))$.

Note that in the case 1 (and 3), the $f(n)$ must be **polinomially** smaller (larger) than $n^{\log_b a}$. For this reason, there are "gaps" where the MT can't be applied (i.e. the cases where $\nexists \epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$ or $f(n) = \Omega(n^{\log_b a + \epsilon})$).

As mentioned, the master method does not always apply. For example, the second example considered above, where the subproblem sizes are unequal, is not covered by the master method.

Let's look at a few examples where the master method does apply.

Example:
Consider the recurrence

$$T(n) = 4T(n/2) + n$$

For this recurrence, there are $a = 4$ subproblems, each dividing the input by $b = 2$, and the work done on each call is $f(n) = n$. Thus $n^{\log_b a}$ is n^2, and $f(n)$ is $O\left(n^{2-\epsilon}\right)$ for $\epsilon = 1$, and Case 1 applies. Thus $T(n)$ is $\Theta\left(n^2\right)$

Example:
Consider the recurrence

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

For this recurrence, there are again $a = 4$ subproblems, each dividing the input by $b = 2$, but now the work done on each call is $f(n) = n^2$. Again $n^{\log_b a}$ is n^2, and $f(n)$ is thus $\Theta\left(n^2\right)$, so Case 2 applies. Thus $T(n)$ is $\Theta\left(n^2 \log n\right)$. Note that increasing the work on each recursive call from linear to quadratic has increased the overall asymptotic running time only by a logarithmic factor.

Example:
Consider the recurrence

$$T(n) = 4T(n/2) + n^3$$

For this recurrence, there are again $a = 4$ subproblems, each dividing the input by $b = 2$, but now the work done on each call is $f(n) = n^3$. Again $n^{\log_b a}$ is n^2, and $f(n)$ is thus $\Omega\left(n^{2+\epsilon}\right)$ for $\epsilon = 1$. Moreover, $4(n/2)^3 \leq cn^3$ for $\frac{1}{2} < c < 1$, so Case 3 applies. Thus $T(n)$ is $\Theta\left(n^3\right)$.

Here we display the theorem.

Theorem 1. Master Theorem

Let $T(n) = aT(\frac{n}{b}) + f(n)$ be a recurrence, where $a \geq 1$, $b > 1$ and $f(n)$ is asymptotically positive. Then, $T(n)$ has the following bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon}), \quad \exists \epsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$;
3. If $\begin{cases} f(n) = \Omega(n^{\log_b a + \epsilon}), & \exists \epsilon > 0 \\ af(\frac{n}{b}) \leq cf(n), & \exists c < 1 \text{ (regularity condition)} \end{cases}$
   Then $T(n) = \Theta(f(n))$.

Recurrence relations not solvable by the master method While the master method is very useful in practice, it is worth keeping in mind that there are recurrence relations that it cannot solve. It is always important to verify that the conditions required by the master method hold, as it is possible for a recurrence relation to superficially look to be in the right form, but still violate the requirements or not fall into any of the cases.

Example, $T(n) = 4T(n/2) + n^2/(\log n)$ satisfies all the explicit requirements: we have $a = 2, b = 2$, and $f(n) = n^2/(\log n)$. Nevertheless, it does not fit any of the cases:

- **Case 1**: $n^2/(\log n)$ is not $\mathrm{O}\left(n^{2-\epsilon}\right)$ for any $\epsilon > 0$, since n^ϵ grows faster than $\log n$.

  To check whether our $f(n)$ is *polynomially smaller* than $O(n^{2-\epsilon})$ we must compute $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ where $g(n) = n^{2-\epsilon}$, and check whether this limit converges to 0:

  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{n^2}{\log(n)}}{n^{2-\epsilon}} = \lim_{n\to\infty} \frac{n^\epsilon}{\log(n)} = \infty$$

  by the ∞-hierarchy, $\forall \epsilon > 0$;

- **Case 2**: $n^2/(\log n)$ is not $\Theta(n^2)$, since it's not $O(n^2)$;

- **Case 3**: $n^2/(\log n)$ is not $\Omega\left(n^{2+\epsilon}\right)$ for any $\epsilon > 0$, since $\left(n^{2+\epsilon}\right)/(n/(\log n)) = n^\epsilon \log n$

  To check whether our $f(n)$ is *polynomially greater* than $O(n^{2+\epsilon})$ we must compute $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ where $g(n) = n^{2+\epsilon}$, and check whether this limit diverges to ∞:

  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{n^2}{\log(n)}}{n^{2+\epsilon}} = \lim_{n\to\infty} \frac{1}{n^\epsilon \log(n)} = 0$$

  $\forall \epsilon > 0$.

So we actually can't use the master method to solve this recurrence relation. We can, however, still derive an upper bound for this recurrence by using a little trick: we find a similar recurrence that is larger than $T(n)$, analyze the new recurrence using the master method, and use the result as an upper bound for $T(n)$ $T(n) = 2T(n/2) + n/(\log n) \leq 2T(n/2) + n$, so if we call $R(n)$ the function such that $R(n) = 2T(n/2) + n$, we know that $R(n) \geq T(n)$. This is something we can apply the master method to: n is $\Theta(n)$, so $R(n)$ is $\Theta(n \log n)$. Since $T(n) \leq R(n)$, we can conclude that $T(n)$ is $\mathrm{O}(n \log n)$. Note that we only end up with O, not Θ; since we were only able apply the master method indirectly, we could not show a tight bound.

For the remaining cases there's another important result in algorithmics, the *Akra-Bazzi Method*.

4.3.4 An alternative to the Master Method: The Akra-Bazzi Method

The solution to virtually all divide and conquer solutions is given by the amazing Akra-Bazzi formula. Quite simply, the asymptotic solution to the general divideand-conquer recurrence

$$T(n) = \sum_{i=1}^{k} a_i T\left(b_i n\right) + g(n)$$

is

$$T(n) = \Theta\left(n^p\left(1 + \int_1^n \frac{g(u)}{u^{p+1}}du\right)\right)$$

where p satisfies

$$\sum_{i=1}^{k} a_i b_i^p = 1$$

A rarely-troublesome requirement is that the function $g(n)$ must not grow or oscillate too quickly. Specifically, $|g'(n)|$ must be bounded by some polynomial. So, for example, the Akra-Bazzi formula is valid when $g(n) = x^2 \log n$, but not when $g(n) = 2^n$ Let's solve the MergeSort recurrence again, using the Akra-Bazzi formula. First, we find the value p that satisfies

$$2 \cdot (1/2)^p = 1$$

Looks like $p = 1$ does the job. Then we compute the integral:

$$\begin{aligned} T(n) &= \Theta\left(n\left(1 + \int_1^n \frac{u-1}{u^2}du\right)\right) \\ &= \Theta\left(n\left(1 + \left[\log u + \frac{1}{u}\right]_1^n\right)\right) \\ &= \Theta\left(n\left(\log n + \frac{1}{n}\right)\right) \\ &= \Theta(n \log n) \end{aligned}$$

The first step is integration and the second is simplification. We can drop the $1/n$ term in the last step, because the $\log n$ term dominates.

Another example can be the following:

$$T(n) = 2T(n/2) + (8/9)T(3n/4) + n^2$$

Here, $a_1 = 2, b_1 = 1/2, a_2 = 8/9$, and $b_2 = 3/4$. So we find the value p that satisfies

$$2 \cdot (1/2)^p + (8/9)(3/4)^p = 1$$

Equations of this form don't always have closed-form solutions, so you may need to approximate p numerically sometimes. But in this case the solution is simple: $p = 2$. Then we integrate:

$$\begin{aligned} T(n) &= \Theta\left(n^2\left(1 + \int_1^n \frac{u^2}{u^3}du\right)\right) \\ &= \Theta\left(n^2(1 + \log n)\right) \\ &= \Theta\left(n^2 \log n\right) \end{aligned}$$

We now write in a decent way the method for the sake of completeness:

The Akra-Bazzi Method

Given a recurrence of the following type:

$$T(x) = \begin{cases} \Theta(1) & \text{if } 1 \le x \le x_0 \\ \sum_{i=1}^{k} a_i T(b_i x) + f(x) & \text{if } x > x_0 \end{cases}$$

where:

- $x \ge 1$ is a real number,
- x_0 is a constant such that $x_0 \ge 1/b_i$ and $x_0 \ge 1/(1-b_i)$ for $i = 1, 2, \ldots, k$
- a_i is a positive constant for $i = 1, 2, \ldots, k$
- b_i is a constant in the range $0 < b_i < 1$ for $i = 1, 2, \ldots, k$
- $k \ge 1$ is an integer constant, and
- $f(x)$ is a nonnegative function that satisfies the polynomial-growth condition: there exist positive constants c_1 and c_2 such that for all $x \ge 1$, for $i = 1, 2, \ldots, k$, and for all u such that $b_i x \le u \le x$, we have $c_1 f(x) \le f(u) \le c_2 f(x)$. (If $|f'(x)|$ is upper-bounded by some polynomial in x, then $f(x)$ satisfies the polynomial-growth condition. For example, $f(x) = x^{\alpha} \lg^{\beta} x$ satisfies this condition for any real constants α and β.)

If $\exists p \in \mathbb{R} s.t. \sum_{i=1}^{k} a_i b_i^p = 1$ then solution to the recurrence is:

$$T(n) = \Theta\left(x^p \left(1 + \int_1^x \frac{f(u)}{u^{p+1}} du\right)\right)$$

Although the master method does not apply to a recurrence such as $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$, the Akra-Bazzi method does. To solve the recurrence, we first find the unique real number p such that $\sum_{i=1}^{k} a_i b_i^p = 1$, as we've already saw (Such a p always exists). The solution to the recurrence is then

$$T(n) = \Theta\left(x^p \left(1 + \int_1^x \frac{f(u)}{u^{p+1}} du\right)\right)$$

The Akra-Bazzi method can be somewhat difficult to use, but it serves in solving recurrences that model division of the problem into substantially unequally sized subproblems. The master method is simpler to use, but it applies only when subproblem sizes are equal.

Chapter 5

Dynamic Programming

Dynamic Programming is a class of algorithms very similar to Recursive ones: it focuses on breaking the problem in a series of *overlapping subproblems*, and then building up solutions to larger and larger problems. The idea in dynamic programming is to store somewhere intermediate resulsts, for later reuse. Dynamic programming is especially useful where recursive algorithms (for example, divide and conquer paradigm) don't work really well: the typical case is where the same recursive call is made many times, even when the problem was already solved in previous iterations. This leads to extremely inefficient solutions, whereas with dynamic programming we:

- Use a table to store intermediate results;
- Recover the result for an already solved problem;
- Skip several repeated recursive calls.

To know if dynamic programming can be applied to a specific case we have to check:

1. **Simple Subproblems**: we should be able to break the main problem into smaller subproblems sharing the same structure.
2. **Optimal substructure of the subproblems**: the optimal solution of a problem is contained whithin the optimal solution of its subproblems.
3. **Overlapping subproblems**: there exists a place where the same subproblems are solved more than once.

Complexity: the complexity of a dynamic programming algorithm is $\Theta(pq)$, where p is the number of subproblems and q is the cost of the single subproblem (is q always constant? Since every subproblem uses solutions of other smaller subproblems, I would say that is always constant...).
One famous example of dynamic programming application is the solution of the **Rod Cutting Problem**.

5.1 Rod Cutting Problem

The Rod Cutting problem is the following: given a rod of length n inches and a table of prices, determine the maximum revenue obtainable by cutting up the rod an selling the pieces. The rod cuts are integral number of inches, and making a cut is free.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

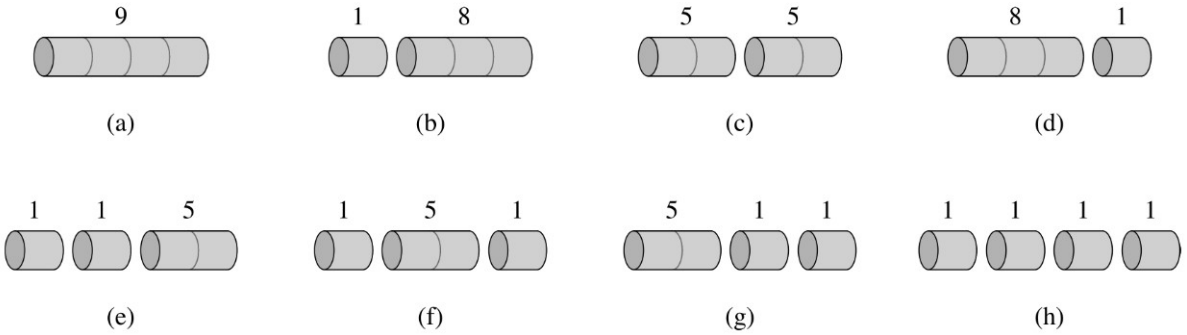


Figure 5.1: For example, there are 8 ways of cutting a rod of length 4

In particular, we can express this problem in the following way:

$$r_n = \max(p_n, r_1 + r_{n-1} + r_2 + r_{n-2} + \cdots + r_{n-1} + r_1) = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Where the r_{n-i} is the recursive component of this problem. If we write it like this, it becomes a little more abstract, but the concept is the following: imagine to have this rope and for every $i = 1, \ldots, n$ you cut the rope in two pieces of length i and $n - i$ and then sum the price of the two obtained parts. Then, pick the best i; but we are not done, because those two parts are now two independent instances of the very same rod cutting problem! We will separately take the obtained pieces and repeat this same procedure until we reached the max revenue. But what happens if we try to solve this in an iterative way?

Algorithm 8 RodCutIterative(p, n)

```
1: if n = 0 then
2:     return 0
3: q = 0
4: for i = 1 to n do
5:     q = max(q, p_i + RodCutIterative(p, n − i))
6: return q
```

Where p is the price list. As we can see, the algorithm calls itself a repeated number of times, to solve the same subproblems repeatedly! So, despite the basic idea being very simple, in reality it's extremely inefficient.

The good news is that we can solve this problem efficiently thanks to Dynamic Programming! The idea is to organize the algorithm in order to solve the subproblems only once, and then store the solutions in an appropriate data structure. We will retrieve the solution of already solved subproblems when we will need them (this is a time-memory trade off!).

5.2 Memoization

There are two main approaches of dynamic programming.

- **Top-down memoization**: we solve the problem in the usual way but storing the solutions of the sub-problems on the way;

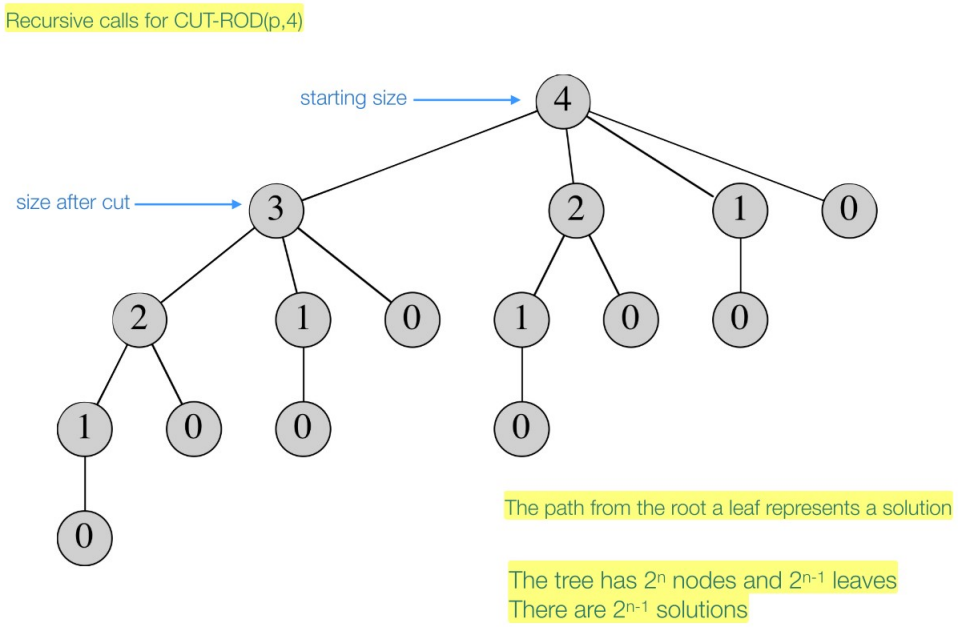


Figure 5.2: We can represent the amount of recursive calls for a rope of length 4. This tree has 2^n nodes, which means exponential complexity!

- **Bottom-up memoization**: defines the sub-problems, orders them by size in increasing order and then proceeds to solve them in such order, using the memorized solutions on the way.

Both methods produce different algorithms with **equivalent** running time.

Algorithm 9 MEMOIZED-CUT-ROD(p, n)

```
1: Let r[0, ..., n] be a new array.
2: for i = 0 to n do
3:     r[i] = −∞
4: return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

5.3 Lowest Common Subsequence Problem

Given two sequences $x = [1, \ldots, m]$ and $y = [1, \ldots, n]$ determine (one possible) longest common subsequence (LCS). With a little abuse of notation we denote such LCS with $LCS(x, y)$.[1] In order to better understand the sense behind the dynamic programming version of the resolution of this algorithm, we see the running time of the **brute force** solution: *check every subsequence of x to see if it is also a subsequence of y.*
Running time:

1. To check if a sequence is a subsequence of y we need to find the first character, and then scan the rest of the sequence $\Rightarrow O(n)$;

2. We have that x contains 2^m subsequences.

Therefore the running time is $O(n2^m)$, which is extremely inefficient for large sequences. In order to better handle the problem we make a **simplification**:

[1]The abuse comes from the fact that we are using a function notation, even though LCS is clearly not a function since there can exist multiple different outputs from the same input.

Algorithm 10 MEMOIZED-CUT-ROD-AUX(p, n, r)

```
1: if n == 0 then
2:     return r[n]
3: if r[n] ≥ 0 then
4:     return q == 0
5: else
6:     q = −∞
7:     for i = 1 to n do
8:         q = max(q, p[i]+MEMOIZED-CUT-ROD-AUX(p, n − i, r))
9: r[n] = q
10: return q
```

Algorithm 11 EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
1:
2: Let r[0, . . . , n] and s[1, . . . , n] be new arrays.
3: r[0] = 0
4: for j = 1 to n do
5:     q = −∞
6:     for i = 1 to j do
7:         if q < p[i] + r[j − i] then
8:             q = p[i] + r[j − i]
9:             s[j] = i
10:    r[j] = q
11: return r and s
```

Algorithm 12 PRINT-CUT-ROD-SOLUTION(p, n)

```
1: (r, s) =EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2: while n > 0 do
3:     print s[n]
4:     n = n − s[n]
```

1. We firstlook for the length $c[x, y]$ of $LCS(x, y)$, not for the actual subsequence itself.

2. We then extend our solution to return the actual LCS.

The strategy is the following: we will consider only prefixes of x and y (e.g. $[1, \ldots, k]$ is a prefix of $[1, \ldots, m]$ when $k < m$), and we will show that the length of the LCS between the prefixes will constitute our optimal subproblems structure.
We define:

$$c[i, j] = |LCS(x[1, \ldots, i], y[1, \ldots, j])|$$

In this way we clearly see that $c[m, n] = |LCS(x, y)|$; furthermore we can express such quantity in terms of other $c[i, j]$. In fact, the key idea behind the solution of the LCS is that:

$$c[i, j] = \begin{cases} 0 \quad \text{if } i = 0 \textit{ or } j = 0; \\ c[i-1, j-1] + 1 \quad \text{if } x[i] = y[j]; \\ \max(c[i, j-1], c[i-1, j]) \quad \text{otherwise.} \end{cases}$$

With this idea we can make a **recursive** (not dynamic) algorithm.

Algorithm 13 RecursiveLCS(x, y, i, j)

```
1: if i = 0 or j = 0 then
2:     return 0
3: if x[i] = y[j] then
4:     c[i, j] = LCS(x, y, i − 1, j − 1) + 1
5: else
6:     c[i, j] = max(LCS(x, y, i − 1, j), LCS(x, y, i, j − 1))
7: return c[i, j] =0
```

This algorithm has a major flaw: in line 6 we have two different recursive calls, leading to a binary recursion tree of depth $m + n$, wich will lead to an exponential complexity of $O(2^{n+m})$. The main reason for this inefficiency is because we are solving multiple times the same subproblems. We know that we can avoid this by using dynamic programming!

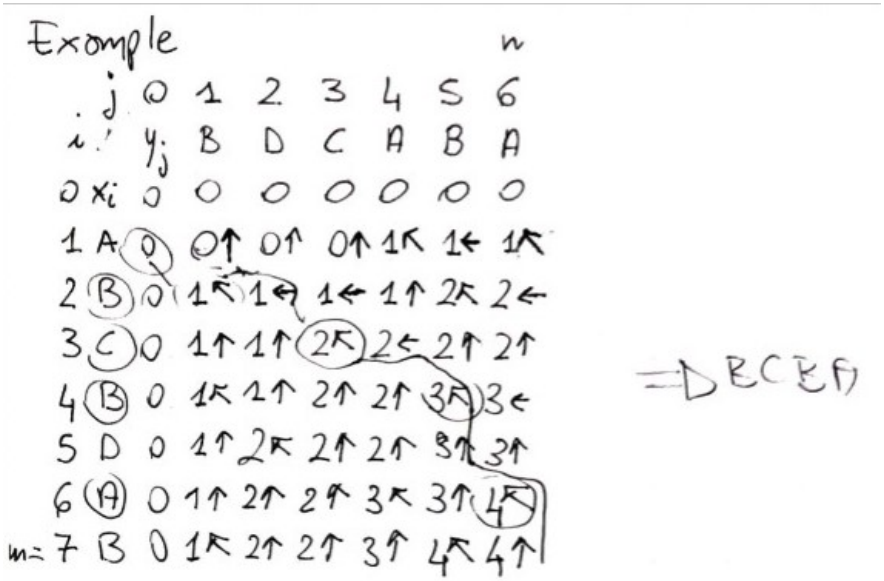
First, we need a way to store the solution to the subproblems: we will use a table (matrix) C with entries $C_{ij} = c[i, j]$. now define a **bottom up** memoized algorithm to find the length of the LCS between x and y.

Algorithm 14 BOTTOM-UP-LCS-LENGTH(x, y, i, j)

```
Let b[1, ..., m, 1, ..., n] and c[0, ..., m, 0, ..., n] be new tables.
for i = 1 to m do
    C[i, 0] = 0
for j = 0 to n do
    C[0, j] = 0
for i = 1 to m do
    for j = 0 to n do
        if x_i == y_i then
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] =↖
        else if c[i − 1, j] ≥ c[i, j − 1] then
            c[i, j] = c[i − 1, j]
            b[i, j] =↑
        else
            c[i, j] = c[i, j − 1]
            b[i, j] =←
return c and b
```

Algorithm 15 PRINT-LCS(b, x, i, j)

```
if i == 0 or j == 0 then
    return NoneL
if b[i, j] ==↖ then
    PRINT-LCS(b, x, i − 1, j − 1)
    print x_i
else if b[i, j] ==↑ then
    PRINT-LCS(b, x, i − 1, j)
else
    PRINT-LCS(b, x, i, j − 1)
```


Exemple
n
j 0 1 2 3 4 5 6
i yj B D C A B A
0 xi 0 0 0 0 0 0 0
1 A 0 0↑ 0↑ 0↑ 1↖ 1← 1↖
2 B 0 1↖ 1← 1← 1↑ 2↖ 2←
3 C 0 1↑ 1↑ 2↖ 2← 2↑ 2↑
4 B 0 1↖ 1↑ 2↑ 2↑ 3↖ 3←
5 D 0 1↑ 2↖ 2↑ 2↑ 3↑ 3↑
6 A 0 1↑ 2↑ 2↑ 3↖ 3↑ 4↖
m=7 B 0 1↖ 2↑ 2↑ 3↑ 4↖ 4↑
=DBCBA

Part III

Data Structures

Chapter 6

Dynamic Sets and Abstract Data types

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time. We call such sets **dynamic**. In a typical implementation of a dynamic set, each element is represented by an object whose attributes can be examined and manipulated if we have a *pointer* to the object. Some kinds of dynamic sets assume that one of the object's attributes is an *identifying key*. If the keys are all different, we can think of the dynamic set as being a *set of key values*. The object may contain *satellite data*, which are carried around in other object attributes but are otherwise unused by the set implementation. It may also have attributes that are manipulated by the set operations; these attributes may contain data or pointers to other objects in the set. Some dynamic sets presuppose that the keys are drawn from a totally ordered set, such as the real numbers, or the set of all words under the usual alphabetic ordering. A total ordering allows us to define the minimum element of the set, for example, or to speak of the next element larger than a given element in a set.

Operations on a Dynamic Set Operations on a dynamic set can be grouped into two categories: **queries**, which simply return information about the set, and **modifying operations**, which change the set. Here is a list of typical operations. Any specific application will usually require only a few of these to be implemented.

- SEARCH(S, k) A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or None if no such element belongs to S.
- INSERT (S, x) A modifying operation that augments the set S with the element pointed to by x. We usually assume that any attributes in element x needed by the set implementation have already been initialized.
- DELETE (S, x) A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)
- MINIMUM(S) A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.
- MAXIMUM(S) A query on a totally ordered set S that returns a pointer to the element of S with the largest key.
- SUCCESSOR(S, x) A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or None if x is the maximum element.

- PREDECESSOR(S, x) A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or None if x is the minimum element.

So making a recap:

Dynamic Sets

Dynamic sets are data structures that support some or all of the following operations.

- SEARCH(S, k) Return x with $x.key = k$.
- INSERT(S, x) Insert x into S.
- DELETE (S, x) Delete x from S.
- MINIMUM(S) Return x minimum $x.key$.
- MAXIMUM(S) Return x maximum $x.key$.
- SUCCESSOR(S, x) Return y next highest key.
- PREDECESSOR(S, x) reutnr y next lowest key.

Where S is a set and x and y are elements, k is a key. NONE is returned if the operation can't be performed.

All these operations and properties define a dynamic set, as we said. We can use the concept of dynamic sets to derive *data structures* which implement some of the operations supported by a dynamic set with the aim to gain some efficiency in terms of implementation of a specific algorithm into a program. These dynamic set data structures are called **Abstract Data Structures** (**ADT**).

Definition of ADT

ADT is a mathematical model of a data structure that it specifies the type of the data stored, the supported operations and the parameters type. This data types are more complex ways - since thet are governed by a specified functioning principle - to store data which rely on simpler structure such as lists or arrays.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an *implementation-independent view*. The process of providing only the essentials and hiding the details is known as **abstraction**. The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented. So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type.

6.1 Linear Data Structures Vs Non-Linear Data Structures

A Data Structure is classified into two categories.

Linear Data Structure

Non-Linear Data Structure

Now let us look at the difference between **Linear** and **Non-Linear** data structures.

Linear ADTs

Data is arranged in a **linear** way when elements are linked one after the other.

- Here, all the data elements can be traversed in one go, but at a time only one element is directly reachable.
- Linear data structures tend to waste memory.
- Linear data structures are easy to implement.
- Array, Queue, Stack, Linked List are linear data structures.

Non-Linear ADTs

Data elements in a **non-linear** data structure are hierarchically related.

- All the data elements cannot be traversed in one go as the nodes are cannot visited sequentially
- Efficient utilization of memory.
- Implementation of non-linear data structures is complex.
- Trees and graphs are non-linear data structures.

Now let us try to get a basic understanding of each of these linear and non-linear data structures. We'll get deeper into each of the following ADT in next chapters.

- An Array is a collection of data items having similar data types.
- A Linked list is a collection of nodes, where each node has a data element and a reference to the next node in the sequence.
- A Stack is a LIFO (Last in First Out) data structure where the element that is added last will be deleted first.
- A Queue is a FIFO (First in First Out) data structure where the element that added is first will be deleted first.
- A Tree is a collection of nodes where these nodes are arranged hierarchically and form a parent-child relationship.
- A Graph is a collection of a finite number of vertices and edges. Edges connect the vertices and represent the relationship between the vertices that connect these vertices.

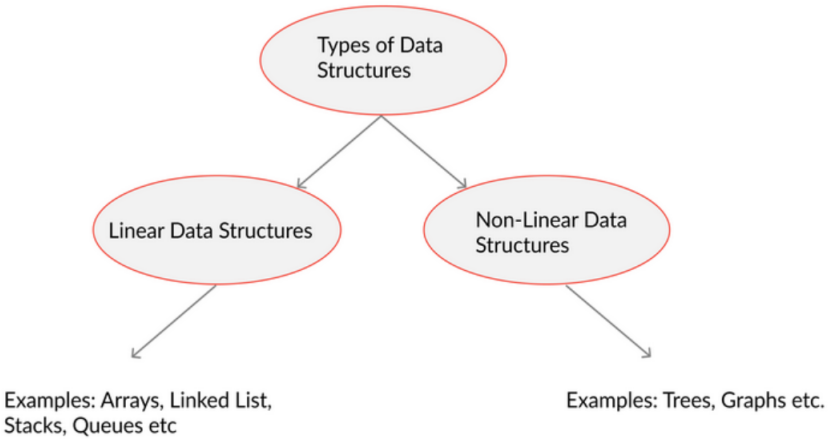


Figure 6.1: Schema of the Linear and Non-linear ADTs

6.2 Lists, Stacks, Queues, Deques

Lists

A **Lists** is a sequential (*contiguous*) collection of values. Each value has a location (an index). Indexes range from 0 to $n-1$. Lists are *heterogeneus*, i.e. values can be of any type (counterexample are strings since they are *homogeneous* because their elements can be only characters). Operations on them are:

- `list.append(x)` Add an item to the end of the list;
- `list.extend(L)` Extend the list by appending all the items in the given list;
- `list.insert(i, x)` Insert an item x at a given position i. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list;
- `list.remove(x)` Remove the first item from the list whose value is x. It is an error if there is no such item;
- `list.pop(i)` Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list;
- `list.index(x)` Return the index in the list of the first item whose value is x. It is an error if there is no such item;
- `list.count(x)` Return the number of times x appears in the list.

Stacks

A **Stack** is a collection of objects (list) following the **LIFO** (Last In First Out) principle. Operations on them are:

- `S.push(e)` add an element on top;
- `S.pop()` Remove and return the top element;
- `S.top()` Return a reference to the top element (index);
- `S.is_empty()` Return true if the stack is empty, false otherwise;
- `len(S)` Return the length of the stack;

Queue

A **Queue** is a collection of objects (list) following the **FIFO** (First In First Out) principle. Operations on them are:

- `Q.enqueue(e)` add an element e to the tail of the queue
- `Q.dequeue()` Remove and return the element in front of the queue
- `Q.first()` Return a reference to the first element (index)
- `Q.is_empty()` Return true if the queue is empty, false otherwise
- `len(Q)` Return the length of the queue

Deque

A **Deque (Deck)** is a double ended queue. It is a queue where you can pop the first or last element, as well as adding an element to the front or to the back. Operations on them are:

- `Q.add_first(e)` add an element e to the front of the deque
- `Q.add_last(e)` add an element e to the back of the deque
- `Q.delete_first()` Return and remove the head (first element) of the deque
- `Q.delete_last()` Return and remove the tail (last element) of the deque
- `Q.first(), Q.last(), Q.is_empty(), len(Q)` Same as before14

To implement a Deque ADT in `Python` we need to use `Collections` module. Just remember that in a deque `append()` adds to the *back* and that `popleft()` removes from the *head*. Also that given a deque D in `Python`, you can get an element at index i as $D[i]$, but indexed access is $O(1)$ *at both ends* but slows to linear time $O(n)$ in the middle elements. For fast random access, use *lists* instead.

Keep in mind that every ADT has useful properties to exploit depending on the domain of application, it's important to choose the correct ADT when needed. To acquire some familiarity with the running times for each operation of the main `Python`'s implementation of some of these data structures (types) visit this site[1].

[1] https://wiki.python.org/moin/TimeComplexity for who prints this document

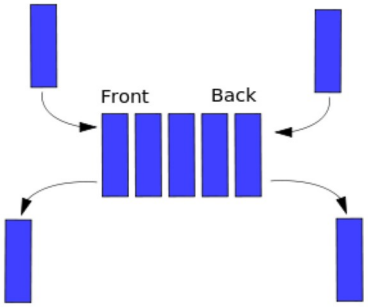


Figure 6.2: A Deque

6.3 *Singly* Linked Lists

Linked Lists a data structure where elements are arranged in linear order, but differently from lists elements are not stored at contiguous memory locations. Also, unlike arrays, where the order is maintained by indexes, in linked lists the order is maintained by means of a **pointer** in each object (we **can't** use indexes to access elements of a linked list).

Keep in mind that...

There's a substantial difference between ADT (*abstract level* of description) and their implementation on a specific programming language (*concrete level*). So even if here we're describing the LL data structure it does not mean that in every programming language will possess the same properties that here we claim. In python, for example LL do have indexes, even if they should not, according to the LL ADT.

Linked Lists (LL) are a collection of Nodes, which havebtwo attributes: the stored value and a pointer to the next element of the LL. Note that every LL has two special nodes: one is called **head** and is the first node, i.e. the one from where we start to **traverse** the list. For this reason every LL must maintain a pointer to the head, or there would be no way to locate that node). The other is called **tail**: is the last node in the LL and it points to nothing (null pointer). A drawback of these ADT is that in order to access to an element we need to traverse the list until we find it, and we cannot traverse the LL in both directions, since the pointers do point to only one global direction.

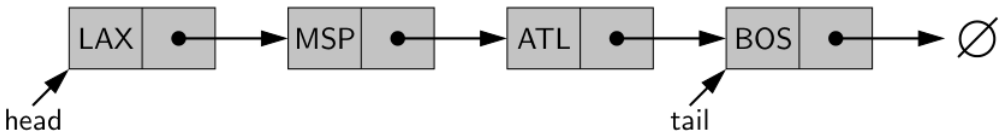


Figure 6.3: A (*singly*) Linked List

6.3.1 Operations on Linked Lists

Adding a node An important property of a linked list is that it does not have a predetermined fixed size; it uses space proportionally to its current number of elements. When using a singly linked list, we can easily insert an element at the head of the list, as shown in Figure 6.4. The main idea is that we create a new node, set its element to

the new element, set its next link to refer to the current head, and then set the list's head to point to the new node.

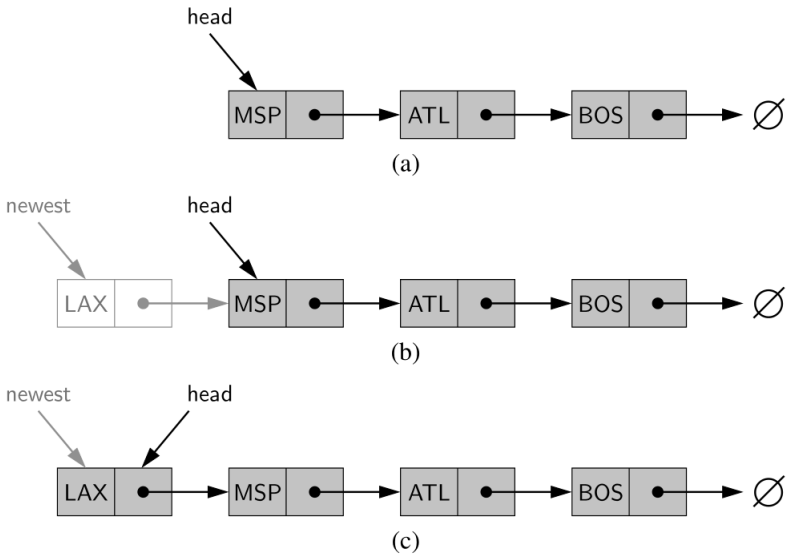


Figure 6.4: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the head reference.

Pseudocode to add an element to the head to a sLL

`add_first`(L, e):
`newest=Node(e)` {create new node instance storing reference to element e}
`newest.next=L.head` {set new node's next to reference the old head node}
`L.head=newest` {set variable head to reference the new node}
`L.size=L.size+1` {increment the node count}

Pseudocode for inserting a new element at the beginning of a singly linked list `L`. Note that we set the next pointer of the new node **before** we reassign variable `L.head` to it. If the list were initially empty (i.e., `L.head` is None), then a natural consequence is that the new node has its next reference set to `None`. To get the `L.size` we must do a traversal of the LL, which is $\Theta(n)$, reason why is common use to create a variable to associate with the LL's size, to avoid multiple traversal for every modification.

We can add an element also to the tail of the LL:

Pseudocode to add an element to the tail to a sLL

`add_first`(L, e):
`newest=Node(e)` {create new node instance storing reference to element e}
`newest.next=if` {set new node's next to reference the None object}
`L.tail.next=newest` {make old tail node point to new node}
`L.tail=newest` {set variable tail to reference the new node}
`L.size=L.size+1` {increment the node count}

Removing a node Removing an element from the **head** of a singly linked list is essentially the reverse operation of inserting a new element at the head, Figure 6.5.

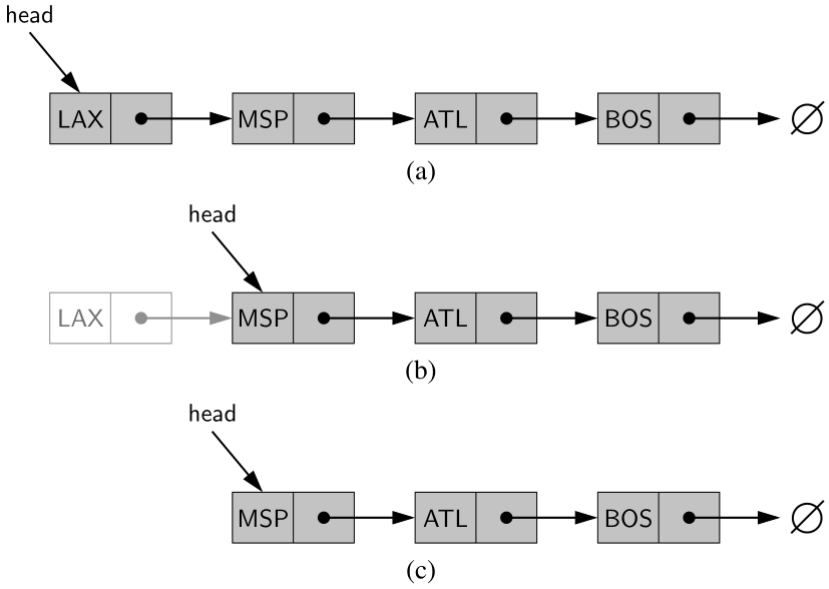


Figure 6.5: Removal of an element at the head of a singly linked list: (a) before the removal; (b) after "linking out" the old head; (c) final configuration.

Pseudocode to remove the head to a sLL

```
remove_first(L):
if L.head is None then
   Error:  L is empty
L.head=L.head.next {make head point to next node (or None)}
L.size=L.size-1 {decrement the node count}
```

Unfortunately, we cannot easily delete the last node of a singly linked list. Even if we maintain a tail reference directly to the last node of the list, we must be able to access the node before the last node in order to remove the last node. *But we cannot reach the node before the tail by following next links from the tail.* The only way to access this node is to start from the head of the list and search all the way through the list. But such a sequence of link-hopping operations could take a long time. If we want to support such an operation efficiently, we will need to make our list **doubly linked**.

To remove a generic element it will suffice to seek for it during a traversal, and when it's found we set the reference of the previous node (`previous.next`) to the next element wrt the current (`current.next`) via `previous.next=current.next` and then we delete the content of the node by assign `None` to its value `current=None` (in some programming languages such as `Java` or `Python`, you don't have to delete explicitly the removed node).

Here we discussed the **Singly Linked List** ADT, anyway, there are other variants of the LL data structure.

1. **Circularly Linked Lists**: are basically LL, but the pointer of last element points to the head of the list. Traversal of this kind of LL is a loop. Even though a circularly linked list has no beginning or end, per se, we must maintain a reference to a particular node in order to make use of the list

2. **Doubly Linked Lists** DLL are a special kind of LL, where every node has two pointers: one pointing to the next note in the DLL and one pointing to the *previous* node in the DLL. It provides greater summetry than the singly linked list and it is efficient when adding or deleting nodes on both sides. It uses *header* and *trailer* **sentinels** (dummy nodes that do not store any elements). Although we could implement a doubly linked list without sentinel nodes, the slight extra space devoted to the sentinels greatly simplifies the logic of our operations. Most notably, the header and trailer nodes never change — only the nodes between them change. Furthermore, we can treat all insertions in a unified manner, because a new node will always be placed between a pair of existing nodes. In similar fashion, every element that is to be deleted is guaranteed to be stored in a node that has neighbors on each side.

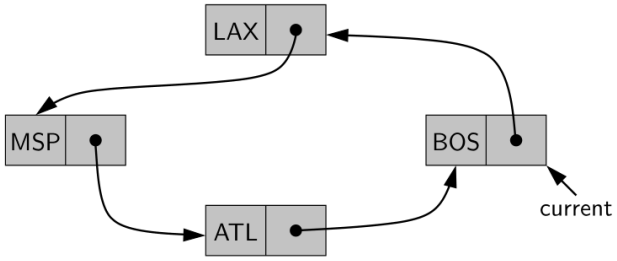


Figure 6.6: CLL

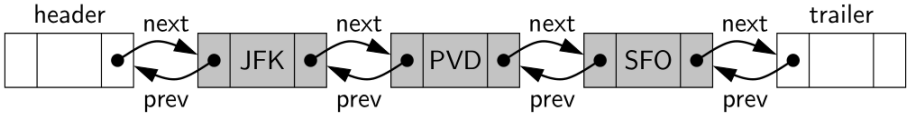


Figure 6.7: DLL

6.3.2 Positional List

It is a data structure that allows us to perform arbitrary insertions and deletions or to refer to elements anywhere in a list without the usage of a numerical index, but via a *variable* indexing. Numerical indexes are good, but they require to scan the entire list to find a specific element and to change dynamically when we update a list (e.g. when you add/remove and element in the intermediate positions of an array) and as a consequence, an index does not always refer to the same element within a list.

6.3.3 Array vs List-Based Data Structures

The dynamic memory allocation or dynamic resizing is a process that the compiler performs whenever, in adding a new element to our array, the latter already has all

Arrays	Linked Lists
An array is a collection of elements of a similar data type.	Linked List is an ordered collection of elements of the same type in which each element is connected to the next using pointers.
Array elements can be accessed randomly using the array index.	Random accessing is not possible in linked lists. The elements will have to be accessed sequentially.
Data elements are stored in contiguous locations in memory.	New elements can be stored anywhere and a reference is created for the new element using pointers.
Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed.	Insertion and Deletion operations are fast and easy in a linked list.
Memory is allocated during the compile time (Static memory allocation).	Memory is allocated during the run-time (Dynamic memory allocation).
Size of the array must be specified at the time of array declaration/initialization.	Size of a Linked list grows/shrinks as and when new elements are inserted/deleted.

Figure 6.8: Recap of the main differences between Arrays and Linked Lists

the memory allocations occupied. In particular, this process consists of creating a new array twice the size of the previously filled array, which is then copied into the new array ($2n$ space complexity).

Chapter 7

Trees

Productivity experts say that breakthroughs come by thinking "nonlinearly." In this section, we discuss one of the most important *nonlinear* data structures in computing — **trees**. Tree structures are indeed a breakthrough in data organization, for they allow us to implement a host of algorithms much faster than when using linear data structures, such as array-based lists or linked lists. Trees also provide a *natural organization for data*, and consequently have become ubiquitous structures in file systems, graphical user interfaces, databases, Web sites, and other computer systems. It is not always clear what productivity experts mean by "nonlinear" thinking, but when we say that trees are "nonlinear," we are referring to an organizational relationship that is richer than the simple "before" and "after" relationships between objects in sequences. The relationships in a tree are **hierarchical**, with some objects being "above" and some "below" others.

The main terminology for tree data structures comes from family trees, with the terms "parent," "child," "ancestor," and "descendant" being the most common words used to describe relationships. There exist many types of hierarchies:

- **Inclusion Hierarchy**: Recursive organization of entities (the "Chinese box" metaphor);
- **Control Hierarchy**: Social Organization - who gives orders to whom; a control system in which every entity has an assigned rank;
- **Level Hierarchy**: Each level is characterized by a particular space-temporal scale for its associated entities and for the processes through which the entities at this level interact with one another.

A tree represents the concept of hierarchy at the highest level of abstraction. We give the formal definition of a tree from Graph Theory.

Definition of a Tree

Let $V = \{v_1, \ldots, v_n\}$ be a set of nodes and the set E is a mapping of the set V in V, $E : V \to V$, thus $T(V, E) = T(V, V \times V)$; E is defined as a set of couples $\{v_i, v_j\}$ where $v_i, v_j \in V$ such that v_i is connected to v_j and thus v_i is the parent of v_j.

If $T(V, E)$ is:

(i) A connected graph of n verteces and $(n-1)$ links,

(ii) A connected graph without a circuit, or

or (iii) A graph in which every pair of verteces is connected with one and only one elementary path[a],

Then $T(V, E)$ is a **tree**.

[a] A path that does not pass through the same vertex twice

Normally, how we define a Tree in CS? More easily, a **free tree**[1] (or tree) is an **undirected graph** in which *any two vertices are connected by exactly one path*, or equivalently a **connected acyclic undirected graph**. If we lose the connection property, we get a **forest**.

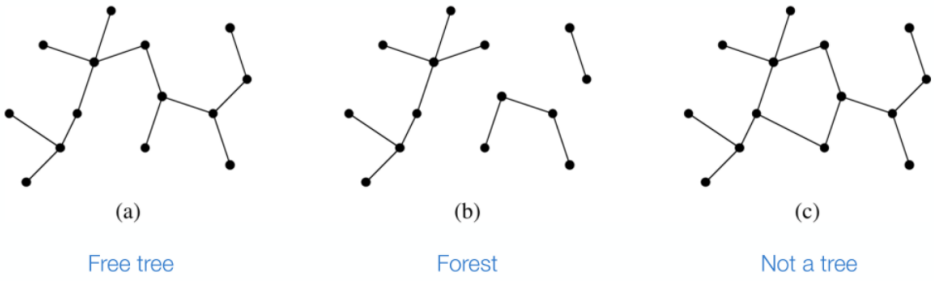


Figure 7.1: Example of a Free Tree(a), a Forest (b) and not a tree (c)

If $G = (V, E)$ is an undirected graph, the following statements are equivalent:

1. G is a free tree;
2. Any two vertices in G are connected by a unique simple path;
3. G is connected, but if any edge is removed from E the resulting graph is disconnected;
4. G is connected and $|E| = |V| - 1$;
5. G is acyclic and $|E| = |V| - 1$;
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

[1] The relationship parent-child here is not present, reason why it's an undirected graph.

7.1 Rooted Trees

What we're going to work with, in reality are **rooted trees**. A Rooted Tree is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements. We typically call the top element the **root** of the tree, but it is drawn as the highest element, with the other elements being connected below.

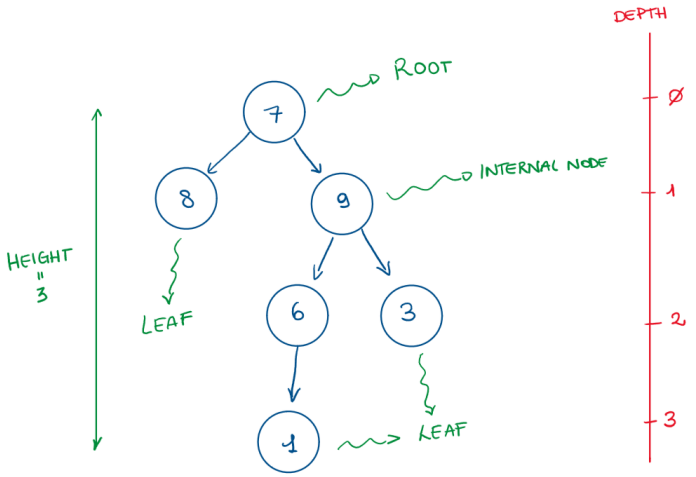


Figure 7.2: An example of rooted tree

Two nodes that are children of the same parent are *siblings*. A node x is **external** if x has no children. A node x is **internal** if it has one or more children. External nodes are also known as **leaves**. A node y is an ancestor of a node x if $x = y$ or y ian ancestor of the parent of x. Conversely, we say that a node x is a descendant of a node y if y is an ancestor of x.

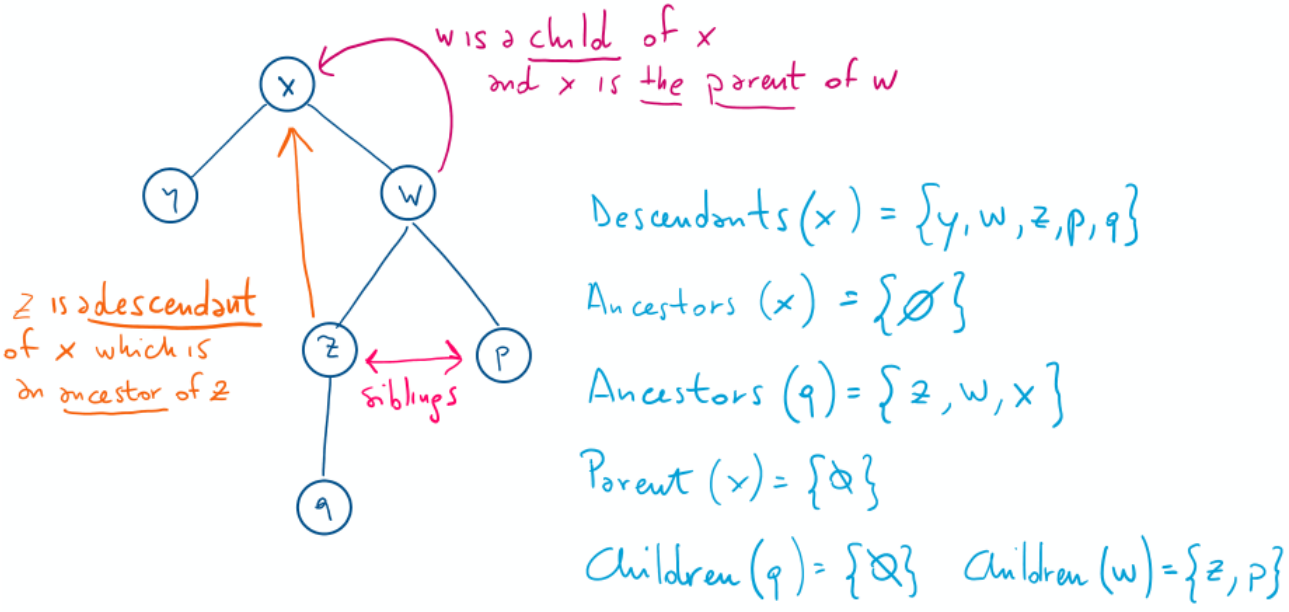


Figure 7.3: Rooted tree's properties

Rooted Tree

A **rooted** tree is a free tree where one of the nodes is called **root**.
Operations on them are:

- `p.element()`: Return the element stored at position p (is used to access to the element in a given node);
- `T.root()`: Return the position of the root of the tree or None if it's empty;
- `T.is_root(p)`: Return true if the node stored at position p is the root;
- `T.parent(p)`: Return the position of the parent of the node stored in p;
- `T.num_children(p)`: Return the number of children of p;
- `T.children(p)`: Generate an iterator of the children of position p;
- `T.is_leaf(p)` or `T.is_External(p)` (specular method: `T.is_Internal(p)`): Return true if p is a leaf;
- `len(T)`: Return the number of nodes in T (the number of positions);
- `T.is_empty()`: Return true is the tree is empty;
- `T.positions()` or `T.iterator()`: Generate an iterator of all positions of tree T;

We also define:

- **subtree** rooted in a node x is the tree induced by the descendants of x and x itself.
- **degree** of x is the number of children of a node x (*outdegree* of a node);
- **ordered tree** is a rooted tree in which the children of each node are ordered. So, if x has 3 children, say $\{y, w, z\}$ we can say that y is the first child, w is the second etc. (i.e. there's an imposed hierarchy in the children)

Finding the depth of a node in a Rooted Tree - $\Theta(n)$ We now see two different ways to find the depth of a node p in a tree T - an iterative version and a recursive version. But before we define formally the *depth* of a Tree:

Definition of Depth of a rooted Tree

Given a tree T and a node p we define the *depth*(T,p) as the number of edges in the simple path from the node p to the root. A root node will have a depth of 0.

Algorithm 16 depthIterative(T, p)

```
1: d = 0
2: while v=T.parent(p) is not None do
3:     d = d + 1
4: return d
```

The worst case of course is when p has a depth that is equal to n, where n is the total number of nodes of the Tree (i.e. a node structured as a "chain"). The complexity is therefore $\Theta(n)$.

Algorithm 17 DEPTHRECURSIVE(T, p)

```
1: if T.is_root(p) then
2:     return 0
3: else
4:     return 1+ DEPTHRECURSIVE(T, T.parent(p))
```

Finding the Height of a Rooted Tree - $\Theta(n^2)$ We now see an algorithm to find the height of a tree T. Remember: *the height of a tree is the maximum of the depths of its leaves.*

Algorithm 18 HEIGHT(T))

```
1: h = 0
2: for v ∈ T.positions() do
3:     // for all the nodes v of the Tree T do
4:     if T.is_leaf(v) then
5:         h = max(h, DEPTH(T, v))
6: return h
```

To define the worst case we need to find a "right" balance between the # of leaves and the depth for each leaf, i.e. a tree with $n/2$ leaves where each one has depth $n/2$, on the right of Figure 7.4, meaning that we iterate over $n/2$ leaves and compute for each $n/2$ work. In general the complexity is $O(n) + O(depth)$ for each leaf of the tree, in the worst case we get a $\Theta(n^2)$. Note that T.positions() can be implemented in $O(n)$.

Finding the height of a Node in a Rooted Tree - $\Theta(n)$ Before introducing the algorithm we must specify that the height of a tree is a specular quantity wrt the depth.

Definition of Height of a rooted Tree

Given a tree T and a node p we define the *height*(T,p) as the number of edges in the longest path from the node p to a leaf node. A leaf node will have a height of 0.

Algorithm 19 HEIGHT2(T, p)

```
1: if T.is_leaf(p) then
2:     return 0
3: else
4:     h = 0
5:     for w ∈ T.children(p) do
6:         h = max(h, HEIGHT2(T, w))
7:     return h + 1
```

Here the worst case is when the tree is structured as a "chain" of length n, and so the complexity is $\Theta(n)$.

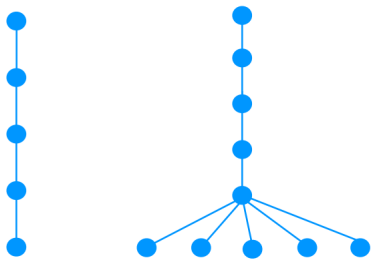
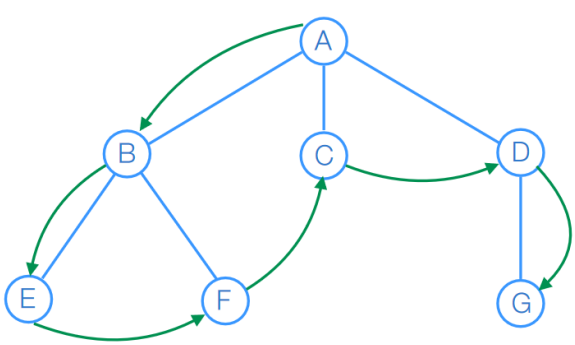


Figure 7.4: On the left, a tree sturctured as a "chain" which is the worst case of HEIGHT2 and DEPTH algorithms. On the right, the worst case tree for the HEIGHT algorithm.

7.1.1 Preorder and Postorder Visits Traversals of General Trees - $O(n)$

Tree traversal is a systematic method to visit the nodes of a tree executing an operation (for example, simply visiting) in each node. We see three main type of traversal schemes:

- **Pre-Order**: we first visit the parent and then the subtrees rooted on its children, in a recursive manner. The root here is always the first visited node.



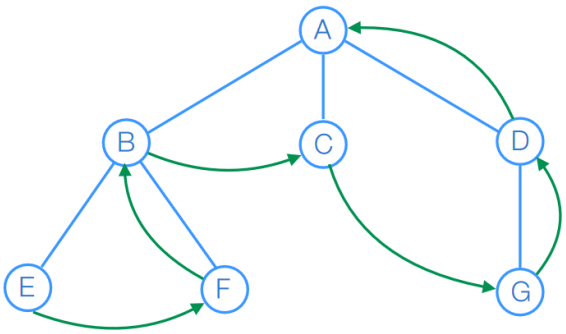

```
preorder(T,p)
    visit(p)
    for each c in T.children(p) do
        preorder(T,c)
```

Pre-order: A, B, E, F, C, D, G

Running time: O(n)

- **Post-Order**: recursively visit the subtrees rooted in the children, and then the parent. Here, contrarily to pre-order visits, the root is the last element to be visited.

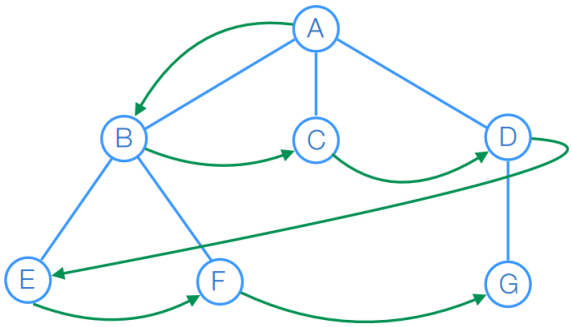
```
postorder(T,p)
    for each c in T.children(p) do
        postorder(T,c)
    visit(p)
```



Post-order: E, F, B, C, G, D, A

Running time: O(n)

- **Breadth-First**: visit all nodes at depth d, before visiting nodes at depth $d+1$.




```
BFT(T,p)
    Q.enqueue(T.root())
    while !Q.isEmpty() do
        p = Q.dequeue()
        visit(p)
        for each c in T.children(p) do
            Q.enqueue(c)
```

All visits we saw have running time $O(n)$.

7.2 Binary Search Trees

A **binary tree** (**BT**) is a special kind of *ordered* tree such that:

- every node has *at most* 2 children;
- every child node is labeled as **right node** or **left node**;
- The left child comes before the right child in the order of children of a node (*ordered* property).

The subtree rooted at a left or right child of an internal node v is called a left subtree or right subtree, respectively, of v. A binary tree is *proper* if each node has either zero or two children. Some people also refer to such trees as being full binary trees. Thus, in a proper binary tree, every internal node has exactly two children. A binary tree that is not proper is *improper*. We will call **external node**, a node without children (a leaf). All the other nodes are called **internal nodes**.

Properties of a binary tree Let T be a binary tree with n nodes (where n_E are external nodes and n_I are internal nodes) and h the height of the tree, then:

1. $2h + 1 \leq n \leq 2^h - 1$ which are the lower and upper bounds of the number of nodes that a BT can potentially have given the height (we recall that the height is equal to zero at the bottom of the tree, complementarily to the depth)[2];
2. $h + 1 \leq n_E \leq 2^{h-1}$, lower and upper bound of the number of external nodes;
3. $h \leq n_I \leq 2^{h-1} - 2$, lower and upper bound of the number of internal nodes;
4. $\lg(n + 1) - 1 \leq h \leq (n - 1)/2$, lower and upper bound of the height given the number of nodes;
5. $n_E = n_I + 1$

A **binary search tree (BST)** is a binary tree where the value of any left child node is smaller than (or equal) the value of its corresponding right child. More formally:

BST

A **binary search tree** (**BST**) T is a binary tree where if y is a node in the left subtree of x then y.key $\leq$ x.key and if y is a key in the right subtree of x then $y.key \geq x.key$, where:

- $x.key$ is the same as `x.element()`, which returns the element stored at position x (is used to access to the element in a given node);
- $x.left$ ($x.right$) returns the position of the left (right) child of x;

We emphasise that the children of a node do have a ordering relationship, i.e. given the node x holds that $x.left.key \leq x.right.key$

7.2.1 BST traversal: InOrder-Tree-Walk - $O(n)$

The in-order visit is a special kind of visit that can be done only in BST. This kind of visit visits the nodes in an **ordered** way, from the smaller element to the larger element in the BST. It follows the following schema.

Algorithm 20 InOrder-Tree-Walk(x)

```
1: if x ≠ None then
2:     InOrder-Tree-Walk(x.left)
3:     print x.key
4:     InOrder-Tree-Walk(x.right)
```

[2] A complete k-ary tree is a k-ary tree in which all leaves have the same depth and all internal nodes have degree k. How many leaves does a complete k-ary tree of height h have? The root has k children at depth 1, each of which has k children at depth 2, etc. Thus, the number of leaves at depth h is k^h. Consequently, the height of a complete k-ary tree with n leaves is $\log_k n$. The number of internal nodes of a complete k-ary tree of height h is

$$\begin{aligned} 1 + k + k^2 + \cdots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\ &= \frac{k^h - 1}{k - 1} \end{aligned}$$

by equation (4.2). Thus, a complete binary tree has $2^h - 1$ internal nodes.

By calling the algorithm on the root, we get the ordered list stored in the tree. Let's see an example and its recursive structure (for the sake of compactness and brevity, we'll show only the inner recursive structure of the left subtree wrt the root):

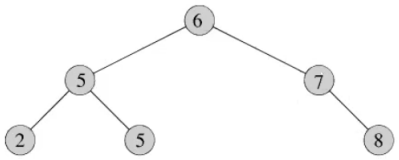


Figure 7.5: Binary Search Tree Traversal

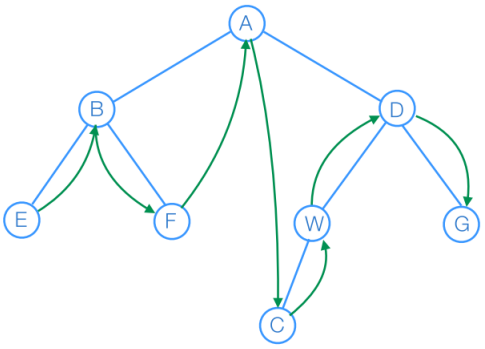


Figure 7.6: The INORDER-TREE-WALK can also be seen as a visit "from left to right".

INORDER-TREE-WALK($T.root()$) (=6)

if $T.root() \neq$ **None**:

(**True**)

INORDER-TREE-WALK($T.root().left$) (=5)

if $T.root().left \neq$ **None**:

(**True**)

INORDER-TREE-WALK($T.root().left.left$) (=2)

if $T.root().left.left \neq$ **None**:

(**True**)

INORDER-TREE-WALK($T.root().left.left.left$) (=None)

if $T.root().left.left.left \neq$ **None**:

(**False**)

print $T.root().left.left.key$ (=2)

INORDER-TREE-WALK($T.root().left.right$) (=None)

if $T.root().left.left.right \neq$ **None**:

(**False**)

print $T.root().left.left.key$ (=5)

INORDER-TREE-WALK($T.root().left.right$) (=5)

if $T.root().left.right \neq$ **None**:

(**True**)

INORDER-TREE-WALK($T.root().left.right.left$) (=None)

if $T.root().left.right.left \neq$ **None**:

(**False**)

print $T.root().left.right.key$ (=5)

INORDER-TREE-WALK($T.root().left.right.right$) (=None)

if $T.root().left.right.right \neq$ **None**:

(**False**)

print $T.root().key$

INORDER-TREE-WALK($T.root().right$) (=7)

...

The output is $2, 5, 5, 6, 7, 8$.

7.2.2 Searching in BST - $O(h)$

The core operation on a binary search tree is **search**, which means that we search for a node with a given value. In the following algorithm, x is the node where the search starts, and k is the value we are looking for.

Algorithm 21 BSTSEARCH(x, k)

```
1: if x ==None or k == x.key then
2:     return x
3: if k < x.key then
4:     return BSTSEARCH(x.left, k)
5: else
6:     return BSTSEARCH(x.right, k)
```

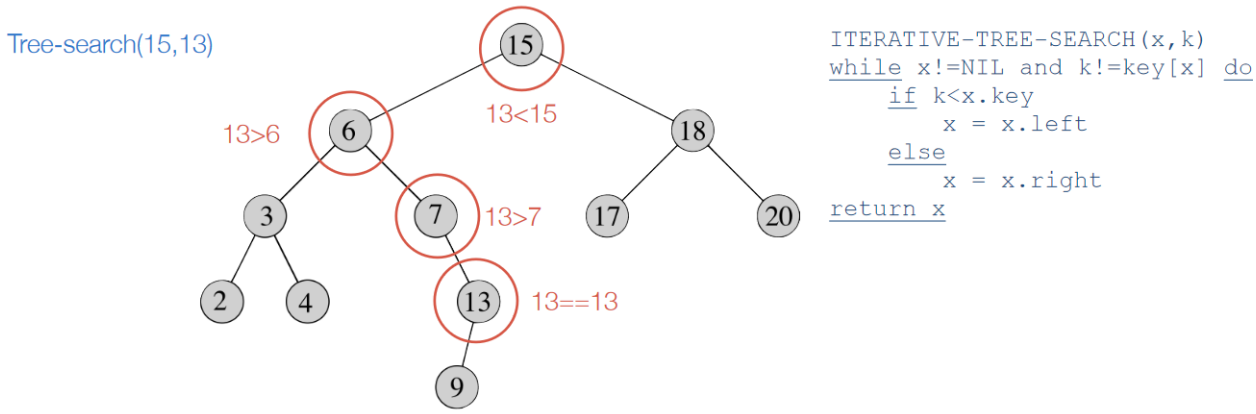


Figure 7.7: An example of BST search

The complexity of BST Serach is $O(h)$ where h is the height of the tree.

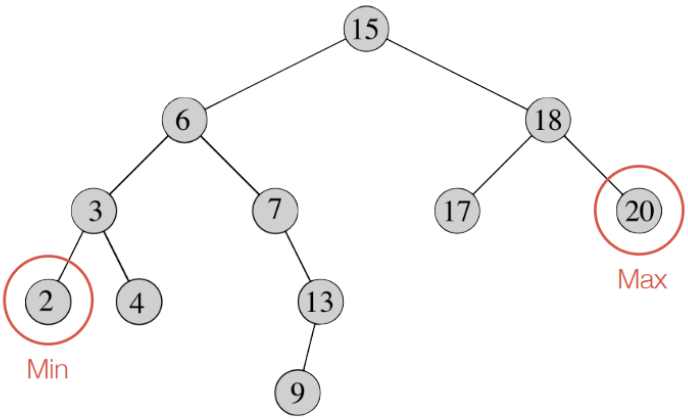
Exercise Suppose we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 364. Which of the following sequences **cannot** be the sequence of nodes examined?

- 1,252,401,398,330,344,397,364
- 924,220,911,244,898,258,362,364
- 925,202,911,240,912,245,364 ← 912 cannot belong to the left subtree of 911;

7.2.3 Minimum and Maximum of a BST

```
TREE-MINIMUM(x)
while x.left!=NIL do
    x = x.left
return x
```

You keep going down the left branch




```
TREE-MAXIMUM(x)
while x.right!=NIL do
    x = x.right
return x
```

You keep going down the right branch

7.3 Heaps

The (***binary***) **heap** data structure is an array object that we can view as a *nearly* ***complete binary tree***, as shown in Figure 7.8. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point (reason why it's called *nearly* CBT[3]).

An array A that represents a **heap** is an object with two attributes:

- *length*(A), which (as usual) gives the number of elements in the array, and
- *heapsize*(A), which represents how many elements in the heap are stored within array A.

That is, although $A[1,\ldots,length(\mathrm{A})]$ may contain numbers, only the elements in $\mathrm{A}[1,\ldots,heapsize(\mathrm{A})]$, where $0 \leq heapsize(\mathrm{A}) \leq length(\mathrm{A})$, are valid elements of the heap. We now list some important **properties** of heaps:

- The root of the heap A is always stored at $A[1]$.

Given an element with index i:

- The *parent* is stored at $\lfloor i/2 \rfloor$
- The *left child* is stored at $2i$
- The *right child* is stored at $2i+1$.

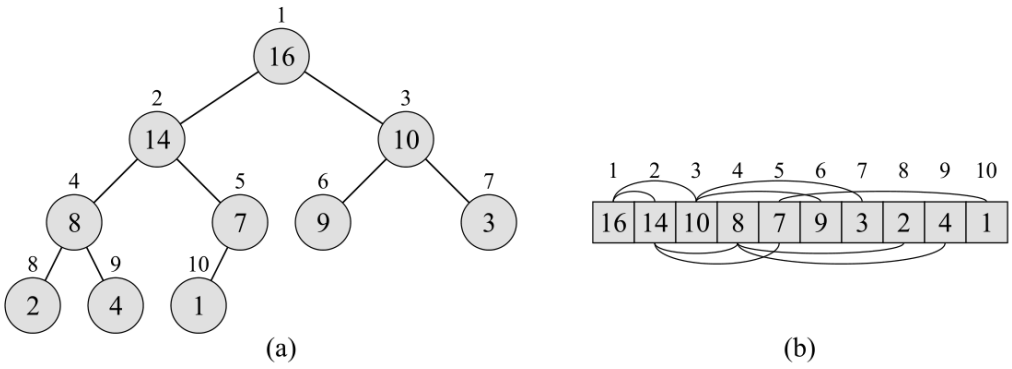


Figure 7.8: A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

We can define two types of heaps:

- **Max-Heap**: a max heap is a heap such that $A[parent(i)] \geq A[i]$, that is, the value of a node is at most the value of its parent. In a max-heap, the largest element is stored at the root.
- **Min-Heap**: a min heap is a heap such that $A[parent(i)] \leq A[i]$. In a min-heap, the smallest element is stored at the root.

Remember that, contrarily to BST, heaps do not have an imposed order on children's values of a given node, the hierarchical relationship of ordering is wrt the parent only.

[3]Complete Binary Tree

7.3.1 Max Heapify - $\Theta(\log n)$

Max-Heapify is a **recursive** algorithm that checks if the element at index i respects the max-heap property. If not, it "floats down" the element in $A[i]$ until the property is respected. The input is an array A and a node index i to check (**NB:** this algorithm doesn't transform a heap in a max heap! If just checks if the max heap property holds wrt the node at index i).

Algorithm 22 MaxHeapify(A, i)

```
1:  l =left(i)
2:  r =right(i)
3:  if l ≤ heapsize(A) and A[l] > A[i] then
4:      largest = l
5:  else
6:      largest = i
7:  if r ≤ heapsize(A) and A[r] > A[largest] then
8:      largest = r
9:  if largest ≠ i then
10:     exchange A[i] with A[largest]
11:     MaxHeapify(A, largest)
```

The running time of the MaxHeapify algorithm is $\Theta(\log n)$. An important note is on the fact that at the 13^{th} line we exchange the values of the two nodes but **not** the indices, so we clear out that MaxHeapify$(A$, largest$)$ is called on the index of that node whose value was found out to be the largest, and only in a second moment, we proceed by looking whether or not the Max-Heap property is not altered after such swapping operation.

7.3.2 Build Max Heap - $O(n \log n)$

We can use the procedure MaxHeapify in a bottom-up manner to convert an array $A[1, \ldots, n]$, where $n =$*length*(A), into a max-heap.

Note that, with an array representation for storing an n-element heap, the leaves are the nodes indexed by: $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$. This is a **key concept**!

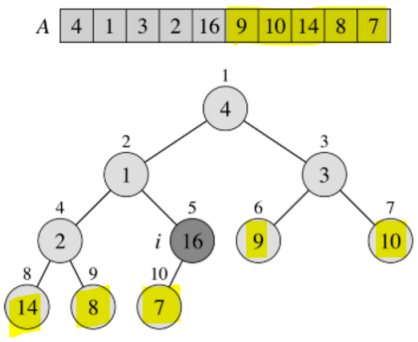


Figure 7.9: Note that the leaves are stored in the second half of the array!

The heap sort algorithm is used to sort a list, but first, we need another algorithm in order to build heap sort: the **BuildMaxHeap** algorithm, which builds a max heap from a given array A as input. The key intuition is the following: we first build a normal heap from the array (with the standard way of building from left to right we

saw at the beginning). Then, we iterate through all the nodes (that are not leaves) of the tree, from the lowest to the highest levels. We know how to do this because we saw before that all the nodes that are not leaves have indexes in $\{1, \ldots, \lfloor n/2 \rfloor\}$. Finally, for each iteration, we call the MaxHeapify algorithm. The end result will be a max heap!

Algorithm 23 BuildMaxHeap(A)

```
1: heapsize(A) = length(A)
2: for i = ⌊length[A]/2⌋ down to 1 do
3:     MaxHeapify(A, i)
```

An upper bound for the BuildMaxHeap algorithm is $O(n \log n)$. That's because MaxHeapify costs $O(\log n)$ and we call it $O(n)$ times. But we can prove that there exists a tighter bound of $O(n)$.

7.3.3 Heap Sort - $O(n \log n)$

We can now introduce the Heap Sort algorithm, which can sort (*in place*) an array taking advantage of the heap data structure. Indeed, this algorithm introduces a new algorithmic design technique: the use of data structure to manage information during the execution of the algorithm! The intuition is as follows:

1. given the input array A, we first call BuildMaxHeap(A): this will ensure that the root of the obtained tree will be the biggest element of the heap;
2. we exchange $A[1]$ with $A[i]$, i going from $length(A)$ down to 2, then do $heapsize(A) = heapsize(A) - 1$;
3. Now we call MaxHeapify$(A, 1)$ (i.e. we call it on the root of the tree). This will ensure that the root stays the biggest element in the heap. Note that his happens thanks to the fact that we are working on a max heap!

Algorithm 24 HeapSort(A)

```
1: BuildMaxHeap(A)
2: for i = length(A) down to 2 do
3:     exchange A[1] with A[i]
4:     heapsize(A) = heapsize(A) − 1
5:     MaxHeapify(A, 1)
```

Running Time: since we call BuildMaxHeap one time at the beginning, and then we call MaxHeapify $n-1$ times, we have a final running time of $O(n) + O((n-1)\log n) = O(n \log n)$. AAoN

7.3.4 Priority Queues

A Priority Queue is a data structure for maintaining a set S of elements, each with an associated *key*. The concept is very similar to a normal queue, where elements are supposed to be added and then "extracted" for some reason, but here *the criterion to decide which element is to extract first is the key associated with each element*. For example, in a **max-priority queue** the element with the largest key will be the next one to be extracted.

Max-priority queue

A **max-priority queue** with a set of elements S supports the following operations:

- Insert(S, x): Insert x in the set of elements S;
- Maximum(S): return the element with the max key in the queue;
- ExtractMax(S): returns and removes the element with the max key in the queue;
- IncreaseKeys(S, x, k): increases the value of x's key to k (where k is assumed to be larger than the current x's key).

Costs of the operations in **list-based** priority queues:

- Insert(S, x): $O(1)$
- Maximum(S): $O(n)$
- ExtractMax(S): $O(n)$

Costs of the operations in priority queues **implemented with heaps**:

- Insert(S, x): $O(\log n)$ (this is costlier!)
- Maximum(S): $O(1)$ (it's the root of the max heap!)
- ExtractMax(S): $O(\log n)$ (I think that the cost is different from the Maximum(S) method because, when you remove the max, you will have to rearrange the heap using MaxHeapify on the root, to obtain again a MaxHeap, like we do in Heap Sort)

Not surprisingly, we can use a heap to implement a priority queue. In a given application, such as job scheduling or event-driven simulation, elements of a priority queue correspond to objects in the application. We often need to determine which application object corresponds to a given priority-queue element, and vice versa.

Chapter 8

Hash Maps

To understand well what Hash Maps are, we begin with a practical example. Suppose that we want to design a system for storing student records, where each student has a unique badge number, and we want to perform efficient queries (for inserting new students data, retrieving data and deleting data using the badge number as key).

8.1 Direct Access Tables

One possible and naive solution are **direct access tables**. With this technique we consider a universe of keys $U = \{0, 1, \ldots, m-1\}$, and define the direct-access table as an array $T[0, \ldots, m-1]$, where $T[i] = (i, data(i)), i \in U$. This technique is very efficient when $|U|$ is relatively small, because we can access each key in constant time, but it's very inefficient for memory when $|U|$ is large, i.e. $|U| >> |K|$. If we consider the problem of the students data, if a badge number is made by 7 digits (e.g. 1227467), we would need a direct address table of size $\approx 9^7$, which of course is unfeasible (and also conceptually horrible).

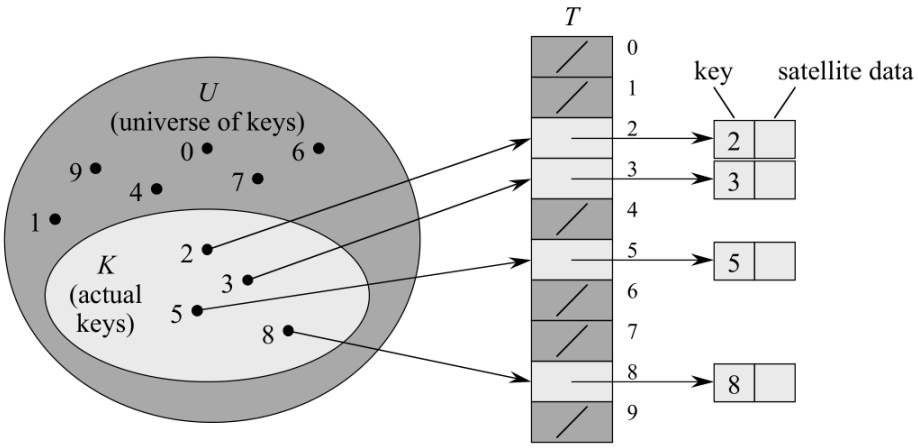


Figure 8.1: How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain None. See how only

.

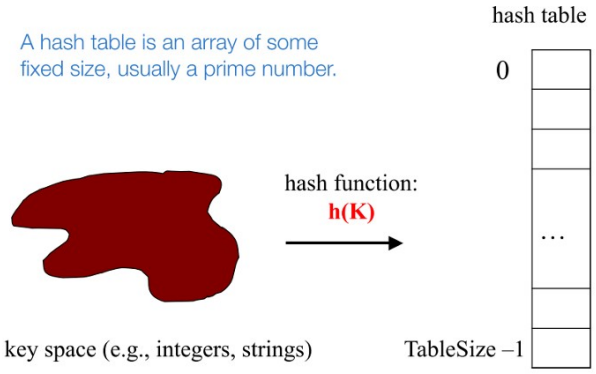


Figure 8.2: A hash table.

Hash Tables are the solution for this problem. Indeed, hash tables are again tables, but the size will be proportional to $|K|$, where K is the number of actually used keys. In this way we will lose the ability to access data directly using the key (e.g. if we have 1000 students, and I store them in a table of length 1000, now the indexes of this table will go from 0 to 999, which are not even correct numbers to express badge numbers!). We thus need a new way to access data: for this reason we will define a function that will map our keys to valid slots of our hash table. Specifically, if our hash table has size m, we will call a **hash map** (or function) a function $h : U \longrightarrow \{0, \ldots, m-1\}$.

So, the core concept is:

- With *direct access tables*, the key k maps to slot $T[k]$;
- With *hash tables*, key k maps (or "hashes") to slot $T[h(k)]$. We call $h(k)$ the *hash value* of key k.

A very common hash function is $h(k) = k \mod m$, where m is the table size. While this technique solves a lot of problems, it also brings new ones. For example, **collisions** might happen. A collision happens when, for two different keys k_1, k_2 we have that $h(k_1) = k(k_2)$ (i.e. the hash map is not injective w.r.t. the used keys). For this reason we say that a hash function is "good" when:

- is simple/fast to compute;
- avoids collisions as much as possible;
- has keys distributed evenly (*uniformly*) among the cells of the hash map.

8.2 Hash functions

A good hash function satisfies (approximately) the assumption of *simple uniform hashing*: each key is equally likely to hash to any of the m slots of the ash table, independently of where any other key has hashed to. Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently. Occasionally we do know the distribution. For example, if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k > 1$, then the has function

$$h(k) = \lfloor km \rfloor$$

satisfies the condition of simple uniform random hashing. A good approach derives the hash value in a way that we expect to be independent of any patterns that might

exist in the data. For example, the "*division method*" computes the hash value as the remainder when the key is divided by a specified prime number. This method frequently gives good results, assuming that we choose a prime number that is unrelated to any patterns in the distribution of keys.

Division Method In the division method for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is $h(k) = k \bmod m$.

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k. Unless we know that all low-order p -bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key.

But, **how to choose** m? *A prime not too close to an exact power of* 2 *is often a good choice for* m. For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2000$ character strings, where a character has 8 bits. We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size $m = 701$. We could choose $m = 701$ because it is a prime near $2000/3$ but not near any power of 2. Treating each key k as an integer, our hash function would be

$$h(k) = k \bmod 701$$

Moltiplication Method The multiplication method for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result. In short, the hash function is $h(k) = \lfloor m(kA \bmod 1) \rfloor$ where "kA mod 1 " means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$. An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2 ($m = 2^p$ for some integer p), since we can then easily implement the function on most computers as follows. Although this method works with any value of the constant A, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. Knuth [3] suggests that $A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$ is likely to work reasonably well.

Collision is a problem because two different keys will point to the same slot in the hash table, and will be almost impossible to avoid when $|U| >> |K|$. For this reason we will now see different techniques to handle collisions.

8.3 Separate Chaining

With Separate Chaining, our hash table doesn't store directly the key, but **points to the head of a linked list**. In this way, even if there are collisions, the new element can be added (or "chained") to the front of the linked list.

With this technique, inserting a new element into the hash table requires $O(1)$ but only if we are sure that the element is not already present in the hash table. Otherwise it will take $O(1 + search)$ (as well as the *delete* method), where *search* is the algorithm where we scan the linked list associated with the slot. We define the **load factor** of a hash table the quantity $\alpha = n/m$, where n is the number of stored elements, and m is the number of slots. The worst case running time of *search* is $\Theta(n)$, i.e. the hash function addresses the elements in a single key. The average depend on how well the

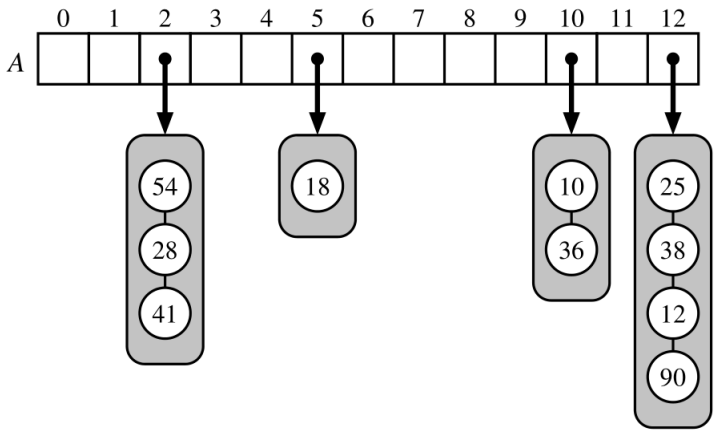


Figure 8.3: Separate Chaining.

hasing function distributes the n keys in the m slots. Keeping track of the load factor is essential to calibrate the size of the hash table, and to maintain a low complexity for the search in the average case.

Separate Chaining under the assumption of Simple Uniform Hashing Under the SUH assumption, we get that for $j = 0, 1, \ldots, m-1$, the list $T[j]$ is long n_j, where $n = n_0 + n_1 + \ldots + n_{m-1}$, so the expected value of the number of elements per key j, n_j, is

$$n_j = E[n_j] = \alpha = n/m$$

The average running time for a successful search in a hash table solving collisions with chaining and adopting simple uniform hashing is $\boldsymbol{\Theta}(1 + \alpha)$.

If m is proportional to n, then $n = O(m)$ and $\alpha = n/m = O(m)/m = O(1)$

8.4 Open Addressing

Separate chaining has the major flaw of using linked lists! It requires the implementation of a second data structure. The key ideas of Open Addressing are the following:

- **All the data will be stored in the table**, thus, a bigger table w.r.t. separate chaining is needed. The general goal is to have a load factor $\alpha < 0.5$, this is crucial since assuming a load factor below 0.5 provides theoretical guarantees that we'll find empty spaces in case collisions happen. If $\alpha \geq 0.5$ it might happen that the open addressing strategy falls;
- When a collision occurs, alternative cells are tried until an empty cell is found, with a process of research depicted by a function $f(\cdot)$, i.e. the **collision resolution strategy**;

More specifically, when a collision occurs, cells $h_0(x), h_1(x), \ldots$ will be tried in succession, where

$$h_i(x) = (h(x) + f(i)) \mod m, \quad \text{with } f(0) = 0$$

Where m is the table size and f is the collision resolution strategy. There are three common collision resolution strategies:

- **Linear Probing**;

- **Quadratic Probing**;
- **Double Hashing**.

8.4.1 Linear Probing, Quadratic Probing and Double Hasing

Linear Probing

In linear probing we have a *linear* collision resolution strategy $f(i) = ci$, where typically $c = 1$. Thus, a probe sequence would look like this: $\{h(k) \mod m, (h(k) + 1) \mod m, (h(k) + 2) \mod m, \dots\}$. The main problem of linear probing is **primary clustering**: any key that hashes into a cluster will require several attempts to resolve collision, and will ultimately add to the cluster! Primary clustering gets worse as the table fills up. The **average number of examined cells in an insertion** using linear probing is roughly:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

For a half full table we obtain 2.5 as the average number of cells examined during an insertion. The disastrous effects come when the load factor is ≥ 0.5.

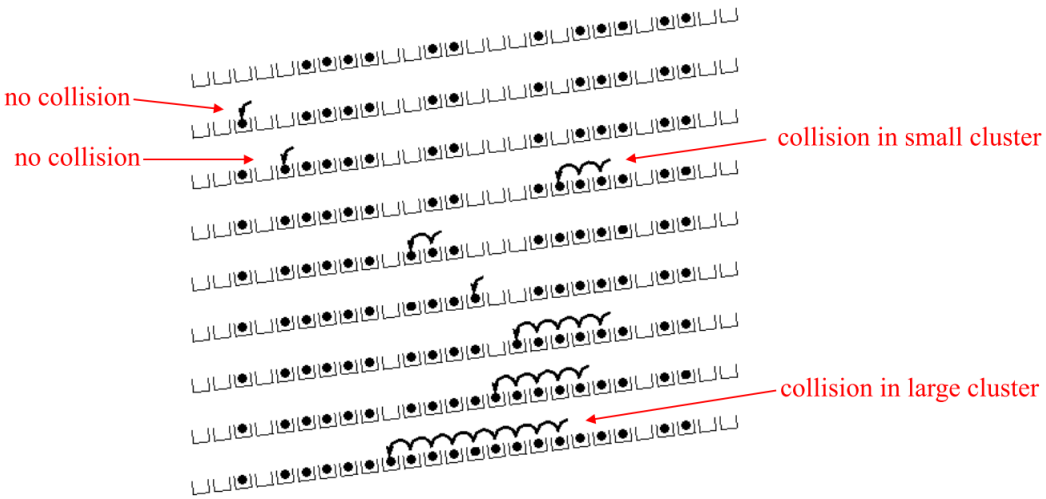


Figure 8.4: A representation of Primary Clustering.

Note that the when we search an element in the array, we follow the same probe sequence as the insert algorithm. More specifically, note that the cost of a successful search of x is equal to the cost of inserting x at the time x was inserted. For example: if $\alpha = 0.5$, at the time of the insertion of x the average cost per insertion was 2.5. The average cost of finding the newly inserted item will be 2.5, no matter how many insertions follow.

Lastly, we have that:

- The average number of examined cells in an **unsuccessful search** using linear probing is roughly $(1 + 1/(1-\alpha)^2)/2$;
- The average number of cells that are examined in a **successful search** is approximately $(1 + 1/(1-\alpha))/2$.

Quadratic Probing

In quadratic probing we have a quadratic collision resolution strategy $f(i) = ci^2$, where typically $c = 1$. Thus, a probe sequence would look like this: $\{h(k) \mod m, (h(k)+1) \mod m, (h(k)+4) \mod m, \dots\}$.The main problem with quadratic probing is no more primary clustering, but **secondary clustering** (aha!): that happens because keys that hash to the same spot will still use the same sequence of probes and will conflict with each other! Also in this case we aim for $\alpha < 0.5$: if this condition is satisfied, quadratic probing will find an empty slot; for bigger *alpha*, quadratic probing *may* find a slot.

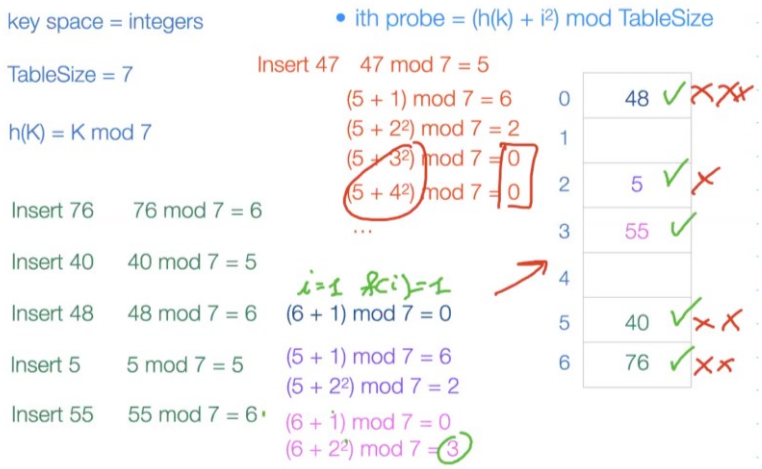


Figure 8.5: An example of secondary clustering.

Double Hashing

With double hashing the collision reosolution strategy is $f(i) = ig(k)$, where g is a **second hash function** (e.g. division, multiplication method,...). A probe sequence would look like this: $\{h(k) \mod m, (h(k)+g(k)) \mod m, (h(k)+2g(k)) \mod m, \dots\}$.

Rehashing

Another technique for avoiding collisions is Rehashing. When the table gets too full, we create a bigger table (usually twice as large) and hash all the items from the original table into the new table. Usually rehashing is performed when:

- The original table is half full ($\alpha = 0.5$);
- An insertion fails;

Note that the cost of rehashing is linear w.r.t. n: in fact we can't just copy data from the old table, because the new table will have a new hash function (the value of m changes), but we have to recompute the hash value for each non deleted key and put the item in its new position in the new table.

Chapter 9

Graphs

Graphs are another very important *non linear* data structure.

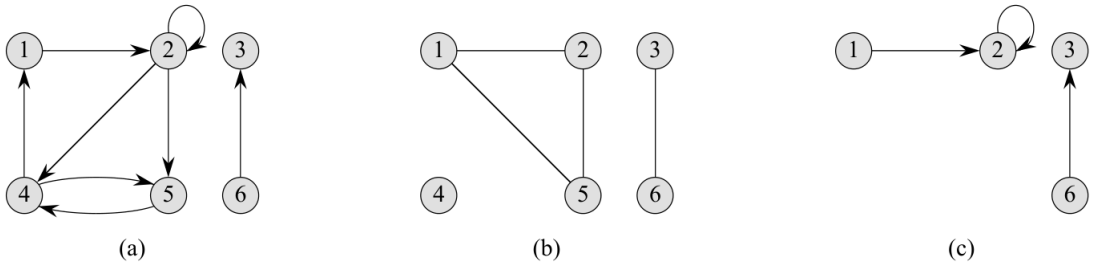


Figure 9.1: *Directed* and *undirected graphs*. (a) A directed graph $G = (V, E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2),(2,2),(2,4),(2,5),(4,1),(4,5),(5,4),(6,3)\}$. The edge (2,2) is a self-loop. (b) An undirected graph $G = (V, E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2),(1,5),(2,5),(3,6)\}$. The vertex 4 is isolated. (c) The subgraph of the graph in part (a) induced by the vertex set $\{1,2,3,6\}$

A **directed graph** (or digraph) G is a pair (V, E), where V is a finite set and E is a binary relation on V. The set V is called the vertex set of G, and its elements are called vertices (singular: vertex). The set E is called the edge set of G, and its elements are called edges. Figure 9(a) is a pictorial representation of a directed graph on the vertex set $\{1,2,3,4,5,6\}$. Vertices are represented by circles in the figure, and edges are represented by arrows. *Note that self-loops-edges from a vertex to itself − are possible* (contrarily to undirected graphs).

In an **undirected graph** $G = (V, E)$, the edge set E consists of *unordered pairs of vertices*, rather than ordered pairs. That is, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. By convention, we use the notation (u, v) for an edge, rather than the set notation $\{u, v\}$, and we consider (u, v) and (v, u) to be the same edge (for undirected graphs). In an undirected graph, self-loops are forbidden, and so every edge consists of two distinct vertices. Figure 9(b) is a pictorial representation of an undirected graph on the vertex set $\{1,2,3,4,5,6\}$

Many definitions for directed and undirected graphs are the same, although certain terms have slightly different meanings in the two contexts.

If (u, v) is an edge in a graph $G = (V, E)$, we say that vertex v is **adjacent** to vertex u. When the graph is undirected, the adjacency relation is *symmetric*. When the graph

is directed, the adjacency relation is *not necessarily symmetric*. If v is adjacent to u in a directed graph, we sometimes write $u \to v$. In parts (a) and (b) of Figure 9 vertex 2 is adjacent to vertex 1, since the edge (1,2) belongs to both graphs. Vertex 1 is not adjacent to vertex 2 in Figure 9(a) since the edge (2,1) does not belong to the graph.

The ***degree** of a vertex* in an undirected graph is the number of edges incident on it. For example, vertex 2 in Figure 9(b) has degree 2. A vertex whose degree is 0 , such as vertex 4 in Figure 9(b), is *isolated*. In a directed graph, the out-degree of a vertex is the number of edges leaving it, and the in-degree of a vertex is the number of edges entering it. The degree of a vertex in a directed graph is its in-degree plus its out-degree. Vertex 2 in Figure 9(a) has in-degree 2, out-degree 3 and degree 5.

We say that a graph is **dense** when $|E| \approx |V|^2$; we say that a graph is **sparse** when $|E| << |V|^2$. Also remember that if G is connected (i.e. there is a path between every pair of vertices), then we have that $|E| \geq |V| - 1$. When we have the equality, then G is a tree.

Other basic definition are given:

- A **path** of length k from a vertex u to v in a graph $G = (V, E)$ is a sequence $p = \{u, u_1, u_2, \ldots, v\}$ where $(u_i, u_{i+1}) \in E$, and $|p| = k$.
- A path is **simple** if all vertices in the path are distinct;
- An undirected graph G is **connected** if there exists a path between every pair of vertices; $|E| \geq |V| - 1$ and if $|E| = |V| - 1$ G is a tree;
- A graph G is **strongly connected** if every two vertices are reachable from each other (this is different from the above definition only in directed graphs).

There are two standard ways to represent a graph: **adjacency lists** and **adjacency matrix**. They both use (and store) the *adjacency* information to describe a graph.

9.1 Adjacency Lists - $\Theta(|V| + |E|)$ memory, $O(|V|)$ edge search

Adjacency lists consist of an array A of $|V|$ (linked) lists, where for each $u \in V$, $A[u]$ consists of a list of all vertices adjacent to u.

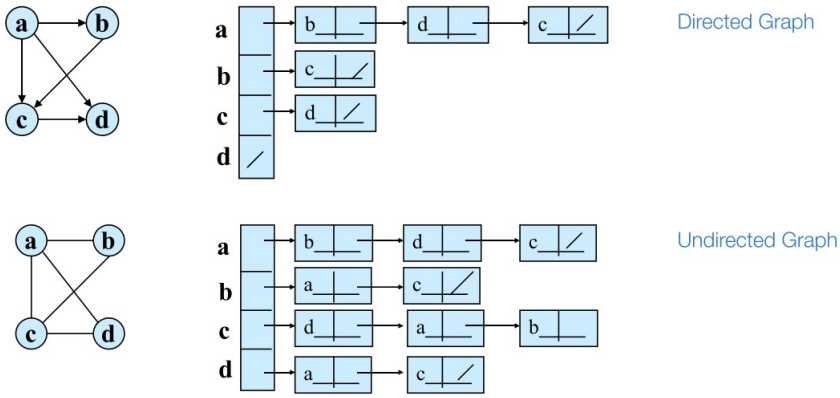


Figure 9.2: Adjacency lists for a directed and an undirected graph. Note that *the order of adjacent nodes in each list is **not** relevant.*

1. For **directed graphs**, where $d_{out}(v)$ is the out degree of the node $v \in V$, we have that:

(a) Sum of lengths of all the adjacency lists is $\sum_{v \in V} d_{out}(v) = |E|$;

(b) Total storage required is $\Theta(|V| + |E|)$

1. For **undirected graphs** where $d(v)$ is the out degree of the node $v \in V$, we have that:

   (a) Sum of lengths of all the adjacency lists is $\sum_{v \in V} d(v) = 2|E|$;

   (b) Total storage required is $\Theta(|V| + |E|)$

The main **pros** in adjacency lists are:

- Space efficient when a graph is *sparse* (you store the edge only if exists);
- Can be modified to support many graph variants.

The main **cons** in adjacency lists are:

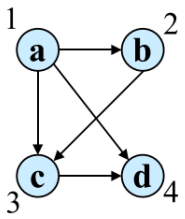
- Determining if an edge $(u, v) \in G$ is not efficient! In fact, we have to search in u's adjacency list, leading to $\Theta(d(u))$ complexity (which is $\Theta(|V| - 1)$ in the worst case).

9.2 Adjacency Matrix - $\Theta(|V|^2)$ memory, $O(1)$ edge search

An adjacency matrix A is a $|V| \times |V|$ matrix. To represent a graph via a adjacency matrix we need first to number verices from 1 to V in some arbitrary way. Then, we can define A:

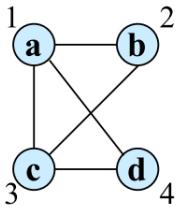
$$A[i, j] = a_{ij} = \begin{cases} 1 & \text{if} \quad (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

It immediately follows that, for undirected graphs, adjacency matrix is symmetric.



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

Directed Graph



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

Undirected Graph

Figure 9.3: Adjacency matrix for a directed and an undirected graph. The convention for the directed graphs' adjacency matrices is that the row index represent the starting point of the edge and the column index the ending one, i.e. an element in the matrix is 1 if $\text{node}_i \to \text{node}_j$

The main **cons** for adjacency matrices are:

- The **storage** required with an adjacency matrix is $\Theta(|V|^2)$ for both directed and undirected graphs: way less efficient than adjacency lists. In alternative, we can use a different format that saves just the existent edges (Sparse Adjacency Matrices);

The main **pros** for adjacency matrices are:

- The time to list all vertices adjacent to u is $\Theta(V)$
- The time to determine if $(u, v) \in E$ is $\Theta(1)$ (!)
- For weighted graphs, we can store weights instead of only 0 and 1.

Up to now we saw how to represent, store and check if an edge is in a certain graph. We now see some different ways to search in a graph.

9.3 Breadth-First Search - $\Theta(|V| + |E|)$

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. *Prim*'s minimum-spanning tree algorithm and *Dijkstra*'s single-source shortest-paths algorithm use ideas similar to those in breadth-first search.

Given a graph $G = (V, E)$ and a distinguished source vertex $s \in V$, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the simple path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

To keep track of progress, breadth-first search colors each vertex *white*, *gray*, or *black*. All vertices start out white and may later become gray and then black. A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

So summarising, this algorithm expands the **frontier** between discovered and undiscovered vertices (the frontier of a vertex is the set of all adjacent vertices). We say that a vertex is **discovered** the first time it is encountered during the search. We say also that a vertex is **finished** when all its frontier is made by discovered (or fininshed) vertices. We now see the pseudocode for the BFS algorithm which takes as input the graph G and the starting vertex s. Note that, since we are using a queue, the order in which we discover nodes (end we then proceed to enqueue them) is important! Remember that, in a queue, the elements who enter first, also get extracted first. For this reason **it's important to follow an order when discovering nodes** (e.g. alphabetical order). In addition to the BFT (Breadth-First Tree), given as input a graph G represented via its adjacency list and a source vertex s, the algorithm outputs 2 different arrays:

- d: is the array containing the lengths of the shortest paths from s to any other node v. Specifically, $d[v]$ = length of shortest path from s to v. We set $d[v] = \infty$ if v is not reachable from s.
- π: is the array containing the last edge of the shortest paths from s to any other vertex. Specifically, $\pi[v] = u$, where (u, v) is the last edge on the shortest path from s to v. The vertex u is called as the *predecessor* of v.

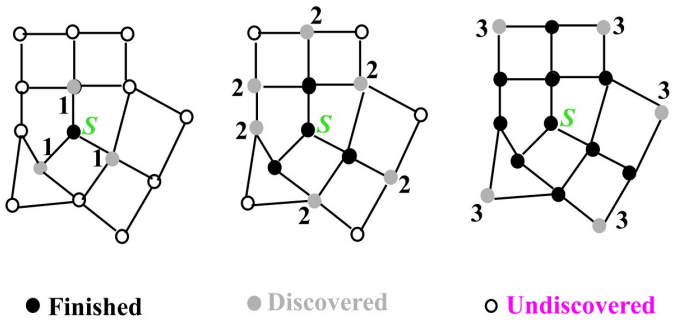


Figure 9.4: Breadth first search algorithm. Note that we can use colors to better identify the types of vertices.

Algorithm 25 BFS(G, s)

```
1:  for u ∈ V[G] − {s} do
2:      color[u] = white
3:      d[u] = ∞
4:      π[u] = None
5:  color[s] = grey
6:  d[s] = 0
7:  π[s] = None
8:  Initialize Queue Q
9:  Q.enqueue(s)
10: while Q ≠ ∅ do
11:     u = Q.dequeue()
12:     for v ∈ Adj[u] do
13:         if color[v] = white then
14:             color[v] = grey
15:             d[v] = d[u] + 1
16:             π[v] = u
17:             Q.enqueue(v)
18:     color[u] = black
```

The procedure BFS works as follows. With the exception of the source vertex s, lines $1 - 4$ paint every vertex white, set $d[u]$ to be infinity for each vertex u, and set the parent of every vertex to be None. Line 5 paints s gray, since we consider it to be discovered as the procedure begins. Line 6 initializes $s.d$ to 0, and line 7 sets the predecessor of the source to be None. Lines $8 - 9$ initialize Q to the queue containing just the vertex s.

The while loop of lines $10 - 18$ iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined. This while loop maintains the following invariant: At the test in line 10, the queue

Q consists of the set of gray vertices. Although we won't use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The for loop of lines $12-17$ considers each vertex v in the adjacency list of u. If v is white, then it has not yet been discovered, and the procedure discovers it by executing lines $14-17$. The procedure paints vertex v gray, sets its distance $v.d$ to $u.d+1$, records u as its parent $v.\pi$, and places it at the tail of the queue Q. Once the procedure has examined all the vertices on u 's

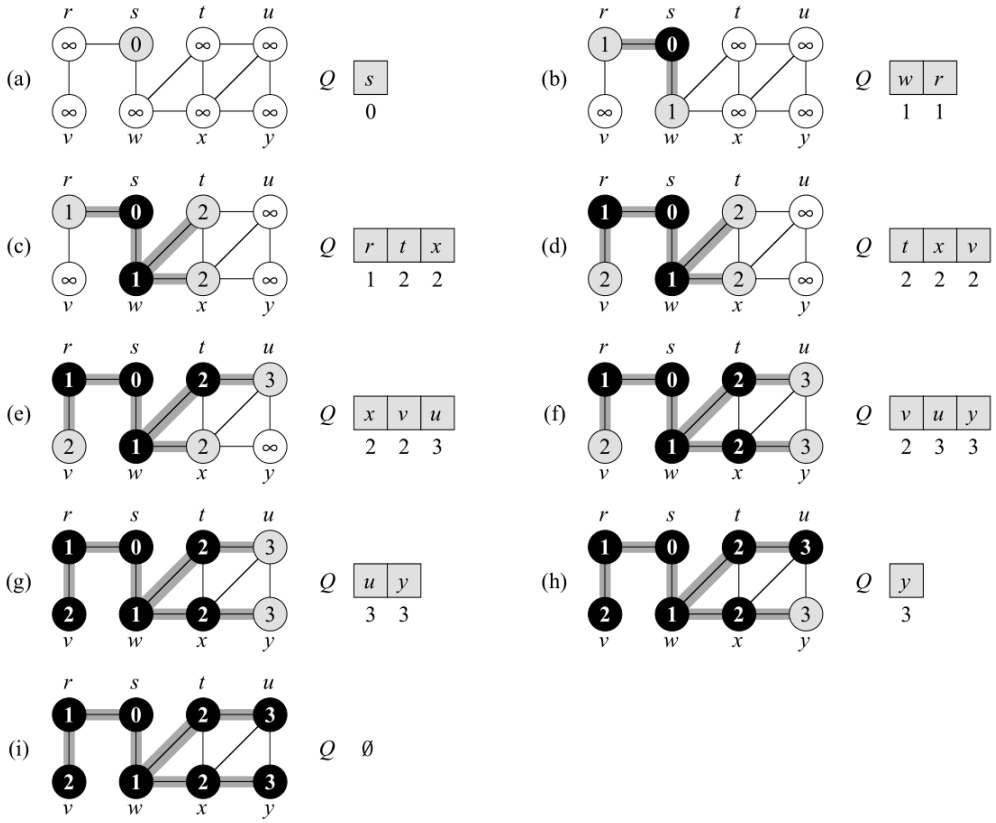


Figure 9.5: The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of u:d appears within each vertex u. The queue Q is shown at the beginning of each iteration of the while loop of lines 10–18. Vertex distances appear below vertices in the queue.

We call **predecessor subgraph** of graph $G(V,E)$ with source s a graph $G\pi = (V\pi, E\pi)$ where:

- $V\pi = \{v \in V : \pi[v] \neq \text{None} \cup \{s\}\}$
- $E\pi = \{(\pi[v], v) \in E : v \in V\pi - \{s\}\}$

With such definitions, the predecessor subgraph $G\pi$ is called a **breadth-first tree** which has the following properties:

- $V\pi$ consists of the vertices reachable from s and for all $v \in V\pi$ there is a **unique** simple path from s to v in $G\pi$, that is also a shortest path from s to v in G.
- The edges in $E\pi$ are called tree edges.

- As a consequence $|E\pi| = |V\pi| - 1$

9.3.1 Running Time

The total running time for BFS is $O(|V| + |E|)$, which means that it's linear in the size of the adjacency list representation of the input graph. We get this result because the algorithm follows the folling steps:

- Initialization (1-4 lines) takes $O(|V|)$;
- Initialization (5-8 lines) takes $O(1)$;
- Since each node is enqueued and dequeued at most once during a BFS, and since each of those operations is $O(1)$, we get that the total time for queuing is $O(|V|)$;
- The adjacency list of each vertex is scanned for each loop at most once, and the sum of lengths of all the adjacency lists is $O(|E|)$;

  $\Rightarrow$ Summing up everything BFS is linear in the size of the adjacency list representation of graph: $O(|V| + |E|)$.

9.4 Depth First Search - $\Theta(|V| + |E|)$

With DFS we explore edges of an directed or undirected graph G out of the **most recently discovered vertex** v. We continue until all the nodes reachable from the original source are discovered. If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source. Note that we don't need to explicitly define a source vertex.

The **output** of the DFS algorithm is the following:

- 2 **timestamps** on each vertex v. Integers between 1 and $2|V|$. The first one is the discovery time $d[v]$ and the second one is the finishing time $f[v]$.
- An array π such that $\pi[v] = u$ is the predecessor of v, such that v was discovered during the scan of u's adjacency list.

We will use the same colouring scheme for vertices as BFS.

Algorithm 26 DFS(G)

```
1: for u ∈ V[G] do
2:     color[u] = white
3:     π[u] = None
4: time = 0
5: for u ∈ V[G] do
6:     if color[u] == white then
7:         DFS-Visit(G, u)
```

Procedure DFS works as follows. Lines $1 - 3$ paint all vertices white and initialize their π attributes to None. Line 4 resets the global time counter. Lines $5 - 7$ check each vertex in V in turn and, when a white vertex is found, visit it using DFS-VISIT. Every time DFS-VISIT (G, u) is called in line 7, vertex u becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex u has been assigned a discovery time u. d and a finishing time $f[u]$.

In each call DFS-VISIT (G, u), vertex u is initially white. Line 1 increments the global variable time, line 2 records the new value of time as the discovery time $d[u]$, and

Algorithm 27 DFS-VISIT(G, u)

```
1: time = time +1 // white vertex u has just been discovered
2: d[u] = time
3: color[u] = gray
4: for v ∈ Adj[u] do // explore edge (u, v)
5:     if color[v] = white then
6:         π[v] = u
7:         DFS-VISIT(G, v)
8: color[u] = black // blacken u; it is finished
9: time = time + 1
10: f[u] = time
```

line 3 paints u gray. Lines $4-7$ examine each vertex v adjacent to u and recursively visit v if it is white. As each vertex $v \in Adj[u]$ is considered in line 4, we say that edge (u, v) is explored by the depth-first search. Finally, after every edge leaving u has been explored, lines $8-10$ paint u black, increment time, and record the finishing time in $f[u]$.

Note that the results of depth-first search may depend upon the order in which line 5 of DFS examines the vertices and upon the order in which line 4 of DFSVISIT visits the neighbors of a vertex. These different visitation orders tend not to cause problems in practice, as we can usually use *any* depth-first search result effectively, with essentially equivalent results.

Note that, the predecessor subgraph $G\pi$ forms a depth-first forest composed of several depth-first tree. With DFS we will have 4 different types of edges (u, v):

- **Tree Edge:** a edge belonging to the depth-first forest.
- **Back Edge:** when v is a predecessor of u in the depth-first tree
- **Forward Edge:** when v is a descendant of u in the depth-first forest
- **Cross Edge:** any other edge. Can go between vertices in the same depth-tree or also in different depth-first trees.

9.4.1 Running Time

The total running time of DFS is $\Theta(|V| + |E|)$ (the same as BFS.).

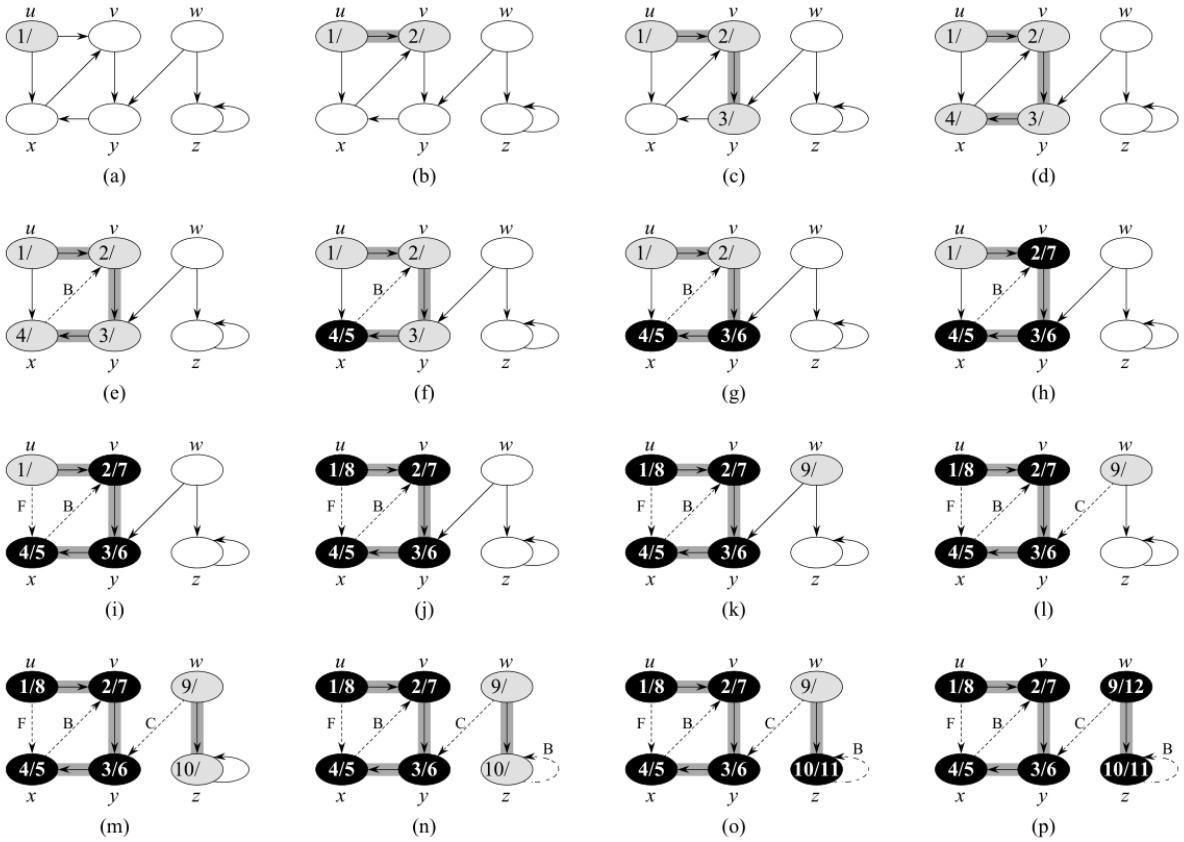


Figure 9.6: The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

9.5 Single-Source Shortest Path Algorithm

The Shortest Path algorithm calculates the shortest (*weighted*) path between a pair of nodes. It's useful for user interactions and dynamic workflows because it works in real time. Use Shortest Path to find optimal routes between a pair of nodes, based on either the number of hops or any weighted relationship value. It can provide real-time answers about degrees of separation, the shortest distance between points, or the least expensive route. But before defining our problem in a formal way, let's introduce some definitions:

Let $G(V, E)$ a **weighted** graph s.t. there exists $w : E \to \mathbf{R}$; weights can be negative, positive or zero.

- We define a **direct path** from v_1 to v_k as $v_1 \rightsquigarrow v_k$, that can be expressed alternatively as $p = v_1 \to v_2 \to \ldots v_{k-1} \to v_k$.
- The **weight** of the direct path p is $w(p) = \sum_i w(v_i, v_{i+1})$
- **Shortest path** from u to v is the $p_{u,v}$ with minimum weight $w(p)$.

We can use dynamic programming to find the shortest path on a graph, but we need to find the *optimal substructure* and a sub-path of a shortest path must be a shortest path. In the case in which we consider negative weights it could be very hard to solve such a problem. So, the problem of finding a Single-Source Shortest Path can be stated in a simplified manner via computing the weight of the shortest path rather than the path itself

$$\delta(u, v) = \min\{w(p) : p \text{ from } u \text{ to } v\}$$

The *Dijkstra* algorithm still rocks in this framework.

9.5.1 Dijkstra Algorithm - $O(|V|^2)$ using min priority queues

The Dijkstra Shortest Path *greedy* algorithm is for weighted graphs and operates by first finding the lowest-weight relationship from the start node to directly connected nodes. It keeps track of those weights and moves to the "closest" node (greedy approach). It recursively performs the same calculation, but now as a cumulative total from the start node.

NB: we don't admit negative weights!

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  for each vertex u ∈ G.V
      INSERT(Q, u)
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
          if v.d changed
              DECREASE-KEY(Q, v, v.d)
```

```
INIT-SINGLE-SOURCE(G, s)
  for each v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0
```

```
RELAX(u, v, w)
  if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
```

Figure 9.7: Dijkstra Algorithm

The **running time** is $O(|V|^2)$ if we a min priority queue. If we instead use a binary min heap is $O((|V| + |E|) \log |V|)$. This cost reduces to $O(|E| \log |V|)$ if all vertices are reachable from source node: this is convenient when $|E| << |V|^2 / \log(|V|)$. In such a case it's much more convenient to use min binary heap.

9.6 All pairs Shortest Path Algorithm: Prim's Algorithm

Prim's algorithm finds the minimum spanning tree of a graph G (i.e. a subgraph containing all the vertices but with the set of edges such that the graph stays connected and s.t. the weights are minimal) via dynamic programming. Prim's algorithm is similar to Dijkstra's Shortest Path algorithm, but rather than minimizing the total length of a path ending at each relationship, it minimizes the length of each relationship individually.

Prim's MST: Pseudo-code

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

Example based on CLRS book.

Algorithmic Methods
Master Degree in Data Science

Gianmaria Silvello

12

Figure 9.8: Prim's Algorithm.

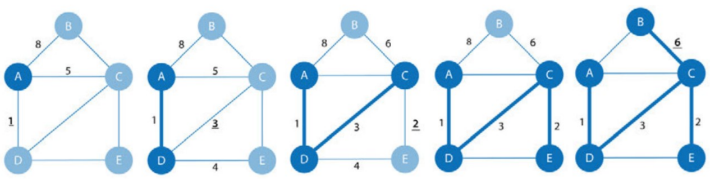


Figure 9.9: Example of Prim's Algorithm application.

9.7 Floyd Warshall Algorithm - $\Theta(|V|^3)$

Is an algorithm to find all pairs shortest path. We admit negative weights but no negative cycles.

1. We consider intermediate vertices of a path. Here we need a Lemma.

   Theorem 1. Lemma. Given a weighteed directed graph $G = (V, E)$ with $w : E \to \mathbb{R}$, let be $p =< v_0, v_1, \ldots, v_k >$ the shortest path from v_0 to v_k and for any $i, j : 0 \leq i \leq j \leq k$ let $p_{ij} =< v_i, v_{i+1}, \ldots, v_j >$ be the subpath of p from v_i to v_j. Then p_{ij} is a shortest path!

   It's easy to prove by contradiction.

2. Given a set of vertices $V = \{1, 2, \ldots, n\}$, consider a subset of V $\{1, 2, \ldots, k\}, k < n$. For any $i, j \in V$ consider all paths from i to j whose intermediate vertices are drawn from $\{1, \ldots, k\}$ and let p be the Shortest path among them.

   (a) if k is not an intermediate node of $p \Rightarrow$ all intermediate nodes are in $\{1, 2, \ldots, k-1\}$ thus, the shortest path from i to j with intermediates in $\{1, 2, \ldots, k-1\}$ is also the shortest path with intermediates in $\{1, 2, \ldots, k\}$.

   (b) If k is intermediate, by the above Lemma, $p_1 : i \to k$ is the shortest path with intermediates in $\{1, 2, \ldots, k-1\}$, and $p_2 : k \to j$ is the shortest path from k to j with intermediates in $\{1, 2, \ldots, k-1\}$.

   We can define a **recursive solution**: define:

$$d_{ij}^{(k)} := \text{Weight of the S.P. from } i \text{ to } j, \text{ with intermediates in } \{1, 2, \ldots, k\}.$$

Therefore we have, for example, that $d_{ij}^{(0)} = w_{ij}$, that is, the weight of the edge that goes from i to j.

In general we have:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0; \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases} \tag{9.1}$$

In the end, the quantity $d_{ij}^{(n)}$ gives us the length of the S.P. from i to j, $\forall i, j \in V$. Such quantities will be stored in a matrix that we call $\left(D^{(n)}\right)_{ij} = d_{ij}^{(n)}$. Note that we also have to define the quantity w_{ij} in the case where $(i, j) \notin E$. More generally we define a weight matrix W as follows:

$$w_{ij} := \begin{cases} 0 & \text{if } i = j; \\ w(i, j) & \text{if } i \neq j \wedge (i, j) \in E; \\ +\infty & \text{if } i \neq j \wedge (i, j) \notin E. \end{cases}$$

We can now see how the pseudocode for the Floyd-Warshall algorithm looks like. Note that, in the way we defined our subproblems, this is a **bottom-up** dynamic programming algorithm (indeed, we first solve the smaller subproblems, aka the subproblems for smaller values of k).

Note that with $D^{(n)}$ we mean the matrix with entries $d_{ij}^{(n)}$. Finally, we should also define a matrix $\Pi^{(k)}$ of predecessors, in order to be able not only to know the weight of the S.P. but also to reconstruct it. The update rule for such matrix is the following:

Algorithm 28 Floyd-Warshall(W)

1: $n = rows[W]$
2: $D^{(0)} = W$
3: **for** $k = 1$ **to** n **do**
4: **for** $i = 1$ **to** n **do**
5: **for** $j = 1$ **to** n **do**
6: $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
7: **return** $D^{(n)}$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Bibliography

[1] Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, "*Introduction to Algorithms, Third Edition*", 2009

[2] Goodrich, Michael T. and Tamassia, Roberto and Goldwasser, Michael H., "*Data Structures and Algorithms in Python*", 2013

[3] Donald E. Knuth. Sorting and Searching, volume 3 of "*The Art of Computer Programming*", Addison-Wesley, 1973. Second edition, 1998.