A Not-Too-Short Summary of *Algorithms*

Author: Igor L.R. Azevedo - *The University of Tokyo & University of Brasilia*
Email: *igorlima1740@gmail.com*
GitHub: https://github.com/igor17400/algorithms-174

WHAT TO EXPECT: This work aims to deliver, as the title suggests, a summary of algorithms that strikes a balance—not too short, yet not overly detailed like a comprehensive textbook. My intention was to explore essential algorithms in computer science with more depth than a typical summary offers, yet not as extensively as a canonical textbook. While this isn't just a collection of info-graphics or codes, it also isn't a divine manuscript descended from the heavens (CLRS book - "Introduction to Algorithms"). Ultimately, I hope this proves useful to someone beyond myself. If you've read this far, thank you, and stay safe!

Contents

1 Color Guide

This document uses four colors to convey specific types of information:

- Color 1 - Indicates super important information, memorize it!
- Color 2 - Used exclusively for arrows, which signify important details, curiosities, or useful symbols and information.
- Color 3 - Marks important information designed to catch your attention.
- Color 4 - Reserved for citations, links, lines, and other objects.

2 Introduction to Algorithms

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform input into the output.

An algorithm is considered **correct** if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem. Contrary to common belief, incorrect algorithms can sometimes be useful, if we can control their error rate.

$\Longrightarrow$ An algorithm can be viewed as a tool for solving a well-specified computational problem

2.1 Efficiency

When assessing the efficiency of algorithms—comparing whether algorithm A is more efficient than algorithm B—we should focus on counting fundamental operations rather than measuring time. Performance is typically expected to vary based on the size of the input. As defined by Cormen et al. (2009) "we are concerned with how the running time of an algorithm increases in the size of the input *in the limit*, as the size of the input increases without bound."

$\Longrightarrow$ $O(\cdot)$ Upper bound | $\Omega(\cdot)$ Lower Bound | $\Theta(\cdot)$ Both

$\Longrightarrow$ $o(\cdot)$ Strict upper bound | $\omega(\cdot)$ Strict lower bound

Two primary measures determine algorithm efficiency: time complexity and space complexity.

2.1.1 *Space Complexity*

The amount of time an algorithm takes to run as a function of the input size.

2.1.2 *Time Complexity*

The amount of memory an algorithm requires to run as a function of the input size.

When evaluating either type of complexity—time or space—we utilize asymptotic notation, which we define briefly below:

- Θ-notation: Represents the asymptotically tight bound of an algorithm's complexity. It defines functions that grow at the same rate asymptotically;
- O-notation: Denotes the upper bound of an algorithm's complexity. It signifies the worst-case scenario of the algorithm's growth rate.
- Ω-notation: Indicates the lower bound of an algorithm's complexity. It defines the best-case scenario of the algorithm's growth rate.
- o-notation: Describes a stricter upper bound than O-notation. It denotes functions that grow faster than a given function but not asymptotically tight.
- ω-notation: Denotes a stricter lower bound than Ω-notation. It signifies functions that grow faster than a given function but not asymptotically tight.

For example, we represent notations such as $O(n)$, $O(2^n)$, $O(n \log n)$, $O(n^2)$, $O(\log n)$, and others. The distinctions between these functions can be visualized through plots, as illustrated in the figure 1.

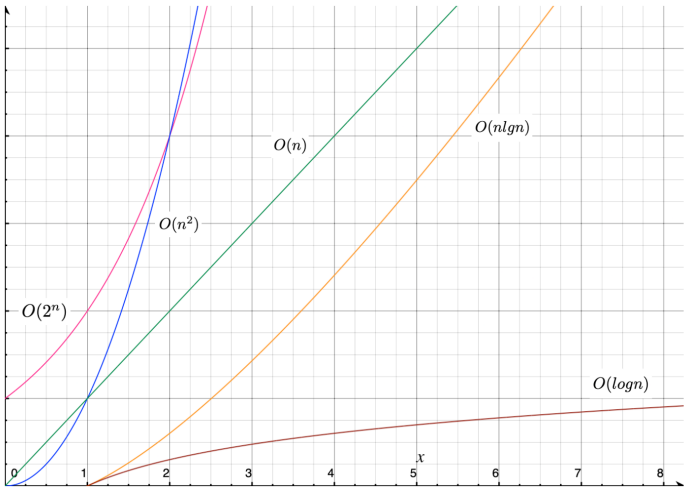

Figure 1: Comparison of complexity graph curves.

3 Interfaces & Data Structures

Using "Lecture 2: Data Structures and Dynamic Arrays" from the course "Introduction to Algorithms" Demaine et al. (2020) as inspiration, let's analyze the difference between an interface and a data structure.

Interface	**Data Structure**
• Specification • Specifies what data can be stored • Defines supported operations and their meanings • Focuses on the problem domain	• Representation • Describes how data is stored • Includes algorithms that support operations • Focuses on providing a solution to a specific problem
Main Interfaces: • Sets • Sequences	**Main Data Structure Approaches:** • Arrays • Pointer-based structures

3.1 Sequence Interface

In summary, a sequence is a collection of ordered objects. Example: $(x_0, x_1, x_2, \cdots, x_{n-1})$.It's important to note that order is extrinsic . Some special cases of sequences include: ***Stack*** and ***Queue***.

> $\Longrightarrow$ Extrinsic order in an ordered collection refers to the characteristic where the sequence and arrangement of elements are explicitly defined and maintained by the data structure itself or by operations performed on it. This means that the order of elements is not inherent to the elements themselves but is imposed externally.

Sequences differ from sets in that elements have positions within the sequence. And the following operations are supported:

1. Initialization (with a constructor)
2. Adding an element at a given position or at the end of the sequence
3. Removing an element at a given position or at the end of the sequence
4. Identifying the position of a given element of the sequence
5. Checking whether the sequence is empty

$\Longrightarrow$ Sequences do not necessarily implement order of the items attributes. That is, take the object (1) <a, b, c> and (2) <c, a, b> they are both sequences even though object (2) is not ordered. When it comes to the item's attributes, the difference is that the object is placed between two other objects. In other words, where the object is located matter .

$\Longrightarrow$ Sequences are like book shelfs (even though the books may not be organized by alphabetical order, the order they're located in the shelf inetivably matters)

3.2 Set Interface

In summary, a set is a collection of objects not necessarily in order. Sequence's about extrinsic order, set is aout intrinsic order.

$\Longrightarrow$ Intrinsic order doesn't necessarily imply any specific sorting or organization; it's simply the way data is initially arranged in a Data Structure.

Sets support (at least) the following operations:

1. Initialization (with a constructor)
2. Adding an element to the set if it's not already there (**no duplicates**)
3. Removing an element from the set
4. Checking whether a given item is in the set
5. Checking whether the set is empty

$\Longrightarrow$ A set has weaker requirements than sequences and therefroe has more implementation alternatives

$\Longrightarrow$ The order of an element in a set doesn't matter by itself. But the interpretation of where the item is located that matters.

$\Longrightarrow$ Sets are like drawers (the order doesn't matter when picking up an element)

3.3 Array Sequence

Arrays excel in *static operations* but are less efficient in dynamic scenarios. When inserting or removing items, arrays require reallocating memory and shifting all subsequent items . This process can be computationally expensive, especially with large arrays or frequent modifications.

3.4 Linked List Sequence

A pointer-based data structure where each item is stored in a node that contains a pointer to the next node in sequence. Each node consists of two fields: `node.item` and `node.next`. Manipulating nodes is straightforward as it involves relinking pointers. The structure maintains a pointer to the first node in the sequence (head). The figure 2 shows an example of a linked list.

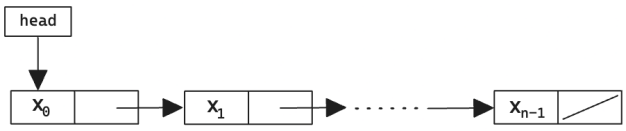

Figure 2: Linked list schematic.

3.5 Dynamic Array Sequence

In Python, a "list" is implemented as a dynamic array. To optimize dynamic operations at the end of the array, an efficient strategy involves allocating extra space to prevent frequent reallocations . This strategy maintains a fill ratio $0 \leq r \leq 1$, representing the ratio of items to allocated space. When the array reaches full capacity (i.e., $r = 1$), additional space, typically $\Theta(n)$ where n is the current size, is allocated at the end to maintain a specified fill ratio, such as 1/2.

Notably, a single operation may require $\Theta(n)$ time for reallocation. However, over any sequence of $\Theta(n)$ operations, the average time per operation becomes $\Theta(1)$. This balancing ensures efficient performance across dynamic operations on average. Take a look on section 3.6 about amortized analysis.

Data Structure	Container	Static	Dynamic		
	`build`(x)	`get_at`(i) `set_at`(i)	`insert_first`(x) `delete_first`()	`insert_last`(x) `delete_last`()	`insert_at`(i) `delete_at`(i)
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	1 (amortized)	n

Table 1: Operations comparison of different data structures - $O(\cdot)$. This table was removed from the lecture 2 notes of the class Introduction to Algorithms Demaine et al. (2020).

3.6 Amortized Analysis

Amortized analysis in algorithms is a method used to analyze the average time complexity of a sequence of operations on a data structure, particularly when individual operations can vary significantly in their time complexity. It focuses on understanding the average cost of operations over time, rather than the worst-case scenario for each operation in isolation.

In simple words we can say that it's a aata structure analysis technique to distribute cost over many operations. An operation has amortized cost $T(n)$ if k operatuions cost at most $\leq kT(n)$

"$T(n)$ amortized" roughly means $T(n)$ "on average" over many operations. Inserting into a dynamic arrayu takes $\Theta(1)$ amortized time.

$\Longrightarrow$ As an illustrative example, consider performing `insert_last()` n times on an initially empty array. The resizing occurs when the array reaches capacities like $1, 2, 4, 8, 16, \ldots, n$, with each resize incurring a cost of $\Theta(1 + 2 + 4 + 8 + \ldots + n) = \Theta\left(\sum_{i=0}^{\log n} 2^i\right) = \Theta(2^{\log n}) = \Theta(n)$.

4 Mathematical induction

Mathematical induction is a proof technique used to establish the truth of an infinite sequence of statements. It consists of two main steps:

- **Base Case:** Verify that the statement is true for the initial value, usually $n = 0$ or $n = 1$.
- **Inductive Step:** Assume that the statement is true for some arbitrary value $n = k$ (inductive hypothesis), and then prove that it is true for $n = k + 1$.

By completing these steps, one can conclude that the statement holds for all natural numbers n.

Example: Let's use mathematical induction to show that

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \tag{1}$$

for all integers $n \geq 1$.
Our induction hypothesis claims that equation 1 holds for every k. That is,

$$1 + 2 + 3 + \cdots + k = \frac{k(k+1)}{2} \tag{2}$$

Then, we want to prove that equation 1 also holds for $k + 1$.

$$1 + 2 + 3 + \cdots + (k+1) = \frac{(k+1)(k+2)}{2} \tag{3}$$

From equations 2 and 3, let's take their left-hand side and compare them.

$$\begin{aligned} 1 + 2 + 3 + \cdots + (k+1) &= 1 + 2 + 3 + \cdots + (k+1) \\ 1 + 2 + 3 + \cdots + (k+1) &= [1 + 2 + 3 + \cdots + k] + (k+1) \end{aligned} \tag{4}$$

Base Case: let's verify the base case $n = 1$. Let's then use equation 2 for this.

$$n = 1, \quad 1 = \frac{1(1+1)}{2} \Rightarrow 1 = 1 \quad (\textit{True})$$

Inductive Step: next, we need to verify that such equality also holds true for $n = k + 1$. By replacing equations 2 and 3 in equation 4 we have

$$1 + 2 + 3 + \cdots + (k + 1) = [1 + 2 + 3 + \cdots + k] + (k + 1)$$

$$1 + 2 + 3 + \cdots + (k + 1) = \frac{k(k + 1)}{2} + (k + 1)$$

$$1 + 2 + 3 + \cdots + (k + 1) = (k + 1)\Big(\frac{k}{2} + 1\Big)$$

Finally,

$$1 + 2 + 3 + \cdots + (k + 1) = \frac{(k + 1)(k + 2)}{2} \tag{5}$$

Which concludes our proof. We can see that our proof consisted on three main steps:

1. Verify that the base case (n = 1) holds true
2. Assume by our induction hypothesis that the equation 1 holds true for $n = k$
3. From the equality 4 where $n = k + 1$ our goal was to show that indeed the sequence $1 + 2 + 3 + \cdots + (k + 1)$ is equal to $\frac{(k+1)(k+2)}{2}$

5 Computing Time Complexity

Computing the time complexity of algorithms can be challenging. Understanding and analyzing the time complexity is crucial for evaluating the efficiency of algorithms. Several methods are available to compute and solve recurrences, which help in determining the time complexity of recursive algorithms. In this section, we will briefly explain some commonly used methods and give some examples to sharp our intuition.

Let's focus on some examples and rules that can help determine the running time of an algorithm.

Examples:

- **Loops**: The total running time is the product of the running time of the statements inside the loop and the number of iterations. In general, this results in $O(n)$.
- **Nested Loops**: The total running time is the product of the sizes of all the loops. For two nested loops, $T(n) = c \cdot n \cdot n = cn^2 = O(n^2)$.
- **Consecutive Statements**: Add the time complexity of each statement. Look at the pseudocode in Algorithm 1, we have $T(n) = c + cn + cn^2 = O(n^2)$
- **If-Then-Else**: The total running time will be based on the test plus the larger of either part of the if/else. Look at Algorithm 2. We can see that $T(n) = c + cn = O(n)$.

Algorithm 1 Consecutive Statements Example

```
1: print(·)                                          ▷ c
2: for i = 0 to n do
3:     print(·)                                      ▷ cn
4:     for j = 0 to n do
5:         for k = 0 to n do
6:             print(·)                              ▷ cn^2
7:         end for
8:     end for
9: end for
```

Loops: The total running time is the product of the running time of the statements inside the loop and the number of iterations. In general, this results in $O(n)$.

Nested Loops: The total running time is the product of the sizes of all the loops. For two nested loops, $T(n) = c \cdot n \cdot n = cn^2 = O(n^2)$.

Tricky Example:

Algorithm 2 If-Then-Else Example

```
print(·)                                   ▷ c
if n == 1 then
    print(·)                               ▷ c
else
    for i = 0 to n do
        print(·)                           ▷ cn
    end for
end if
```

For Loop with Logarithmic Factor: Sometimes, for loops do not result in $O(n)$ but rather $O(\log n)$. Consider the example shown at the code 3. In this example, the for loop runs i times, but at each iteration we multiply the value of i times 2. To understand the total running time, consider the following steps:

- Iteration 1: $i = 1 \rightarrow 2^0$
- Iteration 2: $i = 2 \rightarrow 2^1$
- Iteration 3: $i = 3 \rightarrow 2^2$
- Iteration 4: $i = 4 \rightarrow 2^3$
- And so on, up to iteration k: $i = n \rightarrow 2^{k-1}$

Then, we can obtain a relation between k and n.

$$n = 2^{k-1} => \log(n) = k - 1 => k = \log(n) + 1$$

Thus, we have $O(\log n)$ as the time complexity since i is not growing linearly, but exponentially.

Algorithm 3 Tricky For Loop Example

```
for i = 1 to n do
    i = i × 2
    print(·)                               ▷ O(log i)
end for
```

5.1 Divide-and-Conquer

Before diving into each method for solving the time complexity problem, let's first understand a strategy called "divide-and-conquer," which is very common in algorithms.

Divide-and-conquer is a technique where a problem is divided into smaller subproblems that are similar to the original problem but smaller in size. These subproblems are solved independently, and their solutions are combined to solve the original problem. This approach is particularly effective for problems that can be naturally divided into independent subproblems, leading to efficient recursive algorithms. Examples of divide-and-conquer algorithms include Merge Sort and Quick Sort, which use this technique to achieve optimal time complexities.

5.1.1 Recurrences

A recurrence is an equation or inequality that described a function in terms of its value on smaller inputs.

5.1.2 The maximum-subarray problem

Our goal is to find the subarray which has the largest sum, with the constraint that the subarray must consist of adjacent elements from the original array. For our problem the window size shall be 2.

5.1.3 A brute-force solution

An array of n items has $\binom{n}{2}$ such pairs of subarrays. Since $\binom{n}{2}$ is $\Theta(n^2)$, ans the best we can hope for is to evaluate each pair in constant time, this approach would take $\Omega(n^2)$ time.

5.1.4 A solution using divide-and-conquer

Let's consider the scenario described in the CLRS book Cormen et al. (2009). The goal is to determine which subarray maximizes our profit given that we buy stocks when they are cheap and sell them when the price is higher than the purchase cost. We have an array representing the price fluctuations within a given time range and another array representing the price difference (change) every two consecutive days, as shown in Table 2.

Day	0	1	2	3	···
Price	100	113	110	85	···
Change	-	13	-3	-25	···

Table 2: Stock prices and daily changes

Algorithm 4 Finding Maximum Crossing Subarray

```
1:  function FMCS(A, low, mid, high)                                  ▷ O(n)
2:      left_sum = −∞
3:      sum = 0
4:      max_left = mid
5:      for i = mid downto low do
6:          sum = sum + A[i]
7:          if sum > left_sum then
8:              left_sum = sum
9:              max_left = i
10:         end if
11:     end for
12:     right_sum = −∞
13:     sum = 0
14:     max_right = mid + 1
15:     for j = mid + 1 to high do
16:         sum = sum + A[j]
17:         if sum > right_sum then
18:             right_sum = sum
19:             max_right = j
20:         end if
21:     end for
22:     return (max_left, max_right, left_sum + right_sum)
23: end function
```

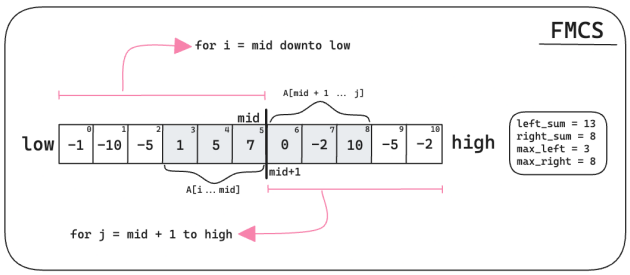

Figure 3: Find Maximum Crossing Subarray Schematic

5.1.5 The code

The pseudocodes shown in 4 and 5 complement each other. The *FMS* function focuses on the divide and conquer strategy, splitting the problem into smaller parts and solving them. The *FMCS* function, on the other hand, is responsible for calculating the maximum subarray that crosses the midpoint (mid). To analyze the algorithm's complexity, we will use a method known as the recursion tree method. For a visualization on how the algorithm works please refer to the figures **??** and 4. In addition, refer to the following presentation with a step by step illustration.

Algorithm 5 Finding Maximum Subarray

```
1:  function FMS(A, low, high)                                          ▷ Θ(n log n)
2:      if high == low then                                             ▷ Base case
3:          return (low, high, A[low])
4:      else
5:          mid = ⌊(low+high)/2⌋
6:          (left_low, left_high, left_sum) = FMS(A, low, mid)
7:          (right_low, right_high, right_sum) = FMS(A, mid + 1, high)
8:          (cross_low, cross_high, cross_sum) = FMCS(A, low, mid, high)   ▷ Fins a maximum subarray that crosses the
    midpoint
9:          if left_sum ≥ right_sum and left_sum ≥ cross_sum then
10:             return (left_low, left_high, left_sum)
11:         else if right_sum ≥ left_sum and right_sum ≥ cross_sum then
12:             return (right_low, right_high, right_sum)
13:         else
14:             return (cross_low, cross_high, cross_sum)
15:         end if
16:     end if
17: end function
```

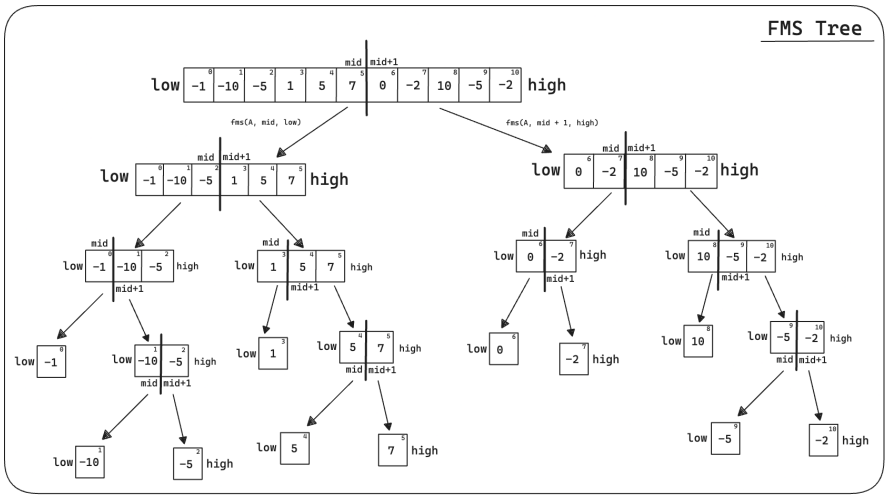

Figure 4: Find Maximum Subarray Tree Schematic

5.1.6 Analyzing the Algorithm

To perform our analysis, let's assume that the original problem size is a power of 2. Denote $T(n)$ as the running time for *FMS* on a subarray of n elements.

For the base case, when $n = 1$, we have $T(n) = \Theta(1)$ (constant time). The recursive case occurs when $n > 1$. Each subproblem is solved on a subarray of $n/2$ elements (our assumption that the original problem size is a power of 2 ensures that $n/2$ is an integer), so we spend $T(n/2)$ time solving each subproblem. Since we solve two subproblems, for the *left* and *right* subarrays, the time spent will be $2T(n/2)$.

Next, we need to analyze the time for the *FMCS* function. If the subarray $A[\text{low} \cdots \text{high}]$ contains n entries (so that $n = \text{high} - \text{low} + 1$), we claim that *FMCS*(A, low, mid, high) takes $\Theta(n)$ time. Since each iteration of the two for loops takes $\Theta(1)$ time, we just need to count how many iterations there are altogether. The first for loop makes $\text{mid} - \text{low} + 1$ iterations, and the second makes $\text{high} - \text{mid}$ iterations. Thus, the total number of iterations is:

$$(\text{mid} - \text{low} + 1) + (\text{high} - \text{mid}) = \text{high} - \text{low} + 1 = n \tag{6}$$

Hence, *FMCS* takes linear time, $\Theta(n)$.

Returning to our *FMS* method, the only remaining analysis concerns the if and else conditions, which all take constant time, $\Theta(1)$. For the recursive case, we have:

$$T(n) = \Theta(1)[\text{base case}] + 2T(n/2)[\text{solving subproblems}] + \Theta(n)[\text{FMCS}] + \Theta(1)[\text{if-else}] \tag{7}$$

Where $[\cdot]$ is the comment from where this function is being considered. Thus,

$$T(n) = 2T(n/2) + \Theta(n) \tag{8}$$

We can express our running time $T(n)$ for *FMS* as:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(n/2) + \Theta(n), & \text{if } n > 1 \end{cases} \tag{9}$$

We will explore this in more detail in the next section, but a binary tree a result from $2T(n/2)$ has $\log n + 1$ levels, and each level costs cn due to $\Theta(n)$. Therefore,

$$T(n) = cn(\log n + 1) \implies T(n) = cn \log n + cn \tag{10}$$

Which then becomes,

$$T(n) = \Theta(n \log n) \tag{11}$$

5.2 Why the height of a binary tree is $\log n$?

The height of the recursion tree is $\log n$ because at each level of the recursion, the problem size is halved. That is imagine our original problem size is a power of 2. Then

1. **Initial Problem Size**: Let's start with a problem size of n.
2. **First Level**: At the first level of recursion, the problem is divided into two subproblems, each of size $n/2$.
3. **Second Level**: Each of these subproblems is further divided into two subproblems, each of size $n/4$.
4. **Subsequent Levels**: This process continues, halving the problem size at each level.
5. **Base Case**: The recursion continues until the problem size is reduced to 1.

The number of levels in this recursion tree is the number of times we can divide n by 2 until we reach 1. This can be expressed as $\log_2 n$. Formally, if we start with a problem of size n and divide it into halves, the height h of the recursion tree is given by:

$$2^h = n \implies h = \log_2 n \tag{12}$$

So, the height of the tree is $\log n$. The height of the tree is the level of the node(s) with the longest path to to root. To get a grasp of what is going on please refer to the figure 5.

$\implies$ The level of a node is its distance (each edge counts as 1) to the root node. So the root node has level 0, its direct children have level 1 etc - (link). For binary trees the number of levels is given by $\log n + 1$

$\implies$ The height of the tree is the level of the node(s) with the longest path to to root. Stated differently, the height is the maximum of all levels - (link). For binary trees the height is given by $\log n$

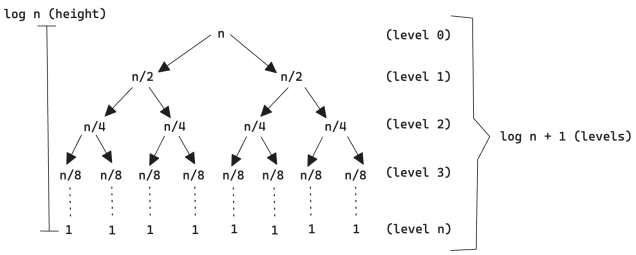

Figure 5: Height and levels of a binary tree

5.3 Recursion Tree Method

Above, we used the idea of a tree to determine the time complexity of an algorithm. Now, we will explore this method in more detail. The recursion tree method visualizes the recurrence as a *tree*, where each node represents a *subproblem*. By summing the costs of all nodes at each level of the tree, this method helps us understand the overall time complexity. It is especially useful for recurrences where the work done at each level forms a clear pattern, allowing for easy summation.

Example: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

Let's assume, as before, that the original problem size is a power of 2, so $n/4$ yields an integer. By looking at the schematic in figure **??**, we can see that at each level, we have three components, each dividing by 4 as indicated by $3T(n/4)$. At each level, we have a time complexity of $\Theta(n^2)$. Note how the subproblem size decreases by a factor of 4 each time we go down one level. We eventually must reach a base condition.

$\Longrightarrow$ How far from the root do we reach one?

We see that the subproblem for a node at depth i is $\frac{n}{4^i}$. Thus, the subproblem size hits 1 when

$$\frac{n}{4^i} = 1 \implies n = 4^i \implies \log_4 n = i \tag{13}$$

Thus, we see that the height of this tree is $\log_4 n$ and its number of levels is given by $\log_4 n + 1$ (i.e., $0, 1, 2, \cdots, \log_4 n \rightarrow$ depth i). Note that the results obtained here are different from those of a binary tree.

Next, we determine the cost at each level of the tree. Each level has three times more nodes than the level above, so the number of nodes at depth i is 3^i. Because the subproblem size reduces by a factor of 4 for each level we go down from the root, each node at depth i ($i = 0, 1, 2, \cdots, \log_4 n - 1$) has a cost of

$$\text{cost per note at } i\text{th level} = \Theta(n^2) = cn^2 = c\left(\frac{n}{4^i}\right)^2 \tag{14}$$

Next, we need to find the cost at each level considering all the nodes at each level. If we have 3^i nodes at each level and each costs $\left(\frac{n}{4^i}\right)^2$, then

$$\text{Cost at each level} = 3^i \cdot c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2 \tag{15}$$

What about the last level? The bottom level at depth $i = \log_4 n$ has

$$\text{Number of nodes at bottom level} = 3^i = 3^{\log_4 n} = n^{\log_4 3} \tag{16}$$

Each node at the bottom level contributes $T(1)$. Since $T(1)$ is a constant time, we have $\Theta(n^{\log_4 3}) = T(1) \cdot n^{\log_4 3}$.
Let's now combine all the costs:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$
$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

Where $\sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2$ is the cost per level and $\Theta(n^{\log_4 3})$ is the cost of the bottom level.
We can solve this equation from two perspectives. We know that

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \tag{17}$$

and

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \tag{18}$$

Using equation (17) we have

$$T(n) = \frac{\left(\frac{3}{16}\right)^{\log_4 n} - 1}{\left(\frac{3}{16}\right) - 1} cn^2 + \Theta(n^{\log_4 3}) \tag{19}$$

which is quite a hard equation to solve. However, if we use equation (18), which requires us to assume that our sum extends to ∞, we have

$$T(n) < \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \tag{20}$$

$$T(n) = \frac{1}{1 - \left(\frac{3}{16}\right)} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \tag{21}$$

$$T(n) = O(n^2) \tag{22}$$

Finally, the time complexity is then given by $T(n) = O(n^2)$.

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

In order to find the height of the tree, let's consider its longest path. At each level, we have $\left(\frac{2}{3}\right)^i n$ subproblems when we look at the longest path. We're interested in when this path is going to reach 1. Hence,

$$\left(\frac{2}{3}\right)^i n = 1 \implies n = \left(\frac{2}{3}\right)^i \implies \log_{3/2} n = i \tag{23}$$

Then we have that $i = \text{range}(0, \cdots, \log_{3/2} n - 1)$. Intuitively, we expect the solution to the recurrence to be at most the number of levels times the cost of each level. That is,

$$O(cn \log_{3/2} n) = O(n \lg n) \tag{24}$$

where $\lg n = \log_2 n$.

It's important to note that each level contributes to a cost of cn up to a certain point. For instance, when the shortest path reaches a subproblem size of 1, it stops contributing to the total cost in subsequent levels.

If such a tree were to be a complete binary tree of height $\log_{3/2} n$, there would be 2^i leaves where at most $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ leaves. Since the total cost of each leaf is constant, the total cost of all leaves could be $\Theta(n^{\log_{3/2} 2})$. This recursion tree is not a complete binary tree; however, it has fewer than $n^{\log_{3/2} 2}$ leaves. To calculate the precise cost of the tree would take more effort. With that in mind, we can test our assumption of $O(n \lg n)$ using the substitution method, which we shall introduce in the next session.

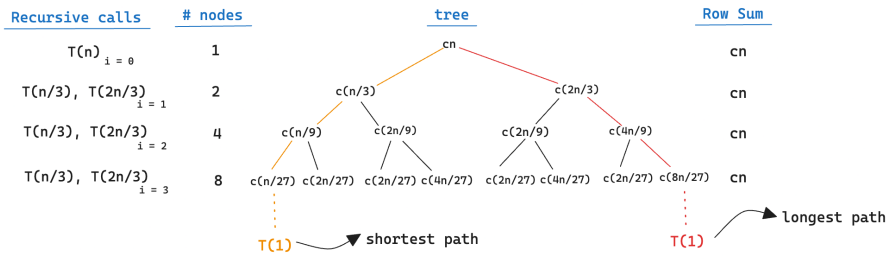

Figure 6: Recursive tree for $T(n) = T(n/3) + T(2n/3) + O(n)$

5.4 Substitution Method

The substitution method involves making an educated guess about the form of the solution and then using mathematical induction to prove that the guess is correct. This method works well for many types of recurrences and is particularly useful when the solution can be easily guessed or when the recurrence has a known solution. Let's see some examples:

Example: $T(n) = T(n/3) + T(2n/3) + O(n)$ - from divide and conquer section

As we have seen before, this example is a bit hard to solve using only the tree method. With that in mind, let's try to solve it using the substitution method. Let's test our assumption that $T(n) = O(n \lg n)$. Then,

$$T(n) = O(n \lg n) \leq d \cdot n \lg n \tag{25}$$
$$T(n) = T(n/3) + T(2n/3) + cn \tag{26}$$
$$T(n) \leq d \cdot \frac{n}{3} \lg(n/3) + d \cdot \frac{2n}{3} \lg(2n/3) + cn = \frac{dn}{3}\Big(\lg(n/3)\Big) + \frac{2dn}{3}\Big(\lg(2n/3)\Big) + cn \tag{27}$$
$$T(n) \leq \frac{dn}{3}\Big(\lg n - \lg 3\Big) + \frac{2dn}{3}\Big(\lg n + \lg 2 - \lg 3\Big) + cn \tag{28}$$
$$T(n) \leq \frac{dn}{3}\Big(\lg n - \lg 3\Big) + \frac{2dn}{3}\Big(\lg n + 1 - \lg 3\Big) + cn \tag{29}$$
$$T(n) \leq \frac{dn}{3} \lg n - \frac{dn}{3} \lg 3 + \frac{2dn}{3} \lg n + \frac{2dn}{3} - \frac{2dn}{3} \lg 3 + cn \tag{30}$$
$$T(n) \leq \Big(\frac{dn}{3} + \frac{2dn}{3}\Big) \lg n - \Big(\frac{dn}{3} + \frac{2dn}{3}\Big) \lg 3 + \frac{2dn}{3} + cn \tag{31}$$
$$T(n) \leq dn \lg n - dn \lg 3 + \frac{2dn}{3} + cn \tag{32}$$

Note, our assumption is that $T(n) \leq d \cdot n \lg n$. Thus, we need to prove that $-dn \lg 3 + \frac{2}{3}dn + cn < 0$, since we have obtained that

$$T(n) \leq dn \lg n - dn \lg 3 + \frac{2}{3}dn + cn \tag{33}$$
$$T(n) \leq dn \lg n - \Big(dn \lg 3 - \frac{2}{3}dn - cn\Big) \tag{34}$$

Hence, let's verify that $dn \lg 3 - \frac{2}{3}dn - cn > 0$

$$dn \lg 3 - \frac{2}{3}dn - cn > 0 \implies d \lg 3 - \frac{2}{3}d > c \implies d > \frac{c}{\lg 3 - \frac{2}{3}} \tag{35}$$

Therefore, by mathematical induction $T(n) \leq dn \lg n$ as long as the inequality $d > \frac{c}{\lg 3 - \frac{2}{3}}$ is satisfied.

5.4.1 Reasoning Process

I think it is easier to see what needs to be done with the substitution method when we convert the expression we want to prove into a negative format. For instance, our initial assumption is $T(n) \leq d \cdot n \lg n$, and we obtained that

$$T(n) \leq d \cdot n \lg n - d \cdot n \lg 3 + \frac{2}{3}d \cdot n + c \cdot n \tag{36}$$

Thus, we need to check that there exists some δ where $T(n) \leq d \cdot n \lg n - \delta$. Because if our assumption is that $T(n) \leq d \cdot n \lg n$, it follows that $T(n) \leq d \cdot n \lg n - \delta$.

This δ is the part of the equation that we need to prove is positive. From our previous steps, we found that

$$\delta = d \cdot n \lg 3 - \frac{2}{3}d \cdot n - c \cdot n \tag{37}$$

We just need to show that $\delta > 0$.

Example: $T(n) = 4T(\frac{n}{2}) + n$

Our first hypothesis will be that $T(n) = O(n^3)$. That is, $T(n) \leq cn^3$. Thus,

$$T(n) = 4T(\frac{n}{2}) + n \leq 4c(\frac{n}{2})^3 + n = 4c \cdot \frac{n^3}{8} + n = \frac{cn^3}{2} + n \tag{38}$$

In order for our hypothesis to be true, we need to rewrite equation above using its negative form. That is,

$$T(n) \leq cn^3 - \delta = cn^3 - \Big(\frac{cn^3}{2} - n\Big) \tag{39}$$

Note how $cn^3 - \Big(\frac{cn^3}{2} - n\Big) = \frac{cn^3}{2} + n$ which is the same we obtained in equation 38. We're rewriting our equation for an easier understanding of what we should do. Proceeding with the analysis we have,

$$\frac{cn^3}{2} - n \geq 0 \implies \frac{cn^3}{2} \geq n \implies \frac{cn^2}{2} \geq 1 \implies cn^2 \geq 2 \tag{40}$$

We see that for $n \geq 1$ and $c \geq 2$ we have $\frac{cn^3}{2} - n \geq 0$. Therefore,

$$cn^3 \geq cn^3 - \left(\frac{cn^3}{2} - n\right) \iff \frac{cn^3}{2} - n \geq 0 \tag{41}$$

By mathematical induction, if equation 41 is indeed true then we have that $T(n) \leq cn^3$.

Even though our analysis until now is correct, we might be able to find some tighter function. Thus, let's check if the hypothesis of $T(n) = O(n^2)$ fits.

$$T(n) = 4c(\frac{n}{2})^2 + n = cn^2 + n \tag{42}$$

We want to show that $T(n) \leq cn^2$. Thus,

$$T(n) \leq cn^2 \implies cn^2 + n \leq cn^2 \tag{43}$$

Let's now analyze negative format of this equation. That is,

$$cn^2 - \delta = cn^2 - (-n) \leq cn^2, \quad \text{where } \delta = -n \tag{44}$$

For equation 44 to hold, n would need to be a negative number, but n cannot be negative. Therefore, this equation doesn't hold. Let's then make a slight change to our hypothesis:

$$T(n) = O(n^2), \text{ where } T(n) \leq c_1 n^2 + c_2 n \tag{45}$$

$\implies$ Remember: for time complexity analysis, we don't consider the lower-order terms. Thus, $O(n^2) = c_1 n^2 + c_2 n + c_3 n + \cdots + c_k n = c_1 n^2$

Then, we have:

$$T(n) = 4T\left(\frac{n}{2}\right) + n \leq 4\left[c_1\left(\frac{n}{2}\right)^2 + c_2\left(\frac{n}{2}\right)\right] + n \tag{46}$$

$$T(n) \leq 4\left[c_1\left(\frac{n^2}{4}\right) + c_2\left(\frac{n}{2}\right)\right] + n = c_1 n^2 + 2c_2 n + n \tag{47}$$

Let's once again compute the negative part of the equation. That is,

$$T(n) \leq c_1 n^2 + c_2 n - \delta, \text{ where } \delta = -c_2 n - n \tag{48}$$

To show that $\delta > 0$, we have:

$$\delta > 0 \implies -c_2 n - n > 0 \implies c_2 \geq 1 \tag{49}$$

Therefore, $c_1 n^2 + c_2 n - \delta \leq c_1 n^2 + c_2 n$ as long as $\delta = -c_2 n - n$ and $c_2 \geq 1$. By mathematical induction:

$$\text{If } c_1 n^2 + c_2 n \geq c_1 n^2 + c_2 n - (-c_2 n - n) \tag{50}$$

$$\text{and If } T(n) \leq c_1 n^2 + c_2 n - (-c_2 n - n) \tag{51}$$

$$\text{then } T(n) \leq c_1 n^2 + c_2 n \tag{52}$$

Which proves that $T(n) = O(n^2)$

$\implies$ I recommend this Youtube video from for a more in depth analysis on how to solve such equations video link.

5.5 Master Method

The master method provides a straightforward way to solve recurrences of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1$ and $b > 1$. This method compares the function $f(n)$ to $n^{\log_b a}$ and determines the time complexity based on this comparison. The master method is powerful for a wide range of recurrences, making it a commonly used tool in algorithm analysis.

Case 1: $f(n) = O(n^c)$ **where** $c < \log_b a$

$$T(n) = O(n^{\log_b a}) \tag{53}$$

Case 2: $f(n) = \Theta(n^c)$ **where** $c = \log_b a$

$$T(n) = O(n^c \log n) = O(n^{\log_b a} \log n) \tag{54}$$

Case 3: $f(n) = \Omega(n^c)$ **where** $c > \log_b a$

$$T(n) = O(f(n)) \tag{55}$$

provided that $af\left(\frac{n}{b}\right) \leq kf(n)$ for some $k < 1$ and sufficiently large n.

Example 1 Consider the recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \tag{56}$$

Here, $a = 2$, $b = 2$, and $f(n) = n$. We compare $f(n)$ with $n^{\log_b a}$:

$$\log_b a = \log_2 2 = 1 \tag{57}$$

Thus, $f(n) = \Theta(n^1)$. By Case 2, we have:

$$T(n) = O(n \log n) \tag{58}$$

Example 2 Consider the recurrence:

$$T(n) = 3T\left(\frac{n}{4}\right) + n^2 \tag{59}$$

Here, $a = 3$, $b = 4$, and $f(n) = n^2$. We compare $f(n)$ with $n^{\log_b a}$:

$$\log_b a = \log_4 3 \tag{60}$$

Since $n^2 = \Omega(n^{\log_4 3})$ and $c > \log_4 3$, by Case 3, we have:

$$T(n) = O(n^2) \tag{61}$$

provided that $3\left(\frac{n}{4}\right)^2 \leq kn^2$ for some $k < 1$ and sufficiently large n.

6 Sorting

We will now explore several algorithms aimed at solving the sorting problem:

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

6.1 Set Interface

Storing items in an array in arbitrary order can implement a set (though not the most efficient approach). Sorting the stored items by key allows:

- Quick retrieval of *min* and *max* values (at the first and last index of the array).
- Rapid search using binary search, achieving $O(\log n)$ time complexity.

6.2 In-Place Algorithm

An *in-place algorithm* is one that operates directly on its input data without requiring extra space proportional to the input size. In other words, it doesn't use additional memory to perform its task. Instead, it typically rearranges elements of the input data structure, such as an array or a list, in a way that avoids needing additional storage.

$\Longrightarrow$ **Memory Efficiency**: In-place algorithms are memory efficient as they modify the input data structure without using additional space.

$\Longrightarrow$ **Space Complexity**: They typically have a space complexity of $O(1)$, meaning constant extra space is used.

Data Structure	Container	Static	Dynamic		
	`build`(x)	`find`(k)	`insert`(x) `delete`(k)	`find_min`() `find_max`()	`find_prev`(i) `find_next`(k)
Array	n	n	n	n	n
Sorted Array	n logn	logn	n	1	logn

Table 3: Operations comparison of different data structures - $O(\cdot)$. This table was removed from the lecture 3 notes of the class Introduction to Algorithms Demaine et al. (2020).

6.3 Non In-Place Algorithm

A *non in-place algorithm* requires additional space proportional to the size of the input to perform its task. It may use extra memory for creating data structures or managing state during computation.

$\Longrightarrow$ **Additional Space**: Non in-place algorithms may use $O(n)$ or more space, where n is the size of the input.

$\Longrightarrow$ **Flexibility**: They can be more flexible in approach and may achieve better time complexity by utilizing extra memory.

6.4 Permutation Sort

$\Longrightarrow$ Permutation Sort - Time complexity : $\Omega(n! \cdot n)$

The pseudocode for permutation sort is shown in Algorithm 6. There are $n!$ permutations of the set A, and at least one of these permutations will be sorted. For each permutation, we check whether it is sorted in $\Theta(n)$ time. Regarding the analysis, we can confirm its correctness by case analysis, as it tries all possibilities (brute force). Since the implementation of the function permutations(A) is unknown, we must assume $\Omega(\cdot)$ instead of $O(\cdot)$. Consequently, verifying if a permutation is sorted takes $O(n)$, and going through all possible permutations takes $\Omega(n!)$. Therefore, the total time complexity is $\Omega(n! \cdot n)$.

Algorithm 6 Permutation Sort Example

```
1: function permutation-sort(A)                          ▷ Ω(n! · n)
2:     for each permutation B in permutations(A) do      ▷ Ω(n!)
3:         if B is sorted then                           ▷ O(n)
4:             return B                                  ▷ O(1)
5:         end if
6:     end for
7: end function
```

6.5 Insertion Sort

$\Longrightarrow$ Insertion Sort (*in-place*) - Time complexity : $\Theta(n^2)$ — Space complexity : $O(1)$

Insertion Sort is a simple and intuitive sorting algorithm that builds the final sorted array (or list) one item at a time. It is much like the way you might sort playing cards in your hands. Based on the algorithm 7 let's analyze the loop invariant and the correctness of insertion sort. Loop Invariant:

- *Initialization*: it's true prior to the first iteration of the loop
- *Maintenance*: it's true before an iteration of the loop, it remains true before the next iteration
- *Termination*: when the loop terminates, the invariant give us a useful property that helps show that the algorithms is correct

The figure 7 shows a step by step schematic for sorting the array using *insertion-sort*.

Algorithm 7 Insertion Sort Example

```
1:  function insertion-sort(A)                                   ▷ Θ(n^2)
2:      for j = 1 to A.length do                                 ▷ Θ(n)
3:          key = A[j]                  ▷ insert A[j] into the sorted sequence A[1, ··· , j − 1]
4:          i = j − 1
5:          while i ≥ 0 and A[i] > key do                        ▷ Θ(n)
6:              A[i + 1] = A[i]
7:              i = i − 1
8:          end while
9:          A[i + 1] = key
10:     end for
11: end function
```

6.6 Merge Sort

$\Longrightarrow$ Merge Sort (*Non in-place*) - Time complexity : $\Theta(n \log n)$ — Space complexity : $O(n)$

Merge Sort is an efficient, stable, and comparison-based sorting algorithm that follows the *divide-and-conquer* paradigm. It divides the input array into two halves, sorts each half, and then merges the two sorted halves. The code for this algorithm was divided into two parts *merge* shown on pseudocode 8 and *merge-sort* shown on pseudocode 9. A schematic of the algorithm can seen on the figure 8. In general we have:

- Divide the problem into smaller subproblems
- Conquer the subproblems by solving them recursively
- Combine the solutions into one final answer

For the *merge* method we see that each basic step of comparing two elements at a time takes a constant time. However, we perform at most n comparissons thus merging takes $\Theta(n)$.

6.6.1 Running time of the divide-and-conquer part

We can solve this problem using the substitution method. Let's start with the hypothesis that $T(n) = O(n \log n)$. For Merge Sort, the recurrence relation is $T(n) = 2T(\lfloor n/2 \rfloor) + cn$. Assuming that the number of elements is a power of two, we have:

$$T(n) = 2T(n/2) + cn \leq dn \log n \tag{62}$$

$$T(n) \leq 2\left(\frac{dn}{2} \log\left(\frac{n}{2}\right)\right) + cn \tag{63}$$

Using the strategy of finding a negative value, we need to find a δ such that:

$$T(n) \leq 2\left(\frac{dn}{2} \log\left(\frac{n}{2}\right)\right) + cn = dn\left(\log(n) - \log(2)\right) + cn \tag{64}$$

$$T(n) \leq dn \log(n) - dn \log(2) + cn \tag{65}$$

This results in:

$$T(n) \leq dn \log(n) - \delta, \text{ where } \delta = dn \log(2) - cn \tag{66}$$

We need to show that $\delta > 0$. Assuming $\log 2 = 1$, we have:

$$\delta > 0 \implies dn - cn > 0 \implies c < d \tag{67}$$

Hence, by mathematical induction, if $T(n) \leq dn \log(n) - dn \log(2) + cn$ is true, and $dn \log(n) \geq dn \log(n) - dn \log(2) + cn$, then $T(n) \leq dn \log(n)$. Therefore, $T(n) = O(n \log(n))$.

Algorithm 8 Merge Example

```
1:  function merge(A, p, q, r)                                   ▷ Θ(n)
2:      n_1 = q − p + 1
3:      n_2 = r − q
4:      Let L[1 ... n_1 + 1] and R[1 ... n_2 + 1] be new arrays
5:      for i = 1 to n_1 do
6:          L[i] = A[p + i − 1]
7:      end for
8:      for j = 1 to n_2 do
9:          R[j] = A[q + j]
10:     end for
11:     L[n_1 + 1] = ∞                                           ▷ Sentinel
12:     R[n_2 + 1] = ∞                                           ▷ Sentinel
13:     i = 1, j = 1
14:     for k = p to r do
15:         if L[i] ≤ R[j] then
16:             A[k] = L[i]
17:             i = i + 1
18:         else
19:             A[k] = R[j]
20:             j = j + 1
21:         end if
22:     end for
23: end function
```

Algorithm 9 Merge Sort Example

```
1:  function merge-sort(A, p, r)                                 ▷ Θ(n log n)
2:      if p < r then
3:          q = ⌊(p + r)/2⌋
4:          merge-sort(A, p, q)
5:          merge-sort(A, q + 1, r)
6:          merge(A, p, q, r)
7:      end if
8:  end function
```

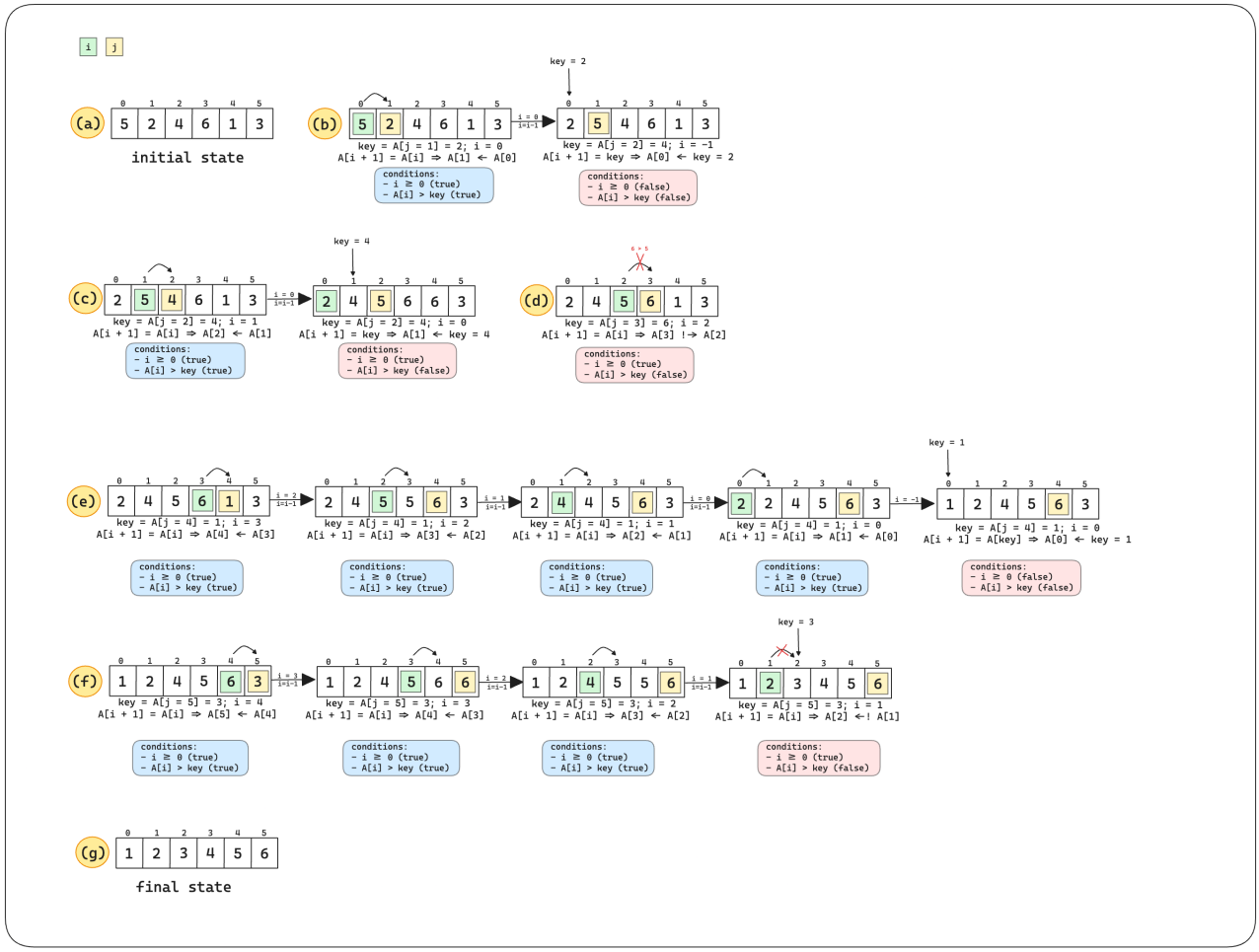

Figure 7: Insertion sort illustration.

6.7 Bubble Sort

⟹ Bubble Sort (*in-place*) - Time complexity : $\Theta(n^2)$ — Space complexity : $O(1)$

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. Its code logic can be seen on the pseudocode 10.

Algorithm 10 Bubble Sort

```
1: function bubble-sort(A, n)                ▷ A: Array to be sorted, n: Length of array
2:     for i ← 0 to n − 2 do
3:         for j ← 0 to n − i − 2 do
4:             if A[j] > A[j + 1] then
5:                 Swap A[j] and A[j + 1]
6:             end if
7:         end for
8:     end for
9: end function
```

6.7.1 Time Complexity

We start at the beginning of the array and move downward, possibly swapping elements, until we reach the previously sorted subsection of the array. Each iteration increases in steps or performs fewer comparisons. The total number of comparisons can be approximated by summing up the iterations:

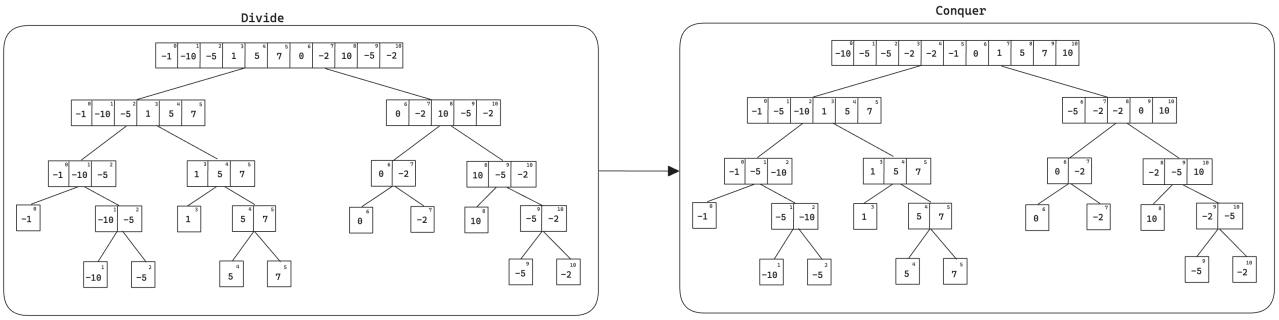

Figure 8: Merge Sort Schematic

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2} \tag{68}$$

This simplifies to $\Theta(n^2)$. This quadratic time complexity arises because in the worst case, we may need to compare each element with every other element, resulting in a total time proportional to n^2.

$\Longrightarrow$ Note that the best case scenario for insertion sort is $\Theta(n)$ whereas for bubble sort is $\Theta(n^2)$

6.8 Heap Sort

$\Longrightarrow$ Heap Sort (*in-place*) - Time complexity : $O(n \log n)$ — Space complexity : $O(1)$

6.8.1 Heaps

The **binary heap** is a data structure represented as an array that forms a nearly complete binary tree. It has some base operations as shown in pseudocode 11 and it adheres to specific heap properties:

- **max-heap**: each parent node is greater than or equal to its children.
- **min-heap**: each parent node is less than or equal to its children.

To visualize how we can conceive a binary tree given a list please refer to the schematic in figure 9.
Viewing a heap as a tree, we define the height of a noe in a heap to be the number of edges on the longest simple downward path from the node to a leaf and we define the hieght of the heap to be the height of its root.

Algorithm 11 Base of the Heap

```
1:  function parent(i)
2:      return ⌊i/2⌋
3:  end function
4:
5:  function left(i)
6:      return 2i + 1
7:  end function
8:
9:  function right(i)
10:     return 2i + 2
11: end function
```

$\Longrightarrow$ Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$

$\Longrightarrow$ The basic opearions of heaps run in a time at most proportional to the height of the tree and thus take $O(\lg n)$ time

6.8.2 Maintaining the heap property

In order to maintain the max-heap property, the procedure *max-heapify* is called. The code for it can be seen in the pseudocode 12. The operation of *max-heapify* can be seen in the figure 10

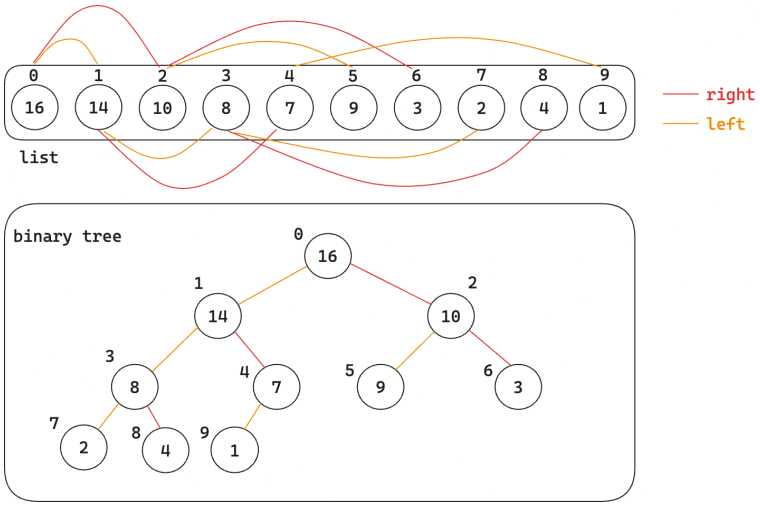

Figure 9: Illustration on how to conceive a binary tree.

Algorithm 12 Max-Heapify

```
1:  function max-heapify(A, i)
2:      largest ← i
3:      l ← left(i)
4:      r ← right(i)
5:      if l < A.heap-size and A[l] > A[largest] then
6:          largest ← l
7:      end if
8:      if r < A.heap-size and A[r] > A[largest] then
9:          largest ← r
10:     end if
11:     if largest ≠ i then
12:         Swap A[i] and A[largest]
13:         max-heapify(A, largest)
14:     end if
15: end function
```

6.8.3 Running time of *max-heapify*

The children's subtrees each have a size of at most $\frac{2n}{3}$, the worst-case scenario occurs when the bottom level of the tree is exactly half full. Thus, we can describe the running time of *max-heapify* with the recurrence relation $T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$.

To solve this recurrence relation using the master theorem, we have $a = 2$, $b = 3$, and $f(n) = \Theta(1) = k$, which implies $c = 0$. Applying case 2 of the master theorem, we get $T(n) = O(n^c \log n) = O(\log n)$. Since the height of the binary tree is $h = \log n$, we can conclude that $T(n) = O(h)$ as well.

6.8.4 Building a Heap

We can use the procedure *max-heapify* in a bottom up manner to convert an array $A[0 \cdots n]$ into a max-heap. The code can be seen on the pseudocode block 13.

Algorithm 13 Building a Max-Heap

```
1: function build-max-heap(A)
2:     A.heap-size ← A.length
3:     for i = ⌊A.length/2⌋ downto 0 do
4:         max-heapify(A, i)
5:     end for
6: end function
```

We can compute a simple upper bound on the running time of *build-max-heap* as follows:

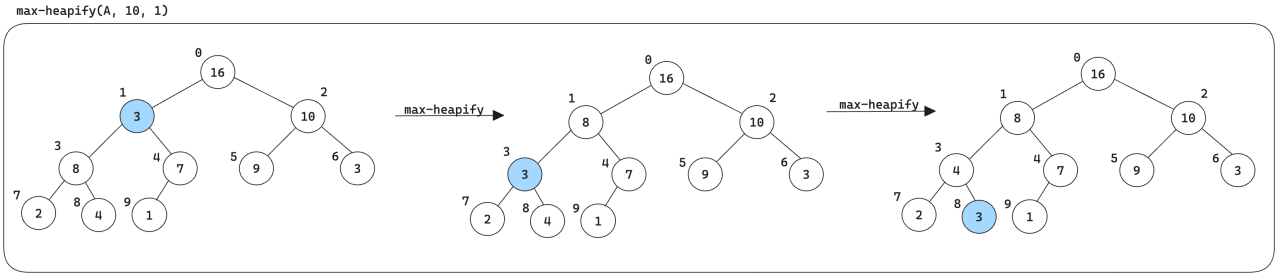

Figure 10: Illustration of how *max-heapify* works.

- Each call to *max-heapify* costs $O(\lg(n))$ time
- *build-max-heap* makes $O(n)$ such calls.
- Thus, the running time is $O(n \lg(n))$
- However, this upper bound, though correct, is not asymptotically tight

  $\Longrightarrow$ An n-element heap has height $\lfloor \lg(n) \rfloor$ and at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of any height h.

Based on the information provided above, we can derive a tighter running time for *build-max-heap*, which is linear. Specifically, *build-max-heap* has a running time of $T(n) = O(n)$. For more details on these mathematical formulations, please refer to Chapter 6, Section 4 of the CLRS book Cormen et al. (2009). Additionally, you can see how *build-max-heap* operates through the slide animation link and the figure 11.

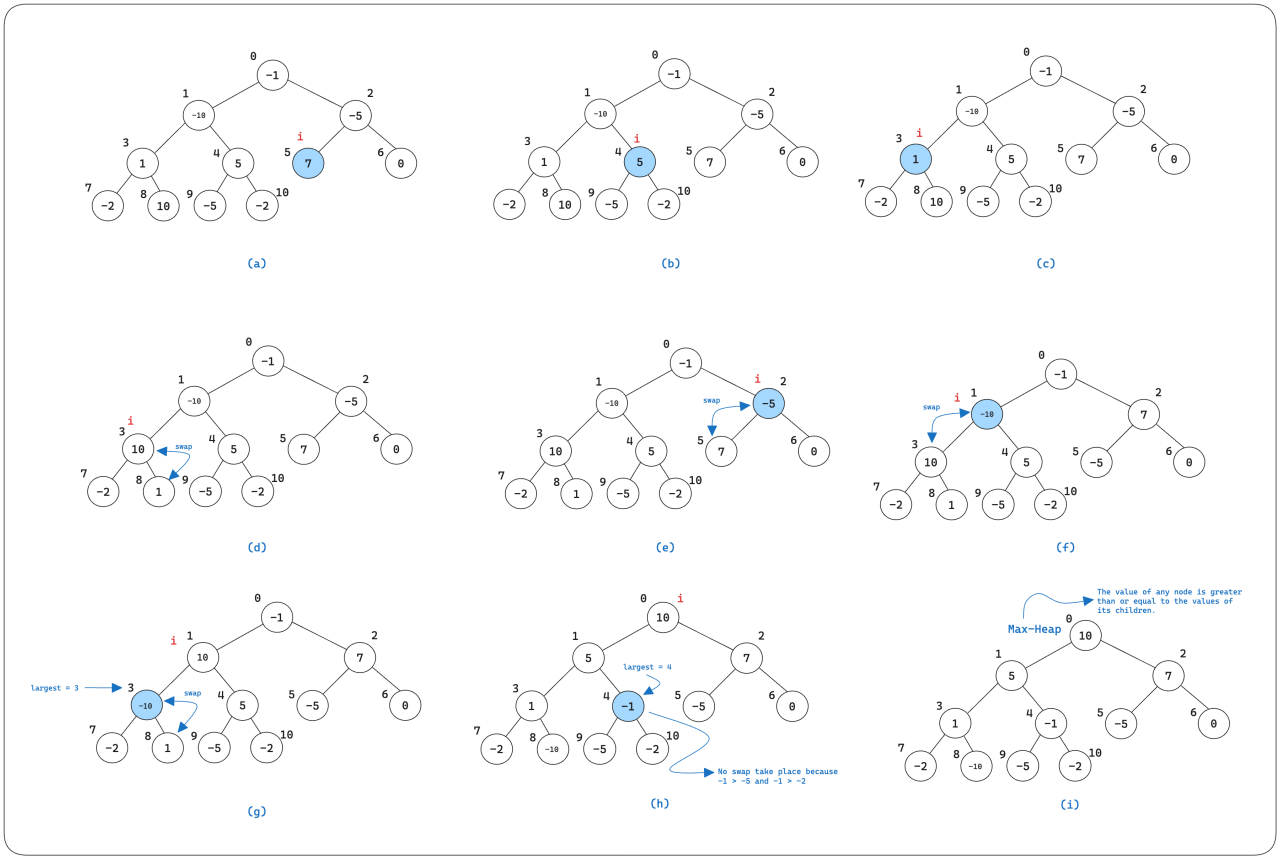

Figure 11: Illustration of how *build-max-heap* works.

6.8.5 At Last, the Heapsort Algorithm

The heapsort algorithm starts by using *build-max-heap* to build a max-heap on the input array $A[\cdots n]$, where $n = A.length$. Please refer to pesudocode 14 for the code for *heap-sort*. For the the step by step on how the heap sort works

please refer to the presentation link and for a step by step process. For a general overview of it please refer to the image 12.

Algorithm 14 Heap Sort

```
1: function heap-sort(A)
2:     build-max-heap(A)
3:     for i = A.length downto 1 do
4:         exchange A[0] with A[i]
5:         A.heap-size = A.heap-size − 1
6:         max-heapify(A, 0)
7:     end for
8: end function
```

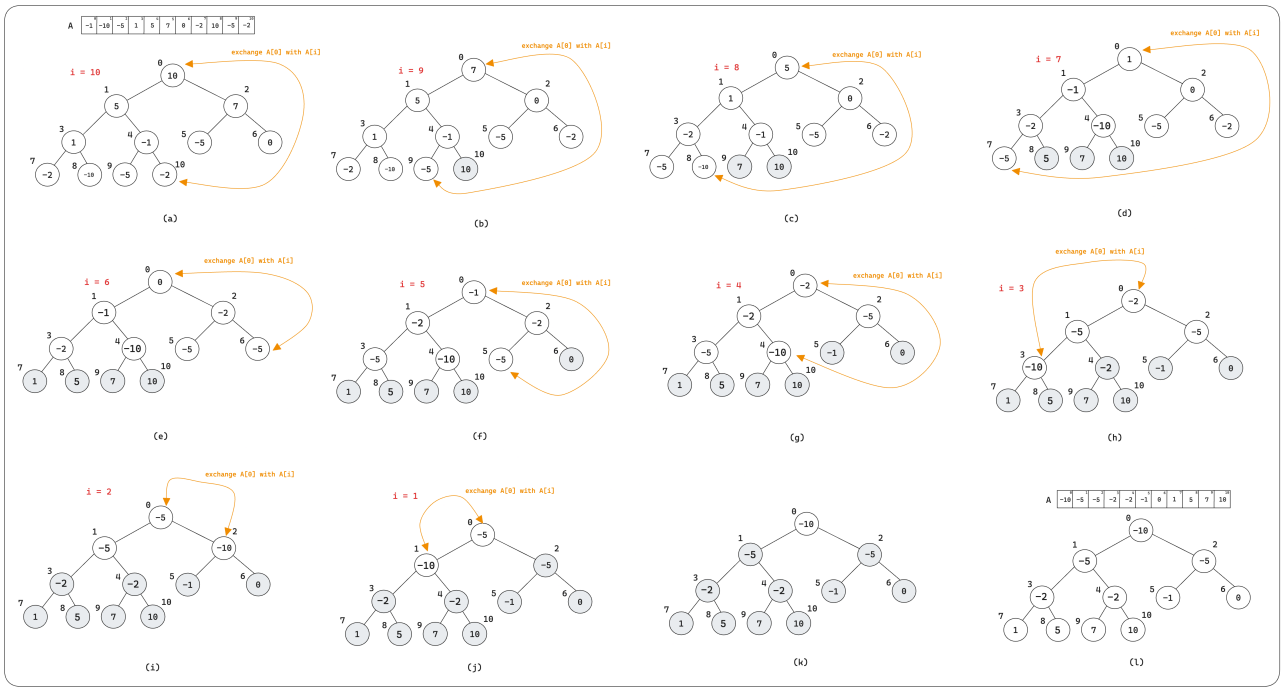

Figure 12: Illustration of the *heap sort* process.

6.9 Quick Sort

⟹ Quick Sort (*in-place*) - Time complexity : $\Theta(n \log n)$ — Space complexity : $O(1)$ — Worst case running time : $\Theta(n^2)$

Quick Sort is a highly efficient sorting algorithm that uses the divide-and-conquer strategy. The algorithm works as follows:

- Choose a pivot element from the array.
- Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.
- Recursively apply the above steps to the sub-arrays.
- Combine the sub-arrays to produce the sorted array.

The pseudcodes 15 and 16 shows how the Quick sort algorithm works. And the animation slides link show a step by step on how the algorithm operates.

Algorithm 15 Quick Sort

```
1: function quicksort(A, p, r)                                  ▷ Θ(n lg(n))
2:     if p < r then
3:         q ← partition(A, p, r)
4:         quicksort(A, p, q − 1)
5:         quicksort(A, q + 1, r)
6:     end if
7: end function
```

Algorithm 16 Quick Sort Partition

```
1: function partition(A, p, r)                                  ▷ Θ(n)
2:     x ← A[r]
3:     i ← p − 1
4:     for j = p to r do
5:         if A[j] ≤ x then
6:             exchange A[i] with A[j]
7:             i = i + 1
8:         end if
9:     end for
10:    exchange A[i + 1] with A[r]
11:    return i + 1
12: end function
```

6.9.1 Performance of quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced.

$\Longrightarrow$ If the partitioning is balanced, the algorithm runs asymptotically as fast as *merge sort*

$\Longrightarrow$ If the partitioning is unbalanced, the algorithm runs asymptotically as slowly as *insertion sort*

6.9.2 Worst-case partitioning

The worst-case scenario for quicksort occurs when the partitioning process results in one subproblem with $n-1$ elements and another with 0 elements. For example, if the array is already sorted in ascending order and the pivot chosen is always the last element, quicksort will exhibit its worst-case behavior. Such scenario yields a $\Theta(n^2)$ running time.

6.9.3 Best-case partitioning

In the best-case scenario, the partitioning process of quicksort divides the array into two nearly equal subarrays (one of size $\lfloor n/2 \rfloor$ and one of size $\lfloor n/2 \rfloor$). This means that each pivot chosen perfectly splits the array such that the sizes of the subarrays are as balanced as possible, ideally with each containing roughly half of the elements. When this occurs, the depth of the recursion is minimized, and the overall time complexity of quicksort is reduced to $\Theta(n \log n)$. That is,

$$T(n) = 2T(n/2) + \Theta(n) \implies T(n) = \Theta(n \log n) \tag{69}$$

This balanced partitioning ensures that the workload is evenly distributed across the recursive calls, resulting in efficient sorting performance.

6.9.4 Balanced partitioning

The average-case running time of quicksort is significantly closer to the best-case scenario than to the worst-case scenario. This means that the performance of quicksort largely depends on the relative ordering of the elements in the input array rather than the specific values of the elements. For instance, if the elements are distributed in such a way that each pivot results in two subarrays of nearly equal size, quicksort will efficiently sort the array. An example of this is an array where the elements are randomly shuffled, leading to balanced partitions during the sorting process.

6.9.5 A randomized version of quicksort

In examining the average-case behavior of quicksort, we assume that all permutations of the input array are equally likely. However, this assumption may not always hold in practice. To address this, we can use a randomized version of quicksort, which helps ensure that the pivot selection is not influenced by the initial ordering of the elements. By randomly selecting the pivot, we aim to achieve more balanced partitions on average, thereby improving the algorithm's performance. The pseudocode for this approach is shown in Algorithms 17 and 18.

> ⟹ I recommend reading this Quora discussion, as it includes comments from one of the authors of the CLRS book, which we are using as a reference for this summary.
>
> ⟹ **Thomas Cormen** comment – *"Both the deterministic and randomized quicksort algorithms have the same best-case running times of $O(n \lg(n))$ and the same worst-case running times of $O(n^2)$. The difference is that with the deterministic algorithm, a particular input can elicit that worst-case behavior. With the randomized algorithm, however, no input can always elicit the worst-case behavior. The reason it matters is that, depending on how partitioning is implemented, an input that is already sorted–or almost sorted–can elicit the worst-case behavior in deterministic quicksort."*

Algorithm 17 Randomized Quick Sort Partition

```
1: function randomized-partition(A, p, r)
2:     i ← random(p, r)                ▷ Return a random number between p and r
3:     exchange A[r] with A[i]
4:     return partition(A, p, r)
5: end function
```

Algorithm 18 Randomized Quick Sort

```
1: function randomized-quicksort(A, p, r)
2:     if p < r then
3:         q = randomized-partition(A, p, r)
4:         randomized-quicksort(A, p, q-1)
5:         randomized-quicksort(A, q+1, r)
6:     end if
7: end function
```

6.10 Radix Sort

⟹ Radix Sort - Time complexity: $\Theta(d \cdot (n + k))$ — Space complexity: $O(n + k)$

Radix Sort is a non-comparative sorting algorithm that sorts numbers by processing individual digits. It operates by sorting the elements digit by digit, starting from the least significant digit (LSD) to the most significant digit (MSD). To achieve this, Radix Sort leverages a stable subroutine, such as Counting Sort, to sort the elements based on each digit. This ensures that the relative order of elements with equal digits is maintained, resulting in a stable sort.

The key idea behind Radix Sort is to treat the elements as strings of digits and process each digit position independently. This approach can be highly efficient for sorting large numbers of small integers or fixed-length strings.

Here's the pseudocode for Radix Sort:

Algorithm 19 Radix Sort

```
1: procedure RadixSort(A, d)
2:     for i = 1 to d do
3:         Use a stable sorting algorithm to sort array A on digit i
4:     end for
5: end procedure
```

In this pseudocode, A is the array to be sorted, and d is the number of digits in the largest number. The algorithm iterates through each digit position and uses a stable sort to order the elements based on the current digit. The choice of a stable sorting algorithm is crucial to preserving the relative order of elements with equal digits. The figure 13 illustrates the radix sort process. Note that the highlighted section demonstrates sorting based on digit positions.

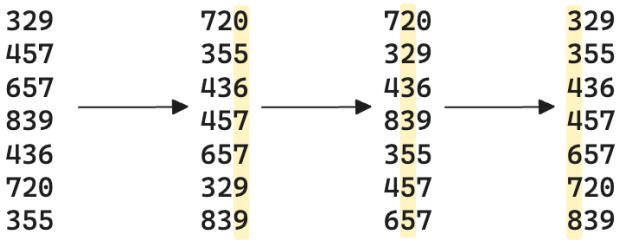

Figure 13: Illustration of the *radix sort* process.

6.11 Time complexity Comparisson

Algorithm	Time Complexity	In-place?	Stable?	Comments
Bubble Sort	$\Theta(n^2)$	Yes	Yes	Simple but inefficient for large datasets
Selection Sort	$\Theta(n^2)$	Yes	No	Always performs $\frac{n(n-1)}{2}$ comparisons
Insertion Sort	$\Theta(n^2)$	Yes	Yes	Efficient for small or nearly sorted datasets
Merge Sort	$\Theta(n \log n)$	No	Yes	Requires additional space for merging
Quick Sort	$\Theta(n \log n)$	Yes	No	Fast but worst-case $\Theta(n^2)$; randomization helps
Heap Sort	$\Theta(n \log n)$	Yes	No	Good worst-case performance, not stable
Counting Sort	$\Theta(n + u)$	No	Yes	$O(n)$ when $u = O(n)$
Radix Sort	$\Theta(d \cdot (n + k))$	No	Yes	Efficient for integers or fixed-length strings

Table 4: Comparison of Sorting Algorithms

7 Maximum Sum Subarray

Although we have previously discussed the maximum sum subarray in section 5.1.2, I would like to explore some additional algorithmic approaches that I find quite interesting.

7.1 Kadane's Algorithm

$\Longrightarrow$ Time complexity: $O(n)$ — Space complexity: $O(1)$

Kadane's algorithm is a popular and efficient method for finding the maximum sum subarray within a given array of integers. It operates in linear time , making it highly efficient for large arrays. The key idea is to use dynamic programming to keep track of the maximum sum subarray ending at each position, and then update a global maximum as needed.

7.1.1 Explanation

The algorithm maintains two variables:

- `current_max`: the maximum sum of the subarray ending at the current position.
- `global_max`: the maximum sum found so far across all positions.

For each element in the array, the algorithm updates `current_max` to be the maximum of the current element itself or the sum of `current_max` and the current element. This decision reflects whether the current element should start a new subarray or extend the existing subarray. `global_max` is then updated to be the maximum of itself and `current_max`. The pseudocode for this algorithm is shown in code block 20.

Algorithm 20 Kadane's Algorithm for Maximum Sum Subarray

```
1: function Kadane(A)                                              ▷ O(n)
2:     current_max ← A[0]
3:     global_max ← A[0]
4:     for i = 1 to A.length - 1 do
5:         current_max ← max(A[i], current_max + A[i])
6:         global_max ← max(global_max, current_max)
7:     end for
8:     return global_max
9: end function
```

8 Window's Algorithms

Window algorithms are a class of algorithms that process a subset (or "window") of elements in a sequential data structure, such as an array or a list. The window can either have a fixed size or a variable size, and it "slides" over the data structure to perform operations on different subsets of the data.

8.1 Sliding Window Fixed Size

$\Longrightarrow$ Time complexity: $O(n)$ — Space complexity: $O(1)$

The sliding window fixed size algorithm maintains a window of a fixed size as it moves through the array. This technique is useful for problems that require examining all contiguous subarrays of a fixed length. We can see one example in the pseudocode 21

Algorithm 21 Sliding Window Fixed Size

```
1: function sliding-window-fixed-size(A, k)
2:     window_sum ← 0
3:     max_sum ← −∞
4:     for i = 0 to k − 1 do
5:         window_sum ← window_sum +A[i]
6:     end for
7:     max_sum ← window_sum
8:     for i = k to length(A) - 1 do
9:         window_sum ← window_sum +A[i] − A[i − k]
10:        max_sum ← max(max_sum, window_sum)
11:    end for
12:    return max_sum
13: end function
```

Example: Given an array $A = [1, 3, 2, 6, -1, 4, 1, 8, 2]$ and window size $k = 3$, the sliding window fixed size algorithm finds the maximum sum of any contiguous subarray of size 3.

8.2 Sliding Window Variable Size

$\Longrightarrow$ Time complexity: $O(n)$ — Space complexity: $O(1)$

The sliding window variable size algorithm adjusts the window size dynamically based on some condition. This technique is useful for problems that require examining subarrays of varying lengths.

Example: Given an array $A = [4, 2, 2, 7, 8, 1, 2, 8, 1, 0]$ and sum $S = 8$, the sliding window variable size algorithm finds the length of the smallest contiguous subarray with a sum greater than or equal to 8.

8.3 Pointers

$\Longrightarrow$ Time complexity: $O(n)$ — Space complexity: $O(1)$

Algorithm 22 Sliding Window Variable Size

```
function sliding-window-variable-size(A, S)
    min_length ← ∞
    window_sum ← 0
    start ← 0
    for end = 0 to length(A) - 1 do
        window_sum ← window_sum + A[end]
        while window_sum ≥ S do
            min_length ← min(min_length, end − start + 1)
            window_sum ← window_sum − A[start]
            start ← start + 1
        end while
    end for
    return min_length if min_length ≠ ∞ else 0
end function
```

Algorithm 23 Two Pointer Technique

```
function two-pointers(A, target)
    left ← 0
    right ← length(A) − 1
    while left < right do
        current_sum ← A[left] + A[right]
        if current_sum == target then
            return left, right
        else if current_sum < target then
            left ← left + 1
        else
            right ← right − 1
        end if
    end while
    return None
end function
```

Using pointers, also known as the two-pointer technique, can efficiently solve problems involving subarrays or sublists. This method involves using two pointers to represent a window's starting and ending positions, which can be adjusted based on specific conditions.

Example: Given a sorted array $A = [1, 2, 3, 4, 6]$ and a target sum $target = 6$, the two-pointer technique finds the indices of two numbers that add up to the target sum.

9 Priority Queues

$\Longrightarrow$ Time complexity: $O(\lg(n))$ — Space complexity: $O(n)$

A priority queue is a data structure that manages a set S of elements, each associated with a value known as a key. Priority queues come in two types: max-priority queues, where the highest key value is given the highest priority, and min-priority queues, where the lowest key value is given the highest priority. These queues allow efficient access to the element with the highest (or lowest) key. They are commonly used in algorithms such as Dijkstra's shortest path algorithm and for scheduling processes in operating systems. The implementation details can be found in pseudocode 24. And an illustration can be seen in Figure **??** and a step by step execution can be seen on the following link.

Algorithm 24 Priority Queue Algorithm

```
1:  function heap-maximum(A, target)
2:      return A[0]
3:  end function
4:
5:  function heap-extract-max(A)                          ▷ O(lg(n))
6:      if A.heap-size < 1 then
7:          error "heap underflow"
8:      end if
9:      max ← A[0]
10:     A[0] ← A[A.heap-size]
11:     max-heapify(A, 0)                                 ▷ Please refet to algorithm 12
12:     return max
13: end function
14:
15: function heap-increase-key(A, i, key)
16:     if key < A[i] then error "new key is smaller than current key"
17:     end ifA[i] ← key
18:     while i > 1 and A[parent(i)] < A[i] do            ▷ Refer to algorithm 11
19:         exchange A[i] with A[parent(i)]
20:         i ← parent(i)
21:     end while
22: end function
23:
24: function max-heap-insert(A, key)
25:     A.heap-size = A.heap-size + 1
26:     A[A.heap-size] = −∞
27:     heap-increase-key(A, A.heap-size, key)
28: end function
```

10 Binary Search Trees (BST)

A binary search tree is a data structure where keys are stored in a manner that satisfies the binary-search-tree property. Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. Conversely, if y is a node in the right subtree of x, then $y.key \geq x.key$.

11 Red-Black Trees

12 AVL Trees

13 Introduction to Graph's Algorithms

14 Minimum Spanning Trees

15 Dynamic Programming

⟹ The following section uses the book Cormen et al. (2009), and the tutorials GeeksforGeeks (2024); NeetCode (2024); Stackademic (2024) as a reference.

A dynamic-programming algorithm solves each subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subproblem. Dynamic programming applies when the subproblems overlap, that is, when subproblems share subsubproblems. In this context, a **divide-and-conquer** algorithm does more work than necessary, repeatedly solving the common subproblems. When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution, typically in a bottom-up fashion
4. Construct an optimal solution from computed information

⟹ Solving the problem initially using divide and conquer is not a bad idea, considering that we optimize the code logic afterwards.

15.1 0/1 Knapsack

Example: Imagine we are going to the supermarket, and we want to buy a selection of products to take with us to the park. We have only one backpack with a certain capacity, and we want to maximize the total value of the products we can fit into our backpack. The condition here is that we can only choose one product of each type.

We enter a supermarket with a backpack that has a capacity of $m = 50$ units. There are $n = 3$ products available:

- Product 1: cost $c_1 = 60$, weight $w_1 = 10$
- Product 2: cost $c_2 = 100$, weight $w_2 = 20$
- Product 3: cost $c_3 = 120$, weight $w_3 = 30$

The goal is to determine the maximum value of the backpack when products are not allowed to be fractioned. In this example, we should take Product 2 and Product 3 to maximize the total value. Thus, the total value would be 220. The algorithm for such an approach can be seen in pseudocode 25. However this algorihtm has a time complexity of $O(2^n)$. We can improve this by using a technique called memoization.

Before showing the optimized version of this algorithm, let's see one illustration to better understand the way this algorithm works. As shown in Figure 15, this is a recursion tree for the following problem. To visualize the tree for the 0/1 knapsack problem, we'll show an example using the following data:

- **Weights:** [1, 2, 3]
- **Values:** [6, 10, 12]
- **Capacity:** 5

We'll illustrate the recursive tree step-by-step for this example which can be visualized in figure 15:

Algorithm 25 0/1 Knapsack

```
1:  function knapsack_helper(level, capacity, weights, values)
2:      if level = len(values) then
3:          return 0
4:      end if
5:                                  ▷ We're excluding the item, as including it would exceed the capacity of our backpack.
6:      if weights[level] > capacity then
7:          return knapsack_helper(level + 1, capacity, weights, values)
8:      end if
9:
10:     profit_with ← values[level]+ knapsack_helper(level + 1, capacity − weights[level], weights, values)
11:     profit_without ← knapsack_helper(level + 1, capacity, weights, values)
12:
13:     return max(profit_with, profit_without)
14: end function
15:
16: function knapsack_naive(capacity, weights, values)                    ▷ O(2^n)
17:     return knapsack_helper(0, capacity, weights, values)
18: end function
```

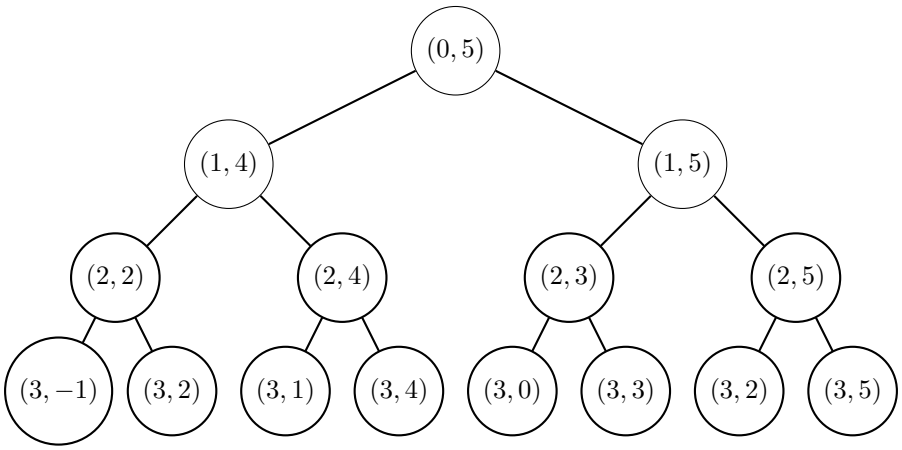

Figure 14: Illustration of the naive implementation of the 0/1 Knapsack algorithm. Each node represents a state with the current item index (level) and remaining capacity - (level, capacity) . The branches show the decision to include or exclude the current item. Each leaf represents the base case where the recursion ends, either because there are no items left to consider or the remaining capacity is zero or negative.

15.1.1 Memoization

Memoization is an optimization technique used to speed up algorithms by storing the results of expensive function calls and reusing them when the same inputs occur again. This avoids the need to recompute results for the same inputs, significantly improving the time complexity from exponential to polynomial in many cases. For the the 0/1 Knapsack problem, this can be optimized as shown in the pesudocode 26. The following list summarized the changes made:

- **Memoization Table:** A dictionary `memo` is introduced to store results of subproblems.
- **Check Memo Table:** Before computing the result, the algorithm checks if the result for the current state `(level, capacity)` is already in the memo table. If so, it returns the stored result.
- **Store Result in Memo Table:** After computing the result for a given state, it is stored in the memo table before returning.

  $\Longrightarrow$ The time complexity with memoization is $O(n \cdot m)$, where n is the number of items and m is the capacity of the knapsack. This is because each subproblem `(level, capacity)` is solved at most once, leading to a polynomial time complexity.

Algorithm 26 0/1 Knapsack with Memoization

```
function knapsack_helper(level, capacity, weights, values, memo)
    if level = len(values) then
        return 0
    end if

    if (level, capacity) in memo then                ▷ Return the value from our memo table
        return memo[(level, capacity)]
    end if
                                  ▷ We're excluding the item, as including it would exceed the capacity of our backpack.
    if weights[level] > capacity then
        memo[(level, capacity)] ← knapsack_helper(level + 1, capacity, weights, values, memo)
        return memo[(level, capacity)]
    end if

    profit_with ← values[level]+ knapsack_helper(level + 1, capacity − weights[level], weights, values)
    profit_without ← knapsack_helper(level + 1, capacity, weights, values)

    memo[(level, capacity)] ← max(profit_with, profit_without)
    return memo[(level, capacity)]
end function

function memoization(capacity, weights, values)          ▷ A 2D array with n rows and m + 1 columns
    N ← len(values)
    M ← capacity                  ▷ A 2D array with N rows and M + 1 columns, initialized with -1's
    memo ← [[−1] ∗ (M + 1) for i in range (N)]
    return memo
end function

function knapsack_naive(capacity, weights, values)                                  ▷ O(n · m)
    memo ← memoization(capacity, weights, values)
    return knapsack_helper(0, capacity, weights, values, memo)
end function
```

15.2 A More "Robust" (Dynamic Programming) Approach to 0/1 Knapsack

$\Longrightarrow$ Both the dynamic programming approach and the recursive memoization approach have time complexity of $O(n \cdot m)$

The pseudcode 27 shows an approach which is more clever using an 2D array to compute our solution. This solution is usually referred to as the true dynamic programming approach to the 0/1 Knapsack problem offers several advantages over the naive recursive approach:

- **Efficiency:** The naive recursive approach has an exponential time complexity $O(2^n)$, making it infeasible for large inputs. In contrast, the dynamic programming approach reduces the time complexity to $O(n \cdot m)$, where n is the number of items and m is the capacity of the knapsack.
- **Avoids Redundant Calculations:** The dynamic programming approach stores intermediate results in a 2D table (memoization) to avoid recalculating the same subproblems multiple times. This significantly reduces the number of calculations required.
- **Iterative Solution:** Unlike the recursive approach, the dynamic programming approach is iterative and does not involve the overhead of function calls, making it more efficient in terms of memory usage and execution speed.

The table in Figure 15 depicts the dynamic programming table for a knapsack problem with weights [1, 2, 3], values [6, 10, 12], and a capacity of 5. In this table:

- The rows represent the index of items, including a row for no items (index 0, 1, or 2).
- The columns correspond to knapsack capacities ranging from 0 to the maximum capacity.
- Each cell $dp[i][w]$ shows the maximum profit that can be obtained using the first i items with a knapsack capacity of w.

$\Longrightarrow$ For a step by step animation please refer to the following link.

Algorithm 27 0/1 Knapsack using Dynamic Programming

```
1:  function knapsack(values, weight, capacity)   ▷ Time Complexity: O(n·m) — Space Complexity: O(n·m)
2:      N ← len(values)
3:      M ← capacity                              ▷ A 2D array with N rows and M + 1 columns, initialized with 0's
4:      dp ← [[0] * (M + 1) for i in range (N)]
                                                  ▷ Fill the first column to reduce edge cases
5:      for i ← 0 to N do
6:          dp[i][0] ← 0
7:      end for                                   ▷ Fill the first row based on the first item's weight and value
8:      for c ← 0 to M + 1 do
9:          if weight[0] ≤ c then
10:             dp[0][c] ← values[0]
11:         end if
12:     end for
13:
14:     for i ← 1 to N do
15:         for c ← 1 to M + 1 do
16:             skip ← dp[i − 1][c]
17:             include ← 0
18:             if c − weight[i] ≥ 0 then
19:                 include ← value[i] + dp[i − 1][c − weight[i]]
20:             end if
21:             dp[i][c] ← max(include, skip)
22:         end for
23:     end for
24:
25:     return dp[N − 1][M]
26: end function
```

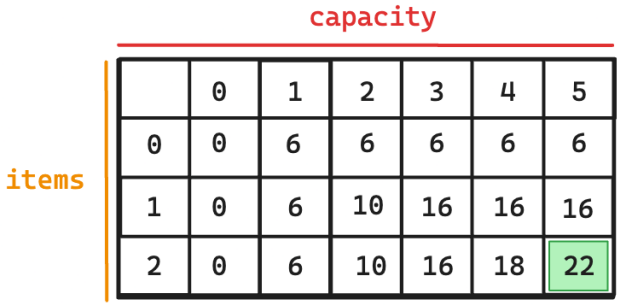

Figure 15: Dynamic programming table for the 0/1 Knapsack problem with weights [1, 2, 3], values [6, 10, 12], and capacity of 5. Each cell $dp[i][m]$ represents the maximum profit for the first i items with a capacity $m = 5$.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Demaine, E., Ku, J., and Solomon, J. (2020). 6.006: Introduction to algorithms - spring 2020. MIT OpenCourseWare. Course Description: This course is an introduction to mathematical modeling of computational problems, as well as common algorithms, algorithmic paradigms, and data structures used to solve these problems. It emphasizes the relationship between algorithms and programming and introduces basic performance measures and analysis techniques.

GeeksforGeeks (2024). 0-1 knapsack problem - dynamic programming. Accessed: 2024-07-31.

NeetCode (2024). Advanced algorithms course. Accessed: 2024-07-31.

Stackademic (2024). Knapsack problem: Dynamic programming solution. Accessed: 2024-07-31.