ALGORITHM
ANALYSIS & DESIGN

Designed by :Amr 01068542949

جامعة المنوفية

2022/2023
First Semester

كلية الحاسبات والمعلومات
Faculty of Computers &Information
Menoufia University جامعة المنوفية

Prepared by
Dr. Sondos Fadl
Menoufia University
Faculty of Computers and Information
Information Technology Department

جامعة المنوفية
كلية الحاسبات والمعلومات

رؤية الكلية

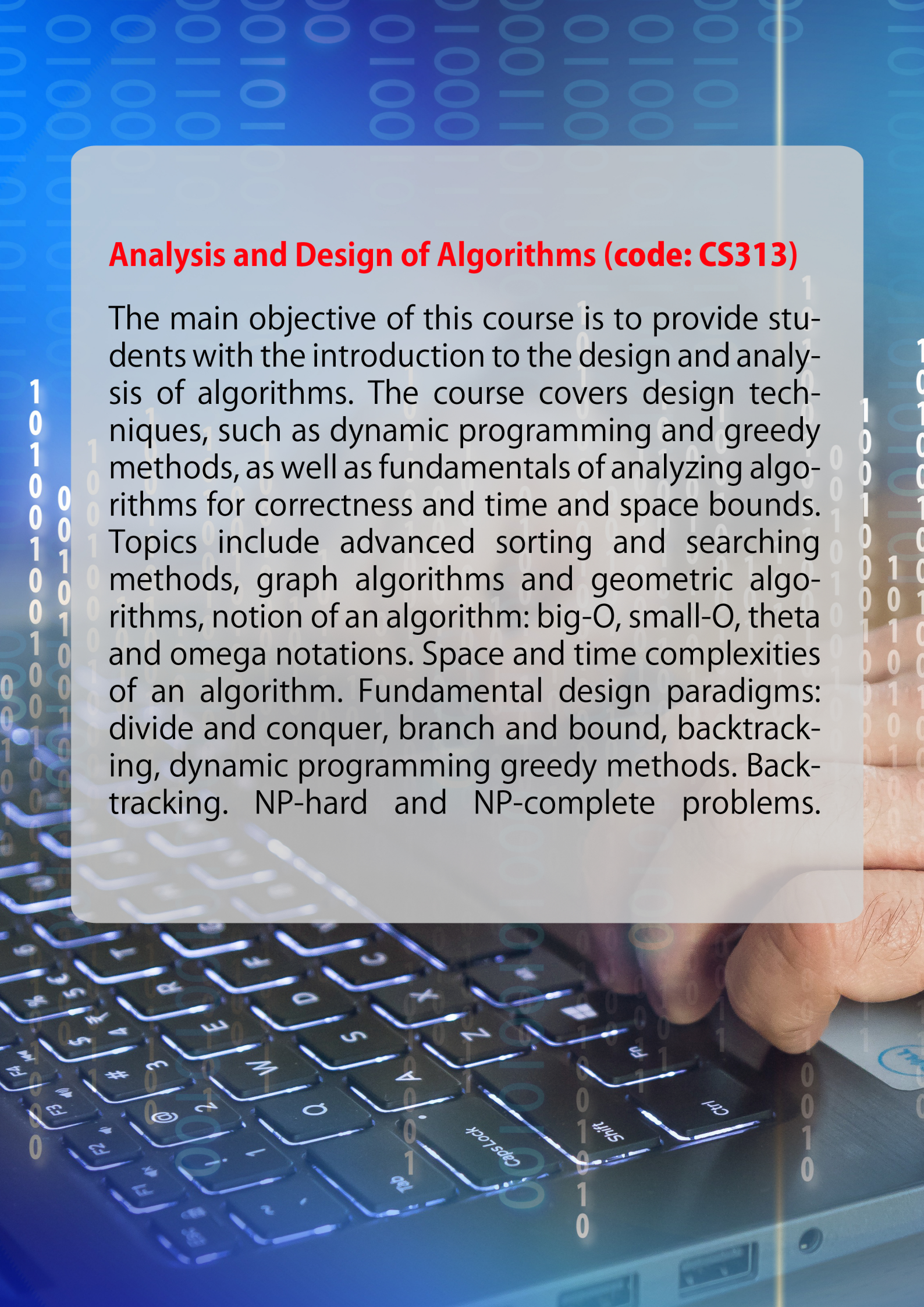
تتطلع الكلية إلى الريادة محليا واقليميا مجالات الحاسبات والمعلومات من خلال تقديم تعليم أكاديمي وبحث علمي متميز وتنمية المجتمع معرفيا ومهاريا.

رسالة الكلية

تهدف الكلية الى اعداد خريج متميز في مجالات علوم الحاسب ونظم وتكنولوجيا المعلومات ودعم القرار وقادر على توظيف مكتسباته العلمية ومهاراته العلمية لتلبية متطلبات سوق العمل ، وانتاج بحث علمي راق وتحقيق أهداف التنمية المستدامة وكسب ثقة المجتمع

Analysis and Design of Algorithms (code: CS313)

The main objective of this course is to provide students with the introduction to the design and analysis of algorithms. The course covers design techniques, such as dynamic programming and greedy methods, as well as fundamentals of analyzing algorithms for correctness and time and space bounds. Topics include advanced sorting and searching methods, graph algorithms and geometric algorithms, notion of an algorithm: big-O, small-O, theta and omega notations. Space and time complexities of an algorithm. Fundamental design paradigms: divide and conquer, branch and bound, backtracking, dynamic programming greedy methods. Backtracking. NP-hard and NP-complete problems.



Contents

1 Introduction **3**

1.1 Algorithms 3

1.2 Why is the algorithm analysis important? 5

1.2.1 Efficiency 8

1.3 Exercises 12

2 Running Time **14**

2.1 Growth of Functions 16

2.2 Calculating the running time of Algorithms 19

2.2.1 Consecutive statements 19

2.2.2 for loops 19

2.2.3 while loops 20

2.3 Real Problems 21

2.4 Summations 24

2.5 Exercises 25

3 Asymptotic Notations **27**

3.1 Big O Notation 27

3.2 Big Ω Notation 28

3.3 Big Θ Notation 28

3.4 Exercises 29

4 Recursion **31**

4.1 The recursion-tree method 32

4.2 Substitution method 33

4.2.1 Guessing method 33

4.2.2 Iteration method 35

4.3 The master method 35

4.4 Exercises 36

5 Divide and Conquer **38**

5.1 Fundamental of Divide & Conquer Strategy: 39

5.2 Applications of Divide and Conquer 39

5.2.1 QuickSort 39

5.2.2 Binary Search 42

5.3 Exercises 44

6 Dynamic Programming **45**

6.1 Overlapping Subproblems 46

6.2 Optimal Substructure 48

6.3 Longest Common Subsequence 48

6.4 Exercises 51

7 Greedy Method **53**

7.1 Activity Selection Problem 54

7.2 Exercises 57

References **58**

Chapter 1

Introduction

1.1 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 [1, 2]. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship [3]. For example, we might need to sort a sequence of numbers into nondecreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the sorting problem: Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$. Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$. For example,

given the input sequence $\langle 31, 41, 59, 26, 41, 58\rangle$, a sorting algorithm returns as output the sequence$\langle 26, 31, 41, 41, 58, 59\rangle$. Such an input sequence is called an instance of the sorting problem [4]. In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem. Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science. As a result, we have a large number of good sorting algorithms at our disposal. Which algorithm is best for a given application depends on—among other factors—the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even tapes.

An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate. Ordinarily, however, we shall be concerned only with correct algorithms. An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

1.2 Why is the algorithm analysis important?

How do you compare two algorithms for solving some problem in terms of efficiency? One way is to implement both algorithms as computer programs and then run them on a suitable range of inputs, measuring how much of the resources in question each program uses [5]. This approach is often unsatisfactory for four reasons. First, there is the effort involved in programming and testing two algorithms when at best you want to keep only one. Second, when empirically comparing two algorithms there is always the chance that one of the programs was "better written" than the other, and that the relative qualities of the underlying algorithms are not truly represented by their implementations. This is especially likely to occur when the programmer has a bias regarding the algorithms. Third, the choice of empirical test cases might unfairly favor one algorithm. Fourth, you could find that even the better of the two algorithms does not fall within your resource budget. In that case you must begin the entire process again with yet another program implementing a new algorithm. But, how would you know if any algorithm can meet the resource budget? Perhaps the problem is simply too difficult for any implementation to be within budget. These problems can often be avoided by using asymptotic analysis. Asymptotic analysis measures the efficiency of an algorithm, or its implementation as a program, as the input size becomes large [6]. It is actually an estimating technique and does not tell us anything about the relative merits of two programs where one is always "slightly faster" than the other. However, asymptotic analysis has proved useful to computer

scientists who must determine if a particular algorithm is worth considering for implementation. The critical resource for a program is most often its running time. However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the space required to run the program (both main memory and disk space). Typically you will analyze the time required for an algorithm (or the instantiation of an algorithm in the form of a program), and the space required for a data structure [8, 9].

Many factors affect the running time of a program. Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer's CPU, bus, and peripheral hardware. Competition with other users for the computer's resources can make a program slow to a crawl. The programming language and the quality of code generated by a particular compiler can have a significant effect. The "coding efficiency" of the programmer who converts the algorithm to a program can have a tremendous impact as well. If you need to get a program working within time and space constraints on a particular computer, all of these factors can be relevant. Yet, none of these factors address the differences between two algorithms or data structures. To be fair, programs derived from two algorithms for solving the same problem should both be compiled with the same compiler and run on the same computer under the same conditions. As much as possible, the same amount of care should be taken in the programming effort devoted to each program to make the implementations "equally efficient." In this sense, all of the factors mentioned above should cancel out of the comparison

because they apply to both algorithms equally. If you truly wish to understand the running time of an algorithm, there are other factors that are more appropriate to consider than machine speed, programming language, compiler, and so forth. Ideally we would measure the running time of the algorithm under standard benchmark conditions. However, we have no way to calculate the running time reliably other than to run an implementation of the algorithm on some computer. The only alternative is to use some other measure as a surrogate for running time. Of primary consideration when estimating an algorithm's performance is the number of basic operations required by the algorithm to process an input of a certain size. The terms "basic operations" and "size" are both rather vague and depend on the algorithm being analyzed. Size is often the number of inputs processed. For example, when comparing sorting algorithms, the size of the problem is typically measured by the number of records to be sorted. A basic operation must have the property that its time to complete does not depend on the particular values of its operands. Adding or comparing two integer variables are examples of basic operations in most programming languages. Summing the contents of an array containing n integers is not, because the cost depends on the value of n (i.e., the size of the input).

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer. If computers were infinitely fast, any correct method

for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice (for example, your implementation should be well designed and documented), but you would most often use whichever method was the easiest to implement. Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.

1.2.1 Efficiency

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software [7].

Example 1.2.1 We will study two algorithms for sorting. The first, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2.

The second, merge sort, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see

that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

For a concrete example, let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of 10 million numbers. (Although 10 million numbers might seem like a lot, if the numbers are eight-byte integers, then the input occupies about 80 megabytes, which fits in the memory of even an inexpensive laptop computer many times over.) Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that just an average programmer implements merge sort, using a high-level language with an inefficient compiler, with the resulting code taking $50n \lg n$ instructions. To sort 10 million

numbers, computer A

$$\frac{2 \cdot \left(10^7\right)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20,000 \text{ seconds (more than 5.5 hours)}$$

while computer B takes

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (less than 20 minutes)}$$

By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs more than 17 times faster than computer A! The advantage of merge sort is even more pronounced when we sort 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours. In general, as the problem size increases, so does the relative advantage of merge sort.

Example 1.2.2 Consider a simple algorithm to solve the problem of finding the largest value in an array of n integers. The algorithm looks at each integer in turn, saving the position of the largest value seen so far. This algorithm is called the largest-value sequential search and is illustrated by the following function:

```
largest(A) {
large = 0;
for (int i=1; i<A.length; i++)
if (A[large] < A[i])
large = i;
return large;
}
```

Here, the size of the problem is n, the number of integers stored in A. The basic operation is to compare an integer's value to that of the largest value seen so far. It is reasonable to assume that it takes a fixed amount of time to do one such comparison, regardless of the value of the two integers or their positions in the array. Because the most important factor affecting running time is normally size of the input, for a given input size n we often express the time T to run the algorithm as a function of n, written as $T(n)$. We will always assume $T(n)$ is a non-negative value. Let us call c the amount of time required to compare two integers in function largest. We do not care right now what the precise value of c might be. Nor are we concerned with the time required to increment variable i because this must be done for each value in the array, or the time for the actual assignment when a larger value is found, or the little bit of extra time taken to initialize currlarge. We just want a reasonable approximation for the time taken to execute the algorithm. The total time to run largest is therefore approximately cn, Because we must make n comparisons, with each comparison costing c time. We say that function largest (and the largest-value sequential search algorithm in general) has a running time expressed by the equation $T(n) = cn$. This equation describes the growth rate for the running time of the largestvalue sequential search algorithm.

1.3 Exercises

1. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?
a) X will always be a better choice for small inputs
b) X will always be a better choice for large inputs
c) Y will always be a better choice for small inputs
d) X will always be a better choice for all inputs

2. Which of the following best describes the useful criterion for comparing the efficiency of algorithms?
a) Time
b) Memory
c) Both of the above
d) None of the above

2. Suppose a machine on the average takes 10^{-8} seconds to execute a single algorithm step. What is the largest input size for which the machine will execute the algorithm in 2 seconds, assuming the number of steps of the algorithm is $T(n) =$
– log n
– $\sqrt{n}$
– n
– n^2
– 2^n ?

3. Suppose we are comparing implementations of insertion sort and

merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

4. What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

5. Suppose you have algorithms with the two running times listed below. (Assume these are the exact number of operations performed as a function of the input size n.) Suppose you have a computer that can perform 10^{10} operations per second, and you need to compute a result in at most an hour of computation. For each of the algorithms, what is the largest input size n for which you would be able to get the result within an hour?

– 2^n

– n^2

Chapter 2

Running Time

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

Suppose you developed a program that finds the shortest distance between two major cities of your country. You showed the program to your friend and he/she asked you "What is the running time of your program?". You answered promptly and proudly "Only 3 seconds". It sounds more practical to say the running time in seconds or minutes but is it sufficient to say the running time in time units like seconds and minutes? Did this statement fully answer the question? The answer is NO. Measuring running time like this raises so many other questions like

- What's the speed of the processor of the machine the program is running on? - What is the size of the RAM? - What is the programming language? - How experience and skillful the programmer is? - and much more

In order to fully answer your friend's question, you should say like "My program runs in 3 seconds on Intel Core i7 8-cores 4.7 GHz processor with 16 GB memory and is written in C++". Who would answer this way? Of course, no one. Running time expressed in time units has so many dependencies like a computer being used, programming language, a skill of the programmer and so on. Therefore, expressing running time in seconds or minutes makes so little sense in computer programming.

You are now convinced that "seconds" is not a good choice to measure the running time. Now the question is how should we represent the running time so that it is not affected by the speed of computers, programming languages, and skill of the programmer? In another word, how should we represent the running time so that we can abstract all those dependencies away?

The answer to the question is simple which is "input size". To solve all of these dependency problems we are going to represent the running time in terms of the input size. If the input size is n (which is always positive), then the running time is some function f of n. i.e.

$$\text{Running Time } = f(n)$$

The functional value of $f(n)$ gives the number of operations required to process the input with size n. So the running time would be the

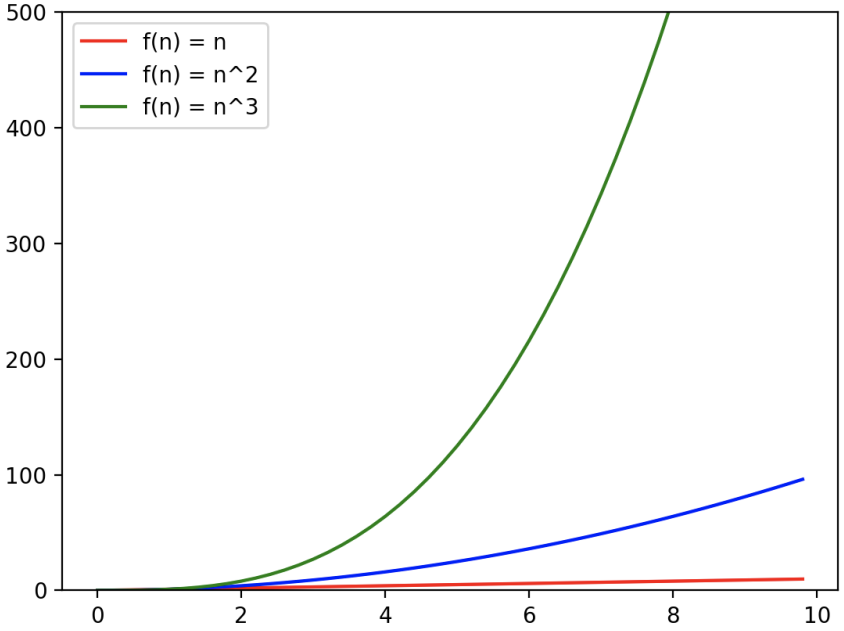
number of operations (instructions) required to carry out the given task. In the other words, "The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of steps so that it is as machine-independent as possible". Function $f(n)$ is monotonically non-decreasing. That means, if the input size increases, the running time also increases or remains constant. Some examples of the running time would be $n^2 + 2n, n^3, 3n, 2^n, \log n$, etc. Having this knowledge of running time, if anyone asks you about the running time of your program, you would say "the running time of my program is n^2 (or $2n, n \log n$ etc)" instead of "my program takes 3 seconds to run". The running time is also called a **time complexity**.

Input size informally means the number of instances in the input. For example, if we talk about sorting, the size means the number of items to be sorted. If we talk about graphs algorithms, the size means the number of nodes or edges in the graph.

2.1 Growth of Functions

In the previous section, I said that the running times are expressed in terms of the input size (n). Three possible running times are n, n^2 and n^3. Among these three running times, which one is better? In other words, which function grows slowly with the input size as compared to others? To find this out, we need to analyze the growth of the functions i.e we want to find out, if the input increases, how quickly the running time goes up. One easiest way of comparing different running times is to plot them and see the natures of the

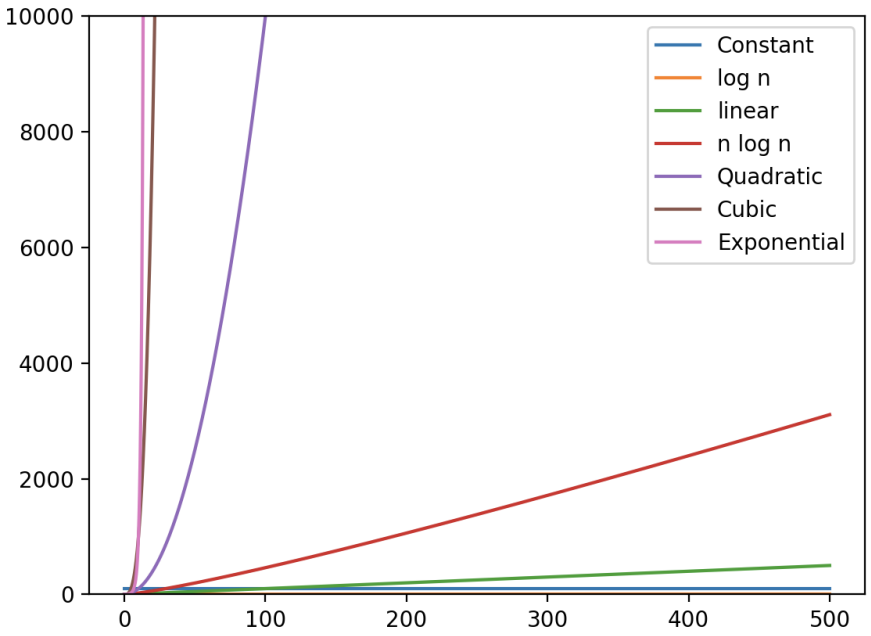
graph. The following figure shows the graphs of n, n^2 and n^3. (x-axis represents the size of the input and y-axis represents the number of operation required i.e. running time) Looking at the figure above,



we can clearly see the function n^3 is growing faster than functions n and n^2. Therefore, running time n is better than running times n^2, n^3. One thing to note here is the input size is very small. I deliberately use the small input size only to illustrate the concept. In computer science especially in the analysis of algorithms, we do the analysis for very large input size.

The table below shows common running times in algorithm analysis. The entities in the table are presented from slower to quicker (best to worst) running times.

Running Time	**Examples**
Constant	$1, 2, 100, 300, \ldots$
Logarithmic	$\log n, 5 \log n, \ldots$
Linear	$n, n + 3, 2n + 3, \ldots$
Polynomial	Quadratic, Cubic, or higher order
Exponential	$2^n, 3^n, 2^n + n^4, \ldots$
Factorial	$\mathrm{n!}, \mathrm{n!} + \mathrm{n}, \ldots$



The figure shows the graphical representations of these running times functions.

2.2 Calculating the running time of Algorithms

2.2.1 Consecutive statements

Let two independent consecutive statements are P_1 and P_2. Let t_1 be the cost of running P_1 and t_2 be the cost of running P_2. The total cost of the program is the addition of cost of individual statement i.e. $t_1 + t_2$. Example: Consider the following code.

int main() {
//1. some code with running time n
//2. some code with running time n^2
return 0;
}

Assume that statement 2 is independent of statement 1 and statement 1 executes first followed by statement 2. The total running time is $T(n) = n + n^2$

2.2.2 for loops

It is relatively easier to compute the running time of for loop than any other loops. All we need to compute the running time is how many times the statement inside the loop body is executed.

for $(i = 0; i < 10; i++)$ **{**
// body
}

The loop body is executed 10 times. If it takes m operations to run the body, the total number of operations is $10 \times m = 10$ m. In general, if the loop iterates n times and the running time of the loop

body are m, the total cost of the program is $n * m$. Please note that we are ignoring the time taken by expression $i = 0, i < 10$ and statement $i++$. If we include these, the total time becomes

$$T(n) = 1 + n + n + 1 + mn$$

In this analysis, we made one important assumption. We assumed that the body of the loop doesn't depend on i. Sometimes the run-time of the body does depend on i. In that case, our calculation becomes a little bit difficult. Consider an example shown below.

for $(i = 0; i < n; i++)$ **{**
if $(i\%2 == 0)$ **{**
print 'Hi'
}
}

In the for loop above, the control goes inside the if condition only when i is an even number. That means the body of if condition gets executed $n/2$ times. The total cost is therefore $T(n) = 1 * n/2 = n/2$.

2.2.3 while loops

while loops are usually harder to analyze than for loops because there is no obvious a priori way to know how many times we shall have to go round the loop. One way of analyzing while loops is to find a variable that goes increasing or decreasing until the terminating condition is met. Consider an example given below

while $(i > 0)$ **{**
// some computation of cost n

i = i / 2

}

How many times the loop repeats? In every iteration, the value of i gets halved. If the initial value of i is 16 , after 4 iterations it becomes 1 and the loop terminates. The implies that the loop repeats $\log_2 i$ times. In each iteration, it does the n work. Therefore the total cost is $T(n) = n * \log_2 n$.

2.3 Real Problems

We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each $j = 1, 2, 3, ..., n$, where $n = length(A)$, we let t_j denote the number of times the while loop test in line 5 is executed for that value of j. When a for or while loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

	times
for $j = 2$ **to** $A.length$	n
$key = A[j]$	$n-1$
$i = j - 1$	$n-1$
while $i > 0$ and $A[i] > key$	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	$n-1$

The running time of the algorithm is the sum of running times

for each statement executed. To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of times columns, obtaining

$$\begin{aligned} T(n) =& n + (n-1) + (n-1) + \sum_{j=2}^{n} t_j + \sum_{j=2}^{n} (t_j - 1) \\ &+ \sum_{j=2}^{n} (t_j - 1) + (n-1) \end{aligned}$$

Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given. For example, in INSERTION-SORT, the **best case** occurs if the array is already sorted. For each $j = 1, 2, 3, ..., n$, we then find that $A[i] \leq key$ when i has its initial value of $j - 1$. Thus $t_j = 1$ for $j = 1, 2, 3, ..., n$, and the best-case running time is

$$\begin{aligned} T(n) &= n + (n-1) + (n-1) + (n-1) + (n-1) \\ &= 5n - 4 = \Theta(n) \end{aligned}$$

We can express this running time as an + b for constants a and b that depend on the statement costs; it is thus a linear function of n.

If the array is in reverse sorted order—that is, in decreasing order—the **worst case** results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \ldots j-1]$, and so $t_j = j$ for $j = 1, 2, 3, ..., n$. Noting that

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

and

$$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) =& n + (n-1) + (n-1) + \left(\frac{n(n+1)}{2} - 1\right) \\ &+ \left(\frac{n(n-1)}{2}\right) + \left(\frac{n(n-1)}{2}\right) + (n-1) = O(n^2) \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs; it is thus a quadratic function of n.

Sometimes we do the average case analysis on algorithms. Most of the time the average case is roughly as bad as the worst case. In the case of insertion sort, when we try to insert a new item to its appropriate position, we compare the new item with half of the sorted item on average. The complexity is still in the order of n^2 which is the worst-case running time.

It is usually harder to analyze the average behavior of an algorithm than to analyze its behavior in the worst case. This is because it may not be apparent what constitutes an "average" input for a particular problem. A useful analysis of the average behavior of an algorithm, therefore, requires a prior knowledge of the distribution of the input instances which is an unrealistic requirement. Therefore often we assume that all inputs of a given size are equally likely and do the probabilistic analysis for the average case.

2.4 Summations

When an algorithm contains an iterative control construct such as a while or for loop, we can express its running time as the sum of the times spent on each execution of the body of the loop. For example, we found in the insertion sort algorithm that the j th iteration of insertion sort took time proportional to j in the worst case. By adding up the time spent on each iteration, we obtained the summation (or series) $\sum_{j=2}^{n} j$. Some important summation formulas given below:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} \text{c or } n = n * n = n^2$$

$$\sum_{i=1}^{\log_2 n} 2^i = 2n - 1$$

$$\sum_{i=1}^{\log_2 n} \text{c } = c * \log_2 n$$

$$\sum_{i=j}^{n} 1 = n - j + 1$$

$$\sum_{i=1}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

$$\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6}$$

$$\sum_{i=1}^{n} \frac{1}{i} = \log n + 1$$

2.5 Exercises

1. How is time complexity measured?
a) By counting the number of algorithms in an algorithm.
b) By counting the number of primitive operations performed by the algorithm on given input size.
c) By counting the size of data input to the algorithm.
d) None of the above

2. What is the time, space complexity of the following code:
fun(a,b){
for $(i = 0; i < n; i++)$ {
$a = a + 2;$}
for $(j = 0; j < m; j++)$ {
$b = b + 3;$}
}

3. What is the best and worst case of the following codes:
a- int search(A, n, item) {
for $(i = 0; i < n; i++)$ {
if $(A[i] == item)$ {
return $A[i]$ }
}
return -1
}

b- for $(i = 0; i < N; i++)$ {

for $(j = N; j > i; j - -)$ {
$a = a + i + j;$
}
}

4. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n - 1$ elements of A. Write pseudocode for this algorithm, which is known as selection sort. Give the best-case and worst-case running times of selection sort.

5. Compute the running time of the function that find the max element in array A.

Chapter 3

Asymptotic Notations

We will use asymptotic notation primarily to describe the running times of algorithms, as when we wrote that insertion sort's worst-case running time is $O\left(n^2\right)$. Asymptotic notation actually applies to functions, however. Recall that we characterized insertion sort's worst-case running time as $an^2 + bn + c$, for some constants a, b, and c. By writing that insertion sort's running time is $O\left(n^2\right)$, we abstracted away some details of this function. Because asymptotic notation applies to functions, what we were writing as $O\left(n^2\right)$ was the function an^2+bn+c, which in that case happened to characterize the worst-case running time of insertion sort. There are three notations that are commonly used.

3.1 Big O Notation

Big-Oh (O) notation gives an upper bound for a function $f(n)$ to within a constant factor. We write $f(n) = O(g(n))$, If there are positive constants n_0 and c such that, to the right of n_0 the $f(n)$ always lies on or below $c^*g(n)$.

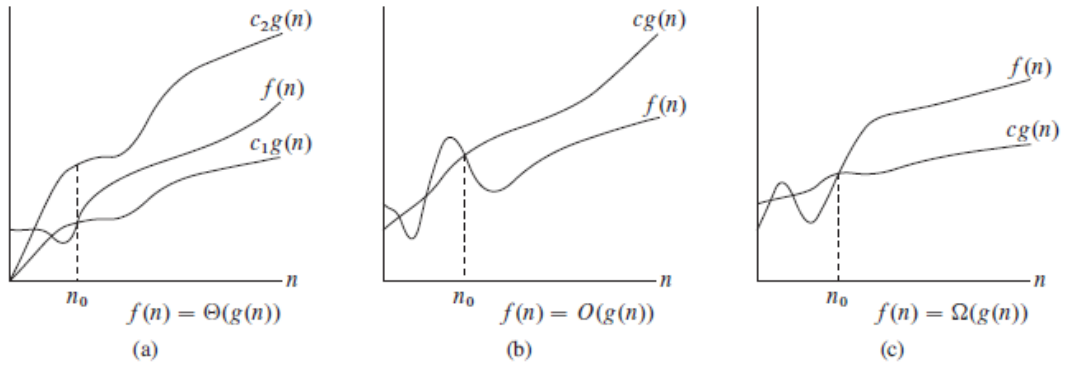
$O(g(n)) = \{f(n)$: there exist positive constants c and n_0
such that $f(n) \leq cg(n)$ for all $n \geq n_0\}$
For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)). Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

3.2 Big Ω Notation

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor. We write $f(n) = \Omega(g(n))$, If there are positive constants n_0 and c such that, to the right of n_0 the $f(n)$ always lies on or above $c * g(n)$.
$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0
such that $cg(n) \leq f(n)$ for all $n \geq n_0\}$
For any value of n, the minimum time required by the algorithm is given by Omega $\Omega(g(n))$.

3.3 Big Θ Notation

Big-Theta(Θ) notation gives tight bound for a function f(n) to within a constant factor. We write $f(n) = \Theta(g(n))$, If there are positive constants n_0 and $c1$ and $c2$ such that, to the right of n_0 the $f(n)$ always lies between $c1 * g(n)$ and $c2 * g(n)$ inclusive.
$\theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0
such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$
If a function $f(n)$ lies anywhere in between $c_1 g(n)$ and $c_2 g(n)$ for all $n \geq n_0$, then $f(n)$ is said to be asymptotically tight bound. ————



3.4 Exercises

1. Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

2. Decide whether these statements are True or False. You must briefly justify all your answers:

a) $f(n) = 3n + 4 = \mathrm{O(n)}$

b) $f(n) = 3n - 2 = \mathrm{O(n)}$

c) $f(n) = an + b = O\left(n^2\right)$

d) $f(n) = 100n + 6 = \mathrm{O(n)}$

e) $f(n) = 6 * 2^n + n^2 = O\left(2^n\right)$

f) $f(n) = 3n^2 + n = \Omega\left(n^2\right)$

g) $f(n) = 6n^3 \neq \theta\left(n^2\right)$

3. Indicate, for each pair of expressions (A, B) in the table below, whether A is O, Ωor Θ of B. Log are base 2. Your answer should be in the form of the table with "yes" or "no" written in each box and prove your answer

A	**B**	***O***	**Ω**	Θ
2^{N+1}	2^{N}			
$N(\log(N))^2$	NLog(N)			

4. Write the big- Θ expression (tight bound) to describe the number of operations required for the following algorithms: "Show the detailed analysis of the solution not just the final order"

Fun(n)

{ For $i = 1 to n$

{ $j = n$

While $j > i$

{$j = j$–1

} } }

Chapter 4

Recursion

In the previous chapters, we discussed analysis of statments, conditions, loops, etc. Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as $T(n) = 2T(n/2) + cn$. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

This chapter offers three methods for solving recurrences:

1. 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.
2. **Substitution method** using iteration and Guessing techniqes.
3. 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. A recurrence of the form in the previous equation that creates a subproblems, 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.

4.1 The recursion-tree method

The Recursion Tree Method is a way of solving recurrence relations. In this method, a recurrence relation is converted into recursive trees. Each node represents the cost incurred at various levels of recursion. To find the total cost, costs of all levels are summed up.
Foe example, we have the running time as: $T(n) = 2T(n/2) + c$
and we need to solve recurrence relation using Tree method by follow the following steps:
Step 1: Draw a recursive tree
Step 2: Calculate the work done or cost at each level and count total no of levels in recursion tree
Step 3: Count the total number of levels:
To Compute the height of the tree by number of rules:
1. If the recursion part in equation contain division as $T(n) = aT(n/b) + cn$, so the hight of tree equal: $height = \log_b n$.
2. If the recursion part in equation contain subtraction as $T(n) = aT(n-b) + cn$ the hight of tree equal: $height = n - b + 1$.
Step 4: calculate cost of each level
Step 5: Sum up the cost all the levels in recursive tree.

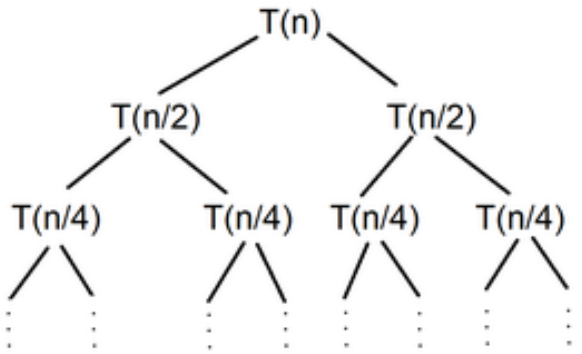


Figure 4.1: Step 1:Recursive tree

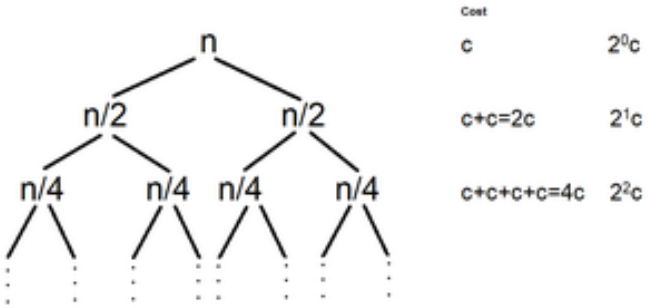


Figure 4.2: Step 2:Recursive Tree with each level cost

4.2 Substitution method

4.2.1 Guessing method

A lot of things in this class reduce to induction. In the substitution method for solving recurrences we 1. Guess the form of the solution. 2. Use mathematical induction to find the constants and show that the solution works.

Example: $T(n) = 2T(\lfloor n/2 \rfloor) + n$ for $n > 1$. We guess that the solution is $T(n) = O(n \log n)$. So we must prove that $T(n) \leq cn \log n$ for some constant c. (We will get to n_0 later, but for— now let's try to prove the statement for all $n \geq 1$.)

As our inductive hypothesis, we assume $T(n) \leq cn \log n$ for all positive numbers less than n. Therefore, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor))$, and

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n \quad (\text{ for } c \geq 1) \end{aligned} \tag{4.1}$$

Now we need to show the base case. This is tricky, because if $T(n) \leq cn \log n$, then $T(1) \leq 0$, which is not a thing. So we revise our induction so that we only prove the statement for $n \geq 2$, and the base cases of the induction proof (which is not the same as the base case of the recurrence!) are $n = 2$ and $n = 3$. (We are allowed to do this because asymptotic notation only requires us to prove our statement for $n \geq n_0$, and we can set $n_0 = 2$.)

We choose $n = 2$ and $n = 3$ for our base cases because when we expand the recurrence formula, we will always go through either $n = 2$ or $n = 3$ before we hit the case where $n = 1$.

So proving the inductive step as above, plus proving the bound works for $n = 2$ and $n = 3$, suffices for our proof that the bound works for all $n > 1$.

Plugging the numbers into the recurrence formula, we get $T(2) = 2T(1) + 2 = 4$ and $T(3) = 2T(1) + 3 = 5$. So now we just need to choose a c that satisfies those constraints on $T(2)$ and $T(3)$. We can choose $c = 2$, because $4 \leq 2 \cdot 2 \log 2$ and $5 \leq 2 \cdot 3 \log 3$. Therefore, we have shown that $T(n) \leq 2n \log n$ for all $n \geq 2$, so $T(n) = O(n \log n)$.

4.2.2 Iteration method

It means to expand the recurrence and express it as a summation of terms of n and initial condition. Deduce the complexity by iteratively substitute the T(N) terms until reaching general formula. Then, get the complexity using both the formula and the termination condition.
Example:

$$\begin{gathered} T(n) = 2T\left(\frac{n}{2}\right) + \mathrm{n} \\ T(n) = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + \mathrm{n} = 4T\left(\frac{n}{4}\right) + n + n = 4T\left(\frac{n}{4}\right) + 2n \\ T(n) = 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2\mathrm{n} = 8\ \mathrm{T}\left(\frac{n}{8}\right) + 3n \\ T(n) = 2^w T\left(\frac{n}{2^w}\right) + wn \\ T(n) = 2^{\log n} + n\log n \\ T(n) = n^{\log 2} + n\log n = n + n\log n \\ \mathrm{O}(n\log n) \end{gathered} \tag{4.2}$$

4.3 The master method

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$$T(n) = aT(n/b) + f(n) \text{ where } a >= 1 \text{ and } b > 1$$

There are following three cases:
1. If $f(n) = O\left(n^c\right)$ where $c < \log_b a$ then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta\left(n^c\right)$ where $c = \log_b a$ then $T(n) = \Theta\left(n^c \log n\right)$

3. If $f(n) = \Omega(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$

Examples of some standard algorithms whose time complexity can be evaluated using Master Method:
Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $\log_b a$ is also 1. So the solution is $\Theta(n \log n)$
Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $\log_b a$ is also 0 . So the solution is $\Theta(\log n)$

It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + n/\log n$ cannot be solved using master method.
Case 2 can be extended for $f(n) = \Theta\left(n^c \log^k n\right)$ If $f(n) = \Theta\left(n^c \log^k n\right)$ for some constant $k >= 0$ and $c = \log_b a$, then $T(n) = \Theta\left(n^c \log^{k+1} n\right)$

4.4 Exercises

2. Use a recursion tree to give an asymptotically tight solution to the following recurrence:
1. $T(n) = T(n/4) + T(n/2) + cn^2$
2. $T(n) = 3T(n/4) + cn^2$
3. $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$
4. $T(n) = 4T\left(\frac{n}{2}\right) + n$

3. For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

1. $T(n) = 3T(n/2) + n^2$
2. $T(n) = 4T(n/2) + n^2$
3. $T(n) = T(n/2) + 2^n$
4. $T(n) = 2^n T(n/2) + n^n$
5. $T(n) = 16T(n/4) + n$
6. $T(n) = 2T(n/2) + n \log n$
7. $T(n) = 2T(n/2) + n/\log n$
8. $T(n) = 2T(n/4) + n^{0.51}$
9. $T(n) = 0.5T(n/2) + 1/n$
10. $T(n) = \sqrt{2}T(n/2) + \log n$
11. $T(n) = 3T(n/2) + n$
12. $T(n) = 3T(n/3) + \sqrt{n}$
13. $T(n) = 4T(n/2) + cn$
14. $T(n) = 3T(n/4) + n \log n$
15. $T(n) = 3T(n/3) + n/2$
16. $T(n) = 6T(n/3) + n^2 \log n$
17. $T(n) = 4T(n/2) + n/\log n$
18. $T(n) = 64T(n/8) - n^2 \log n$
19. $T(n) = 7T(n/3) + n^2$
20. $T(n) = 4T(n/2) + \log n$

Chapter 5

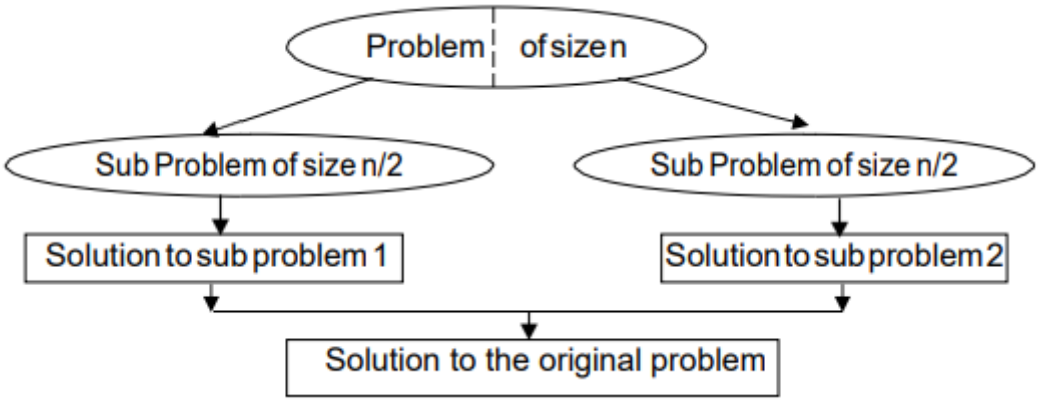
Divide and Conquer

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy. Divide and Conquer algorithm consists of a dispute using the following three steps:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.



5.1 Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy: Relational Formula and Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

5.2 Applications of Divide and Conquer

5.2.1 QuickSort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different

ways.

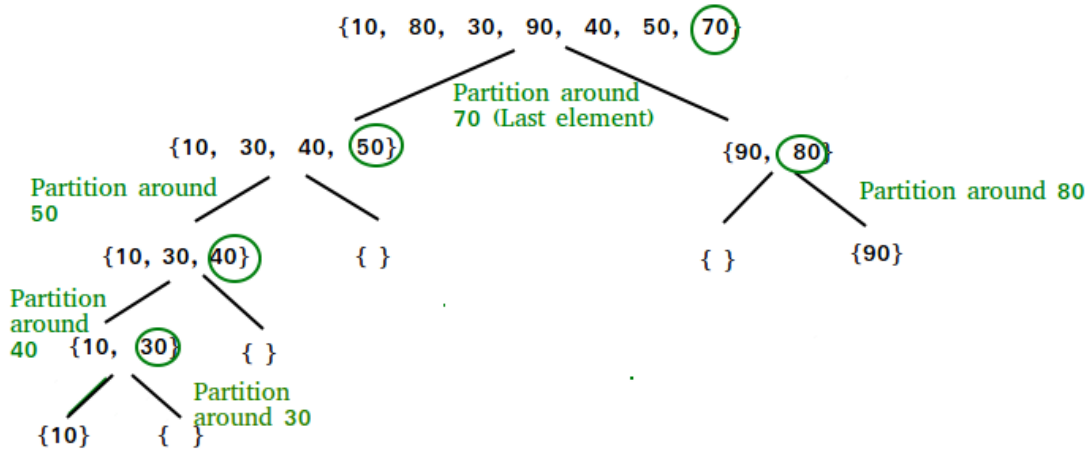
Always pick first element as pivot.

Always pick last element as pivot

Pick a random element as pivot.

Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



PseudoCode for recursive QuickSort function:

quickSort(A,L,h)

{

if (L¡h)

index=partition(A,L,h);

QS(A,L,index-1);

QS(A, index+1,h);

}

partition(A,L,h)
{
Pivot=A(1);
j=L-1;
For (i=1;i<h;i++){
{if (A(i)¡pivot)
j++;
A(j)=A(i);}}
j++;
A(j)=pivot;
Return j
}

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

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

The solution of above recurrence is $O(n2)$.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

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

The solution of above recurrence is $\theta(nLogn)$. It can be solved using case 2 of Master Theorem.

5.2.2 Binary Search

1. In Binary Search technique, we search an element in a sorted array by recursively dividing the interval in half.
2. Firstly, we take the whole array as an interval.
3. If the Pivot Element (the item to be searched) is less than the item in the middle of the interval, We discard the second half of the list and recursively repeat the process for the first half of the list by calculating the new middle and last element.
4. If the Pivot Element (the item to be searched) is greater than the item in the middle of the interval, we discard the first half of the list and work recursively on the second half by calculating the new beginning and middle element.
5. Repeatedly, check until the value is found or interval is empty.

PseudoCode for Binary Search function:

BinarySearch(A, L, h, key)
{
if $(L <= h)$
$m = L + h/2$;
if $(key == A(m))$
return m;
else if $(key > A(m))$
return BinarySearch$(A, m + 1, h, key)$;

else

return BinarySearch$(A, L, m-1, key)$;

else

return 0;

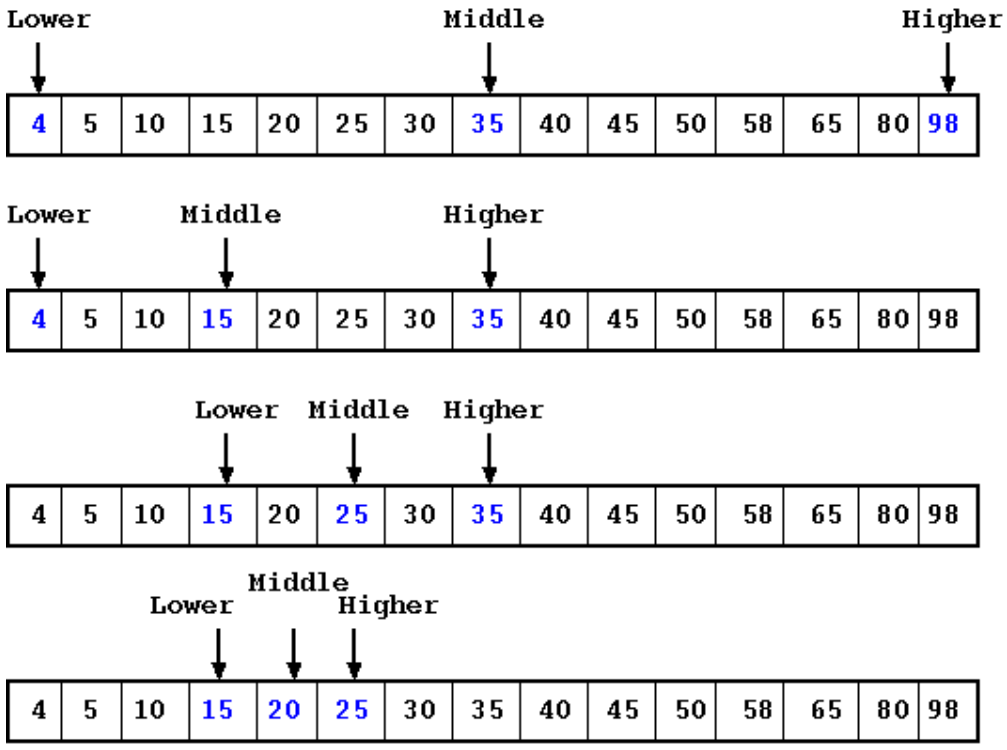
}

Case 1: key = A[mid], it the **best case** $T(n) = 1$ and $\Omega(1)$

Case 2: key $\neq$ A [mid], then we will split the array into two equal parts of size Binary Search. And again find the midpoint of the half-sorted array and compare with search element. Repeat the same process until a search element is found.

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Therefore, binary search complexity is $O(\log_2 n)$



5.3 Exercises

1. We can express insertion sort as a recursive procedure as follows. In order to sort $A[1 \ldots n]$, we recursively sort $A[1 \ldots n-1]$ and then insert $A[n]$ into the sorted array $A[1 \ldots n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

2. use the divide-and-conquer algorithm to find the maximum and minimum element in a given array.
Input: { 70, 250, 50, 80, 140, 12, 14 }
Output: The minimum number in a given array is : 12
The maximum number in a given array is : 250

3. Given two integers x and n, write a function to compute x^n. We may assume that x and n are small and overflow doesn't happen.

4. Using divide and conquer, design an algorithm that returns the inversion count for an array
Input: $A = \{8, 4, 2, 1\}$
Output: 6
Explanation: Given array has six inversions:
$(8, 4), (4, 2), (8, 2), (8, 1), (4, 1), (2, 1)$

Chapter 6

Dynamic Programming

Dynamic Programming bears similarities to Divide and Conquer. Both partition a problem into smaller subproblems and build solution of larger problems from solutions of smaller problems. In Divide and Conquer, work top-down. Know exact smaller problems that need to be solved to solve larger problem. - In Dynamic Programming, (usually) work bottom-up. Solve all smaller size problems and build larger problem solutions from them. Many large subproblems reuse solution to same smaller problem. it is often used for optimization problems which have many solutions; we want the best one.
The main idea of Dynamic Programming

1. Analyze the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution (usually bottom-up).

Following are the two main properties of a problem that suggests that the given problem can be solved using Dynamic programming:
1) Overlapping Subproblems

2) Optimal Substructure

6.1 Overlapping Subproblems

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take an example of following recursive program for Fibonacci Numbers, there are many subproblems that are solved again and again.

int fib(n)
{
if ($n <= 1$)
{return n; }
return fib($n-1$) + **fib**($n-2$);
}

We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:

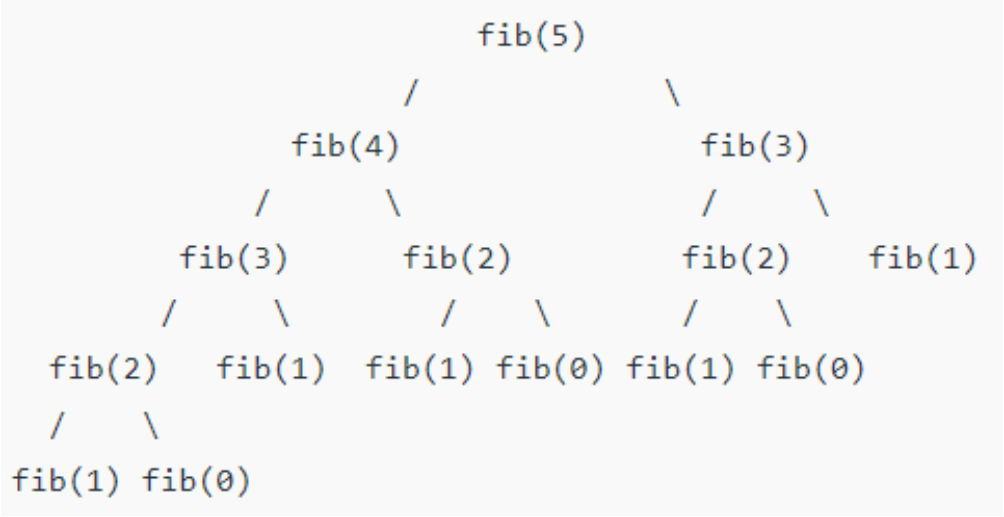


Figure 6.1: Recursion tree for execution of fib(5)

a) **Memoization** (Top Down)

b) **Tabulation** (Bottom Up)

Memoization (Top Down): The memoized program for a problem is similar to the recursive version with a small modification that looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

Tabulation (Bottom Up): The tabulated program for a given problem builds a table in bottom-up fashion and returns the last entry from the table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3), and so on. So literally, we are building the solutions of subproblems bottom-up.

Both Tabulated and Memoized store the solutions of subproblems.

In Memoized version, the table is filled on demand while in the Tabulated version, starting from the first entry, all entries are filled one by one. Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. For example, Memoized solution of the LCS problem doesn't necessarily fill all entries.

6.2 Optimal Substructure

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems. For example, the Shortest Path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithm like Floyd–Warshall and Single Source Shortest path algorithm for negative weight edges like Bellman–Ford are typical examples of Dynamic Programming.

6.3 Longest Common Subsequence

Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, "abc", "abg", "bdf", "aeg", '"acefg", .. etc are subsequences of "abcdefg".

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from $1, 2, ..n-1$. Recall from theory of permutation and combination that number of combinations with 1 element are ${}^nC_1$. Number of combinations with 2 elements are ${}^nC_2$ and so forth and so on. We know that ${}^nC_0 + {}^nC_1 + {}^nC_2 + \ldots {}^nC_n = 2^n$. So a string of length n has $2^n - 1$ different possible subsequences since we do not consider the subsequence with length 0. This implies that the time complexity of the brute force approach will be $O\left(n * 2^n\right)$. Note that it takes $O(n)$ time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

Example: Given two sequences X [1...m] and Y [1.....n]. Find the longest common subsequences to both.

X=ABCBDAB

Y=BDCABA

From the table we can deduct that LCS =4 for "BCBA".

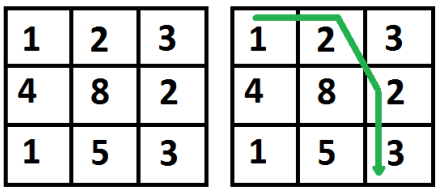
LCS-LENGTH(X, Y)

```
1   m = X.length
2   n = Y.length
3   let b[1..m, 1..n] and c[0..m, 0..n] be new tables
4   for i = 1 to m
5       c[i, 0] = 0
6   for j = 0 to n
7       c[0, j] = 0
8   for i = 1 to m
9       for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	X_i	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)
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

6.4 Exercises

1. Given a cost matrix cost and a position (m, n) in cost, write a function that returns cost of minimum cost path to reach (m, n) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. The total cost of a path to reach (m, n) is the sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells $(i + 1, j)$, $(i, j + 1)$, and $(i + 1, j + 1)$ can be traversed. You may assume that all costs are positive integers.
For example, in the following figure, what is the minimum cost path to (2, 2)?
The path with minimum cost is highlighted in the following figure. The path is $(0, 0)- > (0, 1)- > (1, 2)- > (2, 2)$. The cost of the path is $8(1 + 2 + 2 + 3)$.



2. Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter. For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2},

{2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

3. Given a sequence of matrices, using Dynamic Programming, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Chapter 7

Greedy Method

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

The following are the characteristics of a greedy method:

1. To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.

2. A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

The components that can be used in the greedy algorithm are:

Candidate set: A solution that is created from the set is known as a candidate set.

Selection function: This function is used to choose the candidate or subset which can be added in the solution.

Feasibility function: A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.

Objective function: A function is used to assign the value to the solution or the partial solution.

Solution function: This function is used to intimate whether the complete function has been reached or not.

7.1 Activity Selection Problem

The activity selection problem is a mathematical optimization problem. Our first illustration is the problem of scheduling a resource among several challenge activities. We find a greedy algorithm provides a well designed and simple method for selecting a maximum-size set of manually compatible activities.

Suppose $S = \{1, 2 \ldots . n\}$ is the set of n proposed activities. The activities share resources which can be used by only one activity at a time, e.g., Tennis Court, Lecture Hall, etc. Each Activity "i" has start time s_j and a finish time $\mathrm{f_i}$, where $s_\mathrm{i} \leq \mathrm{f_i}$. If selected activity " i " take place meanwhile the half-open time interval $(\mathrm{s_i}, \mathrm{f_j})$.

Activities i and j are compatible if the intervals $(s_i, \mathrm{f_i})$ and $(\mathrm{s_i}, \mathrm{f_i})$ do not overlap (i.e. i and j are compatible if $s_i \geq f_i$ or $s_i \geq f_i$). The activity-selection problem chosen the maximum-size set of mutually consistent activities.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n ← length[s]
2  A ← {a1}
3  i ← 1
4  for m ← 2 to n
5       do if sm ≥ fi
6            then A ← A ∪ {am}
7                 i ← m
8  return A
```

Example: Given 10 activities along with their start and end time as

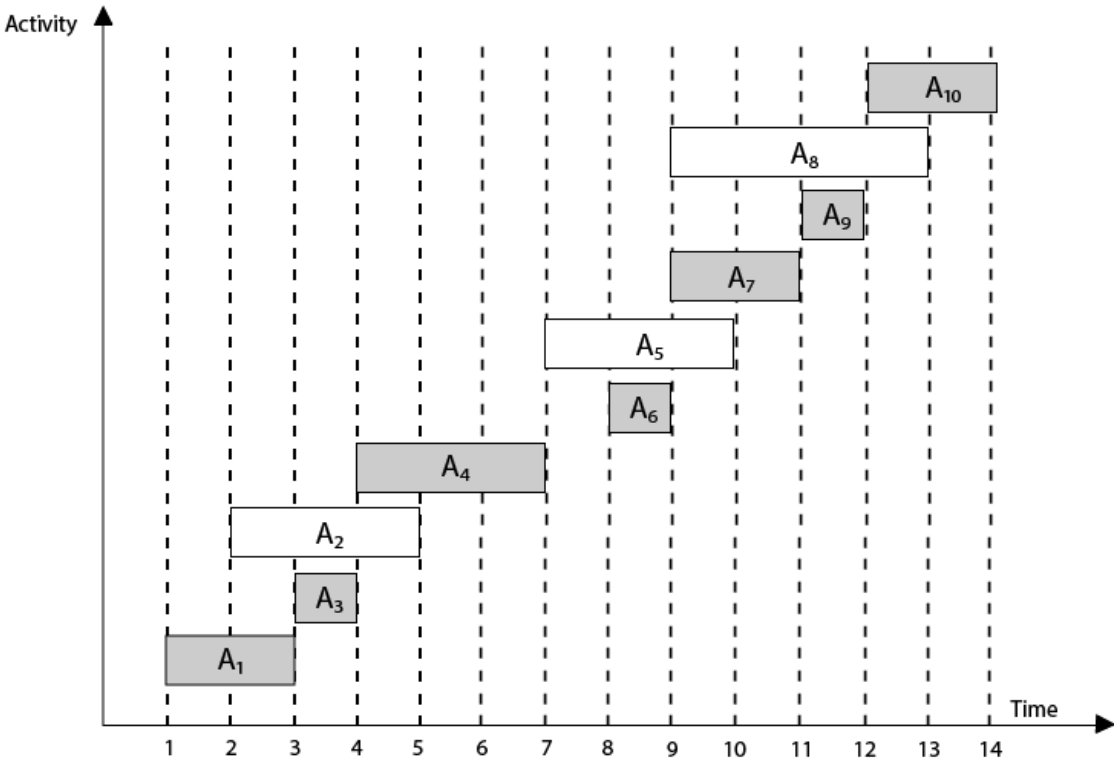
$$S = (A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10})\,.$$
$$S_i = (1, 2, 3, 4, 7, 8, 9, 9, 11, 12)$$
$$f_i = (3, 5, 4, 7, 10, 9, 11, 13, 12, 14)$$

Compute a schedule where the greatest number of activities takes place.

Solution: The solution to the above Activity scheduling problem using a greedy strategy is illustrated below: Arranging the activities in increasing order of end time Now, schedule $\mathrm{A_1}$

Next schedule A_3 as A_1 and A_3 are non-interfering.

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Next skip A_2 as it is interfering.

Next, schedule A_4 as A_1A_3 and A_4 are non-interfering, then next, schedule A_6 as $A_1A_3A_4$ and A_6 are non-interfering.

Skip A_5 as it is interfering.

Next, schedule A_7 as $A_1A_3A_4A_6$ and A_7 are non-interfering.

Next, schedule A_9 as $A_1A_3A_4A_6A_7$ and A_9 are non-interfering.

Skip A_8 as it is interfering.

Next, schedule A_{10} as A_1 A_3 A_4 A_6 A_7 A_9 and A_{10} are non-interfering.

Thus the final Activity schedule is: $\left(\mathbf{A}_1 \quad A_3 \quad A_4 \quad A_6 \quad A_7 \quad A_9 \quad A_{10} \right)$

7.2 Exercises

1. Compare between Divide and conqure, dynamic programming, and greedy method?

References

[1] Cherkassky BV, Goldberg AV, Radzik T (1996) Shortest paths algorithms: Theory and experimental evaluation. Mathematical programming 73(2):129–174

[2] Chernoff H (1952) A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. The Annals of Mathematical Statistics pp 493–507

[3] Chvatal V (1979) A greedy heuristic for the set-covering problem. Mathematics of operations research 4(3):233–235

[4] Cooley JW, Tukey JW (1965) An algorithm for the machine calculation of complex fourier series. Mathematics of computation 19(90):297–301

[5] Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms. MIT press

[6] Knuth DE (1973) Fundamental algorithms

[7] Sedgewick R, Wayne K (2011) Algorithms. Addison-wesley professional

[8] Shaffer CA (1997) A practical introduction to data structures and algorithm analysis. Prentice Hall Upper Saddle River, NJ

[9] Tabassum M, Mathew K, et al (2014) A genetic algorithm analysis towards optimization solutions. International Journal of Digital Information and Wireless Communications (IJDIWC) 4(1):124–142