A Comprehensive Algorithms learning

"Lets explore the world with algorithms"

WRITTEN BY

Anantha Lakshmi Yadavalli

Table of Contents

Preface

I am writing this book as part of our “Algorithm Analysis and Design” course Project at International Institute of Information Technology Hyderabad. My motive is to make a book of Algorithm techniques and discuss the efficiency.

Based on a new classification of algorithm design techniques and a clear outline of analysis methods, Introduction to the Analysis and Design of Algorithms presents the subject in an innovative manner. The book is written in a student-friendly manner, and emphasizes the understanding of ideas while thoroughly covering the material required in an introductory algorithms course. Some questions are added to motivate students' interest and strengthen their skills in algorithmic problem solving at the end of every chapter. The book teaches you the fundamentals of algorithms in a storyline that makes the material enjoyable and easy to digest(In general English) it is designed to be easy to read and understand, although the topic itself is complicated and also teaches students a range of design and analysis techniques. This method helps them to solve problems that arise while performing in computing applications. This algorithm book also explains the design process and the role of algorithms. The book includes topics like Basic of Algorithm Analysis, Divide and Conquer, Dynamic Programming, Greedy Algorithm, etc. I'm going to explain all the topics with examples.

Towards the end of the book The book intended to give an understanding of the algorithm design techniques and an appreciation of the role of algorithms in the broader field of computer science. The reader can increase their interest in algorithms by solving our exercises at the end of every chapter.

Introduction:

We live in a Technological World. The World which is influenced by algorithms day by day. Actually, Algorithms influence our everyday life in the modern world in many different areas of our daily life. Modern Society would be unimaginable without "Algorithms". Although many people are not aware of this, they use algorithms in their daily life. For example when we buy something on Flipkart or Watch something on Youtube/Netflix, we think it's our own choice. But, here comes the algorithm and it turns out that algorithm influences one-third of our decisions on Flipkart and more than Netflix/Youtube. Oh wait….we didn't discuss **what Algorithm is?** Yet. Let's come to the topic of algorithms, it's history and the fields it influences in our daily life.

What is Algorithm:

Anyone who has not studied computer science, mathematics, or a similar profession has probably hardly or not at all come into contact with algorithms in his life – at least not consciously. However, these are unconsciously present in many everyday situations. They make life easier and form the basis for many innovative technologies. However, in order to understand the meaning of algorithms, it is important to know their definition.

- Algorithms are the building blocks of computer programs. They are as important to programming as recipes are to cooking. An algorithm is a well-defined procedure that takes input and produces output . The study of algorithms is one of the key foundations of computer science.Day to Day problem solving and making strategies to handle different situations.

  Example:

  1. Going to picnic
  2. Preparing coffee
  3. Watching TV

Now we can say that problem is a kind of state or barrier to achieve something and problem solving is the process to get that state removed by performing some sequence of activities.

We can define algorithm as:

- Algorithm is a sequence of elementary operations that gives a solution to any problem from a particular specific class of problem in a finite number of steps.
- A Sequence of activities to be processed for getting desired output from given input.
- A formula (Method or Function) set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point.
- There may be more than one way to solve a problem, so there may be more than one algorithm for a problem.

As per the above definitions we can say that,

1. Getting desired or specified output is essential after the algorithm is executed.
2. One will get output only if the algorithm stops after finite time.
3. Activities in an algorithm to be clearly defined in other words for it to be unambiguous

History:

The concept of algorithm has existed since antiquity. Arithmetic algorithms, such as a division algorithm,were used by ancient Babylonian mathematicians. Greek mathematicians used the algorithm ‘sieve of Eratosthenes'. for finding prime numbers, and the ‘euclidean algorithm’ for finding the greatest common divisor of two numbers. Later in the 9th century Arab mathematicians used cryptographic algorithms for code-breaking, based on frequency analysis.Like this lot many evolutions were done. The word algorithm comes from the name of 9th century Persian Muslim mathematician Abu Abdullah Muhammad ibn Musa AI-Khwarizmi.The word algorism originally referred only to the rules of performing arithmetic using Hindu-Arabic numerals nut evolved via European Latiin translation of AI-Khawarizmi’s name into algorithm by 18th century. The use of the word evolved in all definite procedures for solving problems or performing tasks.

Think before writing Algorithms Why?

1. write an algorithm you must first understand the function that needs to be performed very clearly

2. If you are not very clear on how to perform the task then you will not be able to write the algorithm.
3. Once you are very clear on how to perform the task and state the steps clearly and precisely you also need to think about what would be the input specification and the output specification of the algorithm.
4. Notice that the ingredients of the algorithm are input specification and the output specification.
5. Without any of these ingredients the algorithm itself is not complete.
6. An algorithm is the heart of problem solving in Computer Science and also Real Life

Real life
algo

Real Life Algorithms

Ex1: Algorithm for making coffee

Algorithm begins with input specification. It has to be described in a precise and accurate manner.

Inputs for making cup of Coffee:

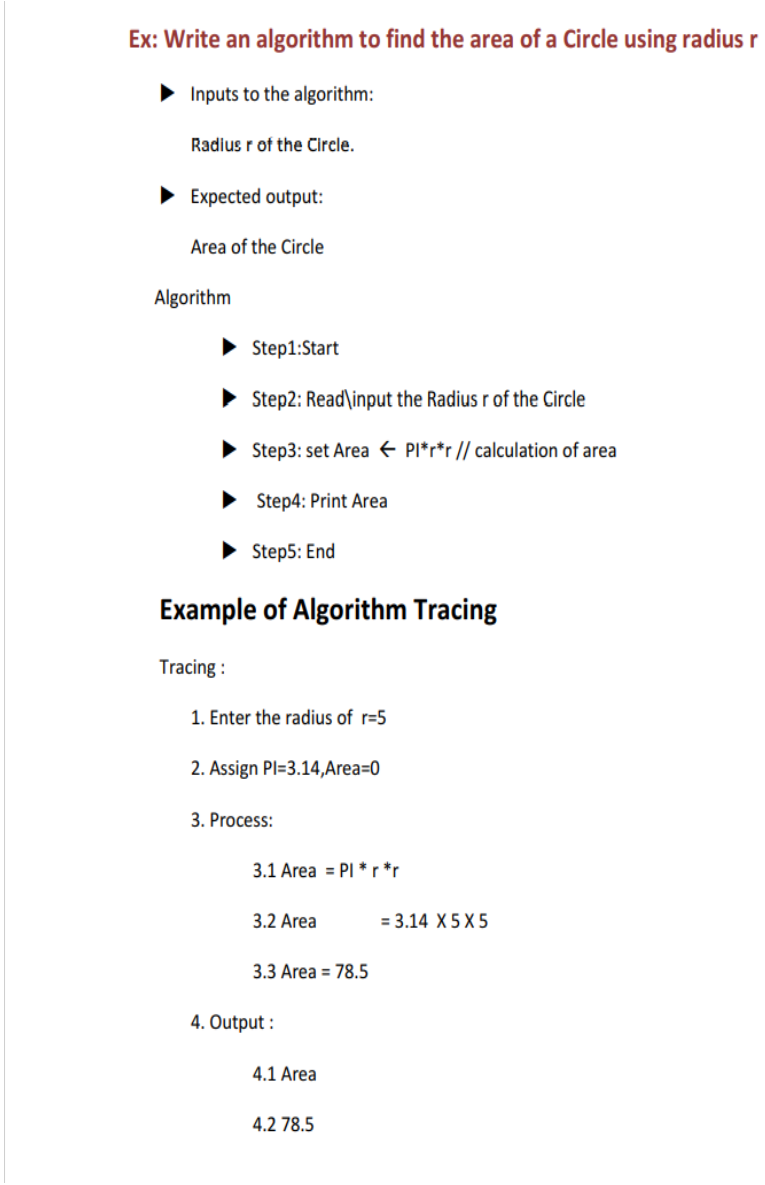
1 cup milk, 1 teaspoon sugar, 1 teaspoon of coffee powder, 1 teaspoon of water.

1. Add 1 teaspoon of coffee powder into the cup
2. Add 1 teaspoon sugar into the cup
3. Add 1 teaspoon of water into the cup
4. Stir the ingredients in the cup until well blended
5. Boil milk for 3 minutes of high flame
6. Add hot milk to the coffee syrup and stir until well blended
7. Finally coffee ready

Output specification:

At the end of the execution of these instructions that is one cup of hot coffee is ready to be served.

Example Algorithm to find area of a Circle:



Ex: Write an algorithm to find the area of a Circle using radius r

- Inputs to the algorithm:

  Radius r of the Circle.

- Expected output:

  Area of the Circle

Algorithm

- Step1:Start
- Step2: Read\input the Radius r of the Circle
- Step3: set Area ← PI*r*r // calculation of area
- Step4: Print Area
- Step5: End

Example of Algorithm Tracing

Tracing :

1. Enter the radius of r=5
2. Assign PI=3.14,Area=0
3. Process:
   - 3.1 Area = PI * r *r
   - 3.2 Area = 3.14 X 5 X 5
   - 3.3 Area = 78.5
4. Output :
   - 4.1 Area
   - 4.2 78.5

Here, I gave the pseudo code for some real life problems like preparing coffee and finding the area of the circle. We can represent algorithms as pseudo codes, flowcharts which are pictorial forms of algorithms with step-by-step process. Now, let's see how Algorithms influence our daily life in different fields.

Characteristics/Properties of an algorithm:

Donald Ervin Knuth has given a list of five properties for an algorithm, They are

1. Finiteness: an algorithm terminates after a finite numbers of steps
2. Definiteness: Each step in the algorithm is unambiguous. This means that the action specified by the step cannot be interpreted (it explain the meaning of step) in multiple ways & can be performed without any confusion
3. Input: An algorithm accepts or receives zero or more inputs
4. Output: An algorithm produces at least one output
5. Effectiveness It consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time

Algorithm Efficiency

1. Speed or Time: An algorithm that takes the least or minimum time to execute
2. Space or Memory: An algorithm that uses the least memory or space in allocation table
3. Code or program or set of Instructions: An algorithm is the very shortest description to generate the code.

4. Speed or Time is the most important factor to build an efficient algorithm

Advantages of an Algorithm:

- It is a step –by – step representation of a solution to a given problem , which is very easy to understand.
- It has produced a definite procedure.
- It is easy to first develop an algorithm , and then convert it into a flowchart and then into a computer program.
- It is independent of programming languages. It is easy to debug as every step has its own logical sequence.
- An algorithm is important in optimizing a computer program according to the available resources.
- Ultimately when anyone decides to solve a problem through better algorithms then searching for the best combination of program speed and least amount of memory consumption is desired .

There are many advantages to using algorithms in our daily life.

Influence of Algorithms in different fields:

Internet:

Algorithms play an important role in many places on the Internet. You decide which search results are listed in which place in Google. You select the posts that are displayed to a user on his social media channels. You determine the suggestions for online shopping which products the buyer might like. From e-commerce to social networks to banner advertising on a website, the algorithms can be found on virtually every website. They control that and how the user can move through the World Wide Web.

In the past decade, “algorithm” has become one of the central buzzwords in academic and public debates on digitalization and the Internet. Google search is essential nowadays for everyone. Whether a student, teacher, job holder, or old person almost everyone is using google to know things better. The order of search results returned by Google is based on a priority rank system called "PageRank". Google Search also provides many different options for customized search, using symbols to include, exclude, specify or require certain search behavior, and offers specialized interactive experiences, such as flight status and package tracking, weather forecasts, currency, unit, and time conversions, word definitions, and more.

Finances:

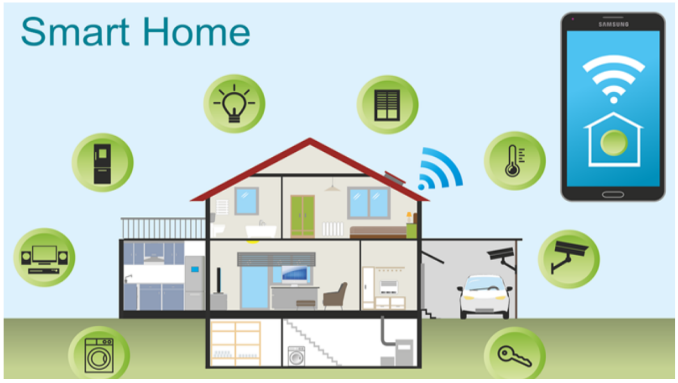
Even modern financial markets can no longer be imagined today without algorithms. More than half of all transactions are now controlled using such formulas. These are, for example, stock transactions, but also online transfers or the calculation of the creditworthiness of an applicant. Especially in the financial markets, where a lot of numbers are juggled anyway, IT specialists will find a large selection of jobs related to programming, monitoring, or optimizing algorithms.

Firstly, while the institutional market has enjoyed a large variety of “algos” serving the equity markets to date, other areas such as futures are still witnessing huge product demand and innovation as a result.

The fixed income market is a prime example and negotiations between industry groups are ongoing as to how practical a fully automated fixed income could really be, given the magnitude of the required shift from telephone to electronic trading.

Thirdly, the retail trading market continues to expand worldwide, which is opening up algorithmic trading to a whole new group and further increasing demand for technological advancement.

Building Technology:



Building technology is currently experiencing a real revolution. It is developing modern technologies with their algorithms that play an increasingly important role in this. For this purpose, networked sensors are installed in all rooms, which collect data such as temperature or humidity. They can evaluate all of this data and combine them with one another. You also observe the heating behavior and can thus control the heat supply more individually and sustainably. The weather forecasts already mentioned can also be included or the current electricity prices. The possibilities that are already being used, or at least conceivable, are diverse.

Algorithms can now work out the best ways to lay out rooms, construct the buildings and even change them over time to meet users' needs. In this way, algorithms are giving architects a whole new toolbox with which to realise and improve their ideas. Algorithms makes systems smart

Algorithms can also be used to extend the capability of designers to think about and generate shapes and arrangements that might not otherwise be possible. Instead of personally drawing floor plans according to their intuition and taste, architects using algorithmic design input the rules and parameters and allow the computer to produce the shape of the building. These algorithms are often inspired by ideas from nature, such as evolution.

Medicine:

Medicine is another and last area of application on this list when it comes to possible industries for computer scientists. Modern technologies are being used more and more frequently in order to optimize medical care for patients or to improve prevention. An example of this is an intelligent microphone that monitors the breathing of asthma patients. It serves as a kind of early warning system so that you can act quickly (more) in the event of an attack. But there are many more innovations in the starting blocks, which have to be programmed or even yet to be invented. It will therefore remain exciting to see which other areas of work for IT specialists will be added in the future – not only in medicine.

It's no secret, the human body is complex and trying to diagnose our illnesses can be quite a task, even for the most seasoned doctors. To better their diagnoses, doctors collect as much information as possible from their patients. This information comes in an array of sources, including doctor's notes, clinical images, genetic tests, demographics, and laboratory results.

Medical algorithms are part of a broader field which usually falls into the medical informatics and medical decision-making category. Medical decisions occur in nearly all areas of medicine including diagnosis, medical test selection, therapy and prognosis to name a few.

Transport:

We are moving towards a society in which it is neither sustainable nor logical for individuals to go to work or move around the city in their own personal vehicle. We need to adopt efficient alternative mechanisms, such as **car sharing, ride sharing and other new trends** that are gradually becoming more popular. These include the **use of less polluting vehicles** that take up less space, such as bicycles or electric scooters.For this we created an Apps to book the cab for sharing online. there are many forms of transport: planes, ships, trains, land vehicles… All this logistics needs to be organized, particularly now that it does not seem that the pace of globalization is going to slow down, despite the obligatory hiatus imposed by Coronavirus.

Safety is a priority criterion of the transport systems operation referred to as a total of means and activities connected with carrying of people and cargo. The main goal of systems of this type is safe carrying people in set quantities and over an assigned area, by means of transport used in the system. Road transport is the most frequently used branch of transport. It is characterized by high speed and flexibility of transport tasks accomplishment resulting from a big number of roads which enable carrying passengers or cargo directly to the place of destination. Road transport can be distinguished from other transport branches by the following features: unlimited access to road transport means, high serviceability, high transport speed, providing services according to the schedule and punctually. The system environment includes all components which belong to it in terms of the carried out analysis. Among the most important conclusions is the need to achieve two goals. First, **develop algorithms for drones, self-driving vehicles**, etc. And second, optimize and **increase the efficiency of last-mile distribution**.

Game:

Another example is gaming. This can take place online as well as offline but is always based on numerous different algorithms. A classic area of application is gambling. Because the “coincidence” that exists in a real casino can only be simulated online. It is all the more important to rely on computer algorithms for such cases, which reproduce the chance as best as possible and thus ensure a fair game without the need to check the results. But other computer games also use numerous algorithms, for example when it comes to so-called “behavior trees”. In theory, it would be possible to give the characters, etc. their own artificial intelligence so that they can learn and decide for themselves. But the algorithm ensures that the developers of such games still maintain control. Video games employ algorithms to respond intelligently to players’ inputs. From the predictable patterns of the alien ships in “Space Invaders”, to the much more responsive ghosts in “Pac Man”, the earliest video games used mathematical processes to mimic the behaviour of thinking beings.

The creation of a video game is no easy task for anyone to undertake. When it comes to programming, a lot of knowledge needs to be obtained to create an efficient and solid product. The game industry is a very competitive business, where games are defined heavily by the products that came before them. This combination is the key to creating real-time simulations that push the envelope on what is considered cutting-edge.

By reading the above pages we know how important Algorithms play in our daily life.

Algorithms for computer science. Most of the parts for theoretical exams and even programming contests are designed from this subject only. If any programming questions there, those come from this topic. Some students fail to understand it properly and they couldn't get the strategy or approach to solve a problem and they feel that they are lagging in logic development or strategy development in a program. Right now, we are covering everything right from the basics. So that you people can answer any type of questions. I will cover the topics such that you just look at the question and solve the problem that I guarantee. I cover those questions which are used for job interviews or even for any other exam or for any entrance exam or GATE exam. Most of the lessons in my book are in the way which you feel comfortable.

Algorithm analysis:

Let again come to discussion: what is an algorithm? Why is it written ? when it is written ? Who will write those algorithms? And let us get the answers for this.
See, algorithm as the common definition everybody knows is "A step-by-step procedure" to solve a computational problem. Then what is a program? We have a question here. Programming is also a step by step procedure to solve a problem. Then what is the difference between Algorithm and a Program.? So by comparing programmes and algorithms I can make you understand the importance of algorithms and the meaning of an algorithm. Lets see

If you know the software development life cycle, there are phases of development of software projects. There are two different phases.
1.design phase and
2.implementation phase
If you are not aware of this. Let me give another example: whatever you can manufacture or construct something by engineering procedures. First, you design and make your design perfect and thorough to construct what you are going to develop and then you start the development. Without developing anything you can't construct anything on a trial and error basis like you constructed something again. It's wrong to destroy and again create a new one.

See software engineers , they code some program and delete that and again they start programming. So that's why we can’t get the feeling that we have wasted up on something we write(a useless program) and think so. Now, the point is first you design and then you write a program. At the design time what do you use? So if you are not familiar with the program then what do you write? So that simple program we can write it in simple english statements that are easy to understand. We are not writing those on the computer, we write those on paper. So that we are not writing in any language we can write it in any MSword or notepad like application. So that's what an algorithm is.

So algorithms are written in design time.Programs are written as implementation time. So first you make a design of what your procedures are going to do, what your softwares going to do and come up with some design and that design you convert into a program. Then what do you call a person who is designing. The person who will design should have the domain knowledge. The knowledge about the problem and solution. Let's take an example: if we are writing a hospital-related programme, we don't know what exactly works in a hospital. Doctors know more than us. In this situation they are capable of writing algorithms. Any language can be used to write algorithms as long as it is understandable for those people who are using it.like if a designer has written an algorithm then a programmer has written the program to it. So designers should understand and also programmers should understand. So the theme of the project who are working on that should be able to understand. We can programme in programming languages. When we write an algorithm we don't need any hardware and operating system. While writing the program we care about the operating system and hardware.

We will study the algorithm to find out if we are achieving results perfectly or not and if our algorithm is efficient or not in terms of time and space. We will see what an algorithm analysis is. What do we do with the program? We just check like we run it and test.
If the problem is having more than one solution or algorithm then the best one is decided by the analysis based on two factors.

1. CPU Time (Time Complexity)
2. Main memory space (Space Complexity).

Let's talk about “Priori Analysis and Posteriori testing”.

A Posteriori analysis	A priori analysis
Posteriori analysis is a relative analysis.	Piori analysis is an absolute analysis.
It is dependent on language of compiler and type of hardware.	It is independent of language of compiler and types of hardware.
It will give exact answer.	It will give approximate answer.
It doesn't use asymptotic notations to represent the time complexity of an algorithm.	It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution.
The time complexity of an algorithm using a posteriori analysis differ from system to system.	The time complexity of an algorithm using a priori analysis is same for every system.
If the time taken by the program is less, then the credit will go to compiler and hardware.	If the algorithm running faster, credit goes to the programmer.

I am giving this table for more information. You need to conduct an a priori power analysis (a priori meaning it is conducted before you do your research) to calculate the minimum number of participants needed to test your study hypotheses / detect a significant effect (if one exists). In a posteriori analysis, we collect actual statistics about the algorithm's consumption of time and space, while it is executing.

Comparison of Two Functions:

Comparison of two functions may be time functions or space functions. if suppose i have function that is n^2 and another is n^3. we don't know which becomes an upper bound and lower bound so for comparison we easily say it by seeing that n^3 is greater than n^2. but if we have to check and we have to prove then we do test by keeping some value for 'n'. The value of n=2 then, n^2=4 and n^3 =8. The value itself says n^3 is greater.

The second method is "applying log on both sides"

$\log n^2$ $\log n^3$

2 log n < 3 log n

here , log n is the same. n^3 is 3 $times$ log n and n^2 is 2 $times$ log n. then obviously we can say that n^3 is greater. This is helpful for complex functions. We have different functions we solve using the logarithmic formulas. Let see a complex function example below. Example:

f(n)=n^2log n g(n)=n$(log\ n)^{10}$

Applying log

log[$n^2 log\ n$] log[n$(log\ n)^{10}$]

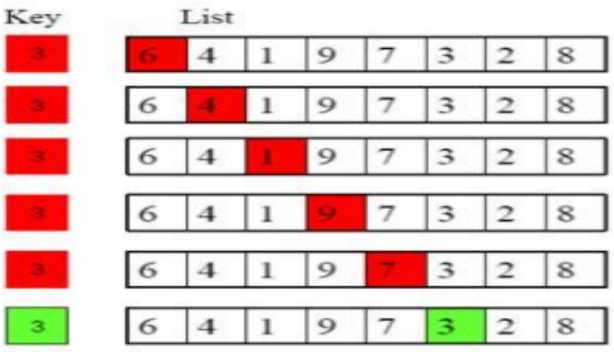
log n^2 + $log\ log\ n$ log n+log$(log\ n)^{10}$

2 log n + log log n log n + 10 log log n

By the underline words we can say f(n) is greater.

Best Worst and Average Case Analysis:

Let's talk about the best, worst and average cases. Let's take an example of linear search.In linear search we search for one key. If that key is found at index zero itself then it's the best case. If that key is found at the last index then that's the worst case. If that key is found at the middle or not at the extremes then that's called the average case. Now we will see the searching of the key by the figure below.



Here our key=3, 3 is found at index=5.

Best case: in case we are searching for a key element which is present at the first index is the best case. In the figure above if key=6, then it is at index=0. Then Best case time for 'n' elements is O(1). B(n)=O(1).

Worst case: If i am searching for a key at the last index. In our example if key=8, then we have to move all the elements in order to reach the element 8. Then Worst case time for 'n' elements is O(n). W(n)=O(n).

Average case: All possible case times/ number of cases.

$$A(n)= (n + 1)/2$$

Asymptotic Notations:

The next topic is Asymptotic Notation. It is the most important topic in Algorithms.This topic is coming from mathematics.Functions belongs to math so this topic belongs to mathematics. As for time complexity we are using functions in algorithms that's the reason notations are also used. The notations are used for representing the symbol form of a function or showing the class of a function. To represent a function in simple and communicable form so that one category will get the idea that this is the most needed topic for algorithms because we can't write a lengthy function every time. We need a simple method for representing the time complexity.let see what are notations used

Symbol	*notation name*	*acts as*
O	big-oh	upper bound
Ω	big- omega	lower bound
Θ	theta	Average bound

Now let me tell something when we find the time complexity of any functions the complexity will be one upon 1<log<$\sqrt{n} < n < n\,logn <$ $n^2 < n^3 <... < 2^n < 3^n <.... < n^n$ $these$. if it is not among these it may be a multiple of these. If it's not even multiple of these then you may not able to exactly represent **Θ** then we can go for other forms to represent those are "big-oh" and "big-omega" notation.

Big-oh notation:

As per the definition , The function f(n)=**O**(g(n)) iff ∃ positive constants c and n_0.such that f(n) < = c*g(n) ∀ n > = n_0. Let's take an example: f(n)= 2n+3.

2n+3 <= …… we can write anything in those dotted lines which is less than or equal to 2n+3. But one thing we have to care about here is we have two terms. I should not write multiple terms, I should only write a single term. That single term can have some coefficient c. if we write "10n". Is it greater? Yes, it is greater than 2n+3.if n=1, then 2(1)+3=5 and 10(1)=10. This case 2n+3<=10n n>=1.here f(n)=2n+3, c=10 and n is g(n).

∴ f (n) = **O**(n).

We can also write 2n+3 < = $5n^2$. then ∴ f(n)=**O**(n^2). the the functions less than n are lower bounds 1<log n< $\sqrt{n}$. 'n' is average bound. The functions greater than n are upper bounds, those functions are n log n < n^2 < n^3 <... < 2^n < 3^n <.... < n^n So we will write the near notation that is f(n)=**O**(n).

Omega notation:

As per the definition, the function f(n)=**Ω**(g(n)) iff ∃ positive constants c and n_0. such that f(n) > = c*g(n) ∀ n > = n_0. Let's take an example: f(n)=2n+3. Now, 2n+3 > = 1*n ∀ n > = 1. Here f(n)=2n+3, c=1 and g(n)=n.

∴ f (n) = **Ω**(n).

We can also write as ∴ f (n) = **Ω**(log n) it also holds because it is lower bound. But we should always take the nearest value.

Theta Notation:

As per the definition , the function f(n)=$\boldsymbol{\theta}$(g(n)) iff ∃ positive constants $c_1, c_2 \; and \; n_0$. such that c_1*g(n)< = f(n) < = c_2*g(n).

Let's take an example: f(n)=2n+3.

Then, 1*n < = 2n+3 < = 5*n.

$$\therefore \; f(n) = \boldsymbol{\theta}(n).$$

Here, $\therefore$ f (n) = $\boldsymbol{\theta}$(n) also satisfy.we can't use 'log n' on both the sides, its not valid on left part(i.e 5*n). Theta notation is the average notation so here I can use only 'n'. We can't use any other value and this is mostly recommended whenever you have a function and you are able to represent in '$\boldsymbol{\theta}$' rather than **Ω** and **O** notations. One more important thing is don't confuse this with best case or worst case. It's not related to best case or worst case.mostly people think that **O** is used for upper bound so it is for worst case and **Ω** is used for lower bound so it is used for best case.No. we can use any notation for best case and worst case.

Question: You write f(n),c_1, c_2 and g(n) by the equation.

Properties of Asymptotic Notations:

General properties:

1. If f(n) is O(g(n)) then a*f(n) is **O**(g(n)) ex:f(n)=$2n^2 + 5$ is **O**(n^2). Then, 7.f(n)=7($2n^2 + 5$)=14n^2+35 is **O**(n^2). This is same for **Ω** case.
2. If f(n) is (g(n)) then a*f(n) is **Ω**(g(n)) ex:f(n)=$2n^2 + 5$ is **Ω**(n^2). Then, 7.f(n)=7($2n^2 + 5$)=14n^2+35 is **Ω**(n^2).

Reflexive property:

1. If f(n) is given then f(n) is O(f(n)). Ex: f(n)=n^2 is O(n^2). similarly , f(n) = Ω(f(n)) , f(n) = O(f(n)).

Transitive property:

If **f(n) = O(g(n)) and g(n) = O(h(n))** ⇒ **f(n) = O(h(n))**

Example:

If $f(n) = n$, $g(n) = n^2$ and $h(n) = n^3$

$\Rightarrow$ n is $O(n^2)$ and n^2 is $O(n^3)$ then n is $O(n^3)$.

Proof:

$f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

By the definition of Big-Oh(O), there exists positive constants c, no such that $f(n) \leq c.g(n)$ for all $n \geq no$

$\Rightarrow f(n) \leq c1.g(n)$

$\Rightarrow g(n) \leq c2.h(n)$

$\Rightarrow f(n) \leq c1.c2h(n)$

$\Rightarrow f(n) \leq c.h(n)$, where, $c = c1.c2$ By the definition, $f(n) = O(h(n))$

Similarly,

$f(n) = \boldsymbol{\Theta}(g(n))$ and $g(n) = \boldsymbol{\Theta}(h(n)) \Rightarrow f(n) = \boldsymbol{\Theta}(h(n))$

$f(n) = \boldsymbol{\Omega}(g(n))$ and $g(n) = \boldsymbol{\Omega}(h(n)) \Rightarrow f(n) = \boldsymbol{\Omega}(h(n))$

$f(n) = \mathbf{O}(g(n))$ and $g(n) = \mathbf{O}(h(n)) \Rightarrow f(n) = \mathbf{O}(h(n))$

$f(n) = \boldsymbol{\omega}(g(n))$ and $g(n) = \boldsymbol{\omega}(h(n)) \Rightarrow f(n) = \boldsymbol{\omega}(h(n))$

Symmetry property:

If $f(n) = \boldsymbol{\Theta}(g(n))$ if and only if $g(n) = \boldsymbol{\Theta}(f(n))$

Example: If $f(n) = n^2$ and $g(n) = n^2$ then

$f(n) = \boldsymbol{\Theta}(n^2)$ and $g(n) = \boldsymbol{\Theta}(n^2)$

Transpose Symmetry:

f(n) = O(g(n)) if and only if g(n) = Ω(f(n))

Example: If $f(n) = n$ and $g(n) = n^2$ then

n is $\mathbf{O}(n^2)$ and n^2 is $\boldsymbol{\Omega}(n)$

Similarly, $f(n) = \mathbf{O}(g(n))$ if and only if $g(n) = \omega(f(n))$

Here we observe, ***max(f(n), g(n)) = O(f(n) + g(n))***

Proof:

Without loss of generality, assume $f(n) \leq g(n)$, $\Rightarrow \max(f(n), g(n)) = g(n)$

Consider, $g(n) \leq \max(f(n), g(n)) \leq g(n)$

$\Rightarrow g(n) \leq \max(f(n), g(n)) \leq f(n) + g(n)$

$\Rightarrow g(n)/2 + g(n)/2 \leq \max(f(n), g(n)) \leq f(n) + g(n)$

From what we assumed, we can write

$\Rightarrow f(n)/2 + g(n)/2 \leq \max(f(n), g(n)) \leq f(n) + g(n)$

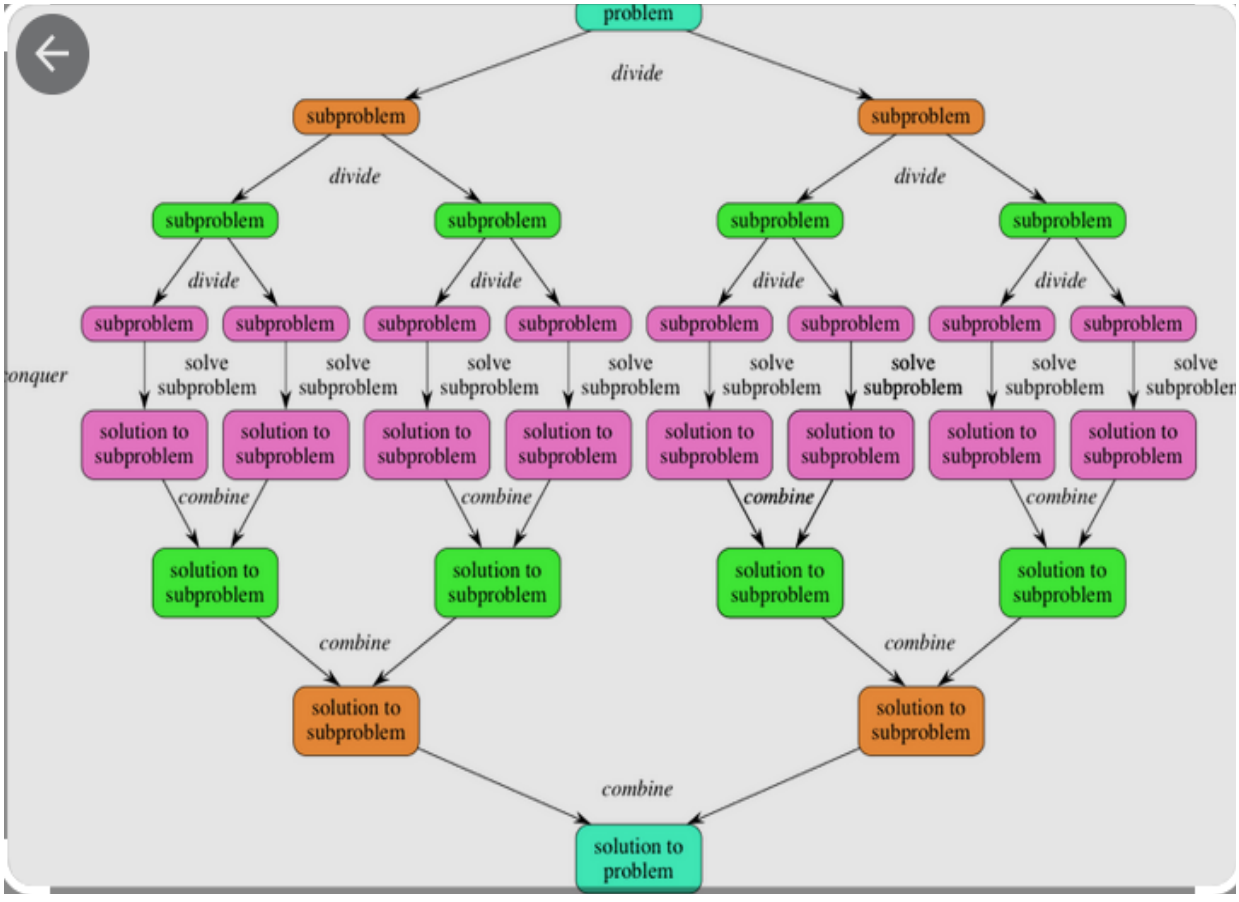
$\Rightarrow (f(n) + g(n))/2 \leq \max(f(n), g(n)) \leq f(n) + g(n)$

By the definition of $\boldsymbol{\Theta}$, $\max(f(n), g(n)) = \boldsymbol{\Theta}(f(n) + g(n))$.

DIVIDE AND CONQUER:

The first topics we discussed were Analysis of algorithms and how to analyze an algorithm and what is the purpose of analysis.

Let's come to the topic “Divide and Conquer”. This topic is related to a strategy for solving a problem like this. We have other strategies also in subjects like 3d method dynamic programming backtracking branch and bound.
If a problem of size is even if say there is a problem named ‘Prob’ with size=n. Then we split that prob into small parts and find the solution for those small parts and combine the whole solutions of small problems in order to solve the “Prob”.



The figure in the last page itself says the approach of the “Divide and Conquer” algorithm.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.

Fundamental of Divide and Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. 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 the Stopping Condition.

There are more applications there for this divide and conquer approach. Binary search, Quicksort, Merge Sort, Closest pair of points, strassen’s algorithm, cooley-tukey fast transform algorithm and karatsuba algorithm for fast multiplication are some of the applications of divide and conquer.

Min - Max problem:

Problem: Analyze the algorithm to find the maximum and minimum element from an array.

```
Algorithm: Max ?Min Element (a [])
Max:  a [i]
Min:   a [i]
For i= 2 to n do
If a[i]> max then
max = a[i]
if a[i] < min then
min: a[i]
return (max, min)
```

Analysis:

Method 1: if we apply the general approach to the array of size n, the number of comparisons required are 2n-2.

Method-2: In another approach, we will divide the problem into subproblems and find the max and min of each group, now max. Each group will compare with the only max of another group and min with min.

Let n = is the size of items in an array

Let T (n) = time required to apply the algorithm on an array of size n. Here we divide the terms as T(n/2).

2 here tends to be the comparison of the minimum with minimum and maximum with maximum as in the above example.

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

T (n) = 2 T $\left(\frac{n}{2}\right)+2$ → Eq (i)

T (2) = 1, time required to compare two elements/items. (Time is measured in units of the number of comparisons)

$T\left(\frac{n}{2}\right)=2T\left(\frac{n}{2^2}\right)+2$ → Eq (ii)

Put eq (ii) in eq (i)

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right)+2\right]+2$$
$$=2^2T\left(\frac{n}{2^2}\right)+2^2+2$$

Similarly, apply the same procedure recursively on each subproblem or anatomy

{Use recursion means, we will use some stopping condition to stop the algorithm}

$$T(n) = 2^i T\left(\frac{n}{2^i}\right)+2^i+2^{i-1}+\ldots\ldots\ldots\ldots\ldots\ldots+2 \quad \text{..... (Eq. 3)}$$

Recursion will stop, when $\frac{n}{2^i}=2 \Rightarrow n=2^{i+1}$ → (Eq. 4)

Put the equ.4 into equation3.

$$T(n) = 2^i T(2) + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2$$

$$= 2^i .1 + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2$$

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

$$= 2^{i+1} + 2^i - 2$$

$$= n + \frac{n}{2} - 2$$

$$= \frac{3n}{2} - 2$$

Number of comparisons requires applying the divide and conquering algorithm on n elements/items = (3n/2)-2.

Number of comparisons requires applying general approach on n elements = (n-1) + (n-1) = 2n-2

From this example, we can analyze how to reduce the number of comparisons by using this technique.

Analysis: suppose we have an array of size 8 elements.

Method1: requires (2n-2), (2x8)-2=14 comparisons

Method2: requires $\frac{3\times 8}{2} - 2 = 10 \ \ comparsions$

It is evident; we can reduce the number of comparisons (complexity) by using a proper technique

Binary Search:

1. In the Binary Search strategy, we recursively divide the interval in half to find an element in a sorted array.

2. To begin, we use the entire array as an interval.

3. If the Pivot Element (the item to be searched) is less than the item in the centre of the interval, we reject the second half of the list and recursively calculate the new middle and final element for the first half of the list.

4. If the Pivot Element (the item to be searched) is greater than the item in the interval's middle, the first half of the list is discarded, and the second half is worked on recursively by computing the new beginning and middle element.

5. Repeatedly, check until the value is found or interval is empty.

Analysis:

1. **Input:** an array A of size n, already sorted in the ascending or descending order.
2. **Output:** analyze to search an element item in the sorted array of size n.
3. **Logic:** Let T (n) = number of comparisons of an item with n elements in a sorted array.

- Set BEG = 1 and END = n
- Find mid = $\text{int}\left(\frac{beg + end}{2}\right)$
- Compare the search item with the mid item.

Case 1: item = A[mid], then LOC = mid, but it the best case and T (n) = 1

Case 2: item ≠A [mid], then we will split the array into two equal parts of size $\frac{n}{2}$.

And again find the midpoint of the half-sorted array and compare it with the search element.

Repeat the same process until a search element is found.

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

{Time to compare the search element with mid element, then with half of the selected half part of array}

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + 1$, putting $\frac{n}{2}$ in place of n.

Then we get: T (n) = $(T\left(\frac{n}{2^2}\right) + 1) + 1$.....By putting T $\frac{n}{2}$ in (1) equation

T (n) = $T\left(\frac{n}{2^2}\right) + 2$..... (Equation 2)

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1$..... Putting $\frac{n}{2}$ in place of n in eq 1.

T (n) = $T\left(\frac{n}{2^3}\right) + 1 + 2$

T (n) = $T\left(\frac{n}{2^3}\right) + 3$..... (Equation 3)

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$..... Putting $\frac{n}{3}$ in place of n in eq1

Put $T\left(\frac{n}{2^3}\right)$ in eq (3)

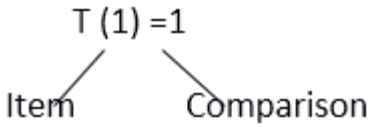
T (n) = $T\left(\frac{n}{2^4}\right) + 4$

Repeat the same process ith times

T (n) = $T\left(\frac{n}{2^i}\right) + i$ .....

Stopping Condition: T (1) =1

At least there will be only one term left that's why that term will compare out, and only one comparison be done that's why



Is the last term of the equation and it will be equal to 1

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

$\frac{n}{2^i}$ Is the last term of the equation and it will be equal to 1

$n = 2^i$

Applying log both sides

$\log n = \log_2 i$

Log n = i log 2

$\frac{\log n}{\log 2} = 1$

$\log_2 n = i$

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

$\frac{n}{2^i}$ = 1 as in eq 5

= T (1) + i

= 1 + i..... T (1) =1 by stopping condition

= 1+ $\log_2 n$

= $\log_2 n$ ..... (1 is a constant that's why ignore it)

Therefore, binary search is of order o ($\log_2 n$)

Merge Sorting:

Merge sort is another Divide and Conquer technique-based sorting algorithm.

It is one of the most effective sorting approaches for constructing a recursive algorithm.

Divide and Conquer Strategy

In this technique, we segment a problem into two halves and solve them individually. After finding the solution of each half, we merge them back to represent the solution of the main problem.

Suppose we have an array A, such that our main concern will be to sort the subsection, which starts at index p and ends at index r, represented by A[p..r].

Divide

If q is assumed **q** to be the central point somewhere in between **p** and **r**, then we will fragment the subarray **A[p..r]** into two arrays **A[p..q]** and **A[q+1, r]**.

Conquer

After splitting the arrays into two halves, the next step is to conquer. In this step, we individually sort both of the subarrays **A[p..q]** and **A[q+1, r]**. In case if we did not reach the base situation, then we again follow the same procedure, i.e., we further segment these subarrays followed by sorting them separately.

Combine

As when the base step is acquired by the conquer step, we successfully get our sorted subarrays **A[p..q]** and **A[q+1, r]**, after which we merge them back to form a new sorted array **[p..r]**.

Merge Sort algorithm

The MergeSort function keeps on splitting an array into two halves until a condition is met where we try to perform MergeSort on a subarray of size 1, i.e., **p == r**.

And then, it combines the individually sorted subarrays into larger arrays until the whole array is merged.

ALGORITHM-MERGE SORT

1. If p<r

2. Then q → (p+ r)/2

3. MERGE-SORT (A, p, q)

4. MERGE-SORT (A, q+1,r)

5. MERGE (A, p, q, r)

Here we called **MergeSort(A, 0, length(A)-1)** to sort the complete array.

As you can see in the image given below, the merge sort algorithm recursively divides the array into halves until the base condition is met, where we are left with only 1 element in the array. And then, the merge function picks up the sorted subarrays and merges them back to sort the entire array.

The following figure illustrates the dividing (splitting) procedure.

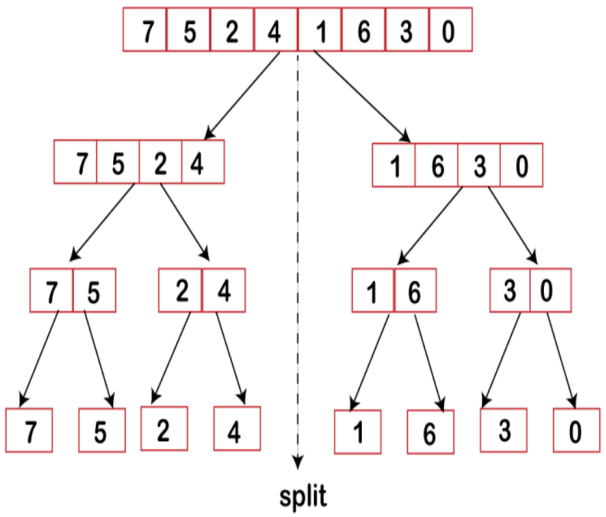


Figure 1: Merge Sort Divide Phase

```
FUNCTIONS: MERGE (A, p, q, r)

1. n 1 = q-p+1
2. n 2= r-q
3. create arrays [1.....n 1 + 1] and R [ 1.....n 2 +1 ]
4. for i ← 1 to n 1
5. do [i] ← A [ p+ i-1]
6. for j ← 1 to n2
7. do R[j] ← A[ q + j]
8. L [n 1+ 1] ← ∞
9. R[n 2+ 1] ← ∞
10. I ← 1
11. J ← 1
12. For k ← p to r
13. Do if L [i] ≤ R[j]
14. then A[k] ← L[ i]
15. i ← i +1
16. else A[k] ← R[j]
17. j ← j+1
```

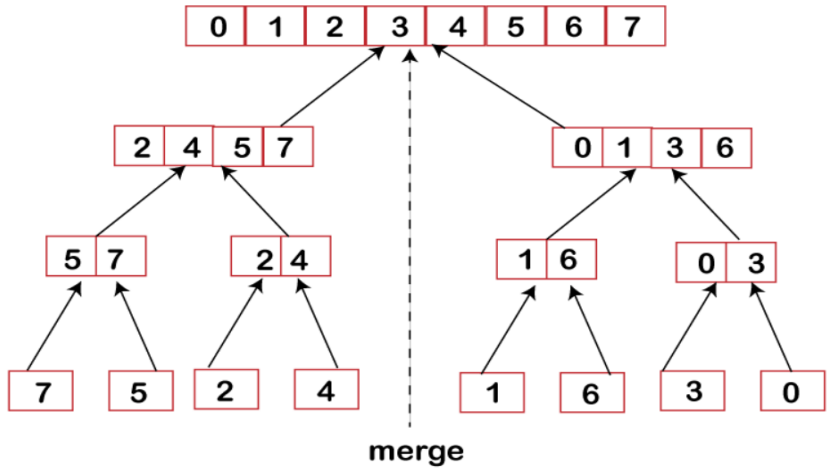


Figure 2: Merge Sort Combine Phase

The merge step of Merge Sort

Mainly the recursive algorithm depends on a base case as well as its ability to merge back the results derived from the base cases. Merge sort is no different algorithm, just the fact here the **merge** step possesses more importance.

To any given problem, the merge step is one such solution that combines the two individually sorted lists(arrays) to build one large sorted list(array).

The merge sort algorithm upholds three pointers, i.e., one for both of the two arrays and the other one to preserve the final sorted array's current index.

Did you reach the end of the array?

No:

Firstly, start with comparing the current elements of both the arrays.

Next, copy the smaller element into the sorted array.

Lastly, move the pointer of the element containing a smaller element.

Yes:

Simply copy the rest of the elements of the non-empty array

Merge( ) Function Explained Step-By-Step

Consider the following example of an unsorted array, which we are going to sort with the help of the Merge Sort algorithm.

A= (36,25,40,2,7,80,15)

Step1: The merge sort algorithm iteratively divides an array into equal halves until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half.

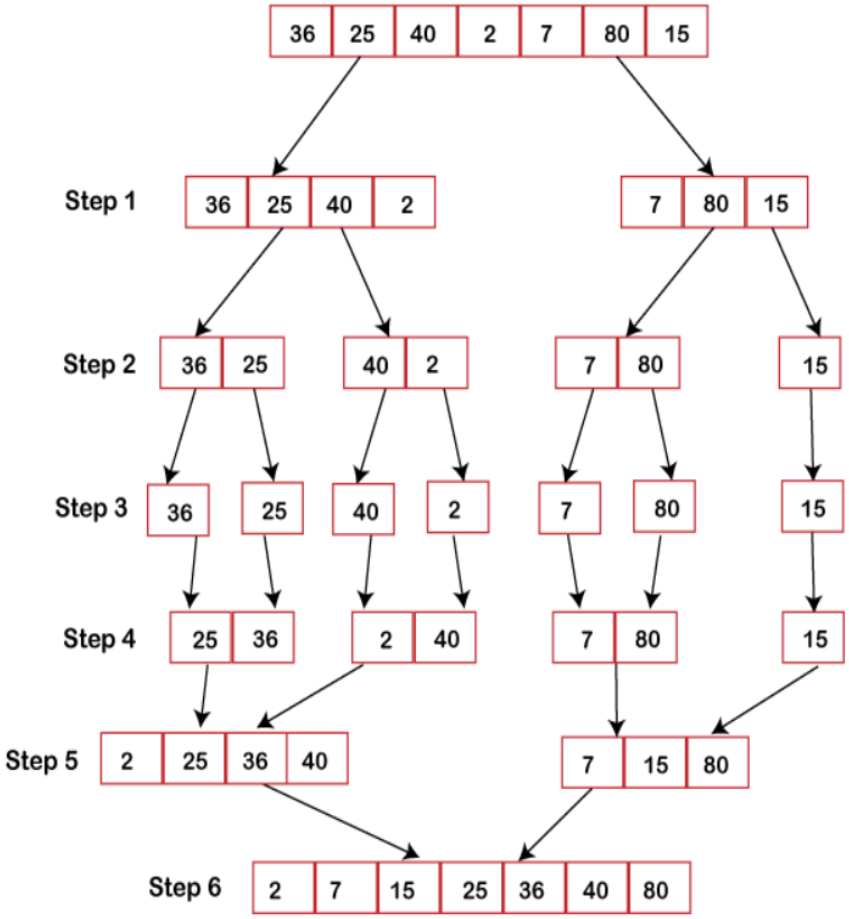
Step2: After dividing an array into two subarrays, we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves.

Step3: Again, we will divide these arrays until we achieve an atomic value, i.e., a value that cannot be further divided.

Step4: Next, we will merge them back in the same way as they were broken down.

Step5: For each list, we will first compare the elements and then combine them to form a new sorted list.

Step6: In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.



Hence the array is sorted.

Analysis of Merge Sort:

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

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

But we ignore '-1' because the element will take some time to be copied in merge lists.

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

Note: Stopping Condition T (1) =0 because at last, there will be only 1 element left that needs to be copied, and there will be no comparison.

Putting $n=\frac{n}{2}$ in place of n in .....equation 1

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$$.....equation2

Put 2 equation in 1 equation

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

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

$$T(n) = 2^2 T\left(\frac{n}{2^2}\right) + 2n \text{.....equation 3}$$

Putting $n = \frac{n}{2^2}$ in equation 1

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} \text{.....equation4}$$

Putting 4 equation in 3 equation

$$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$$

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

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 3n \text{.....equation5}$$

From eq 1, eq3, eq 5.....we get

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in \text{.....equation6}$$

From Stopping Condition:

$$\frac{n}{2^i} = 1 \text{ And } T\left(\frac{n}{2^i}\right) = 0$$

$$n = 2^i$$

Apply log both sides:

$\log n = \log_2 i$

$\log n = i \log 2$

$$\frac{\log n}{\log 2} = i$$

$\log_2 n = i$

From 6 equation

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in$$

$$= 2^i \times 0 + \log_2 n \cdot n$$

$$= T(n) = n.\log n$$

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

Space Complexity: The space complexity of merge sort is **O(n)**.

Merge Sort Applications

The concept of merge sort is applicable in the following areas:

- Inversion count problem
- External sorting
- E-commerce applications

Exercise:

1.**Your Task:**

You don't need to take the input or print anything. Your task is to complete the function **merge()** which takes arr[], l, m, r as its input parameters and modifies arr[] in-place such that it is sorted from position l to position r, and function **mergeSort()** which uses merge() to sort the array in ascending order using merge sort algorithm.

Expected Time Complexity: O(nlogn)

Expected Auxiliary Space: O(n)

Constraints:

$1 <= N <= 10^5$

$1 <= arr[i] <= 10^3$

Example input/output

Input:

N = 10

arr[] = {10 9 8 7 6 5 4 3 2 1}

Output:

1 2 3 4 5 6 7 8 9 10

2. Show that log(n!) = O(n log n). (Hint: To show an upper bound, compare n! with n n . To show a lower bound, compare it with (n/2)n/2 .)

3.

Problem

You are given an array A and B of size N.

You must select a subset of indices from 1 to N such that for any pair of indices $(x, y),\ x \neq y$ in the subset one of the following conditions holds true:

- $A[x] < A[y]$ and $B[x] < B[y]$
- $A[x] > A[y]$ and $B[x] > B[y]$
- $A[x] = A[y]$ and $B[x] \neq B[y]$

Your task is to determine the largest possible size of a subset that satisfies the provided conditions.

Note: Assume 1 based indexing.

Input format

- The first line contains an integer T that denotes the number of test cases.
- For each test case:
  - The first line contains an integer N.
  - The second line contains N space-separated integers that denotes the array A.
  - The third line contains N space-separated integers that denotes the array B.

Output format

For each test case, print the largest possible size of a subset that satisfies the provided conditions in a new line.

Constraints

$1 \leq T \leq 10$
$2 \leq N \leq 10^5$

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solving them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
- It is more proficient than that of its counterpart Brute Force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

Greedy Strategy:

Just like a Divide and Conquer , Greedy is also one of the approaches for solving a problem. Let us understand what this method says and what this method is about?

This method is used to solve ‘optimization’ problems. What are optimization problems? A problem which demands or which requires either minimum result or maximum result. I will explain it with a example :

Ex: i want to travel from location A to location B.i have to cover this journey and this is our problem now. How do you solve this problem? Any idea you have?

We can cover the distance between location A and location B by walking, car, bike ,flight, train,etc.. We can solve this in many ways. But there is a constraint there that is I have to cover this journey within 2hr. This is the condition. We can't go there by walking within 2hrs. We can cover it only if we can go by flight or train. By this we can understand that a problem has many solutions. But, the solution which satisfies our condition is the most feasible solution to a problem.A solution which satisfies the condition given in the problem is a feasible solution.

A minimization problem, so as the problem demands our result should be minimum. Then it is a minimization problem. In our solution also the train takes less cost than flight.this is called as optimal solution. A solution which is feasible and minimum cost that is best results.for every problem there can be only one optimal solution. There can’t be multiple optimum solutions. That means there can be only one minimum cost .There can be more than one solution, there can be more than one feasible solution but there can only be one optimum solution.

Here this problem requires a minimum result. Some other problems require maximum results. If a problem requires either minimum or maximum results then that type of problem is called an “Optimization Problem” or we can say an optimization problem is one which requires either minimum or maximum results. By this we get that optimal solution means achieving the objective of a problem, satisfying the objective of a

problem i.e either minimum result or maximum result. So a greedy method is used for solving optimization problems.

We use some some strategies to solve the optimization problems

1. Greedy method
2. Dynamic programming
3. Branch and Bound

For these strategies the approach is different. Some problems are solved using greedy and some using divide and conquer.now we are going to cover this greedy method and what greedy method says and what its approach is.

Knapsack Problem:

Knapsack problem is a well known problem. First of all we'll know about the problem, then we will see how it can be solved and then it will also understand how the grading method is applicable here.

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "knapsack problem" dates back to the early works of the mathematicianTobias Dantzig (1884–1956), and refers to the commonplace problem of packing the most valuable or useful items without overloading the lug

Let understand the problem below

Knapsack problem:

Objects	0	1	2	3	4	5	6	7
Profits	p	10	5	15	7	6	18	3
Weights	w	2	3	5	7	1	4	1

n=7 and m=15

Here objects are there, every object has profit and weights. Y given a bag is called knapsack the bag capacity is 15 these are seven objects bag capacity 15. There is a path

whose capacity is 15 let us say kg. It can be a cage or pawn also so i am assuming it as kg. Now what we have to do is we have to fill this bag with these objects and we will carry this back to a different place and we will cell away these objects. We get that profit so that profit is the gain that you get by transferring these objects to some location so for transferring you have to carry them in this container or beg so i can also call this knapsack as container. So the problem is the container loading problem. So we find this type of problems in daily life lot of goods are transported daily from one place to another place now the problem is about filling of that container with the objects when it becomes a problem if you have the objects whose total weight is more than the capacity of the container back if you have less all can fit in that no problem at all if you have the objects more in number the way it is greater than the capacity for that then it is a problem so already we have some objects here so i have to fill this bag with these objects such that the profit is maximized . profit is maximized so this is an optimization problem and it is a maximization problem. Can you guess the constraint here? Did you think of constraints here?

The constraint here is the total objects included here in the bag the total weight should be less than or equal to 15. I can give various solutions and can put only one. But here we want the maximum result. The solution which gives maximum results is called optimum solution.

This knapsack problem is for those objects which are divisible I have something of 5kg means i need not take all 5kg i can just take 2kg also 1kg also these objects can be divided so what type of objects if i take an example these may be vegetables 7kg of potatoes 5kg tomatoes something. Instead of taking 5kg i can take 2kg also it can be divided so this not about those objects which are indivisible let take example like a 7kg washing machine can’t be divided it can’t take half of that washing machine so i have to take the full one so here i can take the fraction of an object also so that's how whatever i am including that ca be a fraction starting 0 to 1. You guys can solve and think about the problem. Let's look into another problem:

0-1 knapsack problem

i	v	w
1	5	4
2	4	3
3	3	2
4	2	1

Capacity=6

i \ w	0	1	2	3	4	5	6
0							
1							
2							
3							
4							

A demonstration of the dynamic programming approach.

A similar dynamic programming solution for the 0-1 knapsack problem also runs in pseudo-polynomial time. Assume w 1 , w 2 , … , w n , W w_{1},\,w_{2},\,\ldots ,\,w_{n},\,W} are strictly positive integers. Define m [i , w] m[i,w]} to be the maximum value that can be attained with weight less than or equal to w using items up to i (first i items).

We can define m [i , w] m[i,w]} recursively as follows: **(Definition A)**

- m [0 , w] = 0
- m [i , w] = m [i − 1 , w]
- m[i,\,w]=m[i-1,\,w]} if w i > w (the new item is more than the current weight limit)
- m [i , w] = max (m [i − 1 , w] , m [i − 1 , w − w i] + v i) m[i,\,w]=\max(m[i-1,\,w],\,m[i-1,w-w_{i}]+v_{i})} if w i ⩽ w

The solution can then be found by calculating m [n , W] m[n,W]} . To do this efficiently, we can use a table to store previous computations.

The following is pseudocode for the dynamic program:

// Input:

// Values (stored in array v)

// Weights (stored in array w)

// Number of distinct items (n)

// Knapsack capacity (W)

// NOTE: The array "v" and array "w" are assumed to store all relevant values starting at index 1.

```
array m[0..n, 0..W];

for j from 0 to W do:
    m[0, j] := 0

for i from 1 to n do:
    m[i, 0] := 0


for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

This solution will therefore run in O (n W), O(nW)} time and O (n W) O(nW)} space. (If we only need the value m[n,W], we can modify the code so that the amount of memory required is O(W) which stores the recent two lines of the array "m".)

Greedy Algorithm solves problems by selecting the best option that appears to be the best at the time.

A greedy method can be used to solve a variety of optimization problems.

Although there is no efficient solution to some problems, a greedy algorithm may yield a solution that is close to optimal.

If a problem has the following two properties, a greedy algorithm will work:

1. Greedy Choice Property: Creating a locally optimum solution leads to a globally optimal solution.

To put it another way, an optimal solution can be found by making "greedy" decisions.

2. Optimal substructure: The best solutions have the best subsolutions.

In other words, subproblems with an optimal solution have optimal answers.

Steps for achieving a Greedy Algorithm are:

1. **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
2. **Local Optimal Choice:** In this, the choice should be the optimum which is selected from the currently available
3. **Unalterable:** Once the decision is made, at any subsequent step that option is not altered.

Huffman Codes:

(i) Data can be encoded efficiently using Huffman Codes.

(ii) It is a widely used and beneficial technique for compressing data.

(iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

Suppose we have 10^5 characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8 x 10^5 bits in a file. But we want to compress the file and save it compactly. Suppose only six characters appear in the file:

	A	B	C	D	E	F	Total
Frequency	45	13	12	16	9	5	100

How can we represent the data in a Compact way?

(i) Fixed length Code: Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

For example:

a 000

b 001

c 010

d 011

e 100

f 101

For a file with 10^5 characters, we need 3 x 10^5 bits.

(ii) A variable-length code: It can do considerably better than a fixed-length code, by giving many characters short code words and infrequent character long codewords.

For example:

a 0

b 101

c 100

d 111

e 1101

f 1100

Number of bits = (45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4) x 1000

= 2.24 x 10^5bits

Thus, 224,000 bits to represent the file, a saving of approximately 25%.This is an optimal character code for this file.

Codes for prefixes:

The prefixes of one character's encoding must not be equal to the whole encoding of another character; for example, 1100 and 11001 are not valid codes because 1100 is a prefix of another code word.

Prefix codes are useful because they make encoding and decoding more clear.

For any binary character code, encoding is straightforward: we simply concatenate the code words that describe each character in the file.

With a prefix code, decoding is also relatively simple.

The codeword that begins with encoded data is unambiguous because no codeword is a prefix of another.

Greedy Algorithm for constructing a Huffman Code:

Huffman invented a greedy algorithm that creates an optimal prefix code called a Huffman Code.

In a bottom-up approach, the algorithm creates a tree T that is comparable to the optimal code.

To produce the final tree, it starts with a set of |C| leaves (C being the number of characters) and conducts |C| - 1'merging' operations.

In the Huffman algorithm, 'n' represents the number of characters in a set, z represents the parent node, and x and y represent the left and right children of z, respectively.

Algorithm of Huffman Code:

```
Huffman (C)
1. n=|C|
2. Q ← C
3. for i=1 to n-1
4. do
5. z= allocate-Node ()
6. x=  left[z]=Extract-Min(Q)
7. y= right[z] =Extract-Min(Q)
8. f [z]=f[x]+f[y]
9. Insert (Q, z)
10. return Extract-Min (Q)
```

Example: Find an optimal Huffman Code for the following set of frequencies:

1. a: 50 b: 25 c: 15 d: 40 e: 75

Solution:

Given that: C = {a, b, c, d, e}

f (C) = {50, 25, 15, 40, 75}

n = 5

Q ← c

i.e.

c	15	b	25	d	40	a	50	e	75

for i ← 1 to 4

i = 1 Z ← Allocate node

x ← Extract-Min (Q)

y ← Extract-Min (Q)

c	15	b	25	d	40	a	50	e	75

Left [z] ← x

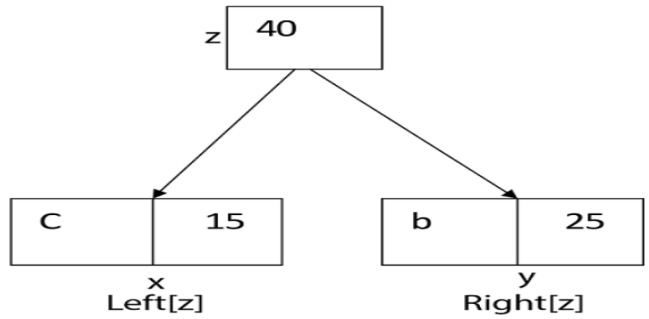
Right [z] ← y

f (z) ← f (x) + f (y) = 15 + 25

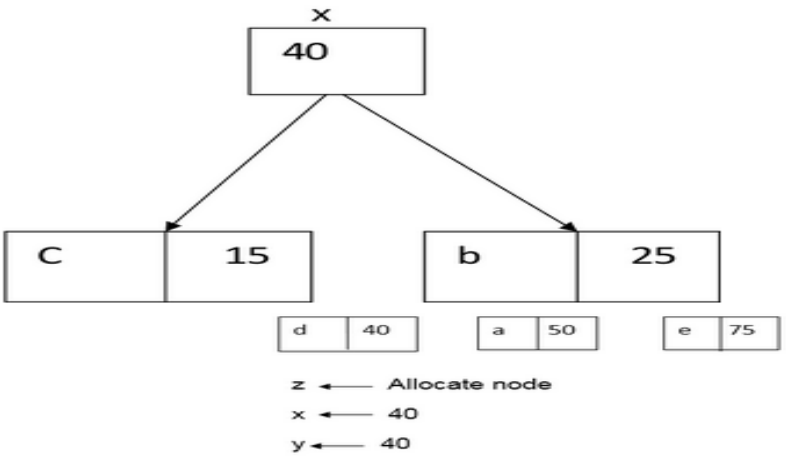
f (z) = 40

d	40	a	50	e	75

i.e.

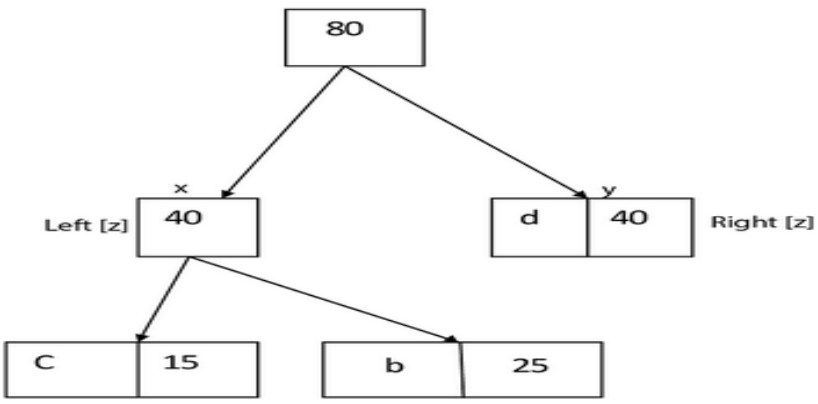


Again for i=2

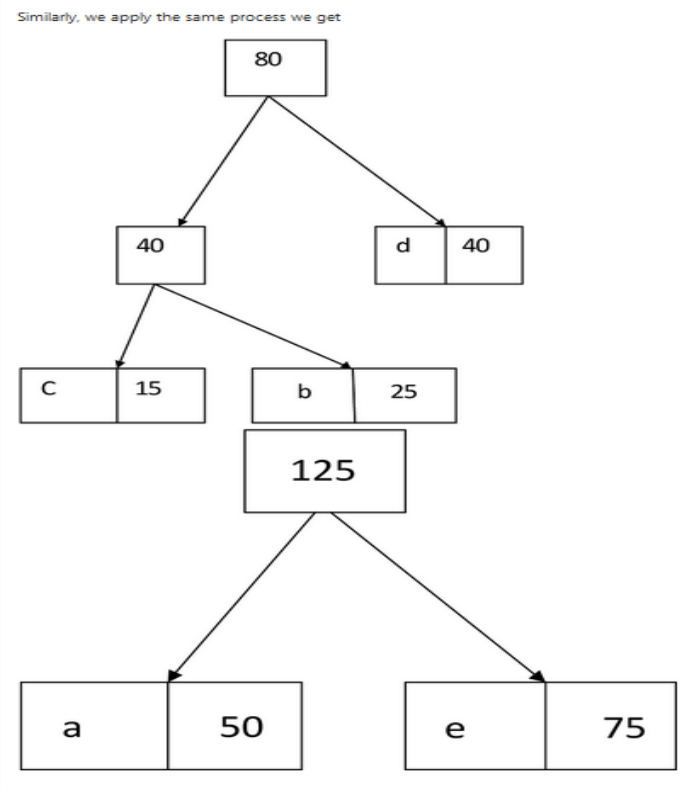


z ← Allocate node
x ← 40
y ← 40

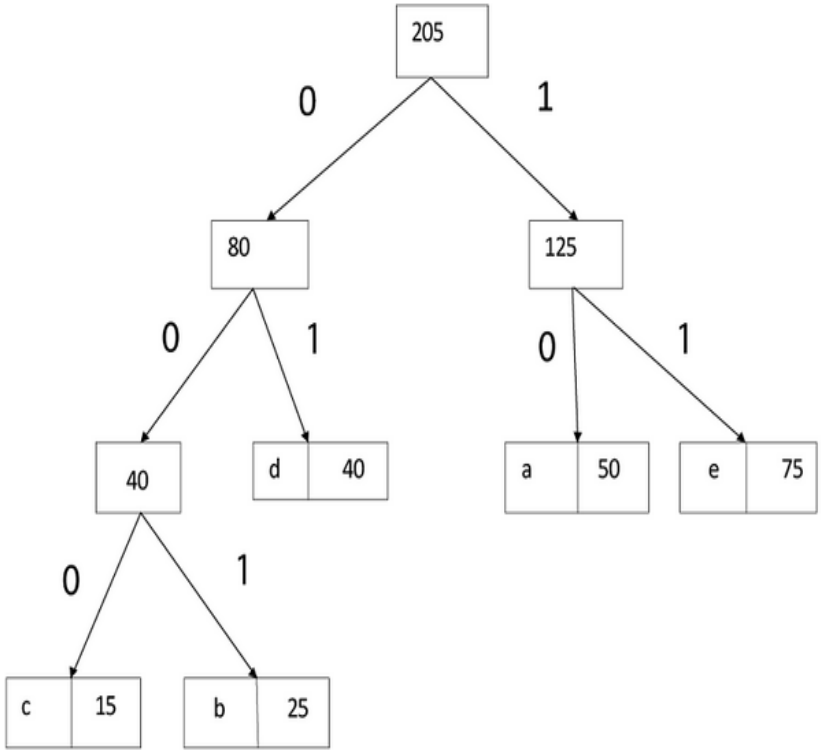
left [z] ← x
right [z] ← y
f (z) = 40 + 40 = 80



Similarly, we apply the same process we get



Thus, the final output is:



Exercise for greedy algorithm:

Problem

There are N (where N is always even) players standing in a line where the coordinates of players are given as $$(X_1,0), (X_2,0)... (X_N,0)$$. $You are required to divide them into two teams such that there is an equal number o$ (P, 0) $if and only if the number of players on the left - hand side is equal to the number of$ (P,0)$$, you are independent to choose their side.
Find the number of such possible coordinates where teams can be divided.

Note: Non-integral coordinates do not exist.

Input format

- The first line contains an integer T denoting the number of test cases.
- For each test case:
  - The first line contains an integer N denoting the number of players.
  - The second line contains an array of space-separated integers denoting array X.

Output format

Print a single line for each test case, denoting the number of coordinates $(P, 0)$ from where the teams can be divided.

Constraints

$1 \leq T \leq 20000$

$1 \leq N \leq 200000$

$0 \leq |X_i| \leq 10^9$

Here, N is always even.

Problem

You are given an binary array A of N integers. You are also given an integer K and you need to ensure that no subarray of size greater than or equal to K has average 1.
For this you can perform the below operation:

- Choose any index and flip the bit.

Find the minimum number of operations to acheive the above condition.

Input:

First line contains number of test cases T. For each test case:

- First line contains two space seperated integers N and K.
- Second line contains N space separated integers of the binary array.

Output:

Print T lines , one for each test case. For each test case:

- Print a single line denoting the minimum number of operations that needs to be performed.

Constraints:

- $1 \leq T \leq 20000$
- $1 \leq N \leq 200000$
- $1 \leq K \leq N$
- $0 \leq A_i \leq 1$
- Sum of N over all test cases will not exceed 200000.

Sample Input	Sample Output
2 5 5	1 1

Dynamic Programming

Dynamic programming is a problem-solving technique that divides problems into subproblems and saves the result for later use, eliminating the need to recalculate the result.The optimal substructure property describes how subproblems are improved to improve the overall solution.Dynamic programming is mostly used to tackle optimization challenges.When we talk about optimization challenges, we're talking about trying to find the simplest or most complex solution to a problem.If an ideal solution to a problem exists, dynamic programming ensures that it will be found.

Dynamic programming is a strategy for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each subproblem only once, and then storing the solutions to prevent repeating calculations, according to the definition.

Let's understand this approach through an example.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

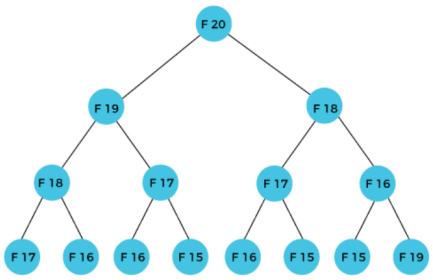
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,…

The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

F(n) = F(n-1) + F(n-2),

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.

How can we calculate F(20)?



F(20) is determined as the sum of F(19) and F(20), as shown in the diagram above.

We try to break the problem into similar subproblems using the dynamic programming approach.

This strategy is used in the aforementioned situation, where F(20) is divided into two subproblems, F(19) and (18).

If we recall the definition of dynamic programming, it states that the same subproblem should not be computed twice.

Even yet, the subproblem is calculated twice in the example above.

F(18) is calculated twice in the case above, and F(17) is calculated twice as well.

However, this strategy is highly valuable because it solves comparable subproblems; however, we must be cautious while saving the results because if we are not particular about storing the result that we have computed only once, it may result in resource waste.

If we take the preceding case,

How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that the same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

The above five steps are the basic steps for dynamic programming. The dynamic programming is applicable that are having properties such as:

Those problems that are having overlapping subproblems and optimal substructures. Here, optimal substructure means that the solution of optimization problems can be obtained by simply combining the optimal solution of all the subproblems.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

Matrix Chain Multiplication:

It is a Method under Dynamic Programming in which previous output is taken as input for next.

Here, Chain means one matrix's column is equal to the second matrix's row [always].

In general:

If $A = \lfloor a_{ij} \rfloor$ is a p x q matrix

$B = \lfloor b_{ij} \rfloor$ is a q x r matrix

$C = \lfloor c_{ij} \rfloor$ is a p x r matrix

Then

$$AB = C \text{ if } c_{ij} = \sum_{k=1}^{q} a_{ik}\ b_{kj}$$

Given following matrices $\{A_1, A_2, A_3, \ldots A_n\}$ and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

$A_1 \times A_2 \times A_3 \times \ldots\ldots \times A_n$

Matrix Multiplication operation is **associative** in nature rather than commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need.

In general, for $1 \leq i \leq p$ and $1 \leq j \leq r$

$$C[i, j] = \sum_{k=1}^{q} A[i, k]B[k, j]$$

It can be observed that the total entries in matrix 'C' is 'pr' as the matrix is of dimension p x r Also each entry takes O (q) times to compute, thus the total time to compute all possible entries for the matrix 'C' which is a multiplication of 'A' and 'B' is proportional to the product of the dimension p q r.

It is also noticed that we can save the number of operations by reordering the parenthesis.

Example1: Let us have 3 matrices, A_1,A_2,A_3 of order (10 x 100), (100 x 5) and (5 x 50) respectively.

Three Matrices can be multiplied in two ways:

1. **A_1,(A_2,A_3):** First multiplying(A_2 and A_3) then multiplying and resulting withA_1.
2. **(A_1,A_2),A_3:** First multiplying(A_1 and A_2) then multiplying and resulting withA_3.

No of Scalar multiplication in Case 1 will be:

(100 x 5 x 50) + (10 x 100 x 50) = 25000 + 50000 = 75000

No of Scalar multiplication in Case 2 will be:

(100 x 10 x 5) + (10 x 5 x 50) = 5000 + 2500 = 7500

To find the best possible way to calculate the product, we could simply parentheses the expression in every possible fashion and count each time how many scalar multiplications are required.

Matrix Chain Multiplication Problem can be stated as "find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplication is minimized".

Development of Dynamic Programming Algorithm

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

Dynamic Programming Approach

Let $A_{i,j}$ be the result of multiplying matrices i through j. It can be seen that the dimension of $A_{i,j}$ is the pi-1 x p_j matrix.

Dynamic Programming solution involves breaking up the problems into subproblems whose solution can be combined to solve the global problem.

At the greatest level of parenthesization, we multiply two matrices

$$(A_{1.....n}=A_{1....k} \text{ x } A_{k+1....n})$$

Thus we are left with two questions:

- How to split the sequence of matrices?
- How to parenthesize the subsequence $A_{1.....k}$ and$A_{k+1.....n}$?

One possible answer to the first question for finding the best value of 'k' is to check all possible choices of 'k' and consider the best among them. But it can be observed that checking all possibilities will lead to an exponential number of total possibilities. It can also be noticed that there exists only O (n^2) different sequences of matrices, in this way do not reach the exponential growth.

Step1: Structure of an optimal parenthesization: Our first step in the dynamic paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from an optimal solution to subproblems.

Let $A_{i....j}$ where i≤ j denotes the matrix that results from evaluating the product

A_i A_{i+1}....A_j.

If i < j then any parenthesization of the product A_i A_{i+1} .....A_j must split that product between A_k and A_{k+1} for some integer k in the range i ≤ k ≤ j. That is, for some value of k, we first compute the matrices $A_{i.....k}$ & $A_{k+1....j}$ and then multiply them together to produce the final product $A_{i....j}$. The cost of computing $A_{i....k}$ plus the cost of computing $A_{k+1....j}$ plus the cost of multiplying them together is the cost of parenthesization.

Step 2: A Recursive Solution: Let m [i, j] be the minimum number of scalar multiplication needed to compute the matrix$A_{i....j}$.

If i=j the chain consists of just one matrix $A_{i....i}=A_i$ so no scalar multiplication is necessary to compute the product. Thus m [i, j] = 0 for i= 1, 2, 3....n.

If i<j we assume that to optimally parenthesize the product we split it between A_k and A_{k+1} where i≤ k ≤j. Then m [i,j] equals the minimum cost for computing the subproducts $A_{i....k}$ and $A_{k+1....j}$+ cost of multiplying them together. We know A_i has dimension p_{i-1} x p_i, so computing the product $A_{i....k}$ and $A_{k+1....j}$takes p_{i-1} p_k p_j scalar multiplication, we obtain

$$m [i,j] = m [i, k] + m [k + 1, j] + p_{i-1}\ p_k\ p_j$$

There are only (j-1) possible values for 'k' namely k = i, i+1.....j-1. Since the optimal parenthesization must use one of these values for 'k' we need only check them all to find the best.

So the minimum cost of parenthesizing the product A_i A_{i+1}.....A_j becomes

$$m [i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{m [i, k] + m [k + 1, j] + p_{i-1}\, p_k p_j\} & \text{if } i < j \end{cases}$$

To construct an optimal solution, let us define s [i,j] to be the value of 'k' at which we can split the product A_i A_{i+1} .....A_j To obtain an optimal parenthesization i.e. s [i, j] = k such that

$$m [i,j] = m [i, k] + m [k + 1, j] + p_{i-1}\ p_k\ p_j$$

Algorithm of Matrix Chain Multiplication:

MATRIX-CHAIN-ORDER (p)

```
1. n   length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n    // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ←  ∞
8. for k  ← i to j-1
9. do q  ← m [i, k] + m [k + 1, j] + pi-1 pk pj
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.
```

We will use tables to construct an optimal solution.

Step 1: Constructing an Optimal Solution:

PRINT-OPTIMAL-PARENS (s, i, j)

1. if i=j
2. then print "A"
3. else print "("
4. PRINT-OPTIMAL-PARENS (s, i, s [i, j])
5. PRINT-OPTIMAL-PARENS (s, s [i, j] + 1, j)
6. print ")"

Analysis: There are three nested loops. Each loop executes a maximum n times.

1. l, length, O (n) iterations.
2. i, start, O (n) iterations.
3. k, split point, O (n) iterations

Body of loop constant complexity

Total Complexity is: O (n^3)

Algorithm with Explained Example:

Question: P {7, 1, 5, 4, 2}

Solution: Here, P is the array of a dimension of matrices.
So here we will have 4 matrices:
A_{17x1} A_{21x5} A_{35x4} A_{44x2}
i.e.
First Matrix A_1 have dimension 7 x 1
Second Matrix A_2 have dimension 1 x 5
Third Matrix A_3 have dimension 5 x 4
Fourth Matrix A_4 have dimension 4 x 2

Let say,
From P = {7, 1, 5, 4, 2} - (Given)
And P is the Position
p_0 = 7, p_1 =1, p_2 = 5, p_3 = 4, p_4=2.
Length of array P = number of elements in P
∴length (p)= 5
From step
Follow the steps in Algorithm in Sequence
According to Step 1 of Algorithm Matrix-Chain-Order

Step 1:
n ← length [p]-1
Where n is the total number of elements
And length [p] = 5
∴ n = 5 - 1 = 4
n = 4
Now we construct two tables m and s.
Table m has dimension [1.....n, 1.....n]
Table s has dimension [1.....n-1, 2.....n]

	1	2	3	4
1	0	35 ∞	48 ∞	42
2		0	20 ∞	28 ∞
3			0	
4				0

m-Table
[1....n, 1.....n]

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

n-Table
[1....n-1, 2.....n]

Now, according to step 2 of Algorithm:

for i ← 1 to n
this means: for i ← 1 to 4 (because n =4)
for i=1
m [i, i]=0
m [1, 1]=0
Similarly for i = 2, 3, 4
m [2, 2] = m [3,3] = m [4,4] = 0
i.e. fill all the diagonal entries "0" in the table m
Now,
l ← 2 to n
l ← 2 to 4 (because n =4)

Based on this we can code.

Exercise:

1.You are going on a long trip. You start on the road at milepost 0. Along the way there are n hotels, at mileposts a1 < a2 < · · · < an, where each ai is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance an), which is your destination.You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is (200 − x)2. You want to plan your trip so as to minimize the total penalty—that is, the sum, overall travel days, of the daily penalties.

Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

2.Suppose two teams, A and B, are playing a match to see who is the first to win n games (for some particular n). We can suppose that A and B are equally competent, so each has a 50% chance of winning any particular game. Suppose they have already played i + j games, of which A has won i and B has won j. Give an efficient algorithm to compute the probability that A will go on to win the match. For example, if i = n − 1 and j = n − 3 then the probability that A will win the match is 7/8, since it must win any of the next three games.

3.Question: P {6, 4, 9, 1, 5}

Solve this using algorithm of matrix chain multiplication(see the example i given in page 69).

Reference links for practicing more problems on online platforms:

1.https://iq.opengenus.org/list-of-dynamic-programming-problems/

2.https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/

3.https://codecrucks.com/divide-and-conquer/

Reference platform I used to write book:

1. Geeks for Geeks
2. Tutorials point
3. Abdul Buri videos

Practicing problems on any coding platform is fine.