

TIME COMPLEXITY ANALYSIS

**“With 100+
Time & Space
Complexity
Cheat Sheets**

$O(\sqrt{N})$

OPENGENUS



Time Complexity Analysis

With Time and Space Complexity Cheat Sheets !

Aditya Chatterjee
Ue Kiao, PhD.

Introduction to this Book

This book “**Time Complexity Analysis**” introduces you to the basics of Time Complexity notations, meaning of the Complexity values and How to analyze various Algorithmic problems.

We have tackled several significant problems and demonstrated the approach to analyze them and arrived at the Time and Space Complexity of the problems and Algorithms.

This is a **MUST-READ book** for all Computer Science students and Programmers. **Do not miss this opportunity.**

You will get a better idea to judge which approach will work better and will be able to make better judgements in your development work.

See the “Table of content” to get the list of exciting topics you will learn about.

Some of the key points you will understand:

- Random Access Memory does not take $O(1)$ time. It is complicated and in general, has a Time Complexity of $O(\sqrt{N})$.
- Multiplication takes $O(N^2)$ time, but the most optimal Algorithm (developed in 2019) takes $O(N \log N)$ time which is believed to be the theoretical limit.
- As per Time Complexity, finding the largest element and the i^{th} largest element takes the same order of time.

It is recommended that you go through this book twice. First time, you may skip the minute details that you may not understand at first go and get the overview.

In the second reading, you will get all the ideas, and this will strengthen your insights.

In 1950s, Computing was not a Science.

It was a collective effort by several Computer Scientists such as Robert Tarjan and Philippe Flajolet who analyzed several computational problems to demonstrate that Computation Problems are equally complicated as Physics and Mathematics Problems.

The ideas captured in this book include some of these analyses which glorified Computer Science and made it a Scientific field.

Book: Time Complexity Analysis

Authors: Aditya Chatterjee; Ue Kiao, PhD.

Contributors (7): Vansh Pratap Singh, Shreya Shah, Vikram Shishupalsingh Bais, Mallika Dey, Siddhant Rao, Shweta Bhardwaj, K. Sai Drishya.

Published: August 2021.

Contact: team@opengenius.org

If you would like to participate in OpenGenus's remote Internship program, apply at: internship.opengenius.org.

It will help you gain valuable practical experience and contribute to the community.

Table of content

#	Chapter	Page Number
1	Introduction to Time and Space Complexity (+ different notations)	1
2	How to calculate Time Complexity?	6
3	Meaning of different Time Complexity	10
4	Brief Background on NP and P	12
5	Does $O(1)$ time exist?: Cost of accessing Memory	16
6	Time Complexity of Basic Arithmetic Operations	35
6.1	Bitwise operations	36
6.2	Addition	38
6.3	Subtraction	42
6.4	Multiplication	47
6.5	Division	48
7	Analysis of Array	50
8	Analysis of Dynamic Array	52
9	Find largest element	55
10	Find Second largest element	57
11	Find i^{th} largest element	67
12	Time Complexity Bound for comparison-based sorting	75
12.1	Analysis of Selection Sort	80
12.2	Analysis of Insertion Sort	87
12.3	Analysis of Bubble Sort	95
12.4	Analysis of Quick Sort	101

13	Bound for non-comparison-based sorting	110
13.1	Analysis of Counting Sort	112
13.2	Analysis of Bucket Sort	119
14	Analysis of Linked List	130
15	Analysis of Hash functions	133
16	Analysis of Binary Search	137
17	Time and Space Complexity Cheat Sheets	140

In addition to this book, you should read:

“[Binary Tree Problems: Must for Interviews and Competitive Coding](#)” by Aditya Chatterjee, Ue Kiao and Srishti Guleria •

Introduction to Time and Space Complexity (+ different notations)

Time Complexity is a notation/ analysis that is used to determine how the number of steps in an algorithm increase with the increase in input size. Similarly, we analyze the space consumption of an algorithm for different operations.

This comes in the analysis of Computing Problems where the theoretical minimum time complexity is defined. Some Computing Problems are difficult due to which minimum time complexity is not defined.

For example, it is known that the Problem of finding the i -th largest element has a theoretical minimum time complexity of $O(N)$. Common approaches may have $O(N \log N)$ time complexity.

One of the most used notations is known as Big-O notation.

For example, if an algorithm has a Time Complexity Big-O of $O(N^2)$, then the number of steps is of the order of N^2 where N is the number of data.

Note that the number of steps is not exactly N^2 . The actual number of steps may be $4 * N^2 + N/3$ but only the dominant term without constant factors is considered.

Different notations of Time Complexity

Different notations of Time Complexity include:

- Big-O notation
- Little-O notation
- Big Omega notation
- Little-Omega notation
- Big-Theta notation

In short:

Notation	Symbol	Meaning
Big-O	O	Upper bound
Little-o	o	Tight Upper bound
Big Omega	Ω	Lower bound
Little Omega	ω	Tight Lower bound
Big Theta	Θ	Upper + Lower bound

This will make sense as you go through the details. We will revisit this table following it.

1. Big-O notation

Big-O notation to denote time complexity which is the **upper bound** for the function $f(N)$ within a constant factor.

$f(N) = O(G(N))$ where $G(N)$ is the big-O notation and $f(N)$ is the function we are predicting to bound.

There exists an N_1 such that:

$f(N) \leq c * G(N)$ where:

- $N > N_1$
- c is a constant

Therefore, Big-O notation is always **greater than or equal to** the actual number of steps.

The following are true for Big-O, but would not be true if you used little-o:

- $x^2 \in O(x^2)$
- $x^2 \in O(x^2 + x)$
- $x^2 \in O(200 * x^2)$

2. Little-o notation

Little-o notation to denote time complexity which is the **tight upper bound** for the function $f(N)$ within a constant factor.

$f(N) = o(G(N))$ where $G(N)$ is the little-o notation and $f(N)$ is the function we are predicting to bound.

There exists an N_1 such that:

$f(N) < c * G(N)$ where:

- $N > N_1$
- c is a constant

Note in Big-O, \leq was used instead of $<$.

Therefore, Little-o notation is **always greater than to the actual number of steps**.

The following are true for little-o:

- $x^2 \in o(x^3)$
- $x^2 \in o(x!)$
- $\ln(x) \in o(x)$

3. Big Omega Ω notation

Big Omega Ω notation to denote time complexity which is the **lower bound** for the function $f(N)$ within a constant factor.

$f(N) = \Omega(G(N))$ where $G(N)$ is the big Omega notation and $f(N)$ is the function we are predicting to bound.

There exists an N_1 such that:

$f(N) \geq c * G(N)$ where:

- $N > N_1$
- c is a constant

Therefore, Big Omega Ω notation is always less than or equal to the actual number of steps.

4. Little Omega ω

Little Omega ω notation to denote time complexity which is the **tight lower bound** for the function $f(N)$ within a constant factor.

$f(N) = \omega(G(N))$ where $G(N)$ is the little Omega notation and $f(N)$ is the function we are predicting to bound.

There exists an N_1 such that:

$f(N) > c * G(N)$ where:

- $N > N_1$
- c is a constant

Note in Big Omega, \geq was used instead of $>$.

Therefore, Little Omega ω notation is always less than to the actual number of steps.

5. Big Theta Θ notation

Big Theta Θ notation to denote time complexity which is the **bound** for the function $f(N)$ within a constant factor.

$f(N) = \Theta(G(N))$ where $G(N)$ is the big Omega notation and $f(N)$ is the function we are predicting to bound.

There exists an N_1 such that:

$0 \leq c_1 * G(N) \leq f(N) \leq c_2 * G(N)$ where:

- $N > N_1$
- c_1 and c_2 are constants

Therefore, Big Theta Θ notation is always less than and greater than the actual number of steps provided correct constant are defined.

In short:

Notation	Symbol	Meaning
Big-O	O	Upper bound
Little-o	o	Tight Upper bound
Big	Ω	Lower bound

Omega		
Little Omega	ω	Tight Lower bound
Big Theta	Θ	Upper + Lower bound

Big-O notation is used in practice as it is an upper bound to the function and reflects the original growth of the function closely. For denoting the Time Complexity of Algorithms and Computing Problems, we advise to use Big-O notation.

Note these notations can be used to denote space complexity as well.

How to calculate Time Complexity?

There are different ways to analyze different problems and arrive at the actual time complexity. Different problems require different approaches and we have illustrated several such approaches.

Some techniques include:

- Recurrence Relation of Algorithms
- Tree based analysis for problems like "Number of comparisons for finding 2nd largest element"
- Mathematical analysis for problems like "Number of comparisons for finding i-th largest element"

and many more approaches.

Recurrence relation is a technique that establishes a equation denoting how the problem size decreases with a step with a certain time complexity.

For example, if an algorithm is dealing with that reduces the problem size by half with a step that takes linear time $O(N)$, then the recurrence relation is:

$$T(N) = T(N/2) + O(N)$$

=> Time Complexity for problem size N = Time Complexity for problem size $N/2$ + Big-O notation of $O(N)$ or N steps

$$\Rightarrow T(N) = N + N/2 + N/4 + \dots + 1$$

$$\Rightarrow T(N) = N \log N$$

This analysis comes in a sorting algorithm which is Quick Sort.

For another sorting algorithm known as Stooge Sort, the recurrent relation is:

$$T(N) = 3 * T(2N/3) + O(1)$$

Solving this equation, you will get: $T(N) = O(N^{\log_3 1.5}) = O(N^{2.7095})$

Solving such recurrence relation requires you to have a decent hold in solving algebraic equations.

Some examples of recurrence relations:

Recurrence	Algorithm	Big-O
$T(N) = T(N/2) + O(1)$	Binary Search	$O(\log N)$
$T(N) = T(N/2) + O(N)$	Quick Sort	$O(N \log N)$
$T(N) = 3 * T(2N/3) + O(1)$	Stooge Sort	$O(N^{2.7095})$
$T(N) = T(N-1) + O(1)$	Linear Search	$O(N)$
$T(N) = T(N-1) + O(N)$	Insertion Sort	$O(N^2)$
$T(N) = 2 * T(N-1) + O(1)$	Tower of Hanoi	$O(2^N)$

Other techniques are not fixed and depend on the Algorithm you are dealing with. We have illustrated such techniques in the analysis of different problems. Going through the analysis of different problems, you will get a decent idea of how to analyze different computing problems.

Note: Recurrence relation techniques can be used to analyze algorithms but not general computing problems.

Let us solve some simple code examples:

Example 1:

```
for (i = 0; i < N; i++) {  
    sequence of statements of  $O(1)$   
}
```

The Time Complexity of this code snippet is $O(N)$ as there are N steps each of $O(1)$ time complexity.

Example 2:

```
for (i = 0; i < N; i++) {  
  for (j = 0; j < N; j++) {  
    sequence of statements of  $O(1)$   
  }  
}
```

The Time Complexity of this code snippet is $O(N^2)$ as there are N steps each of $O(N)$ time complexity. There are two nested for loops.

Example 3:

```
for (i = 0; i < N; i++) {  
  for (j = 0; j < N-i; j++) {  
    sequence of statements of  $O(1)$   
  }  
}
```

Number of steps = $N + (N-1) + (N-2) + \dots + 2 + 1$

Number of steps = $N * (N+1) / 2 = (N^2 + N)/2$

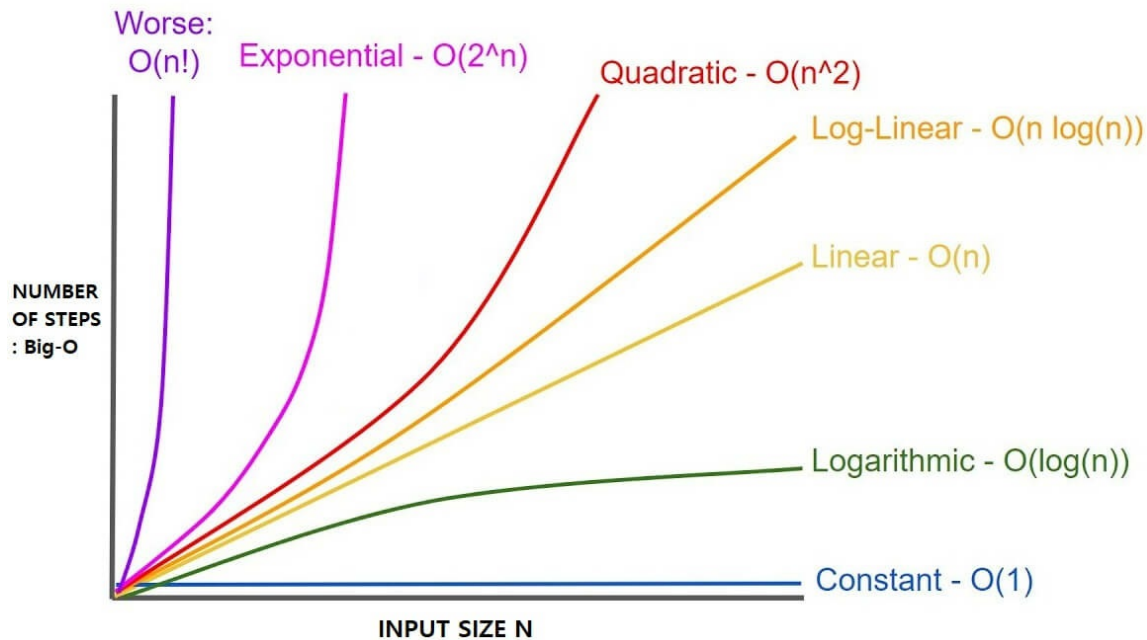
The Time Complexity of this code snippet is $O(N^2)$ as the dominant factor in the total number of comparisons is $O(N^2)$ and the division by 2 is a constant so it is not considered.

Meaning of different Time Complexity

Various Time Complexity Big-O values have different names. Following table lists some of the common Big-O values in increasing order of value/time:

Big-O	Known as
$O(1)$	Constant Time
$O(\log^* N)$	Iterative Logarithmic Time
$O(\log N)$	Logarithmic Time
$O(N)$	Linear Time
$O(N \log N)$	Log Linear Time
$O(N^2)$	Quadratic Time
$O(N^p)$	Polynomial Time
$O(c^N)$	Exponential Time
$O(N!)$	Factorial Time

Following is the visualization of some of the above notations:



Some points:

The target is to achieve the lowest possible time complexity for solving a problem.

For some problems, we need to go through all elements to determine the answer. In such cases, the minimum Time Complexity is $O(N)$ as this is the time required to read the input data.

For some problems, the theoretical minimum time complexity is not proved or known. For example, for Multiplication, it is believed that the minimum time complexity is $O(N \log N)$ but it is not proved. Moreover, the algorithm with this time complexity has been developed in 2019.

If a problem has only an exponential time algorithm, the approach to be taken is to use approximate algorithms. Approximate algorithms are algorithms that get an answer close to the actual answer (not but exact) at a better time complexity (usually polynomial time).

Several real-world problems have only exponential time algorithms so approximate time algorithms are used in practice. One such problem is the Travelling Salesman Problem. Such problems are NP-hard.

Brief Background on NP and P

Computation problems are classified into different complexity classes based on the minimum time complexity required to solve the problem.

Different complexity classes include:

- P (Polynomial)
- NP (Non-polynomial)
- NP-Hard
- NP-Complete

P:

The class of problems that have polynomial-time deterministic algorithms (solvable in a reasonable amount of time).

P is the Set of problems that can be solved in polynomial time.

NP:

The class of problems that are solvable in polynomial time on a non-deterministic algorithm.

Set of problems for which a solution can be verified in polynomial time.

P vs NP

If the solution to a problem is easy to check for correctness, must the problem be easy to solve?

P is subset of NP. Any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time.

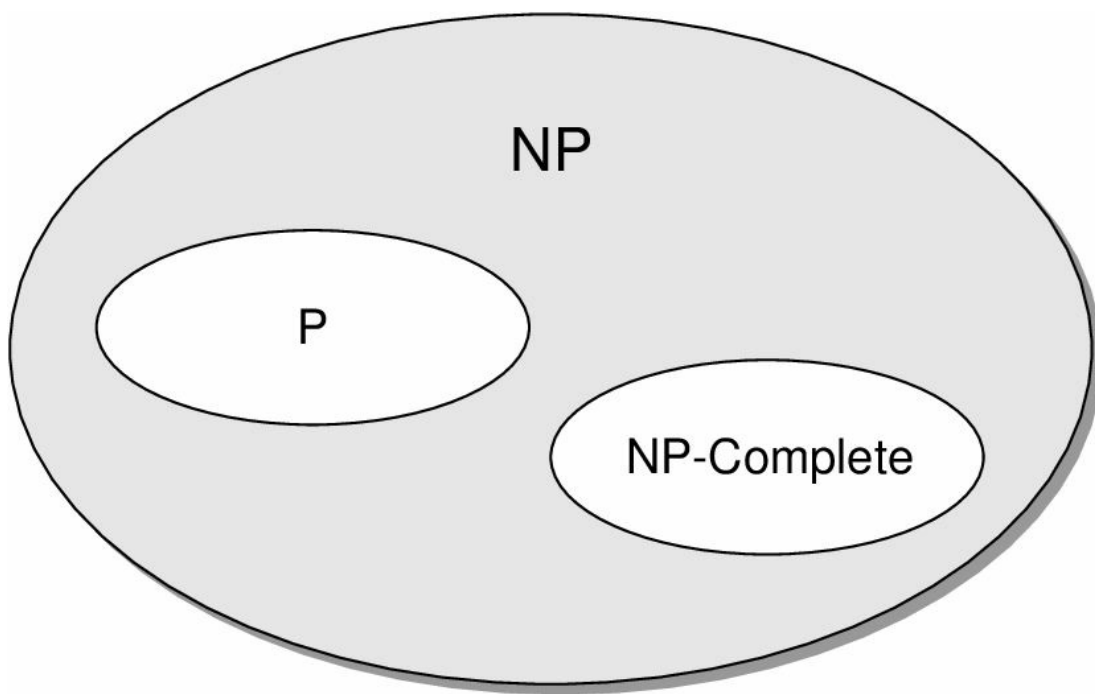
NP - HARD

Problems that are "at least as hard as the hardest problems in NP".

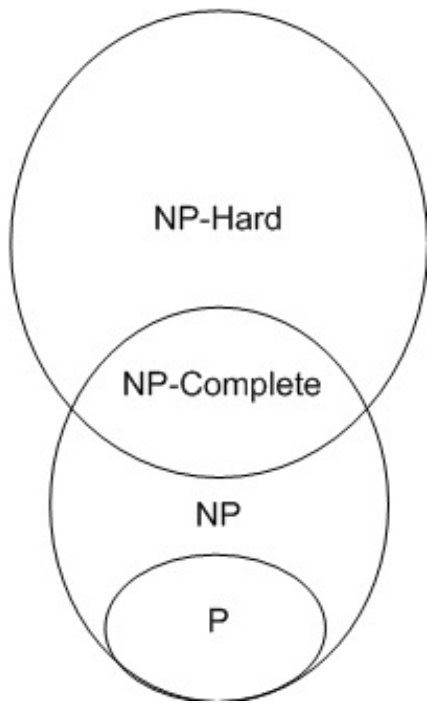
NP-complete

Problems for which the correctness of each solution can be verified quickly, and a brute-force search algorithm can actually find a solution by trying all possible solutions.

Relation between P and NP:



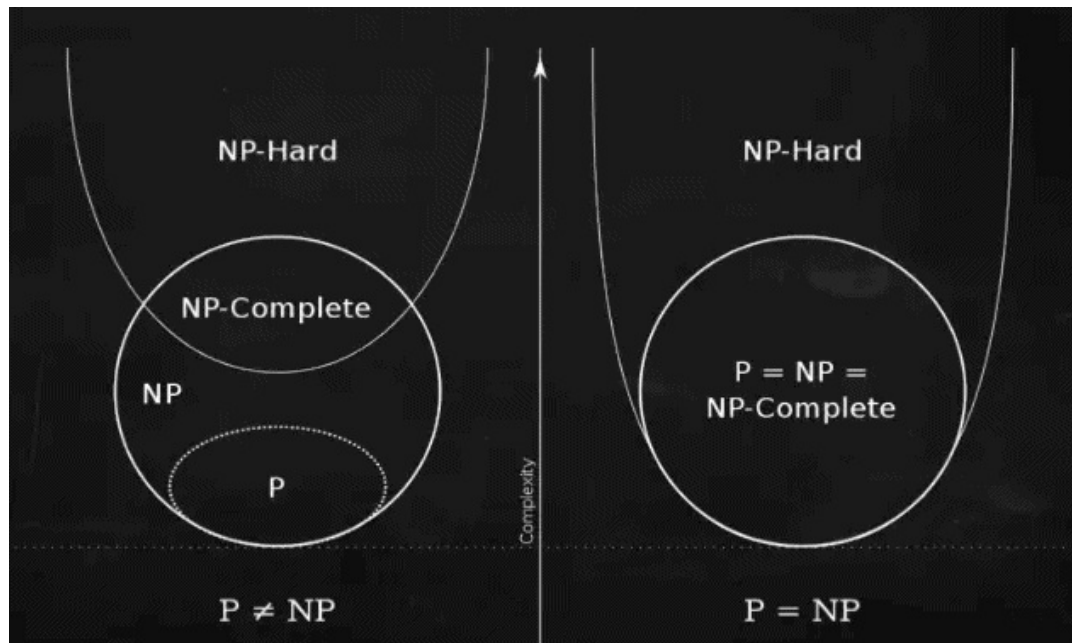
- P is contained in NP.
- NP complete belongs to NP
- NP complete and P are exclusive



One of the biggest unsolved problems in Computing is to prove if the complexity class P is same as NP or not.

P = NP or P \neq NP

If $P = NP$, then it means all difficult problems can be solved in polynomial time.



There are techniques to determine if a problem belongs to NP or not. In short, all problems in NP are related and if one problem is solved in polynomial time, all problems will be solved.

Does $O(1)$ time exist?: Cost of accessing Memory

In this chapter, we have taken an in-depth look at the operations and algorithms that have a constant time in terms of asymptotic notation of time complexities. Is $O(1)$ really a practical way of representing the time complexity of certain algorithms/ operations?

This is one of the most important chapters of this book so follow along carefully to build your fundamentals.

Sub-topics:

1. Getting started with the Memory model
2. Let's talk about $O(1)$
3. Digging deep into access time
4. Testing our claim
5. Conclusions
 - a. Physical analysis
 - b. Analysis of algorithms

Myth: Access memory at a particular address takes constant time $O(1)$.

1. Getting started with the Memory model

Before we dive into the Time Complexity analysis, we need to understand the memory model of Modern Computers. You will see that the real Time Complexity has a direct link with this structure and moreover, we will show that No Physical Computer can go beyond a certain limit by following rules of Physics.

1.1 Random Access Machine (RAM) and External Memory Model

Let us talk about the Random Access Machine (RAM).

A RAM physically consists of a central processing unit (CPU) and a memory. The memory itself has various cells inside it that are indexed by means of positive integers. A cell is capable of holding a bit-string. Talking about the

CPU, it has a finite number of registers (an accumulator and an address register, to be specific).

In a single step, a RAM can either perform an operation (simple arithmetic or Boolean operations) on its registers or access memory. When a portion of the memory is accessed, the content of the memory cell indexed by the content of the address register is either loaded into the accumulator or is directly written from the accumulator.

In this case, two timing models are used which are:

- The **unit-cost RAM** in which each operation has cost one, and the length of the bit-strings that can be stored in the memory cells and registers is bounded by the logarithm of the size of the input.
- The **logarithmic-cost RAM** in which the cost of an operation is equal to the sum of the lengths (bits) of the operands, and the contents of memory cells and registers do not have any specific constraints.

Now that you have an idea about the RAM, let us get an idea about the External Memory Model. An EM machine is basically a RAM that has two levels of memory.

Getting into the specifics, the levels are referred to as cache and main memory or memory and disk, respectively.

The CPU operates on the data that present in the cache. Both cache and main memory are each divided into blocks of B cells, and the data is transferred between them in blocks. The cache has size M and hence consists of M/B blocks whereas the main memory is infinite in size.

The analysis of algorithms in the EM-model actually bounds the number of CPU steps and the number of block transfers. The **time taken for a block transfer to be completed is equal to the time taken by $\Theta(B)$ CPU steps**. In this case, the hidden constant factor is significantly large, and this is why the number of block transfers are taken into consideration in the analysis.

1.2 Levels/Hierarchy of Memory

The CPU cache memory is divided into three levels. The division is done by taking the speed and size into consideration.

- **L1 cache:** The L1 (Level 1) cache is considered the fastest memory that is present in a computer system.

The L1 cache is generally divided into two sections:

- **The instruction cache:** Handles the information about the operation that the CPU must perform.
- **The data cache:** Holds the data on which the operation is to be performed.

The L1 cache is usually 100 times faster than the system RAM.

- **L2 cache:** The L2 (Level 2) cache is usually slower than the L1 cache but has the upper hand in terms of size. The size of the L2 cache depends on the CPU, but generally it is in the range of 256kB to 8MB. It is usually 25 times faster than the system RAM.
 - **L3 cache:** The L3 (Level 3) is the largest but also the slowest cache memory unit. Modern CPUs include the L3 cache on the CPU itself. The L3 cache is also faster than the system RAM, although it is not very significant.

1.3 Virtual Memory

Virtual memory is a section of the volatile memory that is created temporarily on the storage drive for the purpose of handling concurrent processes and to compensate for the high RAM usage.

Usually, virtual memory is considerably slower than the main memory because of the fact that the processing power is being consumed by the transportation of data instead of the execution of instructions. A rise in latency is also observed when a system needs to use virtual memory.

1.4 Virtual Address Translation (VAT) Model

The Virtual Address Translation (VAT) machines are RAM machines that use virtual addresses and also account for the cost of address translations, and it has to be taken into consideration as well during the analysis of various algorithms.

For the purpose of understanding and demonstration, let us assume that:

- $P = \text{Page size } (P = 2^p)$
- $K = \text{Arity of translation tree } (K = 2^k)$
- $d = \text{Depth of translation tree } (d \log_k (\text{max used virtual address}))$

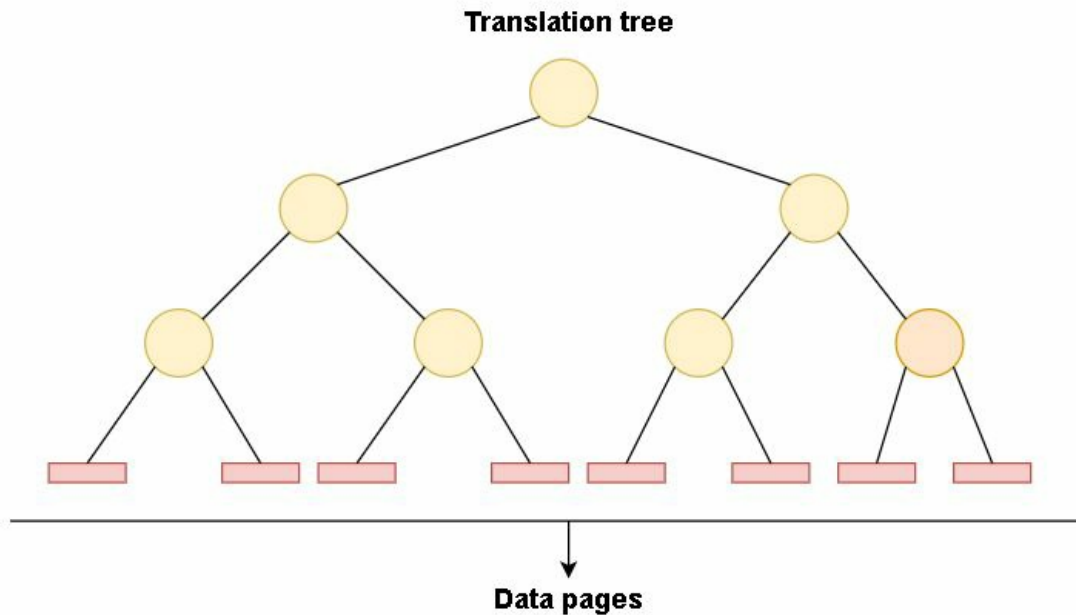
- $W = \text{Size of the TC cache}$
- $\tau = \text{Cost of a cache fault (number of RAM instructions)}$

Both the physical and virtual addresses are strings in $\{0, K-1\}^d \{0, \dots, P-1\}$. Such strings correspond to a number present in the interval $[0, K^d P-1]$ in a natural manner. The $\{0, K-1\}^d$ part of the address is the index and is of length d which is an execution parameter that is fixed prior to the execution. It is also assumed that $d = \lceil \log_k (\text{maximum used virtual address}/P) \rceil$. The $\{0, \dots, P-1\}$ part of the address is called the page offset where P is the page size (as specified before).

Coming to the actual **translation process**, it is basically a **tree walk/traversal**.

We have a K -ary tree T of height d . The nodes of the tree are the pairs (l, i) where $l \geq 0$ and $i \geq 0$. Here l is the layer of the node and i is the total number of nodes itself. The leaves of the tree are present on layer zero and a node (l, i) on layer $l \geq 1$ has K children on the layer $l-1$. In particular, the node $(d, 0)$, the root, has children $(d-1, 0), \dots, (d-1, K-1)$. The leaves of the tree are the physical pages of the main memory of a RAM machine. For example, to translate the virtual address $x_{a-1} \dots x_0 y$, we will start from the root of T and then follow the path which will be described by $x_{a-1} \dots x_0$. This path is referred to as the translation path for the address and it ends in the leaf $(0, \sum_{0 \leq i \leq a-1} x_i K^i)$.

The following figure depicts the process in a generic way:



Conclusion:

If D and i are integers such that $D \geq 0$ and $i \geq 0$, the translation paths for addresses i and $i+D$ differ in at least the $\max(0, \log_k(D/P))$ nodes.

1.5 Translation costs and Cache faults

In a research that was conducted earlier, six simple programs were timed for different inputs, namely:

- permuting the elements of an array of size n
- random scan of an array of size n
- n random binary searches in an array of size n
- heap sort of n elements
- Introsort (hybrid of quick sort, heap sort and insertion sort) of n elements
- Sequential scan of an array of size n

It was found that for some of the programs, the measured running time coincided with the original predictions of the models. However, it was also found that the running of random scan seems to grow as $O(N \log^2 N)$, and that does not really coincide with the original predictions of the models that is $O(N)$.

You might be wondering as to why are the predicted and the measured run times different? (by a factor of $N \log N$).

We will have to do some digging in order for us to understand that. Modern computers have virtual memories, and each individual process has its own virtual address space. Whenever a process accesses memory, the virtual address has to be translated into a physical address (similar working as that of NAT, network address translation). This translation of virtual addresses into physical addresses is not trivial, and hence has a cost of its own.

The translation process is usually implemented as a hardware-supported **walk in a prefix tree** and this tree is stored in the memory hierarchy and because of this, the translation process may also **sustain cache faults**. Cache faults are a type of page fault that occur when a program tries to reference a section of an open file that is not currently present in the physical memory. The **frequency of cache faults is actually dependent upon the locality of the memory accesses** (lesser local memory accesses result in more cache faults).

The **depth of the translation tree is logarithmic in the size of an algorithm's address space** and hence, in the worst case, all memory accesses may lead to a logarithmic number of cache faults during the translation process.

2. Let's talk about $O(1)$

We know that the time complexity for iterating through an array or a linked list is $O(N)$, selection sort is $O(N^2)$, binary search is $O(\log N)$ and a lookup in a hash table is $O(1)$.

However, we will prove that accessing memory is not a $O(1)$ operation, instead it is a **$O(\sqrt{N})$ operation**.

We will try to prove this through both theoretical and practical means, and you will be convinced by the reasoning.

3. Digging deep into access time

Let us take an example.

Try to relate this example with how memory access works in a working

system. Suppose you run a big shop that deals with games. Your shop has a circular storage for games, and you have placed the games in an orderly manner and you remember as to which game can be found in which drawer/place. A customer comes to your shop, and asks for game X. You know where X is placed in your shop. Now, what would be the time taken by you in order for you to grab and bring the game back to the customer? It would obviously be bounded by the distance that you have to walk to get that game, the worst case being when the game is present at either end of the shop, i.e., you will have to walk the full radius r .

Now let us take another example.

Try to relate this example with how memory access works in a working system as well. Now let us suppose that you have upgraded your shop's infrastructure, and its storage capacity has increased tremendously. The radius of your shop is now twice the original radius and hence the storage capacity has increased too. Now, in the case of the worst-case scenario, you will have to walk twice the distance to retrieve a game. But we also have to think about the fact that the area of the shop is now doubled and therefore it can contain four times the original quantity of games.

From this relation, we can infer that N games that can fit into our shop storage is proportional to the square of the radius r of the shop.

Hence, $N \propto r^2$

Since we know that the time taken T to retrieve a game is proportional to the radius r of the shop, we can infer the following relation:

$$\Rightarrow N \propto T^2$$

$$\Rightarrow T \propto \sqrt{N}$$

$$\Rightarrow T = O(\sqrt{N})$$

This scenario is roughly comparable to a CPU which has to retrieve a piece of memory from its library, which is the RAM.

The speed obviously differs significantly, but it is after all bounded by the

speed of light.

In our example, we assumed that the storage space of our shop had a circular infrastructure. So how much data can be fitted within a specific distance r from the CPU?

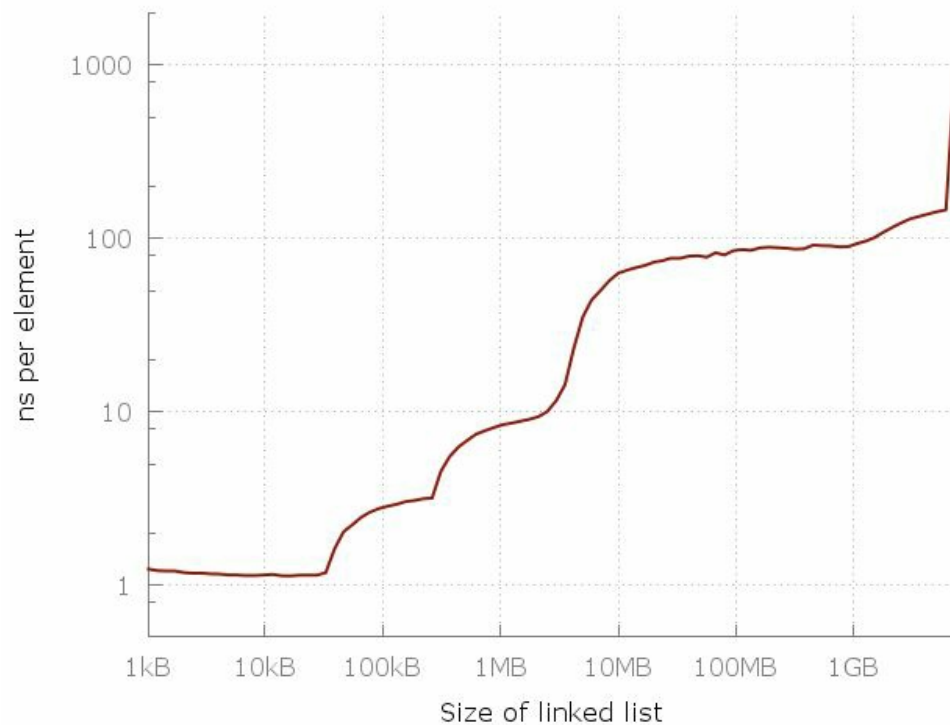
What if the shape of the storage space was spherical? Well, in that case, the amount of memory that could be fitted within a radius r would then become proportional to r^3 .

However, in practice, computers are actually rather flat because of many factors such as form-factor, cooling issues, etc.

4. Testing our claim

Let us say that we have a program that iterates over a linked list of length N , consisting of about 64 to 400 million elements. Each node also contains a 64 bits pointer and 64 bits of dummy data. The nodes of the linked list are also jumbled around in memory such that the memory access for each node is random. We will be measuring iterating through the same list a few times, and then plotting the time taken per element. Well, if the access time really were $O(1)$, then we would get a flat plot (in the graph).

However, interestingly, this is what we actually get:



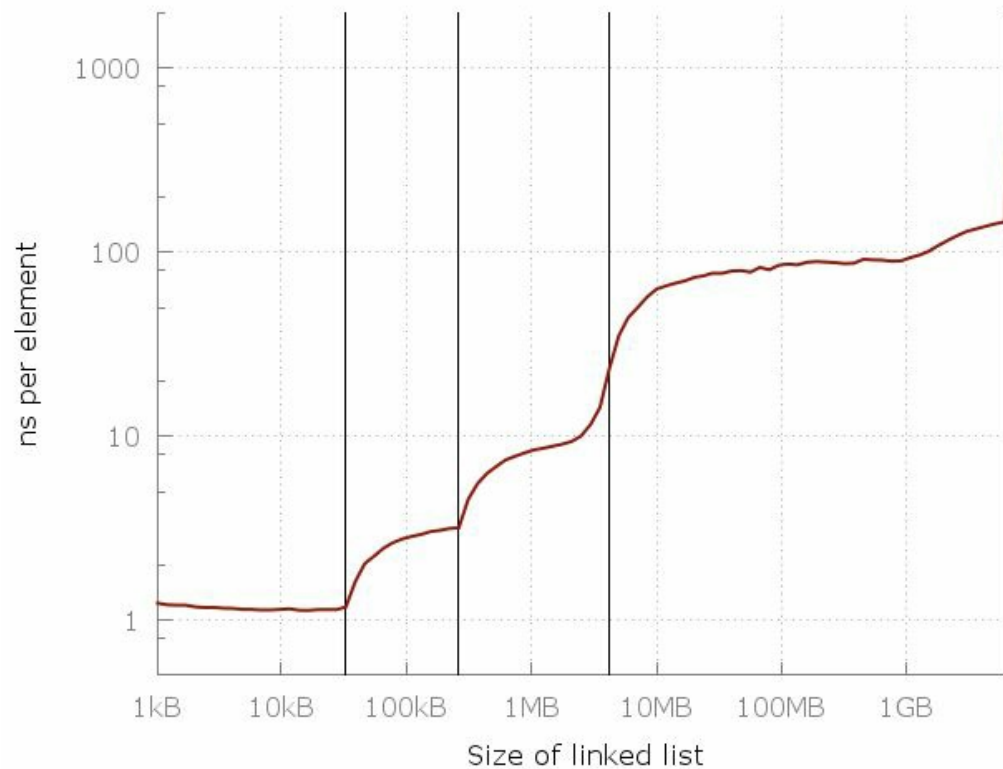
The graph plotted above is a log-log graph, and hence the differences that are visible in the figure are actually huge in size. In the figure, there is a noticeable spike or jump from about one nanosecond per element all the way up to a microsecond.

Now why is that happening?

Try to recall what we spoke about in the “*Getting started with the Memory model*” section. The answer to that question is caching. Off-chip or distant communication in RAM can be quite slow at times and in order to combat this con, the concept of cache was introduced. The cache basically is an on-chip storage that is much faster and closer.

The system on which these tests were conducted had three levels of cache called L1, L2, L3 of 32 kiB, 256 kiB and 4 MiB each respectively with 8 GiB of RAM (with about 6GiB being free at the time of the experiment).

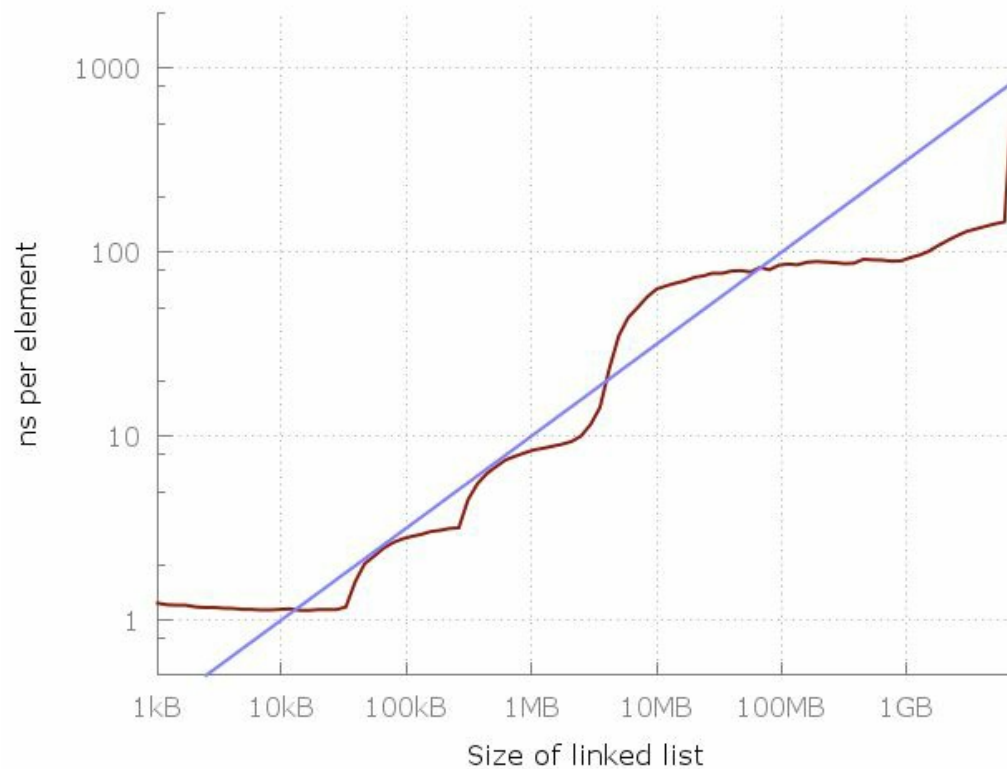
In the following figure, the vertical lines are represented by the cache sizes and the RAM:



You might notice a pattern in the figure posted above. The importance and the role of caches also becomes very clear.

If you notice, you can see that there is a roughly a 10x slowdown between 10kB and 1MB. The same is true for the phase between 1MB and 100MB and for 100MB and 10Gb as well. It appears as if for each 100-fold increase in the usage of the memory, we get a 10-fold slowdown.

Let us compare that with our claim:



In the figure above, the blue line corresponds to a **$O(\sqrt{N})$ cost** each time the memory is accessed.

So, what happens when we reach the latter (right) side of the graph? Will there be a continuous rise in the graph, or would it become flat?

It will actually flatten for a while until the memory could no longer be fit on the SSD and the help of the HDD is needed. From there, we would go to a disk server and then to a datacenter.

For each such jump, the graph would flatten for a while, but the rise will always arguably come back.

5. Conclusions

5.1 Physical analysis

The cost of a memory access depends upon the amount of memory that is actually being accessed as $O(\sqrt{N})$, where N = amount of memory being touched between each memory access.

From this, we can conclude that if one touches the same list/table in a repetitive manner, then:

- Iterating through a linked list will become a $O(N\sqrt{N})$ operation
- the binary search algorithm will become a $O(\sqrt{N})$ operation
- a hash map lookup will also become a $O(\sqrt{N})$ operation.

What we do between subsequent operations performed on a list/table also matters. If a program is periodically touching N amount of memory, then any one memory access should be $O(\sqrt{N})$. So if we are iterating over a list of size K , then it will be a $O(K\sqrt{N})$ operation. Now, if we iterate over it again (without accessing any other part of the memory first), it will become a $O(K\sqrt{K})$ operation.

In the case of an array (size K), the operation will be $O(\sqrt{N} + K)$ because it is only the first memory access that is actually random. However, iterating over it again will become a $O(K)$ operation.

That makes an array more suitable for scenarios where we already know that iteration has to be performed.

Hence, memory access patterns can be of significant importance.

5.2 Analysis of algorithms

Before proceeding with our conclusive remarks on certain algorithms, some assumptions that are necessary for the analysis to be meaningful have been written below.

Moving a single translation path to the translation cache costs more than a single instruction but does not cost more than the instructions that are the equivalent to the size of a page. If at least one instruction is performed for each cell in a page, then the cost of translating the index of that page can be

amortized, i.e., $1 \leq \tau d \leq P$.

The fan-out of the translation tree is at least 2, i.e., $K \geq 2$.

The translation tree suffices to translate all the addresses but is not significantly larger, i.e., $m/P \leq K^d \leq 2m/P$. Because of this, we get $\log(m/P) \leq d \log K = dk \leq \log(2m/P) = 1 + \log(m/P)$. In turn, we have $\log_k(m/P) \leq d \leq \log_k(2m/P) = (1 + \log(m/P)) / k$.

The translation cache is capable of holding at least one translation path, but it is still significantly smaller than the main memory, i.e., $d \leq W < m^\Theta$ where $\Theta \in (0, 1)$.

Sequential access: Let us suppose that we have an array of size n . If b is the base address of an array, then we need to translate addresses in the order $b, b+1, b+2, \dots, b+n-1$. For P consecutive accesses, the translation path is going to stay constant, and so at most $2n/P$ indices must be translated for a total cost of at most $\tau d(2n/P)$. Referring to our first assumption, this would translate to at most $\tau d(n/P + 2) \leq n + 2P$.

The analysis can be improved significantly by keeping the current translation path in the cache and hence limiting the first translation's faults to at most d . The translation path changes after every P^{th} access. A change is observed in this case at most a total of $\lceil n/P \rceil$ times. Whenever the path does undergo a change, the last node will also change as well. The node that is present next to the last node also changes after every K^{th} change of the last node and hence changes at most $\lceil n/(PK) \rceil$ times.

Hence, in total, we incur:

$$d + \sum_{0 \leq i \leq d} i \lceil \frac{n}{PK^i} \rceil \leq 2d + \frac{K}{K-1} \frac{n}{P}$$

The cost is bounded by $2\tau d + 2\tau n/P \leq 2P + 2n/d$, which is asymptotically smaller than the RAM complexity.

Random access: In random access, the worst case occurs when no node of any translation path is in the cache. The total translation cost in this case is bounded by τdn and is at most $\tau \log_k (2n/P)$.

Let us assume that the translation cache satisfies the initial segment property for the purpose of the analysis of a lower bound. The translation path will end in a random leaf of the translation tree and for every leaf, some initial segment of the path that ends in this leaf will be cached. Let u be an uncached node of the translation tree of minimal depth, and let v be a cached node of the translation tree of maximal depth. If the depth v is larger than u by a margin of 2 or more, then it will be better to cache u instead of v . Hence, the expected length of the path cached is at most $\log_k W$ and the expected number of faults incurred during the translation would be $d - \log_k W$.

Hence, the total expected cost is at least $\tau \log_k n/(PW) \leq \tau n(-\log_k W)$, which is asymptotically larger than the RAM complexity.

Binary search: Let us assume that we have to perform n binary searches in an array of length n . Each search will search for a random element of the array. For the sake of simplicity, let us assume the n is a power of two minus one. Now we cache the translation path of the top l layers of the search tree and the translation path of the current node of our search. The top l layers contain $2^{l+1} - 1$ vertices and so we need to store at most $d2^{l+1}$ nodes of the translation tree. Keep in mind that this is only feasible if $d2^{l+1} \leq W$.

The remaining of the $\log N - l$ steps of the binary search causes at most d cache faults. Hence, the total cost per search will be bounded by:

$$\tau d (\log n - l) \leq \tau \log_k (2n/P) (\log n - l) = \tau \log_k \frac{2n}{P} \log \frac{2nd}{W}$$

The expected number of cache misses per search will be at least:

$$\frac{1}{2} \sum_{1 \leq i \leq \log n - p} \left\lceil \frac{1}{k} \log \frac{n}{2^i P} \right\rceil \geq \frac{1}{4} \log_k \frac{2nd}{PW} \log \frac{2nd}{PW}$$

Heapify and Heapsort: An array or a list ($A[1...n]$) storing elements from an ordered list is heap-ordered if $A[i] \leq A[2i]$ and $A[i] \leq A[2i+1] \quad \forall \quad 1 \leq i \leq \lfloor n/2 \rfloor$. An array can be transformed to a heap by calling operation $\text{sift}(i)$ for $i = \lfloor n/2 \rfloor$ down to 1. This function repeatedly changes $z = A[i]$ with the smaller of its two children until the heap property is satisfied. We will use the following translation replacement strategy.

$$z = \min(\log n, \lfloor (W - 2d - 1) / \lfloor \log_k(n/P) \rfloor \rfloor - 1)$$

The extremal translation (for n consecutive addresses, paths to the first and last address in the range) paths ($2d-1$ nodes), non-extremal (nodes that are not on the extremal path) paths of the translation paths for z address (a_0, \dots, a_{z-1}) and one additional translation path a_∞ ($\lfloor \log_k(n/P) \rfloor$ nodes for each) are stored. The additional translation path is only required when $z \neq \log N$. When this array is undergoing siftdown, for any element $A[i]$, a_0 will be equal to the address of $A[i]$, a_1 will be the address of one of the children of i (if $A[i]$ is moved, then this is where will be moved), a_2 will be the address of one of the grandchildren of i (in case i is moved down two levels, it will be moved here), and so on. The additional translation path is only used for the addresses that are more than z levels below the level containing i .

Let's place an upper bound on the number of the translation cache misses. Preparing the extremal paths can cause up to $2d + 1$ misses. Now we have to consider the translation cost for a_i , $0 \leq i \leq z-1$. In the case of a_i , $n/2^i$ unique values are assumed. On the assumption that the siblings in the heap always lie in the same page, the index of each a_i will decrease over time, and hence the total number of translation cache misses will be bound to the number of non-extremal nodes in range. We obtain the following bound in terms of

translation cache misses:

$$\frac{2n}{P} = O\left(\frac{n}{P}\right)$$

If $p + (l-1)k < i \leq p + lk$, where $l \geq 1$ and $i \leq z-1$, we can use $x = n/2^i$ and obtain a bound in terms of translation cache misses of at most:

$$\frac{n}{2^i} \cdot l + \frac{2n}{PK^l} = O\left(\frac{n}{2^i} \cdot l + \frac{2n}{2^i}\right) = O\left(\frac{n}{2^i}(l+2)\right) = O\left(n \frac{i}{2^i}\right)$$

Now, there are a total of $n/2^z$ siftdowns in layers z and above, and they use the additional translation path. For each such siftdown, we only need to translate $\log n$ addresses and each such translation will cause less than d misses. The total will be lesser than $n(\log n)d/2^z$. We get:

$$2d+1+(p+1)O\left(\frac{n}{P}\right) + \sum_{p \leq i \leq z-1} O\left(n \frac{i}{2^i}\right) + \frac{nd \log n}{2^z} = O\left(\tau\left(d + \frac{np}{P}\right)\right)$$

Next, we can prove the lower bound by assuming that $W < n/2P$.

The addresses will start being swept a_0 , then a_1 , and so on. No other accesses occur in the sub-array during this time. Hence, if the Least Recently Used (LRU) strategy is used, there will be at least $pn/(2P)$ translation cache misses to the lowest level of the translation tree. This will give us the $\Omega(np/P)$ part of the lower bound and so the total cost will be $\Omega(\tau(d + np/P))$.

Coming to the sorting phase of heapsort, the element that is stored in the root is repeatedly removed and moved to the rightmost leaf to the root, after which the siftdown process occurs to restore its heap property. The siftdown will start in the root and after accessing the address i of the heap, it moves to

the address $2i$ or $2i+1$. For analysis purposes, let us assume that the data cache and the translation cache are same in terms of size, i.e., $W = M$. The top l layers of the heap will be stored in the data cache and the translation paths to the vertices to these layers will be stored in the translation cache, where $2^{l+1} < M$. Each of the remaining $\log N - l$ sift-down steps might cause d cache misses. The total number of cache faults will hence be bounded by:

$$nd(\log n - l) \leq n \log_K \left(\frac{2n}{P} \right) \log \left(\frac{4n}{W} \right)$$

You must have a deep idea of Time Complexity by now.

With this, the myths must be clear by now. As we move forward, we will assume the memory access take constant time with no cache faults for simplicity.

Time Complexity of Basic Arithmetic Operations

Basic Arithmetic operations like addition, subtraction, multiplication and division are assumed to be constant time operations but in reality, it is not.

For practical implementations, addition and subtraction are assumed to be constant time operations while Multiplication and Division are assumed to be costlier operations compared to addition and subtraction and requires extra clock cycles.

In summary, the Time Complexity of Basic Arithmetic Operations are:

Operation	Usual Time Complexity	Optimal Time Complexity	Assumed Time Complexity
Addition	$O(N)$	$O(N)$	$O(1)$
Subtraction	$O(N)$	$O(N)$	$O(1)$
Multiplication	$O(N^2)$	$O(N \log N)$	$O(1)$
Division	$O(N^2)$	$O(N \log N)$	$O(1)$
Note: N is the number of bits; If number is M , then $N = \log M$			

- *Usual Time Complexity*: Time Complexity of Algorithms that are usually used.
- *Optimal Time Complexity*: Time Complexity of the most optimal algorithm (theoretically) for the given operation.
- *Assumed Time Complexity*: Time Complexity assumed in practice and in the analysis of other algorithms involving these operations.

We will explore each arithmetic operation deeper.

Data is represented in Binary format internally in a Computing Device. So, basic operations like addition (+) work on binary data. To understand the different arithmetic operations and why they are not constant time $O(1)$, we need to get some background on Binary data/ Bitwise operations.

Basics of Bitwise Operations

We know that computer stores all kinds of data (videos, files, photos, etc.) in

the form of binary numbers 0s and 1s. These 0s and 1s are called bits and the various operations that can be carried out on these binary numbers are called bitwise operations.

The various bitwise operators are given below

Name	Symbol	Usage	What it does
Bitwise And	&	$a \& b$	Returns 1 only if both the bits are 1
Bitwise Or		$a b$	Returns 1 if one of the bits is 1
Bitwise Not	~	$\sim a$	Returns the complement of a bit
Bitwise Xor	^	$a \wedge b$	Returns 0 if both the bits are same else 1
Bitwise Left shift	<<	$a \ll n$	Shifts a towards left by n digits
Bitwise Right shift	>>	$a \gg n$	Shifts a towards right by n digits

In short, the time complexity of these bitwise operations are:

Bitwise operation	Time Complexity	Parallel algorithm
AND	$O(N)$	$O(1)$
OR	$O(N)$	$O(1)$
NOT	$O(N)$	$O(1)$
XOR	$O(N)$	$O(1)$
LEFT SHIFT	$O(N)$	$O(1)$
RIGHT SHIFT	$O(N)$	$O(1)$
Note: N is the number of bits		

You will understand the time complexity better once we know how these bitwise operations works and how parallel algorithm is common in this case for all modern computing device.

Let us take an example and see how each of these operators work.

Consider 2 decimal numbers a and b.

a = 25 the binary equivalent of 25 is 00011001

b = 14 the binary equivalent of 14 is 00001110

1. **Bitwise AND** - The Bitwise and returns 1 only if both the bits are 1.

Number a = 25	0	0	0	1	1	0	0	1
Number b = 14	0	0	0	0	1	1	1	0
a & b	0	0	0	0	1	0	0	0

2. **Bitwise OR** - The Bitwise or returns 1 if either of the bits is 1.

Number a = 25	0	0	0	1	1	0	0	1
Number b = 14	0	0	0	0	1	1	1	0
a b	0	0	0	1	1	1	1	1

3. **Bitwise NOT** - The Bitwise not returns the complement of the bit.

Number a = 25	0	0	0	1	1	0	0	1
~ a	1	1	1	0	0	1	1	0

4. **Bitwise XOR** - The Bitwise XOR returns 1 only if one of the bits is zero.

Number a = 25	0	0	0	1	1	0	0	1
Number b = 14	0	0	0	0	1	1	1	0
a ^ b	0	0	0	1	0	1	1	1

5. **Bitwise Left Shift**

Number a = 25	0	0	0	1	1	0	0	1
a << 3	1	1	0	0	1	0	0	0

In the above image, we can see that three 0s have been added on the right side that is after the Least significant bit, causing the shift towards the left side.

6. **Bitwise Right shift**

Number a = 25	0	0	0	1	1	0	0	1
a >> 3	0	0	0	0	0	0	1	1

In the above image, we can see that three 0s have been added on the left side that is after the Most significant bit, causing the shift towards the right side.

Now since we have got the idea of how the bitwise operators work, let us move on to adding two numbers without the addition operator.

Adding two numbers using bitwise operators

Let's first take a look at how addition takes place at the binary level and understand it before trying to do it with bitwise operators.

carry	0	0	1	1	0	0	0	0
Number 1 = 25	0	0	0	1	1	0	0	1
Number 2 = 14	0	0	0	0	1	1	1	0
Sum = 39	0	0	1	0	0	1	1	1

The binary addition is pretty similar to usual addition. From the above example, we can understand that:

$$1 + 0 = 0 + 1 = 1$$

$$0 + 0 = 0$$

$$1 + 1 = 10 \text{ that is the binary equivalent of } 2$$

And another important point to note is that when we get 10, 1 is taken over to the carry and 0 is kept at the bottom itself.

A truth table will give a better understanding of how the binary addition takes place

A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

From the truth table, we infer that

- The carry expression is $A \& B$
- The Sum expression is $A \wedge B$

Using the above two expressions the addition of any two numbers can be done as follows.

Steps

1. Get two positive numbers a and b as input
2. Then checks if the number b is not equal to 0
 - a. Finds the carry value ($a \& b$)
 - b. Finds the sum value ($a \wedge b$) and stores it in the variable a
 - c. Then shifts the carry to the left by 1-bit stores it in b
3. Again, go back to step 2
4. When b becomes 0 it finally returns the sum

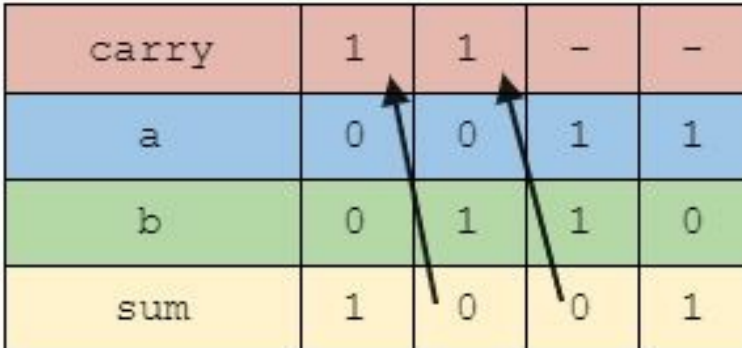
```
def Bitwise_add(a,b):
    while b != 0:
        # Carry value is calculated
        carry = a & b
        # Sum value is calculated and stored in a
        a = a ^ b
        # The carry value is shifted towards left by a bit
        b = carry << 1
    return a # returns the final sum
```

The whole idea behind this code is that the carry gets added again with the

sum value. So, what we do is we find the value of the carry separately using the expression $a \& b$ and use it again to find the sum. We keep on doing this till the carry value becomes 0.

Another important point to note here is that we shift the carry towards the left by 1 bit. That's because the carry value is added to the next bit rather than the current bit. Take a look at this image to get a better understanding.

carry	1	1	-	-
a	0	0	1	1
b	0	1	1	0
sum	1	0	0	1



We keep on repeating this process till the carry value that is $a \& b$ becomes 0. And finally, we get the required sum of the two numbers.

Bitwise add using Recursion

Adding the numbers using the bitwise operators can also be done in a recursive manner. The same logic takes place but instead of a loop, we use a recursive function to do the Addition.

```
def Bitwise_add(a,b):  
    if b == 0:  
        return a  
    else :  
        return Bitwise_add(a^b , (a&b) << 1)
```

In this code, a^b is the sum expression, and $(a\&b) \ll 1$ is the carry expression after shifting. So, it is directly passed as input to the recursive function. And when the carry expression becomes 0 that is b becomes 0, the recursion loop stops.

Time and Space Complexity of bitwise add

The time complexity of the algorithm is $O(N)$

where N is the number of bits in the numbers. The space complexity of the algorithm is $O(1)$.

Given a number M , the number of bits N is equal to $\log M$. So, adding two numbers M is not a constant time $O(1)$ operation. It takes $\log(M)$ time. Base of \log is 2.

Therefore, addition takes linear time.

Note: In Computing, data is of fixed size usually 32 bits or 64 bits. Due to this, N (the number of bits) is limited. In this view, one can consider addition operation to be of constant time as N is limited.

In real system, bitwise operations are executed in constant time $O(1)$ as each bit is processed in parallel. If we assume bitwise operations take linear time, then the time complexity of addition operation is $O(N^2)$ where N is the number of bits.

Therefore, it is practical to assume addition to be $O(N)$ time operation. For simplicity, you can assume addition operation to be $O(1)$ time.

Subtraction operation

Let us say we have two numbers x and y with their respective values as: $x = 5$, $y = 3$.

and we have to find out the difference between these two numbers without using any arithmetic operator like $-$ (minus).

The operators that we will utilize to achieve this task are:

1. XOR
2. Bitwise NOT
3. Left shift

For subtraction of binary numbers, rule are similar to decimal. When a larger digit is to be subtracted from a smaller digit, we take a borrow from the next column to the left.

Let us visualize the numbers in binary form:



X	Y	Diff	Borrow
1	1	0	0
0	1	1	1
1	0	1	0
0	0	0	0

If we take a look at bitwise difference, one can easily deduct that result of bitwise subtraction above can be equivalent to as if we perform xor operation on these number (x=5, y=3), and that will also leave us with the difference.

$x \wedge y$:

x	y	XOR
1	1	0
0	1	1 (with a borrow)
1	0	1
0	0	0

Now, all that is left is the borrow bit, if we perform a bitwise-not on x and then AND it with y we will be left with the borrow bits:

int x= 5 , y= 3 ;		
$\sim x$	y	AND
0	1	0
1	1	1
0	0	0
1	0	0

To summarize the above operation, subtraction will be equal to:

1. use $(x \wedge y)$ to get the difference

2. use $((\sim x) \& y)$ to get carry bit

Implementation of Logic

When it comes to implementation, we can implement this in two ways:

1. Iterative
2. Recursive

We'll take a look at both of them:

1. Iterative Algorithm

For two given integers x, y:

1. get the borrow/carry bit as it contains unset bits of x and common bits of y

int borrow = $(\sim x) \& y$;

2. get the difference using XOR and assign it to x:

x = $x \oplus y$

3. Assign the borrow to y by left shifting it by 1 so when we XOR it with x it gives the required sum.

y = borrow << 1;

4. Repeat the above steps until y becomes 0, in that case our carry will become 0.

5. In the end, you will end up having x set to the difference between x and y. In that case, simply return x

return x;

Sample C++ code:

```
#include <iostream>
using namespace std;

int subtractUsingBitwise(int x, int y)
{
    while (y != 0) // Iterate until carry becomes 0.
```

```

{
    // step 1: get the borrow bit
    int borrow = (~x) & y;

    // step 2: get the difference using XOR
    x = x ^ y;

    // step 3: left shift borrow by 1
    y = borrow << 1;
}
return x;
}

int main()
{
    int x = 5, y = 3;
    int answer = subtractUsingBitwise(x, y);
    cout << "x - y is : "<<answer<<endl;
    return 0;
}

```

2. Recursive Algorithm

for two given integers x, y:

1. Check if y is equal to 0, if so then return x (The base condition of recursion):

if (y==0) then

return x;

2. Else, assign x to the difference and y to the carry bit(left shifted by 1) in each recursive call:

x = x^y;

y = ((~x) & y) << 1;

call function recursively by passing x and y

Example C++ code

```

#include <iostream>
using namespace std;

int subtractUsingBitwise(int x, int y)

```

```

{
    if (y == 0) // step 1: base condition
        return x;

    // step 2: perform bitwise manipulation and assign x and y
    x = x^y;
    y = ((~x) & y) << 1;

    // step 3: call function recursively
    return subtractUsingBitwise(x, y);
}

int main()
{
    int x = 5, y = 3;
    cout << "x - y is " << subtract(x, y);
    return 0;
}

```

Complexity Analysis

Time complexity of bitwise subtraction will be $O(N)$

where N is the number of bits in a number.

Space complexity will remain constant, $O(1)$ as we are not using any extra space in each step.

Note: In Computing, data is of fixed size usually 32 bits or 64 bits. Due to this, N (the number of bits) is limited. In this view, one can consider subtraction operation to be of constant time as N is limited.

In real system, bitwise operations are executed in constant time $O(1)$ as each bit is processed in parallel. If we assume bitwise operations take linear time, then the time complexity of subtraction operation is $O(N^2)$ where N is the number of bits.

Therefore, it is practical to assume subtraction to be $O(N)$ time operation. For simplicity, you can assume subtraction operation to be $O(1)$ time.

Time Complexity of Multiplication

operation

Multiplication is defined as repeated addition so if addition is $O(N)$ time operation, then multiplication is $O(N^2)$ time operation.

This might seem to be simple as it is the fundamental definition of multiplication, but this is not the case.

This table summarizes how the time complexity of multiplication operation improved over the years:

Algorithm	Time Complexity	Discovered
Basic Multiplication	$O(N^2)$	100 BC
Russian Peasant Method	$O(N^2 * \log N)$	1000 AD
Karatsuba algorithm	$O(N^{1.58})$	1960
Toom Cook multiplication	$O(N^{1.46})$	1963
Schonhage Strassen algorithm	$O(N * \log N * \log \log N)$	1971
Furer's algorithm	$O(N * \log N * 2^{O(\log^* N)})$	2007
DKSS Algorithm	$O(N * \log N * 2^{O(\log^* N)})$	2008
Harvey, Hoeven, Lecerf	$O(N * \log N * 2^3_{\log^* N})$	2015
Covanov and Thomé	$O(N * \log N * 2^2_{\log^* N})$	2015
Harvey and van der Hoeven	$O(N * \log N)$	2019

It was widely believed that multiplication cannot be done in less than $O(N^2)$ time but in 1960, the first break-through came with Karatsuba Algorithm which has a time complexity of $O(N^{1.58})$.

With recent break-through, it was thought that the theoretical limit for Multiplication will be $O(N \log N)$ but it was not proved.

Finally, in 2019, an algorithm has been developed that has a time complexity

of $O(N \log N)$ for multiplication. It is a galactic algorithm which means it beats other existing algorithm only for exponentially large numbers (which are not used in practice).

Hence, we know that multiplication has a time complexity of $O(N \log N)$ while usual algorithms in practice have a time complexity of $O(N^2)$.

Time Complexity of Division operation

The Time Complexity of usual algorithm of Division operation is $O(N^2)$. The Time Complexity of Division is directly linked to Multiplication and hence, the theoretical limit of Division algorithm is $O(N \log N)$.

There are many different algorithms for Division which rely on Multiplication Algorithm used internally:

- Newton Raphson division
- Goldschmidt division

As Division relies on Multiplication, the Time Complexity of Division is the Time Complexity of the Multiplication Algorithm used internally.

Hence, theoretical limit of Division algorithm is $O(N \log N)$ where N is the number of bits.

Following is the summary of the Time Complexity of Basic Arithmetic operations:

Operation	Usual Time Complexity	Optimal Time Complexity	Assumed Time Complexity
Addition	$O(N)$	$O(N)$	$O(1)$
Subtraction	$O(N)$	$O(N)$	$O(1)$
Multiplication	$O(N^2)$	$O(N \log N)$	$O(1)$
Division	$O(N^2)$	$O(N \log N)$	$O(1)$
Note: N is the number of bits; If number is M , then $N = \log M$			

Analysis of Array

Array is a linear data structure where elements are arranged one after another. An array is denoted by a memory address M which is the memory address of the first element. In this view, the memory address of i^{th} element = $M + (i-1) * S$

where:

- M is the memory address of first element
- S is the size of each element.

Note: all elements of an array are of the same size.

Array comes in different dimensions like 2D array and 3D array. 2D arrays is an array where each element is a 1D array.

Every D-dimensional array is stored as a 1D array internally. Elements of D-dimensional array are arranged in a 1D array internally using two approaches:

- Row Major
- Column Major

In Row Major, each 1D row is placed sequentially one by one. Similarly, in Column Major, each 1D column is placed sequentially one by one. Based on this, you can find the memory address of a specific element instantly.

From our previous chapter, we learnt that fetching an element at a specific memory address takes $O(\sqrt{N})$ time where N is the block of memory being read.

Once the block of memory is in RAM (Random Access Memory) accessing a specific element takes constant time because we can calculate its relative address in constant time.

Bringing the block of memory from external device to RAM takes $O(\sqrt{N})$ time. As array elements are contiguous in memory, this operation takes place only once. Hence, it is reasonable to assume the time complexity to access an element to be $O(1)$.

Over-writing an element at a specific index takes constant time $O(1)$ because we need to access the specific index at the relative address and add new

element. This is same as accessing an element.

Note: Even in this operation, we need to load the array from external device that consumes $O(\sqrt{N})$ time.

Inserting and deleting elements take linear time depending on the implementation. If we want to insert an element at a specific index, we need to skip all elements from that index to the right by one position. This takes linear time $O(N)$.

If we want to insert element at end of array, we can do it in constant time as we can keep track of length of array as a member attribute of array. This approach is taken by standard array implementation in Java.

Similar is the approach with delete operation in array.

The Time Complexity of different operations in an array is:

Array operation	Real Time Complexity	Assumed Time Complexity
Access i^{th} element	$O(\sqrt{N})$	$O(1)$
Traverse all elements	$O(N + \sqrt{N})$	$O(N)$
Override element at i^{th} index	$O(\sqrt{N})$	$O(1)$
Insert element E	$O(N + \sqrt{N})$	$O(N)$
Delete element E	$O(N + \sqrt{N})$	$O(N)$

Analysis of Dynamic Array

Problem with Array is that it has fixed size. If user does not know then size of array or data beforehand then the solution is to use Linked List.

The problem with Linked list is the memory access is slow. Hence, Dynamic Array comes into picture which is a modified version of Array.

Dynamic Array is also called ArrayList in Java and Vector in C++.

In Dynamic array, the size of the array can be changed at time of execution of program. Basically, what we do in Dynamic Array is create a resize function and the size is adjusted according to input by user.

The key idea is both adding and deleting element will continue as if it is an ordinary array but when there is no space left in the array, the size of the array will be doubled and all existing elements will be copied to the new location.

With addition to resize function, all elements of previous array is copied to new array and then new element is appended. This new array is Dynamic Array.

The previous array memory is then dis-allocated / freed.

Normally, we make the size of array twice but that totally depend upon situation or what user wants.

Operations in a Dynamic Array

- Resize Method
- Add Method (with add at a specific index)
- Delete Method (with delete at a specific index)

Time Complexity:

When we increase the size of array by 1, then elements are copied from previous to new array and then new element is appended, it takes $O(N)$.

When we double the size of array, it will take $O(1)$ and sometimes $O(N)$. Basically, when we increase size one by one, then all elements are copied every time, again create array, and then append. When we double the size, all elements are copied only once and then appended so the average time across several insertion and deletion operations becomes $O(1)$.

Hence, the resize operation takes $O(1)$ in average and $O(N)$ in the worst case.

The logic behind insert and delete operations is the same as in Array data structure. The only extra step in Dynamic Array is resize.

If a new element needs to be inserted but the entire array is filled, then the size of the array is doubled using the resize operation.

If an element is deleted, we can halve the size of the array if possible, using the same resize operation.

Following is the summary table of Time Complexity of different operations in Dynamic Array:

Dynamic Array			
Operation	Worst Case	Average Case	Best Case
Resize	$O(N)$	$O(1)$	$O(N)$
Add	$O(1)$	$O(1)$	$O(1)$
Add at an index	$O(N)$	$O(N)$	$O(1)$
Delete	$O(1)$	$O(1)$	$O(1)$
Delete at an index	$O(N)$	$O(N)$	$O(1)$
<i>N = number of elements in array</i>			
<i>Time to read block of N elements = $O(\sqrt{N})$ not considered.</i>			

Find largest element

This is a simple yet interesting problem especially when we have explored the next few chapters.

The problem is: **Given N elements with no prior information about the elements, what is the minimum Time Complexity to find the largest element.**

Let the N elements be: $a_1, a_2, \dots, a_{N-1}, a_N$

Assume that we are checking N-1 elements, then we have no information of the Nth element which can be the largest element. So, if the Nth element is the largest element, our answer on check N-1 elements will be wrong.

So, it is necessary to check all N elements (provided we have no information about the elements).

Given 2 elements, we can find the larger element by comparing the two elements. So, this needs 1 comparison.

We can replace the comparison with a bitwise alternative, but it still results in 1 operation. So, from the point of Time Complexity, both are same.

When we have 3 elements, the key points are as follows:

1. Compare first 2 elements and find the largest element (say A)
2. The smaller element from comparison in step 1 cannot be the largest element. So, we will eliminate it from prospective elements.
3. We will compare A with the 3rd element.

This process continues for N elements. Hence, to find the largest element among N element, we need N-1 comparisons.

So, the Time Complexity of finding the largest element is **$O(N)$** .

Special cases:

- If the elements are **sorted**, then we can find the largest element in **$O(1)$ time**.

- If we know that elements are **partially sorted**, then we can find the largest element in less than $N-1$ comparisons.
- If the **range of elements is bounded**, then we can find the largest element by finding the last element from the range that is present. If the range is smaller than N , we can do this in less than $O(N)$ time.

Note:

Similarly, the **Time Complexity to find the smallest element is $O(N)$ time.**

The problem brings up a key point:

Some computational problem requires us to process each and every element to compute a specific answer. We cannot assume anything for an input we have not read. For many such problem, reading the input is same as computing the answer.

So, the most basic operation is to read the input. With no extra overhead, we can solve problems like “Finding the largest element”.

Find Second largest element

In this problem, we take our analysis of our previous problem further and find the minimum time complexity or minimum number of comparisons to find the 2nd Largest element.

In short, the Minimum Comparisons to find Second Largest Element or Second Smallest Element is $N + \log N - 2$ comparisons.

Hence, if there are 1024 elements, then we need at least $1024 + 10 - 2 = 1032$ comparisons. Also, note that if we need to find the largest or smallest element, then we need at least 1024 comparisons. Hence, there is an **increase of 8 comparisons only**.

Sub-topics covered in this chapter:

1. Problem statement
2. Divide and conquer algorithm
3. Tournament Method
4. Algorithmic approach
5. Complexity Analysis
6. Minimum number of comparisons required
7. Conclusion

Problem statement

The problem is to find the minimum number of comparisons required to find the second largest element in an unsorted array.

There can be multiple ways to find the second largest element:

1. In a simple approach we can sort the array in descending order and then return the second element which is not equal to the largest element from the sorted array.
As sorting the array requires $N \log N$ number of comparisons therefore, this approach requires **$N \log N$ number of comparisons**.
2. In another approach, we can store two variables and compare all the elements with them so that the variables will only have the values of the two largest elements.

Since we compare all the elements of the array to these two variables, therefore this approach requires **$2N$ number of comparisons**.

3. Another approach can be to traverse the array twice. In the first traversal we find the maximum element and in the second traversal, we find the greatest element less than the element obtained in the first traversal. This requires **$2N-3$ comparisons**.

Now let us discuss if it is possible to come up with a better approach requiring lesser number of comparisons.

Divide and conquer algorithm

The idea is to present you with an efficient Divide and conquer algorithm to find the second largest element and show you the number of comparisons it requires.

Note: This has the most efficient time complexity possible, but it does not make the minimum number of comparisons.

Algorithm:

Let A be the array containing n elements among which we need to find the second largest element.

Step 1: Divide and conquer the array $A[0 : n]$ into two parts, $A[0 : n/2 - 1]$ and $A[n/2:n-1]$

Step 2: Pair from $i=0$ to $n/2-1$. If $A[i] > A[i+n/2]$ then swap $A[i]$ and $A[i + n/2]$

Step 3: Until the size of the array $A[n/2:n-1]$ is greater than 2, divide and conquer recursively to find the largest value and the second largest value.

If the size of the array $A[n/2:n-1]$ is lesser than or equal to 2, then the largest value and the second largest value are calculated.

Assuming that the largest value is $A[p]$, the second largest value is $A[q]$ then,

Step 4: The largest value is $A[p]$ and the second largest value is the larger of $A[p - n/2]$ and $A[q]$.

Implementation in C++

```
#include<iostream>
using namespace std;

void secondLargest(int a[],int *largest, int
    *second,int left,int right)
{
    int i;
    if(left==right)
    {
        *largest=*second=left;
    }
    else if(left==right-1)
    {
        if(a[left]>a[right])
        {
            *largest=left;
            *second=right;
        }
        else
        {
            *largest=right;
            *second=left;
        }
    }
    else
    {
        int k=(right-left+1)/2;
        for(i=left; i<left+k; i++)
        {
            if(a[i+k] < a[i])
            {
                int temp=a[i];
                a[i]=a[i+k];
                a[i+k]=temp;
            }
        }
    }
}
```

```

        secondLargest(a, largest, second,
            left+k, right);
    if(a[*largest-k]>a[*second])
        *second=*largest-k;
    }
}

```

Time Complexity: $O(N)$

Divide and conquer approach does $3N/2 - 2$ comparisons.

So, if there are 1000 elements, we need **1498 comparisons to find the second largest element.**

Now let us see if there is any better approach that requires lesser number of comparisons.

Tournament Method

The basic idea is to compare the elements in a tournament fashion- we split the elements into pairs, compare each pair and then compare the winners of each pair in the same manner.

We start with finding the largest number in an array. We can find the largest element by comparing the elements as tuples, until only one element remains which is again the largest element in the array.

Now, to find the second largest element, we know that the second largest element wins the comparison battle against all other elements except the largest element. Which means that, the second largest element is already compared with the largest element in one of the steps and removed from the comparison tree. The second largest element must be the largest element of the list of loser array which contains all the elements that lost the comparison battle against the largest element and has $\log_2(N)$ elements. Thus, by finding the largest element of this array using the same approach we get the second largest element.

Now let us discuss how to implement this tournament method to find the second largest element.

Algorithmic approach

We start with finding the largest number in an array. We can find the largest element by comparing the elements as tuples, until only one element remains which is again the largest element in the array.

We can find the second largest element as follows:

- Find the largest element.
- Collect all elements of the array that were directly compared to the largest element.
- Find the largest element among them.

While finding the largest element, we also store all the elements, the largest element was compared to in an array `Compared[]`. Every time any element loses in a comparison to the largest element, we store it in this array. Second largest element is one of the elements of this array as it wins over all other elements but loses to the largest element and then find the largest among these elements.

Algorithm:

We design this algorithm in two steps.

Step 1: Here we use `FindMaxTournament()` to find the largest element in the given array.

`FindMaxTournament()` algorithm returns an array of integers `Compared[0..K]` where

- `Compared[0] = K` (i.e., the 0th element of the array holds the length of the array);
- `Compared[1] = max` (i.e., the first element of the array is the largest number;
- `Compared[2], . . . , Compared[K]` are all numbers with which the largest

number has

been compared till now.

Step 2: Here we use FindSecondMax() to find the second largest element in an array, and function FindMaxTournament() is used in it.

First call to the FindMaxTournament() returns the compared array while finding the largest element and the second call to the FindMaxTournament() returns the compared array while finding the second largest element and finally the FindSecondMax() function returns the second largest element.

```
Algorithm FindSecondMax(N, A[1..N]) returns
begin
    Compared ← FindMaxTournament(1,N,A[1..N]);
    Compared2 ← FindMaxTournament(2, Compared[0],
        Compared[2..Compared[0]]);
    return Compared2[1]
end
```

```
Function FindMaxTournament(I,J, A[I..J],N)
    returns Compared[0..K]
begin
    if I = J then //base case
        Compared[0..N];
        Compared[0] ← 1;
        Compared[1] ← A[I];
        return Compared;
    endif
    Compared1 ← FindMaxTournament(I, I+(J-I)/2, A,N);
    Compared2 ← FindMaxTournament(1+I+(J-I)/2,J, A,N);
    if Compared1[1]>Compared2[1] then
        K ← Compared1[0]+1;
        Compared1[0] ← K;
        Compared1[K] ← Compared2[1];
        return Compared1;
    else
        K ← Compared2[0]+1;
        Compared2[0] ← K;
        Compared2[K] ← Compared1[1];
        return Compared2;
    endif
end
```

end

Complexity Analysis

From the above algorithm, we know that the FindMaxTournament() algorithm, has a running time $O(n)$ for an array of size n , therefore

- The first call to FindMaxTournament() has running time $O(n)$.
- The second call to FindMaxTournament() passes an array of size at most $\log n$ and therefore, runs in $O(\log n)$.

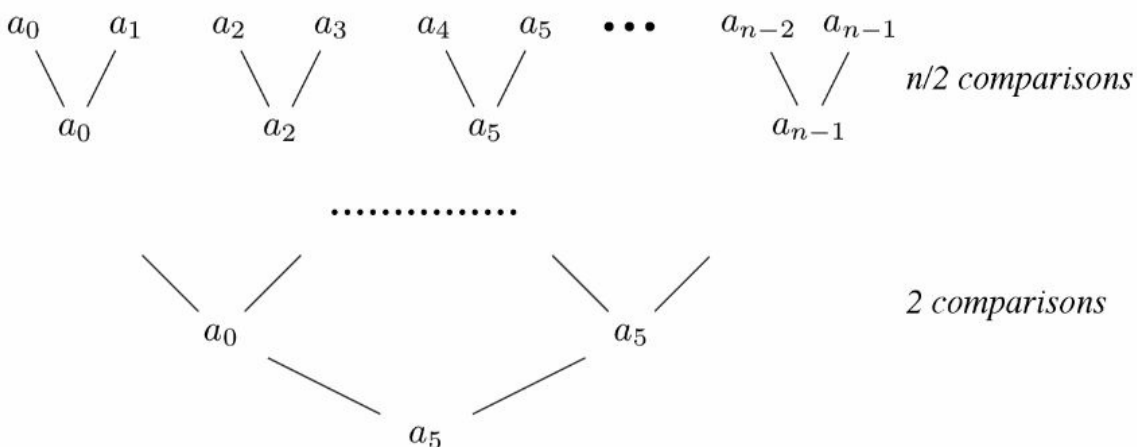
Hence FindSecondMax() has running time $O(N) + O(\log(N)) = O(N)$.

Also, an extra compared array of size k is used and k is $\log N$, for N elements, therefore the space complexity is $O(\log N)$.

Thus, the total time complexity of the algorithm is $O(N)$ and the total space complexity is $O(\log N)$.

Minimum number of comparisons required

When we find the largest element by comparing the elements as tuples, until only one element remains which is again the largest element in the array, then



If the number of elements in the array is a power of 2, ($n = 2^k$) then, to find

the largest element, $n/2$ comparisons must be performed until only one element remains. So, the number of comparisons are:

$$n/2 + n/4 + n/8 + \dots + n/(n/2) + n/n = n - 1$$

So, to find the largest element it takes $N-1$ comparisons. We can also state this from the algorithm that the first call to `FindMaxTournament()` (to find the largest element) passes an array of size N and hence uses $N-1$ comparisons.

Also, as discussed earlier, the second largest element wins the comparison battle against all other elements except the largest element and thus the second largest element must be the largest element of the list of losers array which has $\log N$ elements. Therefore, the second call to `FindMaxTournament()` (to find the second largest element) passes an array of size at most $\log n$ and therefore, it uses $\log N - 1$ comparisons.

As `FindSecondMax()` uses $N - 1 + \log N - 1$ comparisons i.e $N + \log N - 2$ comparisons. Therefore, to find the second largest element, minimum $N + \log N - 2$ comparisons are required.

Conclusion

The time complexity of both the discussed methods is same i.e $O(N)$ but the second method uses lesser number of comparisons.

Therefore, we conclude that the minimum number of comparisons required to find the second largest element is $N + \log N - 2$.

Question

What are the minimum number of comparisons required to find the second largest element in the array `arr={-8,-13,0,-21,-2,18,57,91}`?

Answer: **9**

Find i^{th} largest element

This problem of finding the minimum number of comparisons to find the i^{th} largest element is a critical problem and was one of the pillars to make Computing a field of Science.

This was introduced in the paper titled "Time bounds for selection" by Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronal L. Rivest, and Robert E. Tarjan in 1973. This paper proposed a new algorithm called PICK for the problem that is stated as "Given an array of N integers, we have to pick the i^{th} smallest number". The algorithm demonstrates to solve in no more than **$5.4305 * N$ comparisons**.

Background

The problem can be traced back to the world of sports and the design of tournament to select first and second-best player. The first best player will obviously be the one who won the entire game, however the previous method was denounced where the player who loses the final match was the second best player. Around 1930, this problem came into the realm of algorithmic complexity. It was shown that no more than $N + \lceil \log(N) \rceil - 2$ matches are required, this method conducted a second match for the $\log(N)$ (at most) players other than the winner to determine the second best player.

There are certain notations that we need to understand here.

- i^{th} of S : S is a set of numbers and i^{th} of S denotes the i^{th} smallest element in S .
- rank of x : rank of an element in an array can be defined as the number of elements in S , that are lesser than x .

Also, x has to be such that

$y = \text{rank of } x$
 y^{th} of S has to be x (y^{th} of $S = x$)

See the below figure for better understanding.

$i \theta S \stackrel{\text{def}}{=} (\text{read “}i\text{-th of } S\text{”})$ the i -th smallest element of S , for $1 \leq i \leq |S|$.
 Note that the magnitude of $i \theta S$ increases as i increases. We shall often denote $i \theta S$ by $i \theta$ when S is understood.

$x \rho S \stackrel{\text{def}}{=} (\text{read “}x\text{’s rank in } S\text{”})$ the rank of x in S , so that
 $x \rho S \theta S = x$.

The minimum worst-case cost, that is, the number of binary comparisons required, to select i^{th} of S will be denoted by $f(i, n)$, $|S|=n$, and a new notation is introduced to measure the relative difficulty of computing percentile levels.

$$F(\alpha) \stackrel{\text{def}}{=} \limsup_{n \rightarrow \infty} \frac{f(\lfloor \alpha(n-1) \rfloor + 1, n)}{n}, \quad \text{for } 0 \leq \alpha \leq 1$$

PICK

Pick operates by successively removing subsets of S whose elements are known to be too small or too large to the $i \theta S$, until only i^{th} of S remains. Each subset removed with contain at least one quarter of the remaining elements. PICK will be described in terms of three auxiliary functions $b(i,n)$, $c(i,n)$, $d(i,n)$. In order to avoid confusion, we will omit argument lists for these functions.

Let us see the algorithm

Select m which belongs to S

a) Arrange S into n/c columns of length c and sort each column

b) Select m such that it is the b^{th} of T , where T contains elements from the set whose size is n/c and are the d^{th} smallest element from each column. Use PICK recursively if $n/c > 1$

Compute rank of m

Compare m to every other element x in S for which it is not known whether $m > x$ or $m < x$.

If rank of m is i , then we have got our answer, since m is the i^{th} of S .

If rank of $m > i$, we can disregard/remove D = set of elements greater than m , and n becomes n -number of elements present in D .

If rank of $m < i$, then we can disregard/remove D = set of elements lesser than m , and n becomes n -number of elements present in D .

Go back to Step 1.

Pseudo code for above algorithm would look like:

```
PICK(a[], n, i)
if(n<1) stop
let k=n/c
divide a[] into k pairs
form another pair with dth smallest element from each pair.
m = rank(d)
if m=i, halt
else if(m>i) discard set D = {x>=i}, where x belongs to a
else discard set D = {x<=i}
repeat recursively till k>1
```

Proving that $f(n,i) = O(n)$

The functions $b()$, $c()$ and $d()$ decide the time complexity. We need to choose them wisely as an additional sorting is also done.

Let $h(c)$ denote cost of sorting c elements using Ford and Johnson's algorithm (also known as the merge-insertion sort algorithm) which is a comparison sorting algorithm that involves fewer comparisons in the worst case as compared to insertion and merge sorts. The algorithm involves three steps, let N be the size of array to be sorted.

Split the array into $N/2$ pairs of two elements and order them pairwise, if N is odd, then the last element isn't paired with any element.

The pairs are recursively sorted by their highest value. Again, in case of n being odd, the last element is not considered highest value and left at the end of collection. Now all the highest elements form a list, let's call it main list $(a_1, a_2, \dots, a_{n/2})$ and the remaining ones are the pend list $(b_1, b_2, \dots, b_{n/2})$, for any

given i , $b_i \leq a_i$.

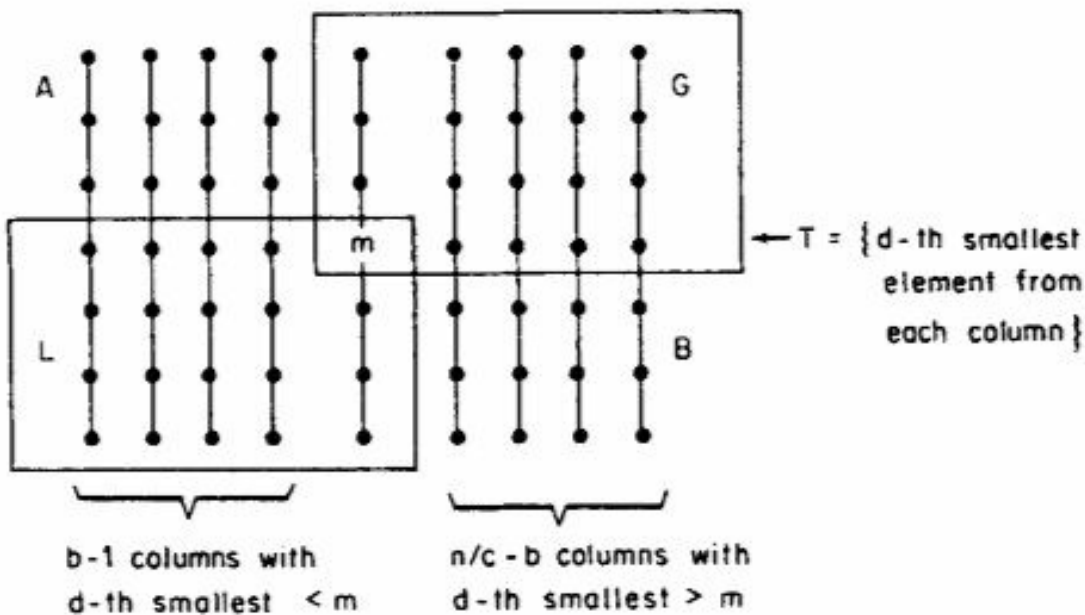
Insert pending elements into the main list, we know that $b_1 \leq a_1$, hence $\{b_1, a_1, a_2, \dots, a_{n/2}\}$. Now the remaining elements should be inserted with the constraint that the size of insertion is $2^N - 1$. This process should be repeated till all the elements are inserted.

The cost of sorting elements is given below

$$h(c) = \sum_{1 \leq j \leq c} \lceil \log_2(3j/4) \rceil.$$

Now the cost of 1a previously described is $N * h(c)/c$. Hence, it's only wise to make c a constant in order for the algorithm to run in linear time.

Let $P(N)$ be the maximum cost for PICK for any i . We can bound the cost of 1b by $P(n/c)$. After 1b, a partial order is formed which can be seen in the below figure.



Since the recursive call to PICK in step 1(b) determines which elements of T are $< m$, and which are $> m$, we separate the columns, where every element of box G is clearly greater than m , while every element in box L is less.

Therefore, only those elements in quadrants A and B need to be compared to m in step 2.

We can easily show that no elements are removed incorrectly in step 3. if rank of m > i, we can say that m is too large and remove all elements greater than m and vice versa. Also, note that at the least all of L or G is removed.

Therefore,

$$P(n) \leq n \cdot h(c)/c + P(n/c) + n \\ + P(n - \min(|L|, |G|))$$

The values of b, c, d decide much of the computation cost. To minimise the above equation we can assume some values and proceed further.

Let $c=21$

$d=11$

$b=n/2 \cdot c \Rightarrow n/42$

By substituting the above values in the equation

$$P(n) \leq 66n/21 + P(n/21) + n + P(31n/42)$$

$$P(n) \leq 58 \cdot n/3 = 19.6n$$

The basis for the induction is that since $h(N) < 19N$ for $N < 105$, any small case can be handled by sorting. PICK runs in linear time because a significant fraction of S is discarded on each pass, at a cost proportional to the number of elements discarded on each step. Note that we must have $c \geq 5$ for PICK to run in linear time.

Values of b, c, d

A reasonable choice for these functions can help the algorithm achieve a linear time. From the above calculations it has been proven that c must be constant since the cost of sorting depends on it. Also, it must be greater than or equal to 5.

We have defined P(N) as maximum cost of PICK for any i, therefore we chose c and d to be constants and b as a function of c, as b decides the value of m (recall 1b) and therefore it would make sense to choose b in terms of c.

Improvements to PICK

Two modifications to PICK are:

PICK1 which yields best overall bound for $F(\alpha)$

PICK2 which is more efficient for i in the ranges of $i < \beta n$ or $i > (1-\beta)n$ for $\beta = 0.203688$.

PICK1 differs by PICK in:

- The elements of S are sorted into columns only once, after which those columns broken by the discard operation are restored to full length by a (new) merge step at the end of each pass.
- The partitioning step is modified so that the number of comparisons used is a linear function of the number of elements eventually discarded.
- The discard operation breaks no more than half the columns on each pass, allowing the other modifications to work well.
- The sorting step implicit in the recursive call to select m is partially replaced by a merge step for the second and subsequent iterations since (3) implies that $1/2$ of the set T operated on at pass j were also in the recursive call at pass $j-1$.

The procedure of PICK1 is relatively lengthy. The optimized algorithm is full of red tape, in principle, for any particular n could be expanded into a decision tree without red-tape computation.

PICK2

By analysis of PICK2, which is essentially the same as PICK with the functions $b(i, n)$, $c(i, n)$, and $d(i, n)$ chosen to be i , 2 , and 1 , respectively, and with the partitioning step eliminated.

Time Complexity

The main result i.e time complexity of PICK, $f(i, n) = O(n)$. However, this has been tuned up to provide tighter results. If you recall the definition of $F(\alpha)$ to measure relative difficulty of computing percentile levels. After the changes made to PICK, we get three expressions here

$$\max_{0 \leq \alpha \leq 1} F(\alpha) \leq 5.4305 \quad (1)$$

and

$$F(\alpha) \leq 1 + 4.4305\alpha/\beta + 10.861[\log_2(\beta/\alpha)]\alpha, \quad \text{for } 0 < \alpha \leq \beta, \quad (2)$$

where $\beta = 0.203688^-$. In Sec. 4 we derive the lower bound:

$$F(\alpha) \geq 1 + \min(\alpha, 1 - \alpha), \quad \text{for } 0 \leq \alpha \leq 1. \quad (3)$$

There is no evidence to suggest that any of the inequalities (1) - (3) is the best possible. In fact, the authors surmise that they can be improved considerably.

Conclusion

The most important result of this paper is that selection can be performed in linear time, in the worst case. No more than **5.4305N comparisons** are required to select the i^{th} smallest of n numbers, for any i , $1 \leq i \leq n$. This bound can be improved when i is near the ends of its range, i.e when i get larger.

Time Complexity Bound for comparison-based sorting

In this chapter, we have explained the mathematical analysis of time complexity bound for comparison-based sorting algorithm. The time complexity bound is $O(N \log N)$ but for non-comparison-based sorting, the bound is $O(N)$.

Table of contents:

1. Analysis of Time Complexity Bound of Sorting
2. Lower bound for the worst case
3. Lower bound for the average case
4. Lower bound for randomized algorithms

Analysis of Time Complexity Bound of Sorting

In comparison sorts, all the determined sorted order is based only on comparisons between the input elements. Merge sort, quicksort and insertion sort are some examples of comparison-based sorting algorithm. The lower bound of the comparison-based sorting algorithm is $N \log_2 N$.

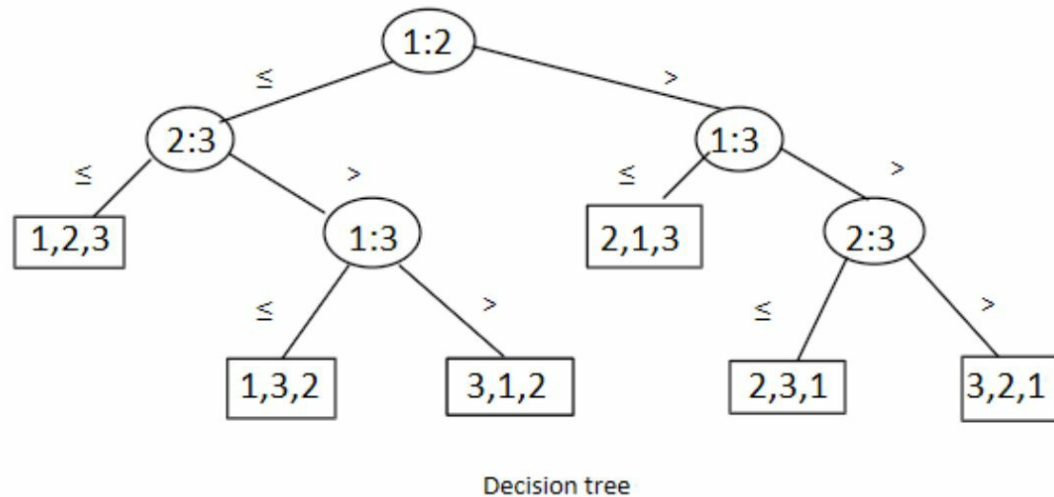
Suppose we have an input sequence $\{a_1, a_2, \dots, a_n\}$ and assume that all inputs are distinct.

We may perform comparison $a_i \leq a_j$ & $a_i > a_j$ between two elements a_i and a_j . Here, which pair will make a comparison depends only on the results of the comparisons made so far.

For comparison between N elements, we can be viewed it as a binary tree where each node represents a comparison and leaf node gives the permutation of the N elements. This binary tree is also known as **decision tree**. A decision tree can be used for any comparison-based sorting algorithm.

Here, each internal node label $i:j$ indicates a comparison between a_i and a_j . For any comparison-based sorting algorithm, if we ignore detail operation and only consider the comparison between the pair of elements, then the

input sequence follows a path in the decision tree from root to leaf.



Lower bound for the worst case

The worst-case number of comparisons for comparison-based sorting algorithm is equal to the **longest path of the decision tree from the root to leaf**.

Let us consider a decision tree of height H with I reachable leaves. There are N inputs with $N!$ possible outputs. Each of the permutations of n elements appears on leaves. A binary tree of height H has maximum 2^H leaves. Path having the maximum height represents the worst case. The decision tree might not be a complete binary tree so it can have less than 2^H leaves.

$$N! \leq I \leq 2^H$$

$$\begin{aligned} \log(N!) &= \log(1) + \log(2) + \dots + \log(N/2) \\ &\quad + \dots + \log(N) \\ &= \log(1) + \dots + \log(N/2) + \dots + \log(N) \geq \\ &\quad \log(N/2) + \log(N/2 + 1) + \dots + \log(N) \\ &\geq \log(N/2) + \log(N/2) + \dots + \log(N/2) \\ &\geq (N/2) * \log(N/2) \\ &= N \log N \end{aligned}$$

By taking logarithms,

$$H \geq \log(N!)$$

$$H = \Omega(N \log N)$$

The lower bound for the worst-case analysis is $\Omega(N \log N)$. This bound is asymptotically tight.

Lower bound for the average case

Suppose we have N distinct elements. So, the decision tree has at least $N!$ leaves and each comparison will give either a_i, a_j or a_j, a_i . If the tree is completely balanced, then depth of each leaf is $\lfloor \log_2 N! \rfloor$ or $\lceil \log_2 N! \rceil$. Let us consider D is a decision tree and $H(D, l)$ is the sum of heights for all l leaves.

$H(l)$ represents the minimum sum of heights.

Decision tree D contains two subtrees. Let the number of leaves in the each subtree be m_1 & m_2 , respectively.

$$m_1, m_2 < l$$

$$m_1 + m_2 = l \text{ \& } m_1 = m_2 = l/2$$

$$H(D, l) = l + H(D_1, m_1) + H(D_2, m_2)$$

$$H(l) = l + \min \{H(m_1) + H(m_2)\}$$

By the assumption of induction hypothesis:

$$H(l) \leq l + \min\{m_1 \log m_1 + m_2 \log m_2\}$$

$$H(l) \leq l + (l/2) \log(l/2) + (l/2) \log(l/2)$$

Average height of the decision tree is -

$$H(N!) / N! \leq \log(N!)$$

$$= N \log N$$

The lower bound for the average case analysis is $\Omega(N \log N)$.

Lower bound for randomized algorithms

Let a randomized algorithm A have a probability distribution over

deterministic algorithms.

The running time of the randomized algorithm is a random variable and the sequences S correspond to the elementary events. The probability distribution over all the deterministic algorithms is A_S . On some input I , the expected number of comparisons made by randomized algorithm A is –

$$\sum_S \Pr(s) (\text{Running of } A_S \text{ on } I)$$

The expected running time of the randomized algorithm is equal to the average over deterministic algorithms. The running time of the deterministic algorithm is $\lfloor \log_2 N! \rfloor$. Average case running time of the randomized algorithm is:

$$\begin{aligned} \text{avg} \sum_S \Pr(s) (\text{Running of } A_S \text{ on } I) &= \sum_S \text{avg}[\Pr(s) (\text{Running of } A_S \text{ on } I)] \\ &= \sum_S \Pr(s) \text{avg}(\text{Running of } A_S \text{ on } I) \\ &\geq \sum_S \Pr(s) \lfloor \log_2 n! \rfloor \\ &= \lfloor \log_2 n! \rfloor \end{aligned}$$

Any randomized comparison-based sorting also takes $\Omega(N \log N)$ steps.

With this mathematical analysis, the key points are:

- The worst case of comparison-based sorting algorithm involves $O(N \log N)$ time.
- The average case of comparison-based sorting algorithm involves $O(N \log N)$ time.
- This means the best case can be better that is $O(N)$.
- Irrespective of algorithm, if the steps involve comparisons, the above bounds are applicable.

Analysis of Selection Sort

Selection sort is a basic comparison-based sorting algorithm. For our previous analysis, we know the lower bounds we are dealing with but each algorithm has different limit (\geq lower limit) on different cases.

We will learn about Time Complexity and Space Complexity of Selection Sort algorithm along with the complete mathematical analysis of the different cases.

In short:

- Worst Case Time Complexity: $O(N^2)$
- Average Case Time Complexity: $O(N^2)$
- Best Case Time Complexity: $O(N^2)$
- Space Complexity: $O(1)$

The number of swaps in Selection Sort are as follows:

- Worst case: $O(N)$
- Average Case: $O(N)$
- Best Case: $O(1)$

We have covered the following in this chapter:

1. Overview of Selection Sort Algorithm
2. Analysis of Worst Case Time Complexity of Selection Sort
3. Analysis of Average Case Time Complexity of Selection Sort
4. Analysis of Best Case Time Complexity of Selection Sort
5. Analysis of Space Complexity of Selection Sort
6. Conclusion

Overview of Selection Sort Algorithm

The basic idea of Selection Sort is to find the smallest element in the unsorted array and add it to the front of the array. In Selection Sort, we maintain two

parts:

- Sorted sub-array
- Unsorted sub-array

In sorted sub-array, all elements are in sorted order and are less than all elements of the unsorted sub-array.

The pseudocode of Selection Sort is as follows:

```
// Selection Sort
array = [a1, a2, ..., aN]

for i from 0 to N-1
    for j from i to N-1
        int smallest_index = smallest(array[j ... N-1])
        if (smallest_index != i)
            swap(i, smallest_index)

// Find the smallest element
smallest(int array):
    smallest_index = 0
    smallest = array[0]
    for index from 1 to M
        if smallest > array[index]
            smallest = array[index]
            smallest_index = index

    return smallest_index
```

With this, you have the basic idea of selection sort.

Time Complexity Analysis of Selection Sort

At the beginning, the size of sorted sub-array (say S1) is 0 and the size of unsorted sub-array (say S2) is N.

At each step, the size of sorted sub-array increases by 1 and size of unsorted sub-array decreases by 1. Hence, for a few steps are as follows:

Step 1: S1: 0, S2: N

Step 2: S1: 1, S2: N-1

Step 3: S1: 2, S2: N-2

and so on till $S1 = N$. Hence, there will be $N+1$ steps.

Hence, $S2 = N - S1$

The Time Complexity of finding the smallest element in a list of M elements is $O(M)$. This is constant for all worst case, average case, and best case.

The time required for finding the smallest element is the size of unsorted sub-array that is $O(S2)$. The exact value of $S2$ is dependent of the step.

For step I , $S1$ will be $I-1$ and $S2$ will be $N-S1 = N-I+1$.

So, the time complexity for step I will be:

$O(N-I+1)$ for find the smallest element

$O(1)$ for swapping the smallest element to the front of unsorted sub-array

I will range from 1 to $N+1$.

Hence, the sum of time complexity of all operations will be as follows:

Sum [$O(N-I+1) + O(1)$] for I from 1 to $N+1$

= Sum [$O(N-I+1)$] + Sum[$O(1)$] ... Equation 1

Sum [$O(1)$] = $1 + 1 + \dots + 1$ [($N+1$) times] = $N+1 = O(N)$

Sum [$O(N-I+1)$] = $N + (N-1) + \dots + 1 + 0 =$

= $1 + 2 + \dots + N$

= $N * (N+1) / 2$

= $(N^2 + N) / 2 = O(N^2) + O(N) = O(N^2)$ [as N^2 is dominant term]

Therefore, from Equation 1, we get:

Sum [$O(N-I+1)$] + Sum[$O(1)$]

$$= O(N^2) + O(N)$$

$$= O(N^2)$$

Hence, the time complexity of Selection Sort is $O(N^2)$.

Worst Case Time Complexity of Selection Sort

The worst case is the case when the array is already sorted (with one swap) but the smallest element is the last element. For example, if the sorted number as a_1, a_2, \dots, a_N , then:

$a_2, a_3, \dots, a_N, a_1$ will be the worst case for our particular implementation of Selection Sort.

Worst Case: $a_2, a_3, \dots, a_N, a_1$

The cost in this case is that at each step, a swap is done. This is because the smallest element will always be the last element and the swapped element which is kept at the end will be the second smallest element that is the smallest element of the new unsorted sub-array. Hence, the worst case has:

- $N * (N+1) / 2$ comparisons
- N swaps

Hence, the time complexity is $O(N^2)$.

Best Case Time Complexity of Selection Sort

The best case is the case when the array is already sorted. For example, if the sorted number as a_1, a_2, \dots, a_N , then:

$a_1, a_2, a_3, \dots, a_N$ will be the best case for our particular implementation of Selection Sort.

This is the best case as we can avoid the swap at each step, but the time spend

to find the smallest element is still $O(N)$. Hence, the best case has:

- $N * (N+1) / 2$ comparisons
- 0 swaps

Note only the number of swaps has changed. Hence, the time complexity is $O(N^2)$.

Average Case Time Complexity of Selection Sort

Based on the worst case and best case, we know that the number of comparisons will be the same for every case and hence, for average case as well, the number of comparisons will be constant.

Number of comparisons = $N * (N+1) / 2$

Therefore, the time complexity will be $O(N^2)$.

To find the number of swaps,

There are $N!$ different combination of N elements

Only for one combination (sorted order) there is 0 swaps.

In the worst case, a combination will have N swaps. There are several such combinations.

Number of ways to select 2 elements to swap = ${}_nC^2 = N * (N-1) / 2$

From sorted array, this will result in $O(N^2)$ combinations which need 1 swap.

So,

0 swap = 1 combination

1 swap = $O(N^2)$ combinations

2 swaps = $O(N^4)$ combinations

...

N swaps = $O(N)$ combinations

Hence, the total number of swaps will be:

$$0 + O(N^2) + 2 * O(N^4) + \dots + N * O(N) \\ = O((N+1)!)$$

Hence, the average number of swaps will be N that is $O((N+1)!)/O(N!)$.

Hence, the average case has:

- $N * (N+1) / 2$ comparisons
- N swaps

Space Complexity of Selection Sort

The space complexity of Selection Sort is $O(1)$.

This is because we use only constant extra space such as:

- 2 variables to enable swapping of elements.
- One variable to keep track of smallest element in unsorted array.

Hence, in terms of Space Complexity, Selection Sort is optimal as the memory requirements remain same for every input.

Conclusion

As a summary:

- Worst Case Time Complexity: $O(N^2)$
- Average Case Time Complexity: $O(N^2)$
- Best Case Time Complexity: $O(N^2)$
- Space Complexity: $O(1)$

The number of swaps in Selection Sort are as follows:

- Worst case: $O(N)$
- Average Case: $O(N)$
- Best Case: $O(1)$

Analysis of Insertion Sort

Insertion sort is another comparison-based sorting algorithm which is the preferred choice for small datasets and when the data is almost sorted.

We will explore time and space complexity of Insertion Sort along with two optimizations. Before going into the complexity analysis, we will go through the basic knowledge of Insertion Sort.

In short:

- The worst-case time complexity of Insertion sort is $O(N^2)$
- The average-case time complexity of Insertion sort is $O(N^2)$
- The time complexity of the best-case is $O(N)$.
- The space complexity is $O(1)$

What is Insertion Sort?

Insertion sort is one of the intuitive sorting algorithms for beginners which shares analogy with the way we sort cards in our hand.

As the name suggests, it is based on "insertion" but how?

An array is divided into two sub arrays namely sorted and unsorted subarray.

How come there is a sorted subarray if our input is unsorted?

The algorithm is based on one assumption that a single element is always sorted.

Hence, the first element of array forms the sorted subarray while the rest create the unsorted subarray from which we choose an element one by one and "insert" the same in the sorted subarray. The same procedure is followed until we reach the end of the array.

In each iteration, we extend the sorted subarray while shrinking the unsorted subarray.

Working Principle

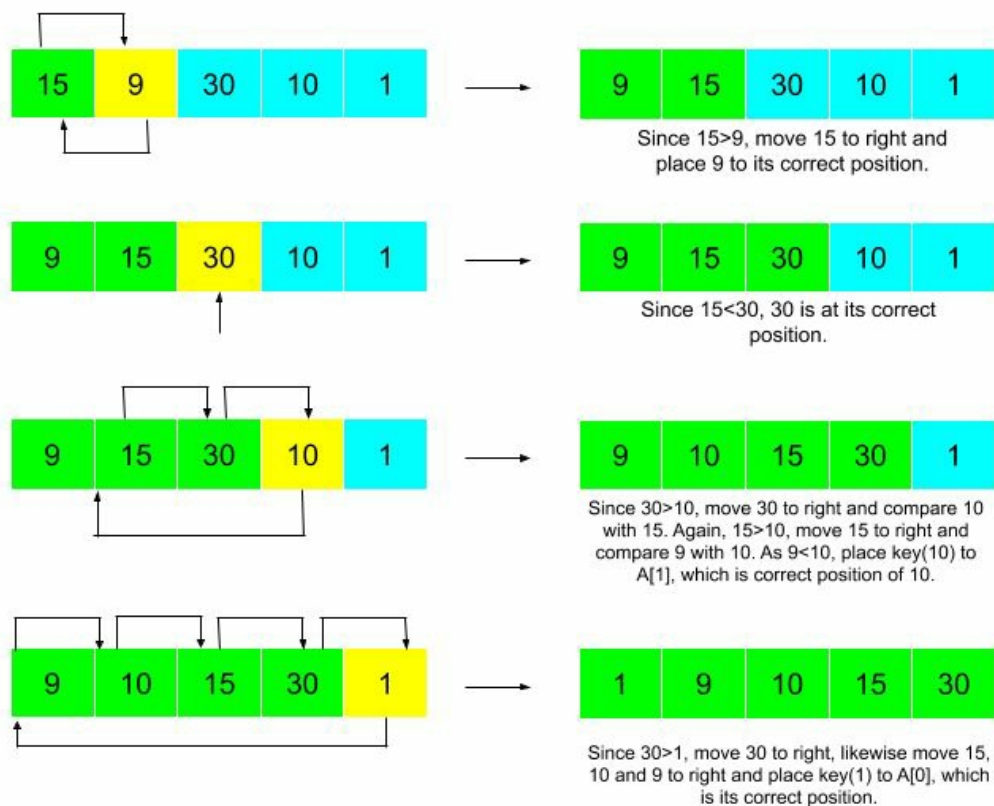
1. Compare the element with its adjacent element.
2. If at every comparison, we could find a position in sorted array where the element can be inserted, then create space by shifting the elements to right and insert the element at the appropriate position.
3. Repeat the above steps until you place the last element of unsorted array to its correct position.

Let us take an example.

Input: 15, 9, 30, 10, 1

Expected Output: 1, 9, 10, 15, 30

The steps could be visualized as:



Pseudocode

1. for $i = 0$ to n
2. $key = A[i]$
3. $j = i - 1$

```
4.   while j >= 0 and A[j] > key
5.       A[j + 1] = A[j]
6.       j = j - 1
7.   end while
8.   A[j + 1] = key
9. end for
```

Complexity

Worst case time complexity: $\Theta(N^2)$

Average case time complexity: $\Theta(N^2)$

Best case time complexity: $\Theta(N)$

Space complexity: $\Theta(1)$

Implementation

```
// C++ program for insertion sort
#include <bits/stdc++.h>
using namespace std;
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, k, j;
    for (i = 1; i < n; i++)
    {
        k = arr[i];
        j = i - 1;
        /*shifting elements of arr[0..i-1] towards right
           if are greater than k */
        while (j >= 0 && arr[j] > k)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = k;
    }
}
int main()
{
    int n,i;
```

```

// size of array
cin>>n;
int arr[n];
for (i = 0; i < n; i++)
    cin >> arr[i];
// input the array
insertionSort(arr, n);
// sort the array
for (i = 0; i < n; i++)
    cout << arr[i] << " ";
cout << endl;
// print the array
return 0;
}

```

Complexity Analysis for Insertion Sort

We examine Algorithms broadly on two prime factors, i.e.,

- Running Time
- Memory usage

Running Time of an algorithm is execution time of each line of algorithm.

As stated, Running Time for any algorithm depends on the number of operations executed. We could see in the Pseudocode that there are precisely 7 operations under this algorithm. So, our task is to find the Cost or Time Complexity of each and trivially sum of these will be the Total Time Complexity of our Algorithm.

We assume Cost of each i operation as C_i where $i \in \{1, 2, 3, 4, 5, 6, 7\}$ and compute the number of times these are executed. Therefore, the Total Cost for one such operation would be the product of Cost of one operation and the number of times it is executed.

We could list them as below:

COST OF LINE	NO. OF TIMES IT IS RUN
C_1	n

Then Total Running Time of Insertion sort ($T(n)$) = $C_1 * n + (C_2 + C_3) * (n - 1) +$

C_2	$n - 1$	$C_4 * \sum_{j=1}^{n-1} (t_j) + (C_5 + C_6) * \sum_{j=1}^{n-1} 1 + C_8 * (n - 1)$
C_3	$n - 1$	
C_4	$\sum_{j=1}^{n-1} (t_j)$	
C_5	$\sum_{j=1}^{n-1} (t_{j-1})$	Memory usage: Memory required to execute the Algorithm. We didn't require any extra
C_6	$\sum_{j=1}^{n-1} (t_{j-1})$	
C_8	$n - 1$	

space. We are only re-arranging the input array to achieve the desired output. Hence, we can claim that there is no need of any auxiliary memory to run this Algorithm. Although each of these operations will be added to the stack but not simultaneously the Memory Complexity comes out to be $O(1)$.

Best Case Analysis

In Best Case i.e., when the array is already sorted, $t_j = 1$

Therefore, $T(n) = C_1 * n + (C_2 + C_3) * (n - 1) + C_4 * (n - 1) + (C_5 + C_6) * (n - 2) + C_8 * (n - 1)$

which when further simplified has dominating factor of n and gives $T(N) = C * (N)$ or $O(N)$

Worst Case Analysis

In Worst Case i.e., when the array is reversely sorted (in descending order), $t_j = j$

Therefore, $T(N) = C_1 * n + (C_2 + C_3) * (n - 1) + C_4 * (n - 1) * (n) / 2 + (C_5 + C_6) * ((n - 1) * (n) / 2 - 1) + C_8 * (n - 1)$

which when further simplified has dominating factor of N^2 and gives $T(N) = C * (N^2)$ or $O(N^2)$.

Average Case Analysis

Let us assume that $t_j = (j-1)/2$ to calculate the average case

Therefore, $T(N) = C1 * n + (C2 + C3) * (n - 1) + C4/2 * (n - 1)(n) / 2 + (C5 + C6)/2 * ((n - 1)(n) / 2 - 1) + C8 * (n - 1)$

which when further simplified has dominating factor of N^2 and gives $T(N) = C * (N^2)$ or $O(N^2)$

Can we optimize it further?

Searching for the correct position of an element and Swapping are two main operations included in the Algorithm.

We can optimize the searching by using Binary Search, which will improve the searching complexity from $O(N)$ to $O(\log N)$ for one element and to $N * O(\log N)$ or $O(N \log N)$ for n elements.

But since it will take $O(n)$ for one element to be placed at its correct position, n elements will take $N * O(N)$ or $O(N^2)$ time for being placed at their right places. Hence, the overall complexity remains $O(N^2)$.

We can optimize the swapping by using Doubly Linked list instead of array, that will improve the complexity of swapping from $O(N)$ to $O(1)$ as we can insert an element in a linked list by changing pointers (without shifting the rest of elements).

But since the complexity to search remains $O(N^2)$ as we cannot use binary search in linked list. Hence, the overall complexity remains $O(N^2)$.

Therefore, we can conclude that we cannot reduce the worst-case time complexity of insertion sort from $O(N^2)$.

Advantages

- Simple and easy to understand implementation
- Efficient for small data
- If the input list is sorted beforehand (partially) then insertion sort takes $O(N+D)$ time where D is the number of inversions.

- Chosen over bubble sort and selection sort, although all have worst case time complexity as $O(N^2)$
- Maintains relative order of the input data in case of two equal values (stable).
- It requires only a constant amount $O(1)$ of additional memory space (in-place Algorithm).

Applications

- It could be used in sorting small lists.
- It could be used in sorting "almost sorted" lists.
- It could be used to sort smaller sub problem in Quick Sort.

Analysis of Bubble Sort

Bubble Sort is another comparison-based sorting algorithm.

we will explore the time and space complexity of Bubble Sort and also, explore an optimization of this algorithm.

Sub-topics:

1. Introduction to Bubble sort
2. Time Complexity Analysis
3. Worst Case Time Complexity
4. Best Case Time Complexity
5. Average Case Time Complexity
6. Space Complexity
7. Comparison with other Sorting Algorithms

Introduction to Bubble sort

Bubble sort is an algorithm that sequentially steps through a list of items and swaps items if they are not in the correct order till the list is sorted.

Pseudocode:

```
1. procedure bubbleSort(A : list of sortable items)
2.   n := length(A)
3.   for i := 0 to n-1 inclusive do
4.     for j := 0 to n-i-1 inclusive do
5.       // the elements aren't in the right order
6.       if A[j] > A[j+1] then
7.         // swap the elements
8.         swap(A[j], A[j+1])
9.       end if
10.    end for
11.  end for
12. end procedure
```

Time Complexity Analysis

In this unoptimized version, the time complexity is $\Theta(N^2)$. This applies to all

the cases including the worst, best and average cases because even if the array is already sorted, the algorithm does not check that at any point and runs through all iterations. Although the number of swaps would differ in each case.

The number of times the inner loop runs

$$= \sum_{j=1}^{N-i-1} 1 \quad (2)$$

$$= (N - i) \quad (1)$$

The number of times the outer loop runs

$$= \sum_{i=1}^{N-1} \sum_{j=1}^{N-i-1} 1 = \sum_{i=1}^{N-1} N - i \quad (3)$$

$$= \frac{N*(N-1)}{2} \quad (4)$$

Optimization

We add an additional check that breaks out of the outer loop if no swaps occur in the inner loop.

Implementation

```
1. void bubbleSort(vector<int> &v) {  
2.     int n = v.size();  
3.     bool swapped = true;  
4.     int i, j;  
5.     for(i = 0; i < n-1; ++i) {  
6.         swapped = false;  
7.         for(j = 0; j < n-i-1; ++j) {  
8.             if(v[j] > v[j+1]) {  
9.                 // elements are out of order  
10.                swap(v[j], v[j+1]);  
11.                // remember a swap occurred  
12.                swapped = true;  
13.            }  
14.        }  
15.        // check if array is sorted
```

```
16.     if(swapped == false) break;
17.   }
18. }
```

Complexity

Now we can analyze the:

- Time complexity $T(N)$
- Number of swaps $S(N)$
- Number of comparisons $C(N)$

for each case. This is done by observing the number of times the lines 8-13 run in each case.

$$T(N) = S(N) + C(N)$$

Time Complexity = Number of Swaps + Number of Comparisons

The relation are as follows:

- $T(N) = T(N-1) + N$
- $C(N) = C(N-1) + (N-1)$
- $S(N)$ depends on the distribution of elements.

Worst Case Time Complexity

$\Theta(N^2)$ is the Worst-Case Time Complexity of Bubble Sort.

This is the case when the array is reversely sort that is in descending order, but we require ascending order or ascending order when descending order is needed.

The number of swaps of two elements is equal to the number of comparisons in this case as every element is out of place.

From equation 2 and 4:

$$T(N) = C(N) = S(N) = \frac{N*(N-1)}{2}$$

Therefore, in the worst case:

- Number of Comparisons: $O(N^2)$ time
- Number of swaps: $O(N^2)$ time

Best Case Time Complexity

$\Theta(N)$ is the Best-Case Time Complexity of Bubble Sort.

This case occurs when the given array is already sorted.

For the algorithm to realize this, only one walk through of the array is required during which no swaps occur (lines 9-13) and the swapped variable (false) indicates that the array is already sorted.

$$T(N) = C(N) = N$$

$$S(N) = 0$$

Therefore, in the best case:

- Number of Comparisons: $N = O(N)$ time
- Number of swaps: $0 = O(1)$ time

Average Case Time Complexity

$\Theta(N^2)$ is the Average Case Time Complexity of Bubble Sort.

The number of comparisons is constant in Bubble Sort so in average case, there is $O(N^2)$ comparisons. This is because irrespective of the arrangement of elements, the number of comparisons $C(N)$ is same.

For the number of swaps, consider the following points:

- If an element is in index I_1 but it should be in index I_2 , then it will take a minimum of $I_2 - I_1$ swaps to bring the element to the correct position.
- An element E will be at a distance of I_3 from its position in sorted array. Maximum value of I_3 will be $N-1$ for the edge elements and it will be $N/2$ for the elements at the middle.

The sum of maximum difference in position across all elements will be:

$$(N-1) + (N-3) + (N-5) \dots + 0 + \dots + (N-3) + (N-1)$$

$$= N \times N - 2 \times (1 + 3 + 5 + \dots + N/2)$$

$$= N^2 - 2 \times N^2 / 4$$

$$= N^2 - N^2 / 2$$

$$= N^2 / 2$$

Therefore, in average, the number of swaps = $O(N^2)$.

Therefore, in the average case time complexity of Bubble sort:

- Number of Comparisons: $O(N^2)$ time
- Number of swaps: $O(N^2)$ time

Space Complexity

The algorithm only requires auxiliary variables for flags, temporary variables and thus the space complexity is $O(1)$.

Comparison with other Sorting Algorithms

The bubble sort algorithm is mainly used to explain theoretical aspects including complexity and algorithm design but rarely used in practice because of how it scales with larger amounts of data. Some complexities of other preferred sorting algorithms are:

ALGORITHM	BEST CASE	AVERAGE CASE	WORST CASE
Bubble Sort	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^2)$
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Quick Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Bucket Sort	$\Omega(n+k)$	$\theta(n+k)$	$O(n^2)$

Note: Bubble Sort and Insertion Sort is efficient for the best case that is when the array is already sorted.

Analysis of Quick Sort

Quick Sort is an optimal sorting algorithm for a practical point of view. If nothing special is known about input data, Quick Sort is often used as a sorting algorithm.

We have explained the different cases like worst case, best case and average case Time Complexity (with Mathematical Analysis) and Space Complexity for Quick Sort.

Sub-topics:

1. Basics of Quick Sort
2. Time Complexity Analysis of Quick Sort
3. Best case Time Complexity of Quick Sort
4. Worst Case Time Complexity of Quick Sort
5. Average Case Time Complexity of Quick Sort
6. Space Complexity

Basics of Quick Sort

Quick Sort is a sorting algorithm which uses divide and conquer technique.

In quick sort, we choose an element as a pivot, and we create a partition of array around that pivot. by repeating this technique for each partition, we get our array sorted

Depending on the position of the pivot, we can apply quick sort in different ways:

1. taking first or last element as pivot
2. taking median element as pivot

Pseudocode:

```
quick_sort(array, start, end)
{
    if(start < end)
    {
        pivot_index = partition(array, start, end);
        quick_sort(array, start, pivot_index-1);
```

```
    quick_sort(array, pivot_index+1, end);  
}  
}
```

Sample code:

```
#include<bits/stdc++.h>  
using namespace std;  
template<class T> void swap(T *a, T *b)  
{  
    T t = *a;  
    *a = *b;  
    *b = t;  
}  
  
int partition(int a[], int start, int end)  
{  
    int pivot = a[end];  
    int i = (start-1);  
    for(int j=start; j<=end-1; j++)  
    {  
        if(a[j]<=pivot)  
        {  
            i++;  
            swap(&a[i], &a[j]);  
        }  
    }  
    swap(a[i+1], a[end]);  
    return (i+1);  
}  
  
void quick_sort(int a[], int start, int end)  
{  
    if(start<end)  
    {  
        int pivot_index = partition(a, start, end);  
        quick_sort(a, start, pivot_index-1);  
        quick_sort(a, pivot_index+1, end);  
    }  
}  
  
int main()  
{  
    int a[] = {5, 2, 3, 4, 1, 6};  
    int n = sizeof(a)/sizeof(a[0]);  
    cout<<"Before"<<endl;  
    for(int i=0; i<n; i++) cout<<a[i]<<" "; cout<<endl;  
    quick_sort(a, 0, n-1);  
    cout<<"After"<<endl;
```

```
for(int i=0; i<n; i++) cout<<a[i]<<" "; cout<<endl;
return 0;
}
```

Output:

```
Before
5 2 3 4 1 6
After
1 2 3 4 5 6
```

Time Complexity Analysis of Quick Sort

The Average case time complexity of quick sort is $O(N \log(N))$.

The derivation is based on the following notation:

T(N) = Time Complexity of Quick Sort for input of size N.

At each step, the input of size N is broken into two parts say J and N-J.

$$T(N) = T(J) + T(N-J) + M(N)$$

The intuition is:

```
Time Complexity for N elements =
    Time Complexity for J elements +
    Time Complexity for N-J elements +
    Time Complexity for finding the pivot
```

where:

- **T(N)** = Time Complexity of Quick Sort for input of size N.
- **T(J)** = Time Complexity of Quick Sort for input of size J.
- **T(N-J)** = Time Complexity of Quick Sort for input of size N-J.
- **M(N)** = Time Complexity of finding the pivot element for N elements.

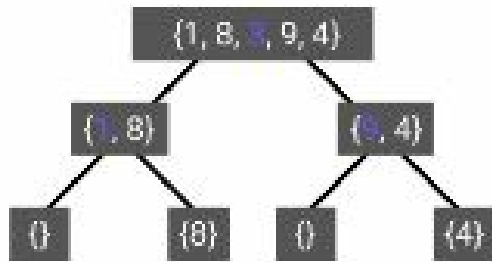
Quick Sort performs differently based on: How we choose the pivot? that is:

$M(N)$ time

How we divide the N elements $\rightarrow J$ and $N-J$ where J is from 0 to $N-1$

On solving for $T(N)$, we will find the time complexity of Quick Sort.

Best case Time Complexity of Quick Sort



$O(N \log(N))$

The best case of quick sort is when we will select pivot as a mean element.

In this case, the recursion will look as shown in diagram, as we can see in diagram the height of tree is $\log N$ and in each level we will be traversing to all the elements with total operations will be $\log N * N$.

As we have selected mean element as pivot then the array will be divided in branches of equal size so that the height of the tree will be minimum.

Pivot for each recursion is represented using blue color.

Time complexity will be $O(N \log N)$

Explanation

Let $T(N)$ be the time complexity for best cases

N = total number of elements

then,

$$T(N) = 2 * T(N/2) + \text{constant} * N$$

$2 * T(N/2)$ is because we are dividing array into two array of equal size.

constant * N is because we will be traversing elements of array in each level of tree.

Therefore,

$$T(N) = 2 * T(N/2) + \text{constant} * N$$

Further, we will divide the array into 2 arrays of equal size so:

$$T(N) = 2 * (2 * T(N/4) + \text{constant} * N/2) + \text{constant} * N == 4 * T(N/4) + 2 * \text{constant} * N$$

For this, we can say that

$$T(N) = 2^k * T(N/(2^k)) + k * \text{constant} * N$$

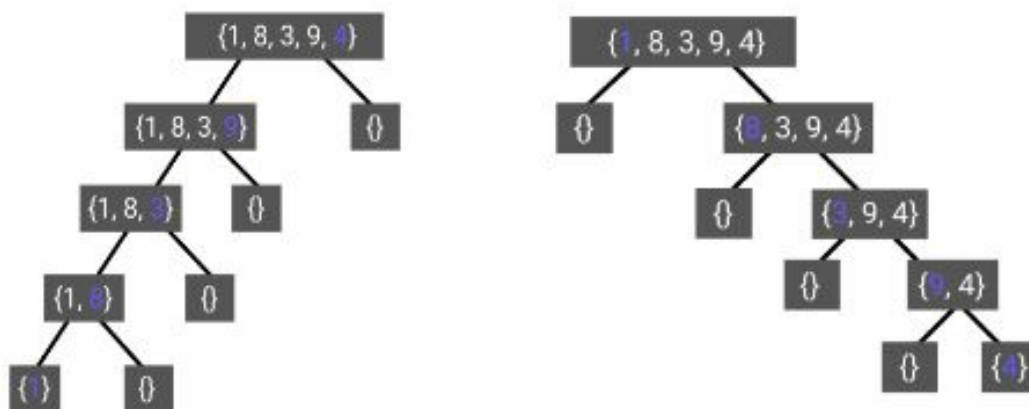
$$\text{then } N = 2^k$$

$$k = \log_2(N)$$

Therefore,

$$T(N) = N * T(1) + N * \log N = O(N * \log_2(N))$$

Worst Case Time Complexity of Quick Sort



$$O(N^2)$$

This will happen when we will when our array will be sorted and we select smallest or largest indexed element as pivot

As we can see in diagram, we are always selecting pivot as corner index

elements.

So, height of the tree will be N and in top node we will be doing N operations.

Then N-1 and so on till 1.

Explanation

Let $T(N)$ be total time complexity for worst case

N = total number of elements

$$T(N) = T(N-1) + \text{constant} * N$$

As we are dividing array into two parts one consisting of single element and other of $n-1$ and we will traverse individual array

$$T(N) = T(N-2) + \text{constant} * (N-1) + \text{constant} * N = T(N-2) + 2 * \text{constant} * N - \text{constant}$$

$$T(N) = T(N-3) + 3 * \text{constant} * N - 2 * \text{constant} - \text{constant}$$

$$T(N) = T(N-k) + k * \text{constant} * N - (k-1) * \text{constant} \dots - 2 * \text{constant} - \text{constant}$$

$$T(N) = T(N-k) + k * \text{constant} * N - \text{constant} * [(k-1) \dots + 3 + 2 + 1]$$

$$T(N) = T(N-k) + k * N * \text{constant} - \text{constant} * [k * (k-1) / 2]$$

Put $N=k$

$$T(N) = T(0) + \text{constant} * N * N - \text{constant} * [N * (N-1) / 2]$$

removing constant terms

$$T(N) = N * N - N * (N - 1) / 2$$

$$T(N) = O(N^2)$$

We can reduce complexity for worst case by randomly picking pivot instead of selecting start or end elements.

Average Case Time Complexity of Quick Sort

$O(N \log(N))$

The overall average case for the quick sort is this which we will get by taking average of all complexities

Explanation

let $T(N)$ be total time taken.

Then for average we will consider random element as pivot

let index i be pivot

Then time complexity will be:

$$T(N) = T(i) + T(N-i)$$

$$T(N) = \frac{1}{N} * \sum_{i=1}^{N-1} T(i) + \frac{1}{N} * \sum_{i=1}^{N-1} T(N-i)$$

As $\sum_{i=1}^{N-1} T(i)$ and $\sum_{i=1}^{N-1} T(N-i)$ are equal likely functions, therefore:

$$T(N) = \frac{2}{N} * \sum_{i=1}^{N-1} T(i)$$

Multiply both side by N

$$N * T(N) = 2 * \sum_{i=1}^{N-1} T(i) \quad \dots\dots\dots(1)$$

put $N = N-1$

$$(N-1) * T(N-1) = 2 * \sum_{i=1}^{N-2} T(i) \quad \dots\dots\dots(2)$$

Subtract 1 and 2, then we will get:

$$N * T(N) - (N-1) * T(N-1) = 2 * T(N-1) + c * N^2 + c * (N-1)^2$$

$$N * T(N) = T(N-1)[2 + N - 1] + 2 * c * N - c$$

$$N * T(N) = T(N-1) * (N+1) + 2 * c * N \text{ [removed } c \text{ as it was constant]}$$

Divide both side by $n*(n+1)$,

$$T(N)/(N+1) = T(N-1)/N + 2*c/(N+1) \dots\dots\dots(3)$$

Put $N = N-1$,

$$T(N-1)/N = T(N-2)/(N-1) + 2*c/N \dots\dots\dots(4)$$

Put $N = N-2$,

$$T(N-2)/N = T(N-3)/(N-2) + 2*c/(N-1) \dots\dots\dots(5)$$

By putting 4 in 3 and then 3 in 2, we will get:

$$T(N)/(N+1) = T(N-2)/(N-1) + 2*c/(N-1) + 2*c/N + 2*c/(N+1)$$

Also, we can find equation for $T(N-2)$ by putting $N = N-2$ in (3)

At last, we will get:

$$T(N) / (N+1) = T(1)/2 + 2*c * [1/(N-1) + 1/N + 1/(N+1) + \dots]$$

$$T(N) / (N+1) = T(1)/2 + 2*c*\log(N) + C$$

$$T(N) = 2*c*\log(N) * (N+1)$$

Now by removing constants,

$$T(N) = \log(N) * (N+1)$$

Therefore,

$$T(N) = O(N * \log(N))$$

Space Complexity

$$O(N)$$

as we are not creating any container other than the given input array.

Therefore, Space complexity will be in order of N .

Bound for non-comparison-based sorting

The analysis we did for the previous few chapters is for sorting algorithms based on comparison. It may seem obvious that we cannot optimize it further, but the truth is we can.

The loophole is that comparison is not necessary for sorting. This is not a simple statement so think about this deeply.

A sorting problem can be seen as a:

- Comparison problem
- Ordering problem
- Matching problem

The idea is if we have an already sorted list, we can sort any list just by matching the ordering of the second list to the first already sorted list.

This process does not require comparison.

Therefore, to find the theoretical limit of a problem, we need to consider different ways to handle the problem. Note formulating the actual algorithm is not necessary.

The theoretical limit is $O(N)$ for sorting as we need linear time for a matching problem. Consider the problem of matching each element to be even or odd. Sorting is a similar problem as comparison is always between two elements.

Another way to think about this is as we need to process all elements, so sorting cannot be done lower than $O(N)$. The best we can do is $O(N)$ while analysis of comparison-based sorting shows lower limit is $O(N \log N)$.

Counting sort, radix sort and bucket sort are some of the non-comparison-based sorting. These algorithms make no comparison. These are also known as linear sorting algorithms. They do not need a comparison decision tree.

Counting sort uses the key values for sorting, where N is the number of

inputs and K is the maximum element. Counting sort takes $O(N+K)$ time which is better than comparison-based sorting. If N is as large as K then this is $O(N)$. Since non-comparison-based sorting algorithms don't make a comparison, it allows sorting at linear time.

Analysis of Counting Sort

Counting Sort is a non-comparison-based sorting algorithm.

In this chapter, we have explored the time complexity of Counting Sort for Average case, Worst case and Best case and also, covered the space complexity using Mathematical analysis.

Table of contents:

1. Introduction to Counting Sort
2. Time Complexity analysis
3. Worst case time complexity
4. Best case time complexity
5. Average case time complexity
6. Space Complexity
7. Conclusion on time and space complexity
8. Comparison with other sorting algorithms

In short,

- Time complexity: $O(N+K)$
- Space Complexity: $O(K)$
- Worst case: when data is skewed and range is large
- Best Case: When all elements are same
- Average Case: $O(N+K)$ (N & K equally dominant)

where:

- N is the number of elements
- K is the range of elements ($K = \text{largest element} - \text{smallest element}$)

Introduction to Counting Sort

Counting Sort is a sorting algorithm that can be used for sorting elements within a specific range and is based on the frequency/ count of each element to be sorted.

It is an integer based, out-place and a stable sorting algorithm.

Algorithm:

Let A be the given array with n elements, B be the output array (containing sorted elements of A) and k is the max element of A.

COUNTING-SORT(A,B,k)

```
step 1 : let C [0...k] be a new array
step 2 : for i = 0 to k :
    C[i] = 0
step 3 : for j = 1 to A.length :
    C[A[j]] = C[A[j]] + 1
    // C[i] now contains the number of
    // elements equal to i.
step 4 : for i = 1 to k :
    C[i] = C[i] + C[i - 1]
    // C[i] now contains the number of
    // elements less than or equal to i.
step 5 : for j = A.length down to 1 :
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

In step 1, we initialize an auxiliary array C of size k . After the for loop of step 2 initializes the array C to all zeros, the for loop of step 3 inspects each input element. If the value of an input element is i, we increment C[i]. Thus, after step 3, C[i] holds the number of input elements equal to i for each integer i = 0, 1, ..., k. Step 4 determine for each i = 0, 1, ..., k how many input elements are less than or equal to i by keeping a running sum of the array C. Finally for loop of step 5 places each element A[j] in its correct sorted position in the output array B.

For a given array A = [2, 5, 3, 0, 2, 3, 0, 3],

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

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

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	

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

(c)

	1	2	3	4	5	6	7	8
B		0						3

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

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

	0	1	2	3	4	5
C	1	2	4	5	7	8

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

Time Complexity analysis

Let us now analyze the time complexity of the above algorithm:

- step 1 takes constant time.
- In step 2, for loop is executed for k times and hence it takes $O(k)$ time.
- In step 3, for loop is executed for n times and hence it takes $O(n)$ time.
- In step 4, for loop is executed for k times and hence it takes $O(k)$ time.
- In step 5, for loop is executed for n times and hence it takes $O(n)$ time.

Thus, the overall time complexity is $O(N+K)$

where:

- N is the number of elements
- K is the range of elements ($K = \text{largest element} - \text{smallest element}$)

The basic intuition behind this can be that, as counting the occurrence of each element in the input range takes k time and then finding the correct index value of each element in the sorted output array takes n time, thus the total time complexity becomes $O(N+K)$.

Counting sort algorithm is a non-comparison-based sorting algorithm that is the arrangement of elements in the array does not affect the flow of algorithm. No matter if the elements in the array are already sorted, reverse sorted or randomly sorted, the algorithm works the same for all these cases and thus the time complexity for all such cases is same that is $O(N+K)$.

Worst case time complexity

Worst case time complexity is when the data is skewed that is the largest element is significantly large than other elements. This increases the range K .

As the time complexity of algorithm is $O(N+K)$ then, for example, when k is of the order $O(N^2)$, it makes the time complexity $O(N+(N^2))$, which essentially reduces to $O(N^2)$ and if k is of the order $O(N^3)$, it makes the time complexity $O(N+(N^3))$, which essentially reduces to $O(N^3)$. Hence, in this case, the time complexity got worse making it $O(K)$ for such larger values of K . And this is not the end. It can get even worse for further larger values of K .

Thus, the worst-case time complexity of counting sort occurs when the range k of the elements is significantly larger than the other elements.

Best case time complexity

The best-case time complexity occurs when all elements are of the same range that is when k is equal to 1.

In this case, counting the occurrence of each element in the input range takes constant time and then finding the correct index value of each element in the sorted output array takes N time, thus the total time complexity reduces to $O(1 + N)$ that is $O(N)$ which is linear.

Average case time complexity

To compute the average case time complexity, first we fix N and take different values of k from 1 to infinity, in this case k computes to be $(k+1/2)$ and the average case will be $N+(K+1)/2$. But as K tends to infinity, K is the dominant factor.

Similarly, now if we vary N , we see that both N and K are equally dominant and hence, we have $O(N+K)$ as average case.

Conclusion on time complexity:

The time complexity of counting sort algorithm is $O(N+K)$ where N is the

number of elements in the array and K is the range of the elements.

Counting sort is most efficient if the range of input values is not greater than the number of values to be sorted. In that scenario, the complexity of counting sort is much closer to $O(N)$, making it a linear sorting algorithm.

Space Complexity

In the above algorithm we have used an auxiliary array C of size k , where k is the max element of the given array. Therefore, the space complexity of Counting Sort algorithm is $O(K)$.

Space Complexity : $O(K)$

Larger the range of elements in the given array, larger is the space complexity, hence space complexity of counting sort is bad if the range of integers are very large as the auxiliary array of that size has to be made.

Conclusion on time and space complexity

- Time complexity: $O(N+K)$
- Space Complexity: $O(K)$
- Worst case: when data is skewed, and range is large
- Best Case: When all elements are same
- Average Case: $O(N+K)$ (N & K equally dominant)

where:

- N is the number of elements
- K is the range of elements ($K = \text{largest element} - \text{smallest element}$)

So here we conclude that:

Counting sort algorithm work best if k is not significantly larger than N . In this case the complexity becomes close to $O(N)$ or linear.

Also, larger the range of elements in the given array, larger is the space complexity.

Comparison with other sorting algorithms

While any comparison-based sorting algorithm requires $O(N \log N)$ comparisons, counting sort has a running time of $O(N)$, when the length of the input list is not much smaller than the largest key value, K , in the input array.

Counting sort is very efficient in the cases where range is comparable to number of input elements as it performs sorting in linear time and can be a advantage in such cases over other sorting algorithms such as quick sort. When in the worst-case quick sort takes $O(N^2)$ time, counting sort only takes $O(N)$ time provided that the range of elements is not very large.

Most sorting algorithms perform in quadratic time ($O(N^2)$), and the two exceptions — heap and merge sort in time ($O(N \log N)$). Counting sort is the only sorting algorithm which performs in linear time (for small range of elements).

There is no comparison between any elements, so it is better than comparison-based sorting techniques.

Since counting sort is suitable for sorting numbers that belong to a well-defined, finite, and small range, it can be used as a subprogram in other sorting algorithms like radix sort which can be used for sorting numbers having a large range.

Analysis of Bucket Sort

Bucket Sort is another non-comparison-based sorting algorithm that uses Counting Sort within it.

We have explained the Time and Space Complexity analysis of Bucket sort along with its algorithm, space complexity and time complexity for worst case, average case and best case in this chapter.

Sub-topics:

1. Introduction to Bucket Sort
2. Algorithm
3. Time complexity analysis
4. Worst case time complexity
5. Average case time complexity
6. Best case time complexity
7. Space complexity
8. Comparison with other sorting algorithms

In short,

- Time complexity: $O(N+K)$
- Space Complexity: $O(N+K)$
- Worst case: $O(N^2)$
- Best Case: $O(N+K)$
- Average Case: $O(N + N^2/K + K)$, $O(N)$ when $K = \Theta(N)$

where:

- N is the number of elements
- K is the number of buckets

Introduction to Bucket Sort

Bucket Sort is a type of sorting algorithm in which the individual elements of the input array are distributed into several groups which are known as buckets. One bucket is capable of holding more than one element simultaneously. Each bucket is then sorted individually, either by making use

of some other sorting algorithm, or by recursively applying the same bucket sorting algorithm.

Finally, when each bucket has been sorted, they are then combined to form the final and complete sorted array.

Bucket Sort is a stable sorting algorithm.

Algorithm

Bucket sort is especially useful when we are dealing with floating-point values.

Here is the algorithm for bucket sort:

```
define function bucketSort(arr[]) as
  step 1: create as many empty buckets (lists or arrays) as the length of the input array
  step 2: store the array elements into the buckets based on their values (described in step 3)
  step 3: for i from 0 to length of arr do
    (a) bucket_index = int((n * arr[i]) / 10) //here n is the length of the input array
    (b) store the element arr[i] at bucket[bucket_index]
  step 4: for i from 0 to length of bucket do:
    (a) sort the elements stored inside bucket[i], either by applying recursion or by
    some other sorting method
  step 5: push the values stored in each bucket starting from lowest index to highest index to a
  new array/list
  step 6: return the sorted array/list
```

Now let us understand as to how this algorithm works with the help of an example. Suppose we want to sort the following array by using Bucket Sort.

0	1	2	3	4	5
5.16	3.79	3.65	4.16	1.63	5.12

Let us follow the algorithm now to sort the array through Bucket Sort.

The size of the input array is 6, so we have to create 6 ($k = 6$) empty buckets.

0
1
2
3
4
5

Now let us start inserting the elements from the input array into the bucket by following the algorithm.

Following step 3 from the algorithm, when $i = 0$, we get

$\text{bucket_index} = \text{int}((6 * 5.16) / 10)$

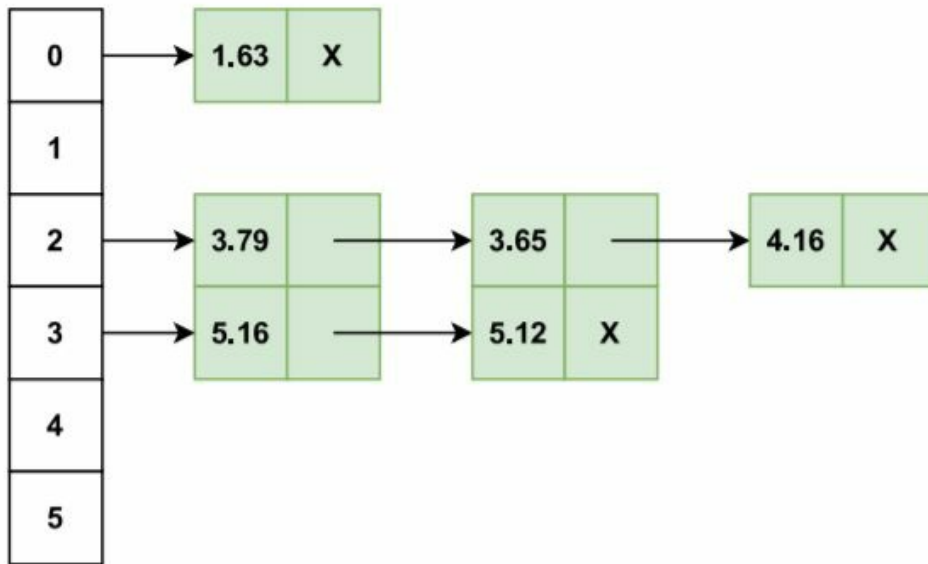
$= \text{int}(30.96 / 10)$

$= \text{int}(3.096)$

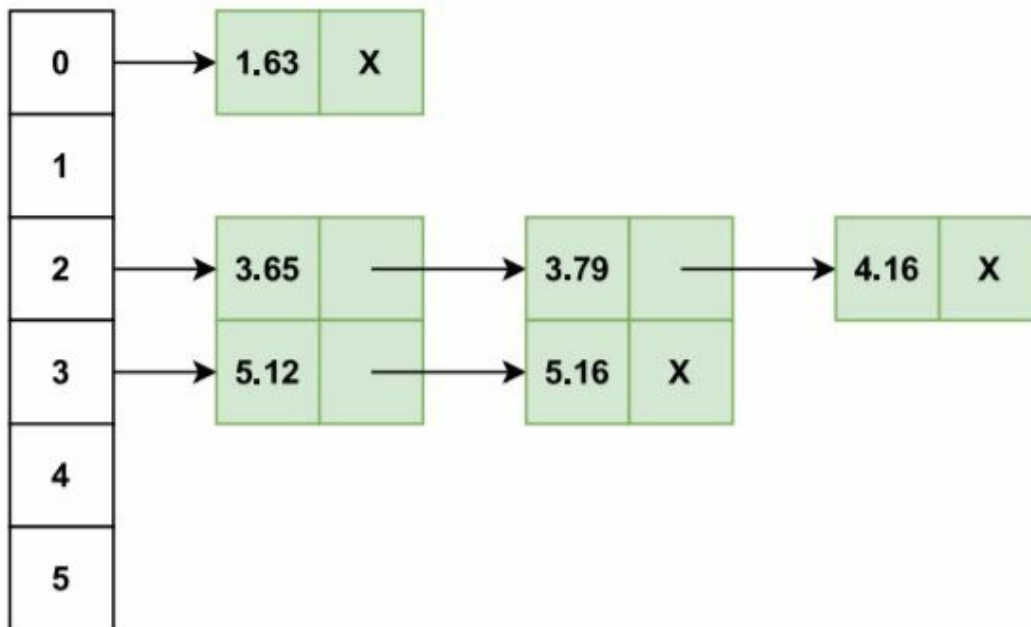
$= 3$

Therefore, the element present at $\text{arr}[0]$, which is 5.16, will be inserted to the bucket at index 3.

Following this step for each element, we will get our final bucket list as follows:



Moving onto step 4, we have to sort the elements stored inside each bucket individually either by recursion or by some other sorting algorithm, and then combine the buckets to get our final sorted array. For our example and for the purpose of analysis, let us use the Bubble sort algorithm to sort each bucket. After the sorting of each individual bucket, we will have the following final bucket structure:



Do you notice something in the final figure?

When you move down in the bucket list from the first index (0) to the last index (5), you will notice that we already have our complete sorted array there. We just simply have to extract it from the bucket list in a similar manner.

Since all the buckets have been sorted individually, we can extract the sorted array/list from the final bucket structure.

We will start pushing the elements stored in the bucket list, starting from the first index (0) to the last index (5) to a new array/list. This operation will result in a sorted array.

Final array obtained is:

0	1	2	3	4	5
1.63	3.65	3.79	4.16	5.12	5.16

Time complexity analysis

The time complexity in Bucket Sort largely depends upon the size of the bucket list and also the range over which the elements in the array/list have been distributed. For example, if the elements in the array don't have a significant mathematical difference between them, it could result in most of the elements being stored in the same bucket, which would significantly increase the complexity of the algorithm.

Consider the following snippet from the algorithm (step 5):

```
for each bucket b in bucket_list do
  for each element i in b do
    push i to array
```

Let us suppose that there is a total of k different buckets, so the outermost

loop will take at least $O(k)$ time.

The inner loop will take at least $O(n)$ time overall, since there are a total of n elements distributed across the bucket list.

Hence, we can conclude that the overall complexity of the Bucket Sort will be $O(n + k)$.

Worst case time complexity

As mentioned earlier, if the elements in the array do not have a significant mathematical difference between them, it could result in most of the elements being stored in the same bucket, which would significantly increase the complexity of the algorithm.

The scenario when every element is stored in the same bucket is when the worst case in terms of time complexity actually happens.

In our case, since we are using the Bubble sort algorithm to sort each bucket, the worst case would occur when we have to first loop over n items in the input array (to store them in the bucket list) and since all of them will be stored inside the same bucket, the worst case of the Bubble sort algorithm would come into play as well. We know that the worst-case time complexity of the Bubble sort algorithm is $O(N^2)$, which occurs when the elements in the array/list are stored in the reverse order and hence the total number of comparisons needed will be N^2 .

Therefore, we get a total runtime of $O(N^2)$.

Average case time complexity

In our bucket sort algorithm, we have four loops (step 1, step 3, step 4 and step 5) which iterate over k buckets and n elements.

The average case occurs when the elements are distributed randomly in the list. Bucket sort runs in linear time in all cases until the sum of the squares of the bucket sizes is linear in the total number of elements.

It takes $O(N)$ time in order for us to assign and store all the elements of the input array into the bucket list.

Since we are using the Bubble sort algorithm to sort each bucket individually, step 4 (from our algorithm) would cost us $O(N^2)$ time.

However, we will need to compute its complexity for the average case.

Step 4 (bubble sort sorting) of our algorithm costs:

$$O\left(\sum_{i=1}^k n_i^2\right)$$

where n_i is the total length of the bucket present at index i and k is the total number of buckets.

Since we are concerned with computing the average time, $E(n_i^2)$ (expectation) is what we are interested in.

Let X_{ij} be the random variable that is 1 if element j is placed in the bucket i , else it will be 0.

We have the value of n_i as:

$$n_i = \sum_{j=1}^n X_{ij}$$

Hence, we get:

$$\begin{aligned} E(n_i^2) &= E\left(\sum_{j=1}^n X_{ij} \sum_{k=1}^n X_{ik}\right) \\ &= E\left(\sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik}\right) \\ &= E\left(\sum_{j=1}^n X_{ij}^2\right) + E\left(\sum_{1 \leq j, k \leq n} \sum_{j \neq k} X_{ij} X_{ik}\right) \end{aligned}$$

Now the summation has been divided into two cases, which are:

(a) $j = k$

(b) $j \neq k$

Since there are k buckets, the probability that an object is distributed to bucket i is $1/k$.

The value of X_{ij} is 1 with probability $1/k$ else it is 0. Therefore, we get:

$$E(X_{ij}^2) = 1^2 \left(\frac{1}{k}\right) + 0^2 \left(1 - \frac{1}{k}\right) = \frac{1}{k}$$

$$E(X_{ij}X_{ik}) = 1 \left(\frac{1}{k}\right) \left(\frac{1}{k}\right) = \frac{1}{k^2}$$

Computing it with the summation now, we get:

$$\begin{aligned} E\left(\sum_{j=1}^n X_{ij}^2\right) + E\left(\sum_{1 \leq j, k \leq n} \sum_{j \neq k} X_{ij}X_{ik}\right) &= n \cdot \frac{1}{k} + n(n-1) \cdot \frac{1}{k^2} \\ &= \frac{n^2 + nk - n}{k^2} \end{aligned}$$

The complexity then becomes:

$$\begin{aligned} O\left(\sum_{i=1}^k E(n_i^2)\right) &= O\left(\sum_{i=1}^k \frac{n^2 + nk - n}{k^2}\right) \\ &= O\left(\frac{n^2}{k} + n\right) \end{aligned}$$

As our final step, we have to concatenate our elements back together in the form of an array to produce our final sorted array. Since there are k buckets,

this step will cost us a total of $O(k)$ time.

Hence, finally, we come to the conclusion that the average case time complexity for this algorithm will be

$$O(n + n^2/k + k).$$

Keep in mind that if k is chosen to be $\Theta(n)$, then bucket sort runs in $O(n)$ average time, provided that we have a uniformly distributed input in the form of an array or list.

Best case time complexity

The best case in Bucket sort occurs when the elements are distributed in a balanced manner across the buckets, i.e., the number of elements spread across in each bucket are identical.

It can get even better depending upon the sorting algorithm that you are using to sort each bucket individually. You can even use different sorting algorithms for different cases. If the elements stored inside the bucket list are already sorted somehow after the partitioning, the complexity will become even better.

Once again, we will have a complexity of $O(k)$ for assigning and pushing elements onto the bucket list and a complexity of $O(n)$ for sorting the elements inside the bucket list.

Hence, we get a total runtime of $O(n + k)$.

Space complexity

The space complexity for Bucket sort is $O(n + k)$, where n is the number of elements and k is the number of buckets.

Hence, the space complexity of this algorithm gets worse with the increase in the size of the input array and the bucket list as well.

Comparison with other sorting algorithms

Let us find out as to how the Bucket sort algorithm fares amongst its other counterpart sorting algorithms.

It can be said that Bucket sort is a generalization or a slightly modified version of Counting sort. Consider the scenario when each bucket is of size 1, then bucket sort will simply have the same working as that of counting sort.

In the scenario when the bucket list only consists of two buckets, the algorithm effectively becomes a version of QuickSort where the pivot value is the middle value of the input elements.

Top-down radix sort can also be categorized as a generalization of bucket sort where both the range of values and the number of buckets in the bucket list are constrained to be a power of two.

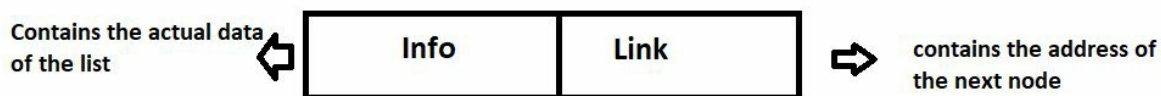
Analysis of Linked List

Singly Linked List is a variant of Linked List which allows only forward traversal of linked lists. This is a simple form, yet it is effective for several problems such as Big Integer calculations.

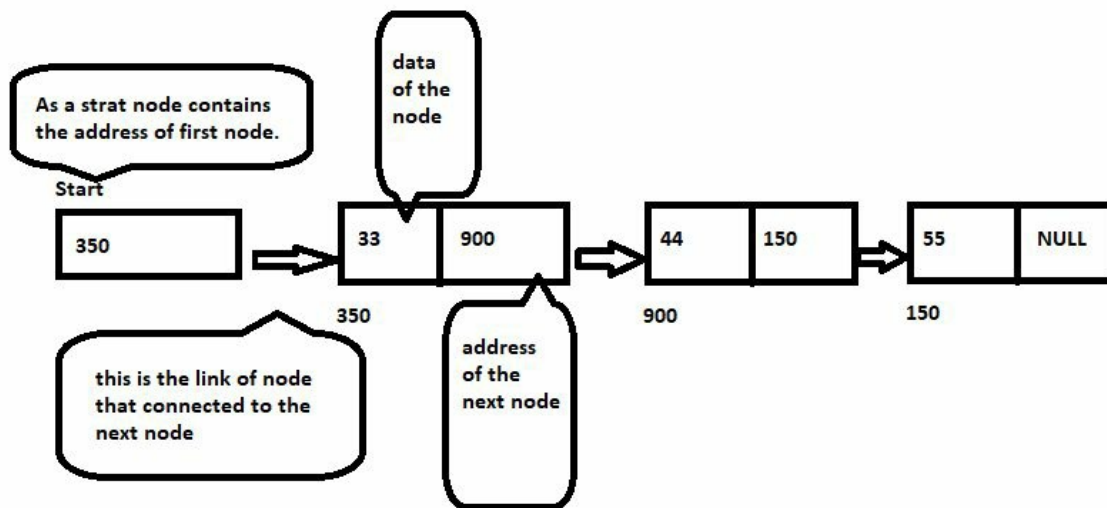
A singly linked list is made up of nodes where each node has two parts:

The first part contains the actual data of the node

The second part contains a link that points to the next node of the list that is the address of the next node.



The beginning of the node marked by a special pointer named START. The pointer points to the first node of the list but the link part of the last node has no next node to point to.



The main difference from an array is:

- Elements are not stored in contiguous memory locations.
- Size of Linked List need not be known in advance. It can increase at runtime depending on number of elements dynamically without any

overhead.

In Singly Linked List, only the pointer to the first node is stored. The other nodes are accessed one by one.

To get the address of i^{th} node, we need to traverse all nodes before it because the address of i^{th} node is stored with $i-1^{\text{th}}$ node and so on.

From the initial chapter on time complexity of memory address, we know that to access a specific element, the time complexity is $O(\sqrt{N})$ where N is block of continuous elements being read.

As Linked List elements are not contiguous, each element access incurs a Time Complexity of $O(\sqrt{N})$.

This is an overhead compared to Array where the overhead is encountered only once.

The advantage of Linked List comes when we have to insert an element at current location or delete current element. This is done in constant time as we need to change the address stored to previous nodes only.

In case of array, for similar operation, we need to shift all other elements which makes the time complexity linear.

Operations at the end of Linked List like inserting element at the end takes linear time because we have to traverse to the end. If we are already at the end, then we can insert the element in constant time.

The summary of Time Complexity of operations in a Linked List is:

Singly Linked List operation	Real Time Complexity	Assumed Time Complexity
Access i^{th} element	$O(\sqrt{N} * N)$	$O(N)$
Traverse all elements	$O(N * \sqrt{N})$	$O(N)$
Insert element E at current point	$O(1)$	$O(1)$
Delete current element	$O(1)$	$O(1)$
Insert element E at		

front	O(1)	O(1)
Insert element E at end	O(N * \sqrt{N})	O(N)

There are other variants of Linked List like:

- Doubly Linked List (each node stores the address of previous node as well)
- Circular Linked List (last node points to the first node)

Time Complexity of different operations in different variants vary. For example, Time Complexity to insert element at front is O(1) in Singly Linked List but it is O(\sqrt{N}) for Doubly Linked List as we have to access the next element to set its previous address to the new element.

We have provided the Time Complexity of all variants of Linked List in the “Time Complexity cheat sheet” chapter.

Analysis of Hash functions

A Hash function converts a data (may be string, number, object) to a N-bit number. An ideal Hash function convert every unique data to a unique N-bit number (known as hash) and distributes data uniformly across all 2^N possibilities (as there are N bits).

As inputs increase and the N-bits remain constant, there will be 2 data D1 and D2 whose hash value is the same. This is known as collision.

We want to know the probability of collision.

Assume that the hash function H hashes to N bits. Let $T = 2^N$ = number of unique hash values.

Assume we will hash M elements.

Probability(collision(T, M)) = Probability of collision with M elements being hashed by a hash function with T unique values.

$$\begin{aligned}\text{Probability}(\text{collision}(T, M)) &= 1 - \text{Probability}(\text{no_collision}(T, M)) \\ &= 1 - \text{Probability}(\text{no_collision}(T, 1)) * \text{Probability}(\text{no_collision}(T, 2)) * \dots \\ &\quad \text{Probability}(\text{no_collision}(T, M))\end{aligned}$$

If there are V elements, then the next element will not collide with the previous V elements if it takes $T - V$ positions.

Therefore, probability that i^{th} element does not collide is: $(T-i)/T$

as there are $T-i$ slots left and T is the total number of slots.

$$\text{Probability}(\text{collision}(T, M)) = 1 - 1 * (T-1)/T * (T-2)/T * \dots * (T-M+1)/T$$

Average total number of collisions =

$$\sum_{i=1}^{M-1} \frac{i}{T} = \frac{M(M-1)}{2T}$$

Based on this, the first collision will happen when

$$M = \sqrt{2} * \sqrt{T}$$

where:

- M = number of elements being hashed.
- T = total number of hash values in the hash function

This means if $\sqrt{2} * \sqrt{T}$ elements are hashed, then we will see the first collision.

Note: T is the total number of possible hash values.

Therefore, if we use a 64 bit hash, then the total number of possible hash values will be 2^{64} . The first collision will take place when we insert $\sqrt{2} * \sqrt{T} = 2^{32.5}$ which is approximately 10^9 that is 1 trillion.

You can imagine or calculate that enormous number of elements that we need to hash to see the first collision if our hash function uses larger number of bits like 256 or 512 bits.

For two collisions, we have:

$$M * (M-1) / 2T = 2$$

$$M = 2 * \sqrt{T}$$

Similarly, for 3 collisions, we have:

$$M * (M-1) / 2T = 3$$

$$M = \sqrt{6} * \sqrt{T}$$

For C collisions, we have:

$$M * (M-1) / 2T = C$$

$$M = \sqrt{2C} * \sqrt{T}$$

For all elements to collide that is M collisions, we have:

$$M * (M-1) / 2T = M$$

$$M = 2T$$

Therefore, for all elements to collide, the elements should be equal to twice the total number of hash values.

COLLISIONS	ELEMENTS TO BE HASHED
1	$1.414 * \sqrt{T}$
2	$2 * \sqrt{T}$
3	$2.449 * \sqrt{T}$
4	$2.828 * \sqrt{T}$
M	$2T$

With this, you can see how the number of collisions increase with the increase in number of elements to be hashed.

If we hash M values and total possible hash values is T, then the expected number of collisions will be:

$$M * (M-1) / 2T = C$$

$$C = M * (M-1) / 2T$$

For example, if there are $2^{16} = 65,536 = T$ and $M = 2000$, then the expected number of collision C is:

$$C = M * (M-1) / 2T$$

$$C = 2000 * (2000 - 1) / 2^{17}$$

$$C = 30.5 \text{ collisions}$$

Conclusions:

The conclusions of our analysis of collision of hash functions are:

1. The first collision will take place when we hash N elements provided the total number of hash values is N^2 .
2. For all elements to collide, the elements should be equal to twice the total

number of hash values.

3. If we hash M values and total possible hash values is T , then the expected number of collisions will be $C = M * (M-1) / 2T$

Based on calculations, you see that this is an effective value.

Analysis of Binary Search

Binary Search algorithm is an efficient comparison-based search algorithm where the key idea is to reduce the size of search space by half in every iteration by exploiting a restriction on the search space that it is in sorted order. When suitable, binary search is chosen over other search algorithms.

The key idea is that if a list is sorted and we compared a number with a random number from the list, we can say whether the potential match lies on the left or the right of the number.

When the number is the middle element, then we can reduce the number of potential matches by half.

Algorithm

The steps involved in Binary search algorithm are:

- Pre-condition: The element list must be sorted.
- Step 1: Find middle element of the array.
- Step 2: Compare the value of the middle element with the target value.
- Step 3: If they match, it is returned.
- Step 4: If the value is less or greater than the target, the search continues in the lower or upper half of the array accordingly.
- Step 5: The same procedure as in step 2-4 continues, but with a smaller part of the array. This continues until the target element is found or until there are no elements left.

Example

Consider the following sorted list in which we want to find 6:

```
-1 3 3 6 11 12 16 17 18 19
```

We chose to compare with the middle element 12 with 6. As $12 > 6$, 6 will lie at the left side of 12.

The potential numbers are:

```
-1 3 3 6 11
```

We compare 6 with the middle element 3. As $3 < 6$, 6 will be at the right side of 3.

The potential numbers are:

```
6 11
```

We compare our target 6 with the middle element 6. As $6 == 6$, we found our target.

In Binary Search, the size of potential number of elements is halved at each step.

$$T(N) = T(N/2) + O(1)$$

$$T(N) = 1 + \dots + 1 \text{ (.... } \log N \text{ times)}$$

$$T(N) = \log N$$

The best case will be that the first middle element is the element to be matched that is it takes $O(1)$ time.

Complexity

- Worst case time complexity: $O(\log N)$
- Average case time complexity: $O(\log N)$
- Best case time complexity: $O(1)$
- Space complexity: $O(1)$

As the search space is reduced by half each time until all elements are eliminated, the worst case is $O(\log N)$.

Remember that the relation is very similar to that of Quick Sort.

Quick Sort: $T(N) = T(N/2) + O(N)$

Binary Search: $T(N) = T(N/2) + O(1)$

Only difference is that in Binary Search, potential data is halved after every constant time operation while for Quick Sort, potential data is halved after a linear time $O(N)$ operation.

Time and Space Complexity Cheat Sheets

In this chapter, we will list the time and space complexity of a large number of Algorithms and Data Structures. This will help you revise the concepts quickly.

Once you have gone through the entire book, you may need to revisit this chapter in periodic intervals to prepare you in a Analysis mindset.

Notation	Symbol	Meaning
Big-O	O	Upper bound
Little-o	o	Tight Upper bound
Big Omega	Ω	Lower bound
Little Omega	ω	Tight Lower bound
Big Theta	Θ	Upper + Lower bound

Recurrence	Algorithm	Big-O
$T(N) = T(N/2) + O(1)$	Binary Search	$O(\log N)$
$T(N) = T(N/2) + O(N)$	Quick Sort	$O(N \log N)$
$T(N) = 3 * T(2N/3) +$	Stooge Sort	$O(N^{2.7095})$

$O(1)$		
$T(N) = T(N-1) + O(1)$	Linear Search	$O(N)$
$T(N) = T(N-1) + O(N)$	Insertion Sort	$O(N^2)$
$T(N) = 2 * T(N-1) + O(1)$	Tower of Hanoi	$O(2^N)$

Big-O	Known as
$O(1)$	Constant Time
$O(\log^* N)$	Iterative Logarithmic Time
$O(\log N)$	Logarithmic Time
$O(N)$	Linear Time
$O(N \log N)$	Log Linear Time
$O(N^2)$	Quadratic Time
$O(N^p)$	Polynomial Time
$O(c^N)$	Exponential Time
$O(N!)$	Factorial Time

Memory operation	Real Time Complexity	Assumed Time Complexity
Access memory at address I	$O(\sqrt{N})$	$O(1)$
Access M contiguous memory starting at		

address I	$O(M + \sqrt{N})$	$O(M)$
Access M memory at distant addresses	$O(M \sqrt{N})$	$O(M)$
<i>Note: N is the block of memory read at once.</i>		

Bitwise operation	Time Complexity	Parallel algorithm
AND	$O(N)$	$O(1)$
OR	$O(N)$	$O(1)$
NOT	$O(N)$	$O(1)$
XOR	$O(N)$	$O(1)$
LEFT SHIFT	$O(N)$	$O(1)$
RIGHT SHIFT	$O(N)$	$O(1)$
<i>Note: N is the number of bits</i>		

Operation	Usual Time Complexity	Optimal Time Complexity	Assumed Time Complexity
Addition	$O(N)$	$O(N)$	$O(1)$
Subtraction	$O(N)$	$O(N)$	$O(1)$
Multiplication	$O(N^2)$	$O(N \log N)$	$O(1)$
Division	$O(N^2)$	$O(N \log N)$	$O(1)$
<i>Note: N is the number of bits; If number is M, then $N = \log M$</i>			

Multiplication		
Algorithm	Time Complexity	Discovered
Basic		

Multiplication	$O(N^2)$	100 BC
Russian Peasant Method	$O(N^2 * \log N)$	1000 AD
Karatsuba algorithm	$O(N^{1.58})$	1960
Toom Cook multiplication	$O(N^{1.46})$	1963
Schonhage Strassen algorithm	$O(N * \log N * \log \log N)$	1971
Furer's algorithm	$O(N * \log N * 2^{O(\log^* N)})$	2007
DKSS Algorithm	$O(N * \log N * 2^{O(\log^* N)})$	2008
Harvey, Hoeven, Lecerf	$O(N * \log N * 2^{3 \log^* N})$	2015
Covanov and Thomé	$O(N * \log N * 2^{2 \log^* N})$	2015
Harvey and van der Hoeven	$O(N * \log N)$	2019
<i>Note: N is the number of bits;</i>		

Division Algorithm (N / N)		
Algorithm	Average case time	Space
Basic Algorithm	$O(\log^2 N)$	$O(\log N)$
Burnikel Ziegler Algorithm	$O(T(N) * \log N)$	$O(\log N)$
Newton Raphson division	$O(T(N))$	$O(\log N)$
N = number of bits, T(N) = Time Complexity of Multiplication		

Exponentiation Algorithm(N^M)		
Algorithm	Average case time	Space
Basic algorithm	$O(T(N) * M)$	$O(\log N)$
Exponentiation by squaring	$O(T(N) * \log M)$	$O(\log N)$

Array		
Operation	Real Time Complexity	Assumed Time Complexity
Access i-th element	$O(\sqrt{N})$	$O(1)$
Traverse all elements	$O(N + \sqrt{N})$	$O(N)$
Override element at i-th index	$O(\sqrt{N})$	$O(1)$
Insert element E	$O(N + \sqrt{N})$	$O(N)$
Delete element E	$O(N + \sqrt{N})$	$O(N)$
<i>$N = \text{number of elements in array}$</i>		

Dynamic Array			
Operation	Worst Case	Average Case	Best Case
Resize	$O(N)$	$O(1)$	$O(N)$
Add	$O(1)$	$O(1)$	$O(1)$
Add at an index	$O(N)$	$O(N)$	$O(1)$
Delete	$O(1)$	$O(1)$	$O(1)$

Delete at an index	$O(N)$	$O(N)$	$O(1)$
<i>$N = \text{number of elements in array}$</i>			
<i>Time to read block of N elements = $O(\sqrt{N})$ not considered.</i>			

Singly Linked List		
Operation	Real Time Complexity	Assumed Time Complexity
Access i-th element	$O(\sqrt{N} * N)$	$O(N)$
Traverse all elements	$O(N * \sqrt{N})$	$O(N)$
Insert element E at current point	$O(1)$	$O(1)$
Delete current element	$O(1)$	$O(1)$
Insert element E at front	$O(1)$	$O(1)$
Insert element E at end	$O(N * \sqrt{N})$	$O(N)$
<i>$N = \text{number of elements in Linked List}$</i>		

Doubly Linked List		
Operation	Real Time Complexity	Assumed Time Complexity
Access i-th element	$O(\sqrt{N} * N)$	$O(N)$
Traverse all elements	$O(N * \sqrt{N})$	$O(N)$
Insert element E		

at current point	$O(\sqrt{N})$	$O(1)$
Delete current element	$O(\sqrt{N})$	$O(1)$
Insert element E at front	$O(\sqrt{N})$	$O(1)$
Insert element E at end	$O(N * \sqrt{N})$	$O(N)$
<i>N = number of elements in Linked List</i>		

Circular Singly Linked List		
Operation	Real Time Complexity	Assumed Time Complexity
Access i-th element	$O(\sqrt{N} * N)$	$O(N)$
Traverse all elements	$O(N * \sqrt{N})$	$O(N)$
Insert element E at current point	$O(\sqrt{N})$	$O(1)$
Delete current element	$O(\sqrt{N})$	$O(1)$
Insert element E at front	$O(\sqrt{N})$	$O(1)$
Insert element E at end	$O(N * \sqrt{N})$	$O(N)$
<i>N = number of elements in Linked List</i>		

Circular Doubly Linked List		
Operation	Real Time Complexity	Assumed Time Complexity
Access i-th element	$O(\sqrt{N} * N)$	$O(N)$
Traverse all		

elements	$O(N * \sqrt{N})$	$O(N)$
Insert element E at current point	$O(\sqrt{N})$	$O(1)$
Delete current element	$O(\sqrt{N})$	$O(1)$
Insert element E at front	$O(\sqrt{N})$	$O(1)$
Insert element E at end	$O(\sqrt{N})$	$O(1)$
<i>N = number of elements in Linked List</i>		

Stack (using Array)				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert / Push	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete / Pop	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Find element E	$O(N)$	$O(N)$	$O(1)$	$O(1)$
Insert element at position I	$O(N)$	$O(N)$	$O(1)$	$O(N)$
Delete element at position I	$O(N)$	$O(N)$	$O(1)$	$O(N)$
<i>N = number of elements in stack</i>				

Stack (using Linked List)				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	

Insert / Push	O(1)	O(1)	O(1)	O(1)
Delete / Pop	O(1)	O(1)	O(1)	O(1)
Find element E	O(N)	O(N)	O(1)	O(1)
Insert element at position I	O(N)	O(N)	O(1)	O(N)
Delete element at position I	O(N)	O(N)	O(1)	O(N)
<i>N = number of elements in stack</i>				

Queue (using Array)				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert / Enqueue	O(1)	O(1)	O(1)	O(1)
Delete / Dequeue	O(1)	O(1)	O(1)	O(1)
Find element E	O(N)	O(N)	O(1)	O(1)
Insert element at position I	O(N)	O(N)	O(1)	O(N)
Delete element at position I	O(N)	O(N)	O(1)	O(N)
<i>N = number of elements in queue</i>				

Queue (using Linked List)		
	Time Complexity	

Operation	Worst	Average	Best	Space Complexity
	Case	Case	Case	
Insert / Enqueue	O(1)	O(1)	O(1)	O(1)
Delete / Dequeue	O(1)	O(1)	O(1)	O(1)
Find element E	O(N)	O(N)	O(1)	O(1)
Insert element at position I	O(N)	O(N)	O(1)	O(N)
Delete element at position I	O(N)	O(N)	O(1)	O(N)
<i>N = number of elements in queue</i>				

Circular Queue (using Linked List or Array)				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert / Enqueue	O(1)	O(1)	O(1)	O(1)
Delete / Dequeue	O(1)	O(1)	O(1)	O(1)
Find element E	O(N)	O(N)	O(1)	O(1)
Insert element at position I	O(N)	O(N)	O(1)	O(N)
Delete element at position I	O(N)	O(N)	O(1)	O(N)
<i>N = number of elements in queue</i>				

Double Ended Queue (using Linked List or Array)				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert / Enqueue at front	O(1)	O(1)	O(1)	O(1)
Insert / Enqueue at end	O(1)	O(1)	O(1)	O(1)
Delete / Dequeue at front	O(1)	O(1)	O(1)	O(1)
Delete / Dequeue at end	O(1)	O(1)	O(1)	O(1)
Find element E	O(N)	O(N)	O(1)	O(1)
Insert element at position I	O(N)	O(N)	O(1)	O(N)
Delete element at position I	O(N)	O(N)	O(1)	O(N)
<i>N = number of elements in queue</i>				

Priority Queue (using Linked List or Array)				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert / Enqueue	O(logN)	O(logN)	O(1)	O(1)
Delete /				

Dequeue	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(1)$
Find element at top	$O(1)$	$O(1)$	$O(1)$	$O(1)$
$N = \text{number of elements in queue}$				

Search Algorithm				
Algorithm	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Linear Search	$O(N)$	$O(N)$	$O(1)$	$O(1)$
Binary Search	$O(\log N)$	$O(\log N)$	$O(1)$	$O(1)$
Search in Hash Map	$O(N)$	$O(1)$	$O(1)$	$O(1)$
Interpolation Search	$O(N)$	$O(\log \log N)$	$O(1)$	$O(1)$
$N = \text{total number of elements}$				

Selection Problem	
Problem	Minimum comparisons
Largest element	$N-1$
2nd Largest element	$N + \log N - 2$
i-th largest element	$\leq 5.4305 * N$
$N = \text{total number of elements}$	

Comparison based Sorting Problem			
Algorithm	Time Complexity		Space
			Stable

	Worst	Average	Best	Complexity	
	Case	Case	Case		
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$	No
Insertion Sort	$O(N^2)$	$O(N^2)$	$O(N)$	$O(1)$	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Yes
Quick Sort	$O(N^2)$	$O(N \log N)$	$O(N \log N)$	$O(\log N)$	No
Bubble Sort	$O(N^2)$	$O(N^2)$	$O(N)$	$O(1)$	Yes
Heap Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(1)$	No
Shell Sort	$O(N^{3/2})$	$O(N^{4/3})$	$O(N \log N)$	$O(1)$	No
<i>N = total number of elements to be sorted</i>					

Non-Comparison based Sorting Problem					
Algorithm	Time Complexity			Space Complexity	Stable
	Worst Case	Average Case	Best Case		
Bucket Sort	$O(N^2)$	$O(N + N^2 / K + K)$	$O(N + K)$	$O(N * K)$	Yes
Counting Sort	$O(N+R)$	$O(N+R)$	$O(N+R)$	$O(N+R)$	Yes
Radix Sort	$O(N * W)$	$O(N * W)$	$O(N)$	$O(N + R)$	Yes
<i>N = total number of elements to be sorted, R = range of element, W = width of element, K= number of buckets</i>					

Non-Traditional based Sorting Problem		
	Time Complexity	

Algorithm	Worst Case	Average Case	Best Case	Space Complexity	Stable
Stooge Sort	$O(N^{2.7095})$	$O(N^{2.7095})$	$O(N^{2.7095})$	$O(N)$	No
Bead Sort	$O(S)$	$O(S)$	$O(N)$	$O(N^2)$	No
Pancake Sort	$O(N)$	$O(N)$	$O(N)$	$O(\log N)$	No
Bogo Sort	$O(N!)$	$O(N!)$	$O(1)$	$O(N)$	No
Bozo Sort	$O(N!)$	$O(N!)$	$O(1)$	$O(N)$	No
<i>N = total number of elements to be sorted, S = sum of elements</i>					

Binary Tree:

Binary Tree				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert Element	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete Element	$O(\log N)$	$O(\log N)$	$O(1)$	$O(1)$
Search Element	$O(N)$	$O(N)$	$O(1)$	$O(1)$
Traverse Tree	$O(N)$	$O(N)$	$O(N)$	$O(1)$
<i>N = number of elements in binary tree</i>				

Binary Search Tree (BST):

Binary Search Tree				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert				

Element	O(1)	O(1)	O(1)	O(1)
Delete Element	O(logN)	O(logN)	O(1)	O(1)
Search Element	O(N)	O(logN)	O(1)	O(1)
Traverse Tree	O(N)	O(N)	O(N)	O(1)
<i>N = number of elements in binary tree</i>				

Balanced BST:

Balanced Binary Search Tree				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert Element	O(1)	O(1)	O(1)	O(1)
Delete Element	O(logN)	O(logN)	O(1)	O(1)
Search Element	O(logN)	O(logN)	O(1)	O(1)
Traverse Tree	O(N)	O(N)	O(N)	O(1)
<i>N = number of elements in binary tree</i>				

Minimum Heap:

Minimum Heap				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Search	O(N)	O(N)	O(1)	O(1)
Insert	O(logN)	O(logN)	O(1)	O(1)
Find minimum	O(1)	O(1)	O(1)	O(1)

Delete minimum	$O(\log N)$	$O(\log N)$	$O(1)$	$O(1)$
Find maximum	$O(N)$	$O(N)$	$O(N)$	$O(1)$
Delete maximum	$O(N)$	$O(N)$	$O(N)$	$O(1)$
<i>N = number of elements in minimum heap</i>				

Maximum Heap:

Maximum Heap				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Search	$O(N)$	$O(N)$	$O(1)$	$O(1)$
Insert	$O(\log N)$	$O(\log N)$	$O(1)$	$O(1)$
Find minimum	$O(N)$	$O(N)$	$O(N)$	$O(1)$
Delete minimum	$O(N)$	$O(N)$	$O(N)$	$O(1)$
Find maximum	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete maximum	$O(\log N)$	$O(\log N)$	$O(1)$	$O(1)$
<i>N = number of elements in maximum heap</i>				

Trie:

Trie				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert	$O(W)$	$O(W)$	$O(W)$	$O(N)$
Delete	$O(W)$	$O(W)$	$O(W)$	

Search	$O(W)$	$O(W)$	$O(W)$	
$N = \text{number of elements}; W = \text{length of element}$				

Van Emde Boas tree				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert	$O(\log \log M)$	$O(\log \log M)$	$O(\log \log M)$	$O(M)$
Delete	$O(\log \log M)$	$O(\log \log M)$	$O(\log \log M)$	
Search	$O(\log \log M)$	$O(\log \log M)$	$O(\log \log M)$	
$M = \text{maximum number of elements stored};$				

X Fast Trie				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert	O(log M)	O(log M)	O(log M)	O(N logM)
Delete	O(log M)	O(log M)	O(log M)	
Search	O(log logM)	O(log logM)	O(log logM)	
<i>N = total number of elements, M = maximum element;</i>				

Y Fast Trie				
Operation	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Insert	$O(\log \log M)$	$O(\log \log M)$	$O(\log \log M)$	$O(N)$
Delete	$O(\log \log M)$	$O(\log \log M)$	$O(\log \log M)$	
Search	$O(\log \log M)$	$O(\log \log M)$	$O(\log \log M)$	
$N = \text{total number of elements, } M = \text{maximum value in domain;}$				

Single Source Shortest Path		
Algorithm	Average case Time Complexity	Space Complexity
Bellman Ford algorithm	$O(V * E)$	$O(V)$
Dijkstra's algorithm with Linked List	$O(V^2)$	$O(V)$
Dijkstra's algorithm with binary heap	$O((E + V) \log V)$	$O(V)$
Dijkstra's algorithm with Fibonacci heap	$O(E + V \log V)$	$O(V)$
Dial's algorithm	$O(E + L * V)$	$O(V)$
Gabow's algorithm	$O(E \log L)$	$O(V)$
Thorup algorithm	$O(E + V \log \log V)$	$O(V)$
<i>E = number of edges, V = number of vertices, L = length of shortest path</i>		

All Pairs Shortest Path		
Algorithm	Average case Time Complexity	Space Complexity
Floyd Warshall algorithm	$O(V^3)$	$O(V^2)$
Williams algorithm	$O(V^3 / 2^{((\log V)^{0.5})})$	$O(V^2)$
Johnson Dijkstra algorithm	$O(E * V + V^2 \log V)$	$O(V^2)$
Pettie algorithm	$O(E * V + V^2 \log \log V)$	$O(V^2)$
Hagerup algorithm	$O(E * V + V^2 \log \log V)$	$O(V^2)$
<i>E = number of edges, V = number of vertices</i>		

Maximum Flow Problem	
Algorithm	Time Complexity
Dinic's algorithm	$O(MN^2)$
Edmonds Karp algorithm	$O(M^2N)$
MPM (Malhotra, Pramodh-Kumar, and Maheshwari)	$O(N^3)$
James B. Orlin	$O(N M)$
N = number of nodes, M = number of edges	

With this, you must have a strong knowledge of Time and Space Complexity of a vast range of Problems and Algorithms.

You must have a strong hold of Problem Analysis and Time and Space Complexity by now.

You know that $O(1)$ is not realistic for a practical operation so analyze the problem at hand and get to know what the best is you can do.

Keep solving problems.

OPENGENUS
iq.opengenus.org