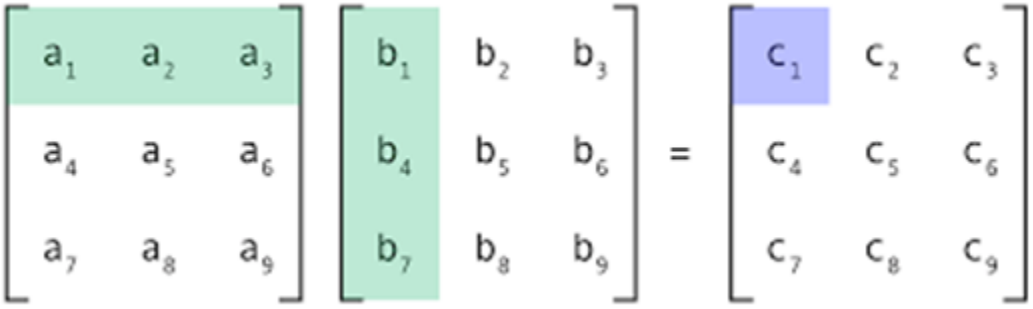
Chapter 0: An Introduction to Complexity Theory

It marvels us to see how fast computers do their tasks, doesn't it? You may feel that as long as we can translate our real-world problems into computer programs, computers can run them to produce answers very fast. However, despite the impressive computation power and memory capacity of present-day computers, most problems we encounter in the real world are too time consuming to solve using computers or too big to hold in them. In fact, this is the history of computers since their early days. As we have better computers, we always try to solve even bigger problems, which keeps us with the same problem of inadequate capacity. Early computers were as big as the entire floor of a big building. Still, the computation power they have were not even one-tenth of the power of your present-day smart-phones. We have come a long way in improving our computers. You would think that is enough. Unfortunately, no, we never had enough – not even we were ever close it.

As the capacity of the computers is scarce, when we write programs, we need to put heavy focus on program efficiency. In classroom, we almost never encounter this issue because the inputs to our problems are quite small. As a result, even an inefficient program can process them very fast (in computer terms). To understand how the execution time of a program can be quite high for real-world inputs consider the simple example of a matrix-matrix multiplication problem A×B = C. You know from elementary school level that to compute the value at a particular row and column index of the output matrix C, we have to pairwise multiply all the elements of that row of A and that column of B, as shown in the following diagram (image is taken from [1]):



Suppose our computer's CPU speed is 2Ghz. Which roughly means it can do one unit of arithmetic operation in every 1 nano-second (10^{-9} second). Now if our input matrices are square and have 10000 rows and columns each, then to compute the value of a single entry in C will cause us to do 10000 multiplications, as we have to multiply an entire row of A with an entire column of B. Hence, for calculating all the entries of C, we need 10000 * 10000^2 = 10000^3 multiplications. That is $(10^4)^3 = 10^{12}$. Thus, to do all the multiplications, the CPU should take at least 10^{12}* 10^{-9} seconds = 1000 seconds =16.67 minutes[2]! You will see that for larger matrices the running time increases like crazy.

Memory space consumption of programs is likewise a great concern. Consider a simple case that you have a file of the NID database of Bangladesh containing the names of all citizens and you are asked to write a program that will print the name of the first individual in the database whose name starts with the word “Aziz.” A simple solution to this problem can be that you read the content of the file in a list in your program. Then traverse the list from the beginning until you find a name that matches the requirement. However, consider that we have more than 50 million = 5 * 10^7 individuals in the file. If the maximum length of a name is 20 characters, then the size of the file can be 10^9 bytes = 100 megabytes. Consequently, your simple program will consume at least that much space in the computer memory. If each entry in the database file has other information besides the names – as it will likely have in the real world – your program could end up consuming over a gigabyte amount space from the RAM just for holding the file content. Given present generation personal computers have 8 to 16 Gigabytes RAM in general, one program consuming over a gigabyte of memory is a huge concern.

Thus, you understand that both CPU computation time and RAM memory space are precious and scarce resources that programmers have to use sparingly. Which entails that programmers can analyze the cost of their program in terms of its CPU execution time and memory space consumption. These two costs are called the **time complexity** and **space complexity** of a program. Typically, the time and space performance of a program varies depending on the specific input, even if different inputs to the program are of the same size/length. What we are concerned most about, is what would be the performance of our program in the worst case. The performance at the worst case is called the **worst-case time and space complexities**. The worst-case complexities are the most important because they dictate how much resource we need to allocate for the program to ensure that it can run safely and maximum how long we may need to wait to get its output. Sometimes people measure the best case and average case time and space complexities also. However, in this book, we only focus on the worst-case complexities exclusively.

Improving Worst Case Performance of a Program

Note that sometimes, some small modification can drastically change the performance of your program. For example, consider the case of searching a matching name in the NID database file again. If you change your earlier program to read just one line at a time from the file, do the first name comparison, then throw away the line if it does not match the searched first name; then your program will only need 20 bytes of storage to hold the data. Furthermore, the new program has the advantage that it stops reading the file as soon as it finds the searched name. So, it will have a better running time also if the search finds the name early. The earlier version must spend the whole file reading to complete before it can start the search.

However, most of the time, we need clever algorithms and data structures to improve program implementations’ performance. That is why data structures and algorithms are two core

elements of computer science studies. Now to consider how a clever algorithm can change the time complexity of a program drastically, consider a simple but classic problem: you are given a sorted increasing sequence of numbers S and a particular value X as input. You need to find the position in S where X occurred. We can solve this problem using the following two function implementations:

Implementation 1

```
Function linearSearch (S, x) {
      L = length(S)
      for (i = 0; i < L; i = i + 1) {
            if (S[i] == x) then return i
      }
      return -1
}
```

Implementation 2

```
Function binarySearch (S, x) {
      Length = length(S)
      L = 0
      H = Length – 1
      while (L <= H) {
            M = (L + H) / 2
            if (x == S[M]) then return M
            else {
                  if (x > S[M]) then L = M + 1
                  else H = M - 1
            }
      }
      return -1
}
```

If we investigate the first implementation, which is called linearSearch, we see that it is a straightforward traversal of the input sequence S. Suppose the length of S is N. In the worst case, we may find x in the last position of S. Hence, the worst-case running time of that implementation is proportional to N.

The second implementation, binarySearch, is cleverer. It divides S into half and check if the middle element is equal, larger, or smaller than x. If it is larger, then x must be located in the first half of S (likewise, if the middle element is smaller than x must be located on the last half of S).

So, it narrows down its scope to that half of S. Then it again compares the middle element of that half with x. It goes on like that unless the search range cannot be made narrower or x is found. The following diagram shows the progression of this implementation (picture is taken from[3]).

In the worst case, the number of steps executed by this implementation is proportional to $\log_2 N$. The reason for this is that at each step the search divides S into half. Hence it can be divided $\log_2 N$ many times in the worst case before L will get larger than H.

Now consider the difference between taking time proportional to N as opposed on $\log_2 N$. For example, if $N = 1024 = 2^{10}$. Then the linearSearch will have a running time that is close to 1024 operations. The binary search will only have a running time proportional to only 10 operations!

---

[1] https://charchithowitzer.medium.com/matrix-multiplication-why-is-it-a-big-deal-cc8ef7490008

[2] CPU actually takes much more time as each multiplication operation involves many steps in the CPU.

[3] https://www.analyticsvidhya.com/blog/2023/09/binary-search-algorithm/

Asymptotic Time/Space Complexity

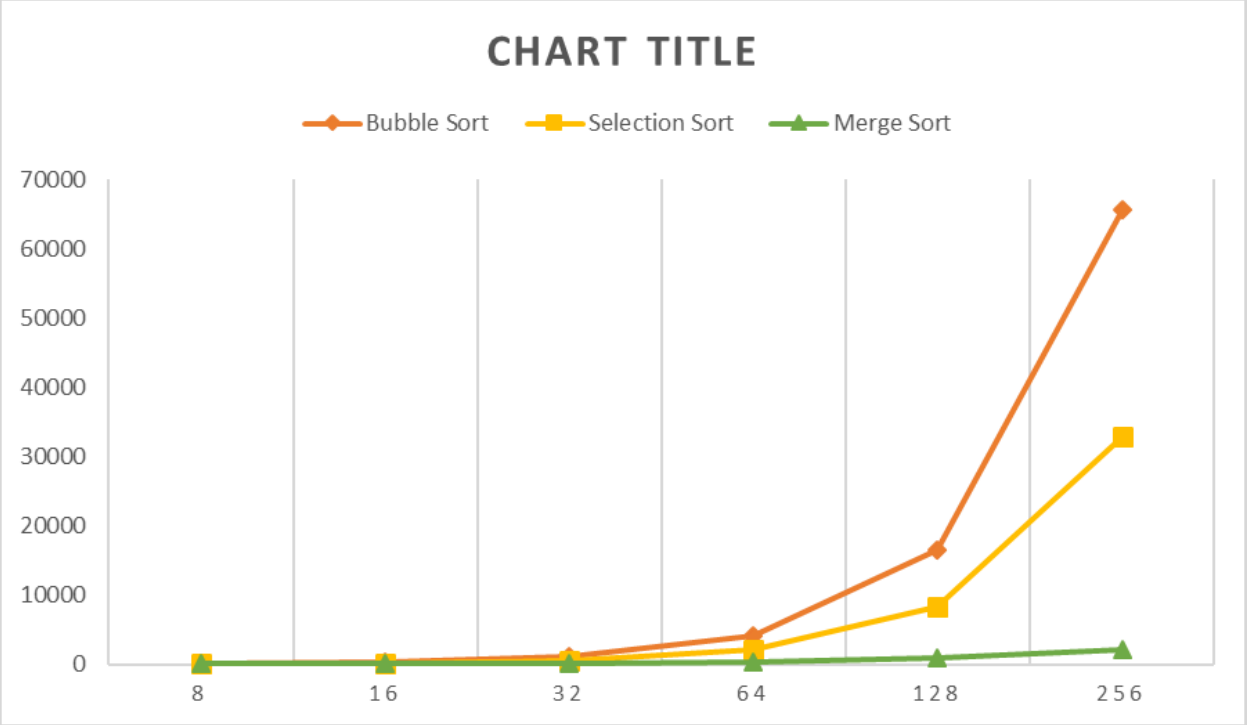
When we say the time/space complexity of an algorithm is proportional to some function of the input length N, suppose N^2. The actual running time or space consumption is a constant multiple of N^2. Suppose that constant is C. Then the running time is actually CN^2. The value of C depends on how we wrote the code keeping the basic logic of our N^2 time algorithm intact, the cost of different operations in the hardware, and the actual number and sizes of the data items we use in the program. Consider the earlier example of linearSearch and let us investigate the exact number of operations CPU will execute in the worst case when the input size is N.

1. Computing length is 1 operation.
2. Incrementing the for-loop index N time is N additions.
3. Comparing the index value with the length of the sequence is N comparisons.
4. Loading the ith element of S in a CPU register is one operation then comparing it with x is another operation. This happens N times in the if statement. So, we have a total of 2N operations.
5. The return statement at the end of the loop is 1 operation.

From the above calculation, we can roughly say that the running time of our linearSearch is equal to 4N + 2 operations. Hence if we have another implementation of linearSearch that supposedly takes 3N + 2 operations, we will call the second implementation better. However, particularly, for time complexity analysis, we hardly ever bother about the constant multiplier difference among implementations. There are several reasons for that. First, the compilers can often optimize the executables to eliminate that level of difference. Second, given different CPU operations take different time, the exact estimates of the constant factor multiplier may be inaccurate. Third, and most importantly, we can easily expect improvement in computer hardware that can overcome constant factor deficiencies.

Actually, for very small input sizes, the difference between a highly efficient and the most inefficient solution of a problem is barely noticeable and hardly matters. The logic is simple, if all implementations complete their run within the time we are willing to wait and use the memory that we can allocate for program executions then why bother. Consequently, we focus on determining whether our program implementations can handle large inputs properly. For that analysis, we analyze how the complexity of an implementation grows as a function of the input.

For example, consider several sorting algorithms. You already learned selection sort and bubble sort in earlier courses. For an input sequence of length n, the running time of selection sort is proportional to $n(n+1)/2$ and that of bubble sort is n^2. A better sorting algorithm is the merge sort (which you will learn later) who's running time is proportional to $n\log_2 n$. Now let us plot these functions for different values of n and see how they grow with the length of the input.



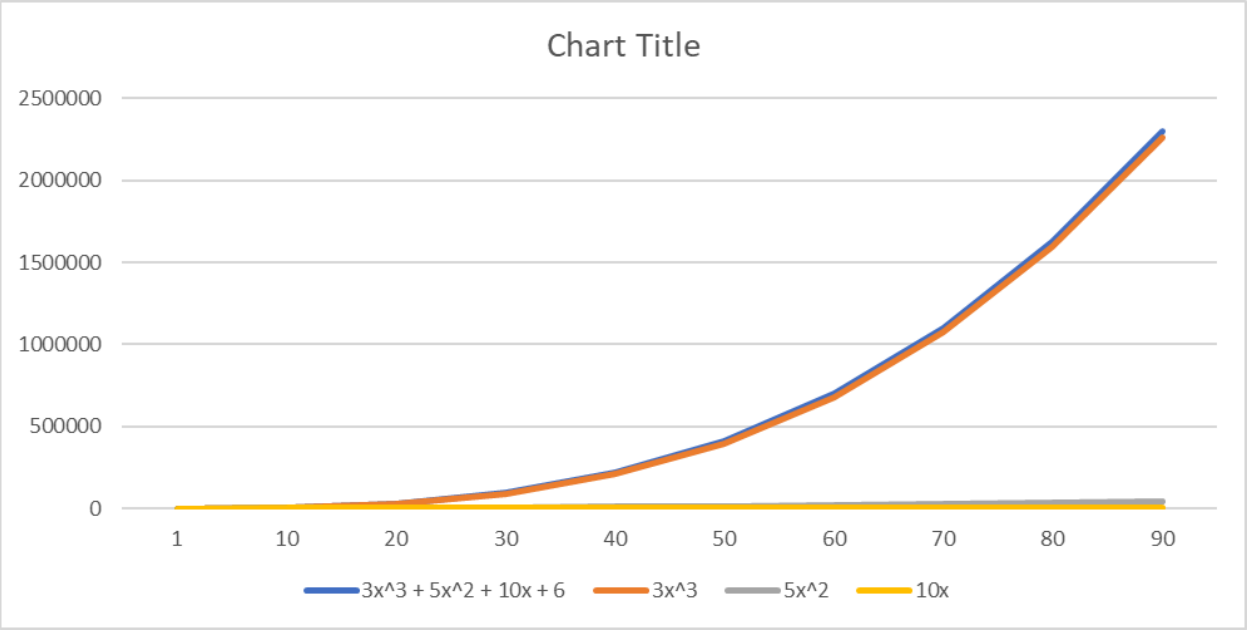
Below are the actual values for their running time.

Input Size	Bubble Sort	Selection Sort	Merge Sort
8	64	28	24
16	256	136	64
32	1024	528	160
64	4096	2080	384
128	16384	8256	896
256	65536	32896	2048
512	262144	131328	4608
1024	1048576	524800	10240

Notice that for the smallest input size, selection sort was nearly as good as the merge sort. However, there is no comparison between them as the input gets larger. Meanwhile, the curves for bubble sort and selection sort follow the same trajectory as input gets larger. Even for input size up to 128, the difference of the three algorithms hardly matters due to the speed of the CPU. Meanwhile, for larger input sizes, the constant factor difference between bubble and selection sorts become irrelevant. So, in general, what matters at the end is what polynomial/logarithmic power of input size N an implementation is. Everything else becomes secondary.

The formal way to express the worst-case running time and space complexity as a function of the input size as the input grows to infinity is to use the big O notation. So, we say time complexity of both bubble and selection sort are $O(n^2)$, that is, order n^2; and the merge sort is $O(n\log_2 n)$, that is, order n log n. These are called asymptotic worst-case time complexities, because they drop all constant multipliers and constant factor addition to the running time. Note that the space complexity for all three sorting algorithms is $O(n)$.

Note that when focusing on the asymptotic time and space complexities of a program implementation, we also drop lower powers of the input sizes if the complexity is a polynomial function of the input. For example, if the time complexity of a program is $an^3 + bn^2 + cn + d$, where n is the input size and a, b, c, and d are constants. Then we say the asymptotic time complexity of the program is $O(n^3)$. The reason for dropping the lower powers of n is the same. For large values of n, the contribution of the lower powers become negligible compared to the largest power in the total running time. You can understand this by just looking at the following example chart.



Finally, if there are multiple inputs or multiple independent attributes of a single input, the asymptotic time and space complexities can be a function of several/all of them. For example, traversing a graph (which you will learn later in this book) having n vertices and m edges has both time and space complexities $O(n + m)$.

Practical Use of Asymptotic Complexity

Remember that when we discuss the asymptotic time and space complexities of our programs, we ignore constant multipliers and constant factor additions to the running time and space requirements. The constant factor additions can be ignored in any analysis. However, the constant coefficients of various power of the input cannot be ignored all the time in real world applications. Thus, the most common use of asymptotic time complexity is in the estimation of the time and space cost of the program for the largest input size our program may need to handle. Then when we go into the implementations, we choose the implementation that is better relative to the others given their asymptotic complexities are reasonable.

This is particularly true for space complexity because if you do not have enough space in the memory for your program to run then you have nothing. While, with time complexity, sometimes you can just wait longer. Consider an example, two program implementations for the same problem taking $C_1 n\log_2 n$ time and $C_2 n\log_2 n$ time. Further assume that your RAM size is 32 Gigabytes. Now if the largest input you have to process is 1/2 Giga entries of 1 byte each. Then $n\log_2 n = 2^{29}*30$ = 15 Gigabytes already. Now if $C_1 = 2$ and $C_2 = 3$. Then you cannot accommodate the largest input in your machine for the second implementation. Your only option is to use the first implementation.

Finally, know that space and time complexities are often two competing aspects of your program. Often you have to spend more time to save space in a program implementation or vice versa. You cannot have the best of the both in many real-world problems.

Exercises:

3 pseudo-codes for time complexity analysis.

3 pseudo-codes for space complexity analysis.

(I think you can find exercises from Keneth Rosen's Discrete mathematics book or Corman's Algorithms book.)

1. Give a big-O estimate for time and space used in this segment of an algorithm.

```
t := 0
for i := 1 to n
   for j := 1 to n
         t := t + i + j
```

2. Give a big-O estimate for the number of operations, where an operation is a comparison or a multiplication, used in this segment of an algorithm (ignoring comparisons used to test the conditions in the for loops, where a_1, a_2, ..., a_n are positive real numbers).

```
m := 0
for i := 1 to n
   for j := i + 1 to n
         m := max(a_i, a_j , m)
```

3. Give time and space complexity of the following algorithm.

Algorithm: Find Maximum Element in an Array

Input: An array of integers, arr

Output: The maximum element in the array

```
Let maxElement = arr[0]
For each element num in arr starting from the second element:
      If num > maxElement,
set maxElement = num
Return maxElement
```

4. Find the space complexity for the following code.

```
function generate_all_subsets(nums):
      all_subsets = [[]]  // Start with an empty set
      for num in nums:
      current_subsets = []
      for subset in all_subsets:
            current_subsets.append(subset  +  [num])    //  Include  the
current element in each subset
            all_subsets.extend(current_subsets)  // Add new subsets to
the result
      return all_subsets
```