Contents

Chapter 1

A Time Complexity Question

A Time Complexity Question - GeeksforGeeks

What is the time complexity of following function fun()? Assume that log(x) returns log value in base 2.

```
 void fun()
{
   int i, j;
   for (i=1; i<=n; i++)
      for (j=1; j<=log(i); j++)
         printf("GeeksforGeeks");
}
```

Time Complexity of the above function can be written as Θ(log 1) + Θ(log 2) + Θ(log 3) + ... + Θ(log n) which is Θ (log n!)

Order of growth of 'log n!' and 'n log n' is same for large values of n, i.e., Θ (log n!) = Θ(n log n). So time complexity of fun() is Θ(n log n).

The expression Θ(log n!) = Θ(n log n) can be easily derived from following Stirling's approximation (or Stirling's formula).

```
      log n! = n log n - n + O(log(n))
```

Sources:
http://en.wikipedia.org/wiki/Stirling%27s_approximation

Source

https://www.geeksforgeeks.org/a-time-complexity-question/

Chapter 2

Advanced master theorem for divide and conquer recurrences

Advanced master theorem for divide and conquer recurrences - GeeksforGeeks

Master Theorem is used to determine running time of algorithms (divide and conquer algorithms) in terms of asymptotic notations.
Consider a problem that be solved using recursion.

```
function f(input x size n)
if(n > k)
solve x directly and return
else
divide x into a subproblems of size n/b
call f recursively to solve each subproblem
Combine the results of all sub-problems
```

The above algorithm divides the problem into **a** subproblems, each of size n/b and solve them recursively to compute the problem and the extra work done for problem is given by f(n), i.e., the time to create the subproblems and combine their results in the above procedure.

So, according to master theorem the runtime of the above algorithm can be expressed as:

```
T(n) = aT(n/b) + f(n)
```

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
f(n) = cost of work done outside the recursive calls like dividing into subproblems and cost of combining them to get the solution.

Not all recurrence relations can be solved with the use of the master theorem i.e. if

- $T(n)$ is not monotone, ex: $T(n) = \sin n$
- $f(n)$ is not a polynomial, ex: $T(n) = 2T(n/2) + 2^n$

This theorem is an advance version of master theorem that can be used to determine running time of divide and conquer algorithms if the recurrence is of the following form :-

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
b > 1, k >= 0 and p is a real number.

Then,

1. if $a > b^k$, then $T(n) = (n^{\log_b a})$
2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = (n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = (n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = (n^{\log_b a})$
3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = (n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = (n^k)$

Time Complexity Analysis –

- **Example-1: Binary Search** – $T(n) = T(n/2) + O(1)$
  $a = 1$, $b = 2$, $k = 0$ and $p = 0$
  $b^k = 1$. So, $a = b^k$ and $p > -1$ [Case 2.(a)]
  $T(n) = (n^{\log_b a} \log^{p+1} n)$
  $T(n) = (\log n)$
- **Example-2: Merge Sort** – $T(n) = 2T(n/2) + O(n)$
  $a = 2$, $b = 2$, $k = 1$, $p = 0$
  $b^k = 1$. So, $a = b^k$ and $p > -1$ [Case 2.(a)]
  $T(n) = (n^{\log_b a} \log^{p+1} n)$
  $T(n) = (n\log n)$

- **Example-3:** $T(n) = 3T(n/2) + n^2$
  $a = 3, b = 2, k = 2, p = 0$
  $b^k = 4$. So, $a < b^k$ and $p = 0$ [Case 3.(a)]
  $T(n) = \ (n^k \log^p n)$
  $T(n) = \ (n^2)$
- **Example-4:** $T(n) = 3T(n/2) + \log^2 n$
  $a = 3, b = 2, k = 0, p = 2$
  $b^k = 1$. So, $a > b^k$ [Case 1]
  $T(n) = \ (n^{\log_b a}\ )$
  $T(n) = \ (n^{\log_2 3})$
- **Example-5:** $T(n) = 2T(n/2) + n\log^2 n$
  $a = 2, b = 2, k = 1, p = 2$
  $b^k = 1$. So, $a = b^k$ [Case 2.(a)]
  $T(n) = \ (n^{\log_b a}\log^{p+1} n\ )$
  $T(n) = \ (n^{\log_2 2}\log^3 n)$
  $T(n) = \ (n\log^3 n)$
- **Example-6:** $T(n) = 2^n T(n/2) + n^n$
  This recurrence can't be solved using above method since function is not of form $T(n) = aT(n/b) + \ (n^k \log^p n)$

GATE Practice questions –

- GATE-CS-2017 (Set 2) | Question 56
- GATE IT 2008 | Question 42
- GATE CS 2009 | Question 35

Source

https://www.geeksforgeeks.org/advanced-master-theorem-for-divide-and-conquer-recurrences/

Chapter 3

Algorithm Practice Question for Beginners | Set 1

Algorithm Practice Question for Beginners | Set 1 - GeeksforGeeks

Consider the following C function.

```
 unsigned fun(unsigned n)
{
    if (n == 0) return 1;
    if (n == 1) return 2;

    return fun(n-1) + fun(n-1);
}
```

Consider the following questions for above code ignoring compiler optimization.
a) What does the above code do?
b) What is the time complexity of above code?
c) Can the time complexity of above function be reduced?

What does fun(n) do?
In the above code, fun(n) is equal to 2*fun(n-1). So the above function returns 2^n. For example, for n = 3, it returns 8, for n = 4, it returns 16.

What is the time complexity of fun(n)?
Time complexity of the above function is exponential. Let the Time complexity be T(n). T(n) can be written as following recurrence. Here C is a machine dependent constant.

```
T(n) = T(n-1) + T(n-1) + C
     = 2T(n-1) + C
```

The above recurrence has solution as $\Theta(2^n)$. We can solve it by recurrence tree method. The recurrence tree would be a binary tree with height n and every level would be completely full except possibly the last level.

```
                  C
              /        \
           C              C
        /     \        /     \
      C        C      C        C
     / \      / \    / \      / \
    .   .    .   .  .   .    .   .
    .   .    .   .  .   .    .   .
       Height of Tree is Θ(n)
```

Can the time complexity of fun(n) be reduced?
A simple way to reduce the time complexity is to make one call instead of 2 calls.

```
 unsigned fun(unsigned n)
{
    if (n == 0) return 1;
    if (n == 1) return 2;

    return 2*fun(n-1);
}
```

Time complexity of the above solution is $\Theta(n)$. T Let the Time complexity be T(n). T(n) can be written as following recurrence. Here C is a machine dependent constant.

```
T(n) = T(n-1) + C
```

We can solve it by recurrence tree method. The recurrence tree would be a skewed binary tree (every internal node has only one child) with height n.

```
           C
          /
         C
        /
       C
      /
     .
   .
 Height of Tree is Θ(n)
```

The above function can be further optimized using divide and conquer technique to calculate powers.

```
 unsigned fun(unsigned n)
{
    if (n == 0) return 1;
    if (n == 1) return 2;
    unsigned x = fun(n/2);
    return (n%2)? 2*x*x: x*x;
}
```

Time complexity of the above solution is Θ(Logn). Let the Time complexity be T(n). T(n) can be approximately written as following recurrence. Here C is a machine dependent constant.

```
T(n) = T(n/2) + C
```

We can solve it by recurrence tree method. The recurrence tree would be a skewed binary tree (every internal node has only one child) with height Log(n).

```
            C
           /
          C
         /
        C
       /
      .
    .
 Height of Tree is Θ(Logn)
```

We can also directly compute fun(n) using bitwise left shift operator '<

```
unsigned fun(unsigned n)
{
   return 1 << n;
}
```

Source

https://www.geeksforgeeks.org/algorithm-practice-question-beginners-set-1/

This article is contributed by **Kartik**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Chapter 4

Amortized analysis for increment in counter

Amortized analysis for increment in counter - GeeksforGeeks

Amortized analysis refers to determining the time-averaged running time for a sequence (not an individual) operation. It is different from average case analysis because here, we don't assumes that the data arranged in average (not very bad) fashion like we do for average case analysis for quick sort. That is, amortized analysis is worst case analysis but for a sequence of operation rather than individual one. It applies to the method that consists for the sequence of operation, where vast majority of operation is cheap but some of the operation are expensive. This can be visualized with the help of binary counter which is implemented below.

Let's see this by implementing a increment counter in C. First, let's see how counter increment works.
Let a variable **i** contains a value 0 and we performs i++ many time. Since, on hardware every operation is performed in binary form. Let binary number stored in 8 bit. So, value is 00000000. Let's increment many time. So, the pattern we finds are as :

00000000, 00000001, 00000010, 00000011, 00000100, 00000101, 00000110, 00000111, 00001000 and so on …..

Steps :
1. Iterate from rightmost and make all one to zero until finds first zero.
2. After iteration, if index is greater than or equal to zero, then make zero lie on that position to one.

```
 #include <bits / stdc++.h>
using namespace std;

int main()
{
    char str[] = "10010111";
    int length = strlen(str);
    int i = length - 1;
    while (str[i] == '1') {
        str[i] = '0';
        i--;
    }
    if (i >= 0)
        str[i] = '1';
    printf("% s", str);
}
```

Output:

```
10011000
```

On a simple look on program or algorithm, its running cost looks proportional to the number of bit but in real, it is not proportional to number of bit. Let's see how !

Let's assume that increment operation is performed k time. We see that in every increment, its rightmost bit is getting flipped. So, number of flipping for LSB is k. For, second rightmost is flipped after a gap, i.e., 1 time in 2 increment. 3rd rightmost – 1 time in 4 increment. 4th rightmost – 1 time in 8 increment. So, number of flipping is k/2 for 2nd rightmost bit, k/4 for 3rd rightmost bit, k/8 for 4th rightmost bit and so on …

Total cost will be the total number of flipping, that is,
C(k) = k + k/2 + k/4 + k/8 + k/16 + …… which is Geometric Progression series and also,
C(k) < k + k/2 + k/4 + k/8 + k/16 + k/32 + …… up to infinity
So, C(k) < k/(1-1/2)
and so, C(k) < 2k
So, C(k)/k < 2
Hence, we find that average cost for increment a counter for one time is constant and it does not depend on the number of bit. We conclude that increment of a counter is constant cost operation.

Refernces :

1. http://www.cs.cornell.edu/courses/cs3110/2013sp/supplemental/recitations/rec21.html
2. http://faculty.cs.tamu.edu/klappi/csce411-s17/csce411-amortized3.pdf

Source

https://www.geeksforgeeks.org/amortized-analysis-increment-counter/

Chapter 5

An interesting time complexity question

An interesting time complexity question - GeeksforGeeks

What is the time complexity of following function fun()?

```
 int fun(int n)
{
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j < n; j += i)
        {
            // Some O(1) task
        }
    }
}
```

For i = 1, the inner loop is executed n times.
For i = 2, the inner loop is executed approximately n/2 times.
For i = 3, the inner loop is executed approximately n/3 times.
For i = 4, the inner loop is executed approximately n/4 times.
…………………………………………………….
…………………………………………………….
For i = n, the inner loop is executed approximately n/n times.

So the total time complexity of the above algorithm is (n + n/2 + n/3 + … + n/n)

Which becomes n * (1/1 + 1/2 + 1/3 + … + 1/n)

The important thing about series (1/1 + 1/2 + 1/3 + … + 1/n) is, it is equal to Θ(Logn) (See thisfor reference). So the time complexity of the above code is Θ(nLogn).

As a side note, the sum of infinite harmonic series is counterintuitive as the series diverges. The value of $\sum_{n=1}^{\infty} \frac{1}{n}$ is ∞. This is unlike geometric series as geometric series with ratio less than 1 converges.

Reference:
http://en.wikipedia.org/wiki/Harmonic_series_%28mathematics%29#Rate_of_divergence
http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap03.htm

This article is contributed by **Rahul**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/interesting-time-complexity-question/

Chapter 6

Analysis of Algorithm | Set 4 (Solving Recurrences)

Analysis of Algorithm | Set 4 (Solving Recurrences) - GeeksforGeeks

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

There are mainly three ways for solving recurrences.

1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect.

```
For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction
to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true
for values smaller than n.

T(n) = 2T(n/2) + n
    <= cn/2Log(n/2) + n
    =  cnLogn - cnLog2 + n
    =  cnLogn - cn + n
    <= cnLogn
```

2) Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

```
For example consider the recurrence relation
T(n) = T(n/4) + T(n/2) + cn2

           cn2
         /      \
     T(n/4)     T(n/2)

If we further break down the expression T(n/4) and T(n/2),
we get following recursion tree.

                cn2
           /           \
       c(n2)/16      c(n2)/4
      /      \          /     \
  T(n/16)     T(n/8)  T(n/8)    T(n/4)
Breaking down further gives us following
                 cn2
            /            \
       c(n2)/16          c(n2)/4
       /      \            /      \
c(n2)/256   c(n2)/64  c(n2)/64    c(n2)/16
 /    \      /    \    /    \       /    \

To know the value of T(n), we need to calculate sum of tree
nodes level by level. If we sum the above tree level by level,
we get the following series
T(n)  = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ....
The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.
We get the sum as (n2)/(1 - 5/16) which is O(n2)
```

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

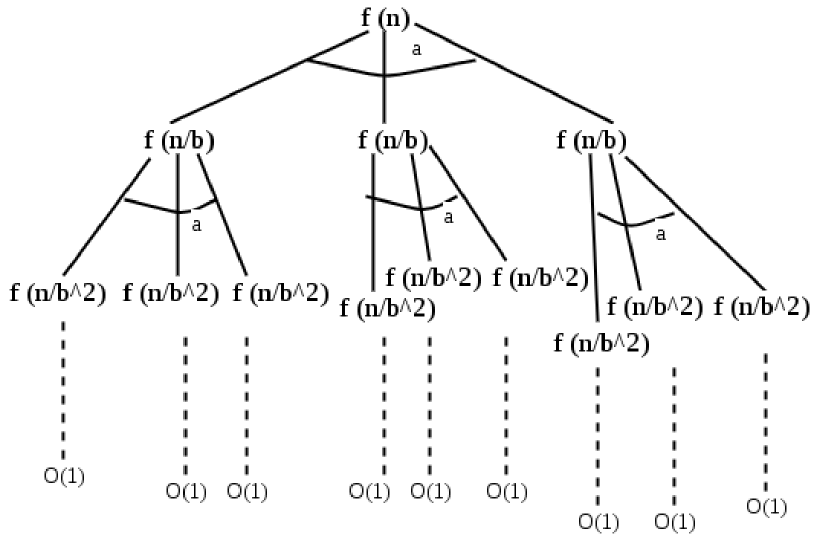
```
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
```

There are following three cases:
1. If $f(n) = \Theta(n^c)$ where $c < \mathrm{Log}_b a$ then $T(n) = \Theta(n^{\mathrm{Log}_b a})$

2. If $f(n) = \Theta(n^c)$ where $c = \mathrm{Log}_b a$ then $T(n) = \Theta(n^c \mathrm{Log}\ n)$

3.If $f(n) = \Theta(n^c)$ where $c > \mathrm{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?
Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method
Merge Sort: T(n) = 2T(n/2) + Θ(n). It falls in case 2 as c is 1 and Log_ba] is also 1. So the solution is Θ(n Logn)

Binary Search: T(n) = T(n/2) + Θ(1). It also falls in case 2 as c is 0 and Log_ba is also 0. So the solution is Θ(Logn)

Notes:
1) It is not necessary that a recurrence of the form T(n) = aT(n/b) + f(n) can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence T(n) = 2T(n/2) + n/Logn cannot be solved using master method.

2) Case 2 can be extended for $f(n) = \Theta(n^c \text{Log}^k n)$
If $f(n) = \Theta(n^c \text{Log}^k n)$ for some constant k >= 0 and c = Log_ba, then $T(n) = \Theta(n^c \text{Log}^{k+1} n)$

Practice Problems and Solutions on Master Theorem.

Next – Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)

References:
http://en.wikipedia.org/wiki/Master_theorem
MIT Video Lecture on Asymptotic Notation | Recurrences | Substitution, Master Method
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Source

https://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/

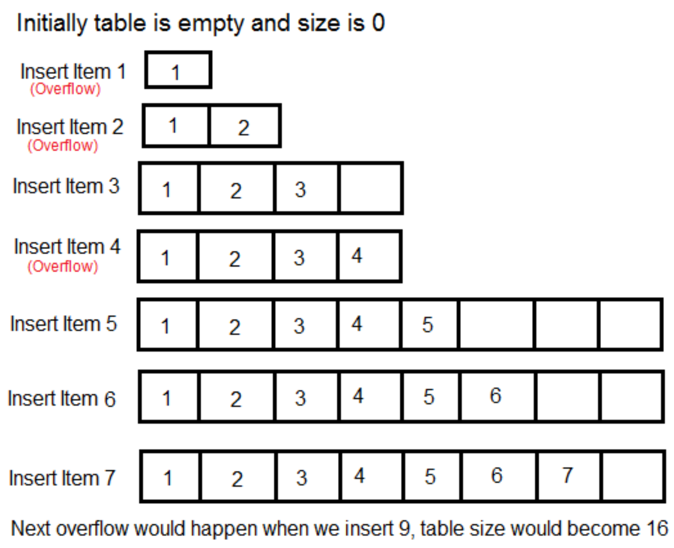
Chapter 7

Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)

Analysis of Algorithm | Set 5 (Amortized Analysis Introduction) - GeeksforGeeks

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.
The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

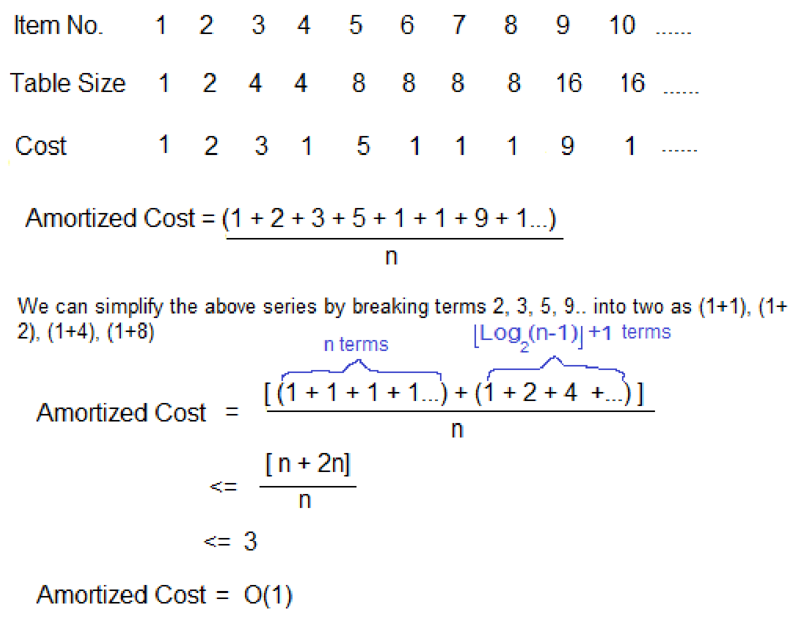
Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.



The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.
1) Allocate memory for a larger table of size, typically twice the old table.
2) Copy the contents of old table to new table.
3) Free the old table.

If the table has space available, we simply insert new item in available space.

What is the time complexity of n insertions using the above scheme?
If we use simple analysis, the worst case cost of an insertion is O(n). Therefore, worst case cost of n inserts is n * O(n) which is O(n^2). This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take Θ(n) time.



Item No.	1	2	3	4	5	6	7	8	9	10	......
Table Size	1	2	4	4	8	8	8	8	16	16	......
Cost	1	2	3	1	5	1	1	1	9	1	......

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1\ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1\ldots)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + \ldots)}^{\lfloor \text{Log}_2(n-1)\rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of dynamic table is used in vectors in C++, ArrayList in Java.

Following are few important notes.
1) Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

2) The above Amortized Analysis done for Dynamic Array example is called ***Aggregate Method***. There are two more powerful ways to do Amortized analysis called ***Accounting Method*** and ***Potential Method***. We will be discussing the other two methods in separate posts.

3) The amortized analysis doesn't involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

Sources:
Berkeley Lecture 35: Amortized Analysis
MIT Lecture 13: Amortized Algorithms, Table Doubling, Potential Method
http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec20-amortized/amortized.htm

Source

https://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/

Chapter 8

Analysis of Algorithms | Big-O analysis

Analysis of Algorithms | Big-O analysis - GeeksforGeeks

In our previous articles on Analysis of Algorithms, we had discussed asymptotic notations, their worst and best case performance etc. in brief. In this article, we discuss analysis of algorithm using Big – O asymptotic notation in complete details.

Big-O Analysis of Algorithms

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

The Big-O Asymptotic Notation gives us the Upper Bound Idea, mathematically described below:

> f(n) = O(g(n)) if there exists a positive integer n_0 and a positive constant c, such that f(n) c.g(n)   n n_0

The general step wise procedure for Big-O runtime analysis is as follows:

1. Figure out what the input is and what n represents.
2. Express the maximum number of operations, the algorithm performs in terms of n.
3. Eliminate all excluding the highest order terms.
4. Remove all the constant factors.

Some of the useful properties on Big-O notation analysis are as follow:

> Constant Multiplication:
> If f(n) = c.g(n), then O(f(n)) = O(g(n)) ; where c is a nonzero constant.

Polynomial Function:
If $f(n) = a_0 + a_1.n + a_2.n^2 +$ —- $+ a_m.n^m$, then $O(f(n)) = O(n^m)$.
Summation Function:
If $f(n) = f_1(n) + f_2(n) +$ —- $+ f_m(n)$ and $f_i(n)$ $f_{i+1}(n)$ i=1, 2, —-, m,
then $O(f(n)) = O(\max(f_1(n), f_2(n),$ —-, $f_m(n)))$.
Logarithmic Function:
If $f(n) = \log_a n$ and $g(n)=\log_b n$, then $O(f(n))=O(g(n))$
; all log functions grow in the same manner in terms of Big-O.

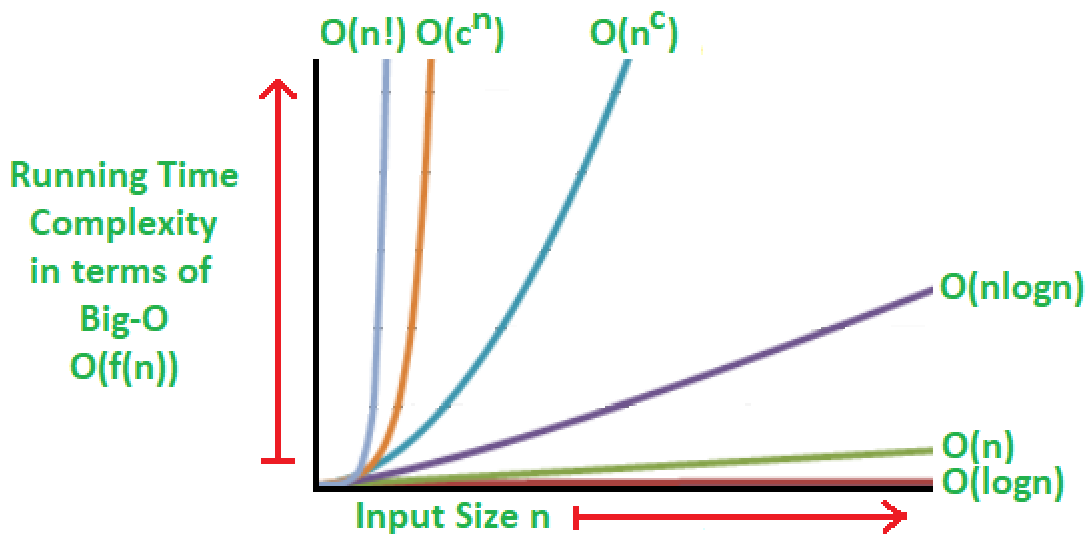
Basically, this asymptotic notation is used to measure and compare the worst-case scenarios of algorithms theoretically. For any algorithm, the Big-O analysis should be straightforward as long as we correctly identify the operations that are dependent on n, the input size.

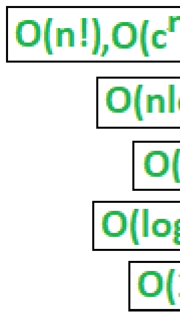
Runtime Analysis of Algorithms

In general cases, we mainly used to measure and compare the worst-case theoretical running time complexities of algorithms for the performance analysis.
The fastest possible running time for any algorithm is O(1), commonly referred to as *Constant Running Time*. In this case, the algorithm always takes the same amount of time to execute, regardless of the input size. This is the ideal runtime for an algorithm, but it's rarely achievable.
In actual cases, the performance (Runtime) of an algorithm depends on n, that is the size of the input or the number of operations is required for each input item.
The algorithms can be classified as follows from the best-to-worst performance (Running Time Complexity):

A logarithmic algorithm – O(logn)
Runtime grows logarithmically in proportion to n.
A linear algorithm – O(n)
Runtime grows directly in proportion to n.
A superlinear algorithm – O(nlogn)
Runtime grows in proportion to n.
A polynomial algorithm – $O(n^c)$
Runtime grows quicker than previous all based on n.
A exponential algorithm – $O(c^n)$
Runtime grows even faster than polynomial algorithm based on n.
A factorial algorithm – O(n!)
Runtime grows the fastest and becomes quickly unusable for even small values of n.

Where, n is the input size and c is a positive constant.





Algorithmic Examples of Runtime Analysis:
Some of the examples of all those types of algorithms (in worst-case scenarios) are mentioned below:

Logarithmic algorithm – O(logn) – Binary Search.
Linear algorithm – O(n) – Linear Search.
Superlinear algorithm – O(nlogn) – Heap Sort, Merge Sort.
Polynomial algorithm – O(n^c) – Strassen's Matrix Multiplication, Bubble Sort, Selection Sort, Insertion Sort, Bucket Sort.
Exponential algorithm – O(c^n) – Tower of Hanoi.
Factorial algorithm – O(n!) – Determinant Expansion by Minors, Brute force Search algorithm for Traveling Salesman Problem.

Mathematical Examples of Runtime Analysis:
The performances (Runtimes) of different orders of algorithms separate rapidly as n (the input size) gets larger. Let's consider the mathematical example:

```
If n = 10,                   If n=20,
    log(10) = 1;                 log(20) = 2.996;
    10 = 10;                     20 = 20;
    10log(10)=10;                20log(20)=59.9;
    102=100;                     202=400;
    210=1024;                     220=1048576;
    10!=3628800;                 20!=2.432902e+1818;
```

Memory Footprint Analysis of Algorithms

For performance analysis of an algorithm, runtime measurement is not only relevant metric but also we need to consider the memory usage amount of the program. This is referred to

as the Memory Footprint of the algorithm, shortly known as Space Complexity.
Here also, we need to measure and compare the worst case theoretical space complexities of algorithms for the performance analysis.

It basically depends on two major aspects described below:

- Firstly, the implementation of the program is responsible for memory usage. For example, we can assume that recursive implementation always reserves more memory than the corresponding iterative implementation of a particular problem.
- And the other one is n, the input size or the amount of storage required for each item. For example, a simple algorithm with a high amount of input size can consume more memory than a complex algorithm with less amount of input size.

Algorithmic Examples of Memory Footprint Analysis: The algorithms with examples are classified from the best-to-worst performance (Space Complexity) based on the worst-case scenarios are mentioned below:

```
Ideal algorithm - O(1) - Linear Search, Binary Search,
  Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Shell Sort.
Logarithmic algorithm - O(log n) - Merge Sort.
Linear algorithm - O(n) - Quick Sort.
Sub-linear algorithm - O(n+k) - Radix Sort.
```

Space-Time Tradeoff and Efficiency

There is usually a trade-off between optimal memory use and runtime performance.
In general for an algorithm, space efficiency and time efficiency reach at two opposite ends and each point in between them has a certain time and space efficiency. So, the more time efficiency you have, the less space efficiency you have and vice versa.
For example, Mergesort algorithm is exceedingly fast but requires a lot of space to do the operations. On the other side, Bubble Sort is exceedingly slow but requires the minimum space.

At the end of this topic, we can conclude that finding an algorithm that works in less running time and also having less requirement of memory space, can make a huge difference in how well an algorithm performs.

Improved By : vitiral

Source

https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

Chapter 9

Analysis of Algorithms | Set 1 (Asymptotic Analysis)

Analysis of Algorithms | Set 1 (Asymptotic Analysis) - GeeksforGeeks

Why performance analysis?
There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?
The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun!
To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it. If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

Given two algorithms for a task, how do we find out which one is better?
One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.
1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.
For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic). To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on

a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Does Asymptotic Analysis always work?
Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is nLogn). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.
Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Next – Analysis of Algorithms | Set 2 (Worst, Average and Best Cases)

References:
MIT's Video lecture 1 on Introduction to Algorithms.

Improved By : Danail Kozhuharov

Source

https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/

Chapter 10

Analysis of Algorithms | Set 2 (Worst, Average and Best Cases)

Analysis of Algorithms | Set 2 (Worst, Average and Best Cases) - GeeksforGeeks

In the previous post, we discussed how Asymptotic analysis overcomes the problems of naive way of analyzing algorithms. In this post, we will take an example of Linear Search and analyze it using Asymptotic analysis.

We can have three cases to analyze an algorithm:
1) Worst Case
2) Average Case
3) Best Case

Let us consider the following implementation of Linear Search.

```
 #include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
       if (arr[i] == x)
         return i;
    }
    return -1;
}
```

```
/* Driver program to test above functions*/
int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

Worst Case Analysis (Usually Done)
In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ(n).

Average Case Analysis (Sometimes done)
In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

```
Average Case Time =

                  =

                  = Θ(n)
```

Best Case Analysis (Bogus)
In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1)
Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information. The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example,Merge Sort. Merge Sort does Θ(nLogn) operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

Next – Analysis of Algorithms | Set 3 (Asymptotic Notations)

References:

MIT's Video lecture 1 on Introduction to Algorithms.

Source

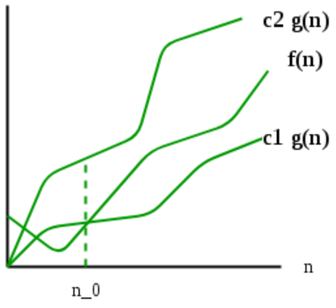
https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/

Chapter 11

Analysis of Algorithms | Set 3 (Asymptotic Notations)

Analysis of Algorithms | Set 3 (Asymptotic Notations) - GeeksforGeeks

We have discussed Asymptotic Analysis, andWorst, Average and Best Cases of Algorithms. The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

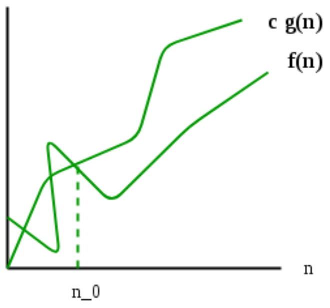


f(n) = theta(g(n))

1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.
$3n^3 + 6n^2 + 6000 = \Theta(n^3)$
Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ has higher values than $\Theta n^2)$ irrespective of the constants involved.
For a given function g(n), we denote Θ(g(n)) is following set of functions.

```
Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such
                 that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}
```

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

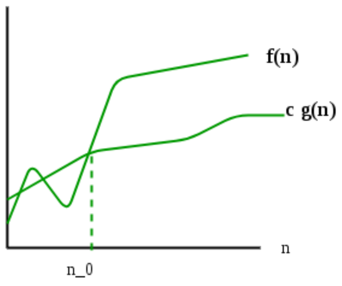


f(n) = O(g(n))

2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.
If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

```
O(g(n)) = { f(n): there exist positive constants c and
                  n0 such that 0 <= f(n) <= cg(n) for
                  all n >= n0}
```



f(n) = Omega(g(n))

3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation< can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

```
Ω (g(n)) = {f(n): there exist positive constants c and
n0 such that 0 <= cg(n) <= f(n) for
all n >= n0}.
```

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

Exercise:
Which of the following statements is/are valid?
1. Time Complexity of QuickSort is Θ(n^2)
2. Time Complexity of QuickSort is O(n^2)
3. For any two functions f(n) and g(n), we have f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).
4. Time complexity of all computer algorithms can be written as Ω(1)

Important Links :

- There are two more notations called **little o and little omega**. Little o provides strict upper bound (equality condition is removed from Big O) and little omega provides strict lower bound (equality condition removed from big omega)
- Analysis of Algorithms | Set 4 (Analysis of Loops)
- Recent Articles on analysis of algorithm.

References:
Lec 1 | MIT (Introduction to Algorithms)

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/

Chapter 12

Analysis of Algorithms | Set 4 (Analysis of Loops)

Analysis of Algorithms | Set 4 (Analysis of Loops) - GeeksforGeeks

We have discussed Asymptotic Analysis, Worst, Average and Best Cases and Asymptotic Notations in previous posts. In this post, analysis of iterative programs with simple examples is discussed.

1) O(1): Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

```
// set of non-recursive and non-loop statements
```

For example swap() function has O(1) time complexity.
A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
     // some O(1) expressions
}
```

2) O(n): Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}
```

```
for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

3) $O(n^c)$: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have $O(n^2)$ time complexity

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
       // some O(1) expressions
    }
}

for (int i = n; i > 0; i -= c) {
    for (int j = i+1; j <=n; j += c) {
       // some O(1) expressions
}
```

For example Selection sort and Insertion Sort have $O(n^2)$ time complexity.
4) O(Logn) Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

For example Binary Search(refer iterative implementation) has O(Logn) time complexity. Let us see mathematically how it is O(Log n). The series that we get in first loop is 1, c, c^2, c^3, … c^k. If we put k equals to Log_cn, we get $c^{\text{Log}_c n}$ which is n.
5) O(LogLogn) Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i)) {
    // some O(1) expressions
}
```

See thisfor mathematical details.
How to combine time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c) {
     // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
     // some O(1) expressions
}
Time complexity of above code is O(m) + O(n) which is O(m+n)
If m == n, the time complexity becomes O(2n) which is O(n).
```

How to calculate time complexity when there are many if, else statements inside loops?
As discussed here, worst case time complexity is the most useful among best, average and worst. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.
For example consider the linear search function where we consider the case when element is present at the end or not present at all.
When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.
How to calculate time complexity of recursive functions?
Time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences. We will soon be discussing recurrence solving techniques as a separate post.

Quiz on Analysis of Algorithms

Next – Analysis of Algorithm | Set 4 (Solving Recurrences)

Source

https://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/

Chapter 13

Analysis of Algorithms | Set 5 (Practice Problems)

Analysis of Algorithms | Set 5 (Practice Problems) - GeeksforGeeks

We have discussed Asymptotic Analysis, Worst, Average and Best Cases, Asymptotic Notations and Analysis of loops in previous posts.

In this post, practice problems on analysis of algorithms are discussed.

Problem 1: Find the complexity of below recurrence:

```
         { 3T(n-1), if n>0,
T(n) =   { 1, otherwise
```

Solution:

```
Let us solve using substitution.
T(n) = 3T(n-1)
     = 3(3T(n-2))
     = 32T(n-2)
     = 33T(n-3)
       ...
       ...
     = 3nT(n-n)
     = 3nT(0)
     = 3n
This clearly shows that the complexity
of this function is O(3n).
```

Problem 2: Find the complexity of the recurrence:

```
         { 2T(n-1) - 1, if n>0,
T(n) =    { 1, otherwise
```

Solution:

```
 Let us try solving this function with substitution.
T(n) = 2T(n-1) - 1
      = 2(2T(n-2)-1)-1
      = 22(T(n-2)) - 2 - 1
      = 22(2T(n-3)-1) - 2 - 1
      = 23T(n-3) - 22 - 21 - 20
        .....
        .....
      = 2nT(n-n) - 2n-1 - 2n-2 - 2n-3
        ..... 22 - 21 - 20

      = 2n - 2n-1 - 2n-2 - 2n-3
        ..... 22 - 21 - 20
      = 2n - (2n-1)
[Note: 2n-1 + 2n-2 + ...... +  20 = 2n ]
T(n) = 1
Time Complexity is O(1). Note that while
the recurrence relation looks exponential
the solution to the recurrence relation
here gives a different result.
```

Problem 3: Find the complexity of the below program:

```
 function(int n)
{
    if (n==1)
       return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

Solution: Consider the comments in the following function.

```
 function(int n)
{
    if (n==1)
       return;
    for (int i=1; i<=n; i++)
    {
        // Inner loop executes only one
        // time due to break statement.
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

Time Complexity of the above function O(n). Even though the inner loop is bounded by n, but due to break statement it is executing only once.

Problem 4: Find the complexity of the below program:

```
 void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j<=n; j = 2 * j)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Solution: Consider the comments in the following function.

```
 void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)

        // Executes O(Log n) times
        for (int j=1; j<=n; j = 2 * j)

            // Executes O(Log n) times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Time Complexity of the above function O(n $\log^2$n).

Problem 5: Find the complexity of the below program:

```
 void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j+n/2<=n; j = j++)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Solution: Consider the comments in the following function.

```
 void function(int n)
{
    int count = 0;

    // outer loop executes n/2 times
    for (int i=n/2; i<=n; i++)

        // middle loop executes  n/2 times
        for (int j=1; j+n/2<=n; j = j++)

            // inner loop executes logn times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Time Complexity of the above function O(n^2logn).

Problem 6: Find the complexity of the below program:

```
 void function(int n)
{
    int i = 1, s =1;
    while (s <= n)
    {
        i++;
        s += i;
        printf("*");
    }
}
```

Solution: We can define the terms 's' according to relation $s_i = s_{i-1} + i$. The value of 'i' increases by one for each iteration. The value contained in 's' at the i^{th} iteration is the sum of the first 'i' positive integers. If k is total number of iterations taken by the program, then while loop terminates if: 1 + 2 + 3 ….+ k = [k(k+1)/2] > n So k = O(√n).

Time Complexity of the above function O(√n).

Problem 7: Find a tight upper bound on complexity of the below program:

```
 void function(int n)
{
    int count = 0;
    for (int i=0; i<n; i++)
        for (int j=i; j< i*i; j++)
            if (j%i == 0)
            {
                for (int k=0; k<j; k++)
                    printf("*");
            }
}
```

Solution: Consider the comments in the following function.

```
 void function(int n)
{
    int count = 0;

    // executes n times
    for (int i=0; i<n; i++)

        // executes O(n*n) times.
        for (int j=i; j< i*i; j++)
            if (j%i == 0)
            {
                // executes j times = O(n*n) times
                for (int k=0; k<j; k++)
                    printf("*");
            }
}
```

Time Complexity of the above function O(n^5).

Source

https://www.geeksforgeeks.org/analysis-algorithms-set-5-practice-problems/

Chapter 14

Analysis of algorithms | little o and little omega notations

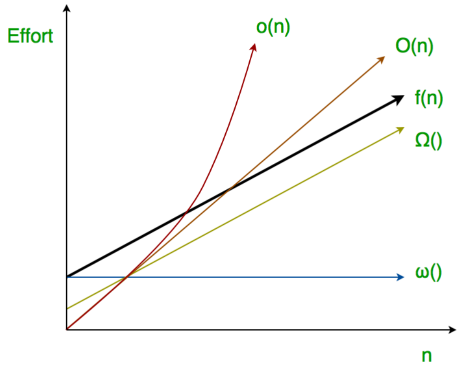
Analysis of algorithms | little o and little omega notations - GeeksforGeeks

The main idea of **asymptotic analysis** is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, mainly because this analysis doesn't require algorithms to be implemented and time taken by programs to be compared. We have already discussed Three main asymptotic notations. The following 2 more asymptotic notations are used to represent time complexity of algorithms.

Little asymptotic notation

Big-O is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function f(n)), even though, as written, it can also be a loose upper-bound. "Little- " (()) notation is used to describe an upper-bound that cannot be tight.

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is (g(n)) (or f(n) E (g(n))) if for **any real** constant c > 0, there exists an integer constant n0 1 such that f(n) 0.



Its means little o() means **loose upper-bound** of f(n).

In mathematical relation,
f(n) = o(g(n)) means
lim f(n)/g(n) = 0
n→∞

Examples:

Is 7n + 8 o($\mathbf{n^2}$)?
In order for that to be true, for any c, we have to be able to find an n0 that makes
f(n) < c * g(n) asymptotically true.
lets took some example,
If c = 100,we check the inequality is clearly true. If c = 1/100 , we'll have to use
a little more imagination, but we'll be able to find an n0. (Try n0 = 1000.) From
these examples, the conjecture appears to be correct.
then check limits,
lim f(n)/g(n) = lim (7n + 8)/(n^2) = lim 7/2n = 0 (l'hospital)
n→∞ n→∞ n→∞

hence 7n + 8 o(n^2)

Little asymptotic notation

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is (g(n)) (or f(n) (g(n))) if for any real constant c > 0, there exists an integer constant n0 1 such that f(n) > c * g(n) 0 for every integer n n0.

f(n) has a higher growth rate than g(n) so main difference between Big Omega (Ω) and little omega ( ) lies in their definitions.In the case of Big Omega f(n)=Ω(g(n)) and the bound is 0<=cg(n)0, but in case of little omega, it is true for all constant c>0.

we use notation to denote a lower bound that is not asymptotically tight.
and, f(n) (g(n)) if and only if g(n) ((f(n)).

In mathematical relation,
if f(n) (g(n)) then,

lim f(n)/g(n) = ∞
n→∞

Example:
Prove that 4n + 6 (1);
the little omega( ) running time can be proven by applying limit formula given below.
if lim f(n)/g(n) = ∞ then functions f(n) is (g(n))
n→∞
here,we have functions f(n)=4n+6 and g(n)=1
lim (4n+6)/(1) = ∞
n→∞
and,also for any c we can get n0 for this inequality 0 <= c*g(n) < f(n), 0 <= c*1 < 4n+6
Hence proved.

References :
Introduction to algorithems

Source

https://www.geeksforgeeks.org/analysis-of-algorithems-little-o-and-little-omega-notations/

Chapter 15

Analysis of different sorting techniques

Analysis of different sorting techniques - GeeksforGeeks

In this article, we will discuss important properties of different sorting techniques including their complexity, stability and memory constraints. Before understanding this article, you should understand basics of different sorting techniques (See : Sorting Techniques).

Time complexity Analysis –
We have discussed best, average and worst case complexity of different sorting techniques with possible scenarios.

Comparison based sorting –
In comparison based sorting, elements of array are compared with each other to find the sorted array.

- **Bubble sort and Insertion sort** –
  Average and worst case time complexity: n^2
  Best case time complexity: n when array is already sorted.
- **Selection sort** –
  Best, average and worst case time complexity: n^2 which is independent of distribution of data.
- **Merge sort** –
  Best, average and worst case time complexity: nlogn which is independent of distribution of data.
- **Heap sort** –
  Best, average and worst case time complexity: nlogn which is independent of distribution of data.
- **Quick sort** –
  It is a divide and conquer approach with recurrence relation:

```
 T(n) = T(k) + T(n-k-1) + cn
```

Worst case: when the array is sorted or reverse sorted, the partition algorithm divides the array in two subarrays with 0 and n-1 elements. Therefore,

```
T(n) = T(0) + T(n-1) + cn
Solving this we get, T(n) = O(n^2)
```

Best case and Average case: On an average, the partition algorithm divides the array in two subarrays with equal size. Therefore,

```
T(n) = 2T(n/2) + cn
Solving this we get, T(n) = O(nlogn)
```

Non-comparison based sorting –
In non-comparison based sorting, elements of array are not compared with each other to find the sorted array.

- **Radix sort** –
  Best, average and worst case time complexity: nk where k is the maximum number of digits in elements of array.
- **Count sort** –
  Best, average and worst case time complexity: n+k where k is the size of count array.
- **Bucket sort** –
  Best and average time complexity: n+k where k is the number of buckets.
  Worst case time complexity: n^2 if all elements belong to same bucket.

In-place/Outplace technique –
A sorting technique is inplace if it does not use any extra memory to sort the array.
Among the comparison based techniques discussed, only merge sort is outplace technique as it requires extra array to merge the sorted subarrays.
Among the non-comparison based techniques discussed, all are outplace techniques. Counting sort uses counting array and bucket sort uses hash table for sorting the array.

Stable/Unstable technique –
A sorting technique is stable if it does not change the order of element with same value.
Out of comparison based techniques, bubble sort, insertion sort and merge sort are stable techniques. Selection sort is unstable as it may change the order of elements with same value. For example, consider the array 4, 4, 1, 3.

In first iteration, minimum element found is 1 and it is swapped with 4 at 0th position. Therefore, order of 4 with respect to 4 at 1st position will change. Similarly, quick sort and heap sort are also unstable.

Out of non-comparison based techniques, Counting sort and Bucket sort are stable sorting techniques whereas radix sort stability depends on underlying algorithm used for sorting.

Analysis of sorting techniques :

- When the array is almost sorted, insertion sort can be preferred.
- When order of input is not known, merge sort is preferred as it has worst case time complexity of nlogn and it is stable as well.
- When the array is sorted, insertion and bubble sort gives complexity of n but quick sort gives complexity of n^2.

Que – 1. Which sorting algorithm will take least time when all elements of input array are identical? Consider typical implementations of sorting algorithms.
(A) Insertion Sort
(B) Heap Sort
(C) Merge Sort
(D) Selection Sort

Solution: As discussed, insertion sort will have complexity of n when input array is already sorted.

Que – 2. Consider the Quicksort algorithm. Suppose there is a procedure for finding a pivot element which splits the list into two sub-lists each of which contains at least one-fifth of the elements. Let T(n) be the number of comparisons required to sort n elements. Then, (GATE-CS-2012)
(A) T(n) <= 2T(n/5) + n
(B) T(n) <= T(n/5) + T(4n/5) + n
(C) T(n) <= 2T(4n/5) + n
(D) T(n) <= 2T(n/2) + n

Solution: The complexity of quick sort can be written as:

```
T(n) = T(k) + T(n-k-1) + cn
```

As given in question, one list contains 1/5th of total elements. Therefore, another list will have 4/5 of total elements. Putting values, we get:

T(n) = T(n/5) + T(4n/5) + cn, which matches option (B).

Source

https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/

Chapter 16

Applications of Hashing

Applications of Hashing - GeeksforGeeks

In this article we will be discussing of applications of hashing.

Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, distinct elements, counting frequencies of items, finding duplicates, etc.

There are many other applications of hashing, including modern day cryptography hash functions. Some of these applications are listed below:

- Message Digest
- Password Verification
- Data Structures(Programming Languages)
- Compiler Operation
- Rabin-Karp Algotithm
- Linking File name and path together

Message Digest:
This is an application of cryptographic Hash Functions. Cryptographic hash functions are the functions which produce an output from which reaching the input is close to impossible. This property of hash functions is called **irreversibility**.

Lets take an **Example**:
Suppose you have to store your files on any of the cloud services available. You have to be sure that the files that you store are not tampered by any third party. You do it by computing "hash" of that file using a Cryptographic hash algorithm. One of the common cryptographic hash algorithms is **SHA 256**. The hash thus computed has a maximum size of 32 bytes. So a computing the
hash of large number of files will not be a problem. You save these hashes on your local machine.

Now, when you download the files, you compute the hash again. Then you match it with the previous hash computed. Therefore, you know whether your files were tampered or not.

If anybody tamper with the file, the hash value of the file will definitely change. Tampering the file without changing the hash is nearly impossible.

Password Verification
Cryptographic hash functions are very commonly used in password verification. Let's understand this using an **Example**:
When you use any online website which requires a user login, you enter your E-mail and password to authenticate that the account you are trying to use belongs to you. When the password is entered, a hash of the password is computed which is then sent to the server for verification of the password. The passwords stored on the server are actually computed hash values of the original passwords. This is done to ensure that when the password is sent from client to server, no sniffing is there.

Data Structures(Programming Languages):
Various programming languages have hash table based Data Structures. The basic idea is to create a key-value pair where key is supposed to be a unique value, whereas value can be same for different keys. This implementation is seen in unordered_set & unordered_map in C++, HashSet & HashMap in java, dict in python etc.

Compiler Operation:
The keywords of a programming language are processed differently than other identifiers. To differentiate between the keywords of a programming language(if, else, for, return etc.) and other identifiers and to successfully compile the program, the compiler stores all these keywords in a set which is implemented using a hash table.

Rabin-Karp Algorithm:
One of the most famous applications of hashing is the Rabin-Karp algorithm. This is basically a string-searching algorithm which uses hashing to find any one set of patterns in a string. A practical application of this algorithm is detecting plagiarism. To know more about Rabin-Karp algo go through Searching for Patterns | Set 3 (Rabin-Karp Algorithm).

Linking File name and path together:
When moving through files on our local system, we observe two very crucial components of a file i.e. file_name and file_path. In order to store the correspondence between file_name and file_path the system uses a map(file_name, file_path)which is implemented using a hash table.

Related articles:

Hashing vs BST
Hashing vs Trie

Source

https://www.geeksforgeeks.org/applications-of-hashing/

Chapter 17

Asymptotic Analysis and comparison of sorting algorithms

Asymptotic Analysis and comparison of sorting algorithms - GeeksforGeeks

It is a well established fact that merge sort runs faster than insertion sort. Using asymptotic analysis we can prove that merge sort runs in O(nlogn) time and insertion sort takes O(n^2). It is obvious because merge sort uses a divide-and-conquer approach by recursively solving the problems where as insertion sort follows an incremental approach.
If we scrutinize the time complexity analysis even further, we'll get to know that insertion sort isn't that bad enough. Surprisingly, insertion sort beats merge sort on smaller input size. This is because there are few constants which we ignore while deducing the time complexity. On larger input sizes of the order 10^4 this doesn't influence the behavior of our function. But when input sizes fall below, say less than 40, then the constants in the equation dominate the input size 'n'.
So far, so good. But I wasn't satisfied with such mathematical analysis. As a computer science undergrad we must believe in writing code. I've written a C program to get a feel of how the algorithms compete against each other for various input sizes. And also, why such rigorous mathematical analysis is done on establishing running time complexities of these sorting algorithms.

Implementation:

```
 //C++ code to compare performance of sorting algorithms
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#define MAX_ELEMENT_IN_ARRAY 1000000001
```

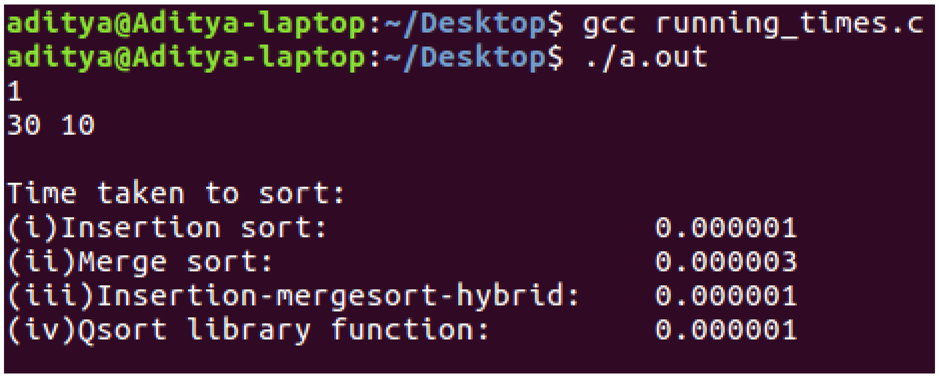
```
int cmpfunc (const void * a, const void * b)
{
    // Compare function used by qsort
    return ( *(int*)a - *(int*)b );
}

int* generate_random_array(int n)
{
    srand(time(NULL));
    int *a = malloc(sizeof(int) * n), i;
    for(i = 0; i < n; ++i)
        a[i] = rand() % MAX_ELEMENT_IN_ARRAY;
    return a;
}

int* copy_array(int a[], int n)
{
    int *arr = malloc(sizeof(int) * n);
    int i;
    for(i = 0; i < n ;++i)
        arr[i] = a[i];
    return arr;
}

//Code for Insertion Sort
void insertion_sort_asc(int a[], int start, int end)
{
    int i;
    for(i = start + 1; i <= end ; ++i)
    {
        int key = a[i];
        int j = i - 1;
        while(j >= start && a[j] > key)
        {
            a[j + 1] = a[j];
            --j;
        }
        a[j + 1] = key;
    }
}

//Code for Merge Sort
void merge(int a[], int start, int end, int mid)
{
    int i = start, j = mid + 1, k = 0;
    int *aux = malloc(sizeof(int) * (end - start + 1));
    while(i <= mid && j <= end)
    {
        if(a[i] <= a[j])
            aux[k++] = a[i++];
        else
            aux[k++] = a[j++];
    }
    while(i <= mid)
        aux[k++] = a[i++];
    while(j <= end)
        aux[k++] = a[j++];
    j = 0;
    for(i = start;i <= end;++i)
        a[i] = aux[j++];
    free(aux);
}

void _merge_sort(int a[],int start,int end)
{
    if(start < end)
    {
        int mid = start + (end - start) / 2;
        _merge_sort(a,start,mid);
        _merge_sort(a,mid + 1,end);
        merge(a,start,end,mid);
    }
}
void merge_sort(int a[],int n)
{
    return _merge_sort(a,0,n - 1);
}


void insertion_and_merge_sort_combine(int a[], int start, int end, int k)
{
    // Performs insertion sort if size of array is less than or equal to k
    // Otherwise, uses mergesort
    if(start < end)
    {
        int size = end - start + 1;

        if(size <= k)
        {
            //printf("Performed insertion sort- start = %d and end = %d\n", start, end);
            return insertion_sort_asc(a,start,end);
        }
        int mid = start + (end - start) / 2;
        insertion_and_merge_sort_combine(a,start,mid,k);
        insertion_and_merge_sort_combine(a,mid + 1,end,k);
        merge(a,start,end,mid);
    }
}

void test_sorting_runtimes(int size,int num_of_times)
{
    // Measuring the runtime of the sorting algorithms
    int number_of_times = num_of_times;
    int t = number_of_times;
    int n = size;
    double insertion_sort_time = 0, merge_sort_time = 0;
    double merge_sort_and_insertion_sort_mix_time = 0, qsort_time = 0;
    while(t--)
    {
        clock_t start, end;

        int *a = generate_random_array(n);
        int *b = copy_array(a,n);
        start = clock();
        insertion_sort_asc(b,0,n-1);
        end = clock();
        insertion_sort_time += ((double) (end - start)) / CLOCKS_PER_SEC;
        free(b);
        int *c = copy_array(a,n);
        start = clock();
        merge_sort(c,n);
        end = clock();
        merge_sort_time += ((double) (end - start)) / CLOCKS_PER_SEC;
        free(c);
        int *d = copy_array(a,n);
        start = clock();
        insertion_and_merge_sort_combine(d,0,n-1,40);
        end = clock();
        merge_sort_and_insertion_sort_mix_time+=((double) (end - start))/CLOCKS_PER_SEC;
        free(d);
        start = clock();
        qsort(a,n,sizeof(int),cmpfunc);
        end = clock();
        qsort_time += ((double) (end - start)) / CLOCKS_PER_SEC;
        free(a);
    }

    insertion_sort_time /= number_of_times;
    merge_sort_time /= number_of_times;
    merge_sort_and_insertion_sort_mix_time /= number_of_times;
    qsort_time /= number_of_times;
    printf("\nTime taken to sort:\n"
            "%-35s %f\n"
            "%-35s %f\n"
            "%-35s %f\n"
            "%-35s %f\n\n",
            "(i)Insertion sort: ",
            insertion_sort_time,
            "(ii)Merge sort: ",
            merge_sort_time,
            "(iii)Insertion-mergesort-hybrid: ",
            merge_sort_and_insertion_sort_mix_time,
            "(iv)Qsort library function: ",
            qsort_time);
}

int main(int argc, char const *argv[])
{
    int t;
    scanf("%d", &t);
    while(t--)
    {
        int size, num_of_times;
        scanf("%d %d", &size, &num_of_times);
        test_sorting_runtimes(size,num_of_times);
    }
    return 0;
}
```

I have compared the running times of the following algorithms:

- **Insertion sort**: The traditional algorithm with no modifications/optimisation. It performs very well for smaller input sizes. And yes, it does beat merge sort
- **Merge sort:** Follows the divide-and-conquer approach. For input sizes of the order 10ˆ5 this algorithm is of the right choice. It renders insertion sort impractical for such large input sizes.
- **Combined version of insertion sort and merge sort:** I have tweaked the logic of merge sort a little bit to achieve a considerably better running time for smaller input sizes. As we know, merge sort splits its input into two halves until it is trivial enough to sort the elements. But here, when the input size falls below a threshold such as 'n' < 40 then this hybrid algorithm makes a call to traditional insertion sort procedure. From the fact that insertion sort runs faster on smaller inputs and merge sort runs faster on larger inputs, this algorithm makes best use both the worlds.
- **Quick sort:** I have not implemented this procedure. This is the library function qsort() which is available in . I have considered this algorithm in order to know the significance of implementation. It requires a great deal of programming expertise to minimize the number of steps and make at most use of the underlying language primitives to implement an algorithm in the best way possible. This is the main reason why it is recommended to use library functions. They are written to handle anything and everything. They optimize to the maximum extent possible. And before I forget, from my analysis qsort() runs blazingly fast on virtually any input size!

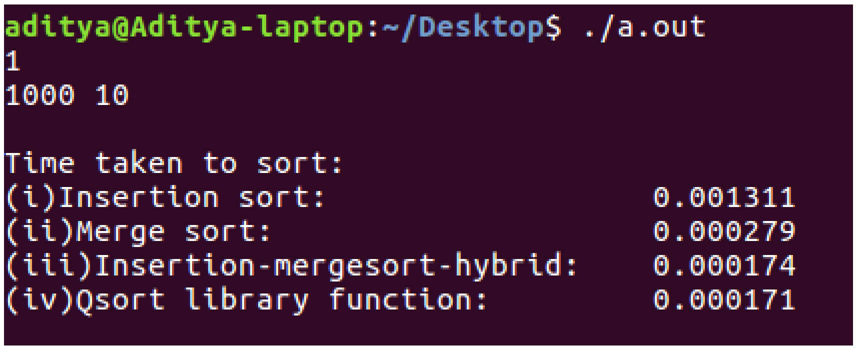
The Analysis:

- **Input:** The user has to supply the number of times he/she wants to test the algorithm corresponding to number of test cases. For each test case the user must enter two space separated integers denoting the input size 'n' and the 'num_of_times' denoting the number of times he/she wants to run the analysis and take average. (Clarification: If 'num_of_times' is 10 then each of the algorithm specified above runs 10 times and the average is taken. This is done because the input array is generated randomly corresponding to the input size which you specify. The input array could be all sorted. Our it could correspond to the worst case .i.e. descending order. In order to avoid running times of such input arrays. The algorithm is run 'num_of_times' and the average is taken.)
  clock() routine and CLOCKS_PER_SEC macro from is used to measure the time taken.
  Compilation: I have written the above code in Linux environment (Ubuntu 16.04 LTS). Copy the code snippet above. Compile it using gcc, key in the inputs as specified and admire the power of sorting algorithms!
- **Results:** As you can see for small input sizes, insertion sort beats merge sort by 2 * 10^-6 sec. But this difference in time is not so significant. On the other hand, the hybrid algorithm and qsort() library function, both perform as good as insertion sort.



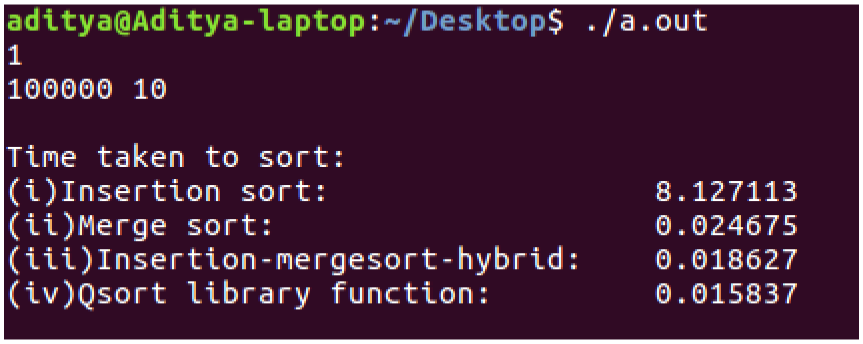
```
aditya@Aditya-laptop:~/Desktop$ gcc running_times.c
aditya@Aditya-laptop:~/Desktop$ ./a.out
1
30 10

Time taken to sort:
(i)Insertion sort:                  0.000001
(ii)Merge sort:                     0.000003
(iii)Insertion-mergesort-hybrid:    0.000001
(iv)Qsort library function:         0.000001
```

The input size is now increased by approximately 100 times to n = 1000 from n = 30. The difference is now tangible. Merge sort runs 10 times faster than insertion sort. There is again a tie between the performance of the hybrid algorithm and the qsort() routine. This suggests that the qsort() is implemented in a way which is more or less similar to our hybrid algorithm i.e., switching between different algorithms to make the best out of them.



```
aditya@Aditya-laptop:~/Desktop$ ./a.out
1
1000 10

Time taken to sort:
(i)Insertion sort:                   0.001311
(ii)Merge sort:                      0.000279
(iii)Insertion-mergesort-hybrid:     0.000174
(iv)Qsort library function:          0.000171
```

Finally, the input size is increased to 10^5 (1 Lakh!) which is most probably the ideal size used in practical scenario's. Compared to the previous input n = 1000 where merge sort beat insertion sort by running 10 times faster, here the difference is even more significant. Merge sort beats insertion sort by 100 times!
The hybrid algorithm which we have written in fact does out perform the traditional merge sort by running 0.01 sec faster. And lastly, qsort() the library function, finally proves us that implementation also plays a crucial role while measuring the running times meticulously by running 3 milliseconds faster!



```
aditya@Aditya-laptop:~/Desktop$ ./a.out
1
100000 10

Time taken to sort:
(i)Insertion sort:                   8.127113
(ii)Merge sort:                      0.024675
(iii)Insertion-mergesort-hybrid:     0.018627
(iv)Qsort library function:          0.015837
```

Note: Do not run the above program with n >= 10^6 since it will take a lot of computing power. Thank you and Happy coding!

Source

https://www.geeksforgeeks.org/asymptotic-analysis-comparison-sorting-algorithms/

Chapter 18

Auxiliary Space with Recursive Functions

Auxiliary Space with Recursive Functions - GeeksforGeeks

Prerequisite : Recursion

Memory used by a program is sometimes as important as running time, particularly in constrained environments such as mobile devices.
For example if we need to create an array of size n, it will require O(n) space. If we need a two-dimensional array of size n x n , it will require O(n^2).

Stack space in recursive calls counts too as extra space required by a program.
For example :

```
 int sum(int sum)
{
   if (n <= 0)
       return 0;
   return n + sum(n-1);
}
```

In the above example function, each call adds a new level to the stack.

```
     Sum(5)
    ->sum(4)
            ->sum(3)
            ->sum(2)
                ->sum(1)
                    ->sum(0)
```

Each of these calls is added to the call stack and takes up actual memory. So code like this would take O(n) time and O(n) auxiliary space.

However, just because you have n calls total doesn't mean it takes O(n) space. Consider the below functions, which adds adjacent elements between 0 and n :
Example:

```
 // A non-recursive code that makes n calls
// but takes O(1) extra space.
int pairSumSequence(int n)
{
    int sum = 0;
    for (int i=0; i<n; i++)
        sum += pairSum(i, i+1);
    return sum;
}

int pairSum(int a, int b)
{
    return a + b ;
}
```

In this example there will be roughly O(n) calls to pairSum. However, those calls do not exist simultaneously on the call stack, so we need only O(1) space.

Source

https://www.geeksforgeeks.org/auxiliary-space-recursive-functions/

Chapter 19

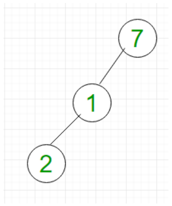
Complexity of different operations in Binary tree, Binary Search Tree and AVL tree

Complexity of different operations in Binary tree, Binary Search Tree and AVL tree - GeeksforGeeks

In this article, we will discuss complexity of different operations in binary trees including BST and AVL trees. Before understanding this article, you should have basic idea about: Binary Tree, Binary Search Tree and AVL Tree.

The main operations in binary tree are: search, insert and delete. We will see the worst case time complexity of these operations in binary trees.

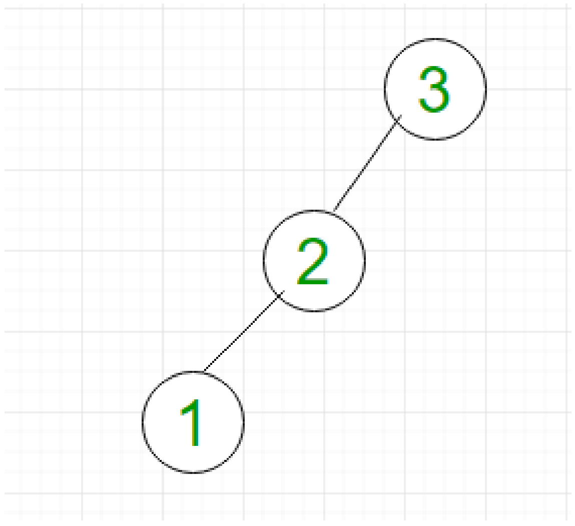
Binary Tree –
In a binary tree, a node can have maximum two children. Consider the left skewed binary tree shown in Figure 1.



- **Searching:** For searching element 2, we have to traverse all elements (assuming we do breadth first traversal). Therefore, searching in binary tree has worst case complexity of O(n).
- **Insertion:** For inserting element as left child of 2, we have to traverse all elements. Therefore, insertion in binary tree has worst case complexity of O(n).

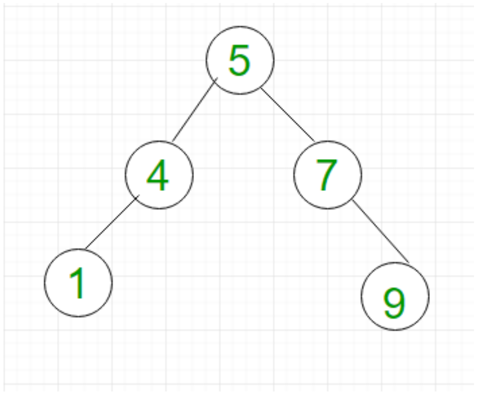
- **Deletion:** For deletion of element 2, we have to traverse all elements to find 2 (assuming we do breadth first traversal). Therefore, deletion in binary tree has worst case complexity of O(n).

Binary Search Tree (BST) –
BST is a special type of binary tree in which left child of a node has value less than the parent and right child has value greater than parent. Consider the left skewed BST shown in Figure 2.



- **Searching:** For searching element 1, we have to traverse all elements (in order 3, 2, 1). Therefore, searching in binary search tree has worst case complexity of O(n). In general, time complexity is O(h) where **h** is height of BST.
- **Insertion:** For inserting element 0, it must be inserted as left child of 1. Therefore, we need to traverse all elements (in order 3, 2, 1) to insert 0 which has worst case complexity of O(n). In general, time complexity is O(h).
- **Deletion:** For deletion of element 1, we have to traverse all elements to find 1 (in order 3, 2, 1). Therefore, deletion in binary tree has worst case complexity of O(n). In general, time complexity is O(h).

AVL/ Height Balanced Tree –
AVL tree is binary search tree with additional property that difference between height of left sub-tree and right sub-tree of any node can't be more than 1. For example, BST shown in Figure 2 is not AVL as difference between left sub-tree and right sub-tree of node 3 is 2. However, BST shown in Figure 3 is AVL tree.



- **Searching:** For searching element 1, we have to traverse elements (in order 5, 7, 9) = 3 = $\log_2 n$. Therefore, searching in AVL tree has worst case complexity of $O(\log_2 n)$.
- **Insertion:** For inserting element 12, it must be inserted as right child of 9. Therefore, we need to traverse elements (in order 5, 7, 9) to insert 12 which has worst case complexity of $O(\log_2 n)$.
- **Deletion:** For deletion of element 9, we have to traverse elements to find 9 (in order 5, 7, 9). Therefore, deletion in binary tree has worst case complexity of $O(\log_2 n)$.

We will discuss questions based on complexities of binary tree operations.

Que-1. What is the worst case time complexity for search, insert and delete operations in a general Binary Search Tree?
(A) O(n) for all
(B) O(Logn) for all
(C) O(Logn) for search and insert, and O(n) for delete
(D) O(Logn) for search, and O(n) for insert and delete

Solution: As discussed, all operations in BST have worst case time complexity of O(n). So, the correct option is (A).

Que-2. What are the worst case time complexities of searching in binary tree, BST and AVL tree respectively?
(A) O(n) for all
(B) O(Logn) for all
(C) O(n) for binary tree, and O(Logn) for others
(D) O(n) for binary tree and BST, and O(Logn) for AVL

Solution: As discussed, search operation in binary tree and BST have worst case time complexity of O(n). However, AVL tree has worst case time complexity of O(logn). So, the correct option is (D).

Source

https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/

Chapter 20

Cyclomatic Complexity

Cyclomatic Complexity - GeeksforGeeks

Cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program. It is computed using the Control Flow Graph of the program. The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if second command might immediately follow the first command.

For example, if source code contains no control flow statement then its cyclomatic complexity will be 1 and source code contains a single path in it. Similarly, if the source code contains one **if condition** then cyclomatic complexity will be 2 because there will be two paths one for true and the other for false.

Mathematically, for a structured program, the directed graph inside control flow is the edge joining two basic blocks of the program as control may pass from first to second.
So, cyclomatic complexity M would be defined as,

$$\mathbf{M = E - N + 2P}$$

where,
E = the number of edges in the control flow graph
N = the number of nodes in the control flow graph
P = the number of connected components

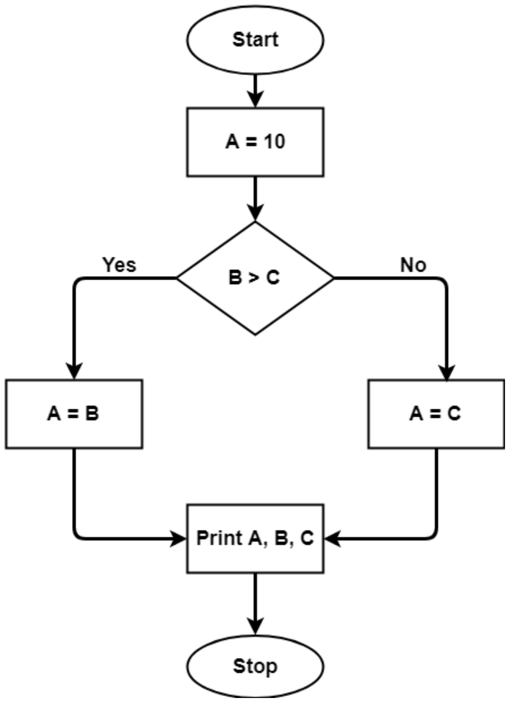
Steps that should be followed in calculating cyclomatic complexity and test cases design are:

- Construction of graph with nodes and edges from code.
- Identification of independent paths.
- Cyclomatic Complexity Calculation
- Design of Test Cases

Let a section of code as such:

```
A = 10
   IF B > C THEN
      A = B
   ELSE
      A = C
   ENDIF
Print A
Print B
Print C
```

Control Flow Graph of above code



The cyclomatic complexity calculated for above code will be from control flow graph. The graph shows seven shapes(nodes), seven lines(edges), hence cyclomatic complexity is 7-7+2 = 2.

Use of Cyclomatic Complexity:

- Determining the independent path executions thus proven to be very helpful for Developers and Testers.
- It can make sure that every path have been tested at least once.
- Thus help to focus more on uncovered paths.
- Code coverage can be improved.
- Risk associated with program can be evaluated.
- These metrics being used earlier in the program helps in reducing the risks.

Reference: https://en.wikipedia.org/wiki/Cyclomatic_complexity

Source

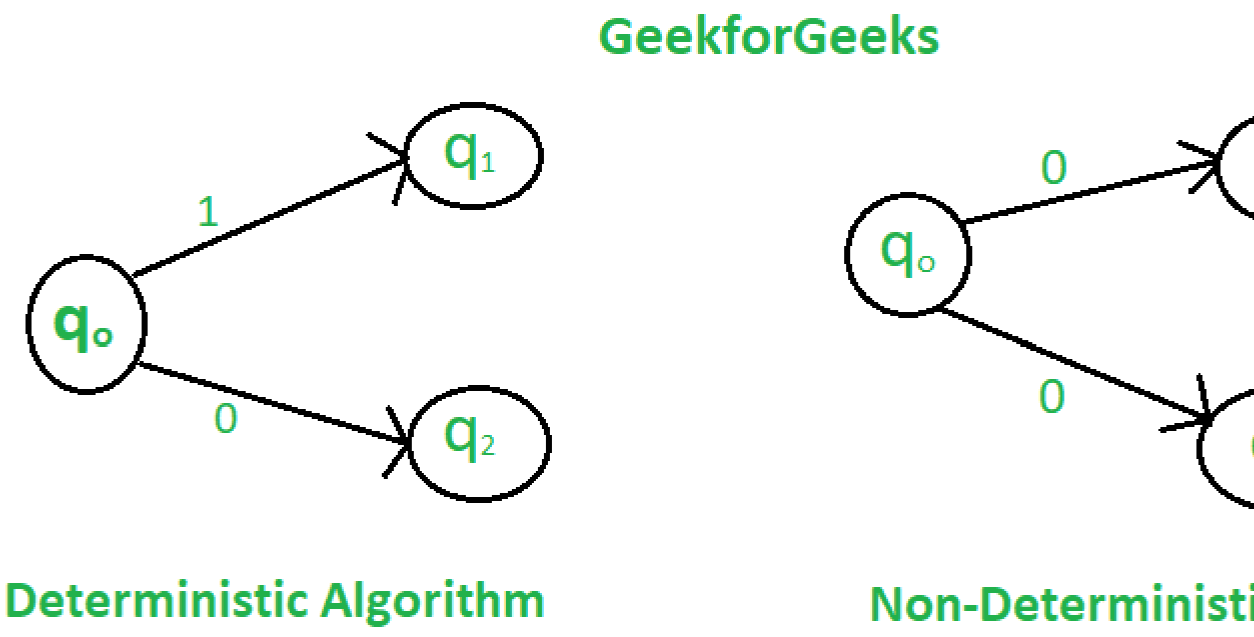
https://www.geeksforgeeks.org/cyclomatic-complexity/

Chapter 21

Difference between Deterministic and Non-deterministic Algorithms

Difference between Deterministic and Non-deterministic Algorithms - GeeksforGeeks

In **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states but in case of **non-deterministic algorithm**, for the same input, the compiler may produce different output in different runs. In fact non-deterministic algorithms can't solve the problem in polynomial time and can't determine what is the next step. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.



To implement a non-deterministic algorithm, we have a couple of languages like Prolog but these don't have standard programming language operators and these operators are not a part of any standard programming languages.

Some of the terms related to the non-deterministic algorithm are defined below:

- **choice(X) :** chooses any value randomly from the set X.
- **failure() :** denotes the unsuccessful solution.
- **success() :** Solution is successful and current thread terminates.

Example :

> **Problem Statement :** Search an element x on A[1:n] where n>=1, on successful search return j if a[j] is equals to x otherwise return 0.
>
> **Non-deterministic Algorithm for this problem :**

```
1.j= choice(a, n)
2.if(A[j]==x) then
    {
       write(j);
       success();
    }
3.write(0); failure();
```

Deterministic Algorithm	Non-deterministic Algorithm
For a particular input the computer will give always same output.	For a particular input the computer will give d
Can solve the problem in polynomial time.	Can't solve the problem in polynomial time.
Can determine the next step of execution.	Cannot determine the next step of execution d

Source

https://www.geeksforgeeks.org/difference-between-deterministic-and-non-deterministic-algorithms/

Chapter 22

Different types of recurrence relations and their solutions

Different types of recurrence relations and their solutions - GeeksforGeeks

In this article, we will see how we can solve different types of recurrence relations using different approaches. Before understanding this article, you should have idea about recurrence relations and different method to solve them (See : Worst, Average and Best Cases, Asymptotic Notations, Analysis of Loops).

Type 1: Divide and conquer recurrence relations –
Following are some of the examples of recurrence relations based on divide and conquer.

```
T(n) = 2T(n/2) + cn
T(n) = 2T(n/2) + √n
```

These types of recurrence relations can be easily solved using master method (Put link to master method).
For recurrence relation T(n) = 2T(n/2) + cn, the values of a = 2, b = 2 and k =1. Here logb(a) = log2(2) = 1 = k. Therefore, the complexity will be Θ(nlog2(n)).
Similarly for recurrence relation T(n) = 2T(n/2) + √n, the values of a = 2, b = 2 and k =1/2. Here logb(a) = log2(2) = 1 > k. Therefore, the complexity will be Θ(n).

Type 2: Linear recurrence relations –
Following are some of the examples of recurrence relations based on linear recurrence relation.

```
T(n) = T(n-1) + n for n>0 and T(0) = 1
```

These types of recurrence relations can be easily soled using substitution method (Put link to substitution method).
For example,

```
T(n) = T(n-1) + n
       = T(n-2) + (n-1) + n
       = T(n-k) + (n-(k-1))….. (n-1) + n
```

Substituting k = n, we get

```
T(n) = T(0) + 1 + 2+….. +n = n(n+1)/2 = O(n^2)
```

Type 3: Value substitution before solving –
Sometimes, recurrence relations can't be directly solved using techniques like substitution, recurrence tree or master method. Therefore, we need to convert the recurrence relation into appropriate form before solving. For example,

```
T(n) = T(√n) + 1
```

To solve this type of recurrence, substitute n = 2^m as:

```
T(2^m) = T(2^m /2) + 1
Let T(2^m) = S(m),
S(m) = S(m/2) + 1
```

Solving by master method, we get

```
S(m) = Θ(logm)
As n = 2^m or m = log2(n),
T(n) = T(2^m) = S(m) = Θ(logm) = Θ(loglogn)
```

Let us discuss some questions based on the approaches discussed.

Que – 1. What is the time complexity of Tower of Hanoi problem?
(A) T(n) = O(sqrt(n))
(D) T(n) = O(n^2)
(C) T(n) = O(2^n)
(D) None

Solution: For Tower of Hanoi, T(n) = 2T(n-1) + c for n>1 and T(1) = 1. Solving this,

```
T(n) = 2T(n-1) + c
        = 2(2T(n-2)+ c) + c  = 2^2*T(n-2) + (c + 2c)
       = 2^k*T(n-k) + (c + 2c + .. kc)
Substituting k = (n-1), we get
T(n) = 2^(n-1)*T(1) + (c + 2c + (n-1)c) = O(2^n)
```

Que – 2. Consider the following recurrence:
T(n) = 2 * T(ceil (sqrt(n))) + 1, T(1) = 1
Which one of the following is true?
(A) T(n) = (loglogn)
(B) T(n) = (logn)
(C) T(n) = (sqrt(n))
(D) T(n) = (n)

Solution: To solve this type of recurrence, substitute n = 2^m as:

```
T(2^m) = 2T(2^m /2) + 1
Let T(2^m) = S(m),
S(m) = 2S(m/2) + 1
Solving by master method, we get
S(m) = Θ(m)
As n = 2^m or m = log2n,
T(n) = T(2^m) = S(m) = Θ(m) = Θ(logn)
```

Source

https://www.geeksforgeeks.org/different-types-recurrence-relations-solutions/

Chapter 23

In-Place Algorithm

In-Place Algorithm - GeeksforGeeks

In-place has more than one definitions. One **strict definition** is.

> An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space used for variables is allowed.

A more **broad definition** is,

> In-place means that the algorithm does not use extra space for manipulating the input but may require a small though nonconstant extra space for its operation. Usually, this space is O(log n), though sometimes anything in o(n) (Smaller than linear) is allowed [Source : Wikipedia]

A **Not In-Place** Implementation of reversing an array

C++

```
 // An in-place C++ program to reverse an array
#include <bits/stdc++.h>
using namespace std;

/* Function to reverse arr[] from start to end*/
void reverseseArray(int arr[], int n)
{
   // Create a copy array and store reversed
   // elements
   int rev[n];
   for (int i=0; i<n; i++)
       rev[n-i-1] = arr[i];

   // Now copy reversed elements back to arr[]
   for (int i=0; i<n; i++)
       arr[i] = rev[i];
}

/* Utility function to print an array */
void printArray(int arr[], int size)
{
   for (int i = 0; i < size; i++)
      cout << arr[i] << " ";
   cout << endl;
}

/* Driver function to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    printArray(arr, n);
    revereseArray(arr, n);
    cout << "Reversed array is" << endl;
    printArray(arr, n);
    return 0;
}
```

Java

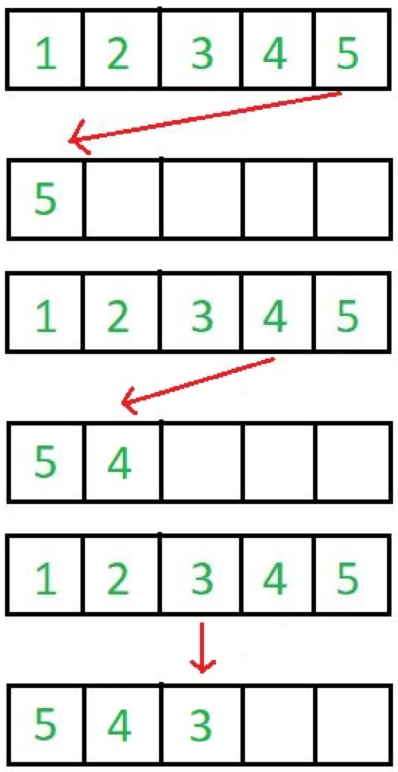
```
 // An in-place Java program
// to reverse an array
import java.util.*;

class GFG
{
    /* Function to reverse arr[]
       from start to end*/
    public static void revereseArray(int []arr,
                                     int n)
    {
        // Create a copy array
        // and store reversed
        // elements
        int []rev = new int[n];
        for (int i = 0; i < n; i++)
            rev[n - i - 1] = arr[i];
        // Now copy reversed
        // elements back to arr[]
        for (int i = 0; i < n; i++)
            arr[i] = rev[i];
    }

    /* Utility function to
       print an array */
    public static void printArray(int []arr,
                                  int size)
    {
    for (int i = 0; i < size; i++)
        System.out.print(arr[i] + " ");
    System.out.println("");
    }

    // Driver code
    public static void main(String[] args)
    {
        int arr[] = {1, 2, 3, 4, 5, 6};
        int n = arr.length;
        printArray(arr, n);
        rever eseArray(arr, n);
        System.out.println("Reversed array is");
        printArray(arr, n);
    }
}

// This code is contributed
// by Harshit Saini
```

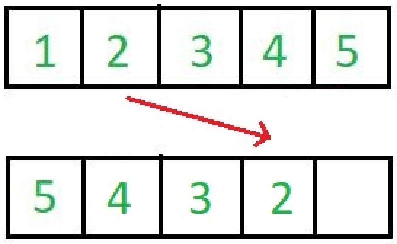
Python3

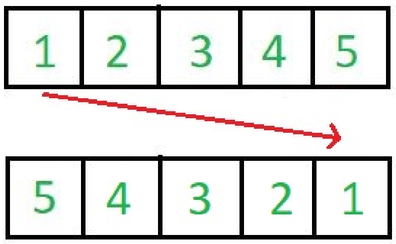
```
 # An in-place Python program
# to reverse an array

''' Function to reverse arr[]
    from start to end '''
def rever eseArray(arr, n):

    # Create a copy array
    # and store reversed
    # elements
    rev = n * [0]
    for i in range(0, n):
        rev[n - i - 1] = arr[i]

    # Now copy reversed
    # elements back to arr[]
    for i in range(0, n):
        arr[i] = rev[i]

# Driver code
if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6]
    n = len(arr)
    print(*arr)
    revereseArray(arr, n);
    print("Reversed array is")
    print(*arr)

# This code is contributed
# by Harshit Saini
```

Output:

```
1 2 3 4 5 6
Reversed array is
6 5 4 3 2 1
```







This needs O(n) extra space and is an example of not-in-place algorithm.

An **In-Place** Implementation of Reversing an array.

C++

```
 // An in-place C++ program to reverse an array
#include <bits/stdc++.h>
using namespace std;

/* Function to reverse arr[] from start to end*/
void revereseArray(int arr[], int n)
{
   for (int i=0; i<n/2; i++)
     swap(arr[i], arr[n-i-1]);
}

/* Utility function to print an array */
void printArray(int arr[], int size)
{
   for (int i = 0; i < size; i++)
      cout << arr[i] << " ";
   cout << endl;
}

/* Driver function to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    printArray(arr, n);
    revereseArray(arr, n);
    cout << "Reversed array is" << endl;
    printArray(arr, n);
    return 0;
}
```

Java

```
 // An in-place Java program
// to reverse an array
import java.util.*;

class GFG
{
    public static int __(int x, int y) {return x;}

    /* Function to reverse arr[]
       from start to end*/
    public static void revereseArray(int []arr,
                                     int n)
    {
        for (int i = 0; i < n / 2; i++)
            arr[i] = __(arr[n - i - 1],
                        arr[n - i - 1] = arr[i]);
    }

    /* Utility function to
       print an array */
    public static void printArray(int []arr,
                                  int size)
    {
        for (int i = 0; i < size; i++)
            System.out.print(Integer.toString(arr[i]) + " ");
        System.out.println("");
    }

    // Driver code
    public static void main(String[] args)
    {
        int []arr = new int[]{1, 2, 3, 4, 5, 6};
        int n = arr.length;
        printArray(arr, n);
        revereseArray(arr, n);
        System.out.println("Reversed array is");
        printArray(arr, n);
    }
}
```

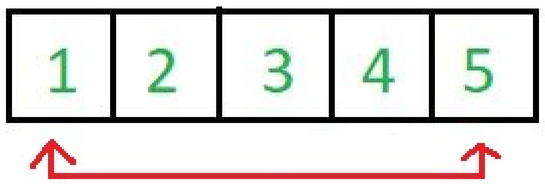
```
// This code is contributed
// by Harshit Saini
```

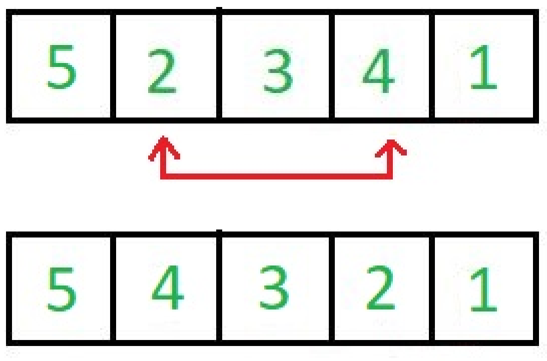
Python3

```
 # An in-place Python program
# to reverse an array

''' Function to reverse arr[]
    from start to end'''
def revereseArray(arr, n):

    for i in range(0, int(n / 2)):
        arr[i], arr[n - i - 1] = arr[n - i - 1], arr[i]


# Driver code
if __name__ == "__main__":

    arr = [1, 2, 3, 4, 5, 6]
    n = len(arr)
    print(*arr)
    revereseArray(arr, n)
    print("Reversed array is")
    print(*arr)

# This code is contributed
# by Harshit Saini
```

Output:

```
1 2 3 4 5 6
Reversed array is
6 5 4 3 2 1
```





This needs O(1) extra space for exchanging elements and is an example of in-place algorithm.

Which Sorting Algorithms are In-Place and which are not?
In Place : Bubble sort, Selection Sort, Insertion Sort, Heapsort.

Not In-Place : Merge Sort. Note that merge sort requires O(n) extra space.

What about QuickSort? Why is it called In-Place?
QuickSort uses extra space for recursive function calls. It is called in-place according to broad definition as extra space required is not used to manipulate input, but only for recursive calls.

Improved By : Harshit Saini

Source

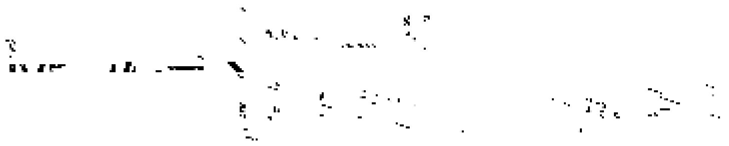
https://www.geeksforgeeks.org/in-place-algorithm/

Chapter 24

Iterated Logarithm log*(n)

Iterated Logarithm log*(n) - GeeksforGeeks

Iterated Logarithm or Log*(n) is the number of times the logarithm function must be iteratively applied before the result is less than or equal to 1.



Applications: It is used in analysis of algorithms (Refer Wiki for details)

C++

```
 // Recursive CPP program to find value of
// Iterated Logarithm
#include <bits/stdc++.h>
using namespace std;

int _log(double x, double base)
{
    return (int)(log(x) / log(base));
}

double recursiveLogStar(double n, double b)
{
    if (n > 1.0)
        return 1.0 + recursiveLogStar(_log(n, b), b);
    else
        return 0;
}
```

```
// Driver code
int main()
{
    int n = 100, base = 5;
    cout << "Log*(" << n << ") = "
         << recursiveLogStar(n, base) << "\n";
    return 0;
}
```

Java

```
 // Recursive Java program to
// find value of Iterated Logarithm
import java.io.*;

class GFG
{
static int _log(double x,
                double base)
{
    return (int)(Math.log(x) /
                 Math.log(base));
}

static double recursiveLogStar(double n,
                               double b)
{
    if (n > 1.0)
        return 1.0 +
               recursiveLogStar(_log(n,
                                b), b);
    else
        return 0;
}

// Driver code
public static void main (String[] args)
{
    int n = 100, base = 5;
    System.out.println("Log*(" + n + ") = " +
                  recursiveLogStar(n, base));
}
}

// This code is contributed by jit_t
```

PHP

```
 <?php
// Recursive PhP program to find
// value of Iterated Logarithm

function _log($x, $base)
{
    return (int)(log($x) / log($base));
}

function recursiveLogStar($n, $b)
{
    if ($n > 1.0)
        return 1.0 +
               recursiveLogStar(_log($n,
                                $b), $b);
    else
        return 0;
}

// Driver code
$n = 100; $base = 5;
echo "Log*(" , $n , ")"," = ",
recursiveLogStar($n, $base), "\n";

// This code is contributed by ajit
?>
```

Output :

```
Log*(100) = 2
```

Iterative Implementation :

```
 // Iterative CPP function to find value of
// Iterated Logarithm
int iterativeLogStar(double n, double b)
{
    int count = 0;
    while (n >= 1) {
        n = _log(n, b);
        count++;
    }
    return count;
}
```

Improved By : jit_t

Source

https://www.geeksforgeeks.org/iterated-logarithm-logn/

Chapter 25

Knowing the complexity in competitive programming

Knowing the complexity in competitive programming - GeeksforGeeks

Prerequisite: Time Complexity Analysis

Generally, while doing competitive programming problems on various sites, the most difficult task faced is writing the code under desired complexity otherwise the program will get a TLE (**Time Limit Exceeded**). A naive solution is almost never accepted. So how to know, what complexity is acceptable?

The answer to this question is directly related to the number of operations that are allowed to perform within a second. Most of the sites these days allow **10^8 operations per second**, only a few sites still allow 10^7 operations. After figuring out the number of operations that can be performed, search for the right complexity by looking at the constraints given in the problem.

Example:
Given an array A[] and a number x, check for a pair in A[] with the sum as x.
where N is:

```
1) 1 <= N <= 103
2) 1 <= N <= 105
3) 1 <= N <= 108
```

For Case 1
A naive solution that is using two for-loops works as it gives us a complexity of O(N^2), which even in the worst case will perform 10^6 operations which are well under 10^8. Ofcourse O(N) and O(NlogN) is also acceptable in this case.

For Case 2
We have to think of a better solution than O(N^2), as in worst case, it will perform 10^{10}

operations as N is 10^5. So complexity acceptable for this case is either O(NlogN) which is approximately 10^6 (10^5 * ~10) operations well under 10^8 or O(N).

For Case 3
Even O(NlogN) gives us TLE as it performs ~10^9 operations which are over 10^8. So the only solution which is acceptable is O(N) which in worst case will perform 10^8 operations.

The code for the given problem can be found on : https://www.geeksforgeeks.org/write-a-c-program-that-given-a-set-a-of-n-numbers-and-another-number-x-determines-whether-or-not-there-exist-t

Source

https://www.geeksforgeeks.org/knowing-the-complexity-in-competitive-programming/

Chapter 26

Loop Invariant Condition with Examples of Sorting Algorithms

Loop Invariant Condition with Examples of Sorting Algorithms - GeeksforGeeks

Loop Invariant Condition:
Loop invariant condition is a condition about the relationship between the variables of our program which is definitely true immediately before and immediately after each iteration of the loop.
For example: Consider an array A{7, 5, 3, 10, 2, 6} with 6 elements and we have to find maximum element max in the array.

```
max = -INF (minus infinite)
for (i = 0 to n-1)
  if (A[i] > max)
      max = A[i]
```

In the above example after 3rd iteration of the loop max value is 7, which holds true for first 3 elements of array A. Here, the loop invariant condition is that max is always maximum among the first i elements of array A.

Loop Invariant condition of various algorithms:
Prerequisite: insertion sort, selection sort, quick sort, bubblesort,

Selection Sort:
In selection sort algorithm we find the minimum element from the unsorted part and put it at the beginning.

```
min_idx = 0

for (i = 0; i < n-1; i++)
```

```
{
   min_idx = i;
   for (j = i+1 to n-1)
      if (arr[j] < arr[min_idx])
         min_idx = j;

    swap(&arr[min_idx], &arr[i]);
}
```

In the above pseudo code there are two loop invariant condition:
1. In the outer loop, array is sorted for first i elements.
2. In the inner loop, min is always the minimum value in A[i to j].

Insertion Sort:
In insertion sort, loop invariant condition is that the subarray A[0 to i-1] is always sorted.

```
for (i = 1 to n-1)
{
   key = arr[i];
   j = i-1;
   while (j >= 0 and arr[j] > key)
   {
      arr[j+1] = arr[j];
      j = j-1;
   }
   arr[j+1] = key;
}
```

Quicksort:
In quicksort algorithm, after every partition call array is divided into 3 regions:
1. Pivot element is placed at its correct position.
2. Elements less than pivot element lie on the left side of pivot element.
3. Elements greater than pivot element lie on the right side of pivot element.

```
quickSort(arr[], low, high)
{
    if (low < high)
    {
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}

partition (arr[], low, high)
{
    pivot = arr[high];
    i = (low - 1)
    for (j = low; j <= high- 1; j++)
        if (arr[j] <= pivot)
            i++;
            swap arr[i] and arr[j]
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Bubble Sort:
In bubble sort algorithm, after each iteration of the loop largest element of the array is always placed at right most position. Therefore, the loop invariant condition is that at the end of i iteration right most i elements are sorted and in place.

```
for (i = 0 to n-1)
  for (j = 0 to j  arr[j+1])
       swap(&arr[j], &arr[j+1]);
```

Source

https://www.geeksforgeeks.org/loop-invariant-condition-examples-sorting-algorithms/

Chapter 27

Master Theorem For Subtract and Conquer Recurrences

Master Theorem For Subtract and Conquer Recurrences - GeeksforGeeks

Master theorem is used to determine the Big – O upper bound on functions which possess recurrence, i.e which can be broken into sub problems.
Master Theorem For Subtract and Conquer Recurrences:
Let T(n) be a function defined on positive n as shown below:

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b) + f(n), & n > 1, \end{cases} ,$$

for some constants c, a>0, b>0, k>=0 and function f(n). If f(n) is $O(n^k)$, then

1. If a<1 then $T(n) = O(n^k)$
2. If a=1 then $T(n) = O(n^{k+1})$
3. if a>1 then $T(n) = O(n^k a^{n/b})$

Proof of above theorem(By substitution method):

From above function, we have:
T(n) = aT(n-b) + f(n)
T(n-b) = aT(n-2b) + f(n-b)
T(n-2b) = aT(n-3b) + f(n-2b)

Now,
$T(n-b) = a^2T(n-3b) + af(n-2b) + f(n-b)$
$T(n) = a^3T(n-3b) + a^2f(n-2b) + af(n-b) + f(n)$
$T(n) = \Sigma^{i=0 \text{ to } n} a^i f(n-ib)$ + constant, where f(n-ib) is O(n-ib)
$T(n) = O(n^k \Sigma^{i=0 \text{ to } n/b} a^i)$

Where,
If a<1 then $\Sigma^{i=0 \text{ to } n/b} a^i = O(1)$, $T(n) = O(n^k)$

If a=1 then $\Sigma^{i=0 \text{ to } n/b} a^i = O(n)$, $T(n) = O(n^{k+1})$

If a>1 then $\Sigma^{i=0 \text{ to } n/b} a^i = O(a^{n/b})$, $T(n) = O(n^k a^{n/b})$

Consider the following program for nth fibonacci number:

```
 #include<stdio.h>
int fib(int n)
{
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}

int main ()
{
  int n = 9;
  printf("%d", fib(n));
  getchar();
  return 0;
}
```

Output

```
34
```

Time complexity Analysis:
The recursive function can be defined as, T(n) = T(n-1) + T(n-2)

- For Worst Case, Let T(n-1)   T(n-2)
  T(n) = 2T(n-1) + c
  where,f(n) = O(1)
   k=0, a=2, b=1;

  $T(n) = O(n^0 2^{n/1})$
  $= O(2^n)$

- For Best Case, Let T(n-2)   T(n-1)
  T(n) = 2T(n-2) + c
  where,f(n) = O(1)
   k=0, a=2, b=2;

  $T(n) = O(n^0 2^{n/2})$
  $= O(2^{n/2})$

More Examples:

- **Example-1**:
  T(n) = 3T(n-1), n>0
       = c, n<=0

Sol:a=3, b=1, f(n)=0 so k=0;

Since a>0, $T(n) = O(n^k a^{n/b})$
$T(n)= O(n^0 3^{n/1})$
$T(n)= 3^n$

- **Example-2**:
  T(n) = T(n-1) + n(n-1), if n>=2
  = 1, if n=1

  Sol:a=1, b=1, f(n)=n(n-1) so k=2;

  Since a=1, $T(n) = O(n^{k+1})$
  $T(n)= O(n^{2+1})$
  $T(n)= O(n^3)$

- **Example-3**:
  T(n) = 2T(n-1) – 1, if n>0
  = 1, if n<=0

  Sol: This recurrence can't be solved using above method
  since function is not of form **T(n) = aT(n-b) + f(n)**

Source

https://www.geeksforgeeks.org/master-theorem-subtract-conquer-recurrences/

Chapter 28

Measure execution time with high precision in C/C++

Measure execution time with high precision in C/C++ - GeeksforGeeks

Execution time : The execution time or CPU time of a given task is defined as the time spent by the system executing that task in other way you can say the time during which a program is running.
There are multiple way to measure execution time of a program, in this article i will discuss 5 different way to
measure execution time of a program.

1. **Using `time()` function in C & C++**.

   time() : time() function returns the time since the Epoch(jan 1 1970) in seconds.
   Header File : "time.h"
   Prototype / Syntax : time_t time(time_t *tloc);
   Return Value : On success, the value of time in seconds since the Epoch is returned, on error -1 is returned.

   Below program to demonstrate how to measure execution time using `time()` function.

```
 #include <bits/stdc++.h>
using namespace std;

// A sample function whose time taken to
// be measured
void fun()
{
    for (int i=0; i<10; i++)
    {
    }
}

int main()
{
    /* Time function returns the time since the
        Epoch(jan 1 1970). Returned time is in seconds. */
    time_t start, end;

    /* You can call it like this : start = time(NULL);
     in both the way start contain total time in seconds
     since the Epoch. */
    time(&start);

    // sync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    fun();

    // Recording end time.
    time(&end);

    // Calculating total time taken by the program.
    double time_taken = double(end - start);
    cout << "Time taken by program is : " << fixed
         << time_taken << setprecision(5);
    cout << " sec " << endl;
    return 0;
}
```

Output:

```
Time taken by program is : 0.000000 sec
```

2. **Using `clock()` function in C & C++.**

    clock() : clock() returns the number of clock ticks elapsed since the program was launched.
    Header File : "time.h"
    Prototype / Syntax : clock_t clock(void);
    Return Value : On success, the value returned is the CPU time used so far as a clock_t; To get the number of seconds used, divide by CLOCKS_PER_SEC.on error -1 is returned.

Below program to demonstrate how to measure execution time using `clock()` function.you can also see this

```
 #include <bits/stdc++.h>
using namespace std;

// A sample function whose time taken to
// be measured
void fun()
{
    for (int i=0; i<10; i++)
    {
    }
}

int main()
{
    /* clock_t clock(void) returns the number of clock ticks
       elapsed since the program was launched.To get the number
       of seconds used by the CPU, you will need to divide by
       CLOCKS_PER_SEC.where CLOCKS_PER_SEC is 1000000 on typical
       32 bit system.  */
    clock_t start, end;

    /* Recording the starting clock tick.*/
    start = clock();

    fun();

    // Recording the end clock tick.
    end = clock();

    // Calculating total time taken by the program.
    double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "Time taken by program is : " << fixed
         << time_taken << setprecision(5);
    cout << " sec " << endl;
    return 0;
}
```

Output:

```
Time taken by program is : 0.000001 sec
```

3. **using `gettimeofday()` function in C & C++.**

   gettimeofday() : The function gettimeofday() can get the time as well as timezone.
   Header File : "sys/time.h".

Prototype / Syntax : int gettimeofday(struct timeval *tv, struct timezone *tz);
The tv argument is a struct timeval and gives the number of seconds and micro seconds since the
Epoch.
struct timeval {
time_t tv_sec; // seconds
suseconds_t tv_usec; // microseconds
};
Return Value : return 0 for success, or -1 for failure.

Below program to demonstrate how to measure execution time using `gettimeofday()` function.

```
 #include <bits/stdc++.h>
#include <sys/time.h>
using namespace std;

// A sample function whose time taken to
// be measured
void fun()
{
    for (int i=0; i<10; i++)
    {
    }
}

int main()
{
    /* The function gettimeofday() can get the time as
       well as timezone.
       int gettimeofday(struct timeval *tv, struct timezone *tz);
      The tv argument is a struct timeval and gives the
      number of seconds and micro seconds since the Epoch.
      struct timeval {
               time_t      tv_sec;     // seconds
               suseconds_t tv_usec;    // microseconds
           };    */
    struct timeval start, end;

    // start timer.
    gettimeofday(&start, NULL);

    // sync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    fun();
    // stop timer.
    gettimeofday(&end, NULL);

    // Calculating total time taken by the program.
    double time_taken;

    time_taken = (end.tv_sec - start.tv_sec) * 1e6;
    time_taken = (time_taken + (end.tv_usec -
                              start.tv_usec)) * 1e-6;

    cout << "Time taken by program is : " << fixed
         << time_taken << setprecision(6);
    cout << " sec" << endl;
    return 0;
}
```

Output:

```
Time taken by program is : 0.000029 sec
```

4. **Using `clock_gettime()` function in C & C++.**

    clock_gettime() : The clock_gettime() function gets the current time of the clock specified by clock_id, and puts it into the buffer pointed to by tp.
    Header File : "time.h".
    Prototype / Syntax : int clock_gettime(clockid_t clock_id, struct timespec *tp);
    tp parameter points to a structure containing atleast the following members :

    struct timespec {
    time_t tv_sec; //seconds
    long tv_nsec; //nanoseconds
    };

    Return Value : return 0 for success, or -1 for failure.
    clock_id : clock id = CLOCK_REALTIME, CLOCK_PROCESS_CPUTIME_ID, CLOCK_MONOTONIC … etc.
    CLOCK_REALTIME : clock that measures real i.e., wall-clock) time.
    CLOCK_PROCESS_CPUTIME_ID : High-resolution per-process timer from the CPU.
    CLOCK_MONOTONIC : High resolution timer that is unaffected by system date changes (e.g. NTP daemons).

Below program to demonstrate how to measure execution time using `clock_gettime()` function.

```
 #include <bits/stdc++.h>
#include <sys/time.h>
using namespace std;

// A sample function whose time taken to
// be measured
void fun()
{
    for (int i=0; i<10; i++)
    {
    }
}

int main()
{
    /* int clock_gettime( clockid_t clock_id, struct
     timespec *tp ); The clock_gettime() function gets
     the current time of the clock specified by clock_id,
     and puts it into the buffer  pointed to by tp.tp
     parameter points to a structure containing
     atleast the following members:
     struct timespec {
               time_t   tv_sec;        // seconds
               long     tv_nsec;       // nanoseconds
           };
    clock id = CLOCK_REALTIME, CLOCK_PROCESS_CPUTIME_ID,
               CLOCK_MONOTONIC ...etc
    CLOCK_REALTIME : clock  that  measures real (i.e., wall-clock) time.
    CLOCK_PROCESS_CPUTIME_ID : High-resolution per-process timer
                               from the CPU.
    CLOCK_MONOTONIC : High resolution timer that is unaffected
                      by system date changes (e.g. NTP daemons).  */
    struct timespec start, end;

    // start timer.
    // clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
    // clock_gettime(CLOCK_REALTIME, &start);
    clock_gettime(CLOCK_MONOTONIC, &start);

    // sync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    fun();

    // stop timer.
    // clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);
    // clock_gettime(CLOCK_REALTIME, &end);
    clock_gettime(CLOCK_MONOTONIC, &end);
    // Calculating total time taken by the program.
    double time_taken;
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;

    cout << "Time taken by program is : " << fixed
         << time_taken << setprecision(9);
    cout << " sec" << endl;
    return 0;
}
```

Output:

```
Time taken by program is : 0.000028 sec
```

5. **Using `chrono::high_resolution_clock` in C++.**

    chrono : Chrono library is used to deal with date and time. This library was designed to deal with the fact that timers and clocks might be different on different systems and thus to improve over time in terms of precision.chrono is the name of a header, but also of a sub-namespace, All the elements in this header are not defined directly under the std namespace (like most of the standard library) but under the std::chrono namespace.

Below program to demonstrate how to measure execution time using `high_resolution_clock` function. For detail info on chrono library see this and this

```
 #include <bits/stdc++.h>
#include <chrono>
using namespace std;


// A sample function whose time taken to
// be measured
void fun()
{
    for (int i=0; i<10; i++)
    {
    }
}

int main()
{
    auto start = chrono::high_resolution_clock::now();
    // sync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    fun();

    auto end = chrono::high_resolution_clock::now();

    // Calculating total time taken by the program.
    double time_taken =
      chrono::duration_cast<chrono::nanoseconds>(end - start).count();

    time_taken *= 1e-9;

    cout << "Time taken by program is : " << fixed
         << time_taken << setprecision(9);
    cout << " sec" << endl;
    return 0;
}
```

Output:

```
Time taken by program is : 0.000024 sec
```

Source

https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/

Chapter 29

NP-Completeness | Set 1 (Introduction)

NP-Completeness | Set 1 (Introduction) - GeeksforGeeks

We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

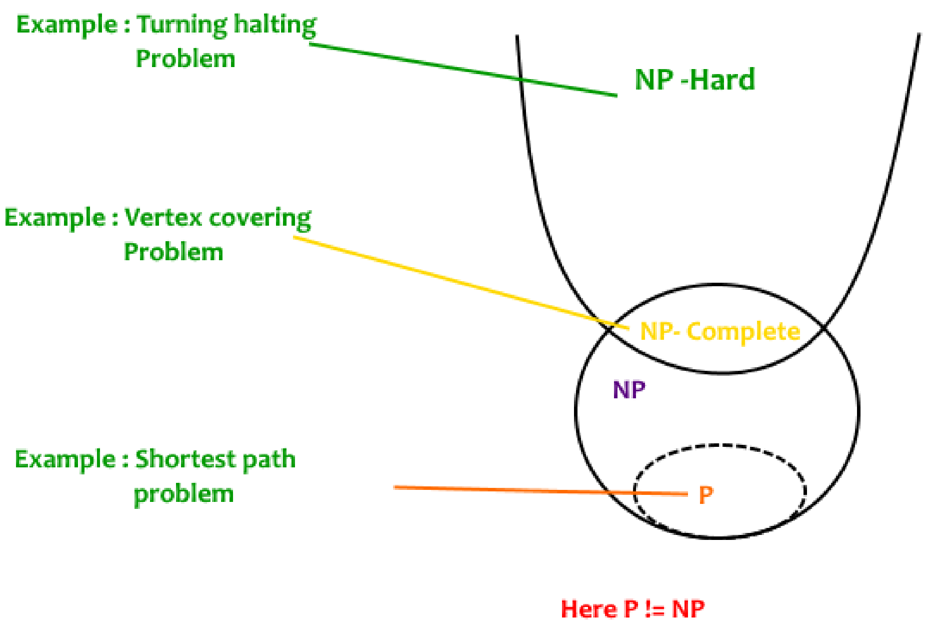
Can all computational problems be solved by a computer? There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source Halting Problem).
Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

What are NP, P, NP-complete and NP-Hard problems?
P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. 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).
Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).

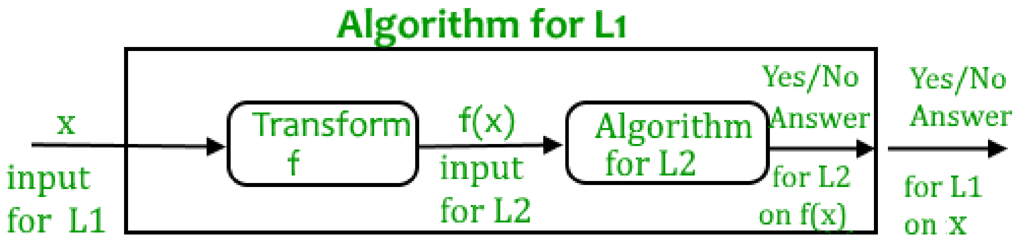
NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.


Example : Turning halting Problem
NP -Hard
Example : Vertex covering Problem
NP- Complete
NP
Example : Shortest path problem
P
Here P != NP

Decision vs Optimization Problems
NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref 2).
For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

What is Reduction?
Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not.
The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

How to prove that a given problem is NP complete?
From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

What was the first problem proved as NP-Complete?
There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (Boolean satisfiability problem)is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

We will soon be discussing more NP-Complete problems and their proof for NP-Completeness.

References:
MIT Video Lecture on Computational Complexity
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
http://www.ics.uci.edu/~eppstein/161/960312.html

Source

https://www.geeksforgeeks.org/np-completeness-set-1/

Chapter 30

Online Algorithm

Online Algorithm - GeeksforGeeks

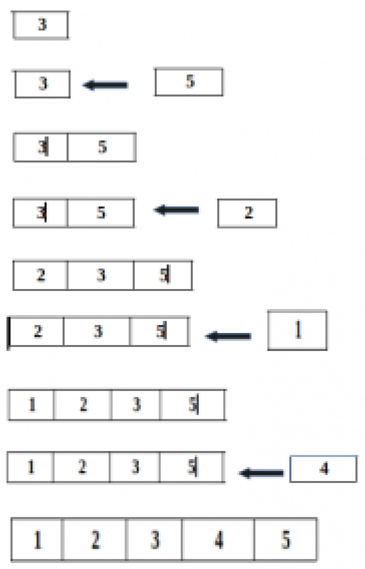
An online algorithm is one that can process its input piece-by-piece in a serial fashion, i.e., in the order that the input is fed to the algorithm, without having the entire input available from the beginning.

In contrast, an **offline algorithm** is given the whole problem data from the beginning and is required to output an answer which solves the problem at hand.

As an example, consider the sorting algorithms **selection sort** and **insertion sort**:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. which requires access to the entire input; it is thus an offline algorithm. On the other hand, insertion sort considers one input element per iteration and produces a partial solution without considering future elements. Thus insertion sort is an online algorithm.

EXAMPLE OF ONLINE ALGORITHM (INSERTION SORT):



Because an online algorithm does not know the whole input, it might make decisions that later turn out not to be optimal,
Note that insertion sort produces the optimum result. Therefore, for many problems, online algorithms cannot match the performance of offline algorithms.

Example of Online Algorithms are :
1. Insertion sort
2. Perceptron
3. Reservoir sampling
4. Greedy algorithm
5. Adversary model
6. Metrical task systems
7. Odds algorithm

Online Problems: There are many problems that offer more than one online algorithm as solution:
1. Canadian Traveller Problem
2. Linear Search Problem
3. K-server problem
4. Job shop scheduling problem
5. List update problem
6. Bandit problem
7. Secretary problem

Reference :
https://en.wikipedia.org/wiki/Online_algorithm

Source

https://www.geeksforgeeks.org/online-algorithm/

Chapter 31

Performance of loops (A caching question)

Performance of loops (A caching question) - GeeksforGeeks

Consider below two C language functions to compute sum of elements in a 2D array. Ignoring the compiler optimizations, which of the two is better implementation of sum?

```
 // Function 1
int fun1(int arr[R][C])
{
    int sum = 0;
    for (int i=0; i<R; i++)
      for (int j=0; j<C; j++)
          sum += arr[i][j];
}

// Function 2
int fun2(int arr[R][C])
{
    int sum = 0;
    for (int j=0; j<C; j++)
      for (int i=0; i<R; i++)
          sum += arr[i][j];
}
```

In C/C++, elements are stored in Row-Major order. So the first implementation has better spatial locality (nearby memory locations are referenced in successive iterations). Therefore, first implementation should always be preferred for iterating multidimensional arrays.

Source

https://www.geeksforgeeks.org/performance-of-loops-a-caching-question/

Chapter 32

Practice Questions on Time Complexity Analysis

Practice Questions on Time Complexity Analysis - GeeksforGeeks

Prerequiste: Analysis of Algorithms

1. What is the time, space complexity of following code:

```
 int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

Options:

1. O(N * M) time, O(1) space
2. O(N + M) time, O(N + M) space
3. O(N + M) time, O(1) space
4. O(N * M) time, O(N + M) space

Output:

```
3. O(N + M) time, O(1) space
```

Explanation: The first loop is O(N) and the second loop is O(M). Since we don't know which is bigger, we say this is O(N + M). This can also be written as O(max(N, M)). Since there is no additional space being utilized, the space complexity is constant / O(1)

2. What is the time complexity of following code:

```
 int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

Options:

1. O(N)
2. O(N*log(N))
3. O(N * Sqrt(N))
4. O(N*N)

Output:

```
4. O(N*N)
```

Explanation:
The above code runs total no of times
= N + (N – 1) + (N – 2) + … 1 + 0
= N * (N + 1) / 2
= 1/2 * N^2 + 1/2 * N
O(N^2) times.

3. What is the time complexity of following code:

```
 int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

Options:

1. O(n)
2. O(nLogn)
3. O(n^2)
4. O(n^2Logn)

Output:

```
2. O(nLogn)
```

Explanation: If you notice, j keeps doubling till it is less than or equal to n. Number of times, we can double a number till it is less than n would be log(n).
Let's take the examples here.
for n = 16, j = 2, 4, 8, 16
for n = 32, j = 2, 4, 8, 16, 32
So, j would run for O(log n) steps.
i runs for n/2 steps.
So, total steps = O(n/ 2 * log (n)) = **O(n*logn)**

4. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?
Options:

1. X will always be a better choice for small inputs
2. X will always be a better choice for large inputs
3. Y will always be a better choice for small inputs
4. X will always be a better choice for all inputs

```
2. X will always be a better choice for large inputs
```

Explanation: In asymptotic analysis we consider growth of algorithm in terms of input size. An algorithm X is said to be asymptotically better than Y if X takes smaller time than y for all input sizes n larger than a value n0 where n0 > 0.

5. What is the time complexity of following code:

```
 int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

Options:

1. O(N)
2. O(Sqrt(N))
3. O(N / 2)
4. O(log N)

Output:

```
4. O(log N)
```

Explanation: We have to find the smallest x such that N / 2^x N
x = log(N)

Source

https://www.geeksforgeeks.org/practice-questions-time-complexity-analysis/

Chapter 33

Practice Set for Recurrence Relations

Practice Set for Recurrence Relations - GeeksforGeeks

Prerequisite – Analysis of Algorithms | Set 1, Set 2, Set 3, Set 4, Set 5

Que-1. Solve the following recurrence relation?
T(n) = 7T(n/2) + 3n^2 + 2
(a) O(n^2.8)
(b) O(n^3)
(c) (n^2.8)
(d) (n^3)

Explanation –
T(n) = 7T(n/2) + 3n^2 + 2
As one can see from the formula above:
a = 7, b = 2, and f(n) = 3n^2 + 2
So, f(n) = O(n^c), where c = 2.
It falls in master's theoremcase 1:
logb(a) = log2(7) = 2.81 > 2
It follows from the first case of the master theorem that T(n) = (n^2.8) and implies O(n^2.8) as well as O(n^3).
Therefore, option (a), (b), and (c) are correct options.

Que-2. Sort the following functions in the decreasing order of their asymptotic (big-O) complexity:
f1(n) = n^√n , f2(n) = 2^n, f3(n) = (1.000001)^n , f4(n) = n^(10)*2^(n/2)
(a) f2> f4> f1> f3
(b) f2> f4> f3> f1
(c) f1> f2> f3> f4
(d) f2> f1> f4> f3

Explanation –
f2 > f4 because we can write f2(n) = 2^(n/2)*2^(n/2), f4(n) = n^(10)*2^(n/2) which clearly

shows that f2 > f4
f4 > f3 because we can write f4(n) = n^10. √2 ^n = n10.(1.414)n , which clearly shows f4> f3
f3> f1:
f1 (n) = n^√n take log both side log f1 = √n log n
f3 (n) = (1.000001)^n take log both side log f3 = n log(1.000001), we can write as log f3 = √n*√n log(1.000001) and √n > log(1.000001).
So, correct order is f2> f4> f3> f1. Option (b) is correct.

Que-3. f(n) = 2^(2n)
Which of the following correctly represents the above function?
(a) O(2^n)
(b) Ω(2^n)
(c) Θ(2^n)
(d) None of these

Explanation – f(n) = 2^(2n) = 2^n*2^n
Option (a) says f(n)<= c*2n, which is not true. Option (c) says c1*2n <= f(n) <= c2*2n, lower bound is satisfied but upper bound is not satisfied. Option (b) says c*2n <= f(n) this condition is satisfied hence option (b) is correct.
Que-4. Master's theorem can be applied on which of the following recurrence relation?
(a) T (n) = 2T (n/2) + 2^n
(b) T (n) = 2T (n/3) + sin(n)
(c) T (n) = T (n-2) + 2n^2 + 1
(d) None of these

Explanation – Master theorem can be applied to the recurrence relation of the following type
T (n) = aT(n/b) + f (n)
Option (a) is wrong because to apply master's theorem, function f(n) should be polynomial.
Option (b) is wrong because in order to apply master theorem f(n) should be monotonically increasing function.
Option (c) is not the above mentioned type, therefore correct answer is (d).

Que-5. T(n) = 3T(n/2+ 47) + 2n^2 + 10*n – 1/2. T(n) will be

(a) O(n^2)
(b) O(n^(3/2))
(c) O(n log n)
(d) None of these

Explanation – For higher values of n, n/2 >> 47, so we can ignore 47, now T(n) will be
T(n) = 3T(n/2)+ 2*n^2 + 10*n – 1/2 = 3T(n/2)+ O(n^2)
Apply master theorem, it is case 2 of master theorem T(n) = O(n^2).
Option (a) is correct.

Source

https://www.geeksforgeeks.org/practice-set-recurrence-relations/

Chapter 34

Pseudo-polynomial Algorithms

Pseudo-polynomial Algorithms - GeeksforGeeks

What is Pseudo-polynomial?
An algorithm whose worst case time complexity depends on numeric value of input (not number of inputs) is called Pseudo-polynomial algorithm.
For example, consider the problem of counting frequencies of all elements in an array of positive numbers. A pseudo-polynomial time solution for this is to first find the maximum value, then iterate from 1 to maximum value and for each value, find its frequency in array. This solution requires time according to maximum value in input array, therefore pseudo-polynomial. On the other hand, an algorithm whose time complexity is only based on number of elements in array (not value) is considered as polynomial time algorithm.

Pseudo-polynomial and NP-Completeness
Some NP-Complete problems have Pseudo Polynomial time solutions. For example, Dynamic Programming Solutions of 0-1 Knapsack, Subset-Sum and Partition problems are Pseudo-Polynomial. NP complete problems that can be solved using a pseudo-polynomial time algorithms are called weakly NP-complete.

Reference:
https://en.wikipedia.org/wiki/Pseudo-polynomial_time

Source

https://www.geeksforgeeks.org/pseudo-polynomial-in-algorithms/

Chapter 35

Python Code for time Complexity plot of Heap Sort

Python Code for time Complexity plot of Heap Sort - GeeksforGeeks

Prerequisite : HeapSort
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

We implement Heap Sort here, call it for different sized random lists, measure time taken for different sizes and generate a plot of input size vs time taken.

```
 # Python Code for Implementation and running time Algorithm
# Complexity plot of Heap Sort
# by Ashok Kajal
# This python code intends to implement Heap Sort Algorithm
# Plots its time Complexity on list of different sizes

# ---------------------Important Note -------------------
# numpy, time and matplotlib.pyplot are required to run this code
import time
from numpy.random import seed
from numpy.random import randint
import matplotlib.pyplot as plt


# find left child of node i
def left(i):
    return 2 * i + 1

# find right child of node i
def right(i):
    return 2 * i + 2

# calculate and return array size
def heapSize(A):
    return len(A)-1


# This fuction takes an array and Heapyfies
# the at node i
def MaxHeapify(A, i):
    # print("in heapy", i)
    l = left(i)
    r = right(i)

    # heapSize = len(A)
    # print("left", l, "Rightt", r, "Size", heapSize)
    if l<= heapSize(A) and A[l] > A[i] :
        largest = l
    else:
        largest = i
    if r<= heapSize(A) and A[r] > A[largest]:
        largest = r
    if largest != i:
       # print("Largest", largest)
        A[i], A[largest]= A[largest], A[i]
       # print("List", A)
        MaxHeapify(A, largest)

# this function makes a heapified array
def BuildMaxHeap(A):
    for i in range(int(heapSize(A)/2)-1, -1, -1):
        MaxHeapify(A, i)

# Sorting is done using heap of array
def HeapSort(A):
    BuildMaxHeap(A)
    B = list()
    heapSize1 = heapSize(A)
    for i in range(heapSize(A), 0, -1):
        A[0], A[i]= A[i], A[0]
        B.append(A[heapSize1])
        A = A[:-1]
        heapSize1 = heapSize1-1
        MaxHeapify(A, 0)


# randomly generates list of different
# sizes and call HeapSort funtion
```

```
elements = list()
times = list()
for i in range(1, 10):

    # generate some integers
    a = randint(0, 1000 * i, 1000 * i)
    # print(i)
    start = time.clock()
    HeapSort(a)
    end = time.clock()

    # print("Sorted list is ", a)
    print(len(a), "Elements Sorted by HeapSort in ", end-start)
    elements.append(len(a))
    times.append(end-start)

plt.xlabel('List Length')
plt.ylabel('Time Complexity')
plt.plot(elements, times, label ='Heap Sort')
plt.grid()
plt.legend()
plt.show()
# This code is contributed by Ashok Kajal
```

Output :

```
Input : Unsorted Lists of Different sizes are Generated Randomly
Output :
1000 Elements Sorted by HeapSort in  0.023797415087301488
2000 Elements Sorted by HeapSort in  0.053856713614550245
3000 Elements Sorted by HeapSort in  0.08474737185133563
4000 Elements Sorted by HeapSort in  0.13578669978414837
5000 Elements Sorted by HeapSort in  0.1658182863213824
6000 Elements Sorted by HeapSort in  0.1875901601906662
7000 Elements Sorted by HeapSort in  0.21982946862249264
8000 Elements Sorted by HeapSort in  0.2724293921580738
9000 Elements Sorted by HeapSort in  0.30996323029421546

Complexity PLot for Heap Sort is Given  Below
```

Source

https://www.geeksforgeeks.org/python-code-for-time-complexity-plot-of-heap-sort/

Chapter 36

Regularity condition in the master theorem.

Regularity condition in the master theorem. - GeeksforGeeks

Master theorem is a direct way to get the solution of a recurrence relation, provided that it is of the following type:

```
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
```

The theorem consists of the following three cases:

1.If f(n) = $\theta(n^c)$ where c < $\log_b a$ then T(n) = $\theta(n^{\log_b a})$

2.If f(n) = $\theta(n^c)$ where c = $\log_b a$ then T(n) = $\theta(n^c \text{Log } n)$

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

Imagine the recurrence **aT(n/b) + f(n)** in the form of a tree.
Case 1 covers the case when the children nodes does more work than the parent node.
For example the equation **T(n)=2T(n/2)+1** falls under the category of case 1 and we can clearly see from it's tree below that at each level the children nodes perform twice as much work as the parent node.

```
            T(n)          ------(1)
           /    \
      T(n/2)  T(n/2)      ------(2)
       /   \   /   \
```

Case 2 covers the case when the children node and the parent node does an equal amount of work.

For example the equation **T(n)=2T(n/2)+n** falls under the category of case 2 and we can clearly see from it's tree below that at each level the children nodes perform as much work as the parent node.

```
            T(n)           ------(n)
           /    \
        T(n/2)  T(n/2)     ------(n)
         /   \   /   \
```

Case 3 covers the scenario that the parent node does more work than the children nodes. **T(n)=T(n/2)+n** is a example of case 3 where the parent performs more work than the child.

```
            T(n)           ------(n)
              |
            T(n/2)         ------(n/2)
              |
```

In case 1 and case 2 the case conditions themselves make sure that work done by children is either more or equal to the parent but that is not the case with case 3.
In case 3 we apply a **regulatory condition** to make sure that the parent does at least as much as the children.

The regulatory condition for case 3 is

```
af(n/b)<=cf(n).
```

This says that f(n) (the amount of work done in the root) needs to be at least as big as the sum of the work done in the lower levels.

The equation **T(n) = T(n/2) + n(sin(n − π/2) + 2)** is a example where regulatory condition makes a huge difference. The equation isn't satisfying case 1 and case 2. In case 3 for large values of n it can never satisfy the regulatory condition. Hence this equation is beyond the scope of master theorem.

Source

https://www.geeksforgeeks.org/regularity-condition-master-theorem/

Chapter 37

Sorting without comparison of elements

Sorting without comparison of elements - GeeksforGeeks

Given an array with integer elements in small range, sort the array. We need to write a non-comparison based sorting algorithm with following assumptions about input.

1. All the entries in the array are integers.
2. The difference between the maximum value and the minimum value in the array is less than or equal to 10^6.

```
Input : arr[] = {10, 30, 20, 4}
Output : 4 10 20 30

Input : arr[] = {10, 30, 1, 20, 4}
Output : 1 4 10 20 30
```

We are not allowed to use comparison based sorting algorithms like QuickSort, MergeSort, etc.

Since elements are small, we use array elements as index. We store element counts in a count array. Once we have count array, we traverse the count array and print every present element its count times.

```
 // CPP program to sort an array without comparison
// operator.
#include <bits/stdc++.h>
using namespace std;

int sortArr(int arr[], int n, int min, int max)
```

```
{
    // Count of elements in given range
    int m = max - min + 1;

    // Count frequencies of all elements
    vector<int> c(m, 0);
    for (int i=0; i<n; i++)
       c[arr[i] - min]++;

    // Traverse through range. For every
    // element, print it its count times.
    for (int i=0; i<=m; i++)
        for  (int j=0; j < c[i]; j++)
          cout << (i + min) << " ";
}

int main()
{
    int arr[] =  {10, 10, 1, 4, 4, 100, 0};
    int min = 0, max = 100;
    int n = sizeof(arr)/sizeof(arr[0]);
    sortArr(arr, n, min, max);
    return 0;
}
```

Output:

```
0 1 4 4 10 10 100
```

What is time complexity?
Time complexity of above algorithm is O(n + (max-min))

Is above algorithm stable?
The above implementation is not stable as we do not care about order of sane elements while sorting.

How to make above algorithm stable?
The stable version of above algorithm is called Counting Sort. In counting sort, we store sums of all smaller or equal values in c[i] so that c[i] store actual position of i in sorted array. After filling c[], we traverse input array again, place every element at its position and decrement count.

What are non-comparison based standard algorithms?
Counting Sort, Radix Sort and Bucket Sort.

Source

https://www.geeksforgeeks.org/sorting-without-comparison-of-elements/

Chapter 38

Tail Recursion

Tail Recursion - GeeksforGeeks

What is tail recursion?
A recursive function is tail recursive when recursive call is the last thing executed by the function. For example the following C++ function print() is tail recursive.

```
 // An example of tail recursive function
void print(int n)
{
    if (n < 0)  return;
    cout << " " << n;

    // The last executed statement is recursive call
    print(n-1);
}
```

Why do we care?
The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use (See this for more details).

Can a non-tail recursive function be written as tail-recursive to optimize it?
Consider the following function to calculate factorial of n. It is a non-tail-recursive function. Although it looks like a tail recursive at first look. If we take a closer look, we can see that the value returned by fact(n-1) is used in fact(n), so the call to fact(n-1) is not the last thing done by fact(n)

C++

```
 #include<iostream>
```

```
using namespace std;

// A NON-tail-recursive function.  The function is not tail
// recursive because the value returned by fact(n-1) is used in
// fact(n) and call to fact(n-1) is not the last thing done by fact(n)
unsigned int fact(unsigned int n)
{
    if (n == 0) return 1;

    return n*fact(n-1);
}

// Driver program to test above function
int main()
{
    cout << fact(5);
    return 0;
}
```

Java

```
 class GFG {

    // A NON-tail-recursive function.
    // The function is not tail
    // recursive because the value
    // returned by fact(n-1) is used
    // in fact(n) and call to fact(n-1)
    // is not the last thing done by
    // fact(n)
    static int fact(int n)
    {
        if (n == 0) return 1;

        return n*fact(n-1);
    }

    // Driver program
    public static void main(String[] args)
    {
        System.out.println(fact(5));
    }
}

// This code is contributed by Smitha.
```

Python 3

```
 # A NON-tail-recursive function.
# The function is not tail
# recursive because the value
# returned by fact(n-1) is used
# in fact(n) and call to fact(n-1)
# is not the last thing done by
# fact(n)
def fact(n):

    if (n == 0):
        return 1

    return n * fact(n-1)

# Driver program to test
# above function
print(fact(5))
# This code is contributed by Smitha.
```

C#

```
 using System;

class GFG {

    // A NON-tail-recursive function.
    // The function is not tail
    // recursive because the value
    // returned by fact(n-1) is used
    // in fact(n) and call to fact(n-1)
    // is not the last thing done by
    // fact(n)
    static int fact(int n)
    {
        if (n == 0)
            return 1;

        return n * fact(n-1);
    }

    // Driver program to test
    // above function
    public static void Main()
    {
        Console.Write(fact(5));
    }
}
```

```
// This code is contributed by Smitha
```

PHP

```
 <?php
// A NON-tail-recursive function.
// The function is not tail
// recursive because the value
// returned by fact(n-1) is used in
// fact(n) and call to fact(n-1) is
// not the last thing done by fact(n)

function fact( $n)
{
    if ($n == 0) return 1;

    return $n * fact($n - 1);
}

    // Driver Code
    echo fact(5);

// This code is contributed by Ajit
?>
```

Output :

```
120
```

The above function can be written as a tail recursive function. The idea is to use one more argument and accumulate the factorial value in second argument. When n reaches 0, return the accumulated value.

C++

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

// A tail recursive function to calculate factorial
unsigned factTR(unsigned int n, unsigned int a)
{
    if (n == 0)  return a;

    return factTR(n-1, n*a);
}
```

```
// A wrapper over factTR
unsigned int fact(unsigned int n)
{
   return factTR(n, 1);
}

// Driver program to test above function
int main()
{
    cout << fact(5);
    return 0;
}
```

Java

```
 // Java Code for Tail Recursion

class GFG {

    // A tail recursive function
    // to calculate factorial
    static int factTR(int n, int a)
    {
        if (n == 0)
            return a;

        return factTR(n - 1, n * a);
    }

    // A wrapper over factTR
    static int fact(int n)
    {
        return factTR(n, 1);
    }

    // Driver code
    static public void main (String[] args)
    {
        System.out.println(fact(5));
    }
}

// This code is contributed by Smitha.
```

Python 3

```
 # A tail recursive function
```

```
# to calculate factorial
def factTR(n, a):

    if (n == 0):
        return a

    return factTR(n - 1, n * a)

# A wrapper over factTR
def fact(n):
    return factTR(n, 1)

# Driver program to test
#  above function
print(fact(5))

# This code is contributed
# by Smitha
```

C#

```
 // C# Code for Tail Recursion
using System;

class GFG {

    // A tail recursive function
    // to calculate factorial
    static int factTR(int n, int a)
    {
        if (n == 0)
            return a;

        return factTR(n - 1, n * a);
    }

    // A wrapper over factTR
    static int fact(int n)
    {
        return factTR(n, 1);
    }

    // Driver code
    static public void Main ()
    {
        Console.WriteLine(fact(5));
    }
}
```

```
// This code is contributed by Ajit.
```

PHP

```
 <?php
// A tail recursive function
// to calculate factorial
function factTR($n, $a)
{
    if ($n == 0) return $a;

    return factTR($n - 1, $n * $a);
}

// A wrapper over factTR
function fact($n)
{
    return factTR($n, 1);
}

// Driver program to test
// above function
echo fact(5);

// This code is contributed
// by Smitha
?>
```

Output :

```
120
```

Next articles on this topic:
Tail Call Elimination
QuickSort Tail Call Optimization (Reducing worst case space to Log n)

References:
http://en.wikipedia.org/wiki/Tail_call
http://c2.com/cgi/wiki?TailRecursion

Improved By : Smitha Dinesh Semwal, jit_t

Source

https://www.geeksforgeeks.org/tail-recursion/

Chapter 39

Time Complexity Analysis | Tower Of Hanoi (Recursion)

Time Complexity Analysis | Tower Of Hanoi (Recursion) - GeeksforGeeks

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

Pseudo Code

```
TOH(n, x, y, z)
{
   if (n >= 1)
   {
      // put (n-1) disk to z by using y
      TOH((n-1), x, z, y)

       // move larger disk to right place
       move:x-->y

      // put (n-1) disk to right place
      TOH((n-1), z, y, x)
   }
}
```

Analysis of Recursion

Recursive Equation : $T(n) = 2T(n-1) + 1$ [illegible] ——-equation-1

Solving it by BackSubstitution :

[illegible] ————equation-2

[illegible] ————equation-3

Put value of T(n-2) in equation–2 with help of equation-3

[illegible] ——equation-4

Put value of T(n-1) in equation-1 with help of equation-4

[illegible]

[illegible]

After Generalization :

[illegible]

Base condition T(0) == 1
n – k = 0
n = k;
put, k = n

[illegible]

It is GP series, and sum is [illegible]

[illegible], or you can say $O(2^n)$ which is exponentioal

Source

https://www.geeksforgeeks.org/time-complexity-analysis-tower-hanoi-recursion/

Chapter 40

Time Complexity of Loop with Powers

Time Complexity of Loop with Powers - GeeksforGeeks

What is the time complexity of below function?

```
 void fun(int n, int k)
{
    for (int i=1; i<=n; i++)
    {
      int p = pow(i, k);
      for (int j=1; j<=p; j++)
      {
          // Some O(1) work
      }
    }
}
```

Time complexity of above function can be written as $1^k + 2^k + 3^k + \ldots n1^k$.

Let us try few examples:

```
k=1
Sum = 1 + 2 + 3 ... n
    = n(n+1)/2
    = n2 + n/2

k=2
Sum = 12 + 22 + 32 + ... n12.
    = n(n+1)(2n+1)/6
    = n3/3 + n2/2 + n/6
```

```
k=3
Sum = 13 + 23 + 33 + ... n13.
    = n2(n+1)2/4
    = n4/4 + n3/2 + n2/4
```

In general, asymptotic value can be written as $\mathbf{(n^{k+1})/(k+1) + \Theta(n^k)}$

Note that, in asymptotic notations like Θ we can always ignore lower order terms. So the time complexity is $\mathbf{\Theta(n^{k+1} / (k+1))}$

Source

https://www.geeksforgeeks.org/time-complexity-of-loop-with-powers/

Chapter 41

Time Complexity of a Loop when Loop variable "Expands or Shrinks" exponentially

Time Complexity of a Loop when Loop variable "Expands or Shrinks" exponentially - GeeksforGeeks

For such cases, time complexity of the loop is O(log(log(n))).The following cases analyse different aspects of the problem.

Case 1 :

```
 for (int i = 2; i <=n; i = pow(i, k))
{
    // some O(1) expressions or statements
}
```

In this case, i takes values 2, 2^k, $(2^k)^k = 2^{k^2}$, $(2^{k^2})^k = 2^{k^3}$, …, $2^{k^{\log_k(\log(n))}}$. The last term must be less than or equal to n, and we have $2^{k^{\log_k(\log(n))}} = 2^{\log(n)} = n$, which completely agrees with the value of our last term. So there are in total $\log_k(\log(n))$ many iterations, and each iteration takes a constant amount of time to run, therefore the total time complexity is O(log(log(n))).

Case 2 :

```
 // func() is any constant root function
for (int i = n; i > 0; i = func(i))
{
   // some O(1) expressions or statements
}
```

In this case, i takes values n, $n^{1/k}$, $(n^{1/k})^{1/k} = n^{1/k^2}$, n^{1/k^3}, …, $n^{1/k^{\log_k(\log(n))}}$, so there are in total $\log_k(\log(n))$ iterations and each iteration takes time O(1), so the total time complexity is O(log(log(n))).

Refer below article for analysis of different types of loops.
https://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/

Source

https://www.geeksforgeeks.org/time-complexity-loop-loop-variable-expands-shrinks-exponentially/

Chapter 42

Time Complexity of building a heap

Time Complexity of building a heap - GeeksforGeeks

Consider the following algorithm for building a Heap of an input array A.

```
BUILD-HEAP(A)
    heapsize := size(A);
    for i := floor(heapsize/2) downto 1
        do HEAPIFY(A, i);
    end for
END
```

A quick look over the above algorithm suggests that the running time is $O(n\log(n))$, since each call to **Heapify** costs $O(\log(n))$ and **Build-Heap** makes $O(n)$ such calls.
This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the running time of **Heapify** depends on the height of the tree 'h' (which is equal to lg(n), where n is number of nodes) and the heights of most sub-trees are small.
The height 'h' increases as we move upwards along the tree. Line-3 of **Build-Heap** runs a loop from the index of the last internal node (heapsize/2) with height=1, to the index of root(1) with height = lg(n). Hence, **Heapify** takes different time for each node, which is $O(h)$.

For finding the Time Complexity of building a heap, we must know the number of nodes having height h.

For this we use the fact that, A heap of size n has at most [illegible] 2^{h+1} [illegible] nodes with height h.

Now to derive the time complexity, we express the total cost of **Build-Heap** as-

[illegible]

(1)

Step 2 uses the properties of the Big-Oh notation to ignore the ceiling function and the constant 2([illegible]). Similarly in Step three, the upper limit of the summation can be increased to infinity since we are using Big-Oh notation.

Sum of infinite G.P. ($x < 1$)

[illegible]

(2)

On differentiating both sides and multiplying by x, we get

[illegible]

(3)

Putting the result obtained in (3) back in our derivation (1), we get

[illegible]

$$= O(n * 2)$$

(4) $$= O(n)$$

Hence Proved that the Time complexity for Building a Binary Heap is $O(n)$.

Reference :
http://www.cs.sfu.ca/CourseCentral/307/petra/2009/SLN_2.pdf

Source

https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/

Chapter 43

Time Complexity where loop variable is incremented by 1, 2, 3, 4 ..

Time Complexity where loop variable is incremented by 1, 2, 3, 4 .. - GeeksforGeeks

What is the time complexity of below code?

```
 void fun(int n)
{
   int j = 1, i = 0;
   while (i < n)
   {
       // Some O(1) task
       i = i + j;
       j++;
   }
}
```

The loop variable 'i' is incremented by 1, 2, 3, 4, … until i becomes greater than or equal to n.

The value of i is x(x+1)/2 after x iterations. So if loop runs x times, then $x(x+1)/2 < n$. Therefore time complexity can be written as $\Theta(\sqrt{n})$.

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/time-complexity-where-loop-variable-is-incremented-by-1-2-3-4/

Chapter 44

Time complexity of recursive Fibonacci program

Time complexity of recursive Fibonacci program - GeeksforGeeks

The Fibonacci numbers are the numbers in the following integer sequence 0, 1, 1, 2, 3, 5, 8, 13…
Mathematically Fibonacci numbers can be written by the following recursive formula.

```
For seed values F(0) = 0 and F(1) = 1
F(n) = F(n-1) + F(n-2)
```

Before proceeding with this article make sure you are familiar with the recursive approach discussed in Program for Fibonacci numbers

Analysis of the recursive Fibonacci program:

We know that the recursive equation for Fibonacci is $T(n) = T(n-1) + T(n-2) + O(1)$.
What this means is, the time taken to calculate fib(n) is equal to the sum of time taken to calculate fib(n-1) and fib(n-2). This also includes the constant time to perform the previous addition.

On solving the above recursive equation we get the upper bound of Fibonacci as $O(2^n)$ but this is not the tight upper bound. The fact that Fibonacci can be mathematically represented as a linear recursive function can be used to find the tight upper bound.
Now Fibonacci is defined as

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

The characteristic equation for this function will be

$$x^2 = x + 1$$
$$x^2 - x - 1 = 0$$

Solving this by quadratic formula we can get the roots as
$x = (1+\sqrt{5})/2$ and $x=(1-\sqrt{5})/2$
Now we know that solution of a linear recursive function is given as
$F(n) = (\alpha_1)^n + (\alpha_2)^n$
where α_1 and α_2 are the roots of the characteristic equation.
So for our Fibonacci function $F(n) = F(n-1) + F(n-2)$ the solution will be

$F(n) = ((1+\sqrt{5})/2)^n + ((1-\sqrt{5})/2)^n$

Clearly $T(n)$ and $F(n)$ are asymptotically the same as both functions are representing the same thing.
Hence it can be said that
$T(n) = O(((1+\sqrt{5})/2)^n + ((1-\sqrt{5})/2)^n)$
or we can write below (using the property of Big O notation that we can drop lower order terms)
$T(n) = O(((1+\sqrt{5})/2)^n)$
$T(n) = O(1.6180)^n$
This is the tight upper bound of fibonacci.\

Fun Fact:
1.6180 is also called the golden ratio. You can read more about golden ratio here: Golden Ratio in Maths

Source

https://www.geeksforgeeks.org/time-complexity-recursive-fibonacci-program/

Chapter 45

Time taken by Loop unrolling vs Normal loop

Time taken by Loop unrolling vs Normal loop - GeeksforGeeks

We have discussed loop unrolling. The idea is to increase performance by grouping loop statements so that there are less number of loop control instruction and loop test instructions

C++

```
 // CPP program to compare normal loops and
// loops with unrolling technique
#include <iostream>
#include <time.h>
using namespace std;

int main() {

  // n is 8 lakhs
  int n = 800000;

  // t to note start time
  clock_t t = clock();

  // to store the sum
  long long int sum = 0;

  // Normal loop
  for (int i = 1; i <= n; i++)
    sum += i;

  // to mark end time
  t = clock() - t;
  cout << "sum is: " << sum << endl;
  cout << "time taken by normal loops:"
      "(float)t / CLOCKS_PER_SEC << " seconds"
       << endl;

  // to mark start time of unrolling
  t = clock();

  // Unrolling technique (assuming that
  // n is a multiple of 8).
  sum = 0;
  for (int i = 1; i <= n; i += 8) {
    sum += i sum += (i + 1);
    sum += (i + 2);
    sum += (i + 3);
    sum += (i + 4);
    sum += (i + 5);
    sum += (i + 6);
    sum += (i + 7);
  }

  // to mark the end of loop
  t = clock() - t;

  cout << "Sum is: " << sum << endl;
  cout << "Time taken by unrolling: "
       "(float)t / CLOCKS_PER_SEC << " seconds";
  return 0;
}
```

Java

```
 // Java program to compare
// normal loops and loops
// with unrolling technique

class GFG
{

    public static void main(String[] args)
    {

    // n is 8 lakhs
    int n = 800000;

    // t to note start time
    double t = (double)System.nanoTime();
    // to store the sum
    long sum = 0;

    // Normal loop
    for (int i = 1; i <= n; i++)
        sum += i;

    // to mark end time
    t = (double)System.nanoTime() - t;

    System.out.println("sum is: "+
        Double.toString(sum));
    System.out.println("time taken by normal loops:" +
        Double.toString(t / Math.pow(10.0, 9.0)));

    // to mark start time
    // of unrolling
    t = (double)System.nanoTime();

    // Unrolling technique
    // (assuming that n is
    // a multiple of 8).
    sum = 0;
    for (int i = 1; i <= n; i += 8)
    {
        sum += i ;
        sum += (i + 1);
        sum += (i + 2);
        sum += (i + 3);
        sum += (i + 4);
        sum += (i + 5);
        sum += (i + 6);
        sum += (i + 7);
    }

    // to mark the end of loop
    t = (double)System.nanoTime() - t;

    System.out.println("sum is: " +
        Double.toString(sum));
    System.out.println("time taken by normal loops:" +
        Double.toString(t / Math.pow(10.0, 9.0)));
    }
}

// This code is contributed
// by Harshit Saini
```

Python3

```
 # Python program to compare
# normal loops and loops
# with unrolling technique
from timeit import default_timer as clock

if __name__ == "__main__":

    # n is 8 lakhs
    n = 800000;

    #t to note start time
    t = clock()

    # to store the sum
    sum = 0

    # Normal loop
    for i in range(1, n + 1):
        sum += i

    # to mark end time
    t = clock() - t

    print("sum is: " + str(sum))
    print("time taken by normal " +
                 "loops:" + str(t))

    # to mark start
    # time of unrolling
    t = clock()

    # Unrolling technique
    # (assuming that n is
    # a multiple of 8).
    sum = 0
    for i in range(1, n + 1, 8):
        sum += i
        sum += (i + 1)
        sum += (i + 2)
        sum += (i + 3)
        sum += (i + 4)
        sum += (i + 5)
        sum += (i + 6)
        sum += (i + 7)

    # to mark the
    # end of loop
    t = clock() - t

    print("Sum is: " + str(sum))
    print("Time taken by unrolling: " +
                              str(t))

# This code is contributed
# by Harshit Saini
```

Output:

```
Sum is: 320000400000
Time taken: 0.002645 seconds

Sum is: 320000400000
Time taken: 0.001799 seconds
```

Please refer loop unrolling for comparison of normal loops and loop unrolling.

Improved By : Harshit Saini

Source

https://www.geeksforgeeks.org/time-taken-loop-unrolling-vs-normal-loop/

Chapter 46

Understanding Time Complexity with Simple Examples

Understanding Time Complexity with Simple Examples - GeeksforGeeks

Lot of students get confused while understanding the concept of time-complexity, but in this article we will explain it with a very simple example:

Imagine a classroom of 100 students in which you gave your pen to one person. Now, you want that pen. Here are some ways to find the pen and what the O order is.

O(n^2): You go and ask the first person of the class, if he has the pen. Also, you ask this person about other 99 people in the classroom if they have that pen & So on,
This is what we call O(n^2).

O(n): Going and asking each student individually is O(N).

O(log n): Now I divide the class in two groups, then ask: "Is it on the left side, or the right side of the classroom?" Then I take that group and divide it into two and ask again, and so on. Repeat the process till you are left with one student who has your pen. This is what you mean by O(log n).

I might need to do the O(n^2) search if only one student knows on which student the pen is hidden. I'd use the O(n) if one student had the pen and only they knew it. I'd use the O(log n) search if all the students knew, but would only tell me if I guessed the right side.

Another Example

Time Complexity of algorithm/code is **not** equal to the actual time required to execute particular code but the number of times a statement execute. We can prove this by using time command. For example, Write a code in C/C++ or any other language to find maximum between N numbers, where N varies from 10, 100, 1000, 10000. And compile that code on Linux based operating system (Fedora or Ubuntu) with below command:

```
gcc program.c -o program
run it with time ./program
```

You will get surprising results i.e. for N = 10 you may get 0.5ms time and for N = 10, 000 you may get 0.2 ms time. Also you will get different timings on different machine. So, we can say that actual time require to execute code is machine dependent (whether you are using pentium1 or pentiun5) and also it considers network load if your machine is in LAN/WAN. Even you will not get same timings on same machine for same code, the reason behind that the current network load.

Now, the question arises if time complexity is not the actual time require executing the code then what is it?
The answer is : Instead of measuring actual time required in executing each statement in the code, we consider how many times each statement execute.
For example:

```
 #include <stdio.h>
int main()
{
    printf("Hello World");
}
```

Output:

```
Hello World
```

In above code "Hello World!!!" print only once on a screen. So, time complexity is constant: O(1) i.e. every time constant amount of time require to execute code, no matter which operating system or which machine configurations you are using.
Now consider another code:

```
 #include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i++) {
        printf("Hello Word !!!");
    }
}
```

Output:

```
Hello Word !!!Hello Word !!!Hello Word !!!Hello Word !!!
Hello Word !!!Hello Word !!!Hello Word !!!Hello Word !!!
```

In above code "Hello World!!!" will print N times. So, time complexity of above code is O(N).

Source : Reddit

The co-author of this article is **Varsha Lokare.**

Source

https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/

Chapter 47

What does ‘Space Complexity’ mean?

What does ’Space Complexity’ mean? - GeeksforGeeks

Space Complexity:
The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.

Auxiliary Space is the extra space or temporary space used by an algorithm.

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort and Heap Sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.

Source

https://www.geeksforgeeks.org/g-fact-86/