Complexity

Rupesh Nasre.
rupesh@cse.iitm.ac.in

August 2021

Algorithms

- For the same problem, there could be multiple algorithms.
- An algorithm is a clearly specified sequence of simple instructions that solve a given problem.
  - An algorithm, by definition, terminates.
  - Otherwise, the sequence of instructions constitutes a *procedure*.
- The algorithm should be so clear to you that you should be able to make a machine understand it.
  - This is called **programming**.

Algorithm Efficiency

- For the same problem, there could be multiple algorithms.
- We prefer the ones that run *fast*.
  - I don't want an algorithm that takes a year to sort!
  - By the way, there are computations that run for months!
  - Operating systems on servers may run for years.

```
[rupesh@aqua ~]$ uptime
 17:15:41 up 228 days,  3:50, 155 users,  load average: 0.21, 0.22, 0.26
```

- We would like to compare algorithms based on their speeds.
  - Mathematical model to capture algorithm efficiency.

Misconceptions

- Program P1 takes 10 seconds, P2 takes 20 seconds, so I would choose P1.
  - Execution time is input-dependent.
  - Execution time is hardware-dependent.
  - Execution time is machine-load dependent.
  - Execution time is run-dependent too!
  - Other factors play a role; for instance:

    whether the program is running in hostel or in DCF

    Or whether in Chennai or Kashmir

    Or whether in May or December!

```
$ time ls >/dev/null

real 0m0.002s
user0m0.000s
sys  0m0.001s


// on another machine
$ time ls >/dev/null

real 0m0.005s
user0m0.000s
sys  0m0.004s
```

Examples

```
a = a + b;
b = a – b;
a = a – b;
```

Irrespective of the values of a and b, this program would take time proportional to three instructions.

```
for (ii = 0; ii < N; ++ii)
    a[ii] = 0;
```

Proportional to N.

```
for (ii = 0; ii < N; ++ii)
    for (jj = 0; jj < M; ++jj)
        mat[ii][jj] = ii + jj;
```

Proportional to N*M.

```
int fun(int n) {
    return (n == 0 ? 1 : 4 * fun(n / 3));
}
```

?

Examples

```
a = a + b;
b = a – b;
a = a – b;
```

```
a[ii] = 0;
```

```
x = y;
if (x > 0)
    y = x + 1;
else
    z = x + 1;
```

```
for (ii = 0; ii < 1000; ++ii)
    a[ii] = 0;
```

All of these are equally efficient!

- They all perform constant-time operations.
- We denote those as O(1).

Examples

```
a[0] = 0;
a[1] = 0;
a[2] = 0;
...
a[n – 5] = 0;
```

```
int fact(int n) {
    if (n > 0) return n * fact(n – 1);
    return 1;
}
```

```
for (ii = 0; ii < 2*n; ++ii)
    a[ii] = 0;
```

All of these are equally efficient!

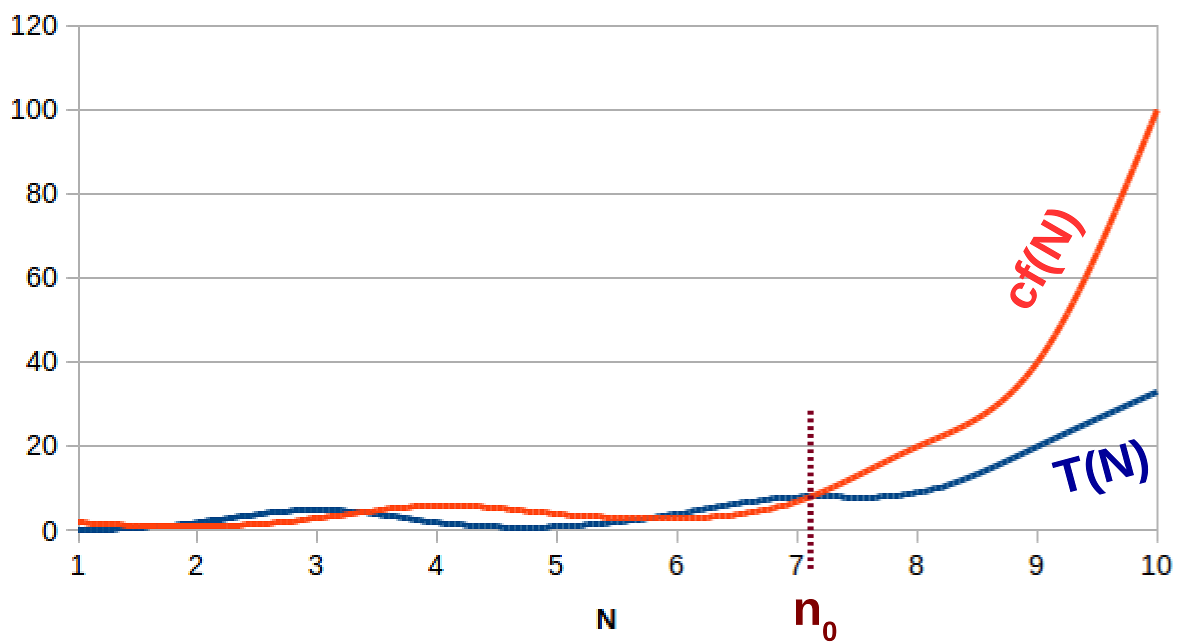
- They all perform linear-time operation *(linear in n)*.
- We denote those as O(n).

Definition

- $T(N) = O(1)$ if $T(N) \le c$ when $N \ge n_0$, for some positive c and n_0.
- $T(N) = O(N)$ if $T(N) \le cN$ when $N \ge n_0$, for some positive c and n_0.
- In general,

  $T(N) = O(f(N))$ if there exist positive constants c and n_0 such that $T(N) \le cf(N)$ when $N \ge n_0$.
- Complexity captures the rate of growth of a function.

Big O



- In general,

  $T(N) = O(f(N))$ if there exist positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

- The complexity is upper-bounded by c*f(N).
- Thus, big O is the worst-case complexity.

Examples

```
a = a + b;
b = a – b;
a = a – b;
```

O(1)

Irrespective of the values of a and b, this program would take time proportional to three instructions.

```
for (ii = 0; ii < N; ++ii)
    a[ii] = 0;
```

O(N)

Proportional to N.

```
for (ii = 0; ii < N; ++ii)
    for (jj = 0; jj < M; ++jj)
        mat[ii][jj] = ii + jj;
```

O(N*M)

Proportional to N*M.

```
int fun(int n) {
    return (n <= 1 ? 1 : 4 * fun(n / 3));
}
```

?

Solving for Time Complexity

$T(n) = c1 + T(n/3)$ and $T(1) = 1$

$= c1 + [c1 + T(n/9)]$

$= 2*c1 + T(n/3^2)$

$= 3*c1 + T(n/3^3)$

$= k*c1 + T(n/3^k)$

If $n == 3^k$,

$T(n) = \log_3 n * c1 + T(1) = c1*\log_3 n + 1 = O(\log_3 n)$

```
int fun(int n) {
    return (n <= 1 ? 1 : 4 * fun(n / 3));
}
```

Types of Complexities

Symbol	Name	Bound	Equation
O(...)	Big O	Upper	T(n) <= cf(n)
Ω(...)	Big Omega	Lower	T(n) >= cf(n)
Θ(...)	Theta	Upper and Lower	c_1f(n) <= T(n) <= c_2f(n)
o(...)	Little O	Strictly Upper	T(n) < cf(n)
ω(...)	Little Omega	Strictly Lower	T(n) > cf(n)

Notes

- Θ means O and Ω. It is a stronger guarantee on the complexity.
- If $T(n)$ is $O(n)$, then $T(n)$ is also $O(n^2)$, also $O(n\log n)$, also $O(n^3)$, $O(n^{100})$, $O(2^n)$; but it is <u>not</u> $O(\log n)$ or $O(1)$.
- Big O is also called Big Oh.
- $T(n) = T(n/2) = T(1000n) = T(n\log 2) = T(2^{\log n})$
- $\mathrm{Log}_2(x)$, that is, *log to the base 2* is sometimes written as lg(x).
- If $T(n) = O(f(n))$ then $f(n) = \Omega(T(n))$.

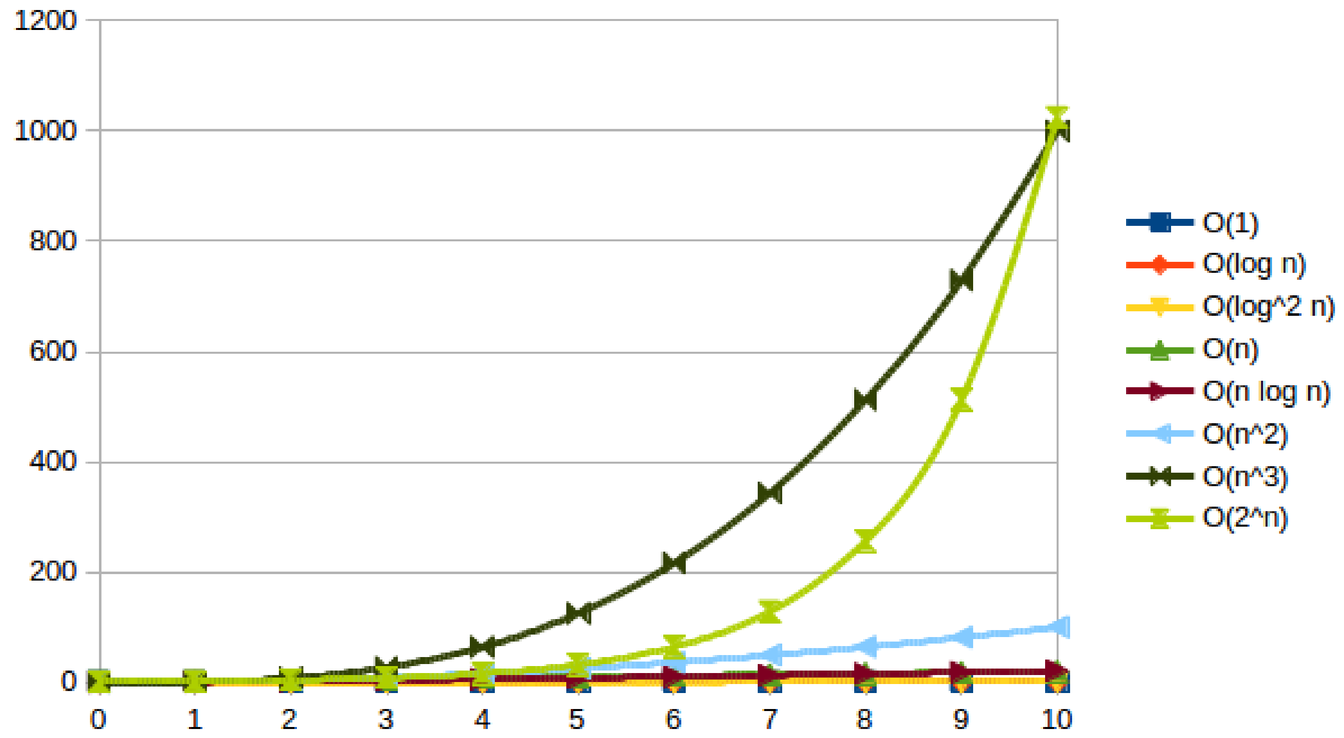
Complexity Arithmetic

- If T1(n) = O(f(n)) and T2(n) = O(g(n)), then
  - T1(n) + T2(n) = **max**(O(f(n), O(g(n)))
  - T1(n) * T2(n) = O(f(n) ***** g(n))

- **Classwork**:
  - Write a C code that requires the use of T1(n) + T2(n).
  - Write a C code that requires the use of T1(n) * T2(n).

Typical Complexities

Function	Name
c	Constant
Log N	Logarithmic
Log^2 N	Log-squared
N	Linear
N log N	Superlinear
N^2	Quadratic
N^3	Cubic
2^N	Exponential



Homework: Find which one grows faster: nlogn or $n^{1.5}$.

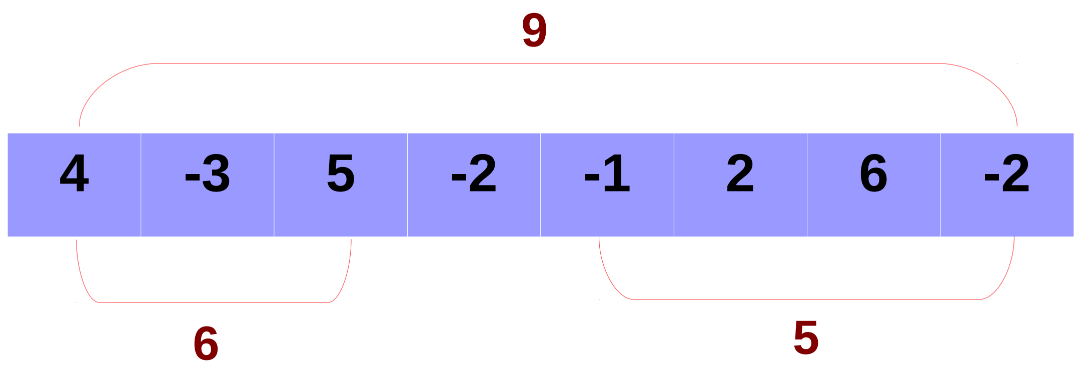
Complexity Comparison

- Given two complexity functions f(n) and g(n), we can determine relative growth rates using *$\lim_{n\to\infty} f(n) / g(n)$*, using L'Hospital's rule.
- Four possible values:
  - The limit is zero, implies $f(n) = o(g(n))$.
  - The limit is $c \neq 0$, implies $f(n) = \Theta(g(n))$.
  - The limit is ∞, implies $g(n) = o(f(n))$.
  - The limit oscillates, implies there is no relation.

Facets of Efficiency

- An algorithm or its implementation may have various facets towards efficiency.
  - Time complexity (which we usually focus on)
  - Space complexity (considered in memory-critical systems such as embedded devices)
  - Energy complexity (e.g., your smartphones)
  - Security level (e.g., program with less versus more usage of pointers)
  - I/O complexity
  - ...

Max. Subsequence Sum



- **Problem Statement**

  Given an array of (positive, negative, zero) integer values, find the largest subsequence sum.

- A subsequence is a consecutive set of elements. If empty, its sum is zero.

MSS: Algorithm 1

Exhaustive Algorithm

For each possible subsequence

    Compute sum

    If sum > current maxsum

        current maxsum = sum

Return current maxsum

How many subsequences?

What is the complexity of this part?

Algorithm 1 takes $O(N^3)$ running time.

Source: mss1.cpp

MSS: Algorithm 1

- Did we perform a tight mathematical analysis?
- To be precise, we need the following number of operations:

$$\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \boxed{\sum_{k=i}^{j} O(1)}$$

$j - i + 1$

We will assume O(1) to be equal to constant 1. This would affect only the constant in BigOh.

MSS: Algorithm 1

- Did we perform a tight mathematical analysis?
- To be precise, we need the following number of operations:

$$\sum_{i=0}^{N-1} \underbrace{\boxed{\sum_{j=i}^{N-1} (j - i + 1)}}_{\substack{\text{sum of first N-i integers} \\ = \\ (N - i)(N - i + 1) / 2}}$$

MSS: Algorithm 1

- Did we perform a tight mathematical analysis?
- To be precise, we need the following number of operations:

$$\sum_{i=0}^{N-1} (N - i)(N - i + 1) / 2$$

$= (N^3 + 3N^2 + 2N) / 6$

$= O(N^3)$

The analysis is tight.
Is the algorithm tight?

MSS: Algorithm 2

- **Observation:**

$$\sum_{k=i}^{j} \mathbf{A[k]} = \mathbf{A[j]} + \sum_{k=i}^{j-1} \mathbf{A[k]}$$

For each starting position i

For each ending position j

Incrementally compute sum

If sum > maxsum

maxsum = sum

Return maxsum

What is the complexity of this algorithm?

Source: **mss2.cpp**

MSS: Algorithm 3

- **Observation**: Discard fruitless subsequences early.
  - e.g., in {1, 2, -8, 4, -3, 5, -2, -1, 2, 6, -2}, we need not consider subsequences {-2} or {-2, -1} or even {-2, -1, 2} or {1, 2, -8, 4}.

For each position

Add next element to sum

If sum > maxsum

Maxsum = sum

Else if sum is negative

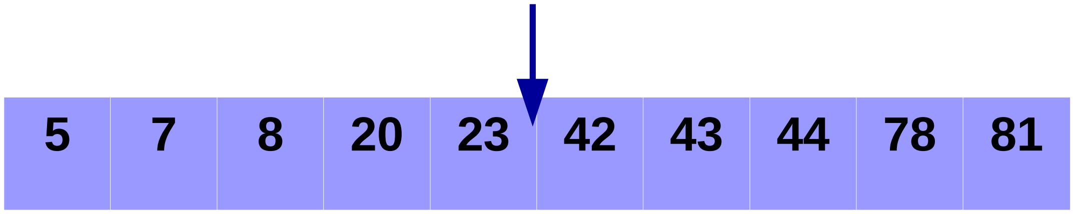
sum = 0

Are you kidding?
This shouldn't work.
This is linear time algorithm!

Source: mss3.cpp

Binary Search

- Go to page number 44.
- Searching in an array takes linear time O(N).
- If the array is sorted already, we can do better.
- We can cut the *search space* by half at every step.



Classwork: Write the code for binary search.
Source: **bsearch.cpp**

Binary Search

- Constant amount of time required to
  - Find the mid element.
  - Check if it is the element to be searched.
  - Decide whether to go to the left or the right.
  - Cut the search space by half.
- $T(N) = T(N/2) + O(1)$
  - Thus, $T(N)$ is $O(\log N)$.

Exercises

- Solve exercises at the end of Chapter 2 of Weiss’s book.