COMP2421 — DATA STRUCTURES AND ALGORITHMS

Asymptotic Time Analysis

Dr. Radi Jarrar
Department of Computer Science
Birzeit University

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Introduction

- Algorithm: a specified set of instructions to be followed to solve a problem.
- Once an algorithm is determined to be true, it is important to determine if the algorithm at hand is “good” enough or not.
- This requires a formal method to analyze the algorithm.
- The main recourses to consider when running an algorithm are the **1) time; & 2) space** the algorithm will require.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

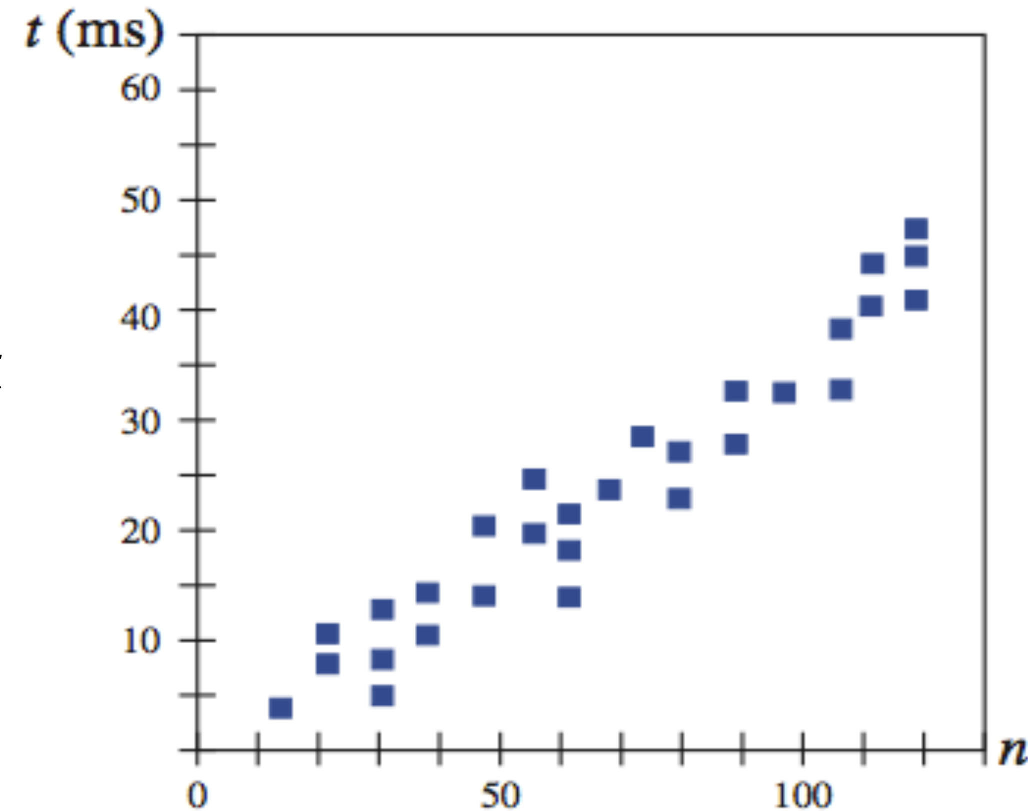
Introduction (2)

- In this chapter we will understand how to estimate the time a program will take.
- General point: the running time of an algorithm increases with the input size. We want to study the relation between the input size and running time of an algorithm.
- We are interested in characterizing an algorithm's running time as a function of the input size. How to measure it?

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Experimental Studies

- If an algorithm is implemented, we can study its running time by executing it on various test inputs and recording the actual time spent in each execution.



STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Experimental Studies (2)

- Experiments can be done on a limited set of input tests
- Difficult to compare two algorithms except if the experiments were held on exactly the same hardware and the same software environments.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Types of Time Analysis

- Best case analysis
  - *Optimistic/not realistic*
- Average case analysis
  - *Is based on statistical methods*
- Worst case analysis
  - *The program's worse case cannot exceed this analysis*

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Primitive Operations

- *We should determine how many steps the algorithm has to perform as a function of the input size in the worst case.*
- Primitive operations: basic computations performed by an algorithm. Assumed to take a constant amount of time in the computer memory.
- Counting primitive operations: this is done by inspecting the pseudocode. We can determine the max number of primitive operations executed by an algorithm as a function of the input size.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Primitive Operations (2)

- Assigning a value to a variable
- Calling a function
- Returning from a function
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Indexing into an array
- Following an object reference

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Primitive Operations (2)

- ***Example 1:***

```
Algorithm  ArrayMax(A, n){
    currentMax ← A[0]
    for i ← 1 to n-1 do
        if A[i] > currentMax then
            currentMax ← A[i]
        increment i //i=i+1

    return currentMax
```

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Primitive Operations (2)

- ***Example 1:***

Algorithm	#operations
`Algorithm ArrayMax(A, n){`	
`currentMax ← A[0]`	2
`for i ← 1 to n-1 do`	2n
`if A[i] > currentMax then`	2(n-1)
`currentMax ← A[i]`	2(n-1)
`increment i //i=i+1`	2(n-1)
`return currentMax`	1
	Total: 8n-3

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Primitive Operations (3)

- *Example* 2: Algorithm **a** that takes an input of positive integer n, which is a power of 2. Output: integer m such that $2^m=n$

```
m ← 0
   while (n>=2)
      n ← n/2
      m++
return m
```

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Primitive Operations (3)

- *Example* 2: Algorithm **a** that takes an input of positive integer n, which is a power of 2. Output: integer m such that $2^m = n$

Algorithm	Operations
m ← 0	1
while (n>=2)	$\log_2(n)$
n ← n/2	$2\log_2(n)$
m++	$2\log_2(n)$
return m	1
	Total: $5\log_2(n)+2$

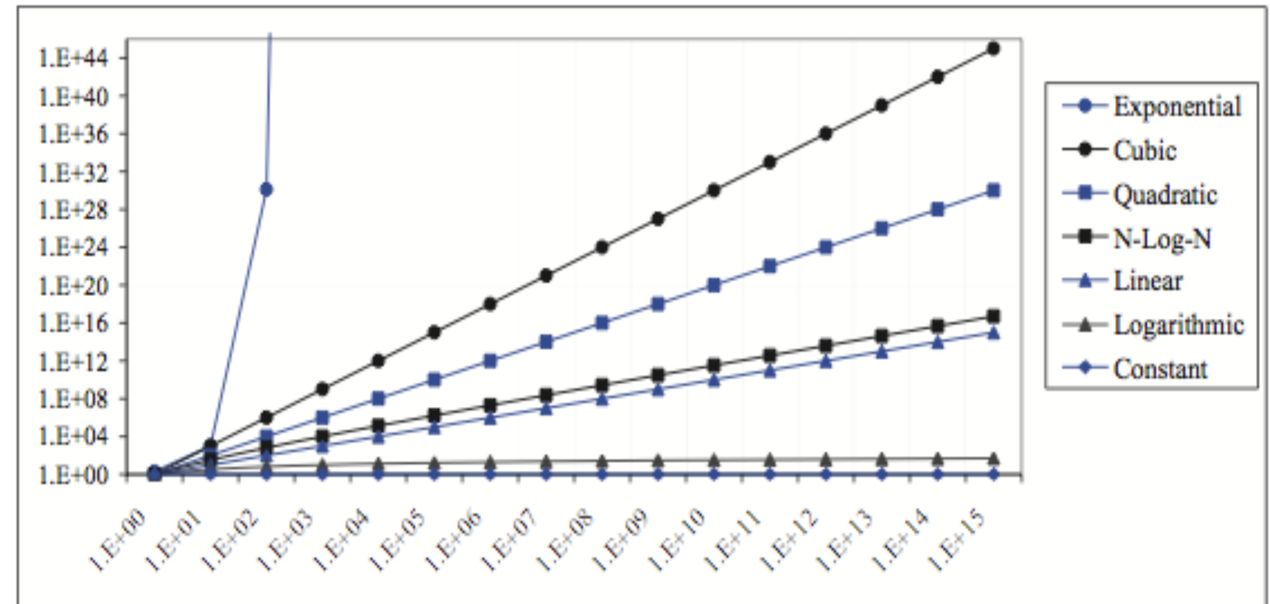
STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Growth Rate of Functions

- Growth rate functions: there are 7 functions that appear in algorithm analysis.

Constant	Logarithmic	Linear	n-log-n	Quadratic	Cubic	Exponential
C or 1	Log n	n	n log n	n^2	n^3	2^n

- *Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or n-log-n time. Algorithms with quadratic or cubic running times are less practical, but algorithms with exponential running times are infeasible for all but the smallest sized inputs.*



STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Analysing Functions

- The general way of analyzing the running time of an algorithm should:
  1. Take into account all possible inputs;
  2. Evaluate the relative efficiency of any two algorithms in independently from the hardware and software environment; and
  3. Performed by studying a high-level description of the algorithm without implementing it or running experiments on it.

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Asymptotic Analysis

- Asymptotic analysis: In the algorithm analysis, we focus on the growth rate of the running time as a function of the input size n.
- That is, we characterize the running times of algorithms by using functions that map the size of the input, n, to values that correspond to the main factor that determines the growth rate in terms of n.
- This approach allows us to focus on the big picture aspects of an algorithm's running time.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Asymptotic Analysis (2)

- The time functions will be represented as T(n) using the "big-O" notation to express an algorithm runtime complexity. For example, the following statement

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

means that an algorithm has a quadratic time complexity.

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Asymptotic Analysis (3)

The analysis required to estimate the resource use of an algorithm is generally a theoretical issue, and therefore a formal framework is required. We begin with some mathematical definitions.

Throughout the book we will use the following four definitions:

Definition 2.1.
$T(N) = O(f(N))$ if there are positive *constants* c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Definition 2.2.
$T(N) = \Omega(g(N))$ if there are positive *constants* c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.

Definition 2.3.
$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Asymptotic Analysis (4)

- The idea of these definitions is to establish a relative order among functions. Given two functions, there are usually points where one function is smaller than the other function, so it does not make sense to claim, for instance, $f(N) < g(N)$. Thus, we compare their **relative rates of growth.**

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Big-Oh Notation

- **Def#1**: Let $T(N)$ and $f(N)$ be functions mapping nonnegative integers to real numbers. We say that $T(N)$ is $O(f(N))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $T(N) \leq c.f(N)$ for $N \geq N_0$.
- This definition is referred as Big-Oh notation, for it is sometimes pronounced as "T(N) is a big-oh of f(N)". We can also say T(N) is order of f(N).
- It means that the function T(N) does not grow faster than f(N). In other words, f(N) is the upper bound of T(N).

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Big-Oh Notation (2)

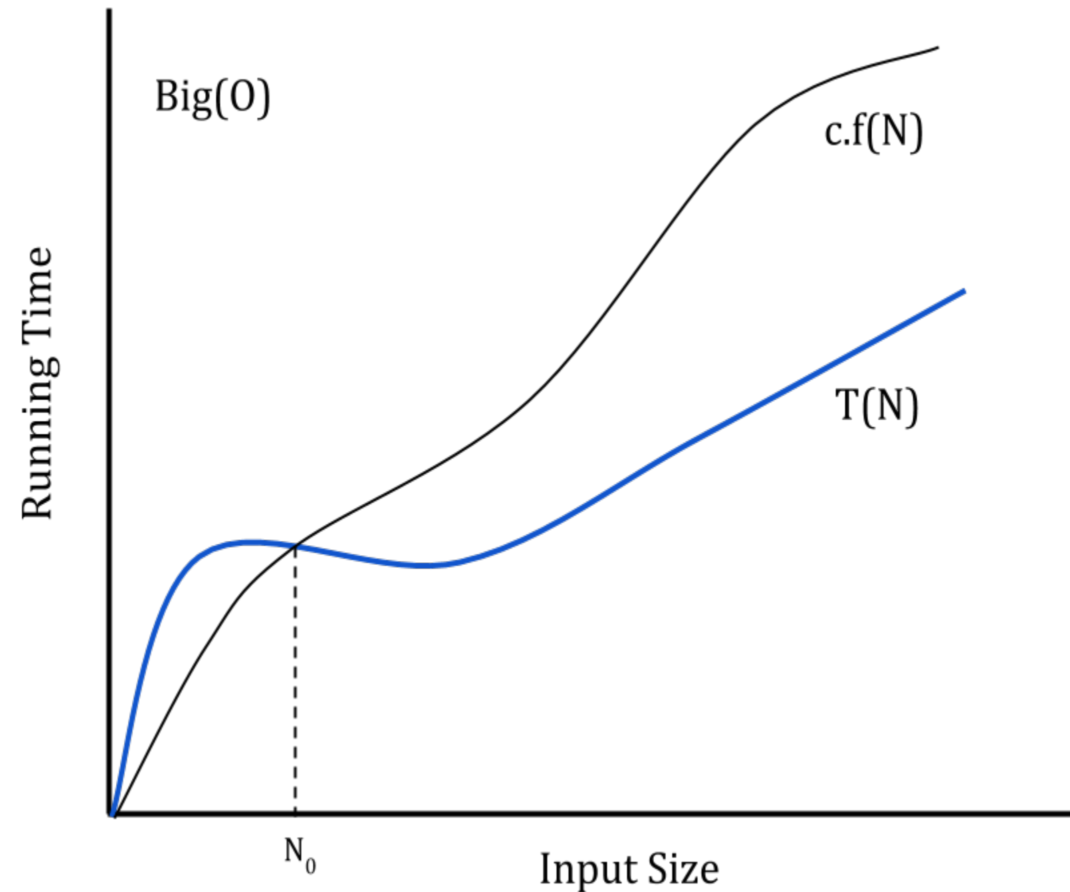


Fig. The Big-Oh notation. The function T(N) is O(f(N)), since $T(N) \le c.f(N)$ when $N \ge N_0$

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Big-Oh Notation (2)

T(n) is O(f(n)) we are guaranteeing that the function T(n) grows at a rate no faster than f(n). Thus, f(n) is the upper bound on T(n)

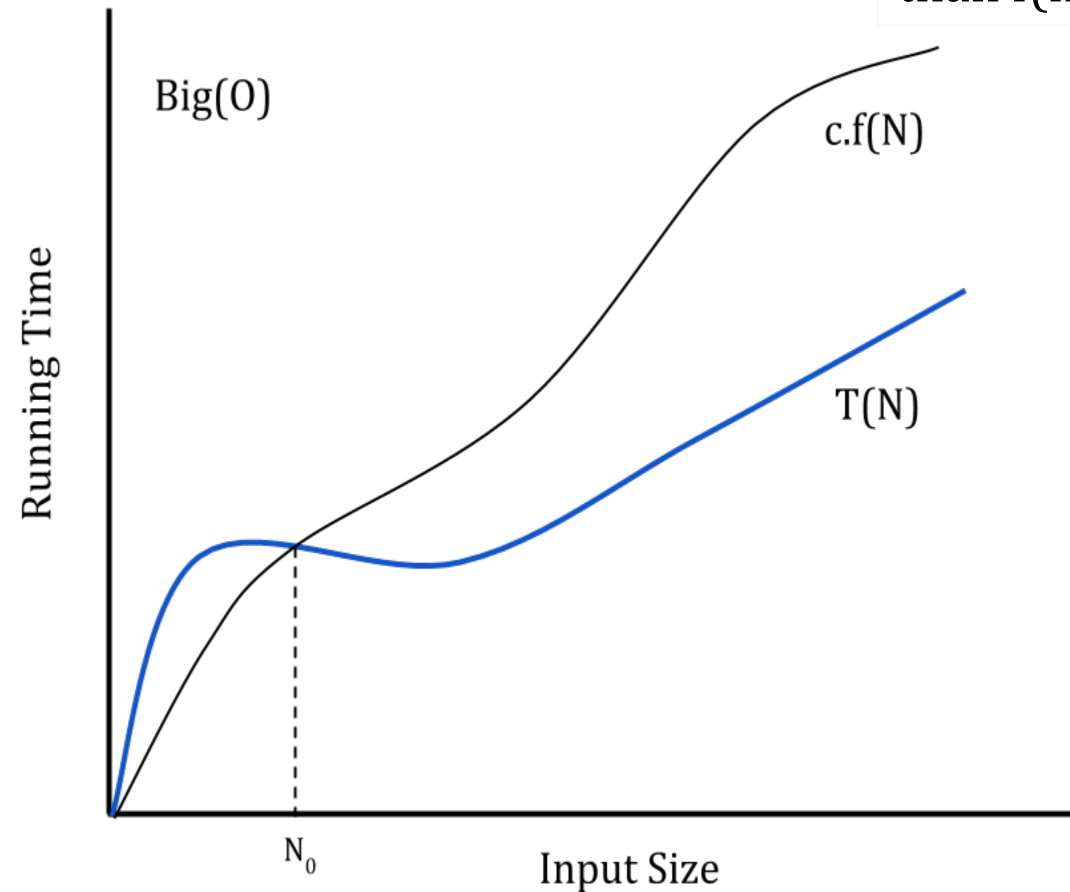


Fig. The Big-Oh notation. The function T(N) is O(f(N)), since $T(N) \leq c.f(N)$ when $N \geq N_0$

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Big-Oh Notation (3)

- **Examples:**
- $1 = O(n)$
- $n = O(n^2)$
- $\log(n) = O(n)$
- $3n + 2 = O(n)$

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Big-Oh Notation (4)

- **Example**: Although 1,000N is larger than N^2 for small values of N, N^2 grows at a faster rate, and thus N^2 will eventually be the larger function. The turning point is N = 1,000 in this case. The first definition says that eventually there is some point n_0 past which $c \cdot f(N)$ is always at least as large as T(N), so that if constant factors are ignored, f(N) is at least as big as T(N). In our case, we have T(N) = 1,000N, f(N) = N^2, n_0 = 1,000, and c=1. We could also use n_0 = 10 and c = 100. Thus, we can say that 1,000N = $O(N^2)$ (order N-squared).

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

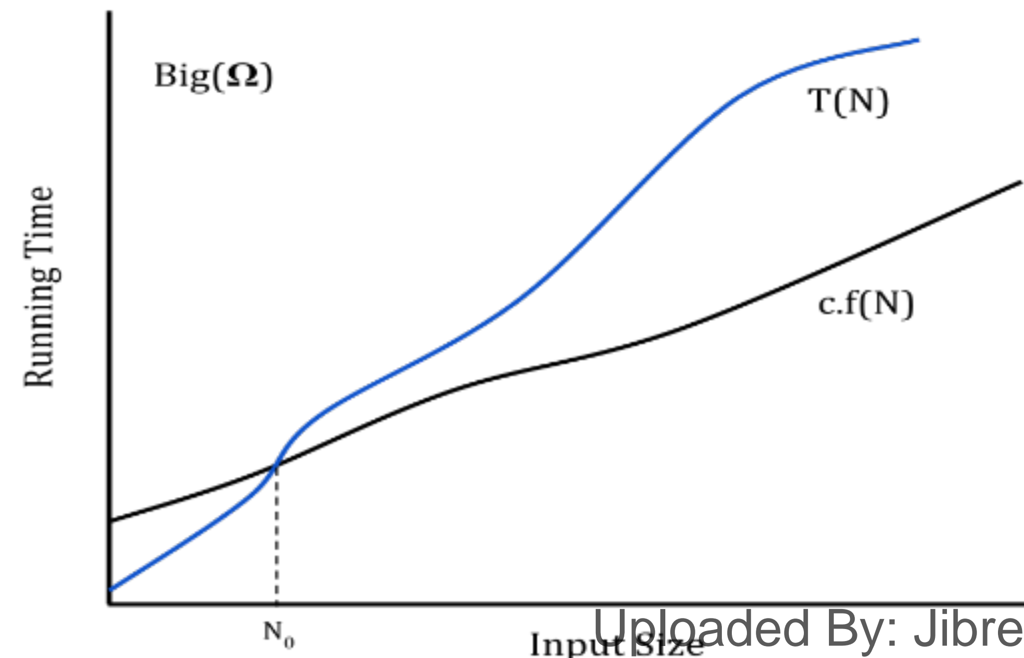
Big-Oh Notation (5)

- The big-Oh notation allows us to say that a function T(N) is "less than or equal to" another function F(N) up to a constant factor and in the asymptotic sense as N grows toward infinity. This ability comes from the fact that the definition uses "≤" to compare T(N) to a f(N) times a constant, c, for the asymptotic cases when $N \geq N_0$.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Big-Omega Notation

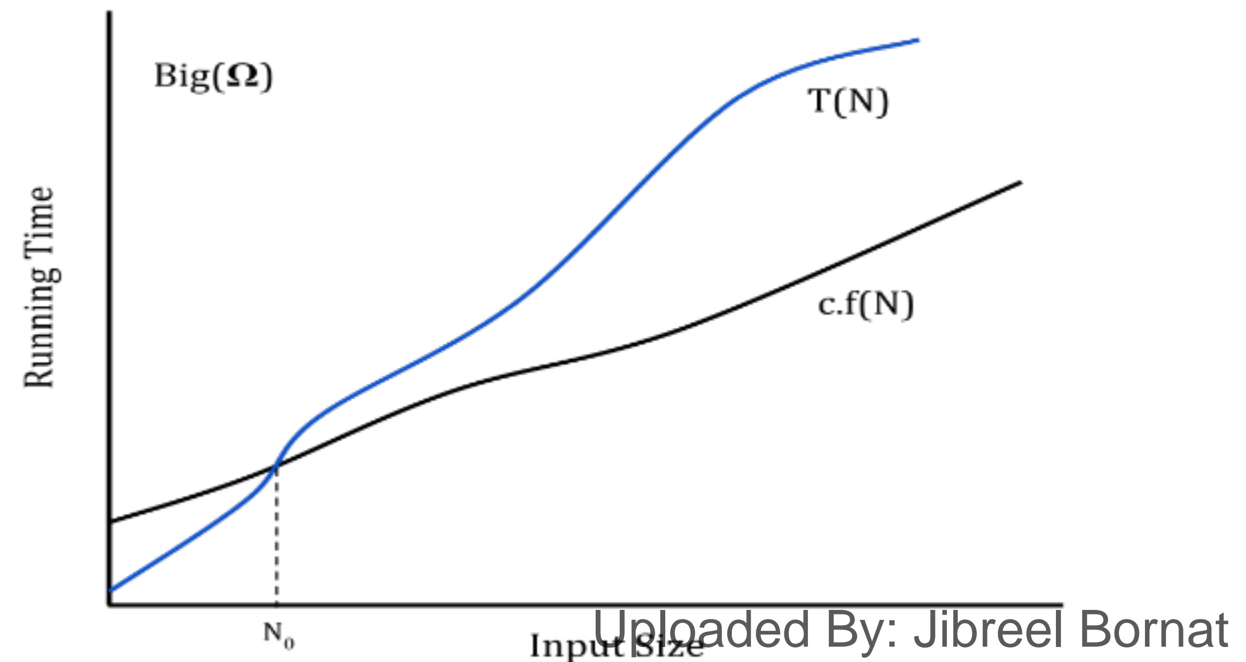
- **DEF#2** T(N) = **Ω(g(N))**
  The second definition, $T(N) = \Omega(g(N))$ (pronounced "omega"), says that the growth rate of $T(N)$ is greater than or equal to (≥) that of $g(N)$.
- Asymptotic lower bound.
- Think of it as the inverse of O(n).



STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Big-Omega Notation (2)

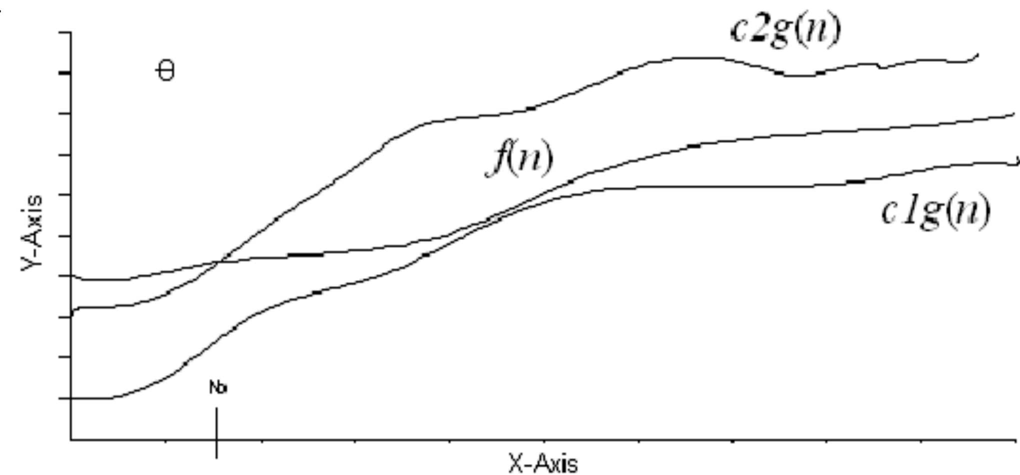
- $n = \Omega(1)$
- $n^2 = \Omega(n)$
- $n^2 = \Omega(n \log(n))$
- $3n + 2 = O(n)$



STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Big-Theta Notation

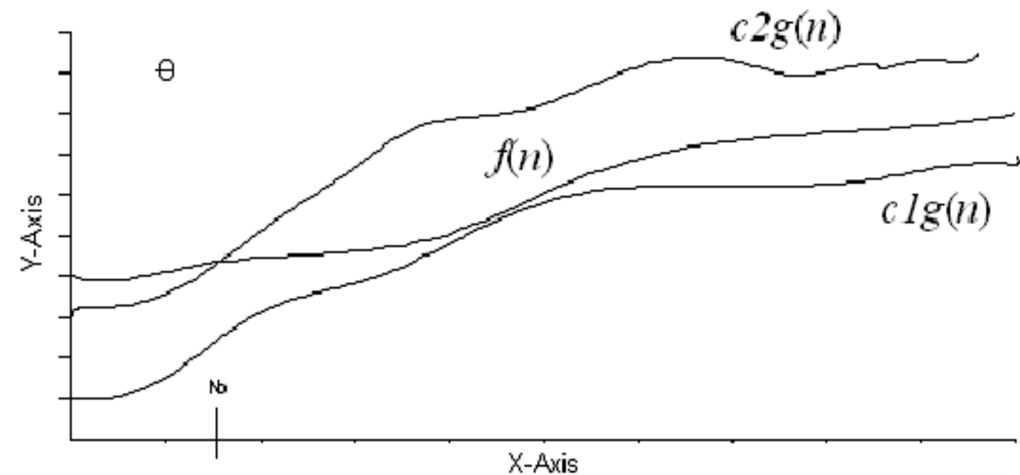
- **DEF#3 T(N) = Θ(h(N))**
  The third definition, $T(N) = \Theta(h(N))$ (pronounced "theta"), says that the growth rate of $T(N)$ equals (=) the growth rate of $h(N)$.
- Asymptotic tight bound (the upper and lower bounds).
- Combines Big-Oh and Big-Omega



STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Big-Theta Notation (2)

- $2n = \Theta(n)$
- $n^2 + 2n + 1 = \Theta(n^2)$
- $3n + 3$ is $O(n)$ and $\Theta(n)$
- $3n + 3$ is $O(n^2)$ BUT IS NOT $\Theta(n^2)$



STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Asymptotic Analysis - Examples

- As an example, N^3 grows faster than N^2, so we can say that $N^2 = O(N^3)$ or $N^3 = \boldsymbol{\Omega}(N^2)$. $f(N) = N^2$ and $g(N) = 2N^2$ grow at the same rate, so both $f(N) = O(g(N))$ and $f(N) = \boldsymbol{\Omega}(g(N))$ are true. When two functions grow at the same rate, then the decision of whether or not to signify this with $\boldsymbol{\Theta}()$ can depend on the particular context. Intuitively, if $g(N) = 2N^2$, then $g(N) = O(N^4)$, $g(N) = O(N^3)$, and $g(N) = O(N^2)$ are all technically correct, but the last option is the best answer. Writing $g(N) = \boldsymbol{\Theta}(N^2)$ says not only that $g(N) = O(N^2)$, but also that the result is as good (tight) as possible.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Asymptotic Analysis - Examples

- **Example**: The function $f(n) = 8n - 2$ is $O(n)$.
- **Justification**: By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n_0 >= 1$ such that $8n - 2 <= cn$ for every integer $n >= n_0$. It is easy to see that a possible choice is $c = 8$ and $n_0 = 1$.

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Asymptotic Analysis - Examples

- **Example**: The function $f(n) = 8n - 2$ is $O(n)$.
- More examples: $2n + 10$ is $O(n)$, $7n - 2$ is $O(n)$, $3n^3 + 20n^2 + 5$ is $O(n^3)$, $3\log n + 5$ is $O(\log n)$
- Example: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.
- Justification: Note that $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5 + 3 + 2 + 4 + 1)n^4 = cn^4$, for c=15,when $n \geq n_0$ =1.
- In fact, we can characterize the growth rate of any polynomial function.

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Asymptotic Analysis

- ***Proposition 4.9***: If $f(n)$ is a polynomial of degree d, that is, $f(n) = a_0 + a_1 n + \cdots + a_d n^d$, and $a^d > 0$, then $f(n)$ is $O(n^d)$.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Examples

- Example 4.10: $5n^2 + 3n \log n + 2n + 5$ is $O(n^2)$.
  Justification: $5n^2 + 3n \log n + 2n + 5 \le (5+3+2+5)n^2 = cn^2$, for $c=15$, when $n \ge n0 = 2$ (note that $n.\log.n$ is zero for $n=1$).
- Example 4.11: $20n^3 + 10n \log n + 5$ is $O(n^3)$.
  Justification: $20n^3 + 10n \log n + 5 \le 35n^3$, for $n \ge 1$.
- Example 4.12: $3 \log n + 2$ is $O(\log n)$.
  Justification: $3 \log n + 2 \le 5 \log n$, for $n \ge 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \ge n_0 = 2$ in this case.
- Example 4.13: $2^n + 2$ is $O(2^n)$.
  Justification: $2^n + 2 = 2n^2 \cdot 2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case.

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Rules

- Loops: the running time of a loop is at most the running time inside the loop times the number of iterations
- Nested loops: the number of times the inner loop is executed times the number of times the outer loop is executed
- Consecutive loops: add the number of times they run which results in the max
- if/else: max of the alternative path (which is the worst case)
  - if *condition*
    - Sequence 1
  - else
    - Sequence 2

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

LOOPS AND BIG-O NOTATION

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation

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
```

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation

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
}
```

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Loops and Big-O Notation

$O(n^c)$: 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
    }
}
```

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Loops and Big-O Notation

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

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation

O(Log n): Time Complexity of a loop is considered as O(Log n) 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 has O(Log n) time complexity.

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Loops and Big-O Notation

O(Log Log n): Time Complexity of a loop is considered as O(Log Log n) 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
}
```

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Examples

Two loops in a row:

```
for(i = 0; i < N; i++){ … sequence of statements …}
for(j = 0; j < M; j++){ … sequence of statements …}
```

The first loop is O(N) and the second loops is O(M). Since we don't know which one is bigger, we can write it as O(N + M), which is also written as O(max(N,M)). Assume that case in which N is greater than N, then the second loop goes to N instead of M it becomes O(N). O(N+M) becomes O(2N) and when you drop the constant it is O(N). O(max(N,M)) becomes O(max(N,N)) which is O(N).

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Examples

What is the time complexity of the following code fragment:

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

for (k = 0; k < N; k++) { sequence of statements }
```

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Examples

What is the time complexity of the following code fragment:

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

for (k = 0; k < N; k++) { sequence of statements }
```

The first loop is $O(N^2)$ and the second loop is $O(N)$. This is $O(max(N^2, N))$ which is $O(N^2)$.

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

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

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

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

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

- The number of iterations of the inner loop depends on the value of the index of the outer loop.
- The inner loop executes N times, then N-1, then N-2, etc, so the total number of times the innermost "sequence of statements" executes is $O(N^2)$.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
int x=0; //constant

for(int i=5*n; i>=1; i--) //runs n times, disregard the
constant
     x=x+3*i;
```

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

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

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

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

The outer will execute N number of times

The inner will execute N, then (N-1), then (N-2), …, 1.

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

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
int b=0;
for(int i=n; i>0; i--)
     for(int j=0; j<i; j++)
          b=b+5;
```

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
int y=1, j=0;

for(j=1; j<=2*n; j=j+2)
     y=y+i;

int s=0;
for(i=1; i<=j; i++)
     s++;
```

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
int b=0;
for(int i=0; i<n; i++)
    for(int j=0; j<i*n; j++)
        b=b+5;
```

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
int b=0;
for(int i=0; i<n; i++)
     for(int j=0; j<i*n; j++)
          b=b+5;
```

The inner loops will run $0 + n + 2n + 3n + 4n + \ldots + n(n-1) = n(0 + 1 + 2 + 3 + 4 + \ldots + n-1) = O(N^3)$

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
int z=0;
for(int i=1; i<=n; i=i*3){
    z = z + 5;
    z++;
    x = 2 * x;
}
```

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
int a = 0;
int k = n*n;

while(k > 1) //runs n^2 {
      for (int j=0; j<n*n; j++) //runs n^2
      { a++; }

      k = k/2;
}
```

Loops and Big-O Notation - Example

```
int a = 0;
int k = n*n;

while(k > 1) //runs n^2 {
     for (int j=0; j<n*n; j++) //runs n^2
     { a++; }

     k = k/2;
}
```

$T(N) = O(n^2 \log n)$.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
for (int i = n; i > 0; i = i / 2){
     for (int j = n; j > 0; j = j / 2){
          for (int k = n; k > 0; k = k / 2){
               count++;
          }
     }
}
```

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
for (int i = n; i > 0; i = i / 2){
	for (int j = n; j > 0; j = j / 2){
		for (int k = n; k > 0; k = k / 2){
			count++;
		}
	}
}
```

$T(N) = O((\log n)^3)$

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Loops and Big-O Notation - Example

```
bool isPrime(int n) {
      if(n == 2)
            return true;
      if(n < 2)
            return false;
      for (int i = 2; i <= sqrt(n); i ++)
            if (n%i == 0) return false;
                  return true;
}
```

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

RECURSION ANALYSIS

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Recursion Analysis

- Analysing the running time *T* of a recursive algorithm is more challenging than a non-recursive one.
- The most common strategy is to write the run time as a function of N: T(N) which indicates the time needed to process N items.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Recursion Analysis

- By tracing carefully through the recursion, we can write a recurrence relation for the algorithm. For example

`T(N) = T(N-1) + 1`

- Then we can repeat the recurrence

`T(N) = (T(N-2) + 1) + 1 = T(N-2) + 2`

`T(N) = (T(N-3) + 1) + 2 = T(N-3) + 3`

- In which we can observe the pattern: `T(N) = T(N-k) + k`
- By tracing the pattern all the way to the base case `T(1)`, we can determine the running time of the algorithm `T(N) = O(N)`.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Recursion Analysis - Example

```
int factorial (int n) {
      if (n == 0)
            return 1;
      else
            return n * factorial (n – 1);
}
```

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Recursion Analysis - Example

- $T(N) = \begin{cases} d, & N = 0 \\ T(N-1) + c, & N > 0 \end{cases}$

- T(N) = T(N-1) + c
- T(N-1) = T(N-2) + c
- T(N) = T(N-2) + 2c
- T(N-2) = T(N-3) + c
- T(N) = T(N-3) + 3c

.   .
.   .
.   .

- The pattern is T(N) = T(N-k) + k.c
- To solve this pattern (generalization), we have to find a value for k
- → notice that T(d) = T(1) = 0
- So N-k = 0 → N =k
- So let k = N→
- T(N) = T(0) + c = d + c.N
- T(N) = O(N)

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Recursion Analysis - Example

```
int power(int x, int n) {
    if (n == 0)
        return 1;
    if (n == 1)
        return x;
    if(n % 2 == 0)
        return power(x * x, n / 2);
    else
        return power(x * x, n / 2) * x;
}
```

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Recursion Analysis - Example

- $T(N) = \begin{cases} d, & N \leq 1 \\ T\left(\frac{N}{2}\right) + c, & N > 1 \end{cases}$

- T(N) = T(N/2) + c
- T(N/2) = T(N/4) + c
- T(N) = $T(N/2^2)$ + 2c
- T(N/4) = T(N/8) + c
- T(N) = $T(N/2^3)$ + 3c
- T(N/8) = T(N/16) + c
- T(N) = $T(N/2^4)$ + 4c

. .

. .

The pattern is T(N) = $T(N/2^k)$ + k.c

- We want to get rid of the $T(N/2^k)$. We can solve it when we reach T(1) →
- T(d) = T(1) = 1 = $\frac{N}{2^k}$
- → 2^k = N → k = log N
- T(N) = T(1) + c log N
- T(N) = d + c log N
- T(N) = O(log N)

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Recursion Analysis - Example

What is the worst-case running time for the following code fragment? Show the recurrence relation.

```
for(j = 1; j <= n; j *= 2){
  // some O(1) operations
}
```

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Recursion Analysis - Example

- $T(N) = \begin{cases} d, & N \leq 1 \\ T\left(\frac{N}{2}\right) + c, & N > 1 \end{cases}$

- T(N) = T(N/2) + c
- T(N/2) = T(N/4) + c
- T(N) = $T(N/2^2)$ + 2c
- T(N/4) = T(N/8) + c
- T(N) = $T(N/2^3)$ + 3c
- T(N/8) = T(N/16) + c
- T(N) = $T(N/2^4)$ + 4c

. .

. .

The pattern is T(N) = $T(N/2^k)$ + k.c

- We want tol get rid of the $T(N/2^k)$. We can solve it when we reach T(1) →
- T(d) = T(1) = 1 = $\frac{N}{2^k}$
- → 2^k = N → k = log N
- T(N) = T(1) + c log N
- T(N) = d + c log N
- T(N) = O(log N)

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

MERGE SORT

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

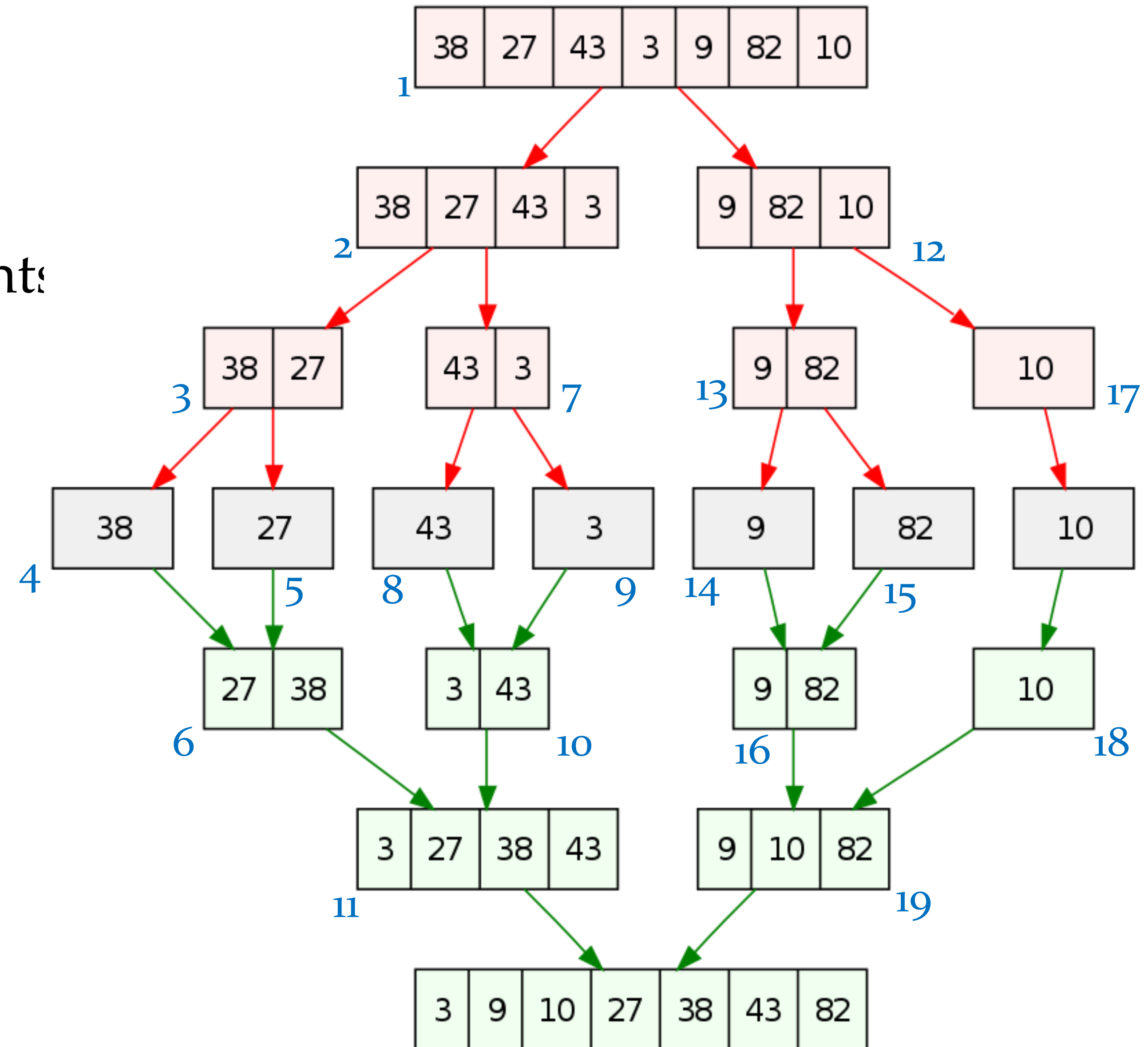
Merge Sort

- Divide and conquer technique
  - Divide an array into two halves
  - Recursively sort each half
  - Merge two halves to make a sorted one
- Algorithm: to sort A[1 .. n] call Merge-Sort(A, a, n)
- `Merge-Sort(A, p, r):`
  - `//Input: A[p .. r]`
  - `//Output: A[p .. r] with sorted numbers`
  - `if p < r then`
    - `q = ⌊ (p + r) / 2 ⌋`
    - `Merge-Sort(A, p, q)`
    - `Merge-Sort(A, q + 1, r)`
    - `Merge(A, p, q, r)`

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Merge Sort

1. Divide the array into two parts
2. Divide the array again until elements cannot be further broken
3. Sort the elements from smallest to largest
4. Merge the divided sorted arrays together
5. The array is now sorted



STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Merge Sort

- To solve Merge sort, we have to write a recurrence relation for the running time.
- Assume N is a power of 2 so that we always split in half
- For N=1, the time for merge sort is constant.
- Otherwise, time to merge sort N number of elements equals to the time to do 2 recursive merge sorts of size N/2 plus the time to merge which is linear.

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Recursion Analysis – Merge Sort

- $T(N) = \begin{cases} d, & N = 0, N = 1 \\ 2T\left(\frac{N}{2}\right) + N, & N > 1 \end{cases}$

- T(N) = 2T(N/2) + N
- T(N/2) = 2T(N/4) + N/2
- T(N) = 2(2T(N/4) + N/2) + N
  = 4T(N/4) + 2N
- T(N/4) = 2T(N/8) + N/4
- T(N) = 4(2T(N/8) + N/4)+ 2N
  = 8T(N/8) + 3N
- T(N/16) = 2T(N/16) + N/16

- T(N) = 16T(N/16) + 4N

- The pattern is $T(N) = 2^k.T(N/2^k) + k.N$

- We want to reduce this in terms of T(1) which is what we know, in this case $N/2^k = 1 \rightarrow 2^k = N \rightarrow k = \log_2 N$
- So now we can substitute the terms such that $\rightarrow 2^{\log_2 N}\ T(1) + \log_2 N\ .\ N$
- We know that $a^{\log_b c} = c^{\log_b a} \rightarrow$
- = N + log N . N → O(N log N)

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

BINARY SEARCH

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Binary Search

- Search for an element in a **sorted array.**

```
int BinarySearch(int A[], int lower, int upper, int key){
    if(lower <= upper){
        int mid = (lower + upper) / 2;

        if(A[mid] == key)
           return mid;
        else
           if(A[mid] < key)
               return BinarySearch(A, mid+i, upper, key);
           else
               return BinarySearch(A, lower, mid-1, key);
    }
    else
        return -1;
}
```

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Binary Search

- Compare x with the middle element
- If x matches the middle, return middle index
- Else if x is greater than the middle element, then x can only lie in the right half sub-array after the mid element. So we recur right half.
- Else, x is smaller so recur the left half
- E.g., Search(23)

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Binary Search

- Compare x with the middle element
- If x matches the middle, return middle index
- Else if x is greater than the middle element, then x can only lie in the right half sub-array after the mid element. So we recur right half.
- Else, x is smaller so recur the left half
- E.g., Search(23)

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

L=0 M=4 U=9

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Binary Search

- Compare x with the middle element
- If x matches the middle, return middle index
- Else if x is greater than the middle element, then x can only lie in the right half sub-array after the mid element. So we recur right half.
- Else, x is smaller so recur the left half
- E.g., Search(23)

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

L=0 M=4 U=9

23 > 16

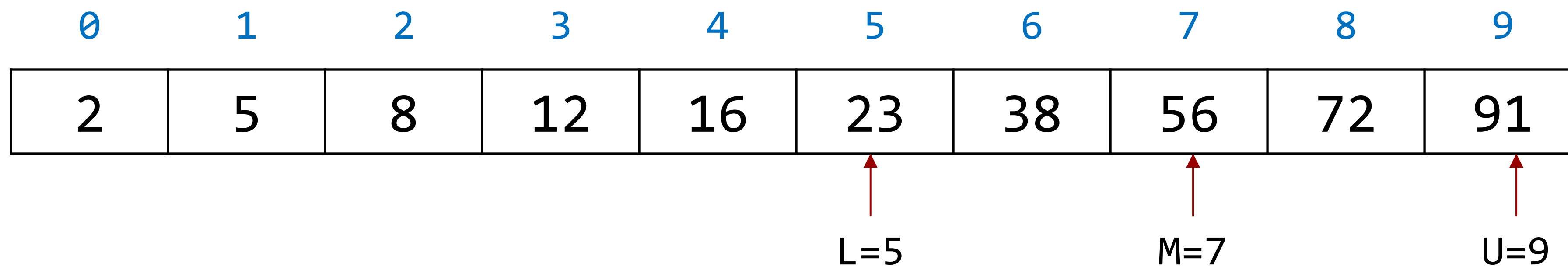
Look in the 2nd half

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Binary Search

- Compare x with the middle element
- If x matches the middle, return middle index
- Else if x is greater than the middle element, then x can only lie in the right half sub-array after the mid element. So we recur right half.
- Else, x is smaller so recur the left half
- E.g., Search(23)



STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Binary Search

- Compare x with the middle element
- If x matches the middle, return middle index
- Else if x is greater than the middle element, then x can only lie in the right half sub-array after the mid element. So we recur right half.
- Else, x is smaller so recur the left half
- E.g., Search(23)

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

L=5 M=7 U=9

23 < 56

Look in the 1st half

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Binary Search

- Compare x with the middle element
- If x matches the middle, return middle index
- Else if x is greater than the middle element, then x can only lie in the right half sub-array after the mid element. So we recur right half.
- Else, x is smaller so recur the left half
- E.g., Search(23)

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

L=5 M=5 U=6

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Binary Search – Recurrence Relation

- $T(N) = \begin{cases} d, & N = 1 \\ T\left(\frac{N}{2}\right) + c, & N > 1 \end{cases}$

- T(N) = T(N/2) + c
- T(N/2) = T(N/4) + c
- T(N) = $T(N/2^2) + 2c$
- T(N/4) = T(N/8) + c
- T(N) = $T(N/2^3) + 3c$
- T(N/8) = T(N/16) + c
- T(N) = $T(N/2^4) + 4c$

. .

. .

The pattern is $T(N) = T(N/2^k) + k.c$

- We want tol get rid of the $T(N/2^k)$. We can solve it when we reach T(1) →
- $T(d) = T(1) = 1 = \frac{N}{2^k}$
- → $2^k = N$ → k = log N
- T(N) = T(1) + c log N
- T(N) = d + c log N
- T(N) = O(log N)

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

INSERTION SORT

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

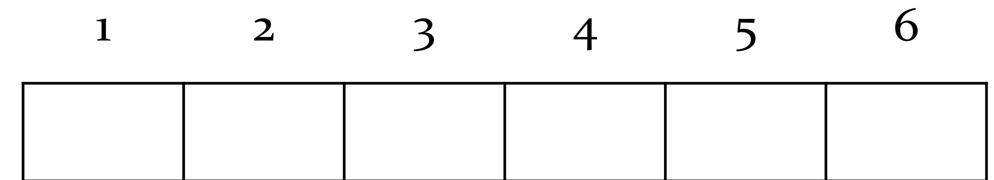
Insertion Sort

- Insertion Sort is a simple sorting algorithm.
- Good for small lists.
- Good for partially sorted lists.

- The time analysis of Insertion Sort depends on the nature of the input data
  - Best case
  - Average case
  - Worst case

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Recursion Analysis - Example




```
void insertion (int[] arr, int size){
      int j, key;

      for (int i = 2; i < size; i++) {
            key= arr[i];
            j=i-1;

            while (j>0 && arr[j]>key){
                  arr[j+1] = arr[j];
                  j--;
            }
            arr[j+1]=key;
      }
}
```

STUDENTS-HUB.com

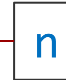
Uploaded By: Jibreel Bornat

Recursion Analysis - Example

```
void insertion (int[] arr, int size){
      int j, key;

      for (int i = 2; i < size; i++) {
            key= arr[i];
            j=i-1;

            while (j>0 && arr[j]>key){
                  arr[j+1] = arr[j];
                  j--;
            }
            arr[j+1]=key;
      }
}
```



STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Recursion Analysis - Example

```
void insertion (int[] arr, int size){
      int j, key;

      for (int i = 2; i < size; i++) {      <- n
            key= arr[i];                    <- (n-1)
            j=i-1;                          <- (n-1)

            while (j>0 && arr[j]>key){
                  arr[j+1] = arr[j];
                  j--;
            }
            arr[j+1]=key;
      }
}
```

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Recursion Analysis - Example

```
void insertion (int[] arr, int size){
	int j, key;

	for (int i = 2; i < size; i++) {      <- n
		key= arr[i];                      <- (n-1)
		j=i-1;                            <- (n-1)

		while (j>0 && arr[j]>key){
			arr[j+1] = arr[j];
			j--;
		}
		arr[j+1]=key;
	}
}
```

Number of times the while loop is executed for that value of i. When i=1, it will execute 1, when i=2 → 2, … so it becomes 2+3+4+5+…+n = $\sum_{i=2}^{n} t_j$

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Recursion Analysis - Example

```
void insertion (int[] arr, int size){
      int j, key;

      for (int i = 2; i < size; i++) {      <- n
            key= arr[i];                   <- (n-1)
            j=i-1;                         <- (n-1)

            while (j>0 && arr[j]>key){     <- Number of times the while loop is executed
                  arr[j+1] = arr[j];       <- sum_{i=2}^{n} (t_j - 1)
                  j--;                     <- sum_{i=2}^{n} (t_j - 1)
            }
            arr[j+1]=key;
      }
}
```

- for loop: n
- `key= arr[i];`: (n-1)
- `j=i-1;`: (n-1)
- `arr[j+1] = arr[j];`: $\sum_{i=2}^{n}(t_j-1)$
- `j--;`: $\sum_{i=2}^{n}(t_j-1)$

Number of times the while loop is executed for that value of i. When i=1, it will execute 1, when i=2 → 2, … so it becomes

2+3+4+5+…+n = $\sum_{i=2}^{n} t_j$

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Recursion Analysis - Example

```
void insertion (int[] arr, int size){
      int j, key;

      for (int i = 2; i < size; i++) {        ← n
            key= arr[i];                      ← (n-1)
            j=i-1;                            ← (n-1)

            while (j>0 && arr[j]>key){        ← (see note)
                  arr[j+1] = arr[j];          ← Σ_{i=2}^{n} (t_i−1)
                  j--;                        ← Σ_{i=2}^{n} (t_i−1)
            }
            arr[j+1]=key;                     ← (n-1)
      }
}
```

Costs: `for` line: n; `key= arr[i];`: $(n-1)$; `j=i-1;`: $(n-1)$; `arr[j+1] = arr[j];`: $\sum_{i=2}^{n}(t_i-1)$; `j--;`: $\sum_{i=2}^{n}(t_i-1)$; `arr[j+1]=key;`: $(n-1)$

Number of times the while loop is executed for that value of i. When i=1, it will execute 1, when i=2 → 2, … so it becomes 2+3+4+5+…+n = $\sum_{i=2}^{n} t_i$

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Recursion Analysis - Example

```
     void insertion (int[] arr, int size){
             int j, key;

C1           for (int i = 2; i < size; i++) {        <- n
C2                   key= arr[i];                    <- (n-1)
C3                   j=i-1;                          <- (n-1)

C4                   while (j>0 && arr[j]>key){
C5                           arr[j+1] = arr[j];      <- Σ_{i=2}^{n} (t_i-1)
C6                           j--;                    <- Σ_{i=2}^{n} (t_i-1)
                     }
C7                   arr[j+1]=key;                   <- (n-1)
             }
     }
```

C5: $\sum_{i=2}^{n}(t_i - 1)$

C6: $\sum_{i=2}^{n}(t_i - 1)$

C4: Number of times the while loop is executed for that value of i. When i=1, it will execute 1, when i=2 → 2, … so it becomes 2+3+4+5+…+n = $\sum_{i=2}^{n} t_i$

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Insertion Sort

- Assumption: constant amount of time is required for each line of the pseudo code → i^{th} line needs c_i time.
- $T(N) = C_1N + C_2(N\text{-}1) + C_3(N\text{-}1) + C_4\sum_{i=2}^{n} t_i + C_5\sum_{i=2}^{n}(t_i - 1) + C_6\sum_{i=2}^{n}(t_i - 1) + C_7(N\text{-}1)$
- The total time equals the cost of each line times the number of times executed for each line.
- Best Case: the array is already sorted. For each i, there will be 1 comparison in the while loop. So t_i equals 1 for each i
- C5 and C6 times zero
- $C_4\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} 1 = 1+1+1+\ldots+1$ [(n-1) times]
- So $T(N) = C_1N + C_2(N\text{-}1) + C_3(N\text{-}1) + C_4(N\text{-}1) + C_5(0) + C_6(0) + C_7(N\text{-}1)$

  $= N(C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7) - (C_2 + C_3 + C_4 + C_7)$

  $= O(N)$

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Insertion Sort

- Worst Case (same for Average Case): the array is reverse sorted. For each i, the while loop will execute i number of times, so $t_i = i$.
- $T(N) = C_1N + C_2(N-1) + C_3(N-1) + C_4\sum_{i=2}^{n} t_i + C_5\sum_{i=2}^{n}(t_i - 1) + C_6\sum_{i=2}^{n}(t_i - 1) + C_7(N-1)$

$\rightarrow 1 + 2 + 3 + \cdots + N = \frac{N(N+1)}{2}$

but $\sum_{i=2}^{n} i = \frac{N(N+1)}{2} - 1$ because we started at 2

- Similarly $\sum_{i=2}^{n}(i - 1) = \frac{N(N-1)}{2}$
- So $T(N) = C_1N + C_2(N-1) + C_3(N-1) + C_4(\frac{N(N+1)}{2} - 1) + C_5(\frac{N(N-1)}{2}) + C_6(\frac{N(N-1)}{2}) + C_7(N-1)$

$= N^2(\frac{C_4}{2} + \frac{C_5}{2} + \frac{C_6}{2}) + N(C_1 + C_2 + C_3 + \frac{C_4}{2} - \frac{C_5}{2} - \frac{C_6}{2} + C_7) - (C_2 + C_3 + C_4 + C_7)$

$= O(N^2)$

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat