

DATA STRUCTURES

Space and Time Complexity

Program Performance

- To compare different solution for the same functionality
- Performance is measured by the amount of space and time needed to run the program.
- Two ways to determine:
 - Analytically: Analysing the program without executing it
 - Experimentally: measure performance by executing program for multiple set of inputs

Criteria for Measurement

- Space
 - Amount of memory program occupies
 - Usually measured in Bytes, KB or MB
 - Instruction space, data space and stack space are parts of total space
- Time
 - Execution time of the program
 - Compilation time not included as it's a one time process
 - Usually measured in number of executions of various statements in the program

Analytical Measurement

- Measure the performance of program without execution
- Performance is measured in terms of the size of the input
- Upper and lower bounds of the program are determined
- Results are not actual
- Commonly used for comparison between algorithms

Experimental Measurements

- Run multiple instances of algorithm for various input sets in a controlled environment
- Underlying software and hardware impacts the results
- Results are actual for the given environment and constraints

Space Complexity

- Space complexity is defined as the amount of memory a program needs to complete execution.
- Why is this of concern?
 - We could be running on a multi-user system where programs are allocated a specific amount of space.
 - We may not have sufficient memory on our computer.
 - There may be multiple solutions, each having different space requirements.
 - The space complexity may define an upper bound on the data that the program might have to handle.

Components of Program Space

- Program Space = Instruction Space + Data Space + Stack Space
- Instruction Space
 - Total memory occupied by instructions
 - Dependent on compiler, compilation options and target system

```
printHello( int n)
{    printf("Hello World!");    }
```

- Constant space is required (not dependent on input size)

Components of Program Space

- Data Space:
 - Compile time allocated memory + dynamically allocated memory for data
 - Dependent on computer architecture and compiler

`int arr[10][20];`  `200*sizeof(int)` compile time

`Eptr nNode = (Eptr)malloc(100*sizeof(Estruct));`

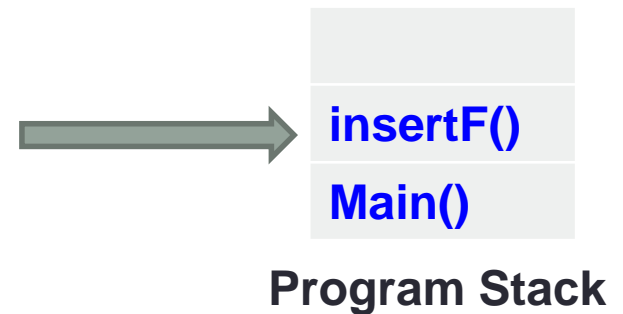


`100*sizeof(Estruct)` run time

Components of Program Space


- Stack Space
 - Every time a function is called, following data (activation record) of calling function is stored on stack
 - Return address
 - Values of local variables and formal parameters
 - Binding of reference parameters

```
EpPtr insertF (EpPtr start, int nData) {  
    return newNode(nData, start);  
}
```




Components of Program Space

- Identify instance characteristics (factors that determine the size of the problem instance e.g. the number of inputs, outputs, magnitude of numbers involved etc.)
- Total space needed by a program:
 - **Fixed part** – independent of instance characteristics [instruction space, space for simple variables, constants etc.]
 - **Variable part** – dynamically allocated space, recursion stack space[depends on space needed by local variables and formal parameters, maximum depth of recursion]
- Therefore, total space = $C + S_p(\text{instance characteristics})$



Fixed Part



Variable Part

Space Complexity (Examples)

```
int abc(int a, int b, int c) {  
    return a + b * c;  
}
```

Space required by variable part: $S_{abc}(ic) = 0$

```
int sum(int a[], int n) {  
    // return the sum of numbers a[0:n-1]  
    int theSum = 0;  
    for (int i=0; i<n; i++)  
        theSum+=a[i];  
    return theSum  
}
```

Space required by variable part: $S_{sum}(n) = 0$

Space Complexity(Examples)

```
int rSum(int a[], int n) {  
    // return sum of numbers a[0:n-1] using recursion  
    if (n>0)  
        return (rSum(a, n-1) + a[n-1]);  
    return 0;  
}
```

$$S_{\text{rSum}}(n) = 2n + 2n + 2n = 6n$$

Reference of a

Value of n

Return address (assuming
address of 2 Bytes)

Space Complexity(Examples)

```
int factorial( int n) {  
    // return n!  
    if (n <= 1)  
        return 1;  
    else  
        return (n*factorial(n-1));  
}
```

$$S_{\text{factorial}}(n) = 2 * \max\{n-1, 1\} + 2 * \max\{n-1, 1\} = 4 * \max\{n-1, 1\}$$

Value of n

Return address (assuming
address of 2 Bytes)

Time Complexity

- Time complexity is the **total computer time** a program needs to execute.
- Why is this a concern?
 - Some computers require upper limits for program execution times.
 - Some programs require a real-time response.
 - If there are many solutions to a problem, typically we'd like to choose the quickest.

Time Complexity

- How do we measure?
 1. Count a particular operation ([operation counts](#))
 2. Count the number of steps ([step counts](#))

Running Example: Insertion Sort

```
for (int i = 1; i < n; i++)           // n is the number of
{                                     // elements in array
    //insert a[i] into a[0:i-1]
    int t = a[i];
    int j;
    for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
    a[j + 1] = t;
}
```


Operation Count Method

- Pick an instance characteristic ... n ,
 n = the number of elements in case of insertion sort.
- Determine count as a function of this instance characteristic.
- Focus only on key operations and ignore all others
- Worst case count = maximum count
- Best case count = minimum count
- Average count

Operation Count : Comparison

```
for (int i = 1; i < n; i++)  
{  
    // insert a[i] into a[0:i-1]  
    int t = a[i];  
    int j;  
    for (j = i - 1; j >= 0 && t < a[j]; j--)  
        a[j + 1] = a[j];  
    a[j + 1] = t;  
}
```

Operation Count : Comparison

```
for (int i = 1; i < n; i++)  
    for (j = i - 1; j >= 0 && t < a[j]; j--)  
        a[j + 1] = a[j];
```

- How many comparisons are made?
- The number of compares depends on **a[]** and **t** as well as on **n**.

Worst Case Operation Count

- Only inner Loop

```
for (j = i - 1; j >= 0 && t < a[j]; j--)  
    a[j + 1] = a[j];
```

$a = [1, 2, 3, 4]$ and $t = 0 \Rightarrow 4$ compares

$a = [1, 2, 3, 4, \dots, i]$ and $t = 0 \Rightarrow i$ compares

Worst Case Operation Count

- Both the loops

```
for (int i = 1; i < n; i++)  
    for (j = i - 1; j >= 0 && t < a[j]; j--)  
        a[j + 1] = a[j];
```

$$\begin{aligned}\text{total compares} &= 1+2+3+\dots+(n-1) \\ &= (n-1)n/2\end{aligned}$$

Step Count Method

- The **operation-count method** omits accounting for the time spent on all but the chosen operation
- The **step-count method** count for all the time spent in all parts of the program
- A **program step** is loosely defined to be a syntactically or semantically meaningful segment of a program for which the execution time is independent of the instance characteristics.
 - 100 adds, 100 subtracts, 1000 multiples can be counted as one step.
 - However, ***n*** adds cannot be counted as one step.

Step Count : insertion sort

	steps/execution (s/e)
for (int i = 1; i < n; i++)	1
{	0
// insert a[i] into a[0:i-1]	0
int t = a[i];	1
int j;	1
for (j = i - 1; j >= 0 && t < a[j]; j--)	1
a[j + 1] = a[j];	1
a[j + 1] = t;	1
}	0

Step Count : Insertion Sort

	s/e	frequency
for (int i = 1; i < n; i++)	1	n-1
{	0	0
// insert a[i] into a[0:i-1]	0	0
int t = a[i];	1	n-1
int j;	1	n-1
for (j = i - 1; j >= 0 && t < a[j]; j--)	1	(n-1)n/2
a[j + 1] = a[j];	1	(n-1)n/2
a[j + 1] = t;	1	n-1
}	0	0

Step Count : Insertion Sort

Total step counts

$$= (n-1) + 0 + 0 + (n-1) + (n-1) + (n-1)n/2 + (n-1)n/2 + (n-1) + 0$$

$$= n^2 + 3n - 4$$

Operation and Step Count

- Two important reasons to determine operation and step counts
 1. To **compare the time complexities** of two programs that compute the same function
 2. To **predict the growth in run time** as the instance characteristic changes

Examples(1)

```
Main() {  
    int n;  
    scanf("%d",&n);  
    printf("\n Value of n is: %d",n);  
}
```

Examples(1)

Main() {	s/e	Frequency
int n;	1	1
scanf("%d",&n);	1	1
printf("\n Value of n is: %d",n);	1	1
}		

Time Complexity $= 1*1 + 1*1 + 1*1$
 $= 3$
 $= \mathbf{O(1)}$
 \Rightarrow constant(independent of input size)

Examples(2)

```
Main() { //omitting print and scan statements
    int i, n, x=0;
    for (i=0; i<n; i++) {
        x=x+1;
    }
}
```

Examples(2)

Main() { //omitting trivial statements	s/e	Frequency
int i, n, x=0;	1	1
for (i=0; i<n; i++) {	1	n+1
x=x+1;	1	n
}		
}		

Time Complexity $= 1*1 + 1*(n+1) + 1*n$
 $= 2*n + 2$
 $= \mathbf{O(n)}$
 \Rightarrow Linear(grows linearly with input size)

Examples(3)

```
Main() {  
    int i, j, x=0;  
    for (i=0; i<n; i++) {  
        x=x+1; }  
    for (i=0; i<n; i++) {  
        for (j=0; j<n; j++) {  
            x=x+1; } }  
}
```


Examples(4)

```
Main() {  
    int n;  
    while ( n > 1) {  
        n = n/2;  
    }  
}
```

Examples(4)

```
Main() {  
    int n;  
    while (n > 1) {  
        n = n/2;  
    }  
}
```

	if n=2	if n=4	if n=8
After 1 st iteration	n=1	n=2	n=4
After 2 nd iteration		n=1	n=2
After 3 rd iteration			n=1
Frequency	1	2	3

Examples(4)

```
Main() {  
    int n;  
    while (n > 1) {  
        n = n/2;  
    }  
}
```

	if n=2	if n=4	if n=8
After 1 st iteration	n=1	n=2	n=4
After 2 nd iteration		n=1	n=2
After 3 rd iteration			n=1
Frequency	1	2	3

Time Complexity = **$O(\log_2 n)$** (grows logarithmically to input size)

Exercise

```
Main() {  
    int i, j, k, n;  
    for (i=0; i<n; i++) {  
        for (j=0; j<i; j++) {  
            if (j%n == 0) {  
                for (k=0; k<n; k++) {  
                    x=x+1;  
                }  
            }  
        }  
    }  
}
```

Exercise

```
Main() {  
    int i, j, k, n, x=0;  
    for (i=1; i<n; i++) {  
        for (j=1; j<i; j++) {  
            if (j%n == 0) {  
                for (k=1; k<n; k++) {  
                    x=x+1;  
                }  
            }  
        }  
    }  
}
```

	s/e	frequency
	1	1
1	n	
	1	$n*(n+1)/2$
1	$n*(n+1)/2$	
	1	n
	1	n-1

Time Complexity = **$O(n^2)$**

Asymptotic Notation

- Describes the behavior of the time or space complexity for large instance characteristic
- **Big O (O)** notation provides an **upper bound** for the function f
- **Big Omega (Ω)** notation provides a **lower-bound**
- **Theta (Θ)** notation is used when an algorithm can be **bounded both from above and below** by the same function
- **Little O (o)** defines a **loose upper bound**.
- **Little Omega (ω)** defines a **loose lower bound**.

Upper Bounds

- Time complexity $T(n)$ is a function of the problem size n . The value of $T(n)$ is the running time of the algorithm in the **worst case**, i.e., the number of steps it requires **at most** with an arbitrary input.
- Example: the sorting algorithm Insertion Sort has a time complexity of $T(n) = n \cdot (n-1)/2$ comparison-exchange steps to sort a sequence of n data elements.
- Often, it is not necessary to know the exact value of $T(n)$, but only an **upper bound** as an estimate.
- e.g., an upper bound for time complexity $T(n)$ of insertion sort is the function $f(n) = n^2$, since $T(n) \leq f(n)$ for all n .

Big O (O) Notation

- In general, just the order of the asymptotic complexity is of interest, i.e., if it is a **linear**, **quadratic**, **exponential** function.
- **Definition:** $f(n) = O(g(n))$ (read as “ $f(n)$ is Big O of $g(n)$ ”) iff positive constants c and n_0 exist such that $f(n) \leq cg(n)$ for all n , $n \geq n_0$.
- That is, $O(g(n))$ comprises all functions f , for which there exists a constant c and a number n_0 , such that $f(n)$ is smaller or equal to $c \cdot g(n)$ for all n , $n \geq n_0$.

Big O Examples

- Example: the time complexity $T(n)$ of Insertion Sort lies in the complexity class $O(n^2)$.
- $O(n^2)$ is the complexity class of all functions that grow **at most quadratically**. Respectively, $O(n)$ is the set of all functions that grow at most **linearly**, $O(1)$ is the set of all functions that are bounded from above by a **constant**, $O(n^k)$ is the set of all functions that grow **polynomially**, etc.

Lower Bounds

- Once an algorithm for solving a specific problem is found, the question arises whether it is possible to design a faster algorithm or not.
- How can we know unless we have found such an algorithm? In most cases a lower bound for the problem can be given, i.e., **a certain number of steps that every algorithm has to execute *at least*** in order to solve the problem.
- e.g., In order to sort n numbers, every algorithm at least has to take a look at every number. So, it needs at least n steps. Thus, $f(n) = n$ is a lower bound for sorting

Omega (Ω) Notation

- Again, only the **order of the lower bound** is considered, namely if it is a linear, quadratic, exponential or some other function. This order is given by a function class using the **Omega (Ω)** notation.
- **Definition:** $f(n) = \Omega(g(n))$ (read as “ $f(n)$ is omega of $g(n)$ ”) iff positive constants c and n_0 exist such that $f(n) \geq cg(n)$ for all $n, n \geq n_0$.
- That is, $\Omega(g)$ comprises all functions f , for which there exists a constant c and a number n_0 , such that $f(n)$ is greater or equal to $c \cdot g(n)$ for all $n \geq n_0$.

Omega Examples

- Let $f(n) = 2n^2 + 7n - 10$ and $g(n) = n^2$. Since with $c = 1$ and for $n \geq n_0 = 2$ we have $2n^2 + 7n - 10 \geq c \cdot n^2$, thus $f(n) = \Omega(g)$.
- This example function $f(n)$ lies in $\Omega(n^2)$ as well as in $O(n^2)$, i.e., it grows **at least quadratically**, but also **at most quadratically**.
- In order to express the exact order of a function the class $\Theta(f)$ is introduced (Θ is the greek letter theta).

Theta (Θ) Notation

- Used when the function f can be **bounded both from above and below** by the same function g .
- **Definition:** $f(n) = \Theta(g(n))$ (read as “ $f(n)$ is theta of $g(n)$ ”) iff positive constants c_1 , c_2 and n_0 exist such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all n , $n \geq n_0$.
- That is, f lies between c_1 times the function g and c_2 times the function g , except possibly when n is smaller than n_0 .

Theta Examples

- As seen above the lower bound for the sorting problem lies in $\Omega(n)$. Its upper bound lies in $O(n^2)$, e.g., using Insertion Sort. Thus, time complexity for insertion sort is not $\Theta(n)$
- e.g., with [Heapsort](#), the upper bound is $O(n \cdot \log(n))$ and the lower bound can also be improved to $\Omega(n \cdot \log(n))$.
- Thus, [Heapsort](#) is an [optimal](#) sorting algorithm, since its upper bound matches the lower bound for the sorting problem and time complexity is $\Theta(n \cdot \log(n))$

Little O (o) Notation

- **Definition:** $f(n)=o(g(n))$ (read as “ $f(n)$ is little o of $g(n)$ ”) iff $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$.
- That is, f has a lower growth rate than g .
- Little oh is also called a loose upper bound.
- Hierarchy of growth rate functions:
 $1 < \log n < n < n \log n < n^2 < n^3 < 2^n < 3^n < n! < n^n$

Little Omega (ω) Notation

- **Definition:** $f(n)=\omega(g(n))$ (read as “ $f(n)$ is little ω of $g(n)$ ”) iff $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$.
- That is, g has a lower growth rate than f .
- Little omega is also called a loose lower bound.
- Hierarchy of growth rate functions:
 $1 < \log n < n < n \log n < n^2 < n^3 < 2^n < 3^n < n! < n^n$

Common Growth Rate Functions

- 1 (**constant**): growth is independent of the problem size n .
- $\log_2 n$ (**logarithmic**): growth increases slowly compared to the problem size (binary search)
- n (**linear**): directly proportional to the size of the problem.
- $n * \log_2 n$ (**$n \log n$**): typical of some divide and conquer approaches (merge sort)
- n^2 (**quadratic**): typical in nested loops
- n^3 (**cubic**): more nested loops
- 2^n (**exponential**): growth is extremely rapid and possibly impractical.
- $n!$ (**factorial**)

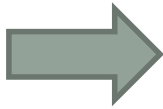
Practical Complexities

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Overly complex programs may not be practical given the computing power of the system.

Recurrence Relation

- A Recurrence Relation is an equation which is defined in terms of itself

- Eg. $A_n = A_{n-1} + 1$, $A_1 = 1$
 $A_n = 2 * A_{n-1}$, $A_1 = 1$
 $A_n = n * A_{n-1}$, $A_1 = 1$  **Boundary Condition**

- Recursive function can be expressed as recurrence relation to find total time required in the program
- Time required to execute recursive problem is dependent on time required by its sub-problem

Recurrence Relation: Sum of Array

```
int rSum(int a[], int size) {  
    if (size > 0)  
        return(rSum(a, size-1) + a[size-1]);  
    else  
        return 0;  
}
```

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1, & n > 0 \\ &= C_2, & n = 0 \end{aligned}$$

Recurrence Relation: Binary Search

```
int BS(int a[], int front, int rear, int key) {  
    if (front == rear)  
        if (a[front] == key)  
            return front;  
        else  
            return -1;  
    mid = (front + rear) / 2;  
    if (a[mid] > key)  
        return BS(a, front, mid, key);  
    else  
        return BS(a, mid+1, rear, key);  
}
```

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1, & n > 1 \\ &= C_2, & n = 1 \end{aligned}$$

Recurrence Relation: Factorial

```
int factorial( int n) {  
    // return n!  
    if (n <= 1)  
        return 1;  
    else  
        return (n*factorial(n-1));  
}
```

Recurrence Relation: Factorial

```
int fact( int n) {  
    // return n!  
    if (n <= 1)  
        return 1;  
    else  
        return (n*factorial(n-1));  
}
```

$$\begin{aligned} \text{fact}(n) &= \text{fact}(n-1) + C_1, & n > 1 \\ &= C_2, & n \leq 1 \end{aligned}$$

Solve Recurrence Relation

- Three methods to solve
 - Substitution
 - Recursive Tree Method
 - Master Theorem

Solve Recurrence Relation

- Three methods to solve
 - **Substitution**
 - Recursive Tree Method
 - Master Theorem

Substitution

- Substitute the given function again and again until a pattern is visible
- Use Boundary/Terminating condition to end substitution
- Find the sum of the terms

Substitution Example(1)

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1, & n > 0 \\ &= C_2, & n = 0 \end{aligned}$$

$$\text{rSum}(n) = \text{rSum}(n-1) + C_1$$

Substitution Example(1)

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1, & n > 0 \\ &= C_2, & n = 0 \end{aligned}$$

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1 \\ &= (\text{rSum}(n-2) + C_1) + C_1 \end{aligned}$$

Substitution Example(1)

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1, & n > 0 \\ &= C_2, & n = 0 \end{aligned}$$

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1 \\ &= (\text{rSum}(n-2) + C_1) + C_1 \\ &= \text{rSum}(n-2) + 2 * C_1 \end{aligned}$$

Substitution Example(1)

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1, & n > 0 \\ &= C_2, & n = 0 \end{aligned}$$

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1 \\ &= (\text{rSum}(n-2) + C_1) + C_1 \\ &= \text{rSum}(n-2) + 2 * C_1 \\ &= \text{rSum}(n-3) + 3 * C_1 \end{aligned}$$

Substitution Example(1)

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1, & n > 0 \\ &= C_2, & n = 0 \end{aligned}$$

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1 \\ &= (\text{rSum}(n-2) + C_1) + C_1 \\ &= \text{rSum}(n-2) + 2 * C_1 \\ &= \text{rSum}(n-3) + 3 * C_1 \\ &\quad \text{Substituting } n \text{ times} \\ &= \text{rSum}(0) + n * C_1 \end{aligned}$$

Substitution Example(1)

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1, & n > 0 \\ &= C_2, & n = 0 \end{aligned}$$

$$\begin{aligned} \text{rSum}(n) &= \text{rSum}(n-1) + C_1 \\ &= (\text{rSum}(n-2) + C_1) + C_1 \\ &= \text{rSum}(n-2) + 2 * C_1 \\ &= \text{rSum}(n-3) + 3 * C_1 \end{aligned}$$

Substituting n times

$$\begin{aligned} &= \text{rSum}(0) + n * C_1 \\ &= C_2 + n * C_1 \\ &= O(n) \end{aligned}$$

Substitution Example(2)

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1, & n > 1 \\ &= C_2, & n = 1 \end{aligned}$$

$$\text{BS}(n) = \text{BS}(n/2) + C_1$$

Substitution Example(2)

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1, & n > 1 \\ &= C_2, & n = 1 \end{aligned}$$

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1 \\ &= (\text{BS}(n/2^2) + C_1) + C_1 \\ &= \text{BS}(n/2^2) + 2 * C_1 \end{aligned}$$

Substitution Example(2)

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1, & n > 1 \\ &= C_2, & n = 1 \end{aligned}$$

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1 \\ &= (\text{BS}(n/2^2) + C_1) + C_1 \\ &= \text{BS}(n/2^2) + 2 * C_1 \\ &= \text{BS}(n/2^3) + 3 * C_1 \end{aligned}$$

Substitution Example(2)

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1, & n > 1 \\ &= C_2, & n = 1 \end{aligned}$$

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1 \\ &= (\text{BS}(n/2^2) + C_1) + C_1 \\ &= \text{BS}(n/2^2) + 2 * C_1 \\ &= \text{BS}(n/2^3) + 3 * C_1 \\ &\text{Substituting } \log_2 n \text{ times} \\ &= \text{BS}(1) + (\log_2 n) * C_1 \end{aligned}$$

Substitution Example(2)

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1, & n > 1 \\ &= C_2, & n = 1 \end{aligned}$$

$$\begin{aligned} \text{BS}(n) &= \text{BS}(n/2) + C_1 \\ &= (\text{BS}(n/2^2) + C_1) + C_1 \\ &= \text{BS}(n/2^2) + 2 * C_1 \\ &= \text{BS}(n/2^3) + 3 * C_1 \\ &\text{Substituting } \log_2 n \text{ times} \\ &= \text{BS}(1) + (\log_2 n) * C_1 \\ &= C_2 + (\log_2 n) * C_1 \\ &= O(\log_2 n) \end{aligned}$$

Substitution Example(3)

$$\begin{aligned} T(n) &= T(n-1) + n, \quad n > 1 \\ &= 1, \quad n = 1 \end{aligned}$$

$$T(n) = T(n-1) + n$$

Substitution Example(3)

$$\begin{aligned} T(n) &= T(n-1) + n, \quad n > 1 \\ &= 1, \quad n = 1 \end{aligned}$$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= (T(n-2) + (n-1)) + n \\ &= T(n-2) + n + (n-1) \end{aligned}$$

Substitution Example(3)

$$\begin{aligned} T(n) &= T(n-1) + n, \quad n > 1 \\ &= 1, \quad n = 1 \end{aligned}$$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= (T(n-2) + (n-1)) + n \\ &= T(n-2) + n + (n-1) \\ &= T(n-3) + n + (n-1) + (n-2) \end{aligned}$$

Substitution Example(3)

$$\begin{aligned} T(n) &= T(n-1) + n, \quad n > 1 \\ &= 1, \quad n = 1 \end{aligned}$$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= (T(n-2) + (n-1)) + n \\ &= T(n-2) + n + (n-1) \\ &= T(n-3) + n + (n-1) + (n-2) \end{aligned}$$

Substituting **n-1** times

$$= T(1) + n + (n-1) + (n-2) + \dots + (n-(n-k)) + 3 + 2$$

Substitution Example(3)

$$\begin{aligned} T(n) &= T(n-1) + n, \quad n > 1 \\ &= 1, \quad n = 1 \end{aligned}$$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= (T(n-2) + (n-1)) + n \\ &= T(n-2) + n + (n-1) \\ &= T(n-3) + n + (n-1) + (n-2) \end{aligned}$$

Substituting **n-1** times

$$\begin{aligned} &= T(1) + n + (n-1) + (n-2) + \dots + (n-(n-k)) + 3 + 2 \\ &= 1 + 2 + 3 + \dots + (n-2) + (n-1) + n \end{aligned}$$

Substitution Example(3)

$$\begin{aligned} T(n) &= T(n-1) + n, \quad n > 1 \\ &= 1, \quad n = 1 \end{aligned}$$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= (T(n-2) + (n-1)) + n \\ &= T(n-2) + n + (n-1) \\ &= T(n-3) + n + (n-1) + (n-2) \end{aligned}$$

Substituting $n-1$ times

$$\begin{aligned} &= T(1) + n + (n-1) + (n-2) + \dots + (n-(n-k)) + 3 + 2 \\ &= 1 + 2 + 3 + \dots + (n-2) + (n-1) + n \\ &= n*(n+1)/2 = (n^2 + 2*n + 1)/2 \\ &= O(n^2) \end{aligned}$$

Exercise

- Find Time Complexity of below recurrence relation:

$$T(n) = 2T(n/2) + 2, \quad n > 2$$

$$= 1, \quad n = 2$$

$$= 0, \quad n = 1$$

Exercise

- Find Time Complexity of below recurrence relation:

$$\begin{aligned}T(n) &= 2T(n/2) + 2, & n > 2 \\ &= 1, & n = 2 \\ &= 0, & n = 1\end{aligned}$$

T.C. of $T(n) = O(n)$

Solve Recurrence Relation

- Three methods to solve
 - Substitution
 - Recursive Tree Method
 - **Master Theorem**

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function and let $T(n)$ be defined on the non-negative integers by the recurrence

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

Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = O(n^{\log_b a - \epsilon})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n , then $T(n) = \Theta(n^{\log_b a + \epsilon})$