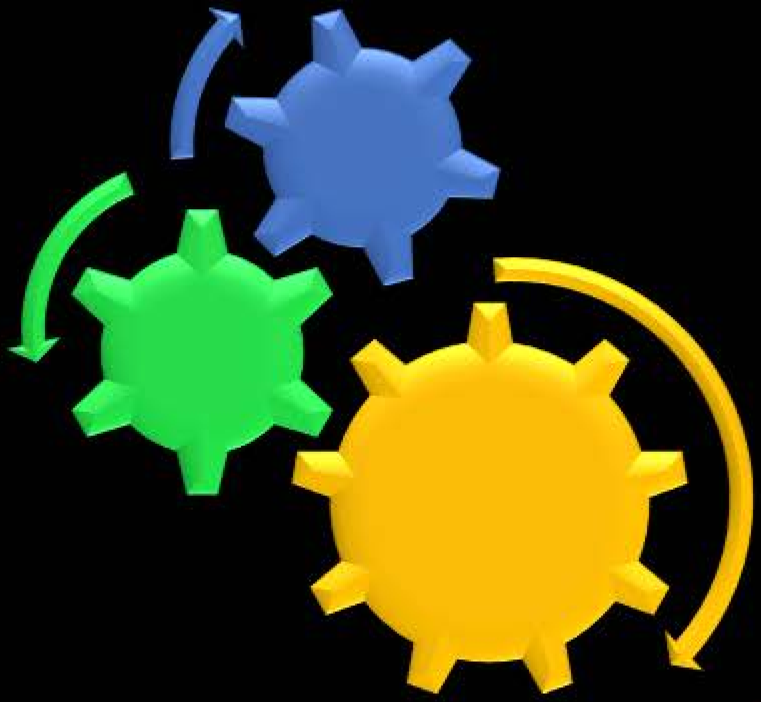


Space Complexity

· Recurrence Relation

Subsitution Method

(Algorithm)

· different type of Series

· Subsitution

· Solution

Solution

Subs $\boxed{n^{\log_b a}}$

Solve problem

$T(n) = 2T(n/2) + \log n$

$n^{\log_2 2}$ $\log n$

n $\log n$

$\theta(n)$

10mins

1min - Master Method

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$a \geq 1$

$b > 1$

$= 2T(n/2) + \log n$

$a = 2$, $b = 2$

$n^{\log_2 2}$

$= \theta\, n^1$

$f(n) = \log_2$

Space Complexity

5%

Scope wise : 2005 Linked Answer question

Space Complexity

Time Complexity - is Represented a function of Input Size.

T(n) ← Step count (programming step)

(RAM)

Space Complexity:

Amount of space (Memory Space) taken by an Algorithm

Cache, MainMemory, disk

as function of Input size.

α [fixed part, variable part]

Space Complexity: Fixed Part

High level Language → . Compiler → Executable file ← passive entity

⇓

put the file in memory (Load) execution — space

process & program ?

A program in in Execution

$\begin{cases} \text{ADD } R_1\ R_2 \\ \text{MUL } R_3\ R_4 \end{cases}$

$\begin{cases} \text{DIV } R_5\ R_1 \\ \text{MOV } R_6\ R_1 \end{cases}$

Space

Fixed part

1. Space taken by code
2. Variable
3. Constant (Numeric)

Space Complexity: Fixed Part

The space needed by algorithm is seen as sum of the following components. A fixed part, this part typically includes

- Instruction space (space for the code)
- Space for simple variables
- Space for constants

Space Complexity: Variable Part

1. Size of Input
2. Extra Space for Execution of program →

Cube Root of an integer

array a[1...n]

Binary Search

Linear Search (a[ ], int n , int x)

↑ Size of Input

CubRoot (m)

Sorting Algorithm

- In place (~~we~~ Do we need extra space for Sorting or the given Input array is sufficient)
- Not Inplace

Merge Sort Not Inplace

Space Complexity

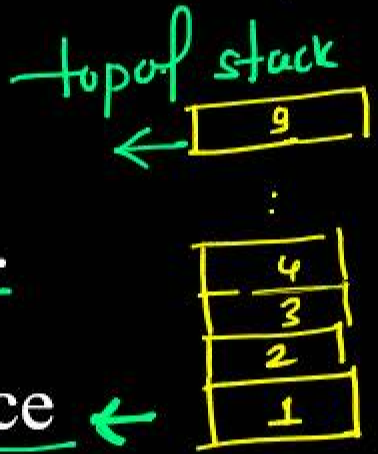
The second part

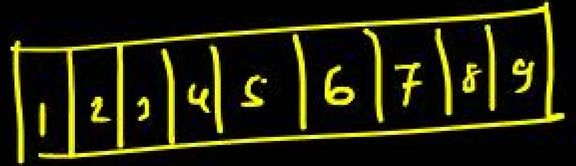
- Variable part, which consists of space needed by component variables whose size depends upon the particular problem instance being solved. This inlcudes
- Auxiliary space
- Space used by input.
- Recursion stack space

Runtime stack

Stack is "Last in first out"

Nested function call & Recursive function



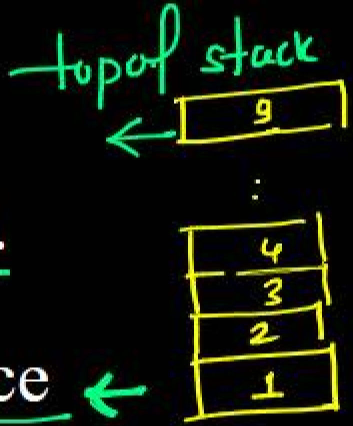


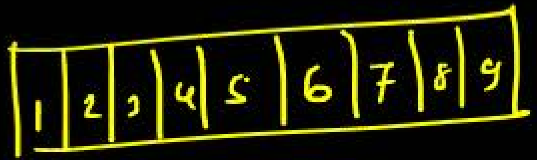
Space Complexity

The second part

- Variable part, which consists of space needed by component variables whose size depends upon the particular problem instance being solved. This inlcudes
- Auxiliary space
- Space used by input.
- Recursion stack space

Stack is "Last in first out"





stack

Code

static data

Heap

stack

Runtime stack

Nested function call

&

Recursive function

Space Complexity Example

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criterion than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort, and Heap Sort use O(1) auxiliary space. The space complexity of all these sorting algorithms is O(n) though.

Space Complexity Example

Space Complexity of Linear Search & Matrix Addition

Linear Search

Suhani

Lower Bound : $\Omega(1)$ $O(n)$ - upper Bound time complexity

10

```
int search(int arr[], int n, int x){
    int i;
    for (i=0; i<n; i++)
        if (arr[i] == x)
         return i;
    return -1;
}
```

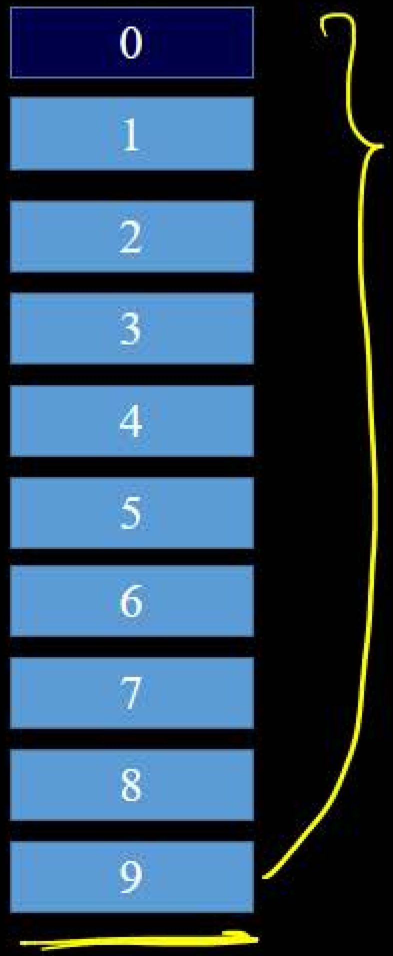
Array Input size

Array of Size n - Input

$\Theta(n) + C$

Linear Search Space Complexity

Array of Size n - $\Theta(n)$



Space Complexity

Space Complexly

Size of the array

Sum of an Array Elements	Total Steps
Algorithm Sum (a,n){	Size of array $\underline{n+c}$
s:= 0.0;	
for i:=1 to n do	
s = s + a[i]; → Access	if 1000 — 1000 Memory Location
return s; ↑ Amount of space Memory Location	100 - 100
} $\underline{a[1]}$, $\underline{a[2]}$, $\underline{a[3]}$ . . -	$\underline{a[n]}$ —
	$\underline{\Theta(n)}$

Space Complexity

matrix

Sum of Matrix (Matrix Addition Algorithm)	Total Steps
`Algorithm Add(a,b,c,n,n){`	a is — arrays — n^2 — n^2 b is — 2d — $n \times n$ — n^2 c - is — 2d — $n \times n^2$ — n^2
`for i := 1 to n do`	
`for j := 1 to n do`	
`c[i,j] :=a[i,j] + b[i,j];`	$3n^2 + C = \Theta(n^2)$
`}` $\boxed{n^2}$ I need Access $\boxed{n^2}$	
Total	$\Theta(n^2)$

Space Complexity

```
int exp(int X,  int Y)  {
        int res = 1, a = X,  b  = Y;
        while ( b != 0  ){
            if ( b%2 == 0) {
                a = a*a; b  = b/2;
            }
            else {
                res = res*a; b  = b-1; }
            }
            return res;
        }
```

time

a

Space Complexity

Array Subscript

- Some constant No. of variables

Space Complexity O(1)

Space Complexity Example

Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require O(n) space. If we create a two-dimensional array of size n*n, this will require $O(n^2)$ space.

Space Complexity Recursive Algorithm

```
int fib(int n){

    if (n <= 1)

    return n;

    return fib(n-1) + fib(n-2);

}
```

0 1 1, 2 3 5, 8, 13

2 3 4 5 6 7

$f(0) = 0$ $fib(0) = 0$

$fib(1) = 1$

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

$fib(2) = fib(1) + fib(0)$

$fib(3) = fib(2) + fib(1)$

Recursive program

Calling Sequence

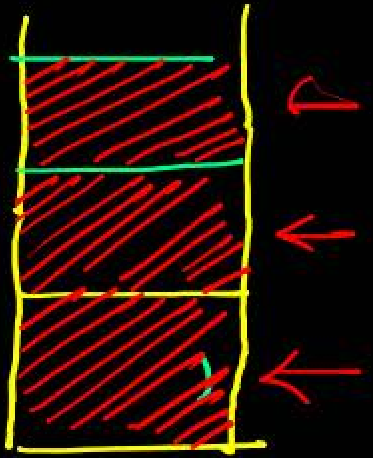
Space Complexity of Recursion

```
int main() {
    f1();
    f2()
}

f1() {
    f2()
}

f2() {
}
```

spend main Execution



Activation Records

List of values

Bass - Assignment I

III

1 ✓
2 ✓
3 ✓
4 ✓

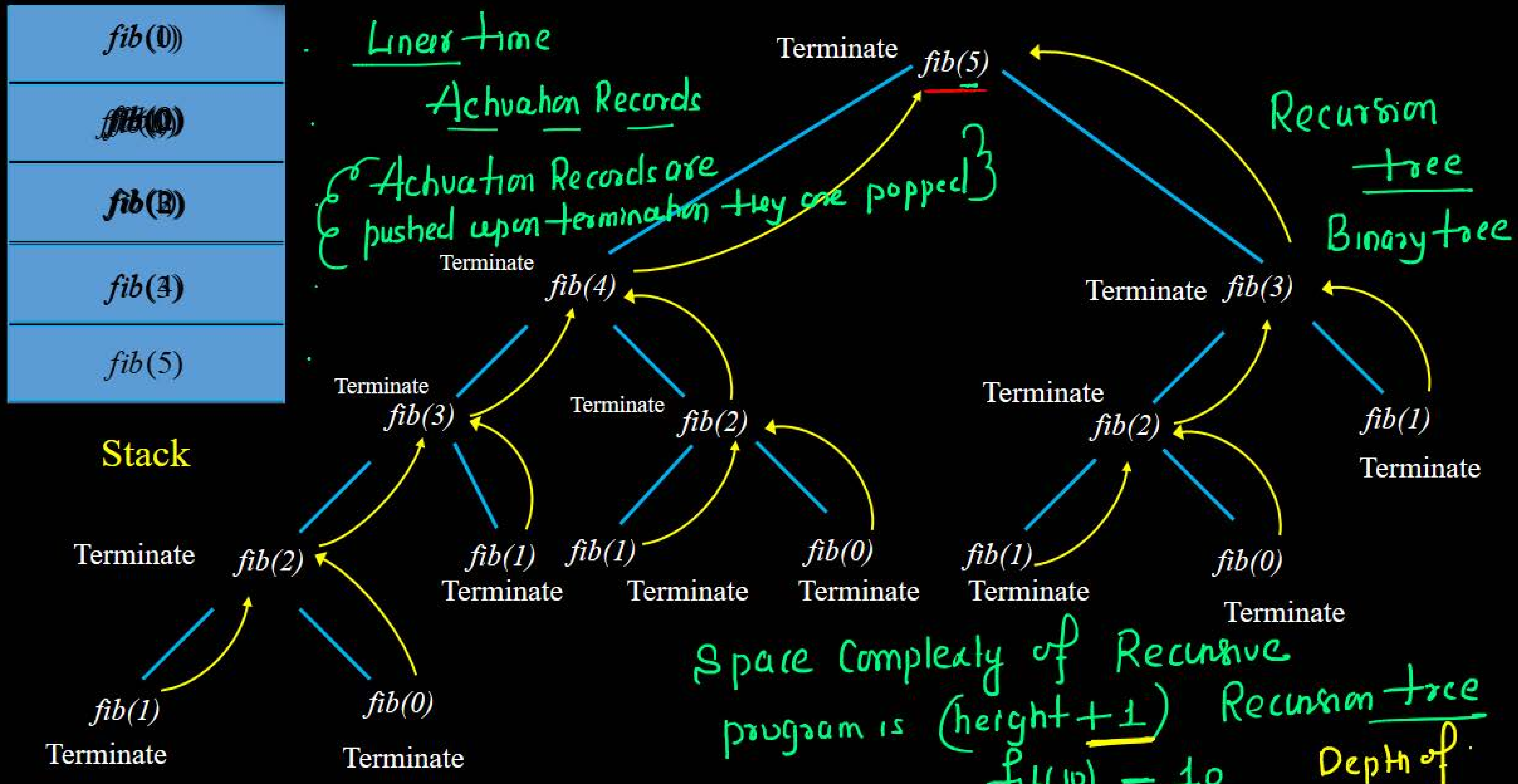
{S1}

Bass - II

1
2
3

1
2
3 - Bas
4

Space Complexity Recursive Algorithm



Linear time

Activation Records

{Activation Records are pushed upon termination they are popped}

Recursion tree

Binary tree

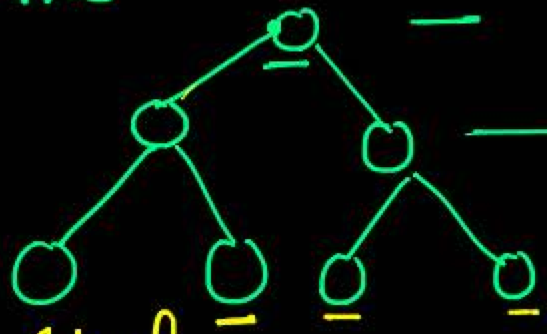
Space Complexity of Recursive program is (height + 1) Recursion tree

Depth of

fib(10) — 10

Binary tree

Root



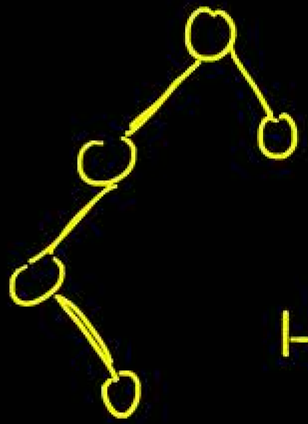
Root is at Height 0

height 1

height 2

1 Leaf

Height of the is longest Root to Leaf path



Binary tree

Height

3

Space Complexity Recursive Algorithm

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

$\Theta(n)$

Space Complexity of program

add(0)
add(1)
add(2)
add(3)
add(4)

add(4)

4+ add(3)

3+ add(2)

2+ add(1)

1+ add(0)

Height

4+1

n

n+1

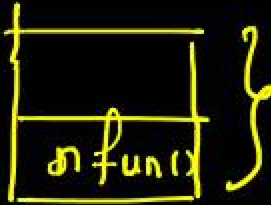
$\Theta(n)$

Space Complexity Iterative Algorithm

```
int fun (int n){
    int mul = 0;
    for (int i = 0; i < n; i++){
        mul += MulPair(i, i+1);
    }
    return sum;
}

int MulPair(int x, int y){
    return x * y;
}
```

1 2 ← terminate

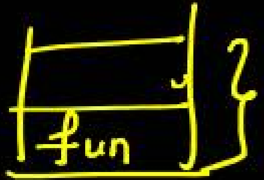


2space

mulpaire

Recursion? X

a Simple function



2 space

1, 2, 3, 4, 5, 6.

$0 + mul(1,2)$

$0 + 2 + mul(2,3)$

$0 + 2 + 6$

Stack space + variable

Constant constant

$\boxed{O(1)}$

GATE 2005 Question 81a

H.W worksheet

```
double foo (int n) {
    int i;
    double sum;

    if (n==0) return 1.0;
    else {
            sum = 0.0;
            for (i =0; i<n; i++)
                sum +=foo(i);
            return sum;
    }
}
```

Recursive program : Draw the Recursion

foo(4) — Height of Recursion

~~oly~~ only

Question

The space complexity of the above function is:

(A) $O(1)$ (B) $O(n)$ (C) $O(n!)$ (D) $O(n^n)$

GATE 2005 Question 81b

```
double foo (int n) {
    int i;
    double sum;

    if (n==0) return 1.0;
    else {
            sum = 0.0;
             for (i =0; i<n; i++)
                  sum +=foo(i);
             return sum;
    }
}
```

Suppose we modify the above function `foo()` and store the values of `foo(i)`, $0 \le I < n$, as and when they are computed. With this modification, the time complexity for function `foo()` is significantly reduced. The space complexity of the modified function would be:

(A) $O(1)$ (B) $O(n)$

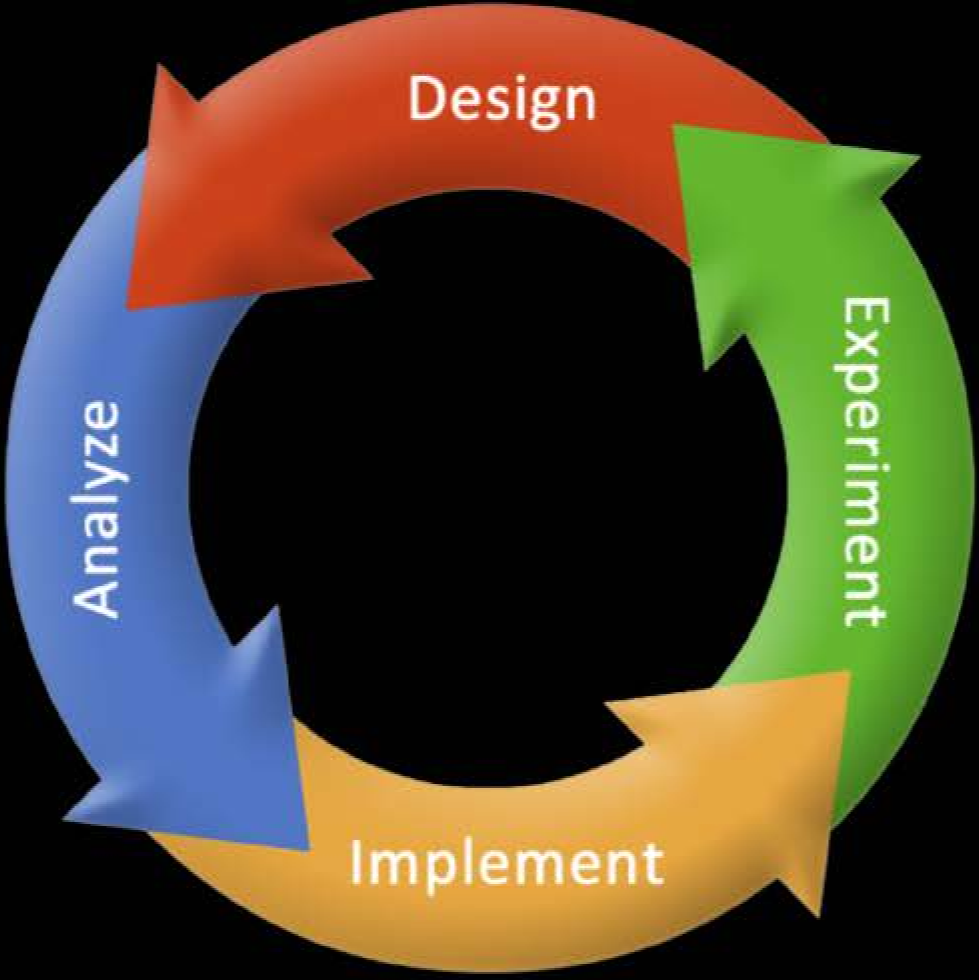
(C) $O(n^2)$ (D) $O(n!)$

GATE 2005 Question 81b

Suppose we modify the above function `foo()` and store the values of `foo(i)`, $0 \le I < n$, as and when they are computed. With this modification, the time complexity for function `foo()` is significantly reduced. The space complexity of the modified function would be:

(A) $O(1)$ (B) $O(n)$

(C) $O(n^2)$ (D) $O(n!)$



1. Asymptotic Notation
2. Asymptotic Complexity Completed
3. Recurrence Relation
   - Substitution ✓
   - ~~Recursion~~ Recurrence tree a α
   - Master Method α
4. Space complexity
5. Design
   - Divide & Conquer
   - Greedy
   - Dynamic programming
   -

Basic Algorithms Design

Basic Design Technique

1. Brute force Method (we find all possible answer from Solution space)

Linear Search

possibility — n Location

Looking & comparing each solution to ~~out~~ our requirement

Hamiltonian Cycle

2. Divide & Conquer

3. Greedy Method

4. Dynamic Programming

5. Branch & Bound

6. Backtracking

Basic Design Technique

- Brute-force or exhaustive search.
- Divide and Conquer.
- Greedy Algorithms.
- Dynamic Programming.
- Branch and Bound Algorithm.
- Randomized Algorithm.
- Backtracking.

Not in syllabus

$\boxed{2^n}$

$\boxed{{}^6C_2}$

Subset sum

aware of Set {}

All possible subset

2^6 — 2^n

S = {1, 6, 9, 7, 8, 10}

Set is collection of element

M = 16

problem : Is there exists elements in the Set where is sum is M (16) yes / NO

Solution — 9+6+1 = 16 {9, 6, 1} {10, 6}

8+7+1 = 16

SubSet of No element

Sub Set of 1 element

Subset of 2 element ✓

Subset of n element

* If I want to check every possible combination the How many combination I need to check!

Design an Algorithm

Let s be a sorted array of n integers. Let t(n) denote the time taken for the most efficient algorithm to determine if there are two elements with sum less than 1000 in s. ~~which of the following statements is true?~~

$O(1)$

(Check first 2 elements)

· Simple

Sorted array
n - integer

what is $t(n)$

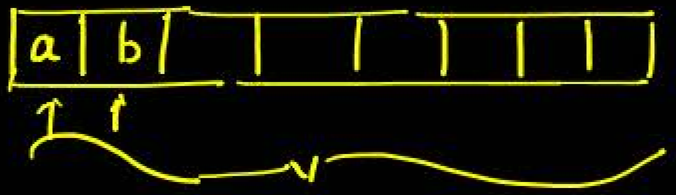
Every possible answer

what is Size of Solution space?

Answer in $^nC_2 = \frac{n(n-1)}{2} = \Theta(n^2)$

1 Better Answer

(II) Binary Search



1000 - x using

1000 - a other

1000 - x - Binary Search - $\log_2 n$

1000 - b -

n times binary - $n \log n$

minimal sum

yes | 1 | 2 | 3 | 1000 | 4000 | 8000 | 8001 | 8003 |

$<$

Any pair whose Sum $< \underline{1000}$

No | 1000 | 1001 | 1002 | 1003 | 1004 | 1006 | 1008 |

Any pair exists whose Sum is $\underline{< 1000}$

minimal sum is <1000 yes
if minimal is Not less than 1000 then Ans is No

first two element is i minimum element ~~at~~ array is already pair

time complexity

```
int fun (a[], n) {
  if (a[0] + a[1] < 1000)
     Yes
  else   No element exist
}
```

Constant Time Algorithm

- An Algorithm independent of input size.

Constant Time Algorithm

The constant time algorithm is the algorithm that does not depends upon the input size.

Design an Algorithm

An array a contains n integers in non-decreasing — Increasing Order order, Describe, ~~using Pascal like~~ pseudo code, a linear time algorithm to find i, j, such that ~~M~~ $a[i] + a[j] = m$ given integer M, if such i, j exist.

pair of element $a[i] + a[j] = M$

1	6	10	12	13	18	20	21

m = 25

i i j j

25

22 ≤

25 ←

Sum exists

Check every pair – n^2 – $^nC_2$

Binary Search – $n \log n$

Linear time:
only 1 scan

Linear Time Algorithm

```
i = 1;

j = n;

while(i != j) {

    if(A[i] + A[j] == M) break;

    else if(A[i] + A[j] < M) i++;

    else j--;

}
```

Condition For Pust

O(n) time

· Linear time

you

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=1}^{j} 1$$

$j-1+1$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} j$$

$i+1 + i+2 + i+3 \dots n$

$$= \sum_{i=1}^{n-1} \left[\frac{n(n+1)}{2} - \frac{i(i+1)}{2} \right]$$

$1+2+3+\dots+i+i+1+i+2+\dots+n$

$$\frac{n(n+1)}{2} - \frac{i(i+1)}{2}$$

$$= \frac{1}{2} \sum_{i=1}^{n-1} \left(n^2 + n - i^2 - i\right)$$

$$= \frac{1}{2} \left[n^2(n-1) + n(n-1) - \frac{(n-1)n(2n-1)}{6} - \frac{n(n-1)}{2} \right]$$

$$= \frac{1}{2} n(n-1) \left[n+1 - \frac{(2n-1)}{6} - \frac{1}{2} \right]$$

$$\frac{6n+6-2n+1-3}{6}$$

$$\frac{4n+4}{6}$$

$$\frac{1}{2} \frac{n(n-1)\cancel{4}^{2}(n+1)}{\cancel{6}_{3}}$$

$$= \frac{1}{3} n(n-1)(n+1)$$

Design an Algorithm

Compute
~~Computing~~ Matrix Transpose

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 6 \\ 3 & 8 & 9 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 7 \\ 4 & 5 & 6 \\ 3 & 8 & 9 \end{bmatrix}$$

what it is called

$A = A^T$?

Symmetric Matrix

```
Algorithm MatTran(A, n) {
    for i := 1 to n    (i:=2)
        for j := 1 to n
            Swap(A[i,j], A[j,i])
}
```

Not correct

is it correct?

H.W transpose

```
Swap(m, n) {
    t = m
    m = n
    n = t
}
```