

Introduction to Algorithm Design

(Performance Analysis of Algorithm)

Analysis of Algorithms

- **Analysis:** predict the cost of an algorithm in terms of resources and performance
- To analyze an algorithm means:
 - developing a formula for predicting *how fast* an algorithm is, based on the size of the input (**time complexity**), and/or
 - developing a formula for predicting *how much memory* an algorithm requires, based on the size of the input (**space complexity**)
- Usually time is our biggest concern
 - Most algorithms require a fixed amount of space
- **Design:** design algorithms which minimize the cost

Algorithm Analysis

- Once we have a correct algorithm for problem, we have to determine the efficiency of that algorithm.
- Analysis: analysis of resource usage of a given algorithm (resource: processor, memory). We are estimating of the amount of resources needed (as function of input size)

There are two aspects of algorithmic performance: 1.Space complexity 2.time complexity

- 1. **Space complexity** : The total amount of memory space needed for the complete execution of the given algorithm.
 - data structures take space
 - what kind of data structures can be used
 - how does choice of data structure affects the runtime?
- 2. **Time complexity** : the total amount of time needed for the complete execution of the given algorithm
 - instructions take time
 - how fast does the algorithm perform?
 - what affects its runtime?

What does “size of the input” mean?

- If we are searching an array, the “size” of the input could be the size of the array
- If we are merging two arrays, the “size” could be the sum of the two array sizes
- If we are computing the n^{th} Fibonacci number, or the n^{th} factorial, the “size” is n
- We choose the “size” to be the parameter that most influences the actual time/space required.

Let The problems have a natural “size”(N)

- N = Amount of data to be processed
- Resources used is proportional to $f(N)$
- f is a function

Algorithm Analysis

When we analyze algorithms we should employ mathematical technique that analyze algorithms independently of specific Implementations, computer or data.

To analyze algorithms:

1. First we start to count the number of significant operations in a particular solution to assess its efficiency.
2. Then we will express the efficiency of algorithms using functions

Algorithm Analysis

Space complexity:

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

Memory Usage while Execution: While executing, algorithm uses memory space for three reasons

1. **Instruction Space:** It's the amount of memory used to save the compiled version of instructions.
2. **Environmental Stack:** Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.
3. **Data Space:** Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

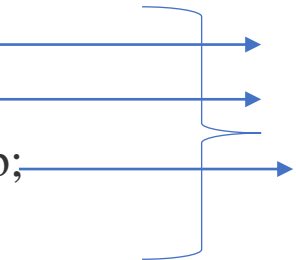
Algorithm Analysis

Let us understand the Space-Complexity calculation through examples.

Assume that to store an information memory required is one unit per word of memory

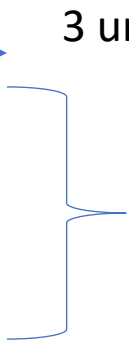
The space taken by variable declaration is fixed(constant) let $i=0$ required 1 unit of memory space.

`a:=5;`
`b:=5;`
`c:=a+b;`



1 In this 3 variables are used . So it takes 3 units of memory

`n , i, sum=0;`
`arr[n];`
`for i=0 to n do`
 `sum=sum+arr[i] ;`



3 unit

the array consists of n elements So, the space occupied by the array is n unit
Also we have 3 variables such as n, i and sum .it take 3 units of memory

Total= n+3
space complexity $S(n) \geq (n+3)$,

Algorithm Analysis

```
isum(a[],n)
```

```
{  s=0;
  for i=1 to n do
    s=s+a[i]
  return(s)
}
```

For variable like n, s, i one unit of space.
For array having n elements n units of space

Total $n+3$
 $S(n) \geq (n+3)$

```
Mat_mult(A[],B[],m,n,p,q)
```

```
{
  if (n!=p) then return(null);
  else
    for i=1 to m do
      for j=0 to q do
        C[i][j]=0 ;
        for k=1 to n do
          C[i][j] = C[i][j] +A[i][j]*B[i][j];
        return(c); }
}
```

$A[m][n] = mn$ units of memory
 $B[p][q] = pq$ units of memory
 $C[m][q] = mq$ units of memory
 $m, n, p, q, i, j, k = \text{each 1 unit of memory} = 7$
 $S(n) = n.n + n.n + n.n + 7 = 3n^2 + 7$
Assuming that m, n, p, q are nearly equal

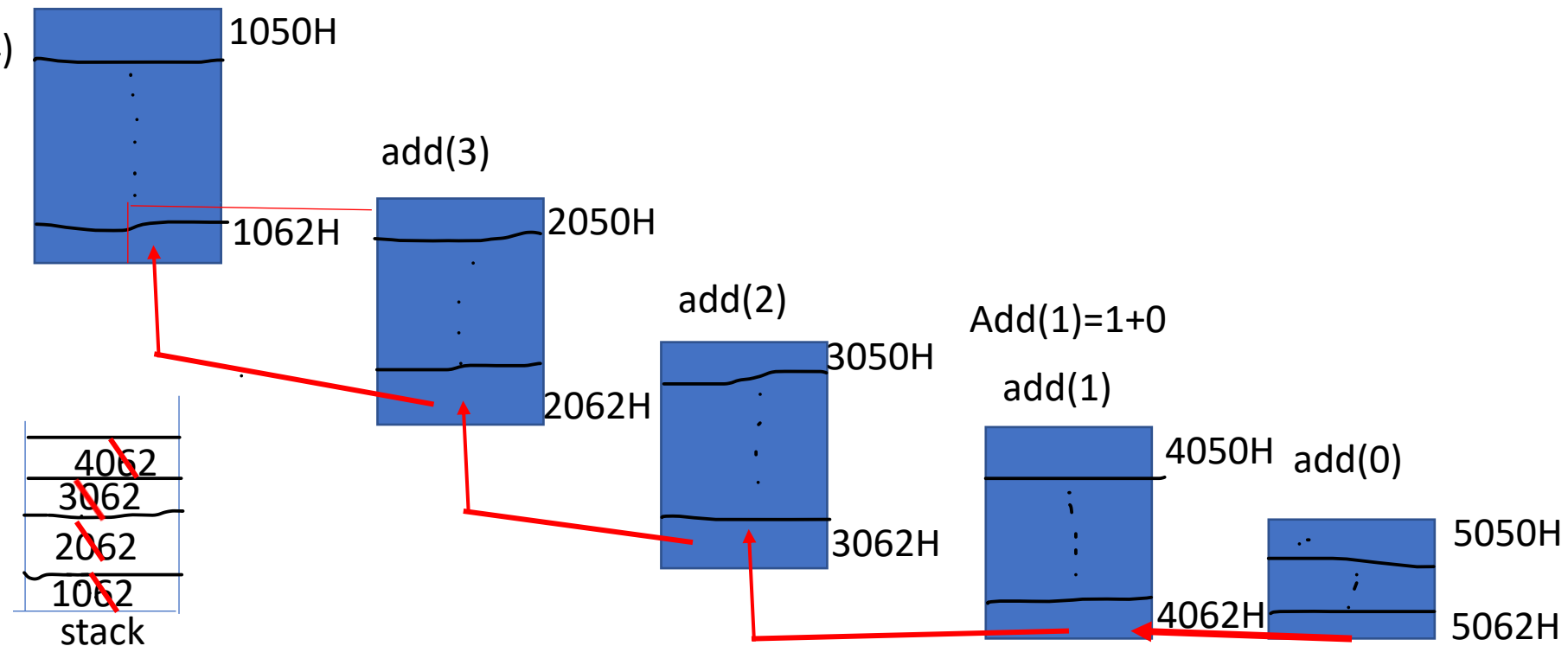
$$S(n) = n \cdot n + n \cdot n + n \cdot n + 7$$

Algorithm Analysis

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

n ----- 1 unit of space
return address ----- 1 unit of space
depth of recursion ----- n+1
Total $S(n) \geq (\text{total space required per call})(\text{depth of recursion}) = 2(n+1)$

1. $\text{add}(4) = 4 + \text{add}(3) = 10$
2. $\rightarrow \text{add}(3) = 3 + \text{add}(2)$
3. $\rightarrow \text{add}(2) = 2 + 1$
4. $\rightarrow \text{add}(1) = 1$
5. $\rightarrow \text{add}(0)$



Algorithm Analysis

Question:

```
swap ( a, b)
{
    int c;
    c = a;
    a = b;
    b = c;
}
```

```
i = n;
while (i >= n)
{
    i = i+1;
}
```

```
function max(A)
//input :an array A storing n integers
//output: the largest element in A(max of
A[1... n]
m=A[1]; i:=2
while i<=n do
    if A[i]>m then m:=A[i]
    i=i+1
return(m)
```

Algorithm Analysis

- Time complexity: Time complexity is the computational complexity that describes the amount of time it takes to run an algorithm.
- Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm
- We assume that each elementary operation takes a fixed amount of time to perform.
- Thus, the amount of time taken is the total number of elementary operations performed by the algorithm.
- Running time of the algorithm in a particular input i.e. the no of primitive operations or steps executed.
- **Frequency count /step count** method: we count number of times one instruction is executing. From that we will try to find the complexity of the algorithm.

Algorithm Analysis

Execution time in a way that depends only on the **algorithm** and its **input**. This can be achieved by choosing an **elementary operation**, which the algorithm performs repeatedly, and define the **time complexity** $T(n)$ as the number of such operations the algorithm performs given an array of length n .

we can choose the comparison $a[i] > \text{max}$ as an elementary operation.

1. since comparisons dominate all other operations in this particular algorithm.

2. Also, the time to perform a comparison is constant: it doesn't depend on the size of a .

The time complexity, measured in the number of comparisons, then becomes $T(n) = n - 1$.

In general, an **elementary operation** must have two properties:

1. There can't be any other operations that are performed more frequently as the size of the input grows.

2. The time to execute an elementary operation must be constant: it mustn't increase as the size of the input grows. This is known as unit cost.

Example:-

Algorithm max(a):

max \leftarrow **a[0]**

for **i = 1 to len(a)-1**

if **a[i] > max** -----**elementary operation**

max \leftarrow **a[i]**

return max

lsearch(a[],x)

{

for **i=1 to length(a) do**

{

if **x==a[i]** **then**

return true

}

}

Frequency count /step count method

To determine the step count of an algorithm, we first determine the number of steps per execution (s/e) of each statement and the total number of times (i.e., frequency) each statement is executed.

Combining these two quantities gives us the total contribution of each statement to the total step count. We then add the contributions of all statements to obtain the step count for the entire algorithm

Example:-

<i>function</i> isum(a[],n)	
s=0;	1
for i=1 to n	n+1
s=s + a[i]	n
return(s)	1

$$\text{Total } T(n) = 1 + n + 1 + n + 1 = 2n + 3$$

Program Segment A:

x = x+2

The frequency count of the statement in the program segment A is 1.(It gets executed once)

Program Segment B:

for k =1 to n do

 x = x+2----- n *1 =n unit times

end

the frequency count of the statement is n, since the **for** loop in which the statement is embedded executes $n(n \geq 1)$ times.

Program Segment C:

for j= 1 to n do

 for i= 1 to n do

 x = x+2----- n*n=n² unit time

the statement is executed $n^2(n \geq 1)$ times, since the statement is embedded in a nested for loop

for i=1 to n

s=s + a[i]

for (i=1; i<=5;i++) ----- 6 times

s=s+a[i]----- 5 times

i=1 true -> s=s+a[1]

i=2 true s=s+a[2]

i=3 true s=s+a[3]

i=4 true s=s+a[4]

i=5 true s=s+a[5]

i=6 false

.

.

.

Frequency count /step count method

Some basic assumptions are

- There is no count for { and } .
- Each basic statement like 'assignment' and 'return' have a count of 1.
- If a basic statement is iterated, then multiply by the number of times the loop is run.
- The loop statement is iterated n times, it has a count of $(n + 1)$. Here the loop runs n times for the true case and a check is performed for the loop exit (the false condition), hence the additional 1 in the count

Frequency count /step count method

Examples for Step-Count Calculation:

1. Sum of elements in an array	Step-count (T.C)	Step-count (Space)
Algorithm Sum(a,n)		
{		
sum = 0;	1	1 word for sum
for i = 1 to n do	n+1	1 word each for i and n
sum = sum + a[i];	n	n words for the array a[]
return sum;	1	
}	Total: 2n+3	(n+3) words

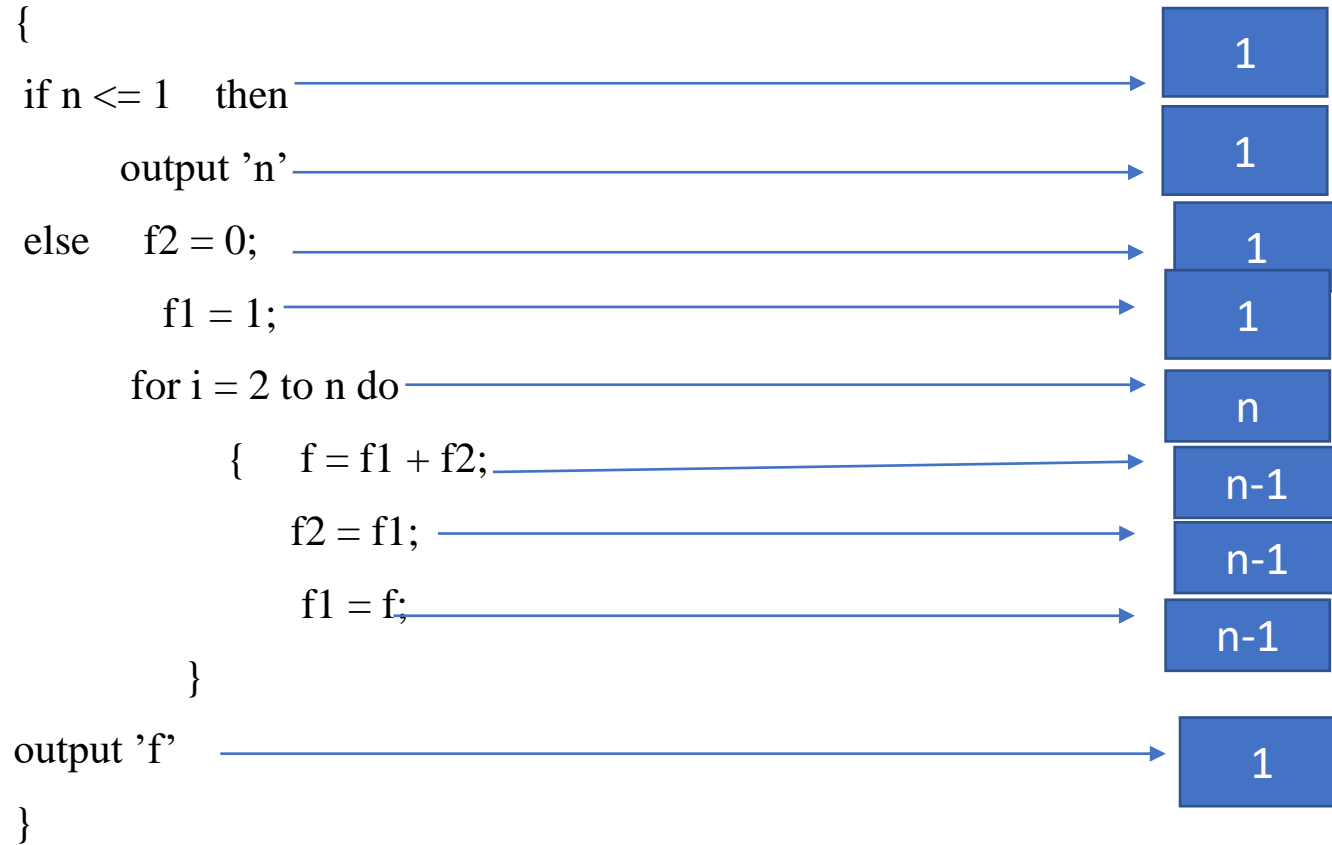
2. Adding two matrices of order m and n	Step Count
Algorithm Add(a, b, c, m, n)	
{	
for i = 1 to m do	m + 1
for j = 1 to n do	m(n + 1)
c[i,j] = a[i,j] + b[i,j] -	m.n
}	
	Total no of steps= 2mn + 2m + 2

The first 'for loop' is executed m + 1 times, i.e., the first m calls are true calls during which the inner loop is executed and the last call (m + 1)th call is a false call.

Frequency count /step count method

3. Fibonacci series

algorithm Fibonacci(n)



$$T(n) = 1 + 1 + 1 + 1 + n + n - 1 + n - 1 + n - 1 + 1 = 4n + 2$$

Frequency count /step count method

function bubble(a [], n)

{

for i=1 to n-1 do{  n

for j=1 to n-i do{  (n-1)(n-i+1)

if (a[j] > a[j+1]) then  (n-1)(n-i)

swap(a[j], a[j+1]); }  (n-1)(n-i)

}

}

lsearch(arr[],n,x)

{

for i=1 to n do

{

if x=arr[i] then

return true

}

return false

}

Frequency count /step count method questions

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

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)

3.What is the time complexity of following code:

```
old2 = 1;  
old1 = 1;  
for (i=3; i<n; i++)  
{  
    result = old2+old1;  
    old1 = old2;  
    old2 = result;  
}
```