



Data Structure & Algorithms

Sunbeam Infotech



Module Pre-requisites

- Java Programming Language
 - Language Fundamentals
 - Methods
 - Class & Object
 - static members
 - Arrays
 - Collections
 - Inner classes

- ✓ ① thinking
- ✓ ② algorithms
- ✓ ③ implementation
- ④ interviews



Module plan

1. Time & Space complexity ✓
2. Linear & Binary search ✓
3. Recursion ✓
4. Basic sorting ✓
5. Linked list ✓
6. Queues ✓
7. Stacks ✓
8. Trees ✓
9. Heap ✓
10. Advanced sorting ✓
11. Hashing ✓
12. Graphs ✓

Problem Solving Techniques

- ① Divide & Conquer
- ② Greedy approach
- ③ Dynamic Programming

Evaluations

- ① Theory - 40 marks - CCE
- ② Lab - 40 marks
- ③ Internal - 20 marks



Data Structure

- Data Structure

- Organizing data in memory
- Processing the data

} efficiently

- Basic data structures

- Array
- Linked List
- Stack
- Queue

- Advanced data structures

- Tree
- Heap
- Graph

- Data structures are ADT. ← Object Oriented

Abstract Data Types

Array ADT : random access

- ① get ele at index
- ② set ele at index

Stack ADT : LIFO

- ① push
- ② pop
- ③ peek
- ④ isEmpty

Queue ADT : FIFO

- ① push
- ② pop
- ③ peek
- ④ isEmpty



Data Structure

✓ Data Structure

- Organizing data in memory
- Processing the data

} efficient

✓ Common data structures

- Array
- Linked List
- Stack
- Queue

exact
analysis
* bytes
* time

asymptotic
analysis
* unit space
* iterations

✓ Advanced data structures

- Tree
- Heap
- Graph

• Asymptotic analysis

- It is not exact analysis

Big O
→ Order of

$1 \rightarrow \text{const } k$
 $n \rightarrow S \propto n$
 $O(n)$
 $n^2 \rightarrow S \propto n^2$
 $O(n^2)$

• Space complexity

- Unit space to store the data (Input space) and additional space to process the data (Auxiliary space).
- $O(1)$, $O(n)$, $O(n^2)$

• Time complexity

- Unit time required to complete any algorithm.
- Approximate measure of time required to complete any algorithm. → based on some factor.



Time complexity

- Time complexity of an algorithm depends on number of iterations in the algorithm.

- Common time complexities

- $O(1) \rightarrow T = k$
- $O(n) \rightarrow T \propto n \rightarrow$ single loop
- $O(n^2) \rightarrow T \propto n^2 \rightarrow$ nested loops - ②
- $O(n^3) \rightarrow T \propto n^3 \rightarrow$ nested loops - ③
- $O(\log n) \rightarrow T \propto \log n \rightarrow$ partitioning
- $O(n \log n) \rightarrow T \propto n \log n \rightarrow$ partitioning repeated

- Asymptotic notations

- Big O – $O(\dots)$ – Upper bound \rightarrow max time – worst case
- Omega – $\Omega(\dots)$ – Lower bound \rightarrow min time – best case
- Theta Θ – $\Theta(\dots)$ – Upper & Lower bound \rightarrow avg time – avg case

① check if num is prime.

$\rightarrow O(n)$:
`for(i=2; i<n; i++) {
 if(n%i == 0)
 pf("not prime");
}`
`if(i==n)
 pf("prime");`

② print table of "n".

$\rightarrow O(1)$: `for(i=1; i<=10; i++)
 pf(num * i);`



Linear Search

- Find a number in a list of given numbers (random order).

88	33	66	99	44	77	22	55	11
----	----	----	----	---------------	----	----	----	----

```
for (i = 0; i < n; i++) {  
    if (a[i] == key)  
        return i; // return index if found  
}  
return -1; // if not found.
```

- Time complexity

- Worst case \rightarrow max iterations $\rightarrow T \propto n \rightarrow O(n)$

- Best case \rightarrow min iterations $\rightarrow T = k \rightarrow \Omega(1) / \underline{O(1)}$
only 1 itr.

- Average case \rightarrow avg iterations $\rightarrow T \propto \frac{n}{2} \rightarrow \Theta(n) / O(n)$
 \downarrow
 $T \propto n$

Space Complexity

Input space: $T \propto n$
 $O(n)$

Aux Space: $S = k$
 $O(1)$



Binary Search

40

* for sorted array only.
while($l \leq r$)

{ $m = (l + r) / 2$

if ($key == a[m]$)
return m ;

if ($key < a[m]$)
 $r = m - 1$;

else

$l = m + 1$;

}

return -1 ;

0	1	2	3	4	5	6	7	8
11	22	33	44	55	66	77	88	99

↑
R

↑
L

↑
m

← left part

← right part

$$2^i \approx n$$

$$2^{(4)} = 16$$

$$i \log 2 = \log n$$

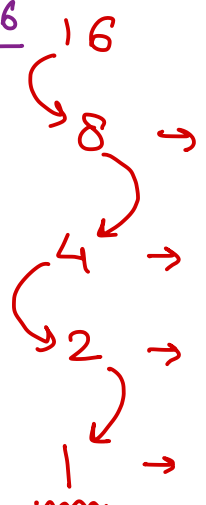
$$i = \frac{\log n}{\log 2}$$

$$T \propto i$$

$$T \propto \frac{\log n}{\log 2}$$

$$T \propto \log n$$

$$\rightarrow \underline{O(\log n)}$$



Recursion - Divide & Conquer

- Function calling itself is called as recursive function.
- To write recursive function consider
 - Explain process/formula in terms of itself
 - Decide the end/terminating condition

Examples:

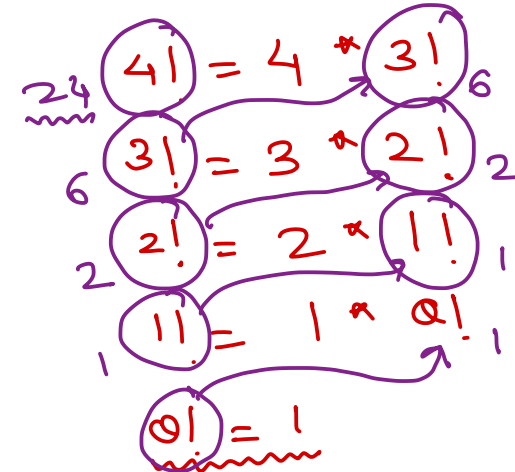
- $n! = n * (n-1)!$
 - $x^y = x * x^{y-1}$
 - $T_n = T_{n-1} + T_{n-2}$
 - $\text{factors}(n) = \text{prime factor of } n * \text{factors}(n / \text{prime factor})$
- base*
- $0! = 1$
 $x^0 = 1$
 $T_1 = T_2 = 1$
 $\text{factors}(1) = 1$

- Function call in recursion
- Pros & Cons of recursion

- On each function call, function activation record or stack frame will be created on stack.

```
int fact(int n) {  
    int r;  
    if(n==0)  
        return 1;  
    r = n * fact(n-1);  
    return r;  
}
```

back-Substitution

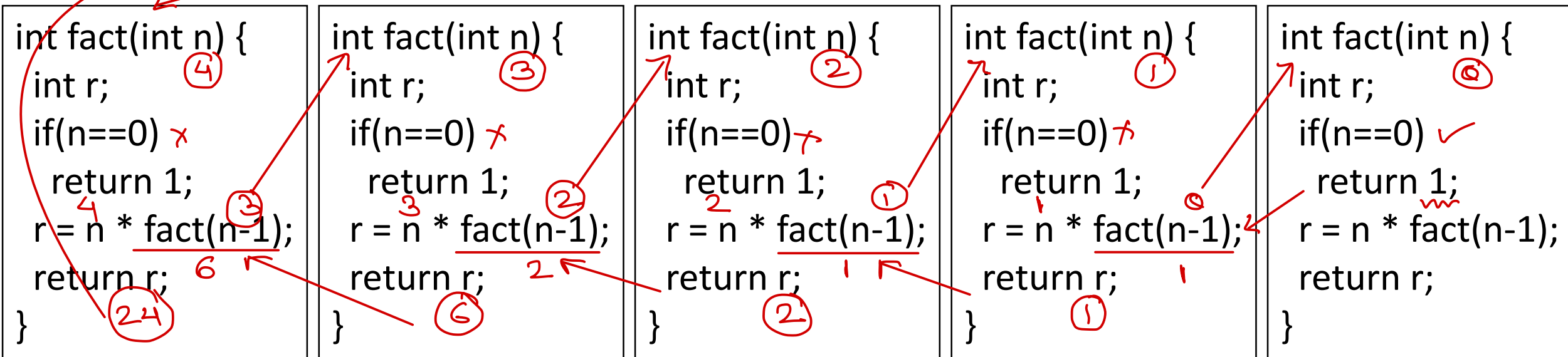


$$\begin{aligned} 2^3 &= 2 * 2^2 \\ 2^2 &= 2 * 2^1 \\ 2^1 &= 2 * 2^0 \\ 2^0 &= 1 \end{aligned}$$

$\text{res} = \text{fact}(4);$
→ return addr.

Recursion

24
res = fact(4)
nm



function activation record / stack frame
is created on stack for each fn call.

- * created when fn called
- * destroyed when fn return

FAR contains

- ① local variables
- ② arguments
- ③ return address

↳ addr of next instrn
to be executed when
fn returns.
= addr of next instrn
after fn call.



Recursion

main() {
res = fact(4);
}

```
int fact(int n) {  
    int r;  
    if(n==0)  
        return 1;  
    r = n * fact(n-1);  
    return r;  
}
```

```
int fact(int n) {  
    int r;  
    if(n==0)  
        return 1;  
    r = n * fact(n-1);  
    return r;  
}
```

```
int fact(int n) {  
    int r;  
    if(n==0)  
        return 1;  
    r = n * fact(n-1);  
    return r;  
}
```

```
int fact(int n) {  
    int r;  
    if(n==0)  
        return 1;  
    r = n * fact(n-1);  
    return r;  
}
```

```
int fact(int n) {  
    int r;  
    if(n==0)  
        return 1;  
    r = n * fact(n-1);  
    return r;  
}
```

~~f(4): n=4, r=~~
~~ret = m()~~

m: res, ...
ret = jvm/os

f(3): n=3, r=

ret = f()

f(4): n=4, r=

ret = m()

m: res, ...
ret = jvm/os

f(2): n=2, r=

ret = f()

f(3): n=3, r=

ret = f()

f(4): n=4, r=

ret = m()

m: res, ...
ret = jvm/os

~~f(1): n=1, r=~~
~~ret = f()~~

f(2): n=2, r=

ret = f()

f(3): n=3, r=

ret = f()

f(4): n=4, r=

ret = m()

m: res, ...
ret = jvm/os

~~f(0): n=0, r=~~
~~ret = f()~~

f(1): n=1, r=

ret = f()

f(2): n=2, r=

ret = f()

f(3): n=3, r=

ret = f()

f(4): n=4, r=

ret = m()

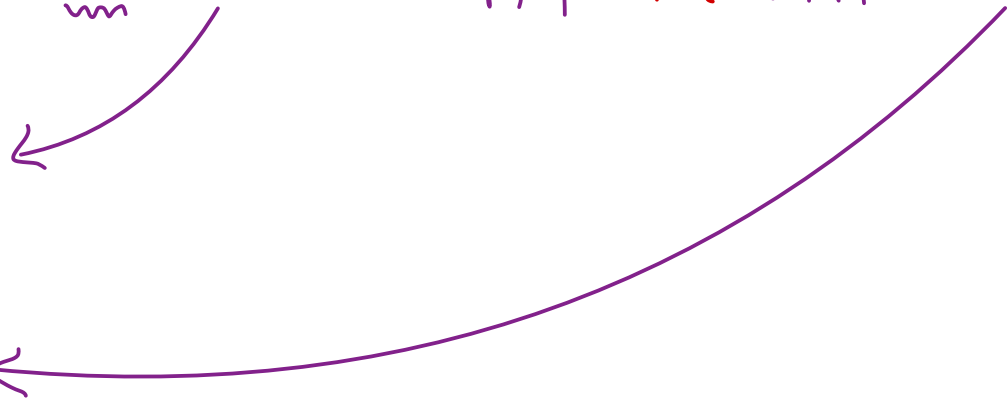
m: res, ...
ret = jvm/os

Binary Search

```
✓  $m = (l + r) / 2;$   
  if (key == a[m])  
    return key  
  if (key < a[m])  
    binSearch(l, m-1, key)  
  else  
    binSearch(m+1, r, key)
```

base cond: right < left
 ↑

0	1	2	3	4	5	6	7	8
11	22	33	44	55	66	77	88	99
<u>L</u>			<u>m-1</u>	<u>m</u>	<u>m+1</u>			<u>R</u>



Sorting

- Arranging array elements in ascending or descending order.
- Popular sorting algorithms
 - ✓• Selection sort
 - ✓• Bubble sort
 - ✓• Insertion sort
 - ✓• Quick sort
 - ✓• Merge sort
 - ✓• Heap sort



Selection Sort

6 4 2 8 3 1

```
for (i=0; i<5; i++) {  
    for (j=i+1; j<6; j++) {  
        if (a[i] > a[j])  
            Swap(a[i], a[j]);  
    }  
}
```

0	1	2	3	4	5
1	2	3	4	6	8

↑ (red arrow from index 4 to index 5)
↑ (blue arrow from index 5 to index 4)

3

$$i = (n-1) + (n-2) + (n-3) + \dots + 1$$
$$i = \frac{n(n-1)}{2}$$

$$T \propto \frac{n^2 - n}{2}$$

$$T \propto n^2 - n$$

$$T \propto n^2$$

$$\underline{O(n^2)}$$

Theory of Approximation

$$n \gg 1$$

$$\boxed{n^2 \gg n}$$

all lower order terms
can be neglected.

i=0	j → 1 to 5	- 5
i=1	j → 2 to 5	- 4
i=2	j → 3 to 5	- 3
i=3	j → 4 to 5	- 2
i=4	j → 5 to 5	- 1

+ 15



Selection Sort

class Person: id, name, age.

0	1	2	3	4	5
1	5	6	4	7	3
B	X	P	G	N	Q
24	20	28	30	39	22

```
for (i=0; i<5; i++) {  
    for (j=i+1; j<6; j++) {  
        if (a[i].age > a[j].age)  
            Swap(a[i], a[j]);  
    }  
}
```

3

}





Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

