SPACE COMPLEXITY
BEST & WORST CASE ANALYSIS

Problem Solving with Computers-II

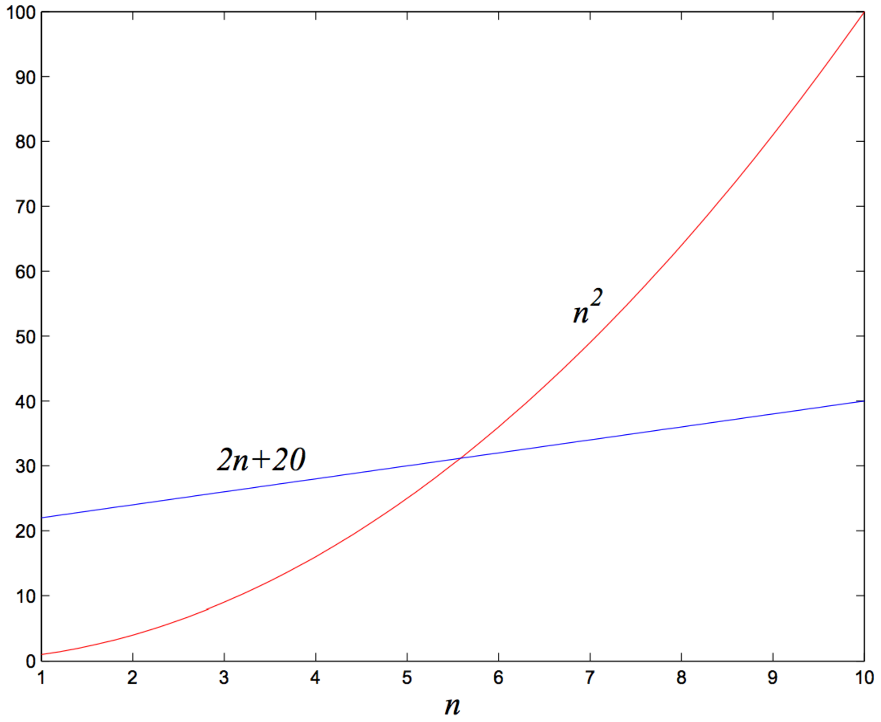
```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

Definition of Big-O

f(n) and g(n) map positive integer inputs to positive reals.

We say f = O(g) if there is a constant c > 0 and k>0 such that
f(n) ≤ c · g(n) for all n >= k.

f = O(g)
means that "f grows no faster than g"

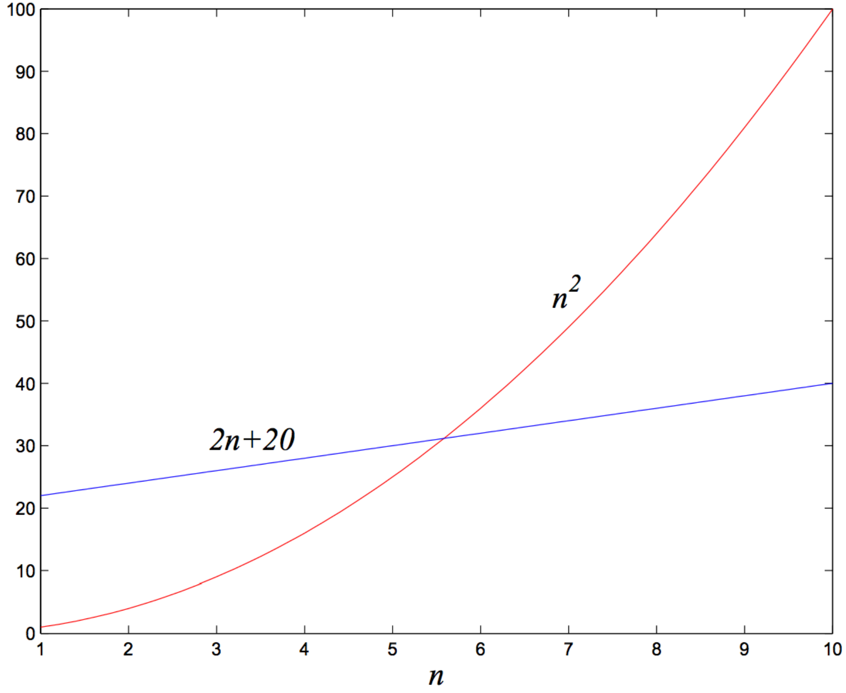


Big-Omega

- f(n) and g(n) map positive integer inputs to positive reals.

We say $f = \Omega(g)$ if there are constants $c > 0$, $k>0$ such that
$c \cdot g(n) \leq f(n)$ for $n >= k$

$f = \Omega(g)$
means that "f grows at least as fast as g"

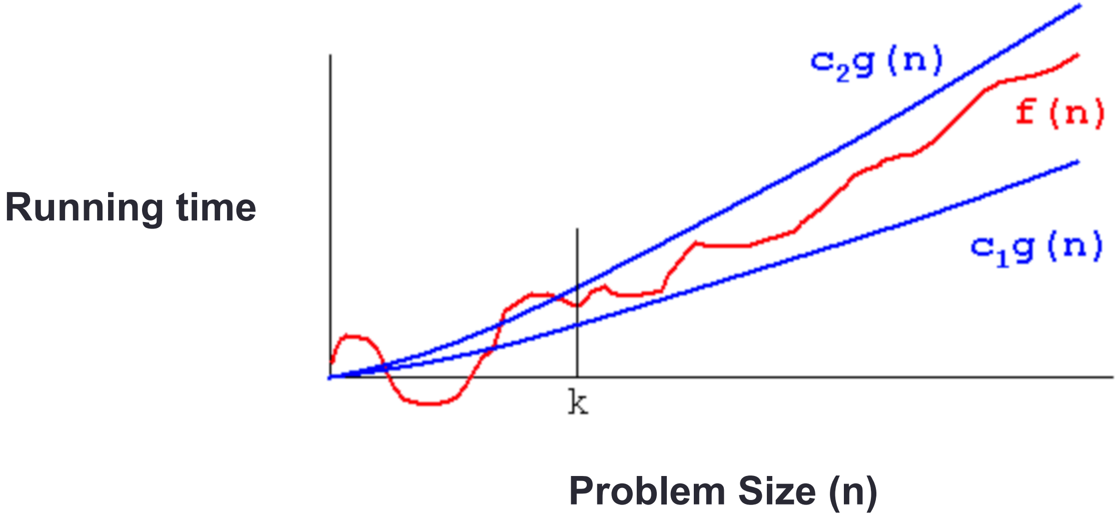


Big-Theta

- $f(n)$ and $g(n)$ map positive integer inputs to positive reals.

We say $f = \Theta(g)$ if there are constants c_1, c_2, k such that

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, for $n >= k$



Space Complexity

Lets S(n) = maximum amount of memory needed to compute F(n)

```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```

What is S(n)? Express your answer in Big-O notation

What is S(n)? Express your answer in Big-O notation

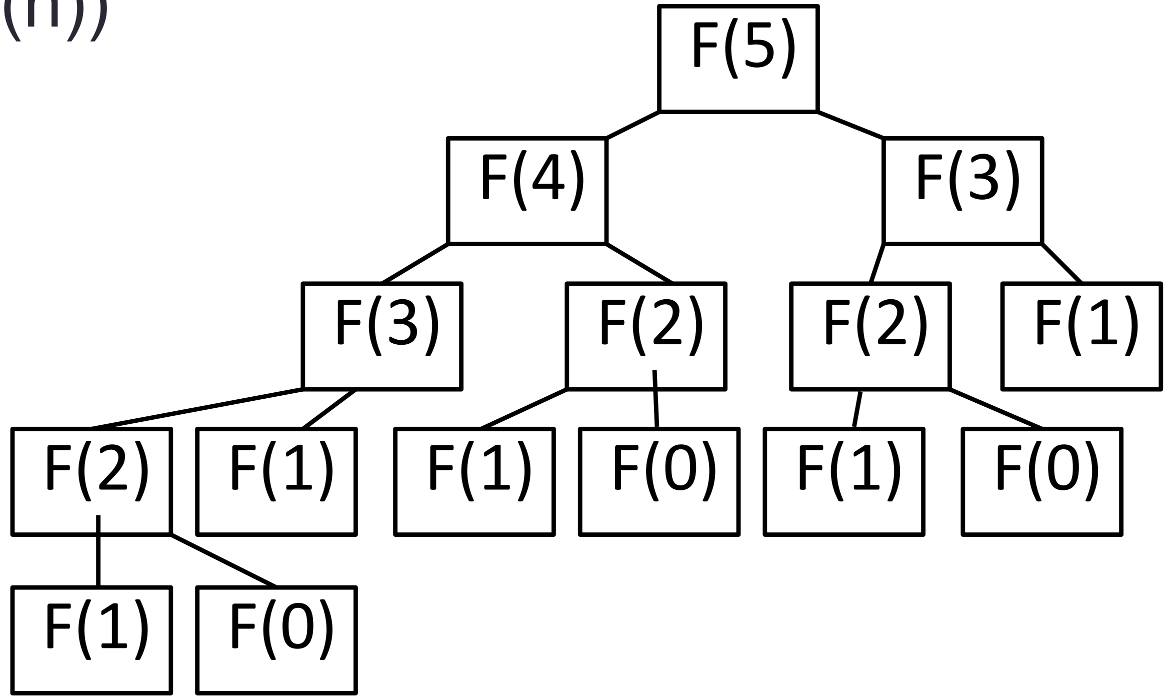
A. $O(1)$

B. $O(\log(n))$

C. $O(n)$

D. $O(n^2)$

E. $O(2^n)$



Tree of recursive calls needed to compute F(5)

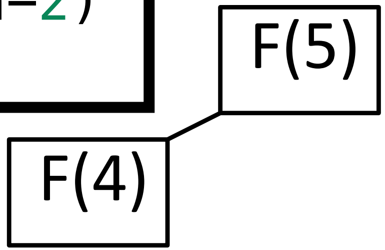
S(n) relates to maximum depth of the recursion

```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```

F(5)

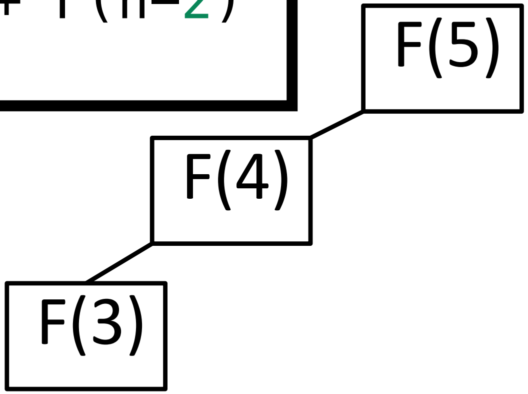
S(n) relates to maximum depth of the recursion

```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



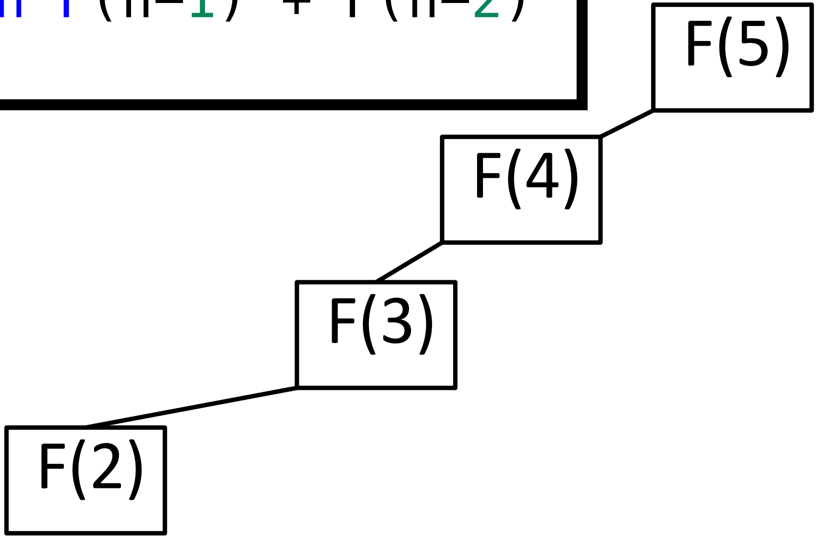
S(n) relates to maximum depth of the recursion

```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



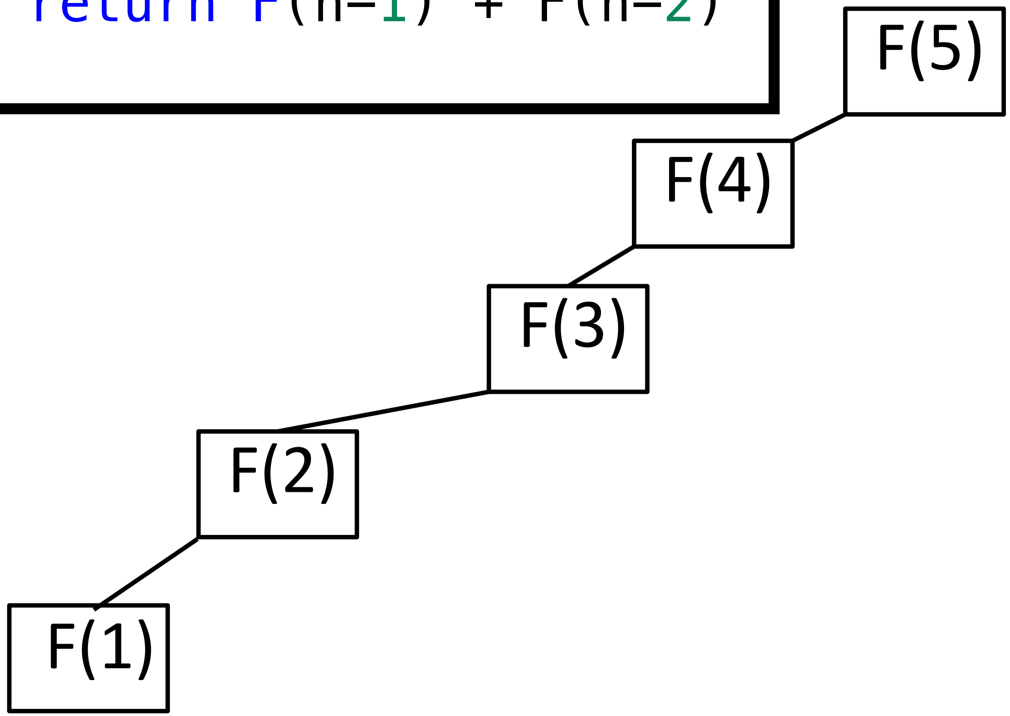
S(n) relates to maximum depth of the recursion

```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



S(n) relates to maximum depth of the recursion

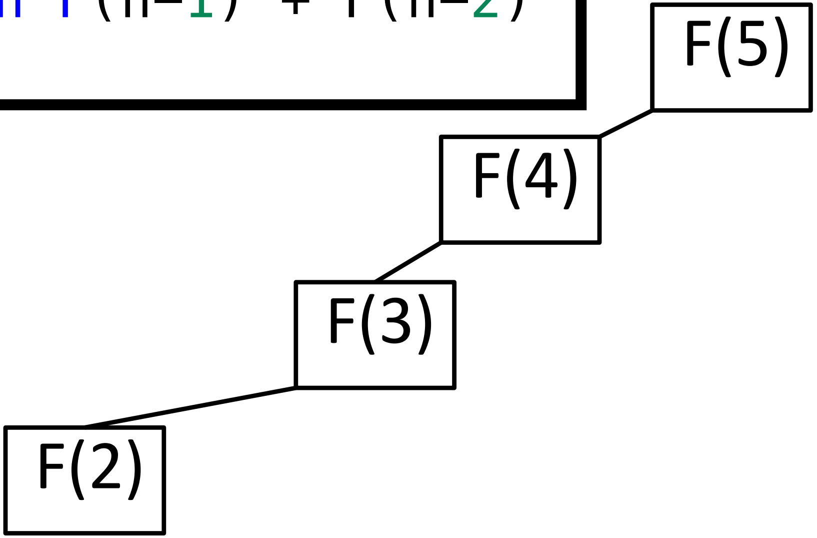
```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

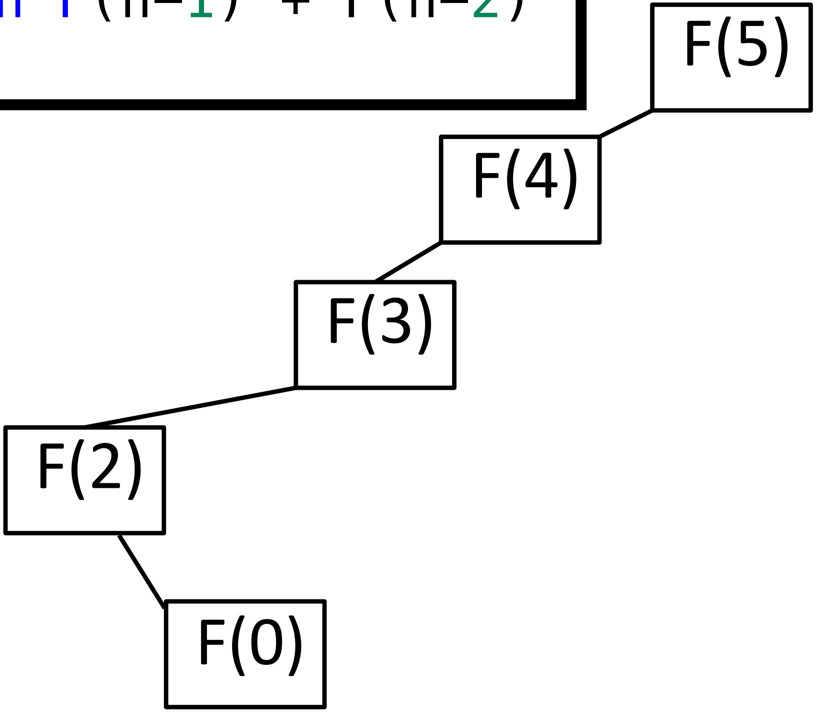
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

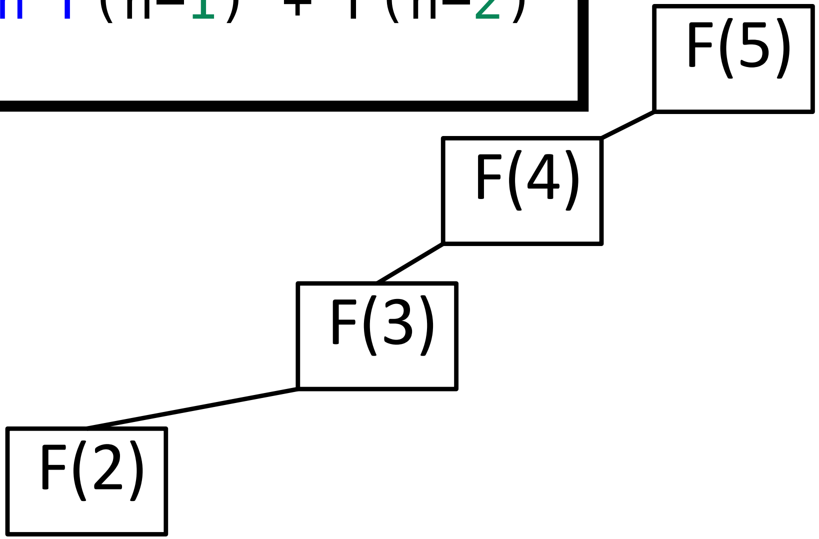
```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



F(5)

F(4)

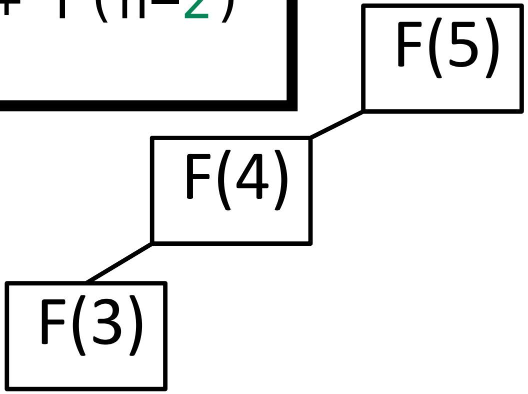
F(3)

F(2)

Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

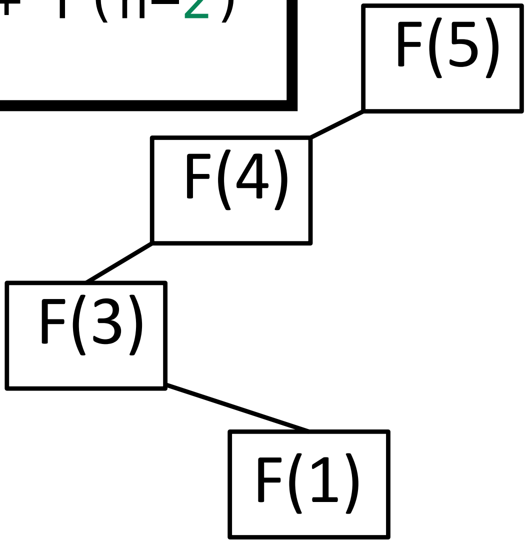
```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

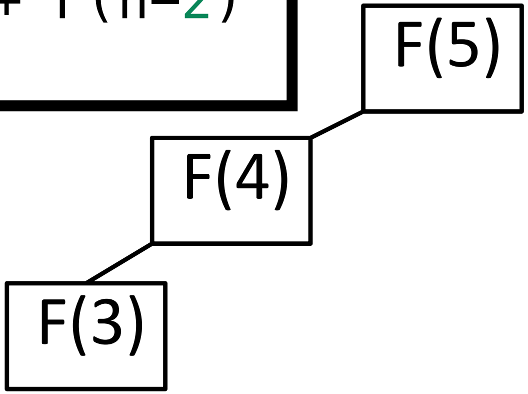
```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

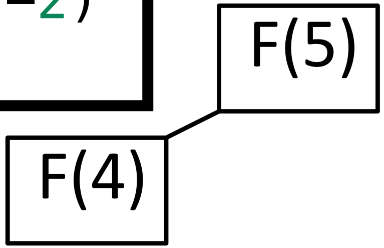
```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

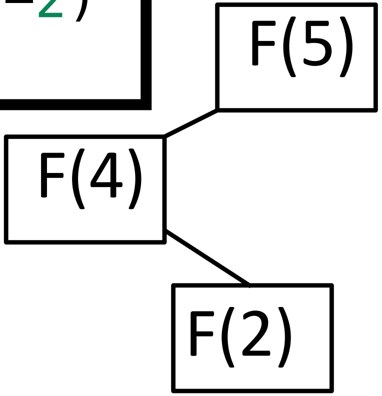
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

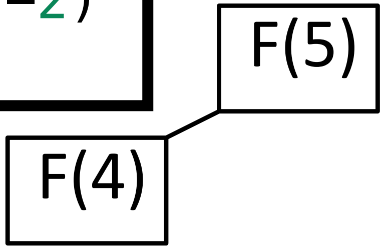
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

```
F(int n){
   if(n <= 1) return 1
   return F(n–1) + F(n–2)
}
```

F(5)

What is the next step ?

A.Recursion ends and F(5) returns

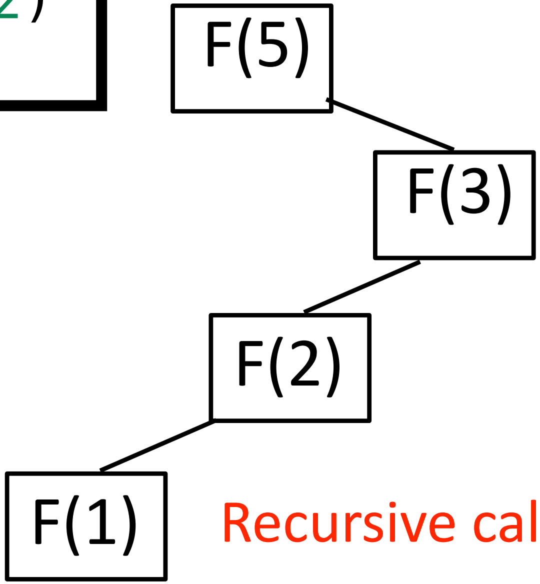
B.F(5) calls F(4)

C.F(5) calls F(3)

D.None of the above

S(n) relates to maximum depth of the recursion

```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Recursive calls continue

Maximum depth of the recursion for F(n) = n

Therefore, S(n) = O(n)

What is the Big-O running time of search in a sorted array of size n?

…using linear search?

Bestcase : searching for min value O(1)

Worstcase: O(n)

…using binary search?

Best case: search mid value O(1)

Worst case: O(log n)

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Worst case analysis of binary search

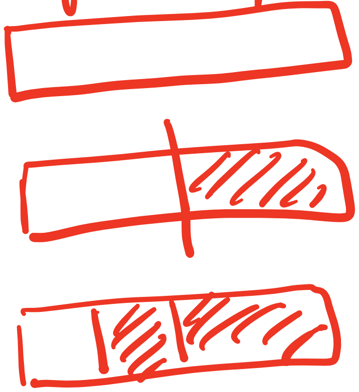
```
bool binarySearch(int arr[], int element, int n){
//Precondition: input array arr is sorted (ascending order)
   int begin = 0;
   int end = n-1;
   int mid;
   while (begin <=  end){
     mid = (end + begin)/2;
     if(arr[mid]==element){
       return true;
     }else if (arr[mid]< element){
       begin = mid + 1;
     }else{
       end = mid - 1;

     }
   }
   return false;
}
```

$O(1)$

$O(1)$

$\}O(1)$

Iteration #	(end-begin)	Size of array (approx)
1	(n-1)	n
2		$\frac{n}{2}$
3		$\frac{n}{4}$
4		
⋮		
k	....	$\boxed{\frac{n}{2^{k-1}}}$



Stopping condition.

$$\frac{n}{2^{k-1}} < 1$$

$$n < 2^{k-1}$$

$$\log_2 n < k-1$$

$$k > \log_2 n + 1$$

\# of iterations $= O(\log n)$

$T(n) = O(\log n)$

A more accurate analysis leads to the same final running time ~~of~~ O(log n). Specifically we will show that the no. of times the loop runs is O(log n)

Loop iteration #	(end - begin)
	$(n-1)$
1	
2	$\frac{(n-1)}{2} - 1$
3	$\frac{1}{2}\left(\frac{n-1}{2} - 1\right) - 1$
	$= \frac{n-1}{4} - \frac{1}{2} - 1$
4	$\frac{1}{2}\left(\frac{n-1}{4} - \frac{1}{2} - 1\right) - 1$

Get a general expression for (end - begin) in iteration k on the k^{th} iteration,

end - begin $= \frac{n-1}{2^{k-1}} - \left(1+\frac{1}{2}+\frac{1}{4}+\cdots\frac{1}{2^{k-2}}\right)$

(summing the geometric series)

$$= \frac{(n-1)}{2^{k-1}} - \frac{\left(1-\frac{1}{2^{k-1}}\right)}{\left(1-\frac{1}{2}\right)}$$

$$= \frac{(n-1)}{2^{k-1}} - 2\left(1-\frac{1}{2^{k-1}}\right)$$

Stop condition for the loop is

when end - begin < 0

$$\frac{n-1}{2^{k-1}} - 2 + \frac{2}{2^{k-1}} < 0$$

Solve for k

$\frac{n+1}{2^{k-1}} < 2$, $k > \log\left(\frac{n+1}{2}\right) + 1$

Therefore # of times the loop runs is $O(\log(n))$

Running time of operations in a sorted array

	Best case	Worst case
Search (Binary search)	O(1)	O(logn)
Min/Max	O(1)	O(1)
Median	O(1)	O(1)
Successor/Predecessor	O(1)	O(1)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

4

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
procedure max($a_1, a_2, \ldots a_n$: integers)
 max:= $a_1$
 for i:= 2 to n
     if max < $a_i$
         max:= x
return max{max is the greatest element}
```

What is the **best case** Big-O running time of max?

A. $O(1)$

B. $O(\log n)$

[✓ C. $O(n)$]

D. $O(n^2)$

E. None of the above

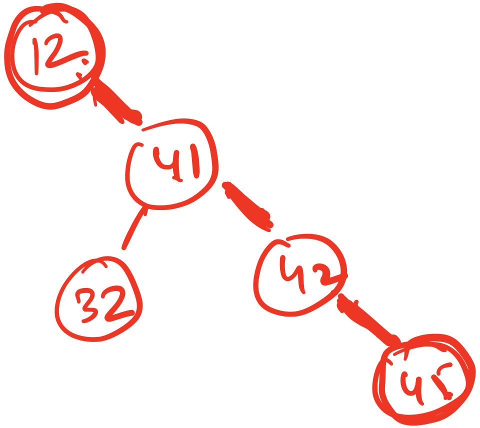
```
procedure max($a_1$,$a_2$, … $a_n$: integers)
  max:= $a_1$
  for i:= 2 to n
      if max < $a_i$
          max:= x
return max{max is the greatest element}
```

What is the **worst case** Big-O running time of max?

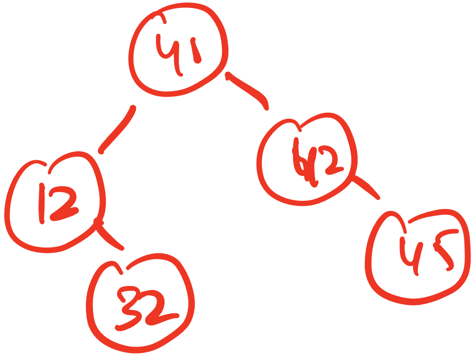
A. O(1)
B. O(log n)
[✓ C.] O(n)
D. O(n^2)
E. None of the above

- Path – a sequence of (zero or more) connected nodes.
- Length of a path - number of edges traversed on the path
- Height of node – Length of the longest path from the node to a leaf node.
- **Height of the tree** - Length of the longest path from the **root** to a leaf node.

H



height = 3



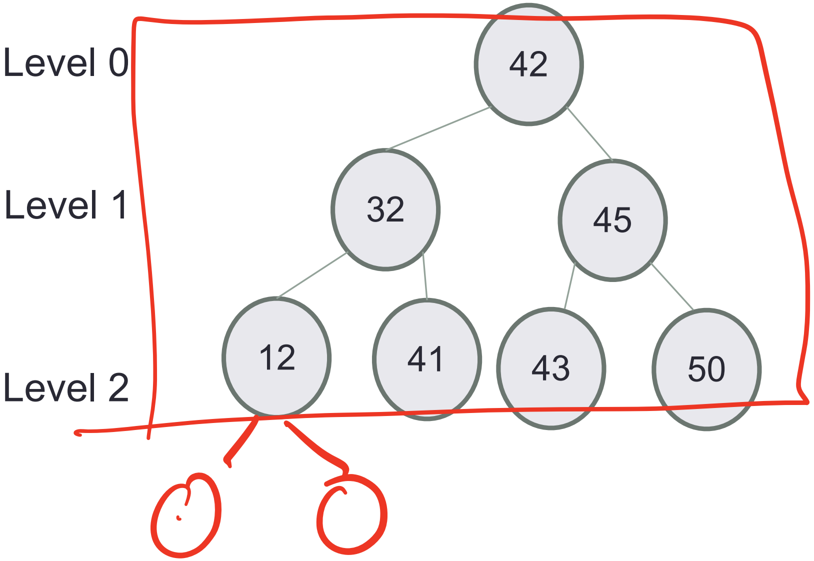
height = 2

BSTs of different heights are possible with the same set of keys

Examples for keys: 12, 32, 41, 42, 45

Types of BSTs

Set : AVL, Red Black



Balanced BST: height $(n) = O(\log n)$

Complete Binary Tree: Every level, except possibly the last, is completely filled, and all nodes on the last level are as far left as possible

Full Binary Tree: A complete binary tree whose last level is completely filled

Relating H (height) and n (#nodes) for a full binary tree

$N = 2^{H+1} - 1$

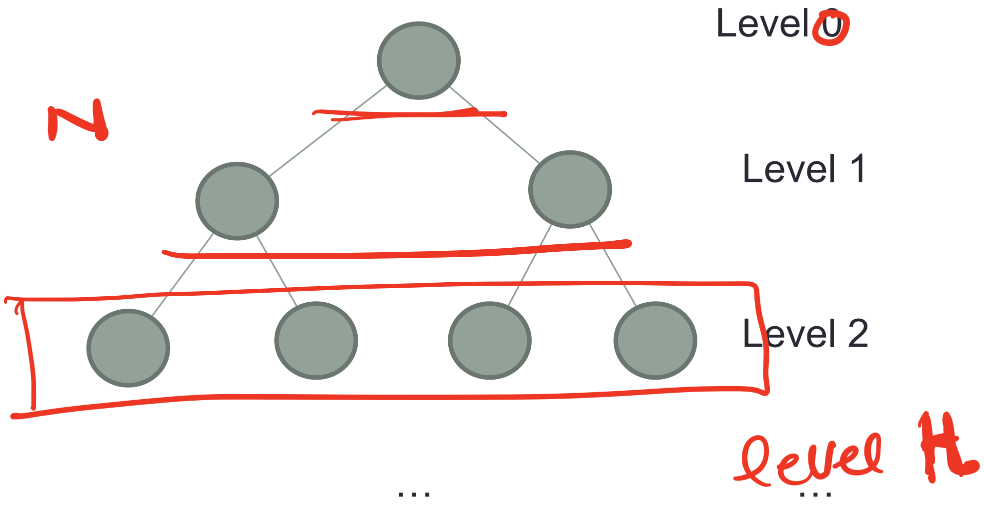
$2^0 + 2^1 + 2^2 + \dots 2^H = N$

$= 2^{H+1} - 1 = N$

$2^{H+1} = N+1$

$H = \log_2(N+1) - 1$

$= O(\log N)$



7 ... 3 2 1 0

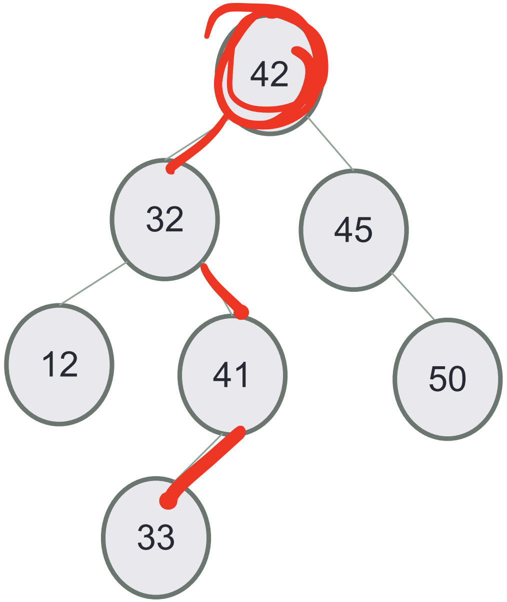
1	1	1	1	1

2^8 $2^7 \dots + 2^2 + 2^1 + 2^0$

1 0 0 0 0 0 0 $= 2^8 - 1$

BST search - best case

Height of the tree



Given a BST with N nodes, in the best case, which key would be searching for?

[✓ A.] root node (e.g. 42)

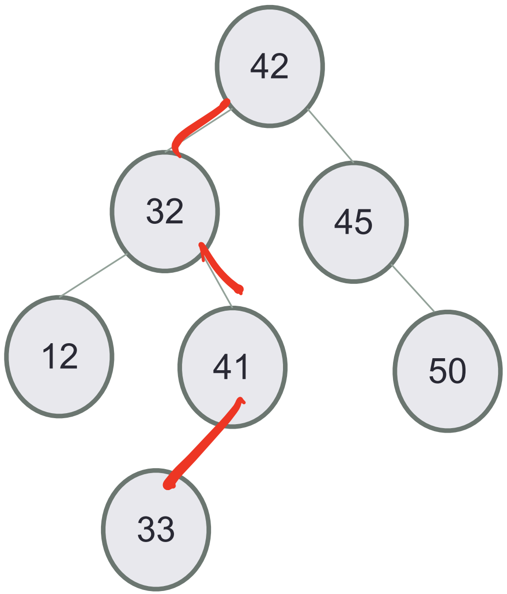
O(1)

B. any leaf node (e.g. 12 or 33 or 50)

C. leaf node that is on the longest path from the root (e.g. 33)

D. any key, there is no best or worst case

BST search - worst case



Given a BST with N nodes, in the worst case, which key would be searching for?

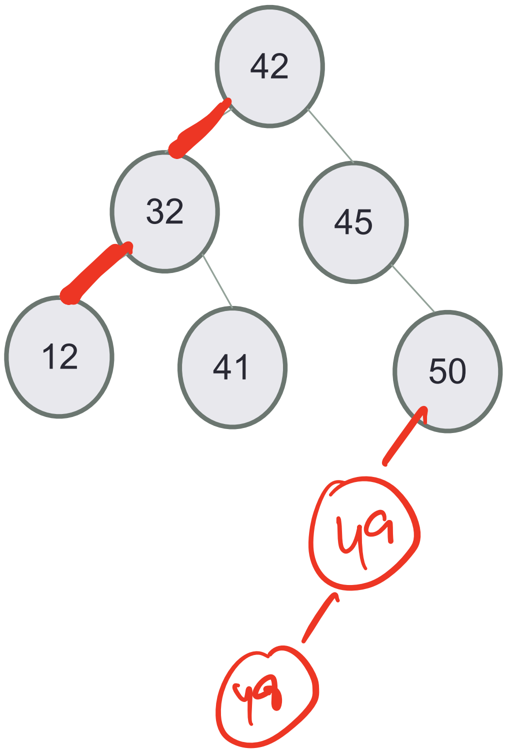
A. root node (e.g. 42)

B. leaf node (e.g. 12 or 41 or 50)

[✓ C.] leaf node that is on the longest path from the root (e.g. 33)

D. a key that doesn't exist in the tree

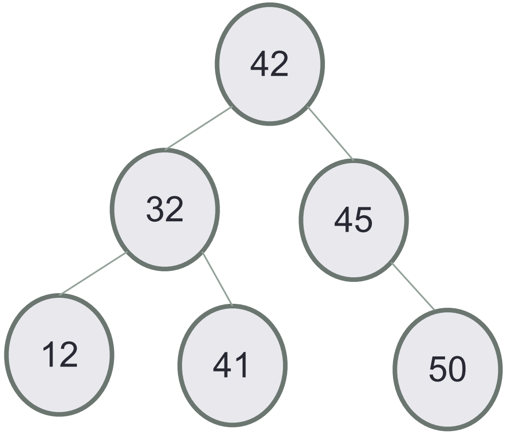
Worst case Big-O of search, insert, min, max



Given a BST of height H with N nodes, what is the running time complexity of searching for a key (in the worst case)?

A. O(1)
B. O(log H)
[✓ C.] O(H)
D. O(H*log H)
E. O(N)

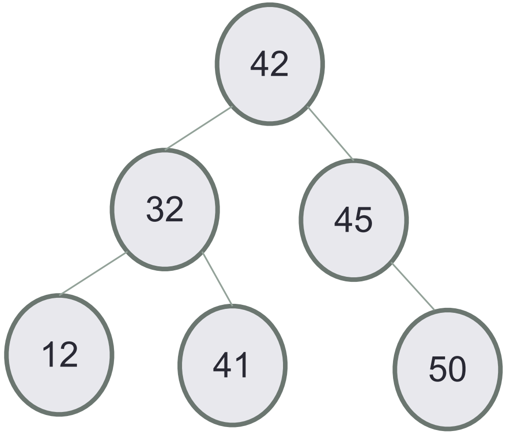
BST operations (worst case)



Given a BST of height H and N nodes, which of the following operations has a complexity of O(H)?

A. min or max
B. insert
C. predecessor or successor
D. delete
E. All of the above

Big O of traversals



In Order: O(N)

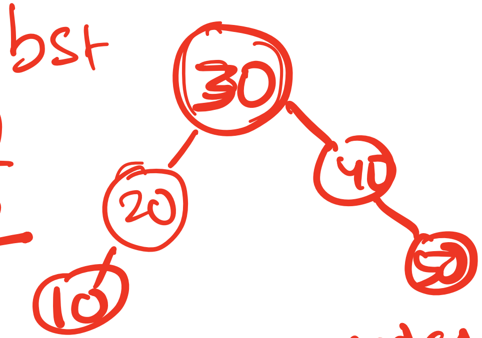
Pre Order: O(N)

Post Order: O(N)

```
procedure Convert(L: sorted linked list with n elements)
    Initialize an empty bst
    while (L is not empty)
        mid:= middle element of L
        remove mid from L
        insert mid to bst
return bst
```

n times

$\frac{n}{2}$



$n \cdot O(n) = O(n^2)$ find mid element of linked list n nodes $= O(n)$

Does the algorithm return a balanced BST for an input (sorted singly linked list) with n keys?

A. Yes

[✓ B. No] because: height of bst $= \frac{n}{2} = O(n)$

First iteration: $O(n) + O(1) + O(1)$

last iteration: $O(1) + O(1) + O(n)$

```
procedure Convert(L: sorted linked list with n elements)
    Initialize an empty bst
    while (L is not empty)
        mid:= middle element of L
        remove mid from L
        insert mid to bst
    return bst
```

$O(1)$

Loop runs n times

upper bound by $O(n)$

$O(1)$

upper bound by $O(\frac{n}{2})$

$= O(n)$

$T(n) = O(1) + n \cdot (O(n) + O(1) + O(n)) = O(n^2)$

What is the Big-O running time complexity of Convert?

The key point to recognize is that the running time of the first & last statements in the while loop vary depending on the iteration number however, bso analysis allows us to upperbound the true running time.

In general, the running time of each statement is:

(1) finding mid element in a LinkedList with m keys : $O(m)$

(2) removing the mid value of linked list (after locating it) is : $O(1)$

(3) Insert value into bst. with k keys is $O(\text{height of bst}) = O(\frac{k}{2}) = O(k)$
(this is because in this specific case height of bst = # of nodes/2

Since the linked list and bst have no more than n keys & the loop runs n times, the overall running time

$$T(n) = O(1) + n \cdot (O(n) + O(1) + O(n))$$

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

```
void foo(int M, int N){
   int i = M;
   while (i >= 1) {
      i = i / 2;
   }
   for (int k = N ; k >= 0; k--){
    for (int j = 1; j < N; j = 2*j){
         cout << "Hello" << endl;
     }
   }
}
```

int i = M;] O(1)

while (i >= 1) → loop runs M/2 times

i = i / 2;] O(1)

for (int k = N ; k >= 0; k--) → loop runs N times

for (int j = 1; j < N; j = 2*j) → loop runs logN times

cout << "Hello" << endl; → O(1)

What is the Big-O running time of foo?

Express in terms of M & N

$$T(n) = O(1) + \frac{M}{2} O(1) + N \cdot \log N \cdot O(1)$$

$$= O(M + N \log N)$$

Balanced trees

- Balanced trees by definition have a height of O(log n)
- A completely filled tree is one example of a balanced tree
- Other Balanced BSTs include AVL trees, red black trees and so on
- Visualize operations on an AVL tree: https://visualgo.net/bn/bst