

# RUNNING TIME ANALYSIS

---

Problem Solving with Computers-II

C++

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

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



# Performance questions

- How efficient is a particular algorithm?
  - **CPU time usage (Running time complexity)**
  - Memory usage
  - Disk usage
  - Network usage
- Why does this matter?
  - Computers are getting faster, so is this really important?
  - Data sets are getting larger – does this impact running times?

# How can we measure time efficiency of algorithms?

- One way is to measure the absolute running time
- Pros? Cons?

```
clock_t t;  
t = clock();
```

```
//Code under test
```

```
t = clock() - t;
```

# Which implementation is significantly faster ?

A.

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

B.

```
function F(n) {  
    Create an array fib[1..n]  
    fib[1] = 1  
    fib[2] = 1  
    for i = 3 to n:  
        fib[i] = fib[i-1] + fib[i-2]  
    return fib[n]  
}
```

C. *Both are almost equally fast*

A better question: How does the running time grow as a function of input size

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

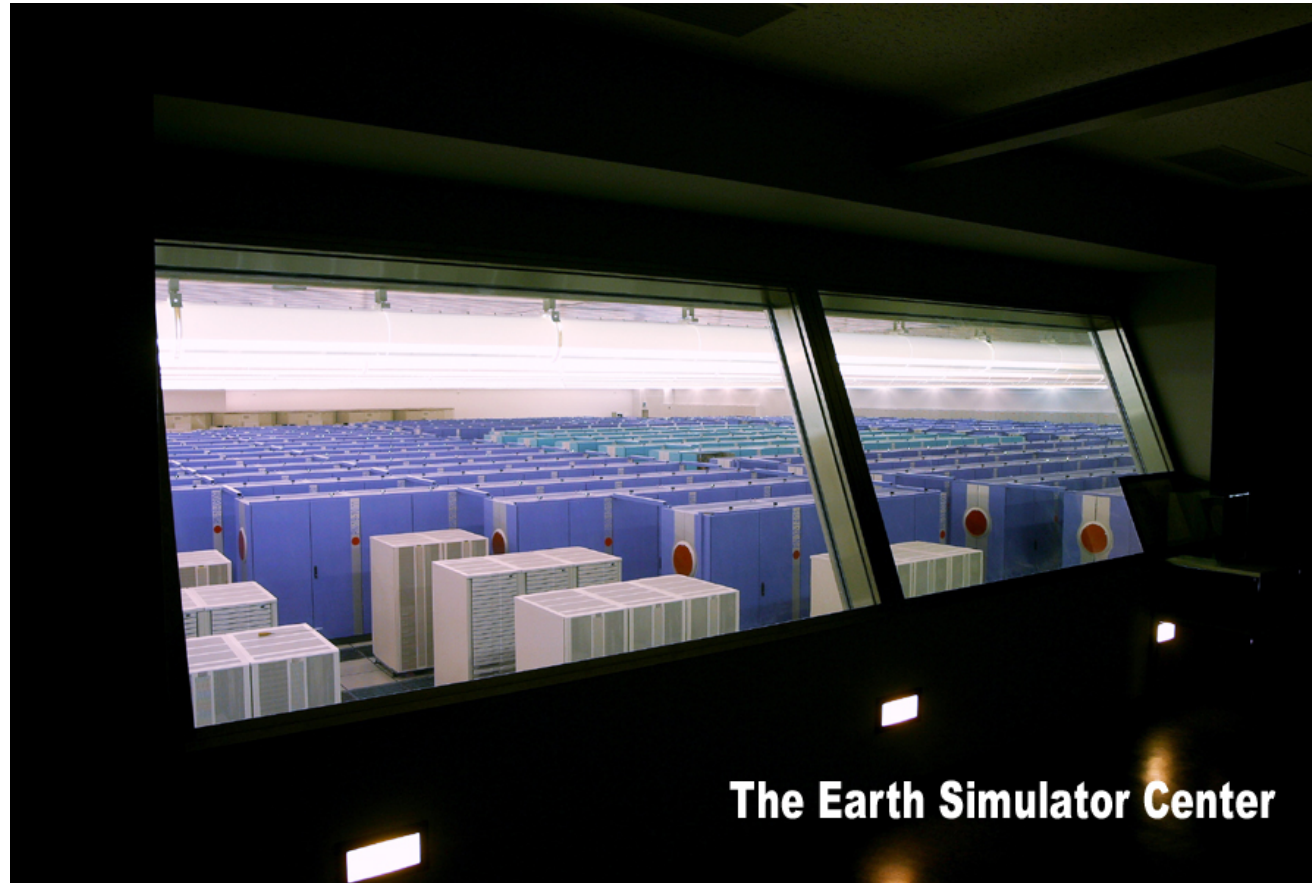
```
function F(n) {  
    Create an array fib[1..n]  
    fib[1] = 1  
    fib[2] = 1  
    for i = 3 to n:  
        fib[i] = fib[i-1] + fib[i-2]  
    return fib[n]  
}
```

The “right” question is: How does the running time grow?

E.g. How long does it take to compute  $F(200)$ ?

....let's say on....

# NEC Earth Simulator



Can perform up to 40 trillion operations per second.

# The running time of the recursive implementation

The Earth simulator needs  $2^{92}$  seconds for  $F_{200}$ .

## Time in seconds

$2^{10}$

$2^{20}$

$2^{30}$

$2^{40}$

$2^{70}$

## Interpretation

17 minutes

12 days

32 years

cave paintings

The big bang!

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

Let's try calculating  $F_{200}$   
using the iterative  
algorithm on my laptop.....

# Goals for measuring time efficiency

- **Focus on the impact of the algorithm:**

Simplify the analysis of running time by ignoring “details” which may be an artifact of the underlying implementation:

- E.g.,  $1000001 \approx 1000000$
- Similarly,  $3n^2 \approx n^2$

- **Focus on trends as input size increases (asymptotic behavior):**

How does the running time of an algorithm increase with the size of the input in the limit (for large input sizes)



# Counting steps (instead of absolute time)

- Every computer can do some primitive operations in constant time:
  - Data movement (assignment)
  - Control statements (branch, function call, return)
  - Arithmetic and logical operations
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm

# Running Time Complexity

Start by counting the primitive operations

```
/* N is the length of the array*/  
int sumArray(int arr[], int N)  
{  
    int result=0;  
    for(int i=0; i < N; i++)  
        result+=arr[i];  
    return result;  
}
```

# Big-O notation

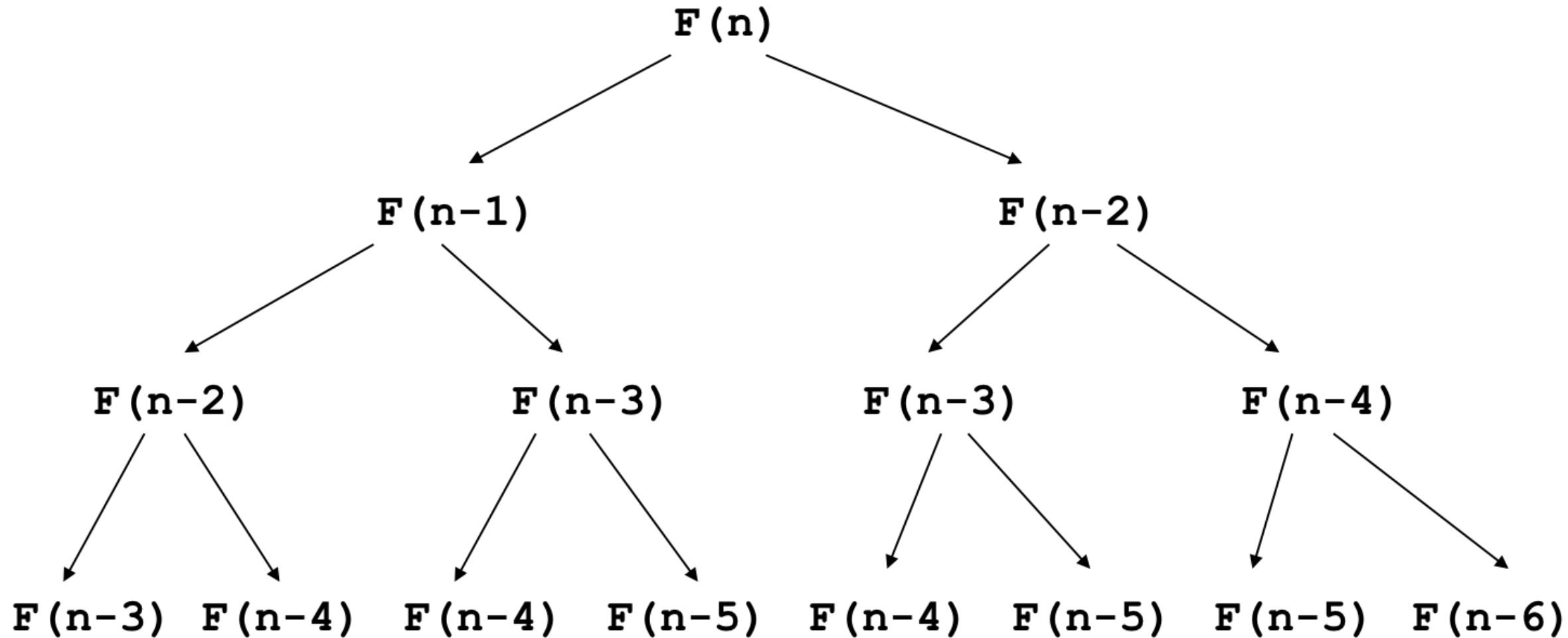
N	Steps = $5*N + 3$
1	8
10	53
1000	5003
100000	500003
10000000	50000003

- Simplification 1: Count steps instead of absolute time
- Simplification 2: Ignore lower order terms
  - Does the constant 3 matter as N gets large?
- Simplification 3: Ignore constant coefficients in the leading term ( $5*N$ ) simplified to N

**After the simplifications,**

**The number of steps grows linearly in N**  
**Running Time =  $O(N)$  pronounced “Big-Oh of N”**

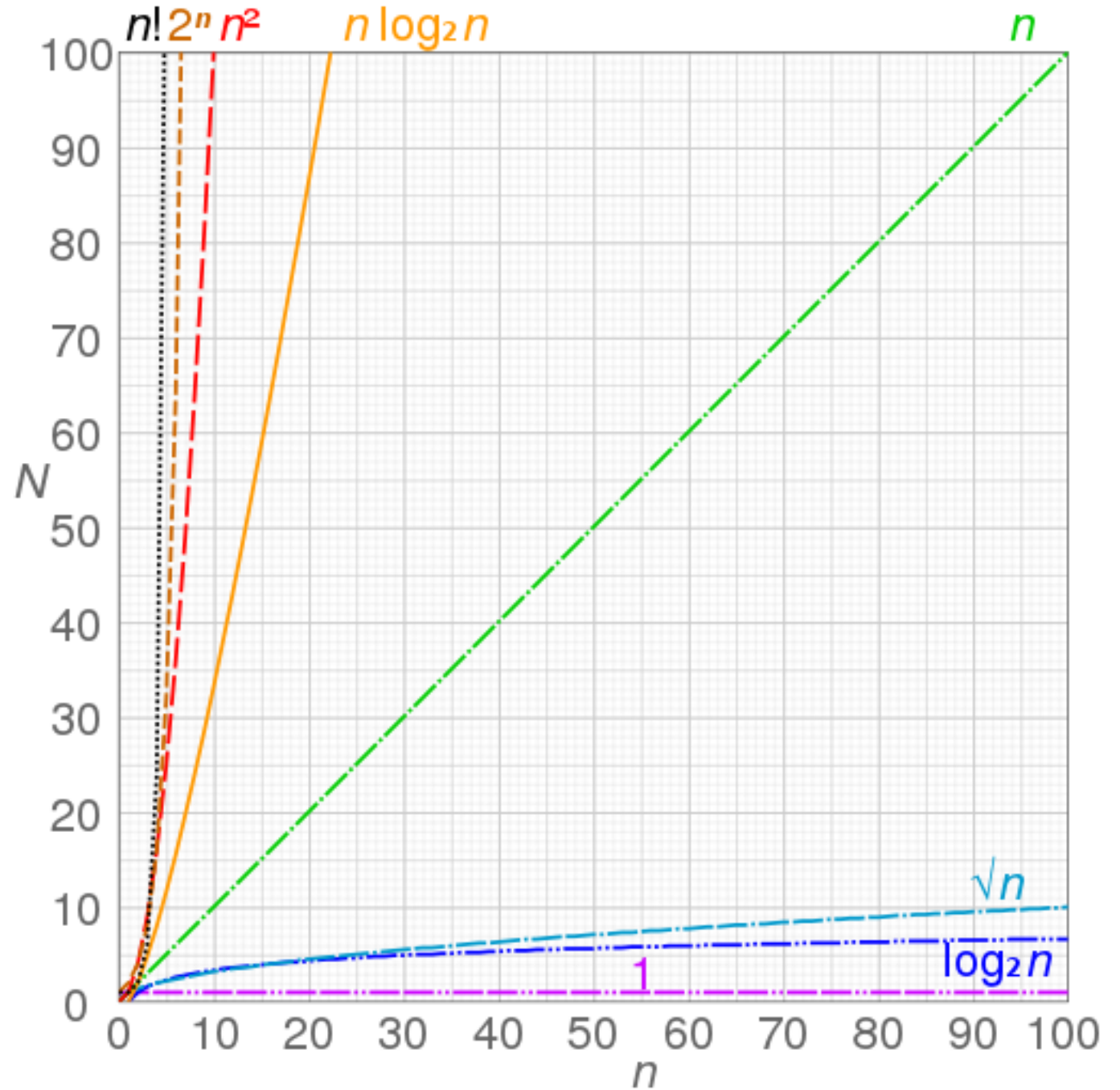
What takes so long? Let's unravel the recursion...



The same subproblems get solved over and over again!

# Orders of growth

- We are interested in how algorithm running time scales with input size
- Big-Oh notation allows us to express that by ignoring the details
- 20n hours v.  $n^2$  microseconds:
  - *which has a higher order of growth?*
  - *Which one is better?*



# Big-O notation lets us focus on the big picture

Recall our goals:

- Focus on the impact of the algorithm
- Focus on asymptotic behavior (running time as  $N$  gets large)

Count the number of steps in your algorithm:  $3 + 5 \cdot N$

Drop the constant additive term :  $5 \cdot N$

Drop the constant multiplicative term :  $N$

**Running time grows linearly with the input size**

Express the count using **O-notation**

**Time complexity =  $O(N)$**

Given the step counts for different algorithms, express the running time complexity using Big-O

1. 10000000

2.  $3*N$

3.  $6*N-2$

4.  $15*N + 44$

5.  $50*N*\log N$

6.  $N^2$

7.  $N^2-6N+9$

8.  $3N^2+4*\log(N)+1000$

**For polynomials, use only leading term, ignore coefficients: linear, quadratic**

# Common sense rules of Big-O

1. Multiplicative constants can be omitted:  $14n^2$  becomes  $n^2$  .
2.  $n^a$  dominates  $n^b$  if  $a > b$ : for instance,  $n^2$  dominates  $n$ .
3. Any exponential dominates any polynomial:  $3^n$  dominates  $n^5$  (it even dominates  $2^n$  ).



## What is the Big O of sumArray2

- A.  $O(N^2)$
- B.  $O(N)$
- C.  $O(N/2)$
- D.  $O(\log N)$
- E. None of the array

```
/* N is the length of the array*/  
int sumArray2(int arr[], int N)  
{  
    int result=0;  
    for(int i=0; i < N; i=i+2)  
        result+=arr[i];  
    return result;  
}
```

## What is the Big O of sumArray2

- A.  $O(N^2)$
- B.  $O(N)$
- C.  $O(N/2)$
- D.  $O(\log N)$
- E. None of the array

```
/* N is the length of the array*/  
int sumArray2(int arr[], int N)  
{  
    int result=0;  
    for(int i=1; i < N; i=i*2)  
        result+=arr[i];  
    return result;  
}
```

# Operations on sorted arrays

- Min :
- Max:
- Median:
- Successor:
- Predecessor:
- Search:
- Insert :
- Delete:

[illegible]

# Next time

- Running time analysis of Binary Search Trees

References:

<https://cseweb.ucsd.edu/classes/wi10/cse91/resources/algorithms.ppt>

<http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf>