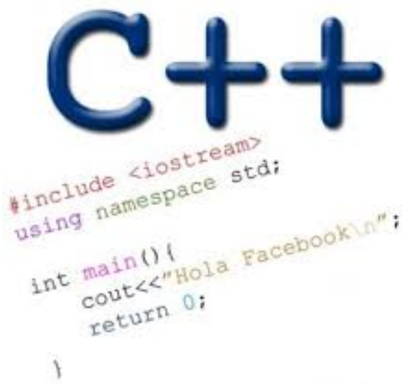
# RUNNING TIME ANALYSIS

Problem Solving with Computers-II



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

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

GitHub

## Performance questions

- How efficient is a particular algorithm?
  - **CPU time usage (Running time complexity)**
  - **Memory usage (Space complexity)**
  - 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.

```
double Fib(int n){
   if(n <= 2) return 1;
   return Fib(n-1) + Fib(n-2);
}
```

B.

```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i-1] + fib[i-2];
    }
    return fib[n-1];
}
```

C. *Both are almost equally fast*

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

```
double Fib(int n){
   if(n <= 2) return 1;
   return Fib(n-1) + Fib(n-2);
}
```

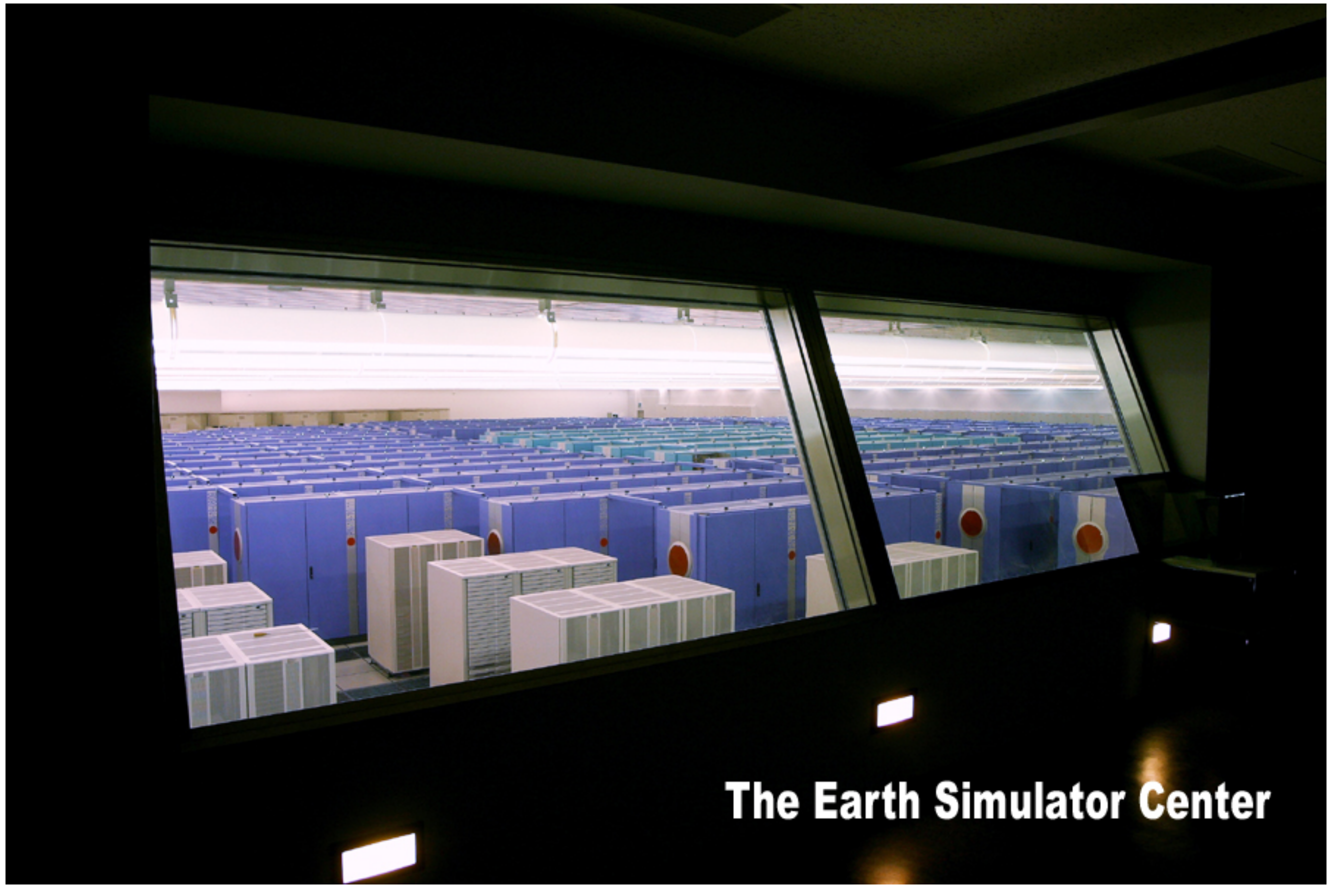
```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i-1] + fib[i-2];
    }
    return fib[n-1];
}
```

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

E.g. How long does it take to compute Fib(200) recursively?

….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 | Interpretation |
|---|---|
| $2^{10}$ | 17 minutes |
| $2^{20}$ | 12 days |
| $2^{30}$ | 32 years |
| $2^{40}$ | cave paintings |
| $2^{70}$ | The big bang! |

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

**Subgoal 1: 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

**Subgoal 1: 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

- **Subgoal 2: Focus on trends as input size increases (asymptotic behavior):**

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

## Subgoal 1: Focus on the impact of the algorithm:

Count operations 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

```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i-1] + fib[i-2];
    }
    return fib[n-1];
}
```

- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm

## Subgoal 1: Focus on the impact of the algorithm:

Count operations instead of absolute time!

```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i-1] + fib[i-2];
    }
    return fib[n-1];
}
```

## Subgoal 1: Focus on the impact of the algorithm:

Count operations instead of absolute time!

```
procedure Fib(n: positive integer)
 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]
```

```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i–1] + fib[i–2];
    }
    return fib[n–1];
}
```

Can we count number of operations on the pseudo code version of the Fib function?

A. Yes

B. No

Our first goal for analyzing runtime was to focus  on the impact of the algorithm. What was our second subgoal?

A. Focus on optimizing the algorithm so that it can be efficient

B. Focus on measuring the time it takes to run the algorithm by time stamping our code.

C. Focus on trends as input size increases (asymptotic behavior)

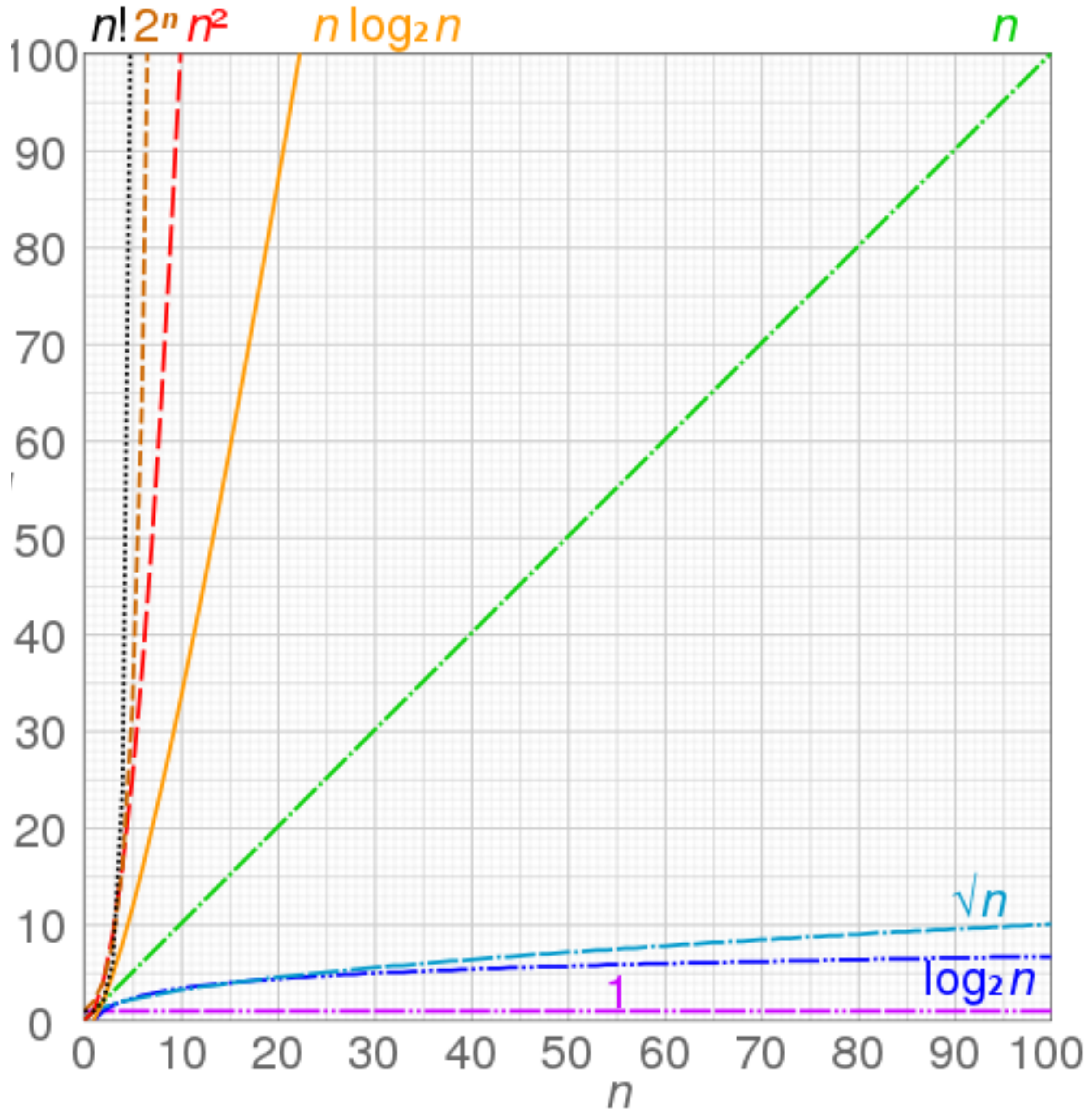
# Orders of growth

An **order of growth** is a set of functions whose asymptotic growth behavior is considered equivalent. For example, $2n$, $100n$ and $n+1$ belong to the same order of growth

Which of the following functions has a higher order of growth?

A. 50n

B. $2n^2$


$n!$
$2^n$
$n^2$
$n \log_2 n$
$n$
$\sqrt{n}$
$\log_2 n$
$n$