RUNNING TIME ANALYSIS

Problem Solving with Computers-II

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)
}
```

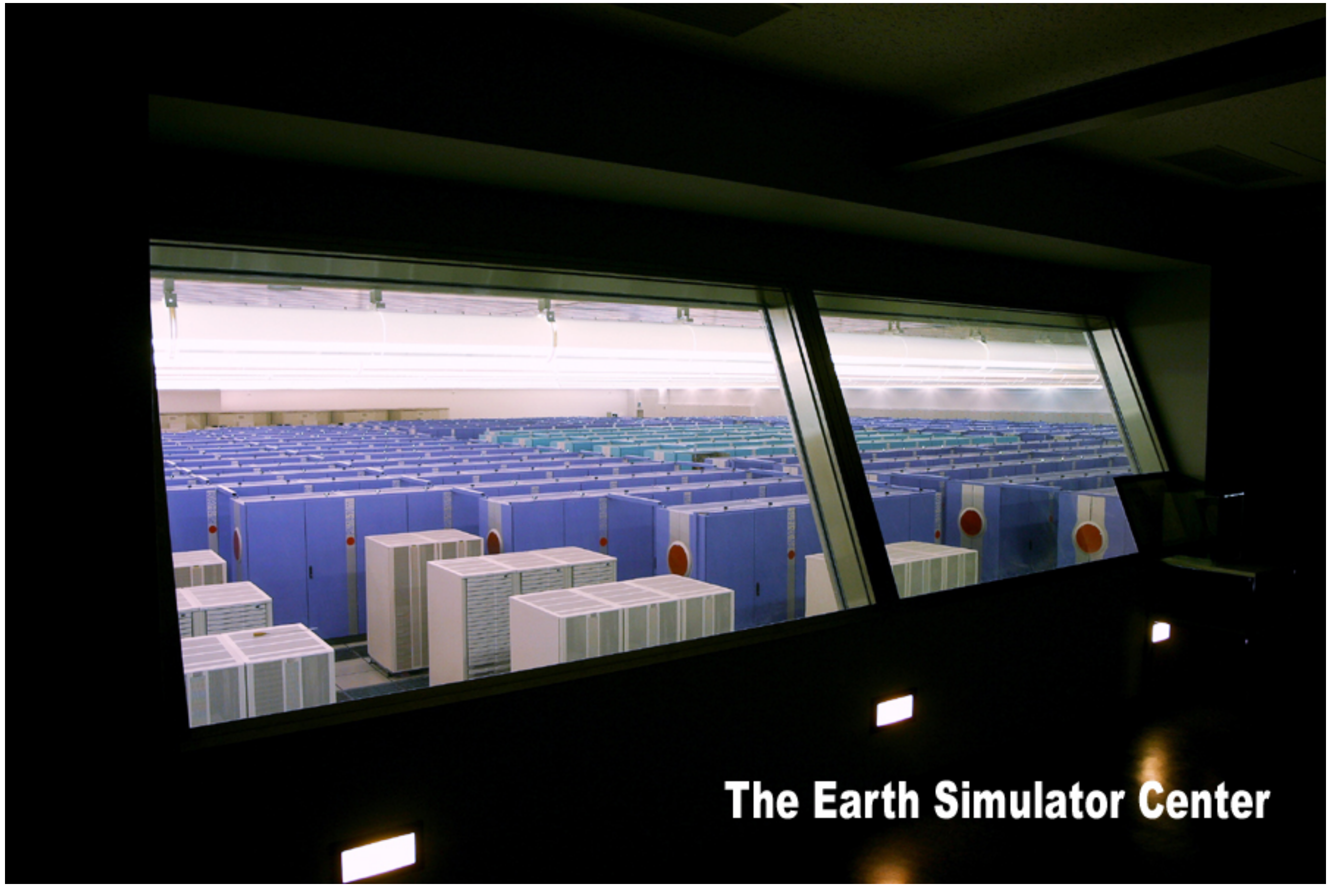
```
function F(n){
 Create an array fib[1..n]
 fib[1] = 1
 fib[2] = 1
 for i = 3 to n:
    fib[i] = fib[i-1] + fib[i-2]
 return fib[n]
}
```

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	Interpretation
2^{10}	17 minutes
2^{20}	12 days
2^{30}	32 years
2^{40}	cave paintings
2^{70}	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

- **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

Goals for measuring time efficiency

- **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)

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

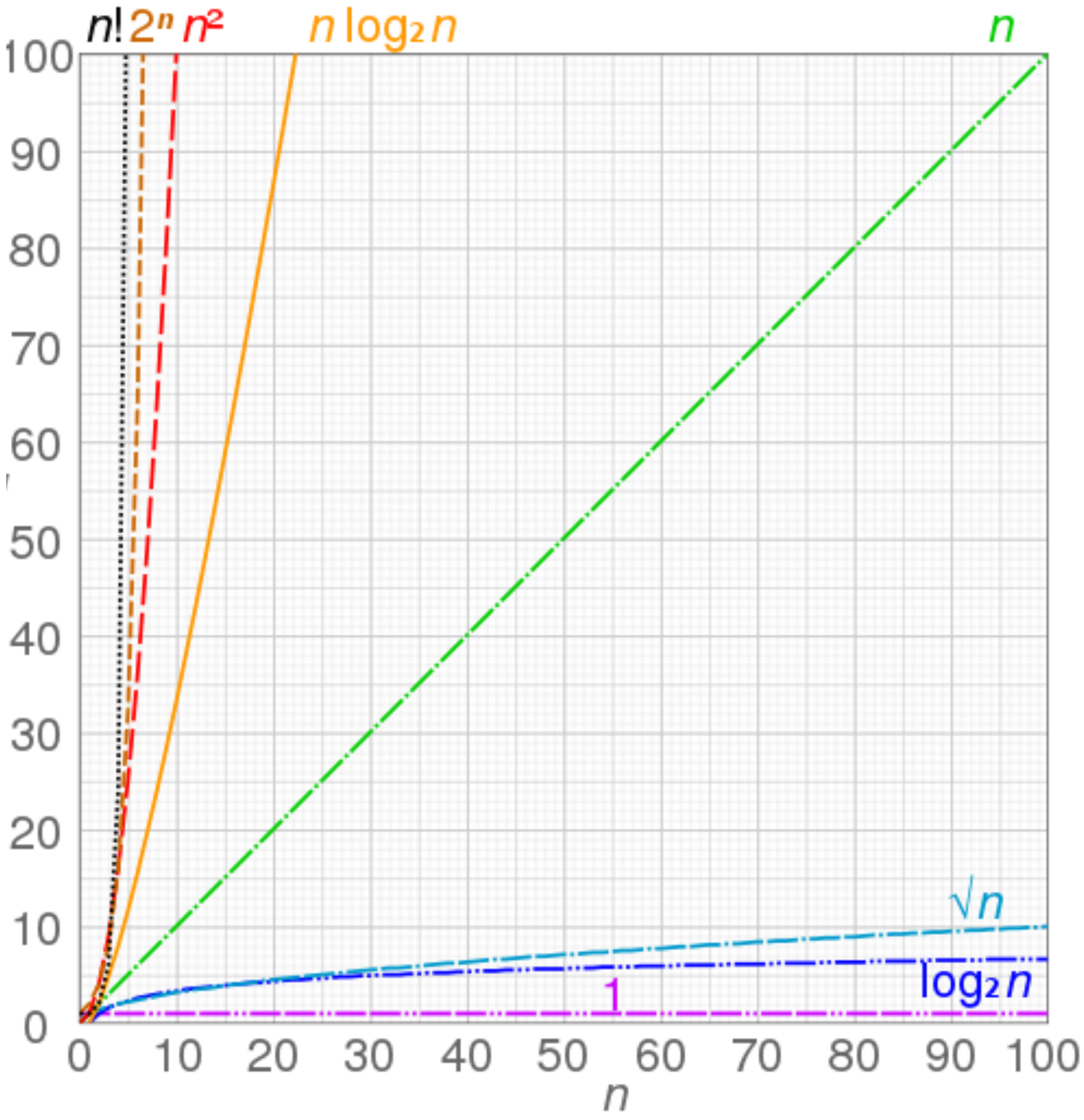
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm

Counting the number of primitive steps

```
/* 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;
}
```

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



Order of growth

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

A. 50n

B. $2n^2$

Big-O notation

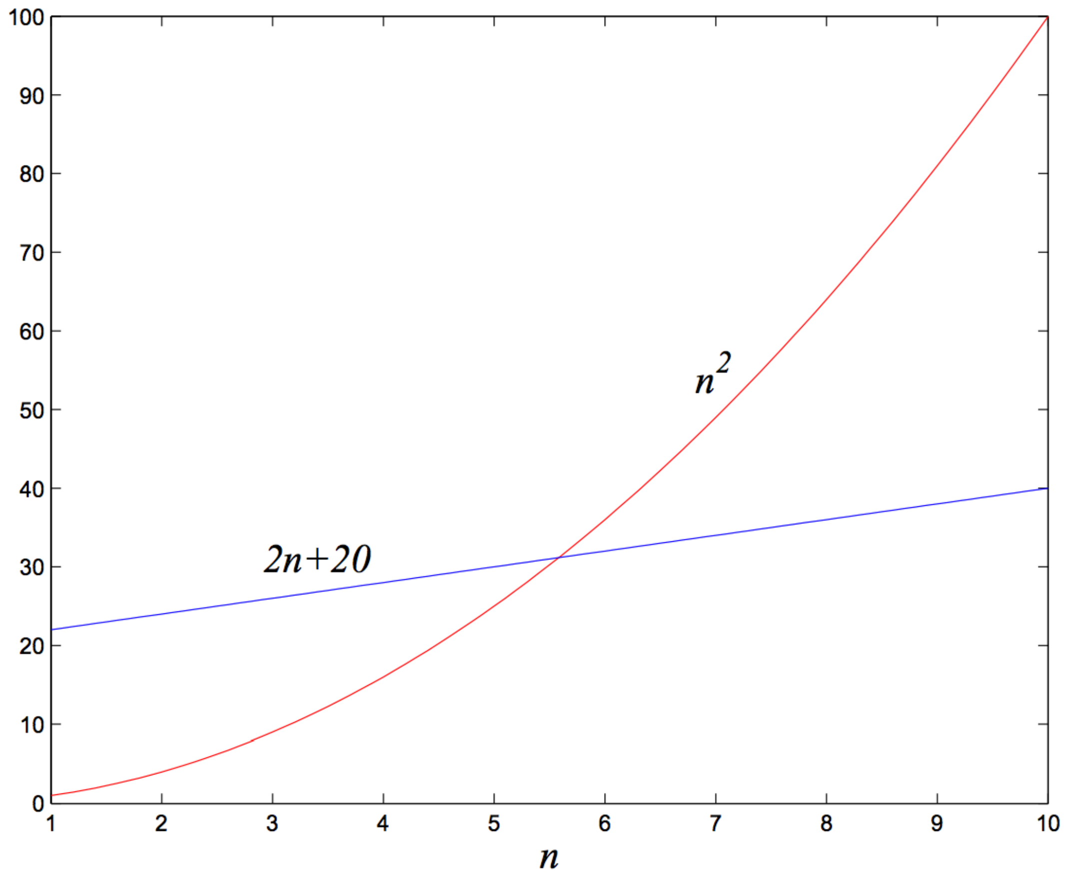
- Big-O notation provides an upper bound on the order of growth of a function

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 $f(n) \le c \cdot g(n)$ for all $n >= k$.

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



What is the Big-O running time of sumArray?

```
/* 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;
}
```

Expressing the running time of sumArray using Big-O notation

N	Steps = 4*n +3
1	7
10	43
1000	4003
100000	400003
10000000	40000003

- 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 (4n) simplified to n

After the simplifications,

The number of steps grows linearly in n
Running Time = O(n) pronounced "Big-Oh of n"

Big-O notation lets us focus on the big picture

Recall our goals:

- **Focus on the impact of the algorithm**
- **Focus on asymptotic behavior (as n gets large)**

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).

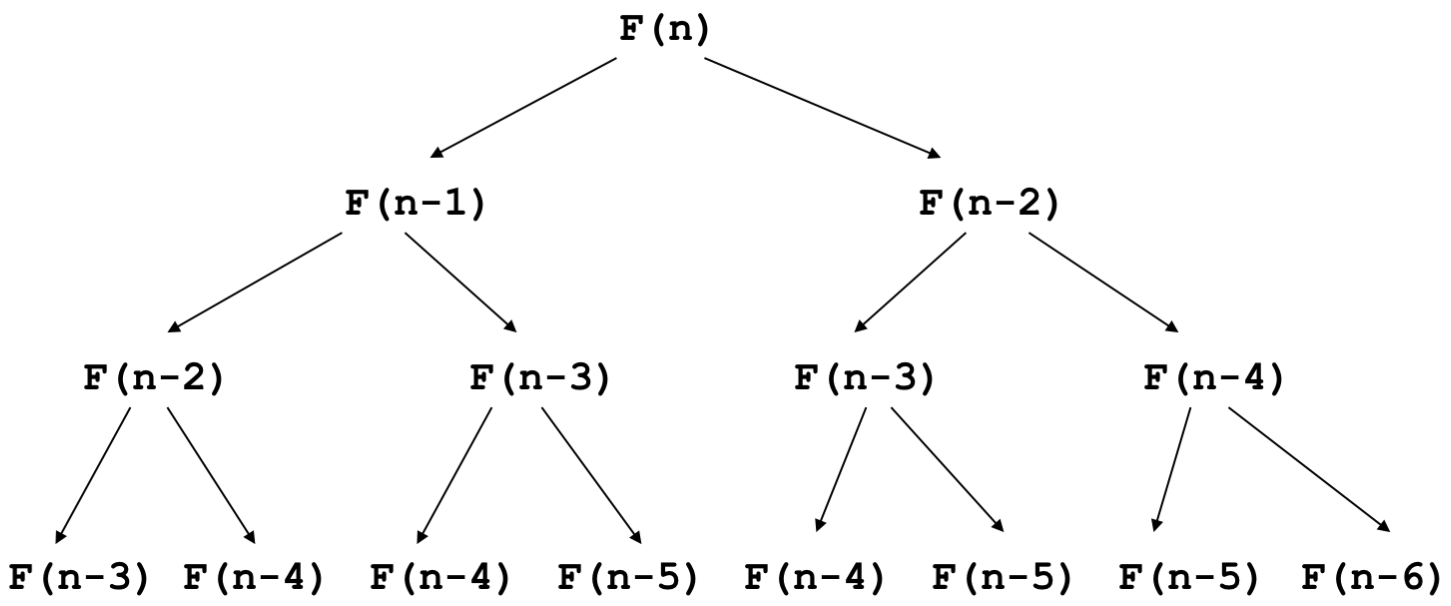
Big-O analysis

```
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]
}
```

Big-O analysis

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

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



The same subproblems get solved over and over again!

What is the Big O running time 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;
}
```

Next time

- Running time analysis : best case and worst case
- 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