RUNNING TIME ANALYSIS

Problem Solving with Computers-II

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*

Fib(n): 1 1 2 3 5 8 . . . .

n 1 2 3 4 5 6 7 . . .

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);
}
```

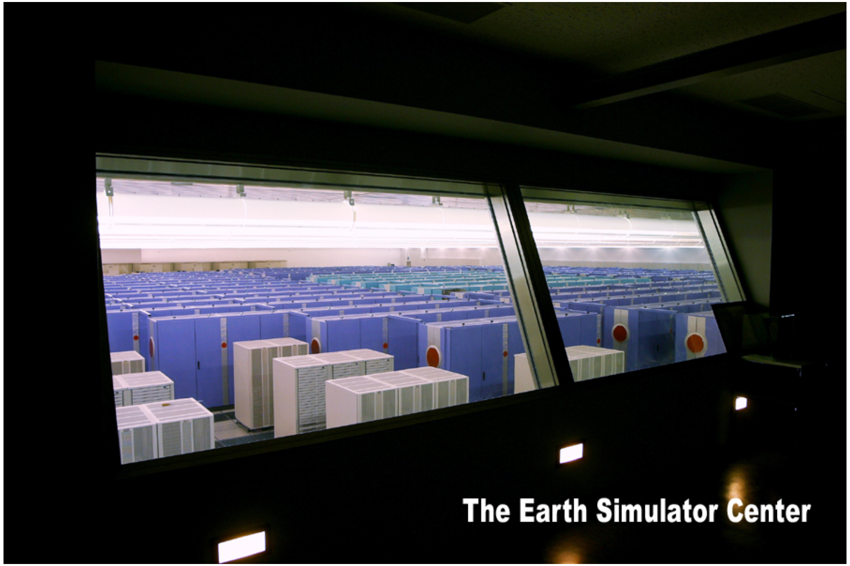
```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i-1] + fib[i-2];
    }
    return fib[n-1];
}
```

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!

$$T(n) = 1 + 2 + 2 + 6 \cdot (n-2)$$
$$= 6n + 5 - 12$$
$$= 6n - 7$$

```
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];
}
```

Loop runs $(n-2)$

Loop

Initialize $i = 2$ — 1

At end $i < n$ — 1

Each iteration, do the following: — 6

$i < n$ — 1

Body of loop $fib[i] = fib[i-1] + fib[i-2]$ — 4

$i{+}{+}$ — 1

1 1 1 1

Subgoal 1: Focus on the impact of the algorithm:

Count operations instead of absolute time!

$T(n) = 3 + (n-2) \cdot 4 = 4n - 3$

$T(n) = 6 \cdot n - 7$

```
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)

$$T(n) = 6n - 7$$

$$= \underbrace{6n^2 + 10\, n \log n + 50}$$

$$= O(n^2)$$

we pick the fastest growing term

Orders of growth

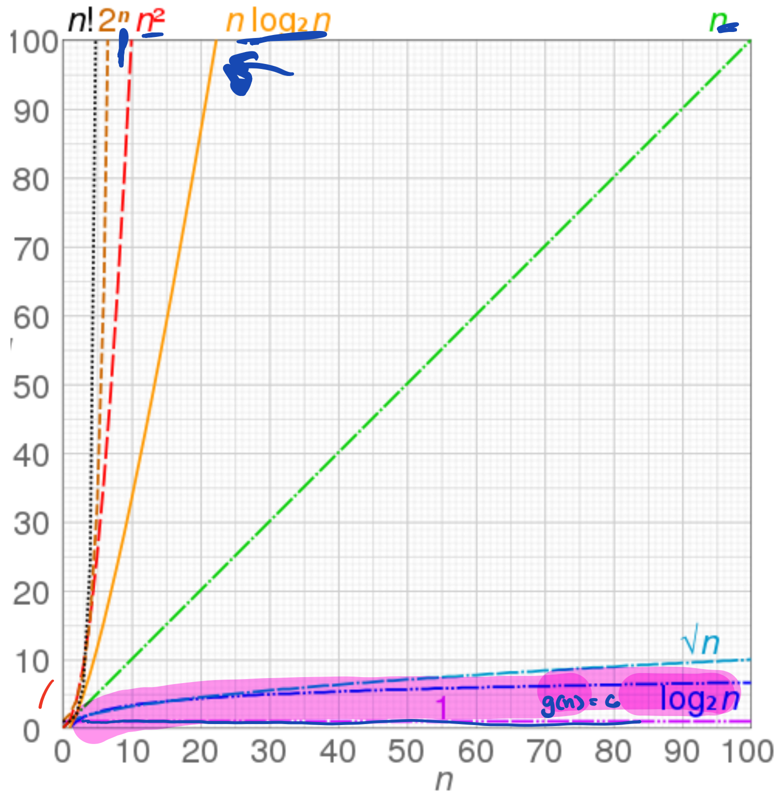
An **order of growth** is a set of functions whose asymptotic growth behavior is considered equivalent. For example, 2*n*, 100*n* and *n*+1 belong to the same order of growth

unit less

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

A. 50n

B. $2n^2$



Big-O notation

$$T(n) = O(g(n))$$

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

$$T(n) = 6n^2 + 10n\log n + 50$$

$$< 6n^2 + 10n^2 + 50$$

$$= 16n^2 + 50$$

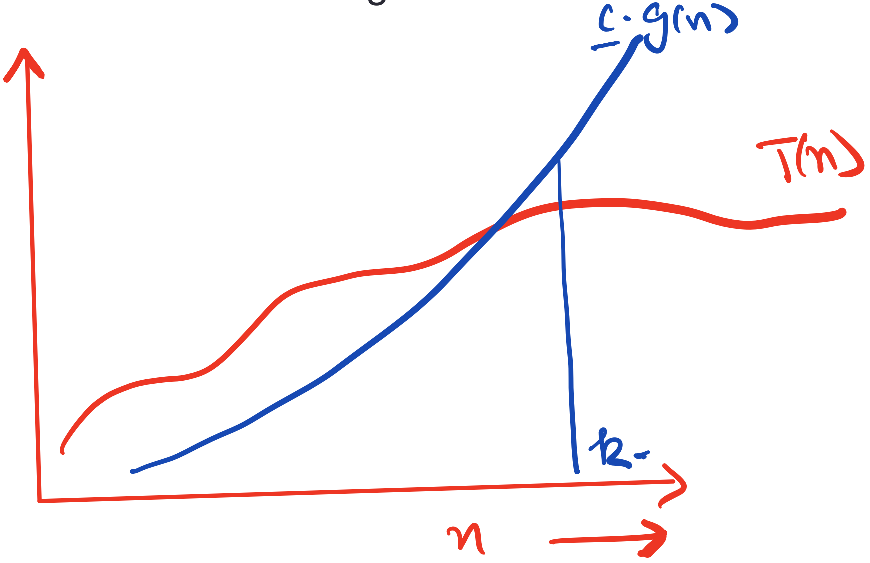
$$< 16n^2 + n^2, \quad n \geq 8$$

$$= 17n^2, \quad n \geq 8$$

$$= O(n^2)$$

$c = 17$

$k = 8$



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

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

$T(n) = 6n^2 + 10n \log n + 50$

$= O(n^2)$

$= O(n^3)$ Yes, but in practice not okay!

What is the Big O running time of Fib?

```
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]
```

$T(n) = 6n - 7$

$= O(n)$

Express in Big-O notation

1. 10000000 = $O(1)$
2. $3*n$ = $O(n)$
3. $6*n-2$ = $O(n)$
4. $15*n + 44$ = $O(n)$
5. $50*n*\log(n)$ = $O(n \log n)$
6. n^2 = $O(n^2)$
7. n^2-6n+9 = $O(n^2)$
8. $3n^2+4*\log(n)+1000$ = $O(n^2)$
9. $3^n + n^3 + \log(3*n)$ = $O(3^n)$

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 notation lets us focus on the big picture

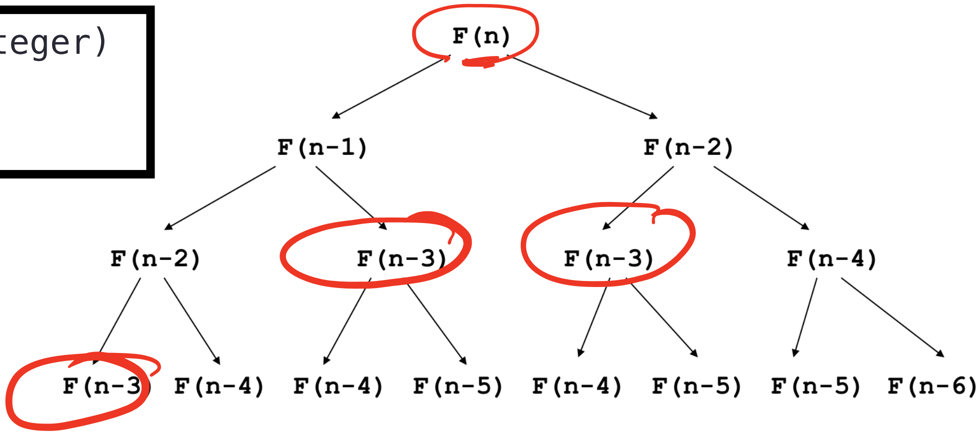
Recall our goals:

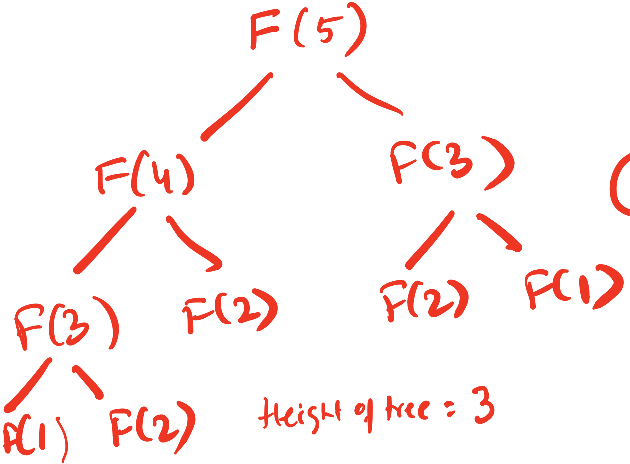
- **Focus on the impact of the algorithm**
- **Focus on asymptotic behavior (as n gets large)**

Big-O analysis

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

```
procedure F(n: a positive integer)
    if(n <= 2) return 1
    return F(n−1) + F(n−2)
```





Height of tree = 3

The same subproblems get solved over and over again!

The number of function calls is bounded by the number of nodes in a binary tree of height (n-2), which is $2^{n-2+1}-1 = 2^{n-1}-1$ (derived in the next lecture)

Running time = $O(2^n)$ ← exponential!!

Another approach to deriving the Big-O of running time

$T(n)$: Running time of $F(n)$

We have the following recurrence relation

$T(2) = T(1) = 1$

$T(n) = T(n-1) + T(n-2) + c$ c is some constant

$\leq 2\,T(n-1) + c$ ($T(n-1) > T(n-2)$, and we can make this approximation in Big-O analysis)

$\leq 2\,(2\,T(n-2) + c) + c$ (Substitute for $T(n-1)$)

$= 2^2\,T(n-2) + 3c$

Repeating this process we get

$T(n) \leq 2^k\,T(n-k) + (2^k - 1)c$

Base case $n-k = 1$

$\Rightarrow k = n-1$

Substitute for k to get

$T(n) \leq 2^{n-1}\,T(1) + (2^{n-1} - 1)c$

$= 2^{n-1}\cdot 1 + 2^{n-1}\cdot c - c$

$= 2^{n-1}(1+c) - c$

$= O(2^n)$ (same result as before!)

)

```
procedure max(a1,a2, … an: integers)
 max:= a1
 for i:= 2 to n
     if max < ai
         max:= x
return max{max is the greatest element}
```

$(n-2)\cdot c + 1$

What is the Big-O running time of max?

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

What is the Big O running time of sum()?

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 sum(int arr[], int n)
{
      int result=0;
      for(int i=0; i < n; i+=2)
            result+=arr[i];
      return result;
}
```

What is the Big O running time of sum()?

A. $O(n^2)$
B. O(n)
C. O(n/2)
D. O(log n)
E. None of the ~~array~~ above

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

$2^{k-1} \geq n$

$k-1 \geq \log_2(n)$

$k \geq \log_2(n)+1$

# iteration	i
1	1
2	2
3	4
4	8

What is the Big O running time of sum()?

k 2^{k-1}

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 sum(int arr[], int n)
{
    int result = 0;
    for(int i=0; i < n; i=i+2)
            result+=arr[i];
    for(int i=1; i < n; i=i*2)
            result+=2*arr[i];
    return result;
}
```

$O(1)$

$O(n)$

$O(\log n)$

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

$= O(n)$

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