RUNNING TIME ANALYSIS

Problem Solving with Computers-II

Problem: Fibonacci Numbers

<u>Definition:</u>

The Fibonacci numbers are the sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,…

Defined by

$F_0 = F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

<u>Problem:</u> Given n, compute F_n.

Slide Credit: Prof. Daniel Kane, UCSD

Which implementation is significantly faster ?

A.

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

B.

```
F(int n){
    Initialize fib[0 . . . n]
    fib[0] = fib[1] = 1

    for i = 2 : n
      fib[i] = fib[i-1] + fib[i-2]

    return fib[n]
}
```

C. *Both are almost equally fast*

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

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

….let’s say on….a supercomputer that can compute 40 trillion operations per sec

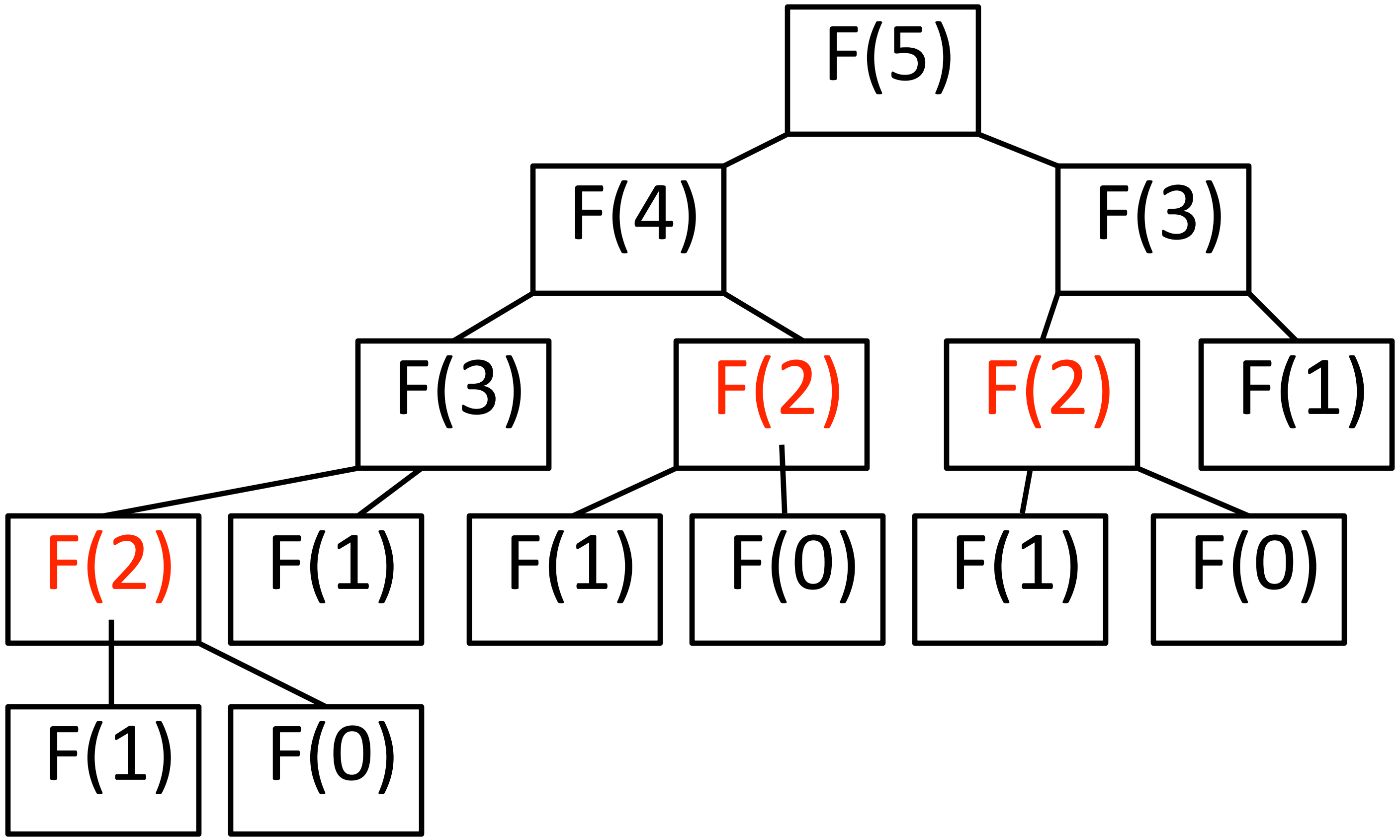
How long does it take to compute Fib(200) recursively?
….let's say on…. a supercomputer that runs 40 trillion operations per second

It will take approximately 2^{92} seconds to compute F_{200}.

Time in seconds	Interpretation
2^{10}	17 minutes
2^{20}	12 days
2^{30}	32 years
2^{40}	35000 years (cave paintings)
2^{50}	35 million years ago
2^{70}	Big Bang

Why So Slow?

Too many recursive calls.



Credit: Prof. Daniel Kane, UCSD

Improved Algorithm

Lets compute T(n) = number of lines of code `Fib`(n) needs to execute.

```
F(int n){
    Initialize fib[0 . . . n]
    fib[0] = fib[1] = 1

    for i = 2 : n
     fib[i] = fib[i-1] + fib[i-2]

    return fib[n]
}
```

2 lines

2(n-1) lines

1 line

T(n) = 2n+1

Credit: Prof. Daniel Kane, UCSD

Question: Runtime

Is $T(n) = 2n+1$ an accurate description of this algorithm? A. Yes. B. No

```
F(int n){
    Initialize fib[0 . . . n]
    fib[0] = fib[1] = 1

    for i = 2 : n
     fib[i] = fib[i−1] + fib[i−2]

    return fib[n]
}
```

2 lines

2(n-1) lines

1 line

$T(n) = 2n+1$

Credit: Prof. Daniel Kane, UCSD

Bottom Line

What we really care about is how long it takes program to run on a real machine.

Unfortunately, this depends on:

- CPU speed
- Memory architecture
- Compiler optimizations
- Background processes

Too much to consider for every analysis

Credit: Prof. Daniel Kane, UCSD

Run Time Analysis Approach

Goal 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

Run Time Analysis Approach

Goal 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

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

Run Time Analysis Approach

Goal 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

Goal 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)

Run Time Analysis Approach

Goal 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

Count operations instead of absolute time!

Goal 2: Focus on trends as input size increases:

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

Describe asymptotic running time using well known functions

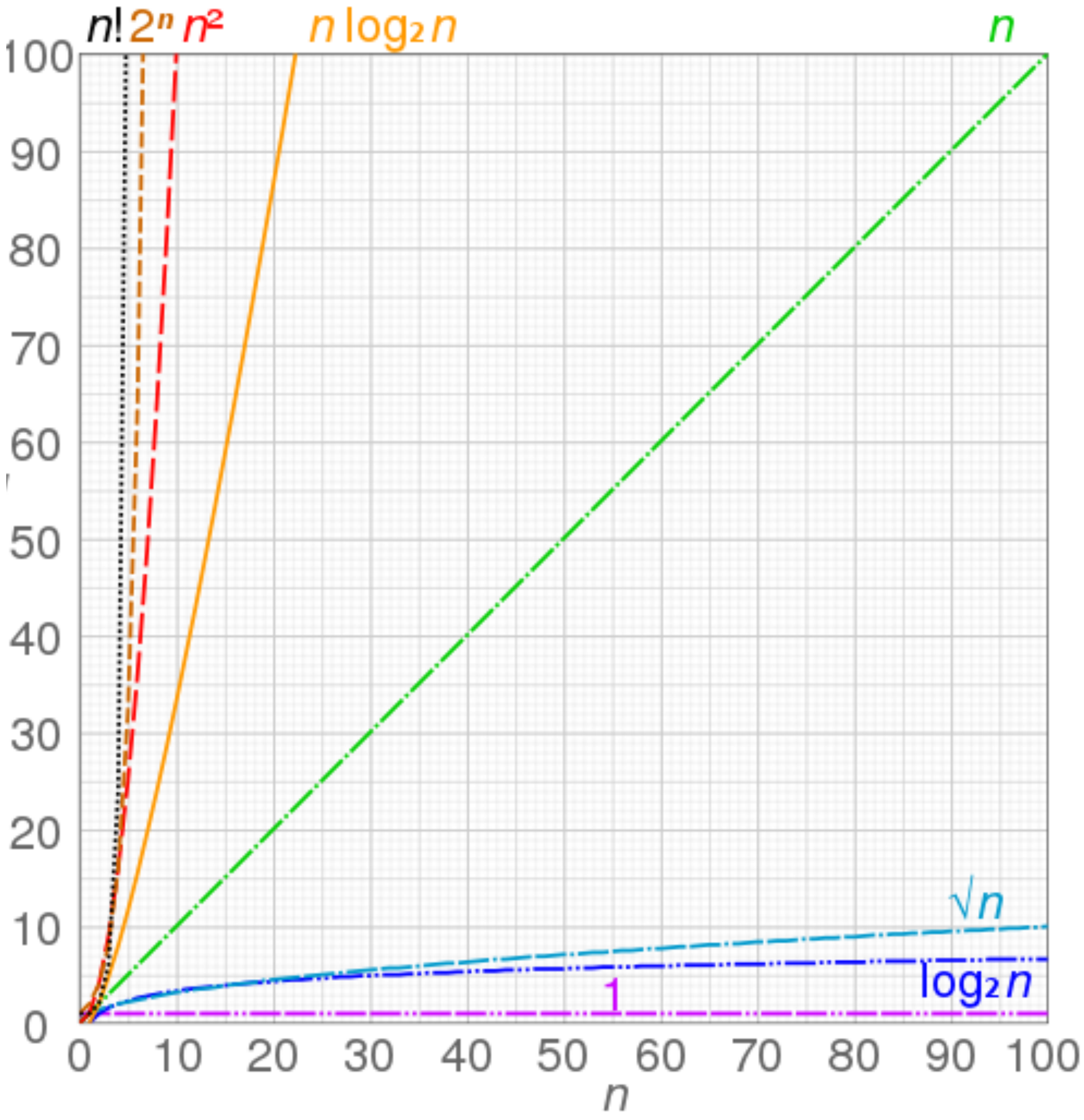
Orders of growth

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. $2n^2$



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

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

Express in Big-O notation

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. $3n^2+4*\log(n)+1000$
9. $3^n + n^3 + \log(3*n)$

Common sense rules

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).

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

```
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}
```

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 above

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

```
/* 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;
}
```

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

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

```
/* 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;
}
```

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

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

```
/* 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;
}
```

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

Next time

- Running time analysis : best case and worst case
- Running time analysis of Binary Search Trees

Credits and references:

Slides by Professors Sanjoy Das Gupta and Daniel Kane at UCSD
http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf