RUNNING TIME ANALYSIS

Problem Solving with Computers-II

GitHub

Problem: Fibonacci Numbers

Definition:

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 $n \geq 2$

Problem: Given n, compute F_n.

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 A[0 . . . n]
    A[0] = A[1] = 1

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

    return A[n]
}
```

C. *Both are almost equally fast*

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 A[0 . . . n]
    A[0] = A[1] = 1

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

    return A[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?

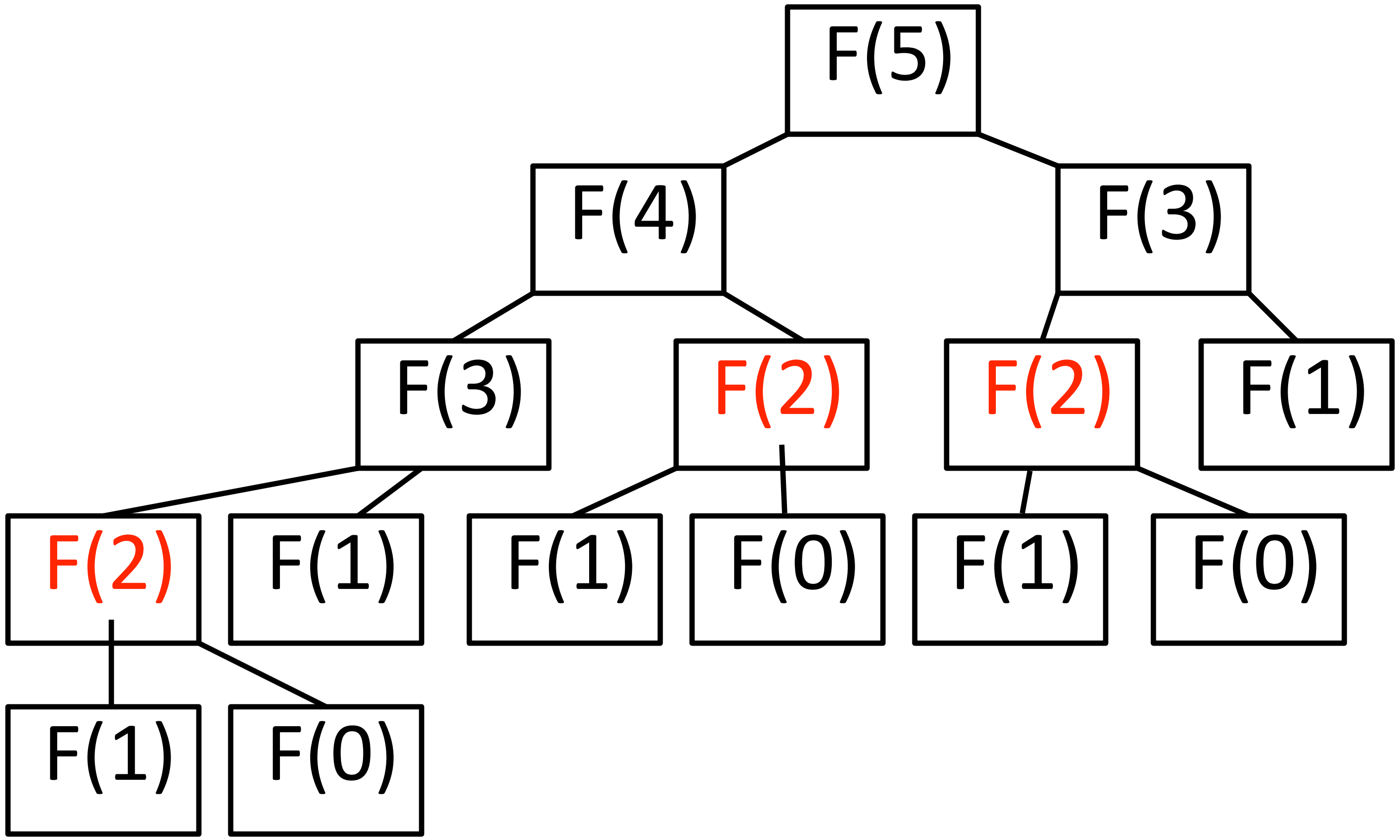
….let's say on…. a supercomputer that runs 40 trillion operations per second

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.



Bottom Line

We want to analyze the **impact of the algorithm on running time,** separate from other hardware dependent artifacts that affect time:

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

Too much to consider for every analysis **if we analyzed absolute time**

Bottom Line

We want to analyze the **impact of the algorithm on running time,** separate from other hardware dependent artifacts that affect time:

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

Too much to consider for every analysis **if we analyzed absolute time**

Count operations instead of absolute time!

Machine model used for analysis

Goal: Count primitive operations instead of absolute time!

- Every computer can do some **primitive operations** in constant time:
  - Data movement (assignment)
  - Data load/store (accessing an element of an array)
  - 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
- The important assumption is that each primitive operation takes a **constant amount of time**

Iterative Fibonacci Algorithm

Lets compute T(n) = **number of primitive operations** to execute **F**(n)

```
F(int n){
    Initialize A[0 . . . n]
    A[0] = A[1] = 1
    for (int i = 2; i <= n ; n++)
     A[i] = A[i−1] + A[i−2]
    return A[n]
}
```

3 ops

1 op 1 op 2 ops

2 ops 2 ops

1 op

T(n) = 9n - 7

Iterative Fibonacci Algorithm

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

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

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

    return A[n]
}
```

2 lines

2(n-1) lines

1 line

T(n) = 2n+1

Effect of constant factors

For the iterative fib, we derived two expressions for the running time

$T(n) = 9n - 7$

$T(n) = 2n + 1$

Discuss: how much do the constant factors matter as n gets large?

- Think about 9n - 7 vs 9n and 2n + 1 vs. 2n
- What about 9n vs 2n?

Analogy: Types of roads and orders of growth

Think of algorithms as **cars traveling a distance.**

- **Running time T(n):** Effort (or fuel) needed to complete the trip
- **Input sizen n**: The distance the car needs to go
- **Order of growth:** Effort needed to drive a car on different types of road: **smooth highway, winding mountain, or congested city street**

$9n$ vs. $2n$

SUV on a highway **Sedan on a highway**

Both cars take a similar level of effort (linear order of growth) when traveling on a highway

Orders of growth

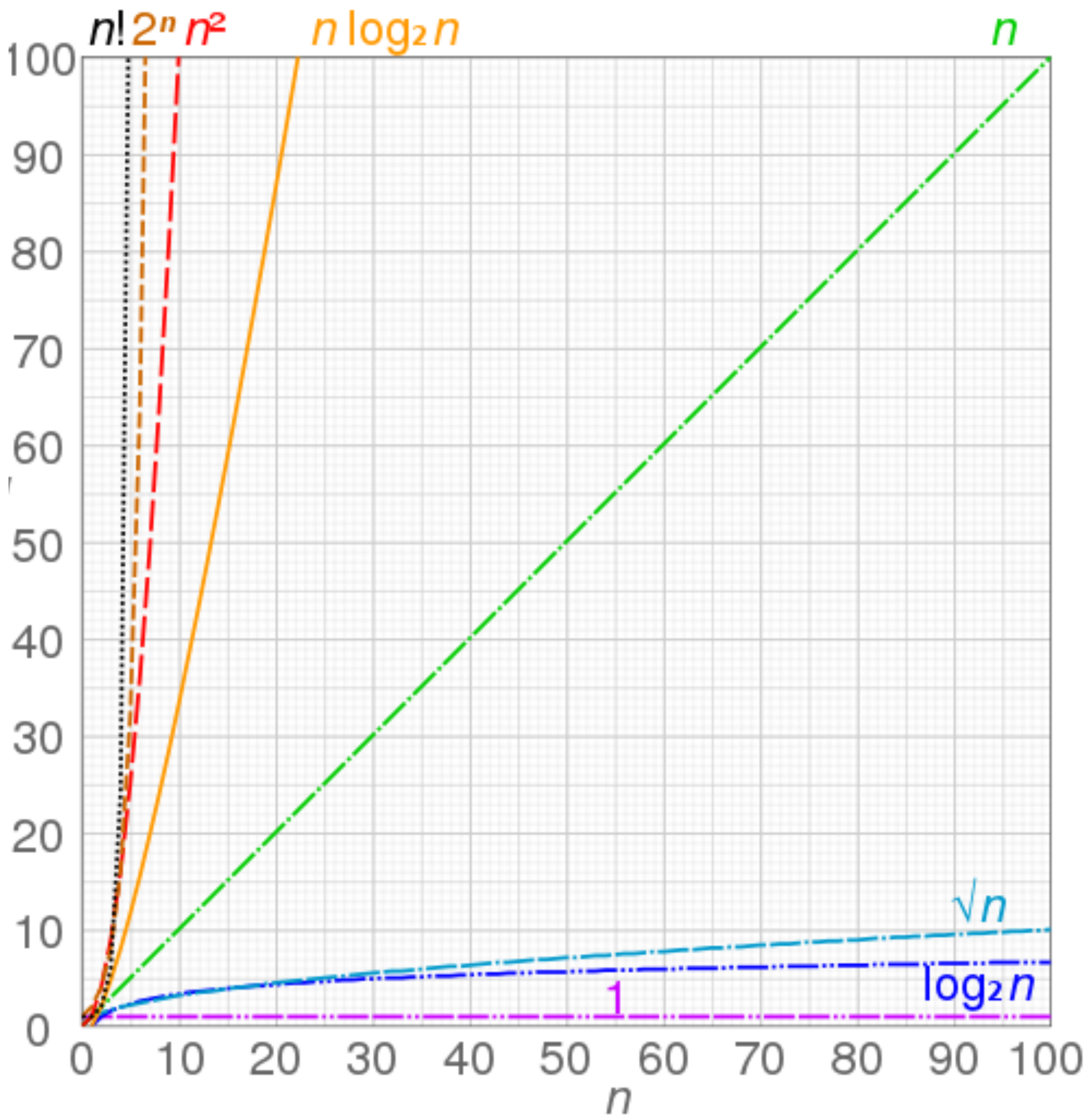
Order of growth analogy: Effort to drive on different types of road

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. $3n$
3. $6n-2$
4. $15n + 44$
5. $50n\log(n)$
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

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

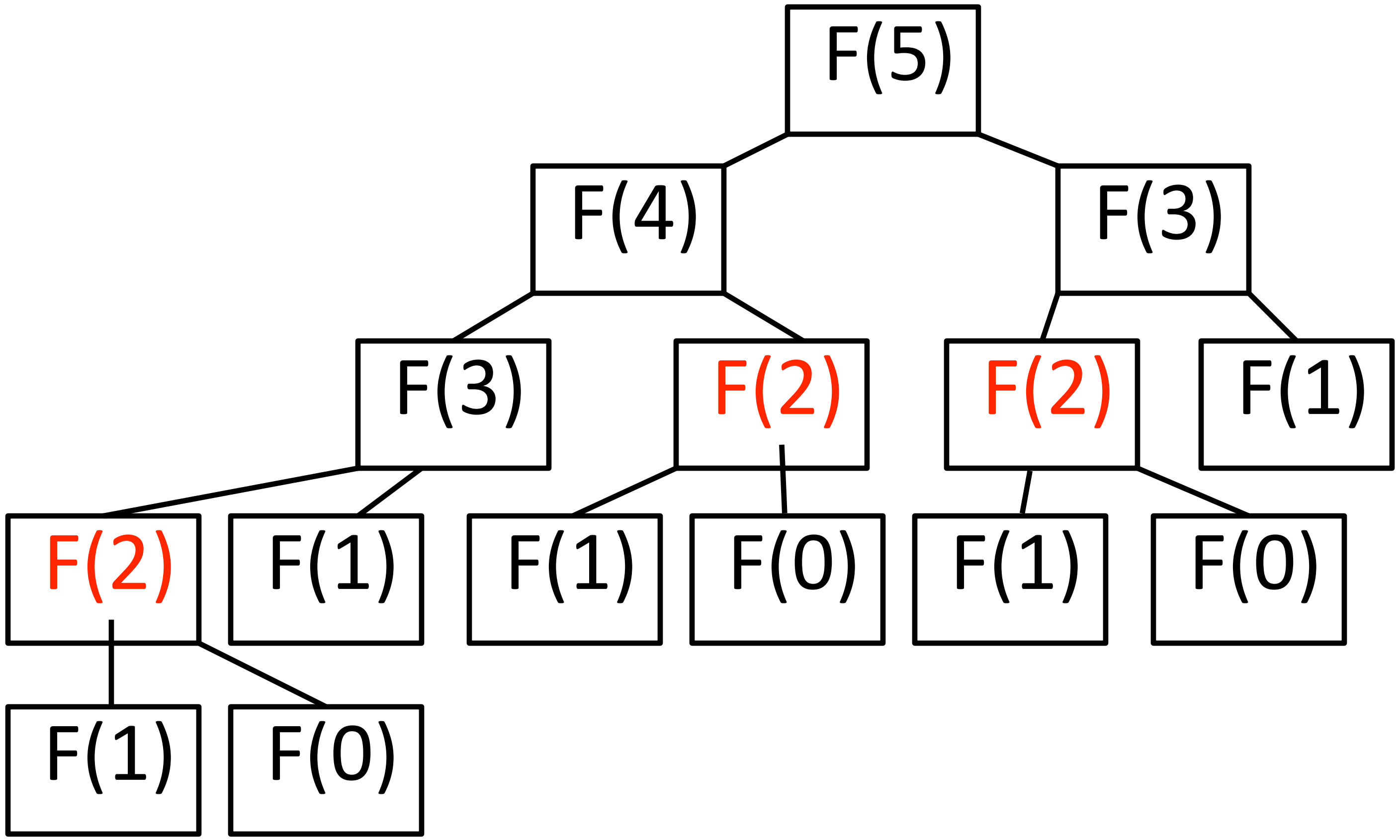
What is the Big O running time of sum()?

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

A. $O(n^2)$
B. O(n)
C. O(n^3)
D. O(log n)
E. None of the above

Why Big-O is useful in analysis of recursive fib?

Derive $T(n) = O(2^n)$



Emprical Analysis: Recursive Fibonacci Running Time

For recursive fibonacci algorithm, we derived that $T(n) = O(2^n)$

How well does this represent practice?

Observation: Time grows fast — roughly 1.6x per n.

Hypothesis: Exponential growth, like $T(n) = a * b^n$?

n	Time (ms)
40	788.09
41	1270.18
42	2070.68
43	3391.74
44	6411.54
45	9589.44
50	100329.11

Tested on my machine

Ratios between consecutive n:

- $n = 41$ to 42: $2070.68/1270.18 \approx 1.63$
- $n = 42$ to 43: $3391.74/2070.68 \approx 1.64$
- $n = 43$ to 44: $6411.54/3391.74 \approx 1.89$
- $n = 44$ to 45: $9589.44/6411.54 \approx 1.50$
- **Average**: ~1.66

Confirming Exponential Growth

$T(n) = a * b^n \rightarrow \log_2(T(n)) =$

Confirming Exponential Growth

$T(n) = a * b^n \rightarrow \log_2(T(n)) = \log_2(a) + n \log_2(b)$

Calculate:

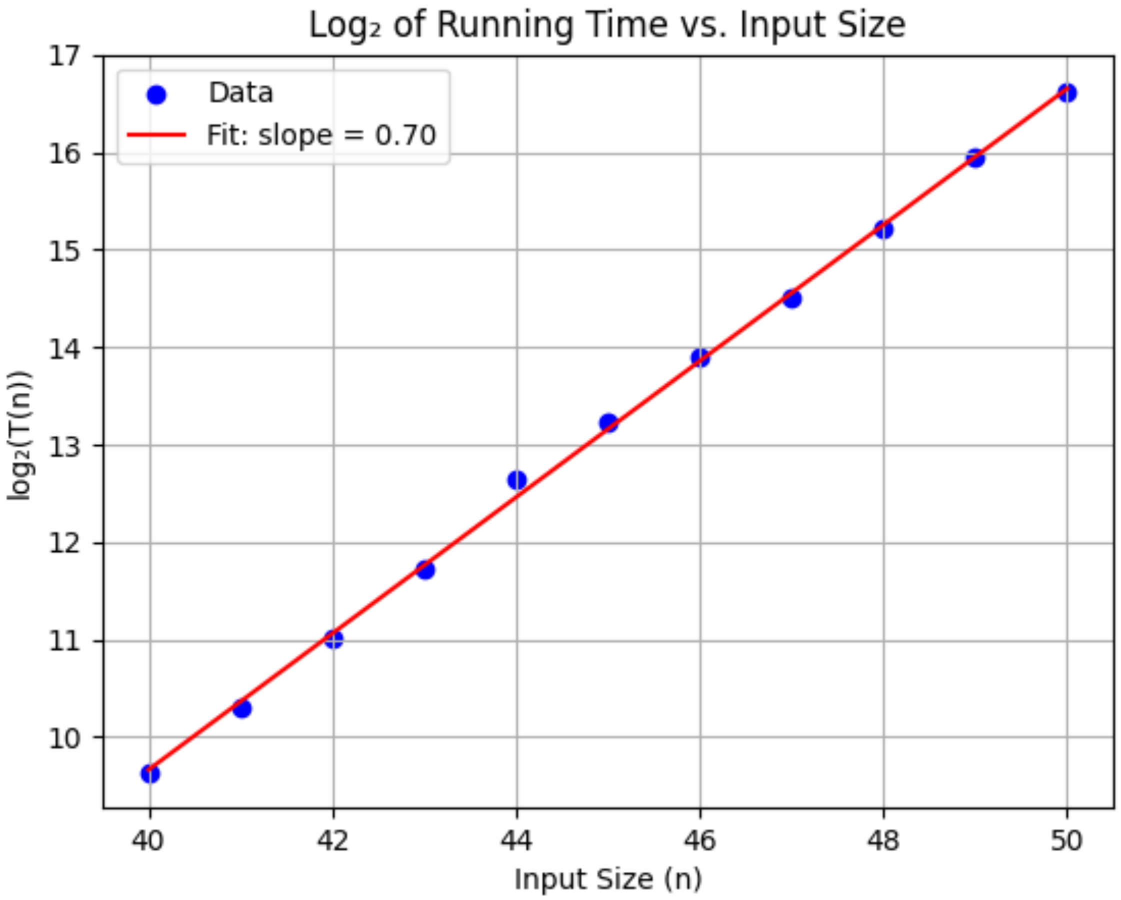
$\log_2(788.09) \approx 9.62$ (n=40)

$\log_2(100329.11) \approx 16.61$ (n=50)

Slope = (16.61 - 9.62) / (50 - 40) ≈ 0.70

$b \approx 2^{0.7} \approx 1.62 \approx \varphi$ (1.618)

$a \approx 2^{-18.39}$



Lab01: Do a similar empirical analysis for the 3-sum problem!!

Comparing predictions for T(200)

How does our prediction for T(200) compare with Prof. Dasgupta's (2^{92} s)?

- Our empirical result: $T(n) \approx 2^{(-18.39+0.7n)}$ ms $\approx 2^{(-28.39+0.7n)}$ s
- Our prediction for $T(200) \approx 2^{111}$ s
- Dasgupta's prediction $= 2^{92}$ s
- Our predicted running time is larger by a factor of $2^{19} = 5 * 10^5$

- What can account for the difference in the results?

Lab01: Do a similar empricial analysis for the 3-sum problem!!

Next time

- Abstract Data Types (OOP implementation of LinkedList)

Credits and references:

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