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.

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

What is the main takeaway so far?

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

Theoretical analysis - Big-Oh (Today)

Questions of interest:

- Why is Algo A so slow? ← looking for insight
- How do we quantify efficiency?
- Is Algo A better than Algo B?
- When will my code finish running?

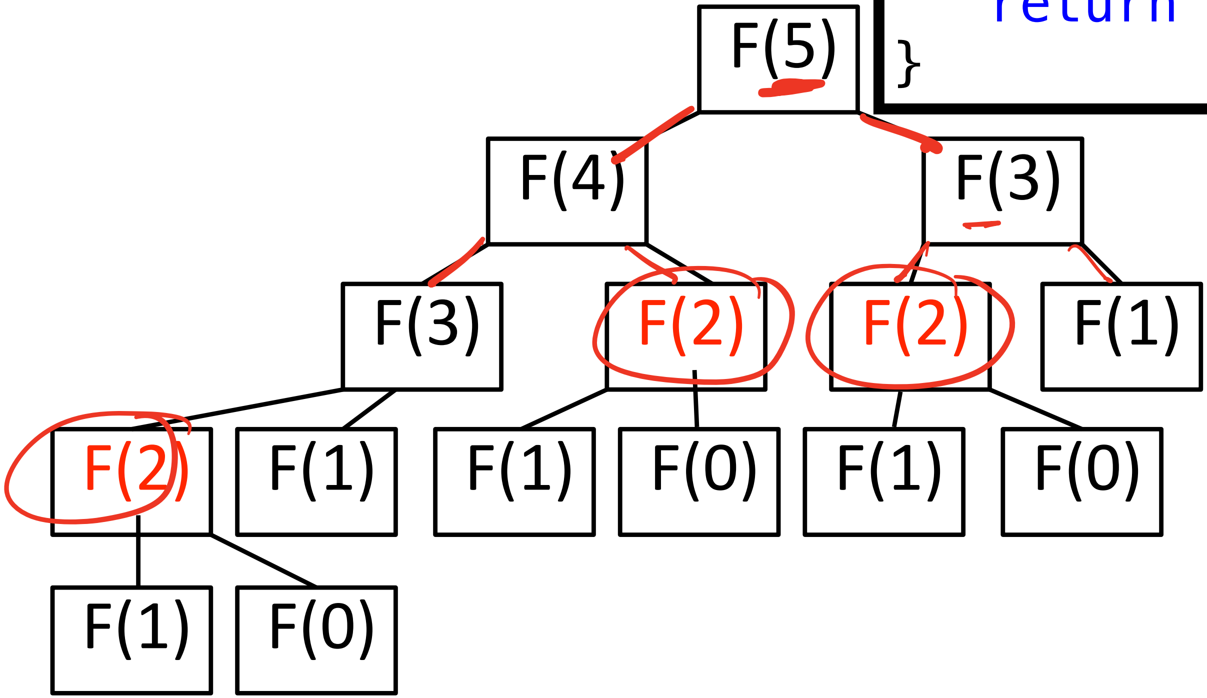
Theory + real data (Lab 01)

+ Practical question

Why So Slow?

Too many recursive calls.

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



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**

Big idea: Count operations instead of absolute time!

Machine model used for analysis

Big Idea: 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
- **Assumption:** 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 ; i++)
    A[i] = A[i-1] + A[i-2]

    return A[n]
}
```

1 op

4 ~~2~~ ops

1 op   1 op   2 ops

2 ops   2 ops

2 ops

1 op

+ 3 5 ops

Loop runs (n-1)

(n-1)(7+3)+1+1

+ 2 ops

$T(n) = 5 + 10(n-1) + 4$

$= 10n - 1$

$T(n) = 10n - 1$

Iterative Fibonacci Algorithm

Running time

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

$\boxed{T(n) = 2n+1}$

Effect of constant factors

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

$$T(n) = 10n - \cancel{3}$$

$$T(n) = 2n + \cancel{1}$$

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

- Think about 10n - 3 vs. 10n and 2n + 1 vs. 2n
- What about 10n vs 2n?

10n 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

10n vs. 2n

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

Both cars take a similar level of effort (linear) when traveling on a highway.
Think about effort to drive on a smooth highway vs. winding mountain vs. off-road jungle trek

n^2 2^n

Orders of growth

Analogy: Trips that need a similar effort have the same **order of growth**

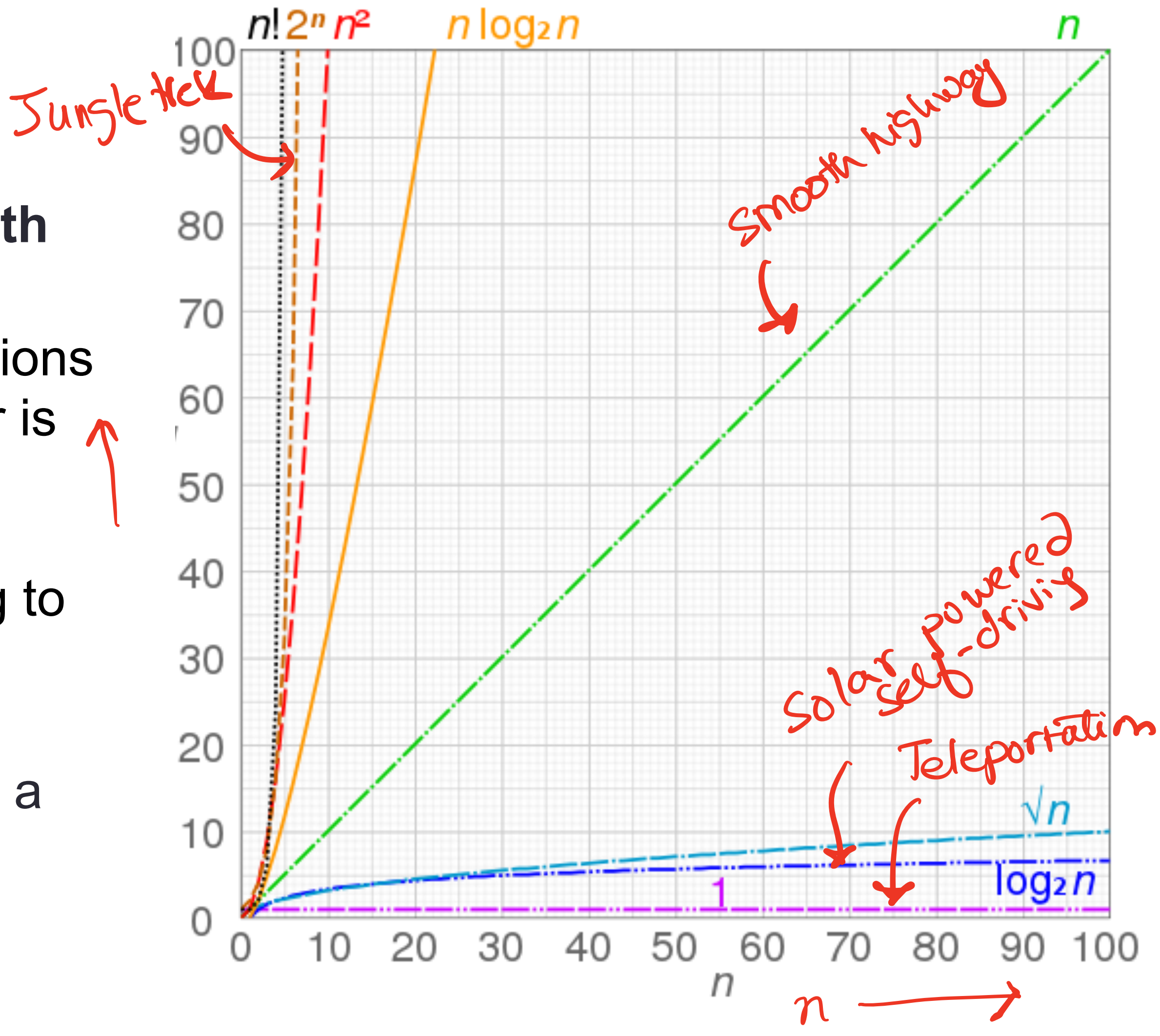
An **order of growth** is a set of functions whose (asymptotic) growth behavior is considered equivalent.

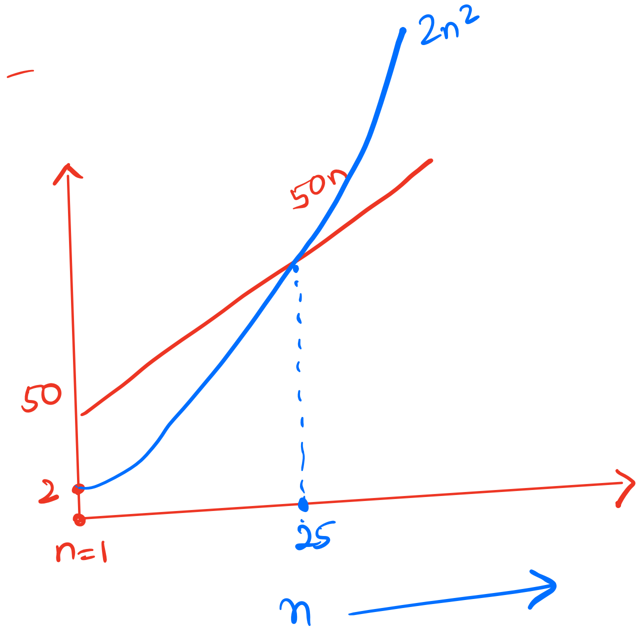
For example, $2n$, $100n$ and n belong to the same order of growth (linear)

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

A. 50n

B. $2n^2$





Takeaway: Quadratic function will always overtake a linear function irrespective of the constants

Orders of growth

$$2^n > n^2 > n\log n > n > \log n > 1$$

Big-O notation

- Big-O notation provides an asymptotic upper bound on the running time
- Its like saying "No matter how bad it gets, the effort won't exceed this level of difficulty"

$T(n) = n^2 + \log n$

we write

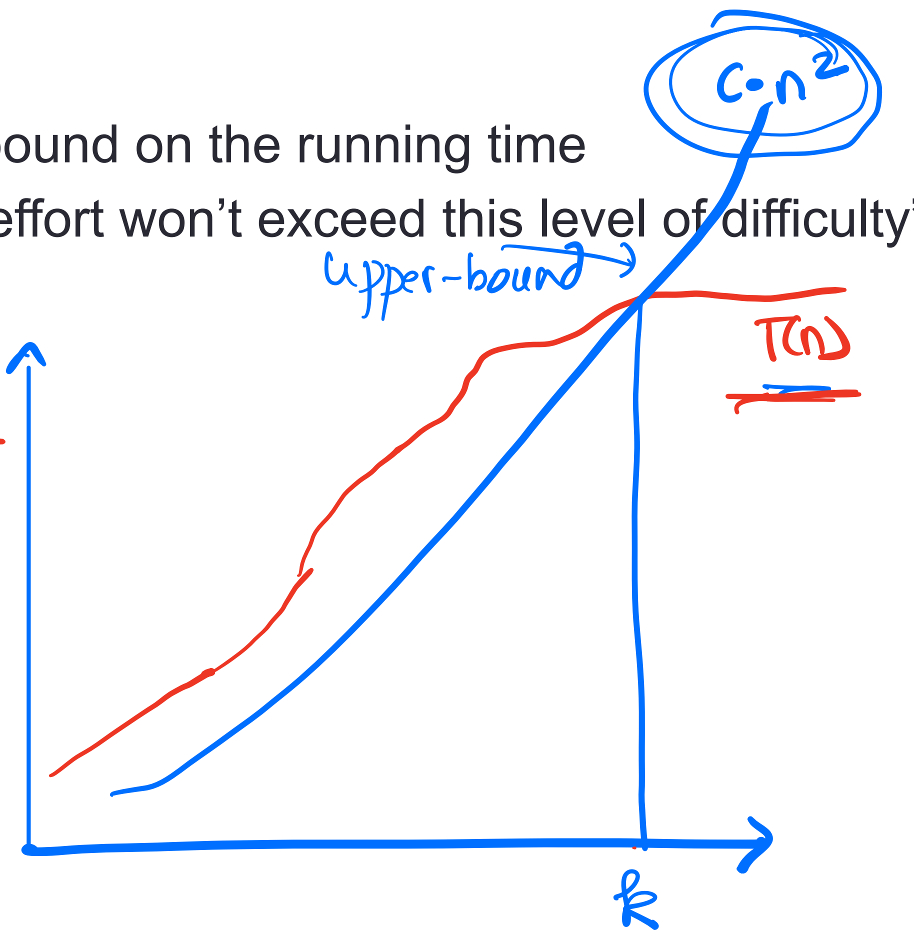
$T(n) = O(n^2)$

const

we mean.

$T(n) \leq c \cdot n^2$, for $n \geq k$

const



$T(n) = 3n^2 + \underline{n \log n}$

To Show: $T(n) = O(n^2)$, we need to show there exist constants $c, k > 0$

such that $T(n) \leq c \cdot n^2$, for $n \geq k$

(given)

$T(n) = 3n^2 + n \log_2 n$

$\leq 3n^2 + n^2$, for $n \geq 2$ (because $n \geq \log_2 n$)

$= \underset{\downarrow}{4} n^2$, for $n \geq \underset{\downarrow}{2}$

c $\qquad$ k

Since we found that there exist positive constants $c = 4$ and $k = 2$, therefore. $T(n) = O(n^2)$

Definition of Big-O

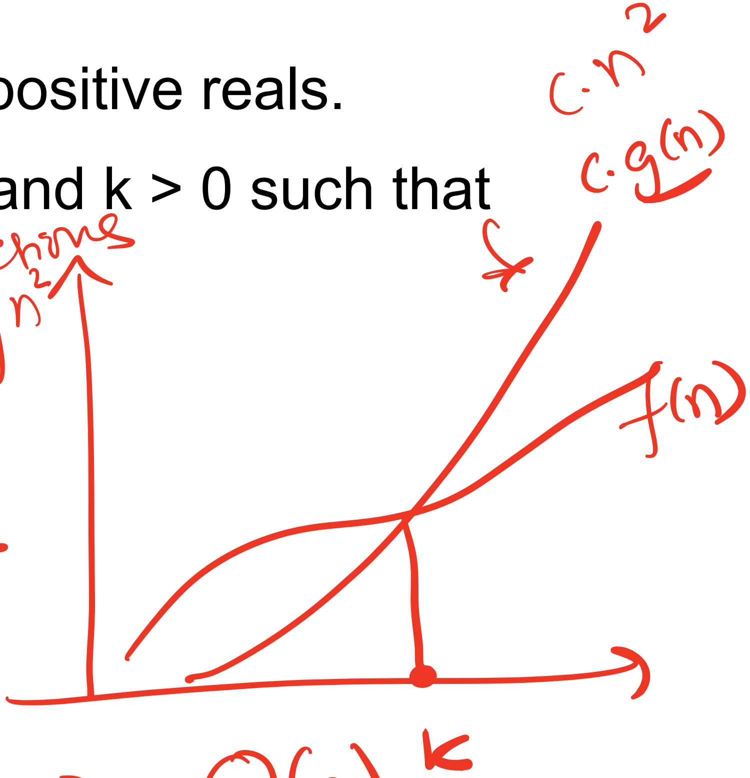
f(n) and g(n) map positive integer inputs to positive reals.

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.

Set of all functions upper bounded by n^2

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

$O(n^2) = \{ 1, \log(n), n, 3n, n^2, 3n^2 + \log n \}$



$T(n) = O(n^2)$ means $T(n) \in O(n)$

$T(n) = O(n)$

Express in Big-O notation

1. 10000000 = O(1)
2. 3n = O(n)
3. 6n-2 = O(n)
4. 15n + 44 = O(n)
5. 50nlog(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) $O(3^n)$

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()?

$T(n) = 3n + 5$

```
/* 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;
}
```

$T(n) = O(1) + \frac{n}{2} O(1) + O(1)$

$= O(n)$

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;
}
```

Iteration number	Value of i
1	1
2	2
3	2^2
4	2^3
5	2^4
$\boxed{7}$	$\boxed{2^6}$
⋮	
k	2^{k-1}

On iteration k, value of i is 2^{k-1}.

(See next page for steps on how to get to this answer)

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

Loop will end when loop variable (i) becomes greater than or equal to n.

Let's assume loop ends after the k^{th} iteration, when $i \geq n$

Plug in value of i in terms of iteration number (k)

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

Take log (base 2) on both side

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

$$\boxed{k \geq \log_2(n) + 1}$$

upper bound on the number of times the loop runs

$$T(n) \leq c_1 + c_2 \cdot (\log n + 1) + c_3$$

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

$$= O(\log n)$$

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

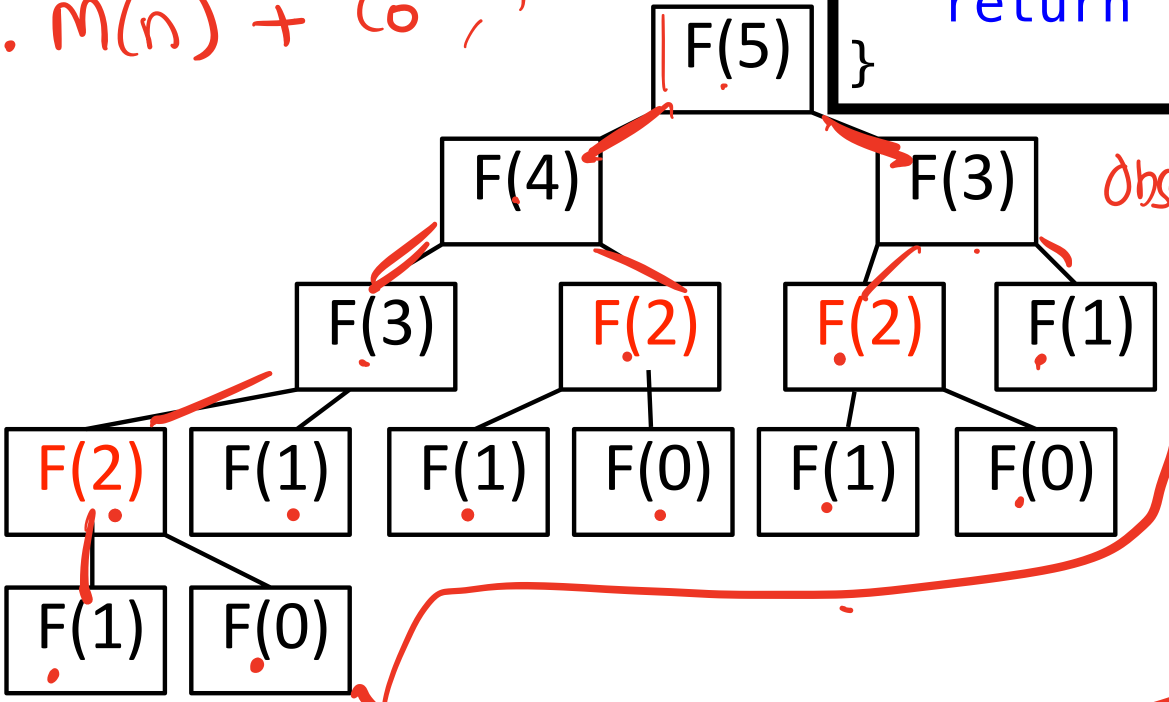
Derive $T(n) = O(2^n)$

$T(n) = c \cdot M(n) + c_0$, for large n

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

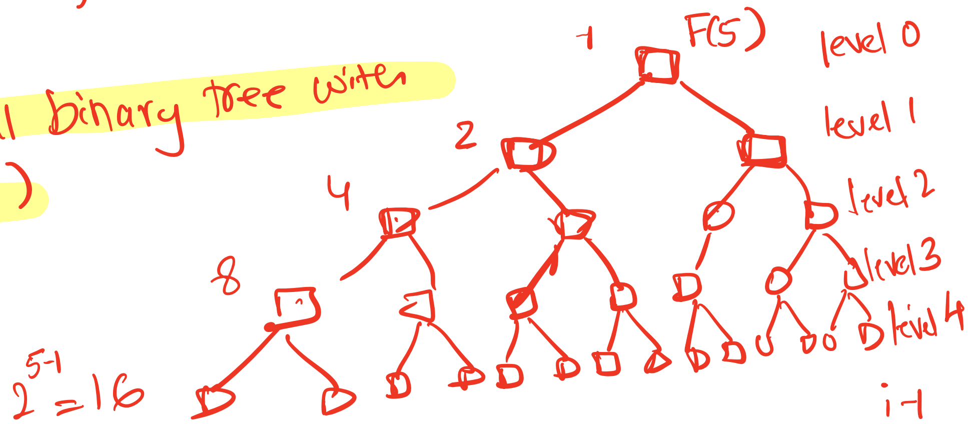
Observe:

For every function call we perform some const time operation.



Running time $T(n)$ is proportional to the total number of function calls needed to calculate $F(n)$, let's denote that by $M(n)$. We will now upper bound $M(n)$ by adding more function calls to get a Full binary tree

Example of a full binary tree with 4 levels (n=5)



Note that number of nodes at level i is 2^{i-1}

Add all the nodes on levels 0 to $n-1$

$M(n) \leq$ Number of nodes in a tree with $n-1$ levels

$= 1 + 2 + 4 + 8 + 16 + \dots 2^{n-1}$

$= 2^0 + 2^1 + 2^2 + \dots \dots 2^{n-1}$

$= 2^n - 1$

$= O(2^n)$

$T(n) = c_1 \cdot M(n) + c_2$ (for constants c_1, c_2)

$= c_1 \, O(2^n) + c_2$

$= O(2^n)$

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

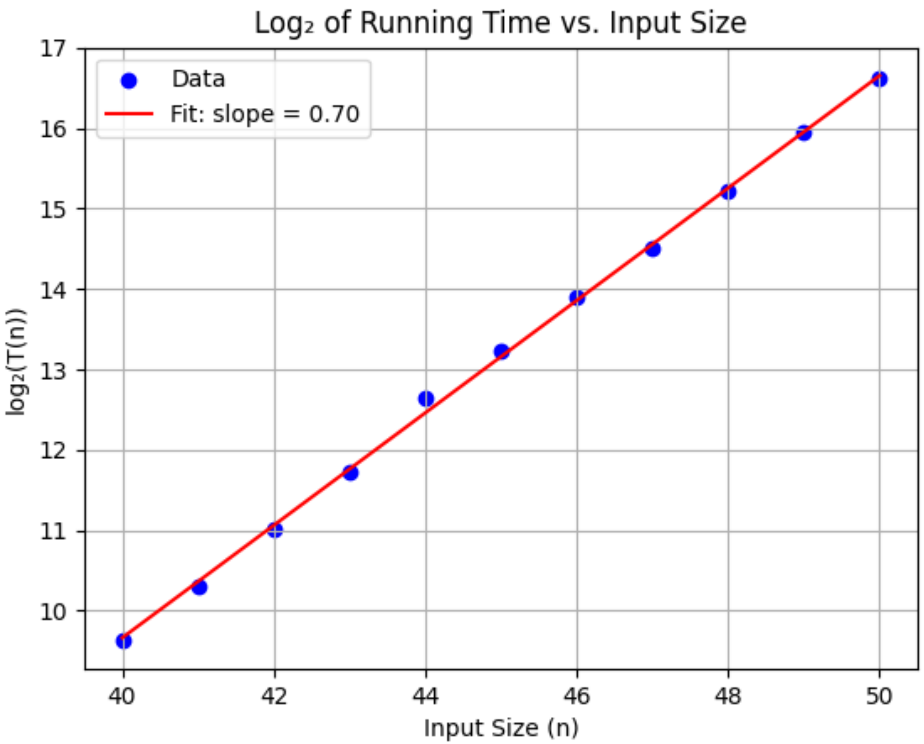
$T(n) = a * b^n \rightarrow \log_2(T(n)) = \log_2(a) + n \log_2(b)$

Calculate:

$\log_2(788.09) \approx 9.62$ (n=40)
$\log_2(100329.11) \approx 16.61$ (n=50)

Slope = $(16.61 - 9.62) / (50 - 40) \approx 0.70$

$b \approx 2^{0.7} \approx 1.62 \approx \varphi$ (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
https://cseweb.ucsd.edu/~dasgupta/book/toc.pdf