RUNNING TIME ANALYSIS

Problem Solving with Computers-I

https://ucsb-cs24-sp17.github.io/

Performance questions

- How efficient is a piece of code?
  - **CPU time usage (Running time complexity)**
  - Memory usage
  - Disk usage
  - Network usage

Which implementation is faster?

```
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]
}
```

A. *Recursive* algorithm

B. *Iterative* algorithm

C. *Both are equally fast*

Let's learn some surprising facts!

demo

What we really care about is how the running time scales 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 scale?

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

….let’s say on….

Ack: Prof. Sanjoy Das Gupta

NEC Earth Simulator

Can perform up to 40 trillion operations per second.

Ack: Prof. Sanjoy Das Gupta

The running time of the recursive implementation

The Earth simulator needs 2^{95} 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)
}
```

Ack: Prof. Sanjoy Das Gupta

What is the fundamental difference between the two

```
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]
}
```

Algorithm Analysis

- Focus on primitive operations:
  - Data movement (assignment)
  - Control statements (branch, function call, return)
  - Arithmetic and logical operation

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

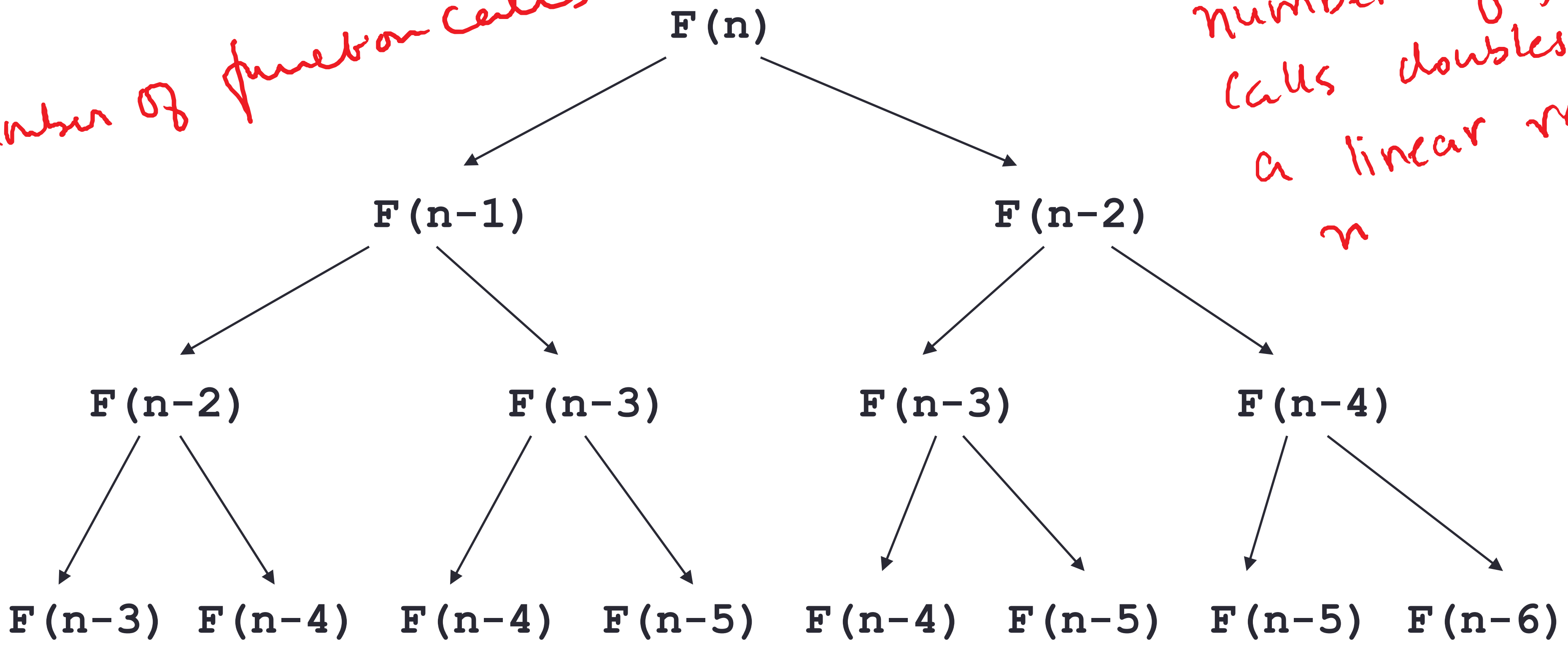
```
function F(n){
    if(n == 1) return 1
    if(n == 2) return 1
return F(n-1) + F(n-2)
}
```

Post mortem on the recursive function

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

Total number of function calls $\propto 2^N$

number of function calls doubles with a linear reduction in n



The same subproblems get solved over and over again!

Ack: Prof. Sanjoy Das Gupta

How bad is exponential time?

Need $2^{0.694n}$ operations to compute F_n.

Eg. Computing F_{200} needs about 2^{140} operations.

How long does this take on a fast computer?

40 trillion operations per second on NEC supercomputer -> 2^{95} seconds

Running time analysis of the iterative algorithm

```
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]
```

n primitive operation

2 operations

$(n-3) * c_2$ operation

1 operation

Total: $n + 2 + (n-3)c_2$

The number of operations is proportional to *n*.
[Previous method: $2^{0.7n}$]

F_{200} is now reasonable to compute, as are F_{2000} and F_{20000}.

Notice its linear in n

We just did an asymptotic analysis of the two algorithms

Asymptotic Analysis

- Goal: to simplify the analysis of running time by ignoring “details” which may be an artifact of the underlying implementation:
  - E.g., 1000001 ≈ 1000000
  - Similarly, $3n^2 \approx n^2$
- Capture the essence: how the running time of an algorithm increases with the size of the input in the limit (for large input sizes)

How do you do the analysis:

- Count the number of primitive operations executed as a function of input size.
- Express the count using **O-notation** to express

What is big-Oh about?

- Intuition: avoid details when they don't matter, and they don't matter when input size (N) is big enough
  - For polynomials, use only leading term, ignore coefficients: linear, quadratic

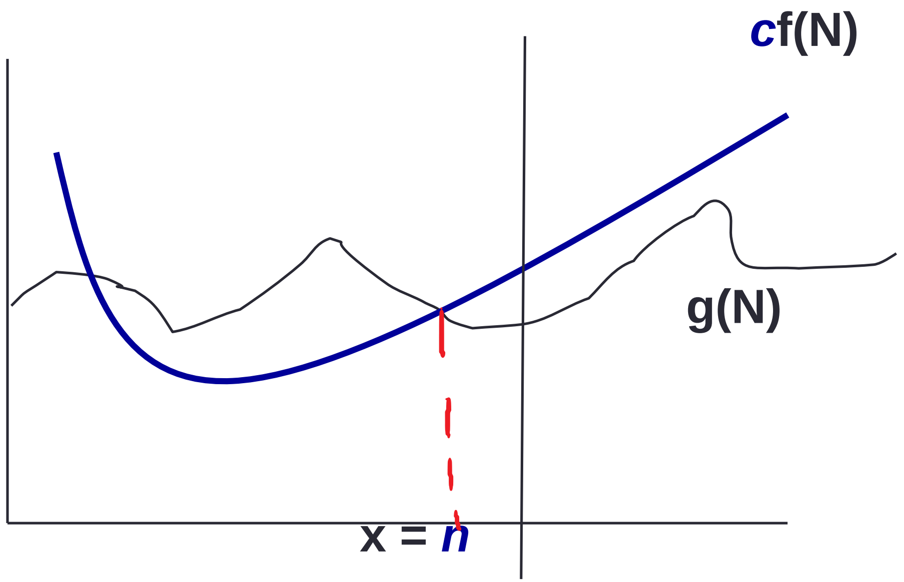
$y = 3x$ $\quad y = 6x-2$ $\quad y = 15x + 44$ $\quad O(x)$

$y = x^2$ $\quad y = x^2-6x+9$ $\quad y = 3x^2+4x$ $\quad O(x^2)$

- Compare algorithms *in the limit*
- 20N hours v. N^2 microseconds:
  - *which is better?* As N grows large 20N is better

Big-O: More formal definition

- The big-oh Notation:
  - Asymptotic upper bound
- Formally:
  - A function $g(N)$ is $O(f(N))$ if there exist constants c and n such that $g(N) < cf(N)$ for all $N > n$
  - f(n) and g(n) are functions over non-negative integers
- O-notation is an upper-bound, this means that N is $O(N)$, but it is also $O(N^2)$; we try to provide *tight* bounds.
- Used for worst case analysis



Writing Big O

- Simple Rule: Ignore lower order terms and constant factors:
  - $50n \log n$ is $O(n \log n)$
  - $7n – 3$ is $O(n)$
  - $8n^2 \log n + 5 n^2 + n + 1000$ is $O(n^2 \log n)$
- Note: even though $50 n \log n$ is $O(n^5)$, it is expected that such approximation be as tight as possible (***tight upper bound***).

Comparing asymptotic running times

N	$O(\log N)$	$O(N)$	$O(N \log N)$	$O(N^2)$
10	0.000003	0.00001	0.000033	0.0001
100	0.000007	0.00010	0.000664	0.1000
1,000	0.000010	0.00100	0.010000	1.0
10,000	0.000013	0.01000	0.132900	1.7 min
100,000	0.000017	0.10000	1.661000	2.78 hr
1,000,000	0.000020	1.0	19.9	11.6 day
1,000,000,000	0.000030	16.7 min	18.3 hr	318 centuries

An algorithm that runs in O(n) is better than one that runs in $O(n^2)$ time
Similarly, O(log n) is better than O(n)
Hierarchy of functions: $\log n < n < n^2 < n^3 < 2^n$

10^6 instructions/sec, runtimes

Next time

- More linked list with classes