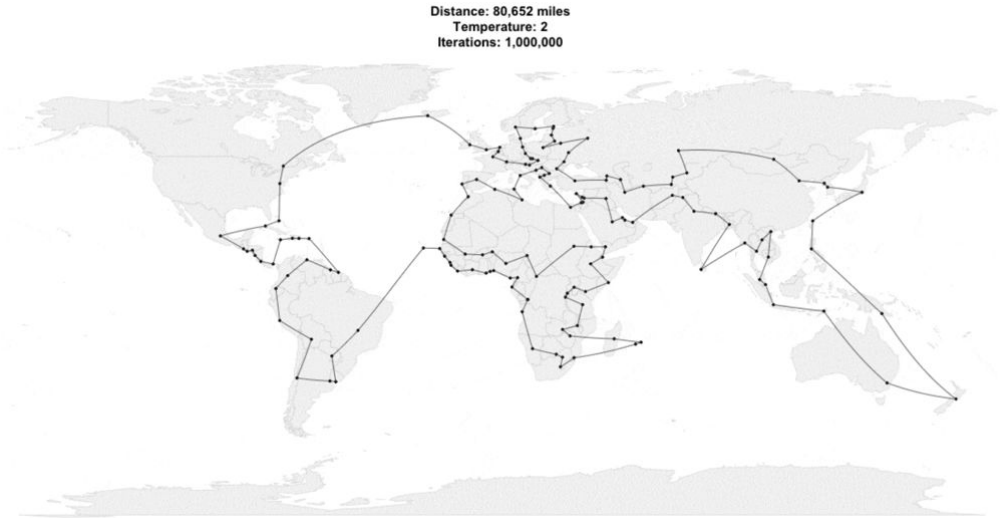


Lecture 15

Asymptotics 3

CS61B, Fall 2024 @ UC Berkeley

Slides Credit: Josh Hug

Intuitive Definitions of Functions

If my function f(n) is…	Doubling N	Adding 1 to N
$\Theta(1)$	Doesn't affect runtime	
$\Theta(\log n)$ (base independent)	Adds 1 to runtime	Affects runtime minimally
$\Theta(n)$	Doubles runtime	Adds 1 to runtime
$\Theta(n^2)$	Quadruples runtime	Adds n to runtime
$\Theta(2^n)$ (base dependent)	Squares runtime	Doubles runtime

Summations

Lecture 13, CS61B, Spring 2025

Summations

Analyzing Programs

- Nested For Loops
- Amortized Analysis
- Recursive Analysis

Mergesort

Summations and Recursive Functions

Sometimes, it's not easy to get an explicit runtime function, but it is possible to Theta bound the function. Often, you can reduce a runtime function to one of:

- A summation $\Sigma_{i=1 \text{ to } N} f(i) = f(1) + f(2) + f(3) + \ldots + f(N)$
- A recursively defined function

Triangle Summation

Example: $f(N) = \Sigma_{i=1 \text{ to } N}\, i = 1 + 2 + 3 + 4 + \ldots + N \in \Theta(N^2)$

Approach 1: Algebra

$f(N) = 1 + 2 + 3 + \ldots + (N - 3) + (N - 2) + (N - 1) + N$

$f(N) = (N - 1) + (N - 2) + (N - 3) + \ldots + 3 + 2 + 1 \quad +N$

$2f(N) = N + N + \ldots + N \text{ (N-1 copies)} + N + N$

$\therefore f(N) = N(N + 1)/2$

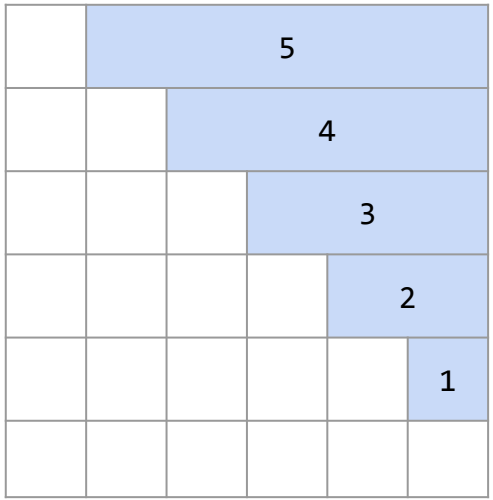
Triangle Summation

Example: $f(N) = \Sigma_{i=1 \text{ to } N}\ i = 1 + 2 + 3 + 4 + \ldots + N \in \Theta(N^2)$

Approach 2: Geometry

The sum is given by the area of this step-wise triangle, which is approximately a triangle.

Area of triangle is base * height/2 = N * N/2 $\in \Theta(N^2)$



Summation of Halves

What is 1 + ½ + ¼ + ⅛ + …

Summation of Halves

What is 1 + ½ + ¼ + ⅛ + …

2

Algebraic proof:

Let x = 1 + ½ + ¼ + ⅛ + …

Then 2x = 2 + 1 + ½ + ¼ + ⅛ + …

So 2x - x = 2 + 0…

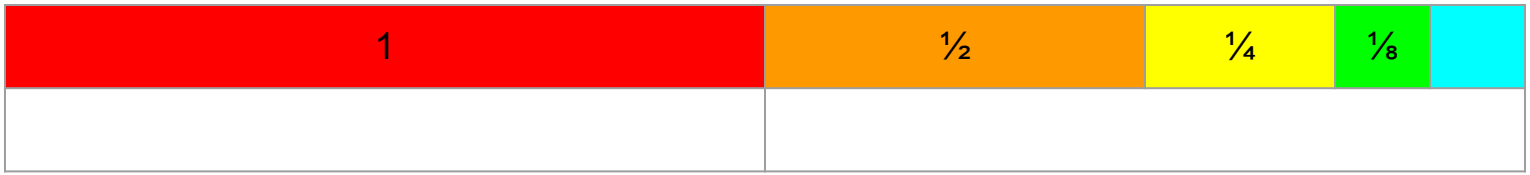
So x = 2

Summation of Halves

What is 1 + ½ + ¼ + ⅛ + ...

2

Geometric proof: Each step moves halfway to the "2", so you always approach, but never hit, 2.



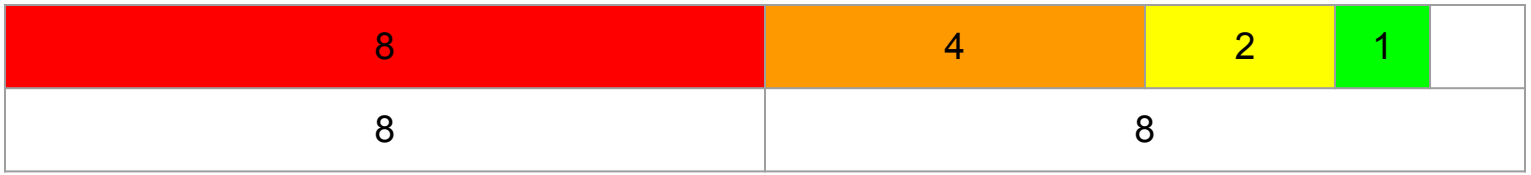
Summation of Powers of 2

What is 1 + ½ + ¼ + ⅛ + …

2

If you multiply everything in the below by 2^N, you get the sequence

$2^N+2^{N-1}+\ldots+2^1+\underbrace{1 + ½ + ¼ + ⅛ + \ldots}_{1} = 2*2^N$



So $1+2+4+\ldots+2^N = 2*2^N-1$

Infinite Summation

For any $r < 1$, $\Sigma_{i=1 \text{ to } \infty} r^i = 1+r+r^2+r^3+...$ the sum can be evaluated:

Let $x =$ $1+$ $r+r^2+r^3...$

Then $r*x =$ $r+r^2+r^3...$

So $x-r*x = 1$

$x = 1/(1-r)$

(Note: If $r \geq 1$, then the sum diverges to infinity. Formally, the power series has a radius of convergence of 1)

Exponential Summation

Show that for any $r > 1$, $\Sigma_{i=0 \text{ to } N} r^i = 1+r+r^2+r^3+...+r^N \in \Theta(r^N)$ by showing:

$\Sigma_{i=0 \text{ to } N} r^i = 1+r+r^2+r^3+...+r^N \in \Omega(r^N)$

$\Sigma_{i=0 \text{ to } N} r^i = 1+r+r^2+r^3+...+r^N \in O(r^N)$

Exponential Summation

Show that for any $r > 1$, $\Sigma_{i=0 \text{ to } N} r^i = 1+r+r^2+r^3+...+r^N \in \Theta(r^N)$ by showing:

$\Sigma_{i=0 \text{ to } N} r^i = 1+r+r^2+r^3+...+r^N \in \Omega(r^N)$

The sum is greater than the last term, so $\Sigma_{i=1 \text{ to } N} r^i > r^N \geq 1*(r^N)$

$\Sigma_{i=0 \text{ to } N} r^i = 1+r+r^2+r^3+...+r^N \in O(r^N)$

The sum is the same as $r^N*(1+1/r+(1/r)^2+...+(1/r)^N) = r^N*(\Sigma_{i=1 \text{ to } N} (1/r)^i)$

Note that $\Sigma_{i=1 \text{ to } N} (1/r)^i < \Sigma_{i=1 \text{ to } \infty} (1/r)^i = 1/(1-1/r)$, because if $r>1$, then $1/r < 1$.

So the sum is $\leq r^N*(1/(1-1/r))$, where $(1/(1-1/r))$ is a constant

Repeat After Me...

There is no magic shortcut for asymptotic analysis problems

- Function analysis often requires careful thought.
- CS70 and especially CS170 will cover this in much more detail.
- This is not a math class, though we'll expect you to know these:
  - $1 + 2 + 3 + \ldots + N$ $= N(N+1)/2$ $= \Theta(N^2)$ ← Sum of First Natural Numbers (Link)
  - $1^k+2^k+3^k+ \ldots + N^k$ $(k >= 0)$ $= \Theta(N^{k+1})$ ← Generalization of ^
  - $1 + 2 + 4 + 8 + \ldots + 2^N$ $= 2*2^N - 1$ $= \Theta(2^N)$ ← Sum of First Powers of 2 (Link)
  - $k^0+k^1+k^2+ \ldots + k^N$ $(k > 1)$ $= \Theta(k^N)$ ← Generalization of ^

Repeat After Me...

There is no magic shortcut for asymptotic analysis problems

- Function analysis often requires careful thought.
- CS70 and especially CS170 will cover this in much more detail.
- This is not a math class, though we'll expect you to know these:
  - $1 + 2 + 3 + \ldots + N$ $= N(N+1)/2$ $= \Theta(N^2)$ ← Sum of First Natural Numbers (Link)
  - $1^k+2^k+3^k+ \ldots + N^k$ $(k >= 0)$ $= \Theta(N^{k+1})$ ← Generalization of ^
  - $1 + 2 + 4 + 8 + \ldots + N$ $= 2*N - 1$ $= \Theta(N)$ ← Alternate form of sum of powers
  - $k^0+k^1+k^2+ \ldots + N$ $(k > 1)$ $= \Theta(N)$ ← Generalization of ^

Repeat After Me...

There is no magic shortcut for asymptotic analysis problems

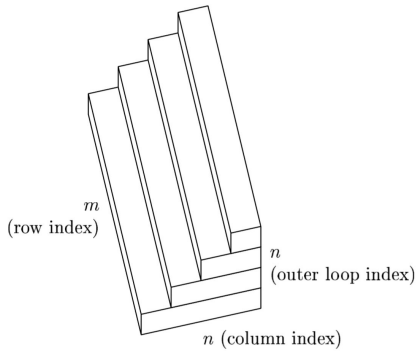
- Function analysis often requires careful thought.
- CS70 and especially CS170 will cover this in much more detail.
- This is not a math class, though we'll expect you to know these:
  - $\sum_{i=1 \text{ to } N} i$ $= N(N+1)/2$ $= \Theta(N^2)$ ← Sum of First Natural Numbers (Link)
  - $\sum_{i=1 \text{ to } N} i^k$ $(k >= 0)$ $= \Theta(N^{k+1})$ ← Generalization of ^
  - $\sum_{i=0 \text{ to } N} 2^i$ $= 2*2^N - 1$ $= \Theta(2^N)$ ← Sum of First Powers of 2 (Link)
  - $\sum_{i=0 \text{ to } N} k^i$ $(k > 1)$ $= \Theta(k^N)$ ← Generalization of ^

Repeat After Me...

There is no magic shortcut for asymptotic analysis problems

- Function analysis often requires careful thought.
- CS70 and especially CS170 will cover this in much more detail.
- This is not a math class, though we'll expect you to know these:
  - $1^k+2^k+3^k+ \ldots + N^k \qquad (k \geq 0) \qquad = \Theta(N^{k+1})$
  - $k^0+k^1+k^2+ \ldots + k^N \qquad (k > 1) \qquad = \Theta(k^N)$
- Strategies:
  - Find exact sum.
  - Write out examples.
  - Draw pictures.

QR decomposition runtime, from "Numerical Linear Algebra" by Trefethen.



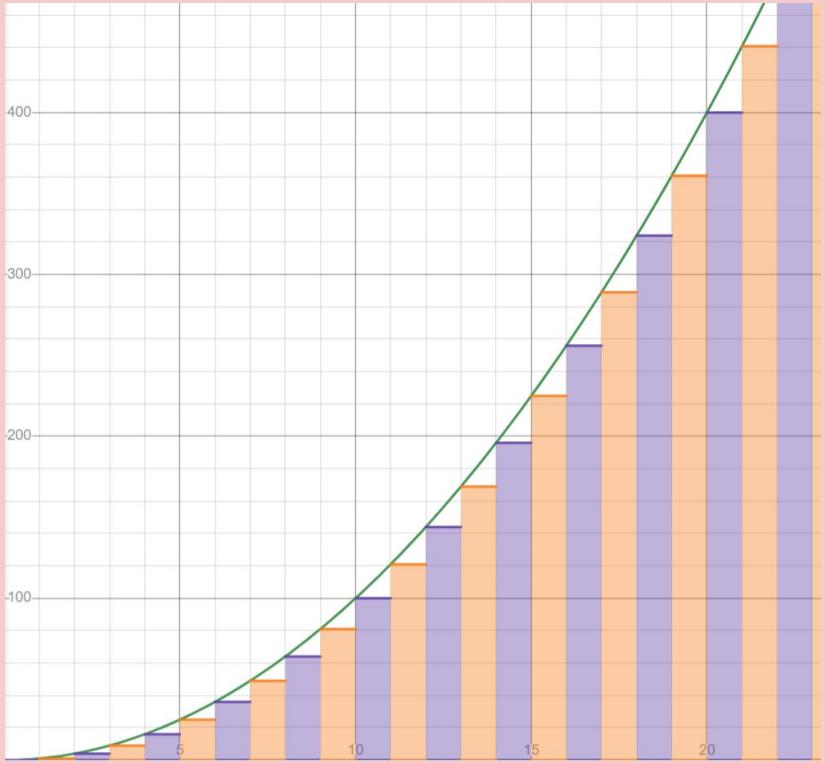
The $m \times n$ rectangle at the bottom corresponds to the first pass through the outer loop, the $m \times (n-1)$ rectangle above it to the second pass, and so on.

Generalizing the two cases you need to know (Out of Scope)

- $1^k+2^k+3^k+ \dots + N^k$ $(k >= 0)$ $= \Theta(N^{k+1})$
- $k^0+k^1+k^2+ \dots + k^N$ $(k > 1)$ $= \Theta(k^N)$

$\Sigma_{i=0 \text{ to } N} f(i)$ (f continuous, increasing)

$$\int_0^N f(x)dx \approx$$



Generalizing the two cases you need to know (Out of Scope)

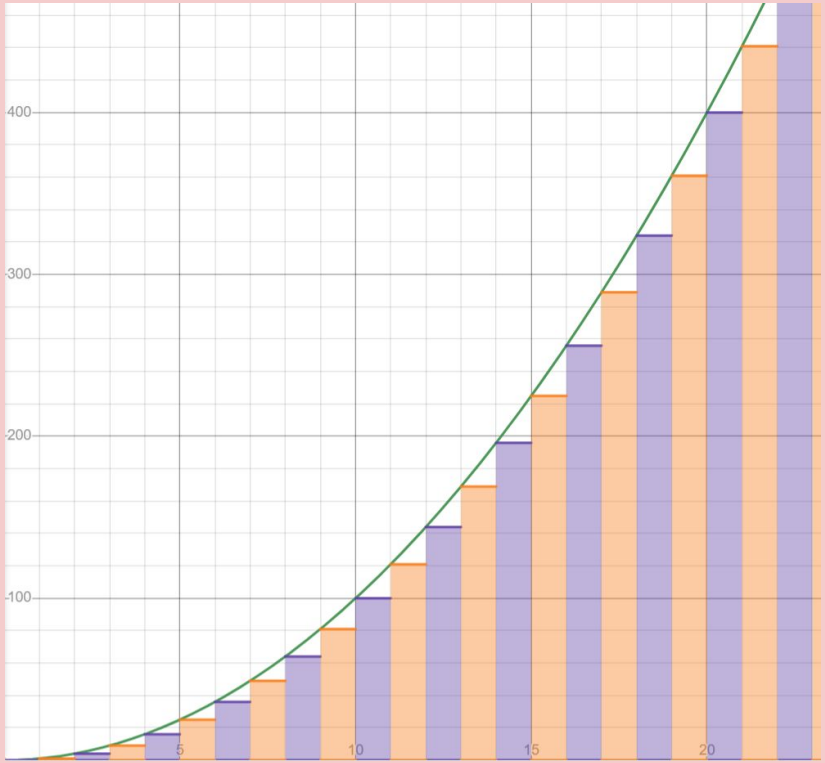
- $1^k+2^k+3^k+ \dots + N^k$ $(k >= 0)$ $= \Theta(N^{k+1})$
- $k^0+k^1+k^2+ \dots + k^N$ $(k > 1)$ $= \Theta(k^N)$

$\Sigma_{i=0 \text{ to } N} f(i) \in \Theta(\int_0^N f(x))$ (for most functions)

Some shortcuts exist

But they tend to be limited in scope

$$\int_0^N f(x)dx \approx$$



Summary (No pun intended)

Summations are difficult to evaluate

- Even minor changes to the function can yield much harder problems
- Though you can solve a decent chunk of problems by using calculus

In practice, you won't be working with the raw math and proofs, because that adds too much complexity.

Fortunately, you don't have to!

Just as in programming, we can abstract the formal math, and use the tools we proved work.

Let's discuss how to actually use these tools to compute the runtime of programs.

Nested For Loops

Lecture 15, CS61B, Fall 2024

Summations

Analyzing Programs

- **Nested For Loops**
- Amortized Analysis
- Recursive Analysis

Mergesort

Measuring a Program's Performance

Expressing a program's efficiency boils down to two main steps:

1. Define some **metric** by which the program's efficiency will be evaluated, and measure the program's efficiency according to that metric.
    - Since we'll use a Theta bound in step 2, we can ignore constant factors in this step, which is useful in simplifying runtimes
2. Classify the resulting function in a Theta class.
    - Often requires summations or special tricks to get to a form we can actually solve.

Loops Example:

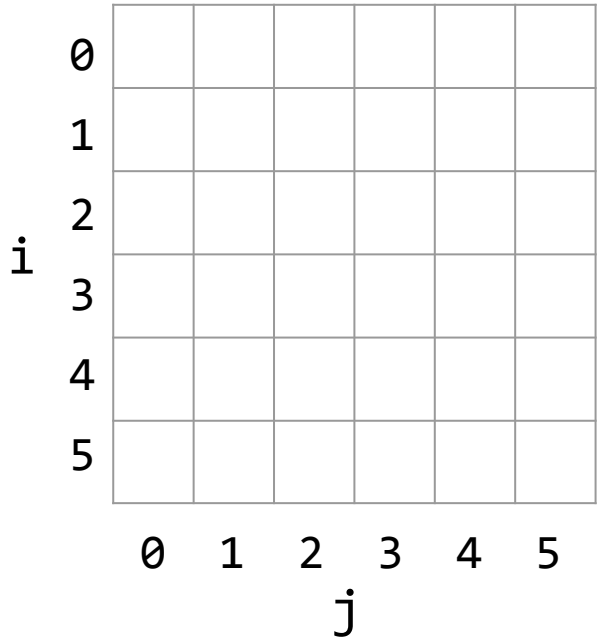
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)). By simple, we mean there should be no unnecessary multiplicative constants or additive terms.

```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int ZUG = 1 + 1;
        }
    }
}
```

A. 1
B. log N
C. N
D. N log N
E. N^2
F. Other

Note that there's only one case for this code and thus there's no distinction between "worst case" and otherwise.

Loops Example



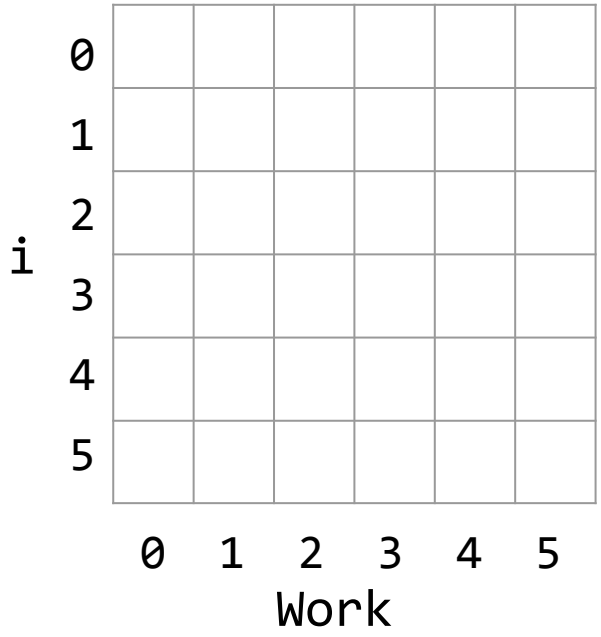

```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            //1 unit of work
        } } }
```

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Units of work C(N):

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Loops Example



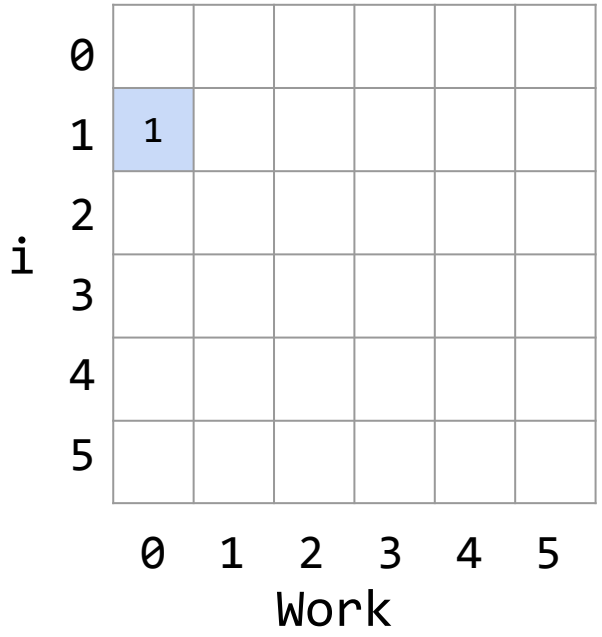

```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        //i units of work
    }
}
```

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Units of work C(N):

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Loops Example



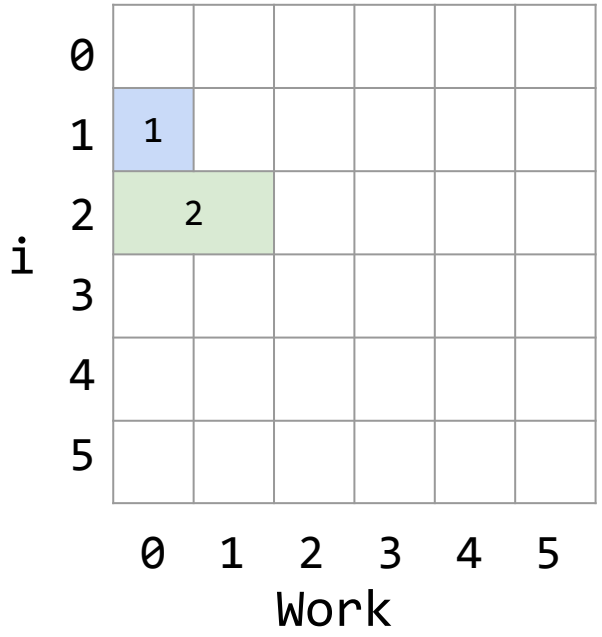

```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        //i units of work
    }
}
```

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Units of work C(N):

N	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1																	

Loops Example



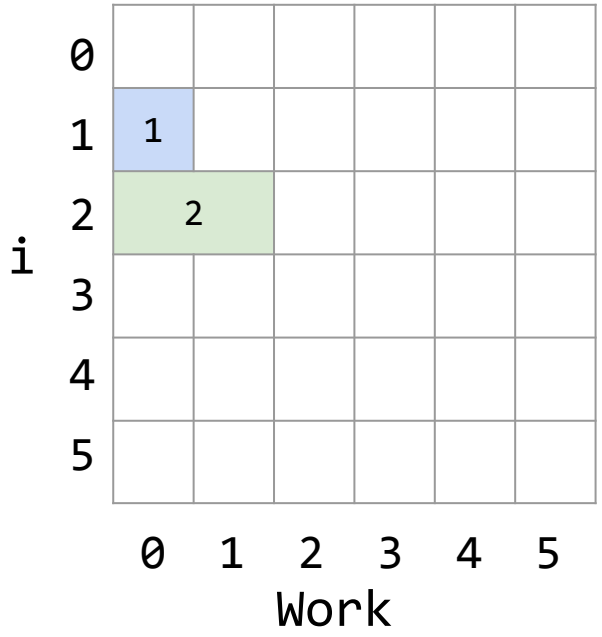

```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        //i units of work
    }
}
```

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Units of work C(N):

N	**1**	**2**	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1	3																

Loops Example




```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        //i units of work
    }
}
```

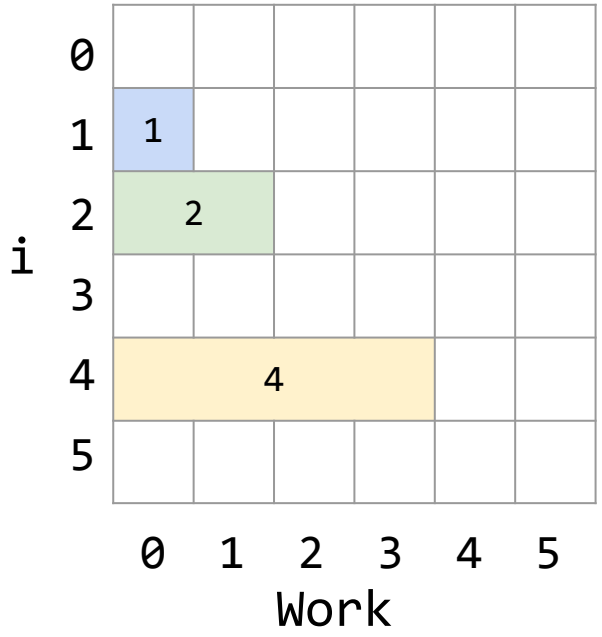
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Units of work C(N):

N	**1**	**2**	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1	3	3															

N=3 doesn't do anything extra

Loops Example



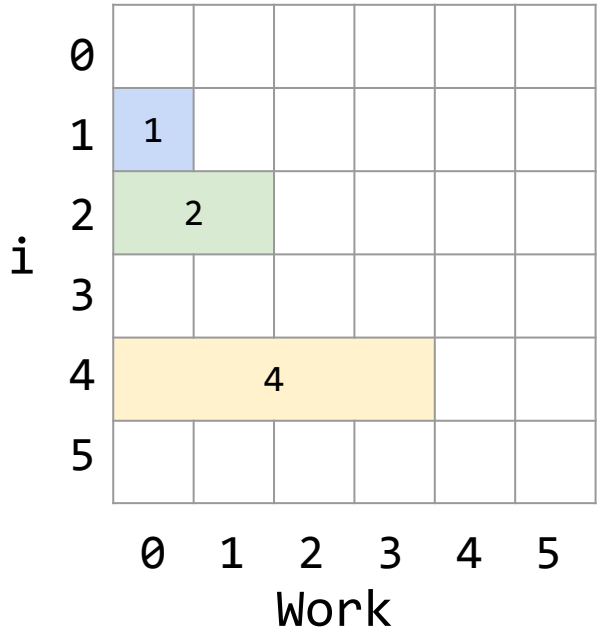

```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        //i units of work
    }
}
```

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Units of work C(N):

N	**1**	**2**	3	**4**	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1	3	3	7														

Loops Example




```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        //i units of work
    }
}
```

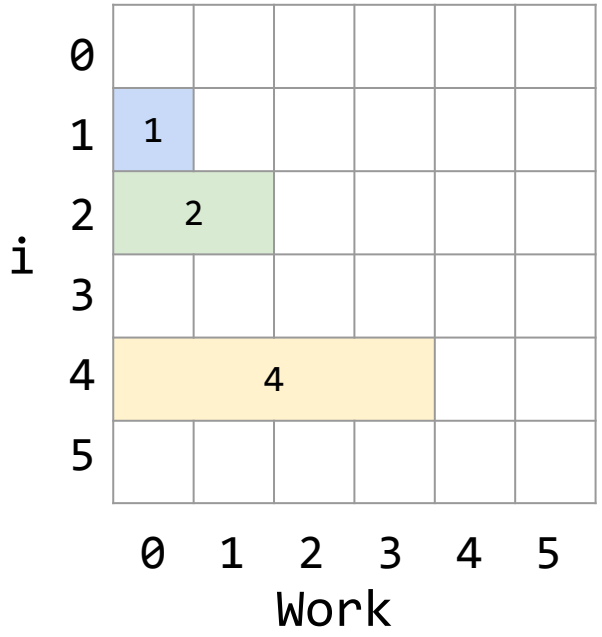
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Units of work C(N):

N	**1**	**2**	3	**4**	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1	3	3	7	7	7	7											

N=4,5,6,7 all print 7 times

Loops Example




```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        //i units of work
    }
}
```

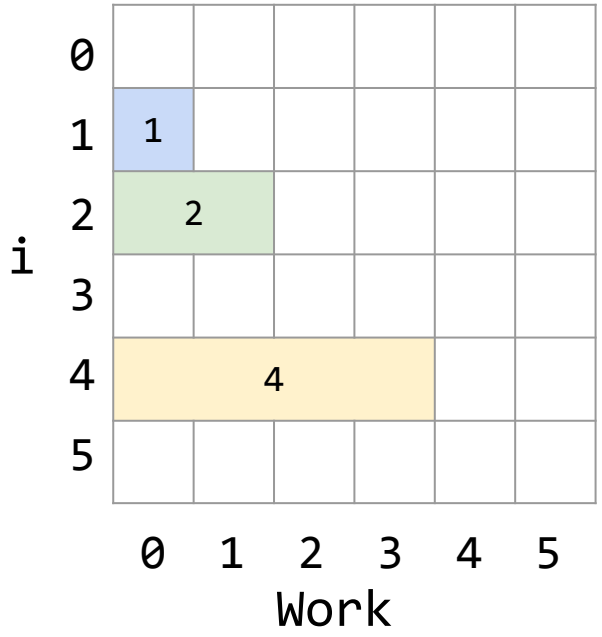
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Units of work C(N):

N	**1**	**2**	3	**4**	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1	3	3	7	7	7	7											

N=4,5,6,7 all print 7 times

Loops Example




```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        //i units of work
    }
}
```

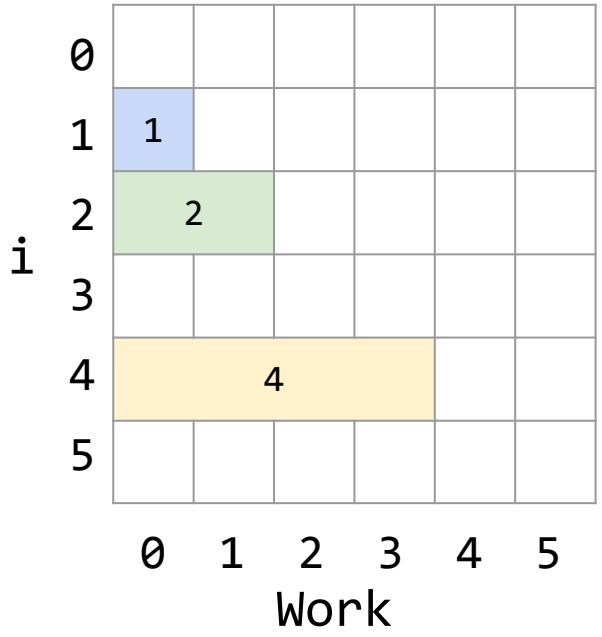
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Units of work C(N):

N	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	16	17	18
C(N)	1	3	3	7	7	7	7	15	15	15	15	15	15	15	15			

These N all print 15 times

Loops Example




```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        //i units of work
    }
}
```

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

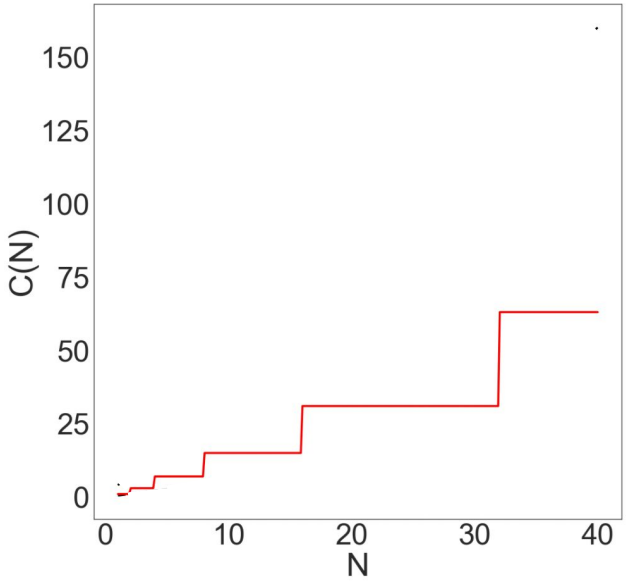
Units of work C(N):

N	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18
C(N)	1	3	3	7	7	7	7	15	15	15	15	15	15	15	15	31	31	31

C(N) = 1 + 2 + 4 + … + N, if N is a power of 2

Loops Example

We're trying to find the order of growth of C(N):



Units of work C(N):

N	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18
C(N)	1	3	3	7	7	7	7	15	15	15	15	15	15	15	15	31	31	31

C(N) = 1 + 2 + 4 + … + N, if N is a power of 2

Loops Example:

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

A. 1
B. log N
C. N
D. N log N
E. N^2
F. Other

```
public static void printParty(int N) {
  for (int i = 1; i<=N; i = i * 2) {
    for (int j = 0; j < i; j += 1) {
      System.out.println("hello");
      int ZUG = 1 + 1;
```

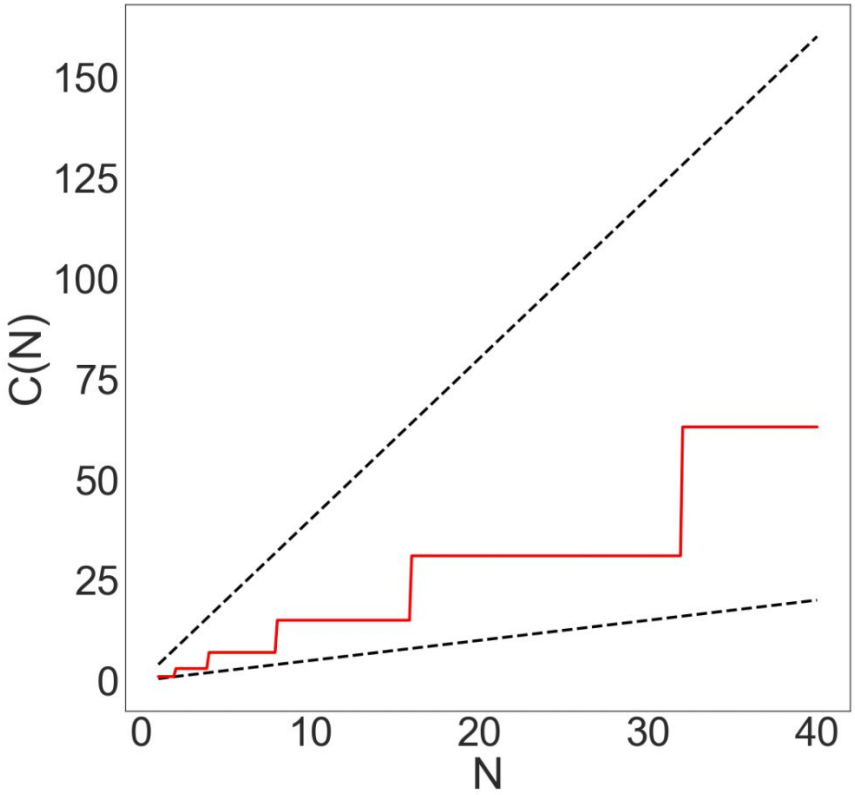
Cost model C(N):

N	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18
C(N)	1	3	3	7	7	7	7	15	15	15	15	15	15	15	15	31	31	31

C(N) = 1 + 2 + 4 + … + N, if N is a power of 2

Loops Example 2 [attempt #3] (no yellkey)

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.



The "peaks" of C(N) are when $N = 2^k$.

Total = $1+2+4+...+N \in \Theta(N)$

The "troughs" of C(N) are when $N = 2^k-1$.

Total = $1+2+4+...+(N+1)/2 \in \Theta((N+1)/2)$
$\in \Theta(N)$

A. 1
B. log N
C. N
D. N log N
E. N^2
F. Something else

Amortized Analysis

Lecture 15, CS61B, Fall 2024

Summations

Analyzing Programs

- Nested For Loops
- **Amortized Analysis**
- Recursive Analysis

Mergesort

Surely no function does this, right?

Let's play with this function a bit

```
public static void printParty(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        //i units of work
    }
}
```

Surely no function does this, right?

Add Θ(N) work: Total runtime is still Θ(N)

```
public static void printParty(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        //i units of work
    }
    for (int i = 1; i <= n; i++) {
        //1 unit of work
    }
}
```

Surely no function does this, right?

Combine the for loops. No asymptotic change in work done

```
public static void printParty(int n) {
    for (int i = 1; i <= n; i++) {
        //1 unit of work
        if(i is a power of 2) {
            //i units of work
        }
    }
}
```

Surely no function does this, right?

Put things in separate functions. Changes an O(1) thing to more O(1) things, so no runtime difference.

```
public static void printParty(int n) {
    for (int i = 1; i <= n; i++) {
        foo();
    } }
public static void foo() {
    if(i is a power of 2) {
            bar(i);
    }
    // 1 unit of work
}
public static void bar(i) {
    //i units of work
}
```

Surely no function does this, right?

Rename functions and define what we do in the commented code. No change in runtime.

```
public void addMany(int n) {
    for (int i = 0; i < n; i++) {
        addLast(1);
    } }
public void addLast(int value) {
    if(this.length == arr.length) {
        resize(this.length * 2);
    } //Happens every time length is 2^k
    //addLast code takes 1 unit of work
}
public void resize(int i) {
    //resizing takes i units of work
}
```

Why geometric resizing is faster

When we discussed ArrayLists, we handwaved why geometric resizing is better than linear resizing. Now that we know asymptotics, we can finally prove this.

```
public void addLast(int x) {
    if (size == items.length) {
        resize(size + RFACTOR);
    }
    items[size] = x;
    size += 1;
}
```

After N addLasts, runtime is $\Theta(N)$ →

← After N addLasts, runtime is $\Theta(N^2)$

```
public void addLast(int x) {
    if (size == items.length) {
        resize(size * RFACTOR);
    }
    items[size] = x;
    size += 1;
}
```

Why geometric resizing is faster

Even though the worst-case resize is still Θ(N), they happen so infrequently with geometric resizing that we get Θ(1) runtime on average *regardless of how we order List operations*.

```
public void addLast(int x) {
    if (size == items.length) {
        resize(size + RFACTOR);
    }
    items[size] = x;
    size += 1;
}
```

Each addLast takes on average Θ(N) time

Each addLast takes on average Θ(1) time

```
public void addLast(int x) {
    if (size == items.length) {
        resize(size * RFACTOR);
    }
    items[size] = x;
    size += 1;
}
```

Amortized Runtime

This is known as **Amortized Runtime**

- Any single operation may take longer, but if we use it over many operations, we're guaranteed to have a better average performance
- So amortized runtime gives a better estimate of how much time it takes to use something in practice

addLast is Θ(1) amortized

```
public void addLast(int x) {
    if (size == items.length) {
        resize(size * RFACTOR);
    }
    items[size] = x;
    size += 1;
}
```

Recursive Analysis

Lecture 15, CS61B, Fall 2024

Summations

Analyzing Programs

- Nested For Loops
- Amortized Analysis
- **Recursive Analysis**

Mergesort

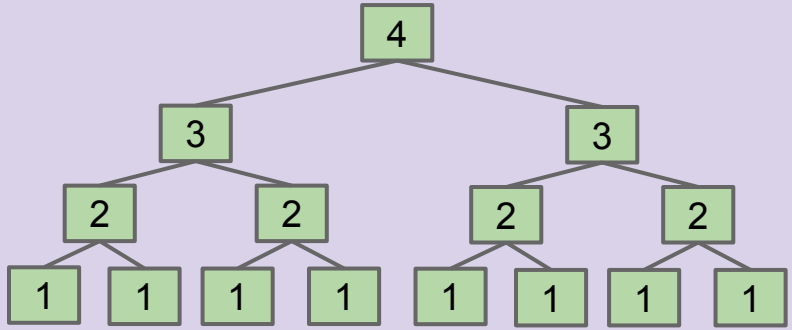
Recursion, Approach 1: Intuitive

Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1);
}
```

Using your intuition, give the order of growth of the runtime of this code as a function of N?

A. 1
B. log N
C. N
D. N^2
E. 2^N



Recursion, Approach 1: Intuitive

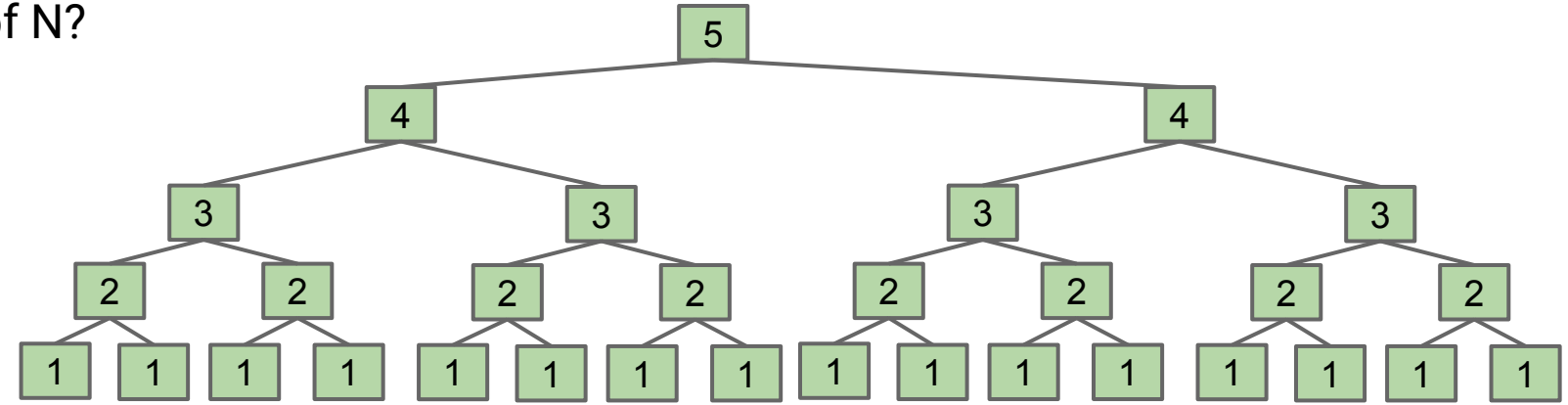
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1);
}
```

2^N: Every time we increase N by 1, we double the work!

Using your intuition, give the order of growth of the runtime of this code as a function of N?

A. 1
B. log N
C. N
D. N^2
E. 2^N



Recursion, Approach 2: Exact Counting

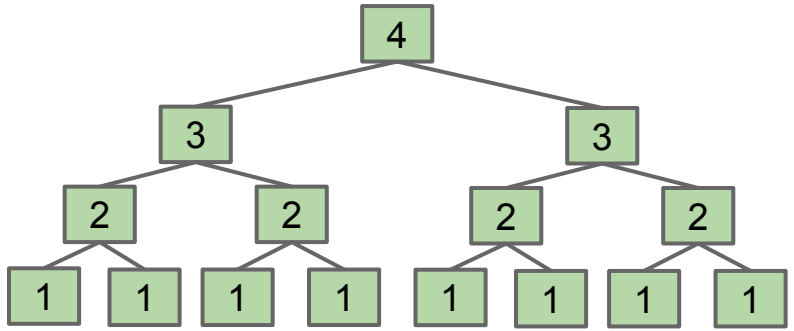
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1);
}
```

Another approach: Count number of calls to f3, given by C(N).

Each function call does a constant amount of work (not counting recursive calls), so $C(N) \in \Theta(R(N))$

- C(1) = 1
- C(2) = 1 + 2
- C(3) = 1 + 2 + 4



Recursion, Approach 2: Exact Counting: http://yellkey.com/happy

Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

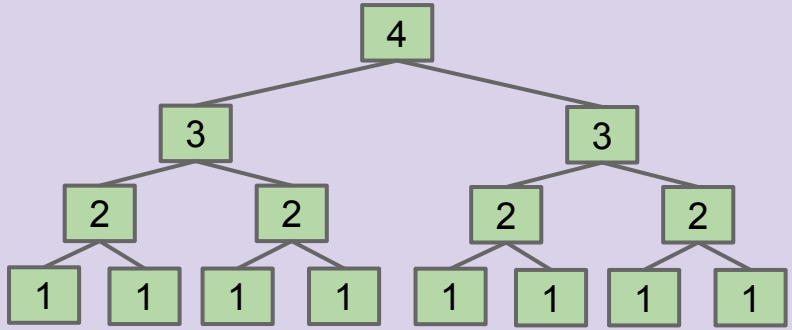
```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1);
}
```

Another approach: Count number of calls to f3, given by C(N).

- C(3) = 1 + 2 + 4
- C(N) = 1 + 2 + 4 + … + ???

What is the final term of the sum?

A. N
B. 2^N
C. 2^N-1
D. 2^{N-1}
E. $2^{N-1}-1$



Recursion, Approach 2: Exact Counting

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

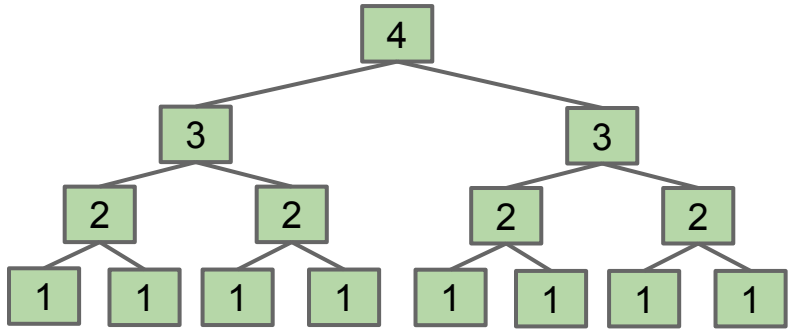
```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1);
}
```

Another approach: Count number of calls to f3, given by C(N).

- C(3) = 1 + 2 + 4
- C(N) = 1 + 2 + 4 + … + ???

What is the final term of the sum?

D. 2^{N-1}



Recursion, Approach 2: Exact Counting

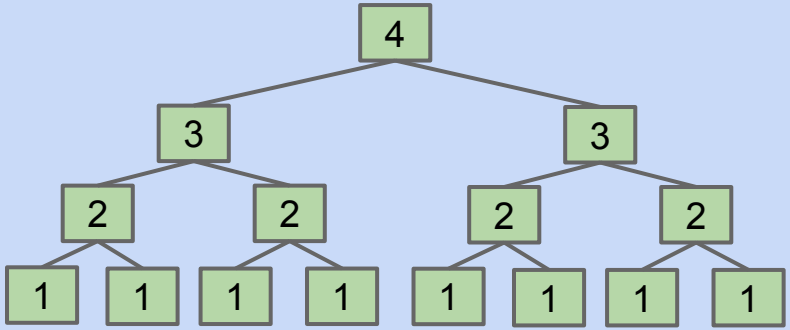
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1);
}
```

Another approach: Count number of calls to f3, given by C(N).

- $C(N) = 1 + 2 + 4 + \ldots + 2^{N-1}$

Give a simple expression for C(N).



Recursion, Approach 2: Exact Counting

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

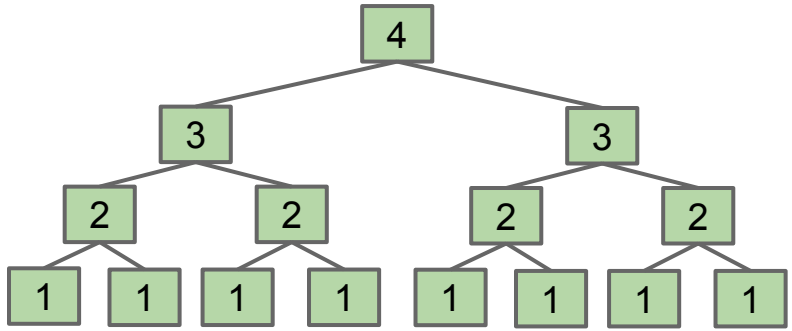
```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1);
}
```

Another approach: Count number of calls to f3, given by C(N).

- $C(N) = 1 + 2 + 4 + \ldots + 2^{N-1}$

Give a simple expression for C(N).

- $C(N) = 2(2^{N-1}) - 1$
- $C(N) = 2^N - 1$



Recursion, Approach 2: Exact Counting

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

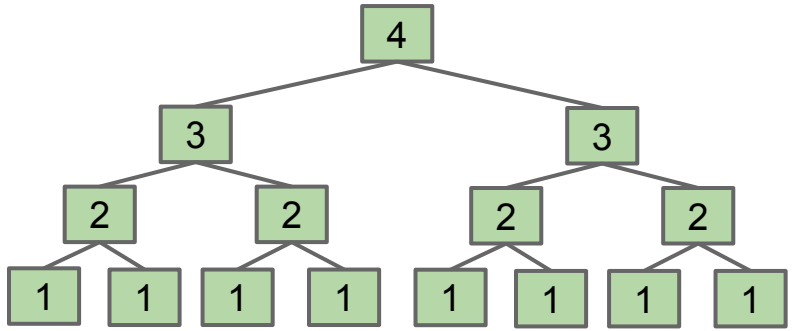
```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1);
}
```

Another approach: Count number of calls to f3, given by C(N).

- $C(N) = 1 + 2 + 4 + \ldots + 2^{N-1}$
- Solving, we get $C(N) = 2^N - 1$

Since work during each call is constant:

- $R(N) = \Theta(2^N)$



Recursion, A minor change

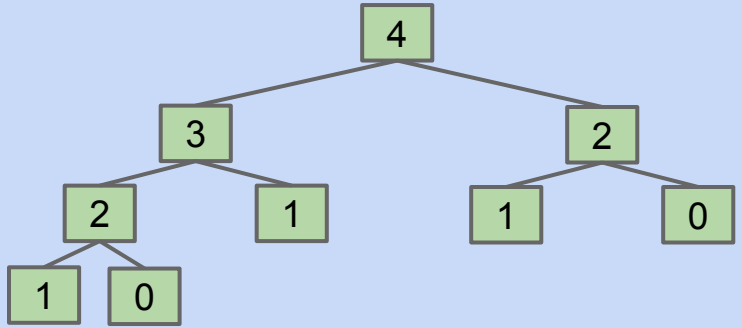
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
public static int fib(int n) {
    if (n <= 1)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

What happens if we make this tiny change?

- C(0) = 1
- C(1) = 1
- C(2) = 1+1+1
- C(3) = 1+1+2+1
- ?????

Give a simple expression for C(N).



Recursion, A minor change (Out of Scope)

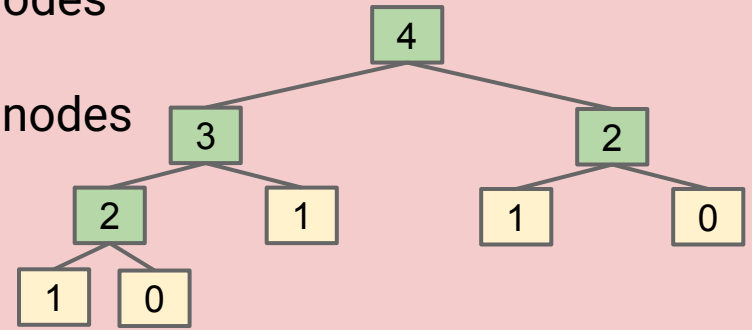
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
public static int fib(int n) {
    if (n <= 1)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

For this one, we'll have to be a bit more creative:

- fib(n) returns the nth Fibonacci number F_n
- In the tree on the right, there are F_n yellow nodes
  - Why? Each leaf adds 1 to the final sum
- In the tree on the right, there are F_n-1 green nodes
  - Why? To sum k 1s, we do k-1 +s

$C(N)$ = # yellow + # green = $2(F_N)-1 = \Theta(F_N)$



Recursion, A minor change (Out of Scope)

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
public static int fib(int n) {
    if (n <= 1)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

$C(N)$ = # yellow + # green = $2(F_N)-1 = \Theta(F_N)$

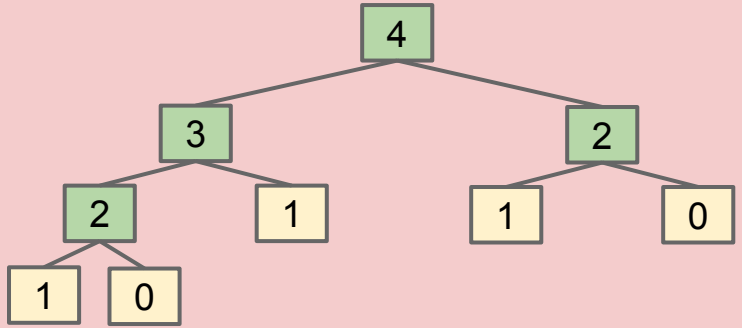
If you do enough math, you find that $F_N \in \Theta(\varphi^N)$

Where $\varphi = (1+\sqrt{5})/2 \approx 1.618$

Each function call does 1 unit of work

So $R(N) = \Theta(1.618^N)$

In conclusion: **There is no Magic Shortcut for Asymptotic Analysis**



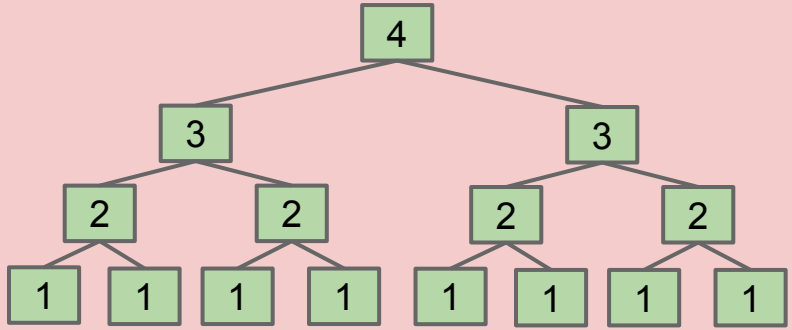
Recursion, Approach 3: Recurrence Relations (Out of Scope for 61B)

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1)
}
```

Third approach: Count number of calls to f3, given by a "recurrence relation" for C(N).

- C(1) = 1
- C(N) = 2C(N-1) + 1



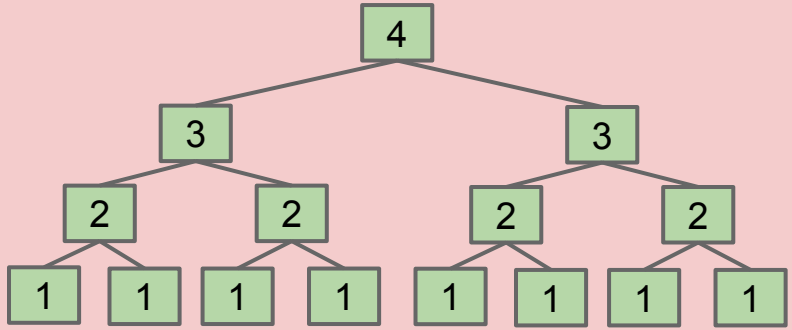
Recursion, Approach 3: Recurrence Relations (Out of Scope for 61B)

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1)
}
```

Third approach: Count number of calls to f3, given by a "recurrence relation" for C(N).

- C(1) = 1
- C(N) = 2C(N-1) + 1



More technical to solve. Won't do this in our course. See next slide for solution.

Recursion, Approach 3: Recurrence Relations (Out of Scope for 61B)

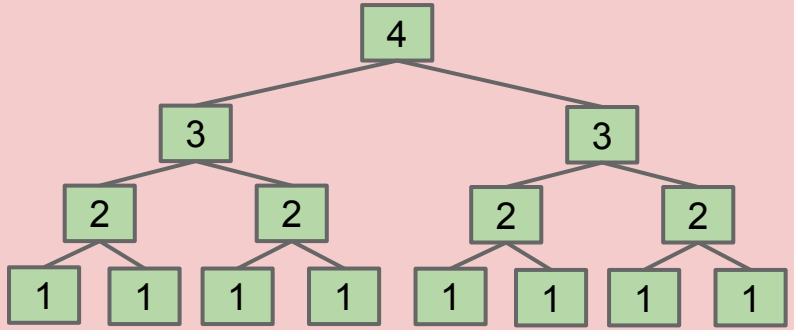
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

```
public static int f3(int n) {
    if (n <= 1)
        return 1;
    return f3(n-1) + f3(n-1)
}
```

This approach not covered in class. Provided for those of you who want to see a recurrence relation solution.

Third approach: Count number of calls to f3, given by a "recurrence relation" for C(N).

$$
\begin{aligned}
C(1) &= 1 \\
C(N) &= 2C(N-1)+1 \\
&= 2(2C(N-2)+1)+1 \\
&= 2(2(2C(N-2)+1)+1)+1 \\
&= 2(\cdots 2\cdot 1+1)+1)+\cdots 1 \\
&= \underbrace{2(\cdots 2}_{N-1})\cdot 1+1)+\cdots 1 \\
&= 2^{N-1}+2^{N-2}+\cdots+1 = 2^N-1 \in \Theta(2^N)
\end{aligned}
$$



Mergesort

Lecture 15, CS61B, Fall 2024

Summations

Analyzing Programs

- Nested For Loops
- Amortized Analysis
- Recursive Analysis

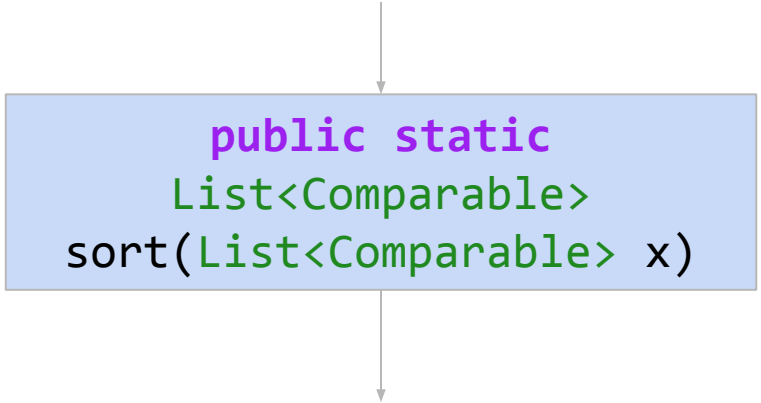
Mergesort

Sorting

Along with matrix multiplication, sorting is one of the problems that pops up most often in asymptotic analysis.

- Given a list of Comparables, return them in sorted order
  - Assumes the comparison method has certain properties, and runs in Θ(1) time.

```
x = List.of({“he”, “is”, “the”, “agoyatis”, “of”, “mr.”, “conchis”})
```




```
x = {“agoyatis”, “conchis”, “he”, “is”, “mr.”, “of”, “the”}
```

Mergesort Pseudocode

Mergesort is a recursive way to sort a list:

- Split the list into two parts
- Sort the two lists individually
- Merge the two lists together

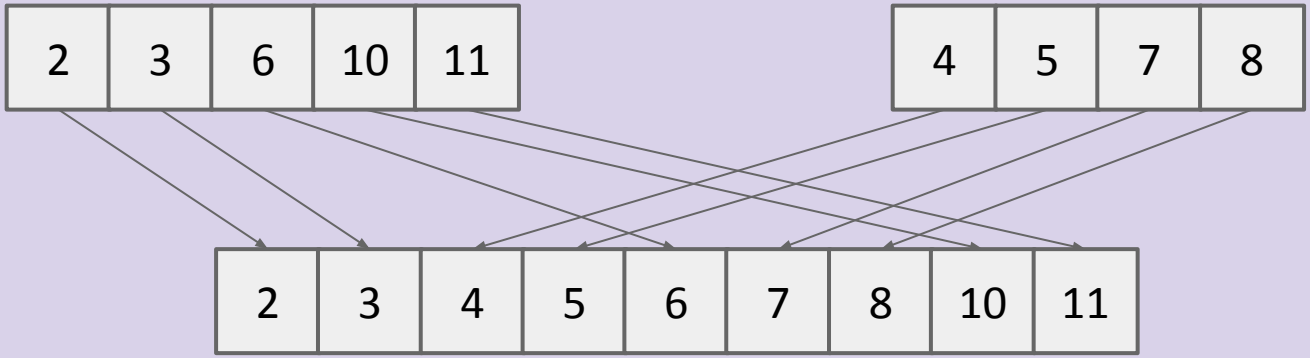
```
static List<Comparable> sort(List<Comparable> x)
    List<Comparable> firsthalf = sort(x.sublist(0, x.size()/2));
    List<Comparable> secondhalf = sort(x.sublist(x.size()/2, x.size()));
    return merge(firsthalf, secondhalf);
}
```

The Merge Operation

Given two sorted arrays, the merge operation combines them into a single sorted array by successively copying the smallest item from the two arrays into a target array.

Merging Demo (Link)

Merge Runtime



How does the runtime of merge grow with N, the total number of items?

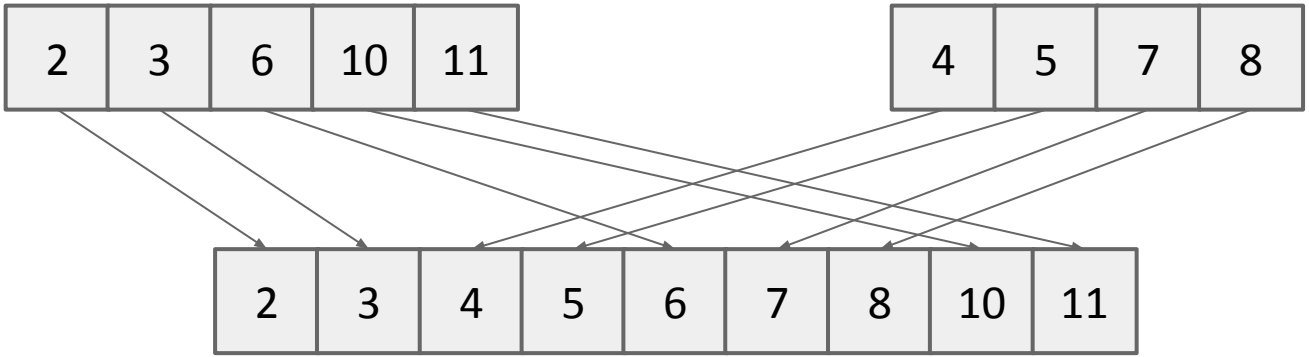
A. $\Theta(1)$

B. $\Theta(\log N)$

C. $\Theta(N)$

D. $\Theta(N^2)$

Merge Runtime



How does the runtime of merge grow with N, the total number of items?

C. Θ(N). Why? Θ(1) time per element in the merged list, and the merged list has exactly N items

Determining Mergesort Runtime

```
static List<Comparable> sort(List<Comparable> x)
    List<Comparable> firsthalf = sort(x.sublist(0, x.size()/2));
    List<Comparable> secondhalf = sort(x.sublist(x.size()/2, x.size()));
    return merge(firsthalf, secondhalf);
}
```

Since we don't care about the list itself, let's simplify our code a bit

- Our runtime should be in terms of x.size(), so let's let int n = x.size()
- merge takes Θ(n) time, so let's replace that with "n units of work"

Determining Mergesort Runtime

```
static void sortRuntime(int n)
    sortRuntime(n/2);
    sortRuntime(n/2);
    //n units of work
}
```

Since we don't care about the list itself, let's simplify our code a bit

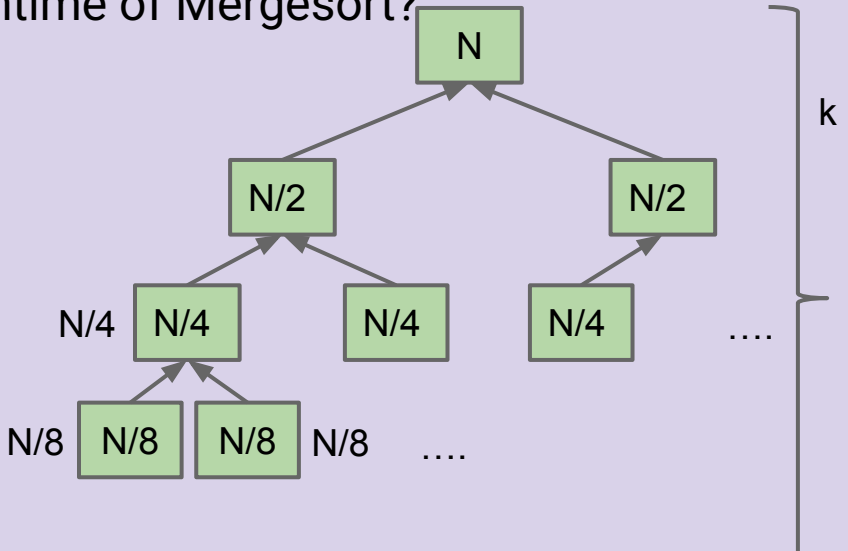
- Our runtime should be in terms of x.size(), so let's let int n = x.size()
- merge takes Θ(n) time, so let's replace that with "n units of work"

Example 5: Mergesort Order of Growth

```
static void sortRuntime(int n)
    sortRuntime(n/2);
    sortRuntime(n/2);
    //n units of work
}
```

For an array of size N, what is the worst case runtime of Mergesort?

A. Θ(1)
B. Θ(log N)
C. Θ(N)
D. Θ(N log N)
E. Θ(N^2)



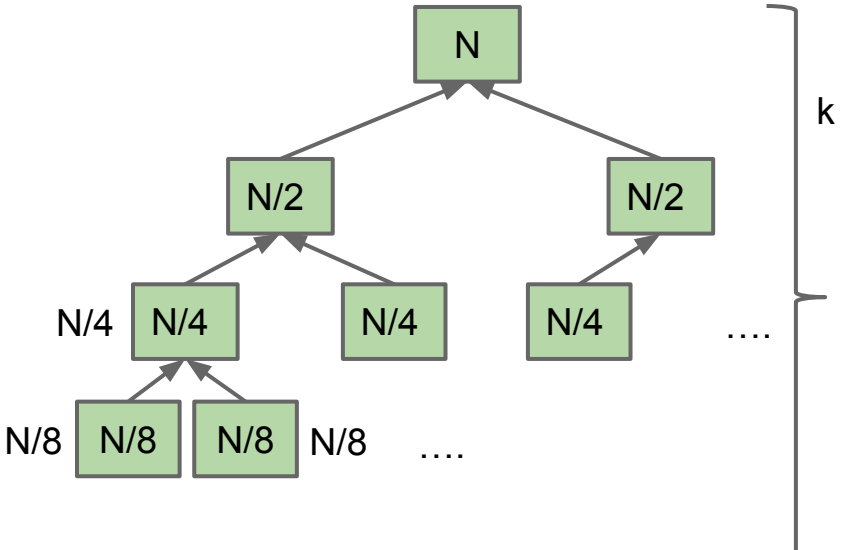
Example 5: Mergesort Order of Growth

Mergesort has worst case runtime = Θ(N log N).

- Every level has N units of work.
  - Top level takes N units of work.
  - Next level takes N/2 + N/2 = N units of work.
  - One more level down: N/4 + N/4 + N/4 + N/4 = N.
- Thus, total runtime is Nk, where k is the number of levels.
  - How many levels? Goes until we get to size 1.
  - $k = \log_2(N)$.
- Overall runtime is Θ(N log N).

Exact count explanation is tedious.

- Omitted here.



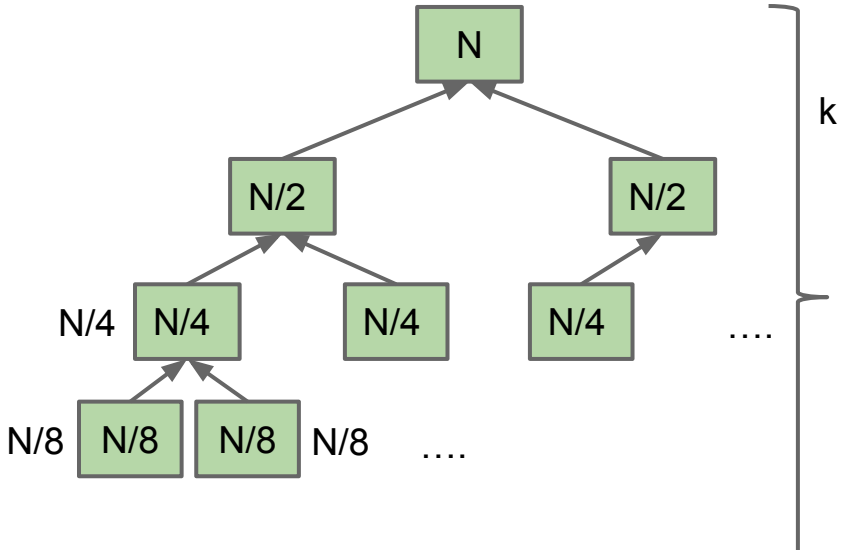
Mergesort using Recurrence Relations (Extra)

C(N): Number of calls to mergesort + number of array writes.

$$C(N) = \begin{cases} 1 & : N < 2 \\ 2C(N/2) + N & : N \geq 2 \end{cases}$$

$$\begin{aligned} C(N) &= 2(2C(N/4) + N/2) + N \\ &= 4C(N/4) + N + N \\ &= 8C(N/8) + N + N + N \\ &= N \cdot 1 + \underbrace{N + N + \ldots + N}_{k = \lg N} \\ &= N + N \lg N \in \Theta(N \lg N) \end{aligned}$$

Only works for $N=2^k$. Can be generalized at the expense of some tedium by separately finding Big O and Big Omega bounds.



Using Sorting as a Tool

Recall from Lecture 11 the dup functions, which checked if a **sorted** array contained any duplicates.

- dup1 took $\Theta(N^2)$ runtime, while dup2 took $\Theta(N)$ runtime

What if the input wasn't sorted?

```
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

dup1

dup2

```
public static boolean dup2(int[] A) {
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

Using Sorting as a Tool

What if the input wasn't sorted?

- dup1 still works normally, but it takes $\Theta(N^2)$ runtime
- dup2 no longer works… Can we fix it?

```
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

dup1

dup2

```
public static boolean dup2(int[] A) {
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

Using Sorting as a Tool

What if the input wasn't sorted?

- Solution: Sort A first!
- What's our new runtime?
  - Sorting took Θ(N log N) time, the rest of dup2 took Θ(N) time

```
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

dup1

dup2

```
public static boolean dup2(int[] A) {
  A = A.sort()
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

Using Sorting as a Tool

What if the input wasn't sorted?

- dup1 still works normally, but it takes $\Theta(N^2)$ runtime
- If we use sort as a **black box**, we can modify dup2 so it runs in $\Theta(N \log N)$ time!
- Can we do better? Yes, we can get $\Theta(N)$... once we get to hashing

```
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

dup1

dup2

```
public static boolean dup2(int[] A) {
  A = A.sort()
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

Linear vs. Linearithmic (N log N) vs. Quadratic

N log N is basically as good as N, and is vastly better than N^2.

- For N = 1,000,000, the log N is only 20.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

(from Algorithm Design: Tardos, Kleinberg)

Summary

Theoretical analysis of algorithm performance requires **careful thought**.

- There are **no magic shortcuts** for analyzing code.
- In our course, it's OK to do exact counting or intuitive analysis.
  - Know how to sum $1^k+2^k+...+N^k$ and $k^0+k^1+...+k^N$.
  - We won't be writing mathematical proofs in this class.
- Many runtime problems you'll do in this class resemble one of the five problems from today. See textbook, study guide, and discussion for more practice.
- This topic has one of the highest skill ceilings of all topics in the course, and is a modern research topic

Different solutions to the same problem, e.g. sorting, may have different runtimes.

- N^2 vs. N log N is an enormous difference.
- Going from N log N to N is nice, but not a radical change.

Once you prove runtime for one problem, you may be able to use it in other problems to speed things up!

Binary Search (Intuitive)

Lecture 15, CS61B, Fall 2024

Summations

Analyzing Programs

- Nested For Loops
- Amortized Analysis
- Recursive Analysis

Mergesort

Binary Search (If time allows)

- **Intuitive Approach**
- Exact Approach

Binary Search (demo: https://goo.gl/3VvJNw)

Trivial to implement?

- Idea published in 1946.
- First correct implementation in 1962.
  - Bug in Java's binary search discovered in 2006.

See Jon Bentley's book Programming Pearls.

See http://goo.gl/gQI0FN

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Binary Search (Intuitive)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find runtime in terms of N = hi - lo + 1 [i.e. # of items being considered]

- Intuitively, what is the order of growth of the worst case runtime?
  - A. 1
  - B. $\log_2 N$
  - C. N
  - D. $N \log_2 N$
  - E. 2^N

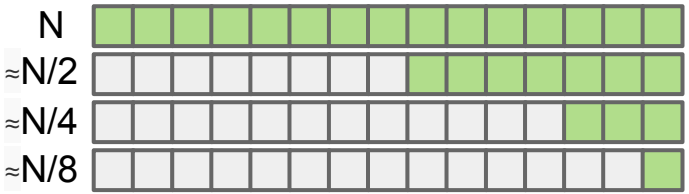
Binary Search (Intuitive)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find runtime in terms of N = hi - lo + 1 [i.e. # of items being considered]

- Intuitively, what is the order of growth of the worst case runtime?

  B. $\log_2 N$



Why? Problem size halves over and over until it gets down to 1.

- If C is number of calls to binarySearch, solve for $1 = N/2^C \rightarrow C = \log_2(N)$

Log Time Is Really Terribly Fast

In practice, logarithmic time algorithms have almost constant runtimes.

- Even for incredibly huge datasets, practically equivalent to constant time.

N	$\log_2 N$	Typical runtime (seconds)
100	6.6	1 nanosecond
100,000	16.6	2.5 nanoseconds
100,000,000	26.5	4 nanoseconds
100,000,000,000	36.5	5.5 nanoseconds
100,000,000,000,000	46.5	7 nanoseconds

Binary Search Exact (Bonus)

Lecture 15, CS61B, Fall 2024

Summations

Analyzing Programs

- Nested For Loops
- Amortized Analysis
- Recursive Analysis

Mergesort

Binary Search (If time allows)

- Intuitive Approach
- **Exact Approach**

This section is available as a pre-recorded video.

It's not ”out of scope” since it’s just another example problem using the same techniques used throughout the lecture.

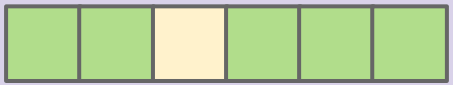
Binary Search (Exact Count): Not a Live Video (no yellkey)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find worst case runtime in terms of N = hi - lo + 1 [i.e. # of items]

Each call does constant work (w/o recursive calls)

N=6



- What is C(6), number of total calls for N = 6?

  A. 6
  B. 3
  C. $\log_2(6)=2.568$
  D. 2
  E. 1

Binary Search (Exact Count)

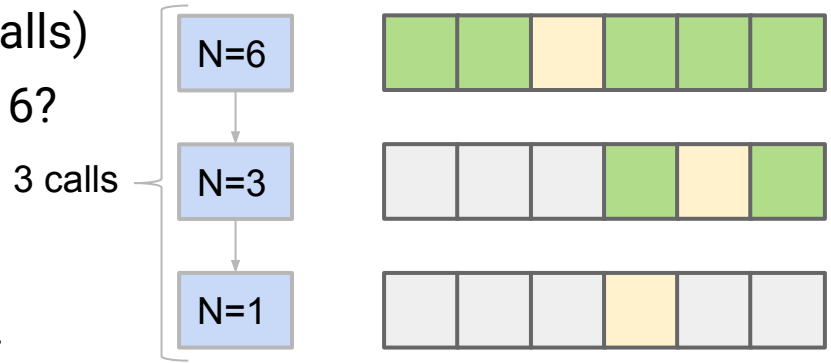
```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find worst case runtime in terms of N = hi - lo + 1 [i.e. # of items]

Each call does constant work (w/o recursive calls)

- What is C(6), number of total calls for N = 6?

  B. 3



Three total calls, where N = 6, N = 3, and N = 1.

Binary Search (Exact Count)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find worst case runtime in terms of N = hi - lo + 1 [i.e. # of items]

- Number of `binarySearch` calls.

N	1	2	3	4	5	6	7	8	9	10	11	12	13
C(N)	1					3							

N=1

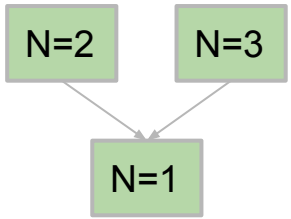
Binary Search (Exact Count)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find worst case runtime in terms of N = hi - lo + 1 [i.e. # of items]

- Number of `binarySearch` calls.

N	1	2	3	4	5	6	7	8	9	10	11	12	13
C(N)	1	2	2			3							



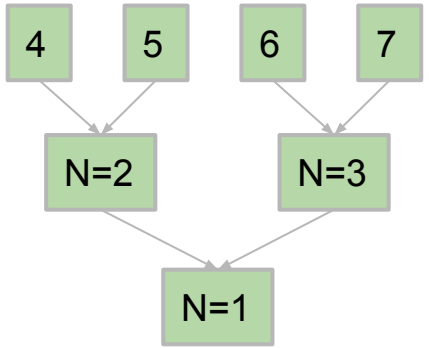
Binary Search (Exact Count)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find worst case runtime in terms of N = hi - lo + 1 [i.e. # of items]

- Number of `binarySearch` calls.

N	1	2	3	4	5	6	7	8	9	10	11	12	13
C(N)	1	2	2	3	3	3	3						



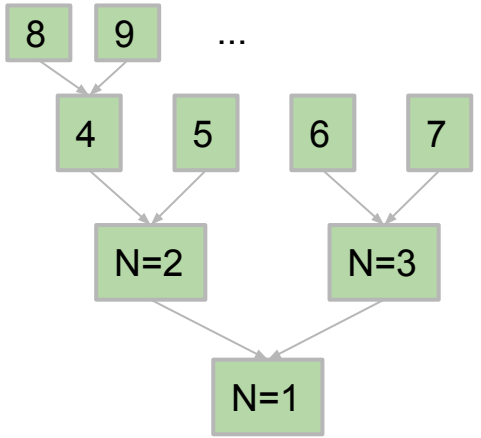
Binary Search (Exact Count)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find worst case runtime in terms of N = hi - lo + 1 [i.e. # of items]

- Number of `binarySearch` calls.

N	1	2	3	4	5	6	7	8	9	10	11	12	13
C(N)	1	2	2	3	3	3	3	4	4	4	4	4	4



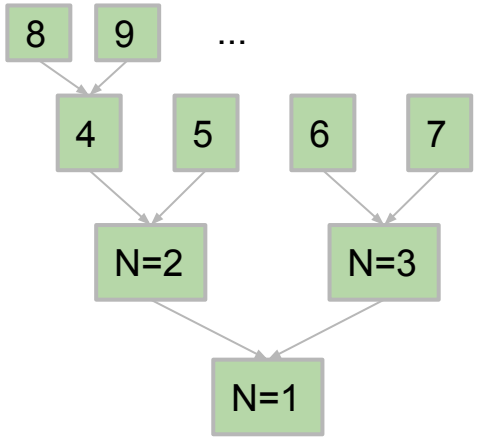
Binary Search (Exact Count)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find worst case runtime in terms of N = hi - lo + 1 [i.e. # of items]

- Number of `binarySearch` calls.

N	1	2	3	4	5	6	7	8	9	10	11	12	13
C(N)	1	2	2	3	3	3	3	4	4	4	4	4	4



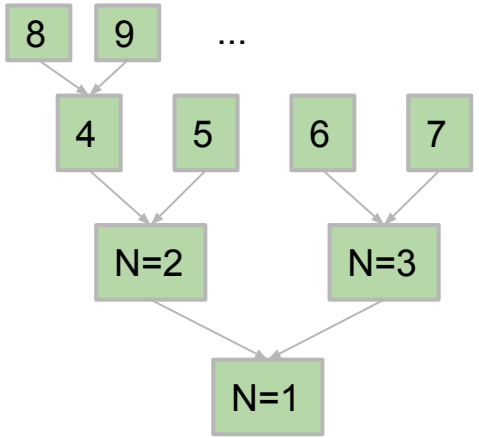
$$C(N) = \lfloor \log_2(N) \rfloor + 1$$

Binary Search (Exact Count)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find worst case runtime in terms of N = hi - lo + 1 [i.e. # of items]

- Number of `binarySearch` calls.
- $C(N) = \lfloor \log_2(N) \rfloor + 1$
- Since each call takes constant time, $R(N) = \Theta(\lfloor \log_2(N) \rfloor)$
  - This f(N) is way too complicated. Let's simplify.



Handy Big Theta Properties

Goal: Simplify $\Theta(\lfloor \log_2(N) \rfloor)$

For proof:
See online textbook exercises.

- Three handy properties to help us simplify:
  - $\lfloor f(N) \rfloor = \Theta(f(N))$ [the floor of f has same order of growth as f]
  - $\lceil f(N) \rceil = \Theta(f(N))$ [the ceiling of f has same order of growth as f]
  - $\log_P(N) = \Theta(\log_Q(N))$ [logarithm base does not affect order of growth]

$\lfloor \log_2(N) \rfloor = \Theta(\log N)$

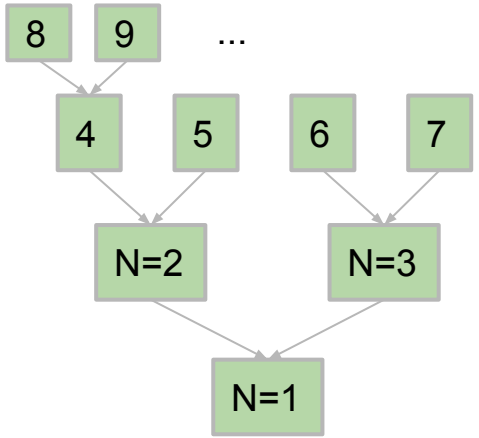
Since base is irrelevant, we omit from our big theta expression. We also omit the parenthesis around N for aesthetic reasons.

Binary Search (Exact Count)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Goal: Find worst case runtime in terms of N = hi - lo + 1 [i.e. # of items]

- Number of `binarySearch` calls.
- $C(N) = \lfloor \log_2(N) \rfloor + 1 = \Theta(\log N)$
- Since each call takes constant time, $R(N) = \Theta(\log N)$



... and we're done!

Binary Search (using Recurrence Relations)

```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return m;
}
```

Approach: Measure number of string comparisons for N = hi - lo + 1.

- C(0) = 0
- C(1) = 1
- C(N) = 1 + C((N-1)/2)

Can show that C(N) = Θ(log N). Beyond scope of class, so won't solve in slides.