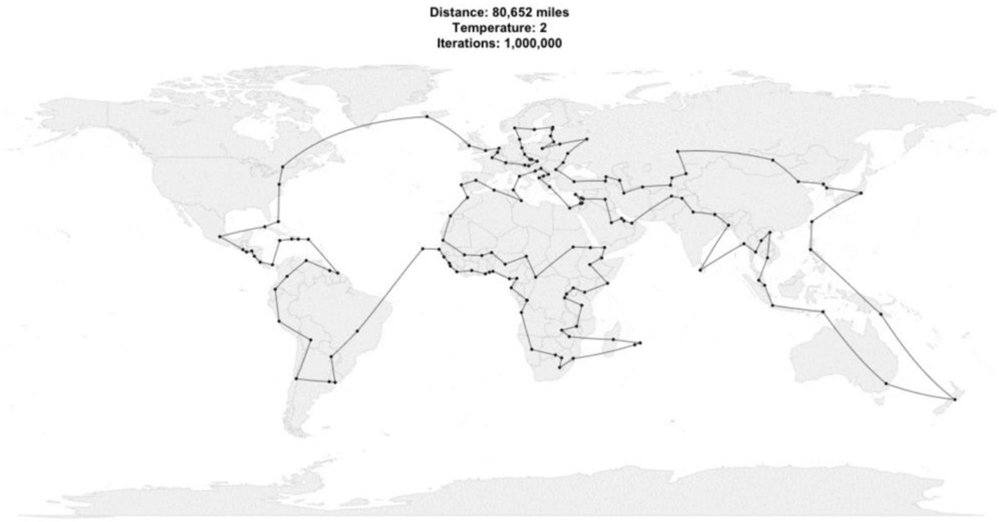


CS61B, 2019

Lecture 15: Asymptotics II: Analysis of Algorithms

- Review of Asymptotic Notation
- Examples 1-2: For Loops
- Example 3: A Basic Recurrence
- Example 4: Binary Search
- Example 5: Mergesort

Example 1/2:For Loops

Loops Example 1: Based on Exact Count

Find the order of growth of the worst case runtime of dup1.

N = 6

i \ j	0	1	2	3	4	5
0		==	==	==	==	==
1			==	==	==	==
2				==	==	==
3					==	==
4						==
5						

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
      if (A[i] == A[j])
         return true;
return false;
```

Worst case number of == operations:

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

operation	worst case count
==	$\Theta(N^2)$

Worst case runtime: $\Theta(N^2)$

Loops Example 1: Simpler Geometric Argument

Find the order of growth of the worst case runtime of dup1.

N = 6

i \ j	0	1	2	3	4	5
0		==	==	==	==	==
1			==	==	==	==
2				==	==	==
3					==	==
4						==
5						

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
      if (A[i] == A[j])
         return true;
return false;
```

Worst case number of == operations:

- Given by area of right triangle of side length N-1.
- Area is $\Theta(N^2)$.

operation	worst case count
==	$\Theta(N^2)$

Worst case runtime: $\Theta(N^2)$

Loops Example 2 [attempt #1]: http://yellkey.com/?

Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$. 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 2: Prelude to Attempt #2

i \ j	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

```java
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;
```

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

Cost model C(N), println("hello") calls:

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

Loops Example 2: Prelude to Attempt #2

i \ j	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

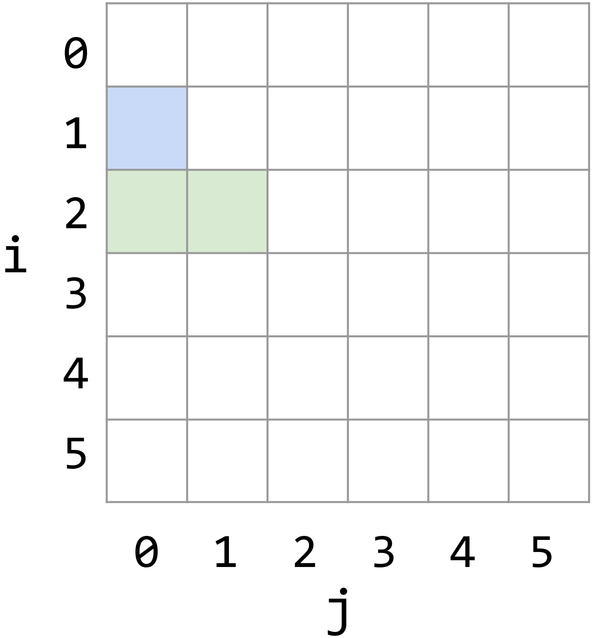
```java
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int ZUG = 1 + 1;
```

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

Cost model C(N), println("hello") calls:

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

Loops Example 2: Prelude to Attempt #2



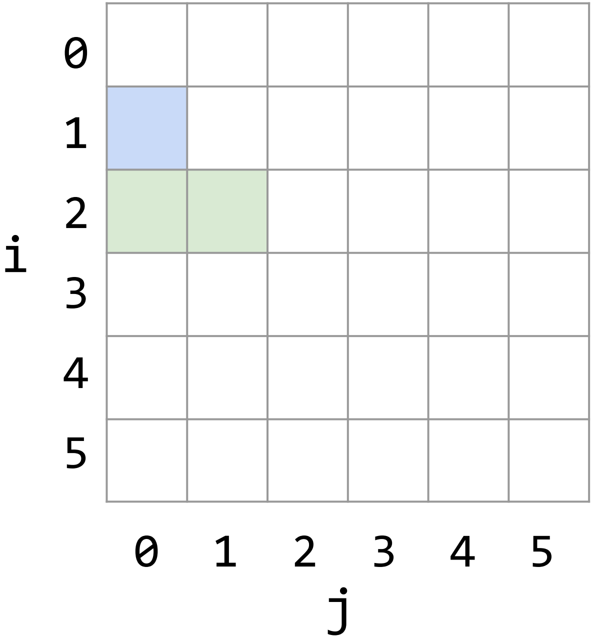

```java
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int ZUG = 1 + 1;
```

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

Cost model C(N), println("hello") calls:

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 2: Prelude to Attempt #2




```java
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;
```

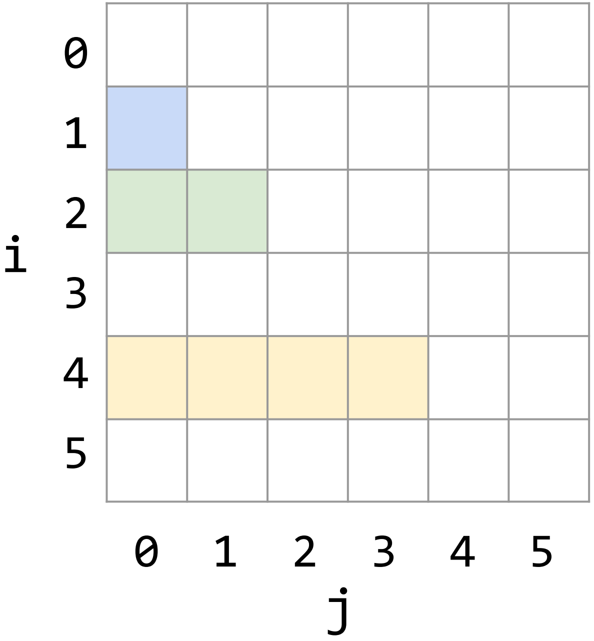
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

Cost model C(N), println("hello") calls:

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=2 and 3 both print 3 times

Loops Example 2: Prelude to Attempt #2



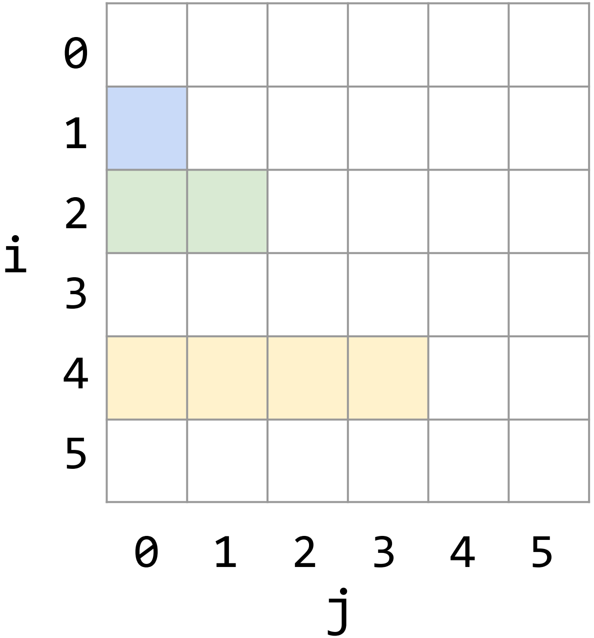

```java
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int ZUG = 1 + 1;
```

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

Cost model C(N), println("hello") calls:

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 2: Prelude to Attempt #2




```
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;
```

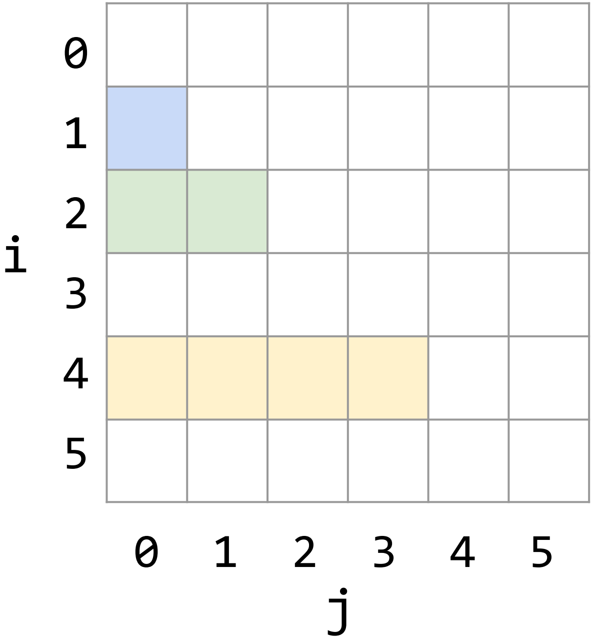
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

Cost model C(N), println(“hello”) calls:

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 2: Prelude to Attempt #2




```
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;
```

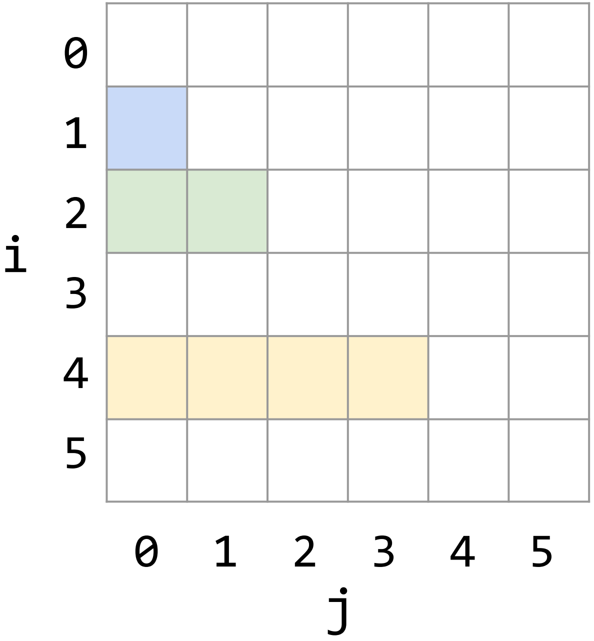
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

Cost model C(N), println("hello") calls:

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 2: Prelude to Attempt #2




```
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;
```

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

Cost model C(N), println("hello") calls:

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 #2]: http://yellkey.com/rangerange

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), println("hello") calls:

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

$C(N) = 1 + 2 + 4 + \ldots + N$, if N is a power of 2

Loops Example 2: Prelude to Attempt #3

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

N	C(N)	0.5 N	2N
1	1	0.5	2
4	1 + 2 + 4 = 7	2	14
7	1 + 2 + 4 = 7	3.5	14
8	1 + 2 + 4 + 8 = 15	4	16
27	1 + 2 + 4 + 8 + 16 = 31	13.5	54
185	… + 64 + 128 = **255**	92.5	370
715	… + 256 + 512 = **1023**	357.5	1430

Loops Example 2 [attempt #3]: http://yellkey.com/controlcontrol

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

N	C(N)	0.5 N	2N
1	1	0.5	2
4	7	2	14
7	7	3.5	14
8	15	4	16
27	31	13.5	54
185	**255**	92.5	370
715	**1023**	357.5	1430

```java
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;
```

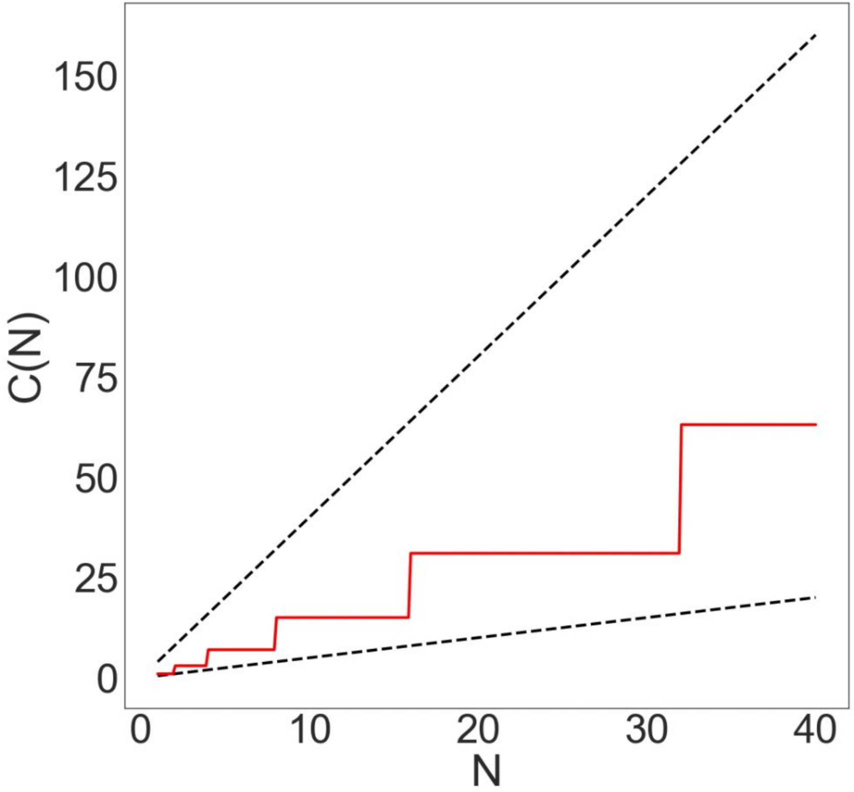
Cost model C(N), println("hello") calls:

- $R(N) = \Theta(1 + 2 + 4 + 8 + \ldots + N)$ if N is power of 2.

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

Loops Example 2 [attempt #3]: http://shoutkey.com/TBA

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



$R(N) = \Theta(1 + 2 + 4 + 8 + \ldots + N)$

$= \Theta(N)$

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

Can also compute exactly:

- $1 + 2 + 4 + \ldots + N = 2N - 1$
- Ex: If N = 8
  - LHS: 1 + 2 + 4 + 8 = 15
  - RHS: 2*8 - 1 = 15

Repeat After Me...

There is no magic shortcut for these problems (well… usually)

- Runtime 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:
  - $1 + 2 + 3 + \ldots + Q = Q(Q+1)/2 = \Theta(Q^2)$ ← Sum of First Natural Numbers (Link)
  - $1 + 2 + 4 + 8 + \ldots + Q = 2Q - 1 = \Theta(Q)$ ← Sum of First Powers of 2 (Link)

Where Q is a power of 2.

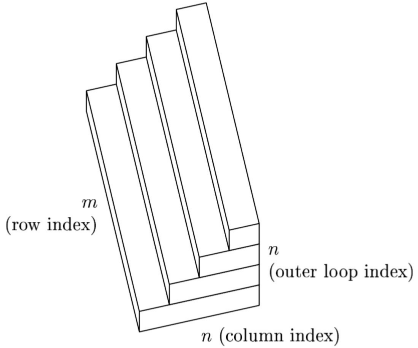
```
public static void printParty(int n) {
  for (int i = 1; i <= n; i = i * 2) {
    for (int j = 0; j < i; j += 1) {
      System.out.println("hello");
      int ZUG = 1 + 1;
```

Repeat After Me...

There is no magic shortcut for these problems (well… usually)

- Runtime 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:
  - $1 + 2 + 3 + \ldots + Q = Q(Q+1)/2 = \Theta(Q^2)$ ← Sum of First Natural Numbers (Link)
  - $1 + 2 + 4 + 8 + \ldots + Q = 2Q - 1 = \Theta(Q)$ ← Sum of First Powers of 2 (Link)
- 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.

Example 3: Recursion

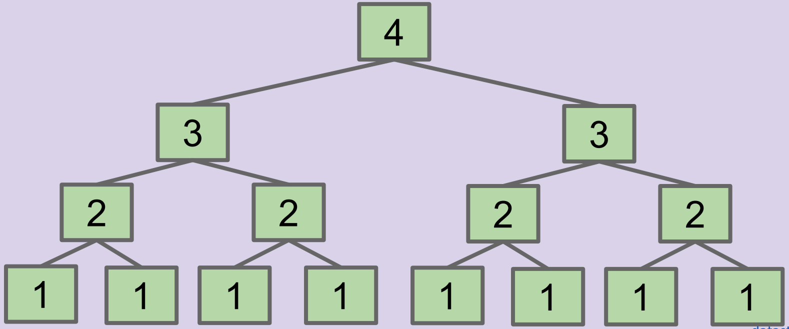
Recursion (Intuitive): http://yellkey.com/personal

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

```java
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 (Intuitive)

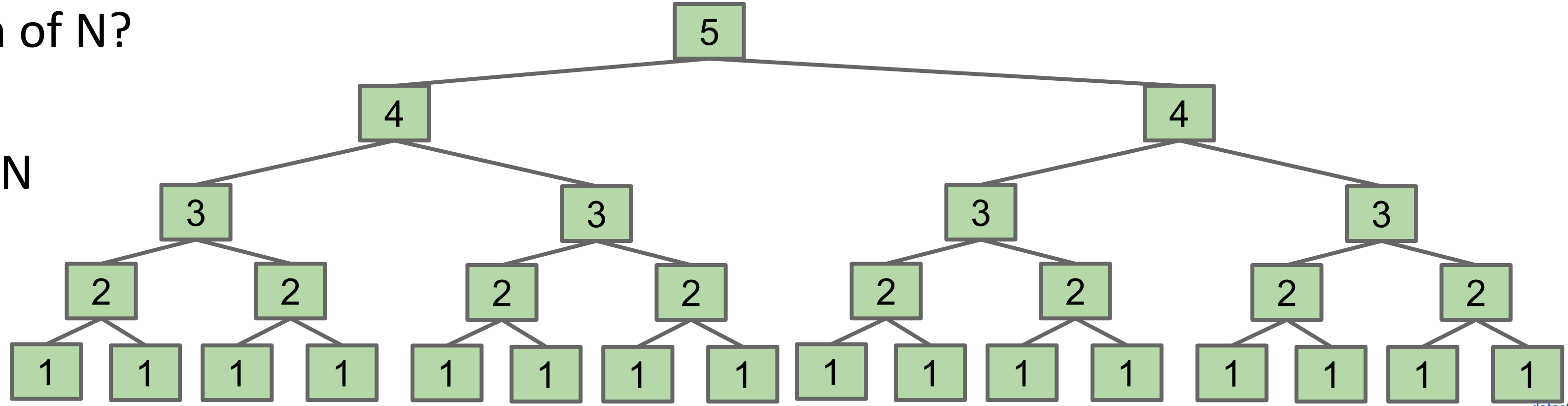
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
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



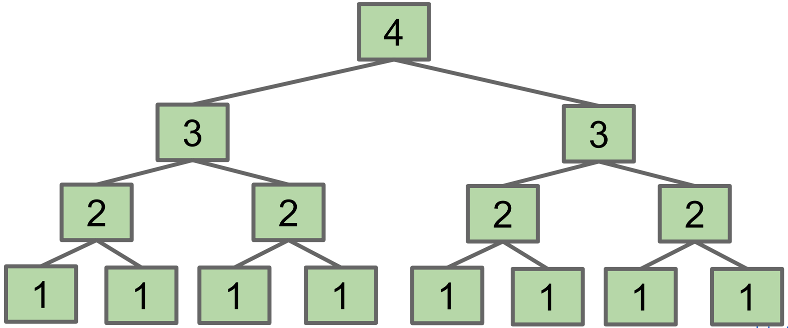
Recursion and Exact Counting

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



Recursion and Exact Counting: http://yellkey.com/similar

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

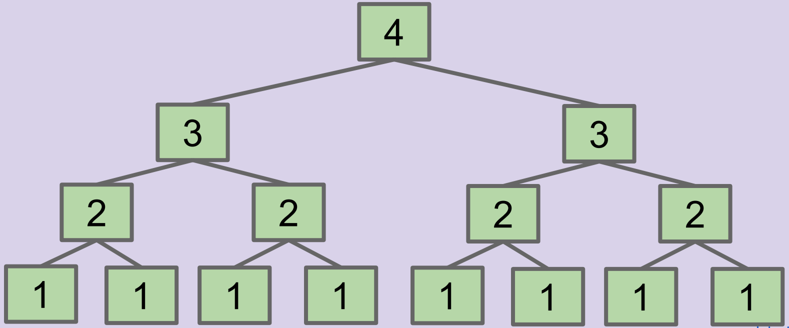
```
public static int f3(int n) {
   if (n <= 1)
       return 1;
   return f3(n-1) + f3(n-1);
}
```

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

- $C(3) = 1 + 2 + 4$
- $C(N) = 1 + 2 + 4 + \ldots + ???$

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 and Exact Counting

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

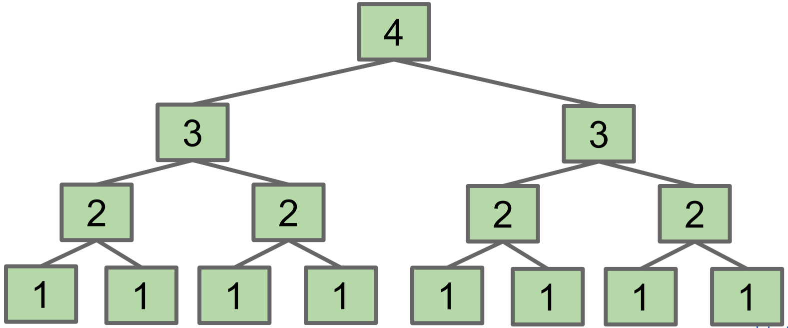
```
public static int f3(int n) {
   if (n <= 1)
       return 1;
   return f3(n-1) + f3(n-1);
}
```

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 and Exact Counting

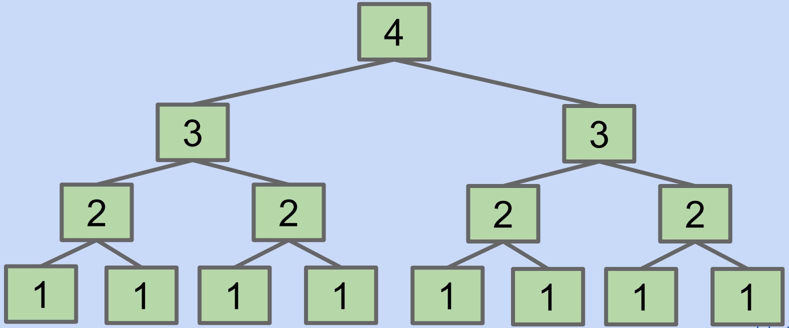
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

```
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).



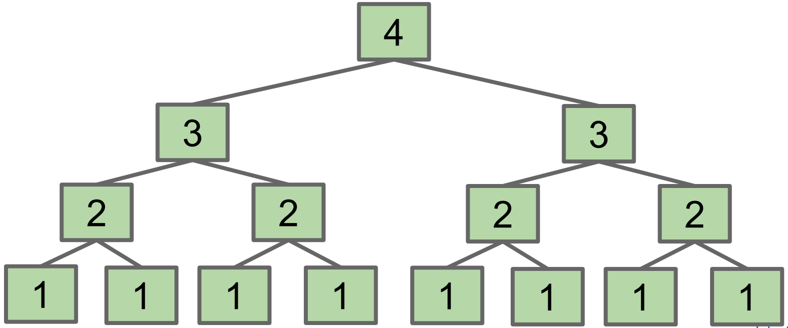
Recursion and Exact Counting

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).

- $C(N) = 2^N - 1$
- Why? It's the Sum of First Powers of 2.
  - See next slide for details.

Recursion and Exact Counting, Solving for C(N)

$$C(N) = 1 + 2 + 4 + 8 + \ldots + 2^{N-1}$$

We know that the Sum of the First Powers of 2 from before, i.e. as long as Q is a power of 2, then:

$$1 + 2 + 4 + 8 + \ldots + Q = 2Q - 1$$

Thus, since $Q = 2^{N-1}$, we have that:

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

Recursion and Exact Counting

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

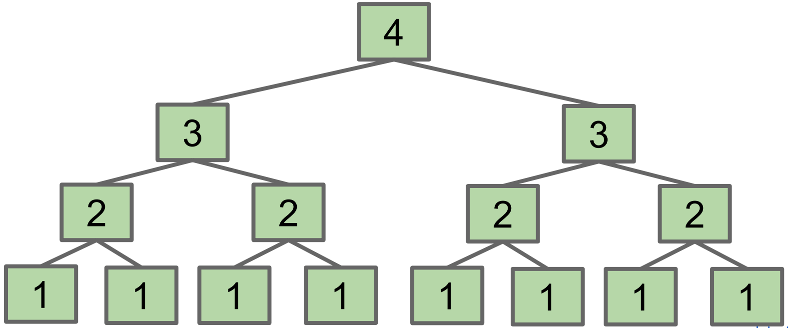
```
public static int f3(int n) {
   if (n <= 1)
       return 1;
   return f3(n-1) + f3(n-1);
}
```

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:

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



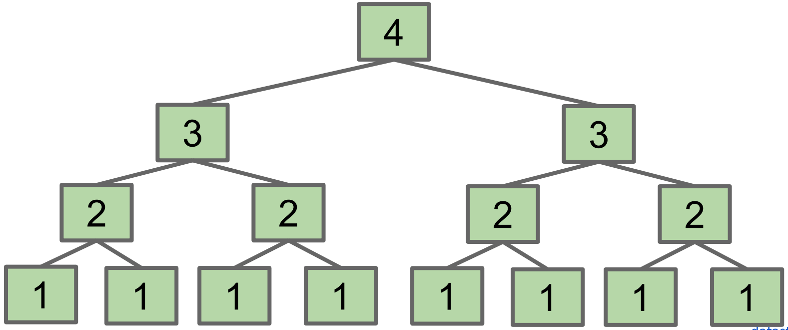
Recursion and Recurrence Relations

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)
}
```

A 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 and Recurrence Relations (Extra, Outside 61B Scope)

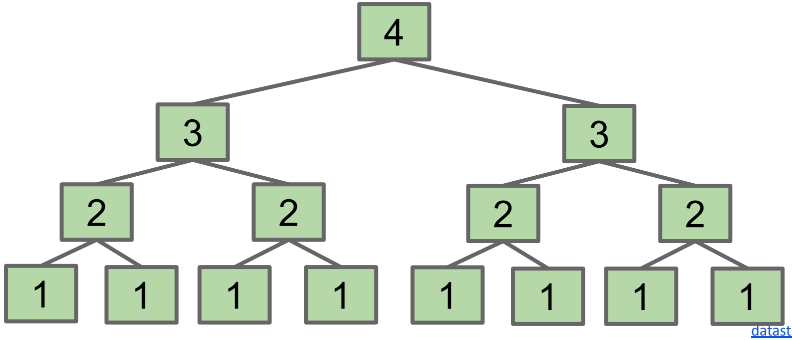
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

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

A 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 and Recurrence Relations (Extra, Outside 61B Scope)

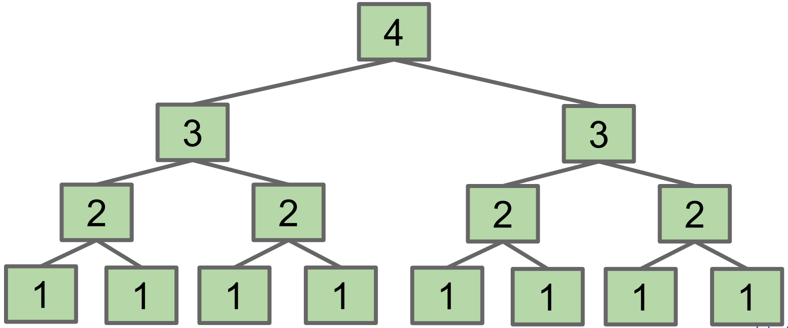
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

```java
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.

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

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



Example 4: Binary Search

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): http://yellkey.com/daughter

```java
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

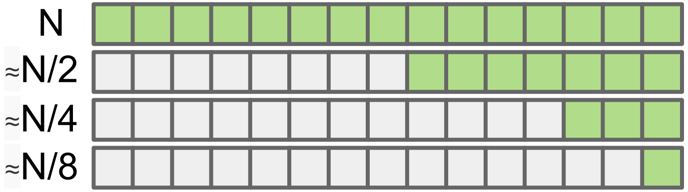
Binary Search (Intuitive)

```
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 $1 = N/2^C \rightarrow C = \log_2(N)$

Example 4: Binary Search Exact (Optional) (see web video)

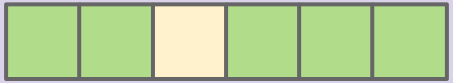
Binary Search (Exact Count): http://yellkey.com/enter

```java
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]

Cost model: Number of `binarySearch` 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)

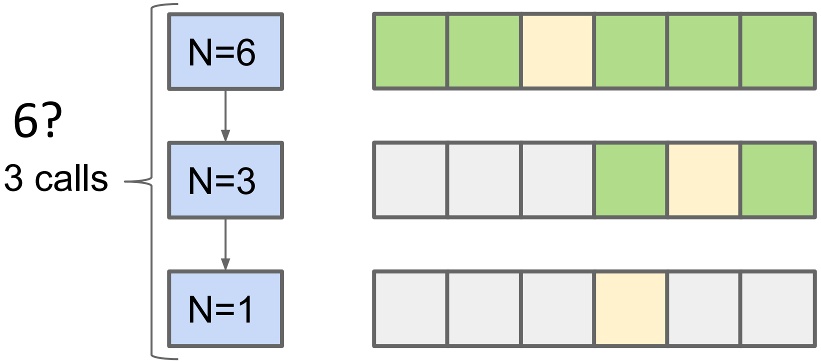
```
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else return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

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

Cost model: Number of `binarySearch` 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]

- Cost model: Number of `binarySearch` calls.

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

N=1

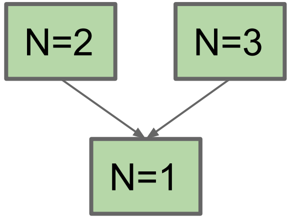
Binary Search (Exact Count)

```
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]

- Cost model: Number of `binarySearch` calls.

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



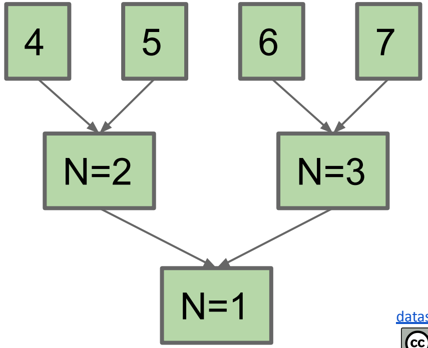
Binary Search (Exact Count)

```
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]

- Cost model: 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						



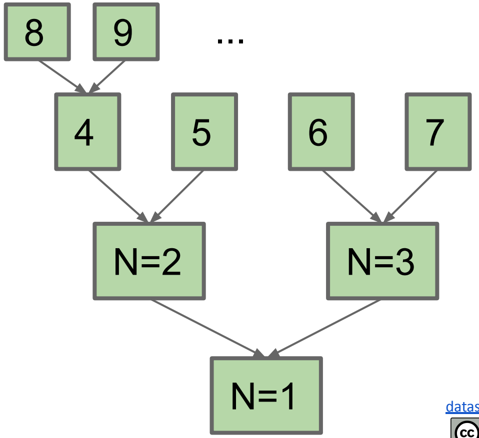
Binary Search (Exact Count)

```
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]

- Cost model: 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)

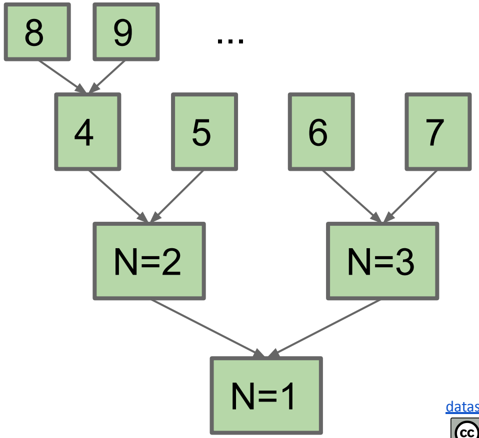
```java
static int binarySearch(String[] sorted, String x, int lo, int hi)
    if (lo > hi) return -1;
    int m = (lo + hi) / 2;
    int cmp = x.compareTo(sorted[m]);
    if (cmp < 0) return binarySearch(sorted, x, lo, m - 1);
    else if (cmp > 0) return binarySearch(sorted, x, m + 1, hi);
    else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

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

- Cost model: 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

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

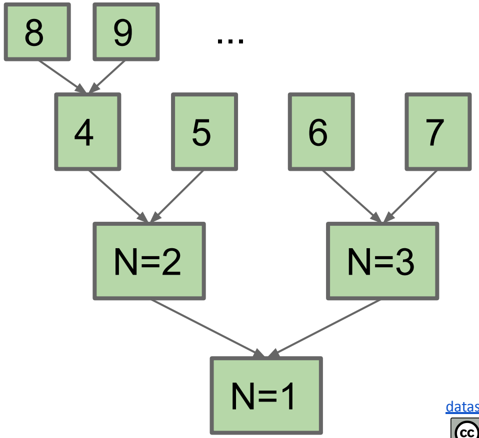


Binary Search (Exact Count)

```
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]

- Cost model: 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]

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

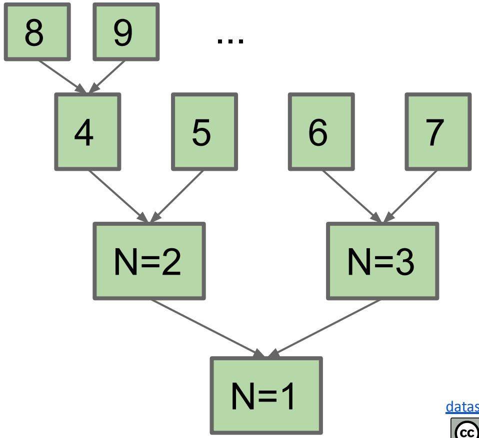
Since base is irrelevant, we omit from our big theta expression. We also omit the parenthesis around N for aesthetic reasons.

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]

- Cost model: 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 $C(N) = \Theta(\log N)$. Beyond scope of class, so won't solve in slides.

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

Example 5: Mergesort

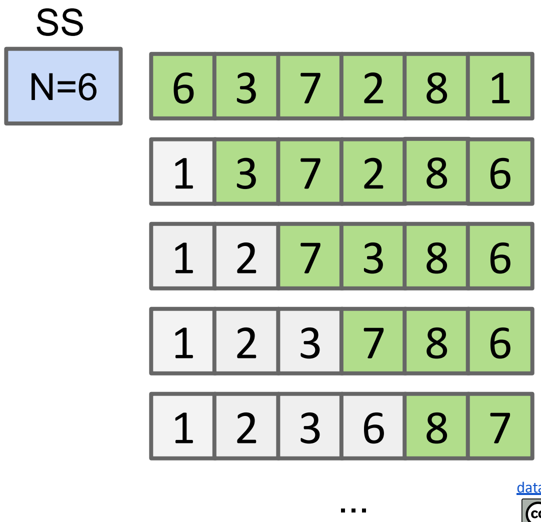
Selection Sort: A Prelude to Mergesort/Example 5

Earlier in class we discussed a sort called selection sort:

- Find the smallest unfixed item, move it to the front, and ‘fix’ it.
- Sort the remaining unfixed items using selection sort.

Runtime of selection sort is $\Theta(N^2)$:

- Look at all N unfixed items to find smallest.
- Then look at N-1 remaining unfixed.
- …
- Look at last two unfixed items.
- Done, sum is 2+3+4+5+...+N = $\Theta(N^2)$



Selection Sort: A Prelude to Mergesort/Example 5

Earlier in class we discussed a sort called selection sort:

- Find the smallest unfixed item, move it to the front, and ‘fix’ it.
- Sort the remaining unfixed items using selection sort.

Runtime of selection sort is $\Theta(N^2)$:

- Look at all N unfixed items to find smallest.
- Then look at N-1 remaining unfixed.
- …
- Look at last two unfixed items.
- Done, sum is $2+3+4+5+...+N = \Theta(N^2)$

SS

~36 AU N=6

SS

~4096 AU N=64

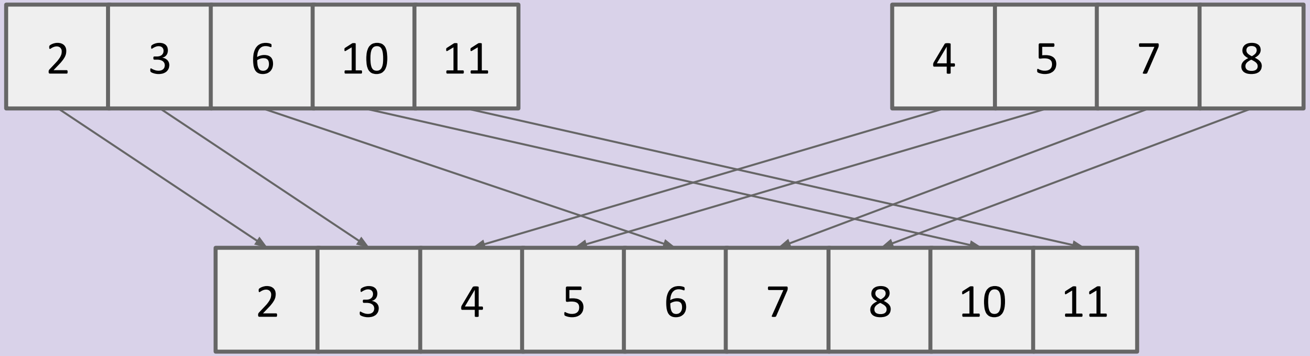
Given that runtime is quadratic, for N = 64, we might say the runtime for selection sort is 4,096 arbitrary units of time (AU).

The Merge Operation: Another Prelude to Mergesort/Example 5

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)

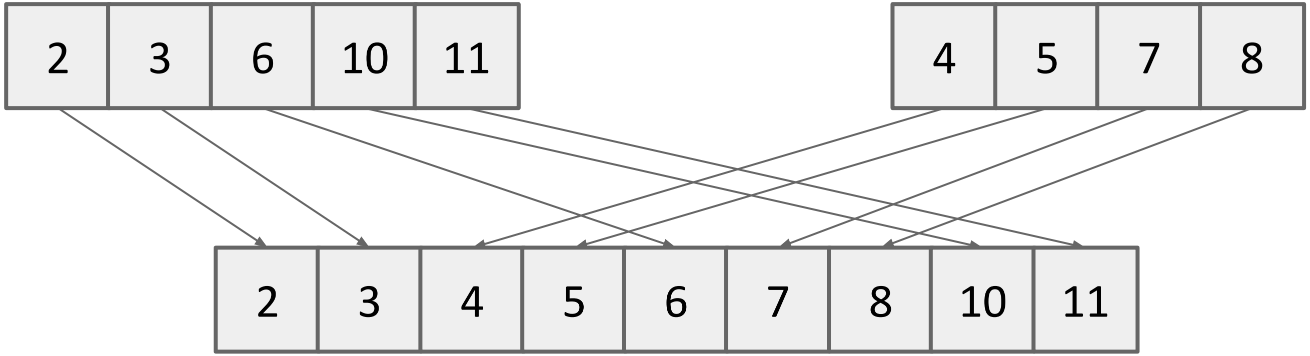
Merge Runtime: http://yellkey.com/report



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: http://shoutkey.com/TBA



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

C. Θ(N). Why? Use array writes as cost model, merge does exactly N writes.

Using Merge to Speed Up the Sorting Process

Merging can give us an improvement over vanilla selection sort:

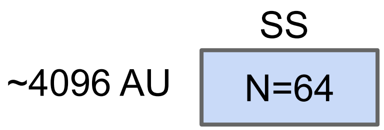
- Selection sort the left half: $\Theta(N^2)$.
- Selection sort the right half: $\Theta(N^2)$.
- Merge the results: $\Theta(N)$.

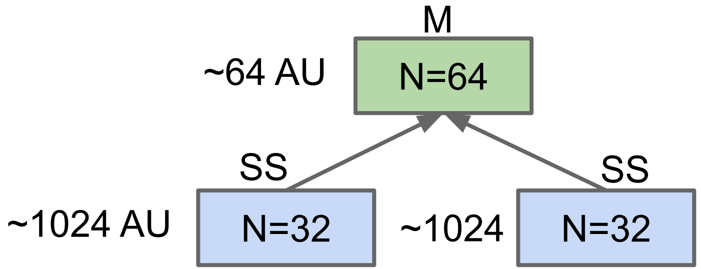
N=64: ~2112 AU.

- Merge: ~64 AU.
- Selection sort: ~2*1024 = ~2048 AU.

Still $\Theta(N^2)$, but faster since $N+2*(N/2)^2 < N^2$

- ~2112 vs. ~4096 AU for N=64.



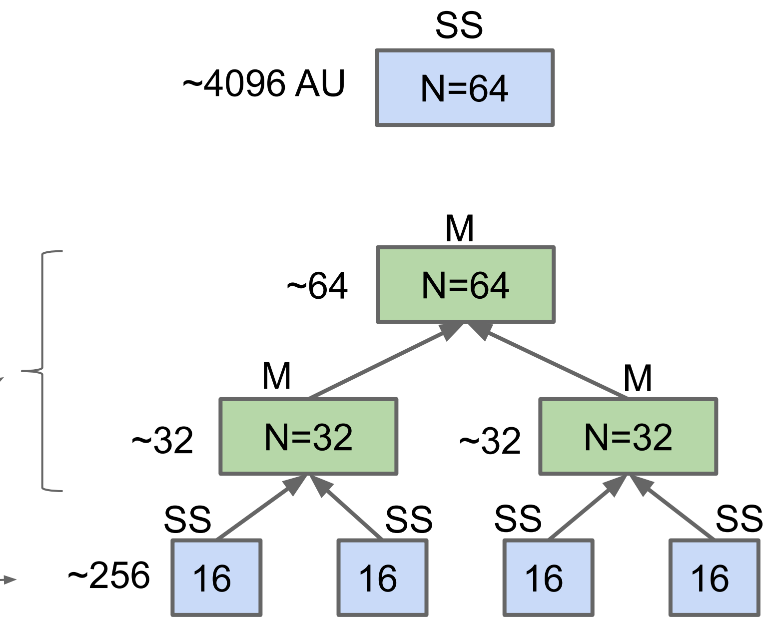


Two Merge Layers

Can do even better by adding a second layer of merges.

Runtime for each sort:

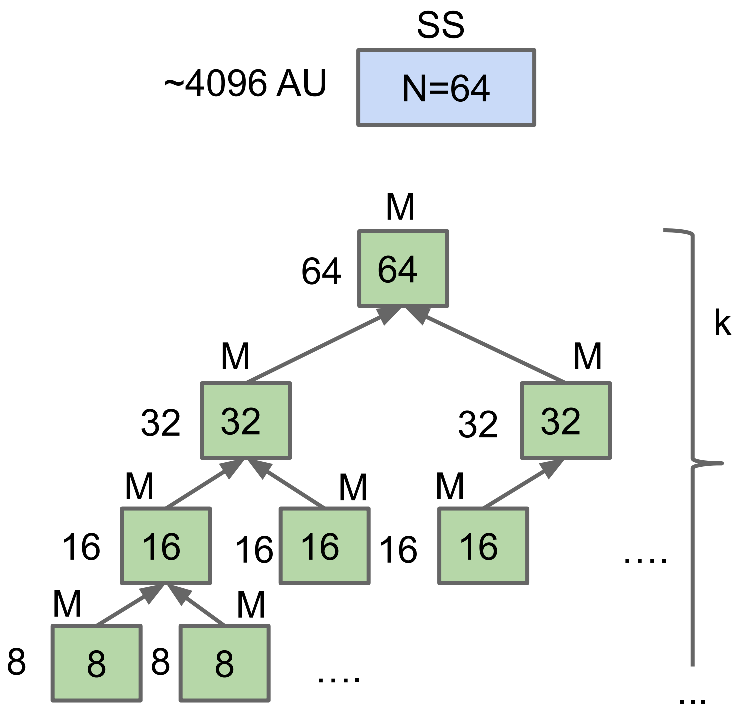
- Selection sort only: ~4096 AU.
- One layer of merges: ~2112 AU.
- Two layers of merges: ~1152 AU.
  - Merge: ~64 AU + 2*~32 AU.
  - Selection sort: 4*~256.



Example 5: Mergesort

Mergesort does merges all the way down (no selection sort):

- If array is of size 1, return.
- Mergesort the left half: Θ(??).
- Mergesort the right half: Θ(??).
- Merge the results: Θ(N).



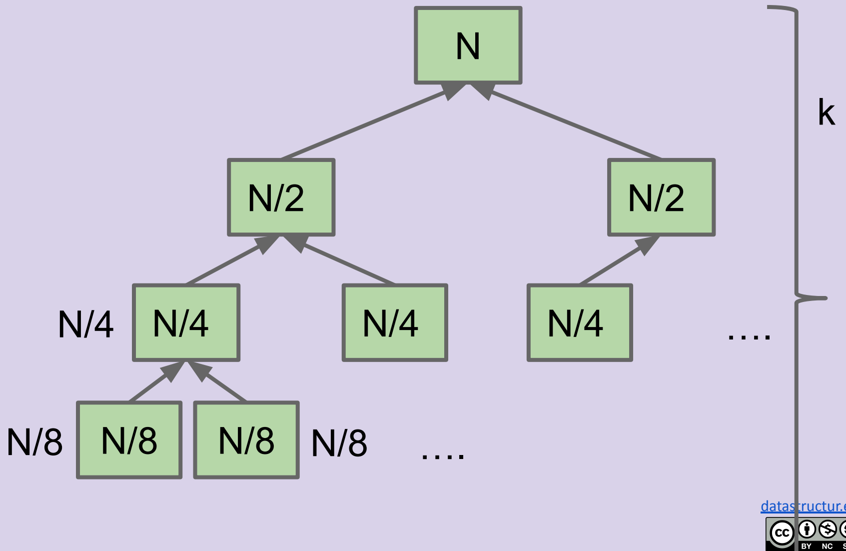
Total runtime to merge all the way down: ~384 AU

- Top layer: ~64 = 64 AU
- Second layer: ~32*2 = 64 AU
- Third layer: ~16*4 = 64 AU
- Overall runtime in AU is ~64k, where k is the number of layers.
- $k = \log_2(64) = 6$, so ~384 total AU.

Example 5: Mergesort Order of Growth, yellkey.com/job

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

A. $\Theta(1)$
B. $\Theta(\log N)$
C. $\Theta(N)$
D. $\Theta(N \log N)$
E. $\Theta(N^2)$



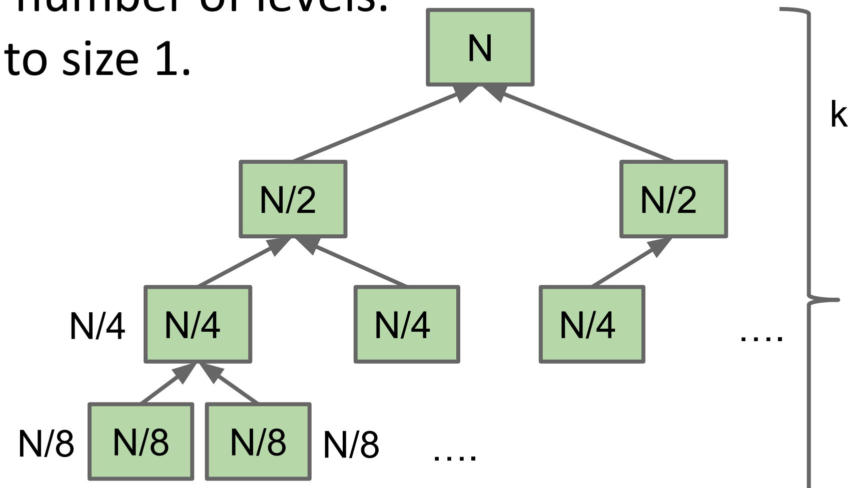
Example 5: Mergesort Order of Growth

Mergesort has worst case runtime = $\Theta(N \log N)$.

- Every level takes ~N AU.
  - Top level takes ~N AU.
  - Next level takes ~N/2 + ~N/2 = ~N.
  - 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 $\Theta(N \log N)$.

Exact count explanation is tedious.

- Omitted here. See textbook exercises.



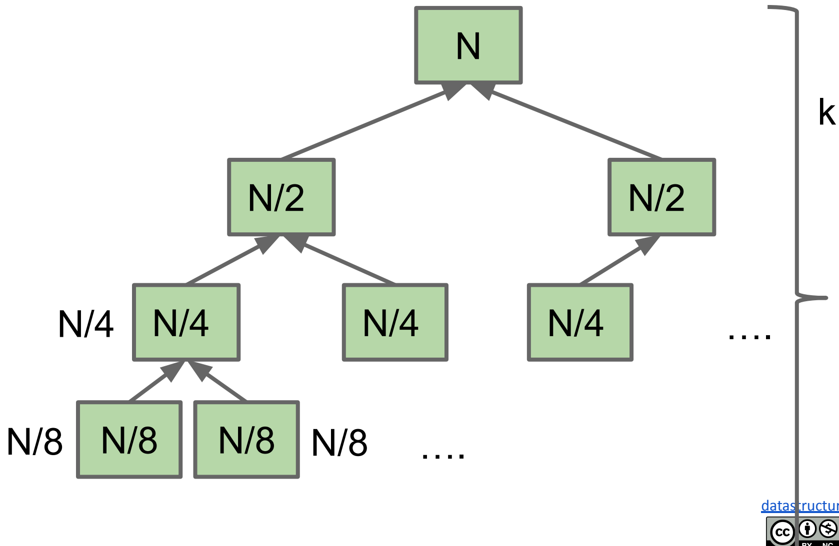
Mergesort using Recurrence Relations (Extra)

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

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

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

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



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 + 2 + 3 … + N and 1 + 2 + 4 + … + 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.

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.