Announcements

Upcoming Deadlines:

- Project 2 basics autograder due 3/2 at 11:59 PM.
- Project 2 due 3/7 at 11:59 PM. Autograder will be minimal since it largely will involve manual testing.
- No lab this week: It will be project 2 office hours instead.

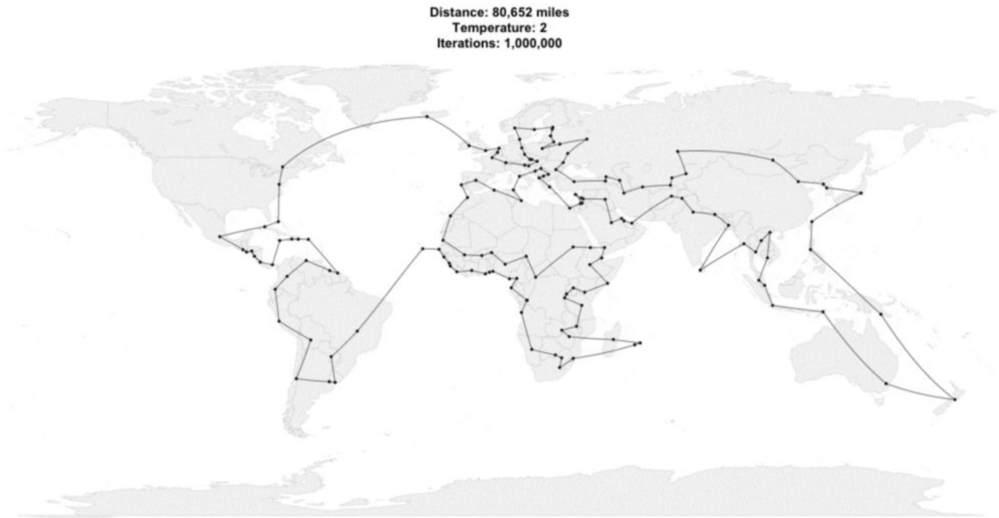
Lecture today/friday somewhat more slowly paced (given project).

- Today: Practice with what we learned Monday.

See study guides for each lecture starting Monday:

Asymptotics I
[video] [slides] [guide]

- Webcast viewers, do all B-level problems in guide before watching this lecture.



CS61B

Lecture 18: 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

Summary of Asymptotic Notations

	Informal meaning:	Family	Family Members
Big Theta $\Theta(f(N))$	Order of growth is f(N).	$\Theta(N^2)$	$N^2/2$ $2N^2$ $N^2 + 38N + N$
Big O $O(f(N))$	Order of growth is less than or equal to f(N).	$O(N^2)$	$N^2/2$ $2N^2$ $\lg(N)$
Big Omega $\Omega(f(N))$	Order of growth is greater than or equal to f(N).	$\Omega(N^2)$	$N^2/2$ $2N^2$ e^N

From discussion

Example 1/2:For Loops

Monday's Lecture

We discussed ways to analyze algorithm performance. To understand how code scales, we symbolically count number of executions of a ***representative operation*** as a function of input size N.

- Focus on behavior for large N: ***ignore lower order terms***.
- ***Ignore constant multiplicative factors***.

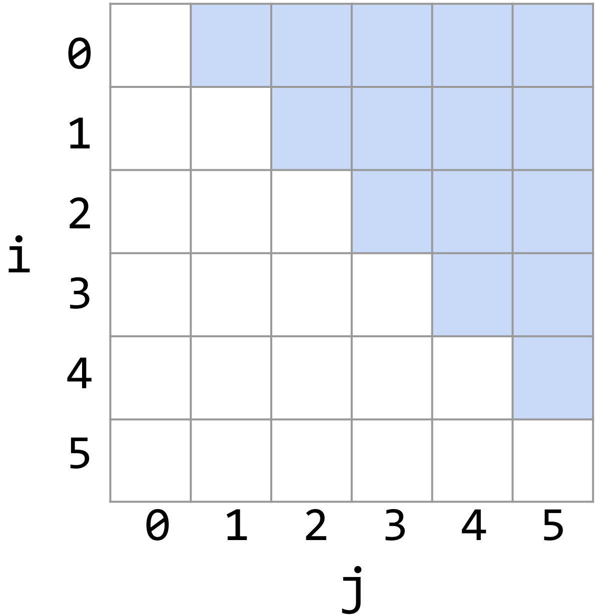
```java
int count = 0, N = a.length;
for (int i = 0; i < N; i++) {
    if (a[i] == k) {
        count += 1;
    }
}
a[k] += count;
```

Big Theta formalizes our intuitive simplifications.

operation	count	simplified count
increment	N+1 to 2N+1	$\Theta(N)$

Example 1: Nested For Loops

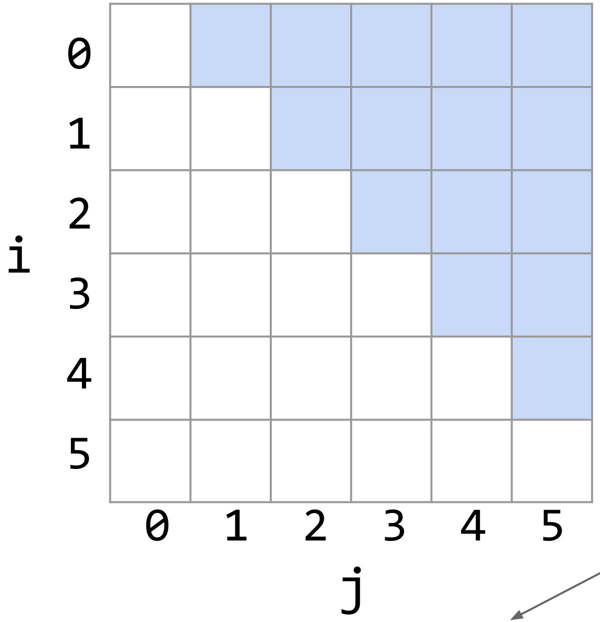
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)) in the worst case.




```
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;
```

Example 1: Nested For Loops

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




```
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;
```

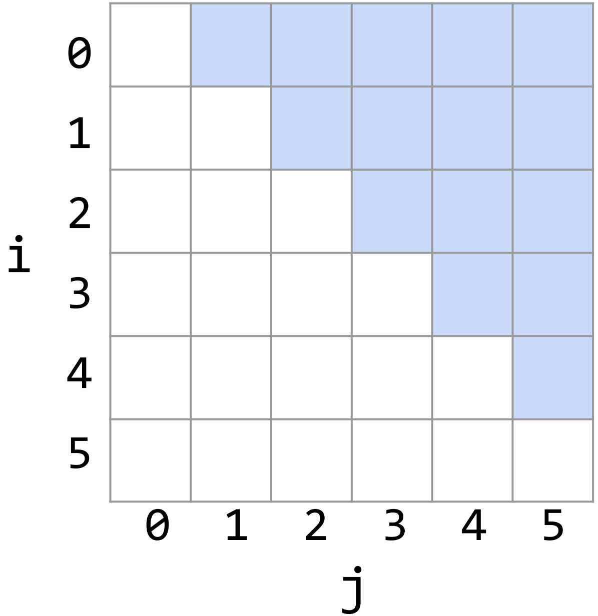
Worst case number of j += 1 calls:

$1 + 2 + 3 + \ldots + (N - 3) + (N - 2) + (N - 1) = \mathbf{N(N\text{-}1)/2}$

$\underbrace{1 + (N\text{-}1) = N \qquad N \qquad N}_{\text{(N-1)/2 of these combinations}}$

Example 1: Nested For Loops

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




```
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 j += 1 calls:

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

Overall worst case runtime: $\Theta(N^2)$

Nested For Loops II: PollEv.com/jhug or JHUG to 37607

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 printIndices2(int n) {
    for (int i = 1; i < n; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int A = 1 + 1;
        }
    }
}
```

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

Nested For Loops II: PollEv.com/jhug or JHUG to 37607

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

```
public static void printIndices2(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int A = 1 + 1;
```

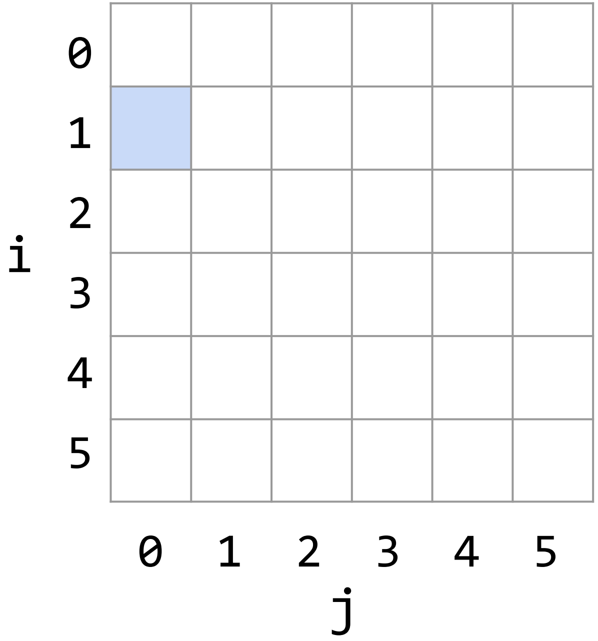
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

Cost model, println("hello") calls:

1	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18	19	20

n=1

Nested For Loops II: PollEv.com/jhug or JHUG to 37607




```
public static void printIndices2(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int A = 1 + 1;
```

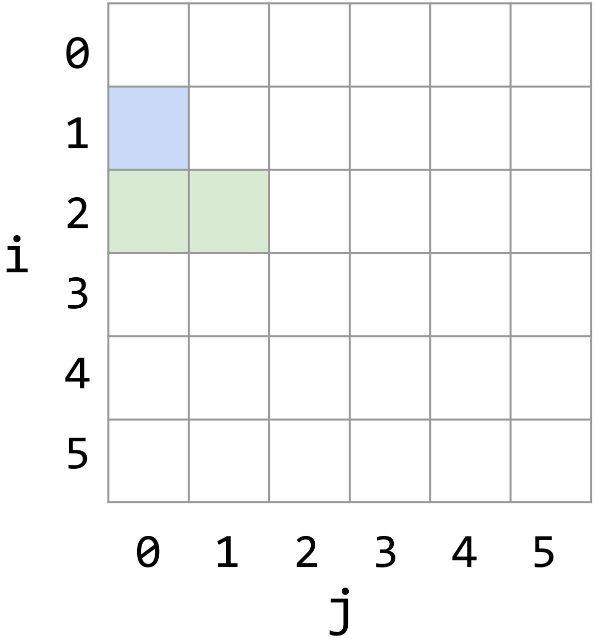
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

Cost model, println("hello") calls:

1	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18	19	20

n=1

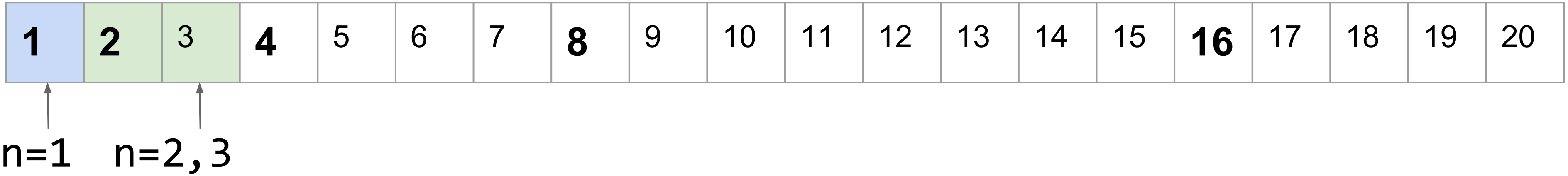
Nested For Loops II: PollEv.com/jhug or JHUG to 37607



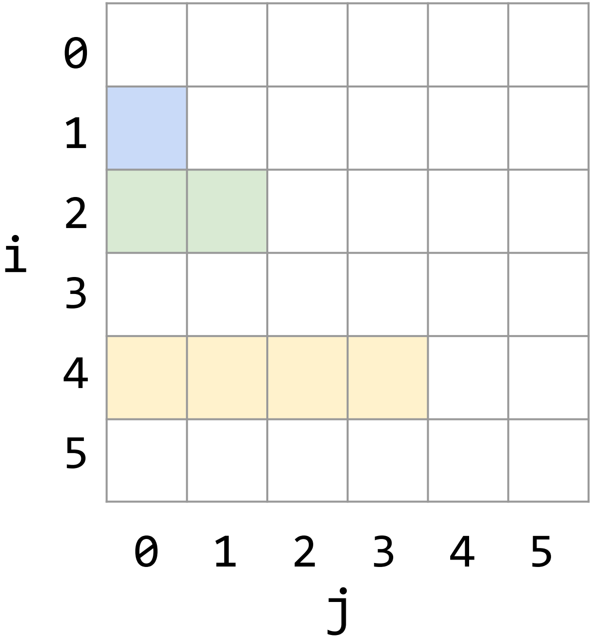

```java
public static void printIndices2(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int A = 1 + 1;
```

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

Cost model, println("hello") calls:



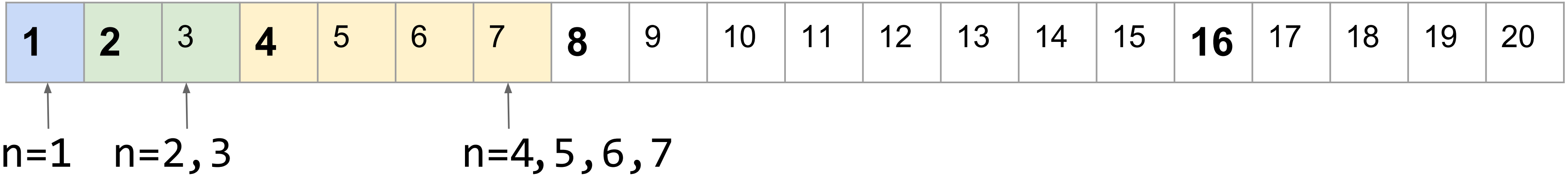
Nested For Loops II: PollEv.com/jhug or JHUG to 37607



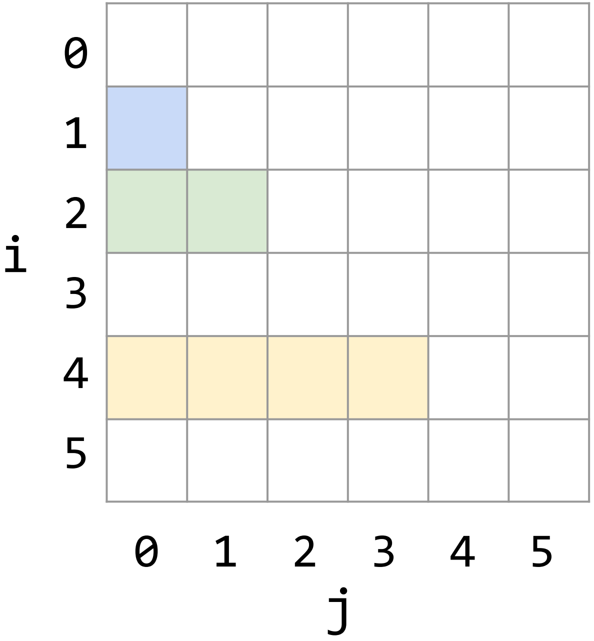

```java
public static void printIndices2(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int A = 1 + 1;
```

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

Cost model, println("hello") calls:



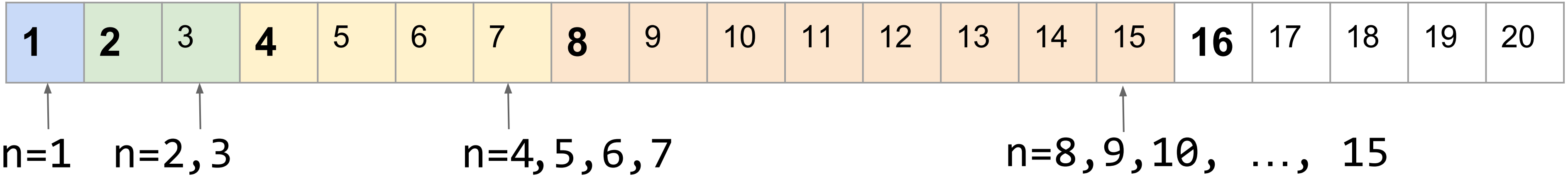
Nested For Loops II: PollEv.com/jhug or JHUG to 37607




```
public static void printIndices2(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int A = 1 + 1;
```

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

Cost model, println("hello") calls:



Nested For Loops II: PollEv.com/jhug or JHUG to 37607

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

"Worst case" here is irrelevant, all cases the same.

Cost model, println("hello") calls:

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

```
public static void printIndices2(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int A = 1 + 1;
```

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

Cost model, println("hello") calls:

1	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18	19	20

n=1 n=2,3 n=4,5,6,7 n=8,9,10, ..., 15

Nested For Loops II: PollEv.com/jhug or JHUG to 37607

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

N	R(N)
1	1
4	1 + 2 + 4 = 7
7	1 + 2 + 4 = 7
8	1 + 2 + 4 + 8 = 15
27	1 + 2 + 4 + 8 + 16 = 31
185	… + 64 + 128 = 255
715	… + 256 + 512 = 1023

```java
public static void printIndices2(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int A = 1 + 1;
```

"Worst case" here is irrelevant, all cases the same.

Cost model, println("hello") calls:

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

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

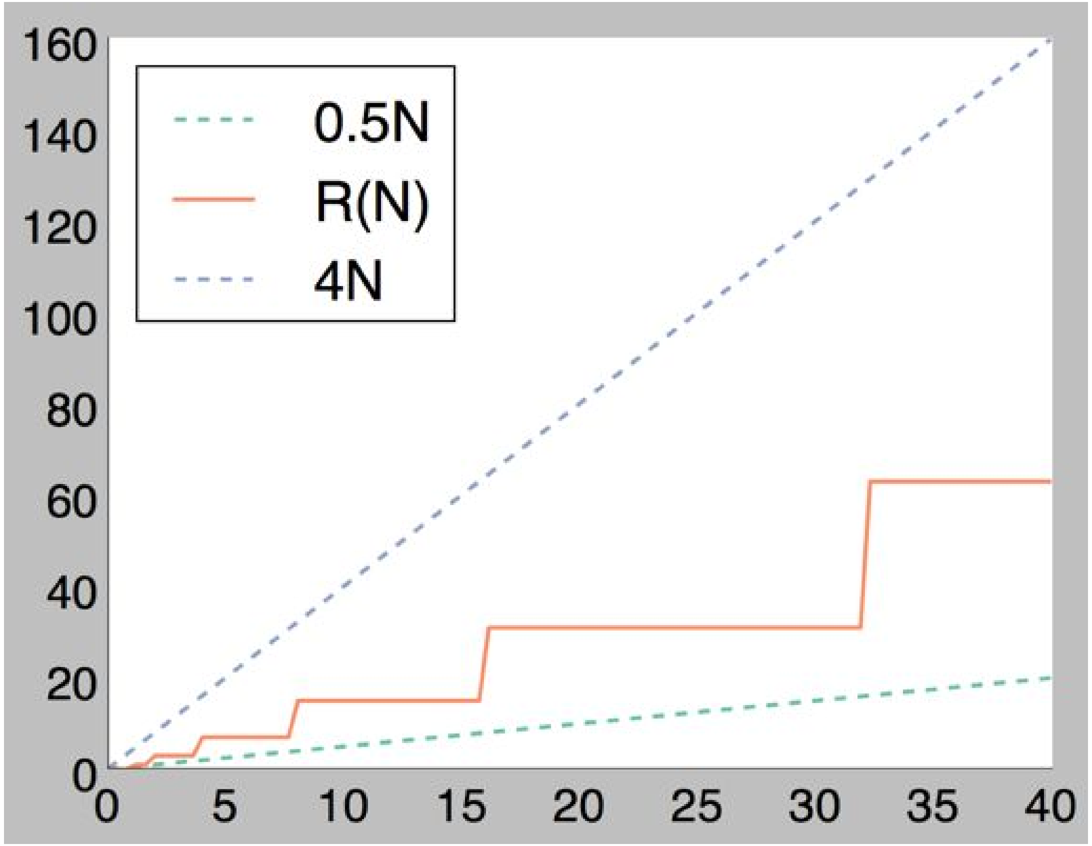
Nested For Loops II: PollEv.com/jhug or JHUG to 37607

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

N	R(N)	1 * N < R(N)	2 * N > R(N)
1	1	1	2
4	1 + 2 + 4 = 7	4	8
7	1 + 2 + 4 = 7	7	14
8	1 + 2 + 4 + 8 = 15	8	16
27	1 + 2 + 4 + 8 + 16 = 31	27	54
185	… + 64 + 128 = 255	185	370
715	… + 256 + 512 = 1023	715	1430

Nested For Loops II: PollEv.com/jhug or JHUG to 37607

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



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

$= \Theta(N)$

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

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:
  - $1 + 2 + 3 + \ldots + N = N(N+1)/2 = \Theta(N^2)$
  - $1 + 2 + 4 + 8 + \ldots + N = 2N - 1 = \Theta(N)$

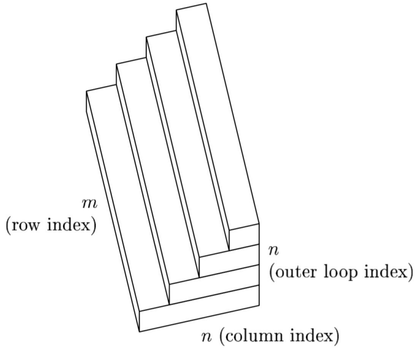
```java
public static void printIndices2(int n) {
    for (int i = 1; i < n; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int A = 1 + 1;
```

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:
  - $1 + 2 + 3 + \ldots + N = N(N+1)/2 = \Theta(N^2)$
  - $1 + 2 + 4 + 8 + \ldots + N = 2N - 1 = \Theta(N)$
- Strategies:
  - Write out examples
  - Draw pictures

QR decomposition runtime, from a numerical linear algebra textbook



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

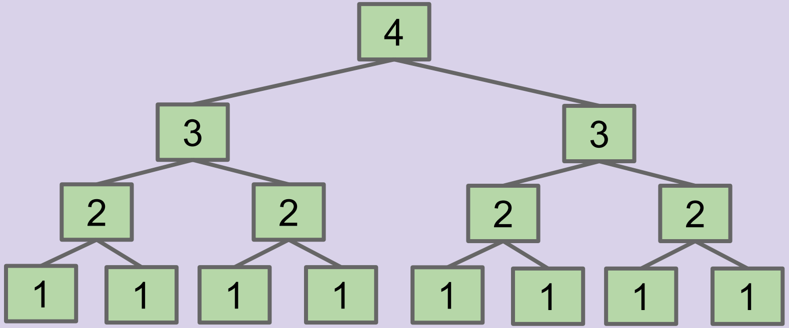
Recursion (Intuitive): PollEv.com/jhug or JHUG to 37607

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)
}
```

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



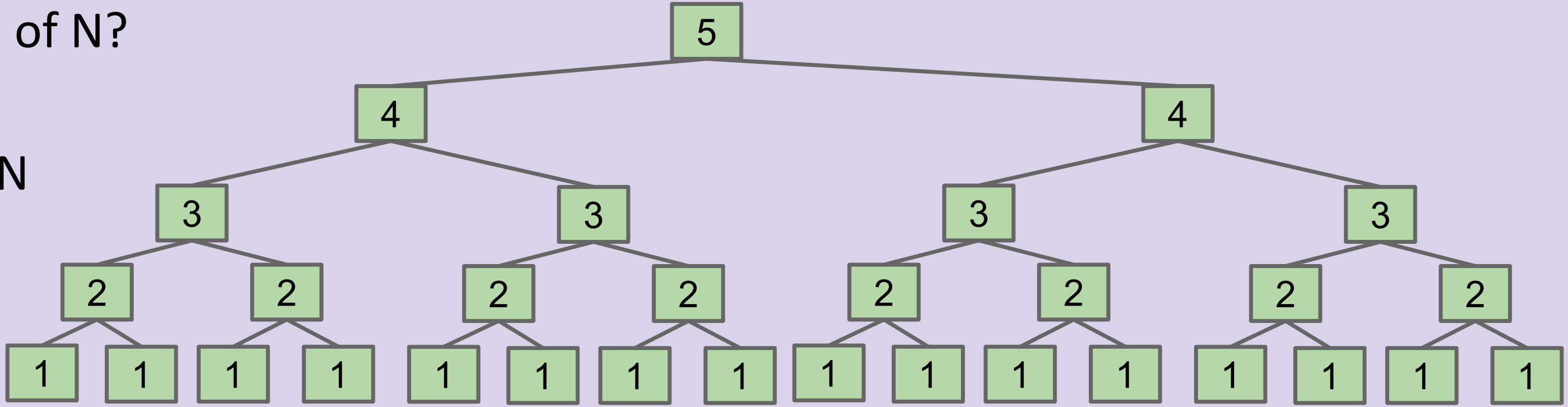
Recursion (Intuitive): PollEv.com/jhug or JHUG to 37607

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)
}
```

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



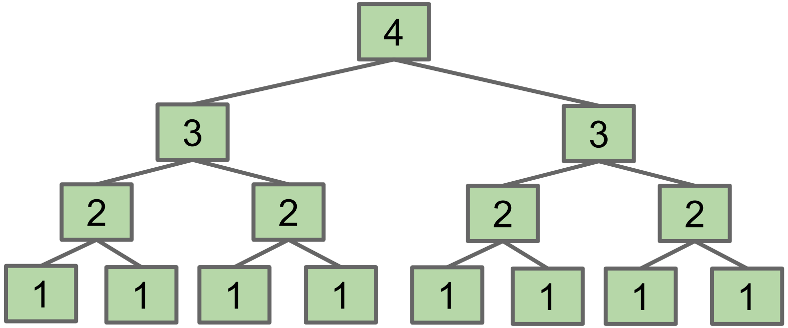
Recursion and Recurrence Relations

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)
}
```

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

- C(1) = 1



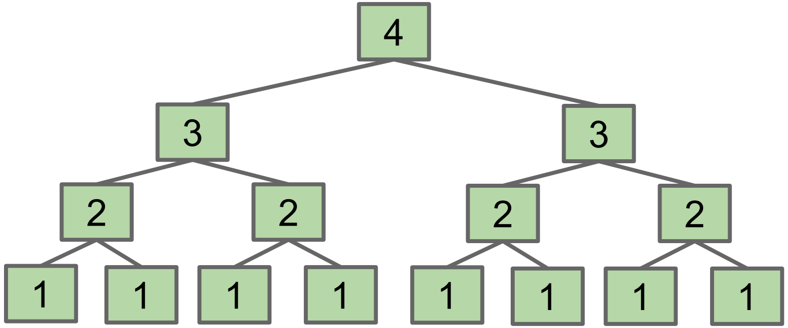
Recursion and Recurrence Relations

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)
}
```

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

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



Recursion and Recurrence Relations

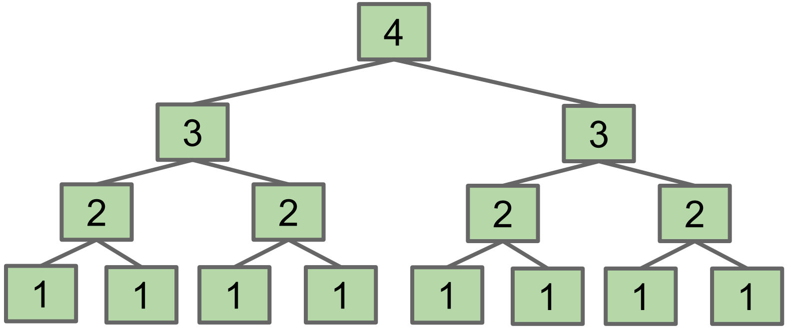
Find a simple f(N) such that the runtime R(N) $\in \Theta(f(N))$.

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

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

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

Possible to solve mechanically (with algebra), but we won't. Instead, we'll use intuition in 61b.



C(N) = 1 + 2 + 4 + 8 + … + ???

Recursion and Recurrence Relations: PollEv.com/jhug or JHUG to 37607

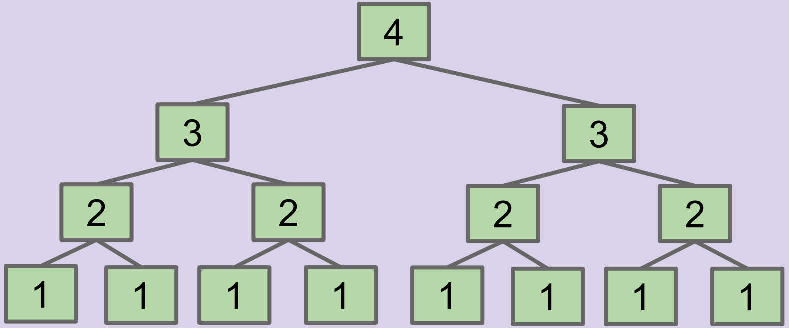
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

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

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

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

Possible to solve mechanically (with algebra), but we won't. Instead, we'll use intuition in 61b.



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

A. 2^N

B. 2^N-1

C. 2^{N+1}

D. $2^{N+1} + 1$

Sums of Powers of 2 (Revisited)

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

This is just the same sum we saw before, where $Q = 2^{N-1}$:

- $1 + 2 + 4 + 8 + \ldots + Q = 2Q - 1 \qquad = \Theta(Q)$

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

Recursion and Recurrence Relations

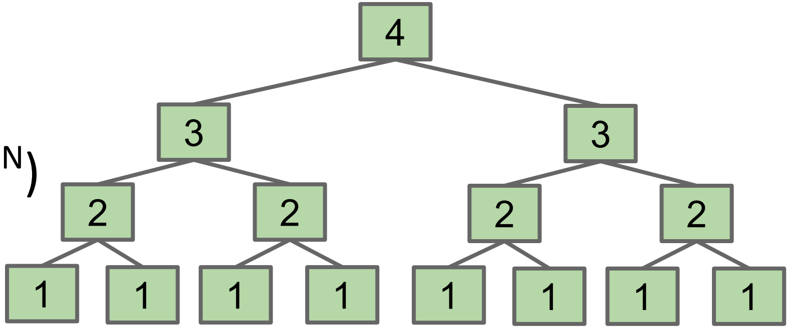
Find a simple f(N) such that the runtime $R(N) \in \Theta(f(N))$.

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

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

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

Possible to solve mechanically (with algebra), but we won't. Instead, we'll use intuition in 61b.



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

Recursion and Recurrence Relations (Extra)

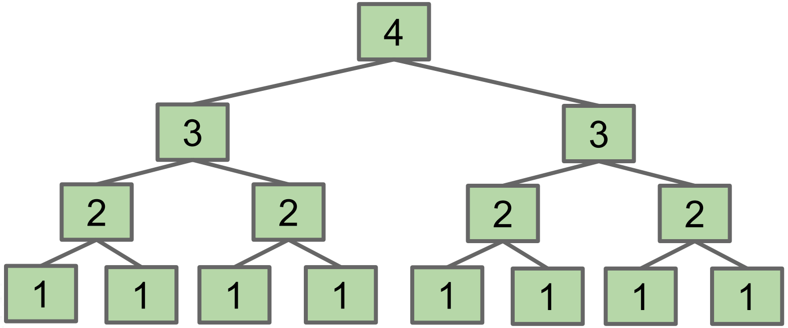
Find a simple f(N) such that the runtime R(N) ∈ Θ(f(N)).

This approach not covered in class. Provided for those of you who really want the algebra.

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

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: http://goo.gl/iSbyRV)

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;
}
```

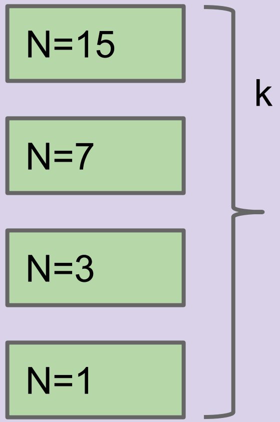
Binary Search (Intuitive): PollEv.com/jhug or JHUG to 37607

```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;
}
```

Let N = hi - lo + 1.

for simplicity: assume $N=2^k-1$ for some k

- What is the order of growth of the runtime of binarySearch?
  A. 1
  B. log N
  C. N
  D. N log N
  E. 2^N



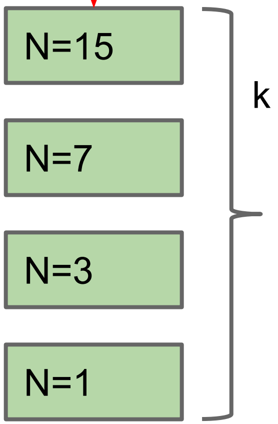
Binary Search

```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;
}
```

for simplicity: assume $N=2^k-1$ for some k

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

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



Binary Search

```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;
}
```

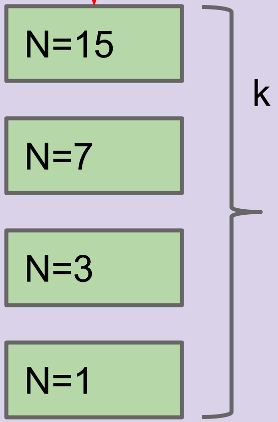
for simplicity: assume $N=2^k-1$ for some k

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

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

Give C(N) in terms of k:

C(N) = ???



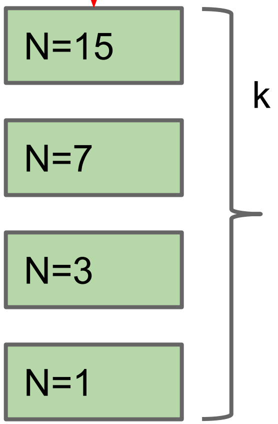
Binary Search

```
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;
}
```

for simplicity: assume $N=2^k-1$ for some k

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

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



$$C(N) = \underbrace{1 + 1 + \ldots + 1 + 0}_{k} = k$$

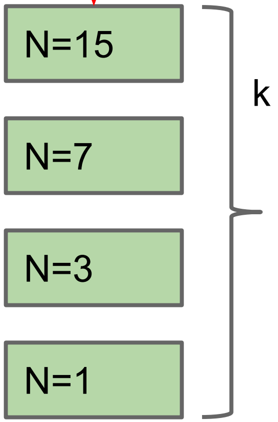
Binary Search

```
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;
}
```

for simplicity: assume $N=2^k-1$ for some k

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

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



$$C(N) = \underbrace{1 + 1 + \ldots + 1 + 0}_{k} = k$$

lg is short for base 2

$$k = \lceil \lg N \rceil = \Theta(\log N)$$

Unproven BigTheta Properties We've Just Used

Some easy-to-prove properties:

$$\log_K(N) \in \Theta(\log_Q(N))$$

Base of logarithm doesn't matter. We'll usually omit it completely!

$$\lceil f(N) \rceil \in \Theta(f(N))$$

$$\lfloor f(N) \rfloor \in \Theta(f(N))$$

Log Time Is Really Terribly Fast

Throughout this course we will see ways of doings things in constant vs log time. In practice, the difference is minimal.

N	$\log_2 N$	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

A Note on Solving Recurrence Relations

The entire goal is to find a pattern that yields a closed form solution for C(N).

- Use whatever tricks you'd like.
- This is not CS70.
  - We'll not deviate too terribly far from the patterns you'll see today and in discussion 8 (summing very simple arithmetic and geometric series).
  - We will not write rigorous proofs.

Example 5: Merge Sort

Selection Sort: A Prelude to 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.

This algorithm requires $\Theta(N^2)$ comparisons.

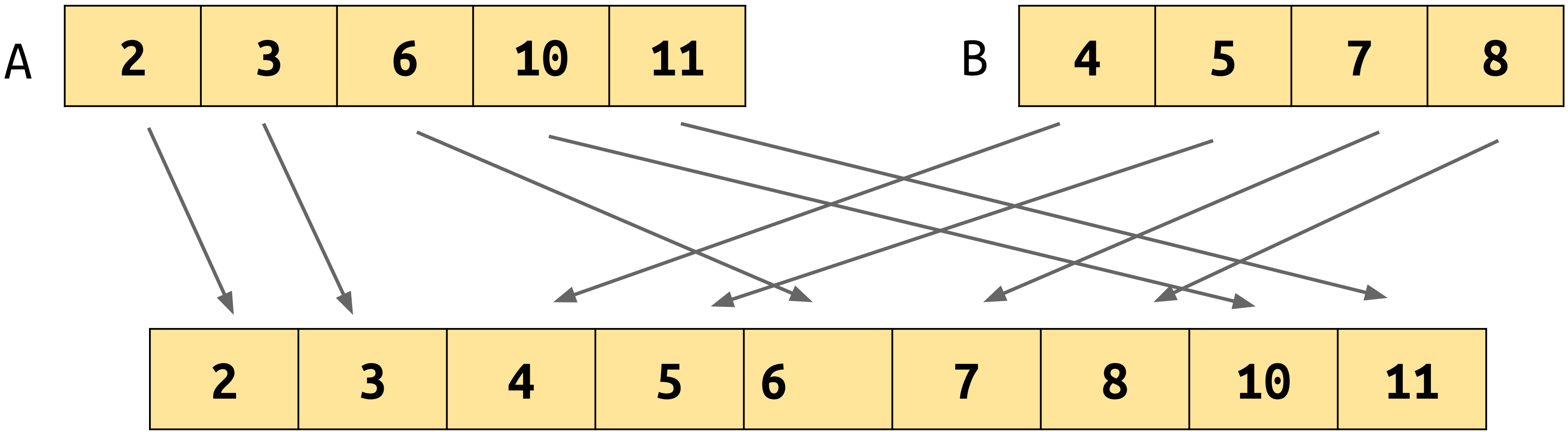
- Look at all N unfixed items to find smallest.
- Then look at N-1 remaining unfixed.
- …
- Look at last two unfixed items.
- Done, sum is $2+3+4+5+...+N = \Theta(N^2)$

SS

~2048 compares

N=64

Array Merging of N Total Items (A.length + B.length = N)



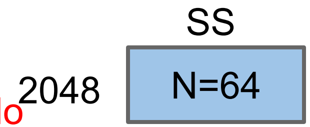
What is the runtime for merge? $\Theta(1)$, $\Theta(N)$, $\Theta(N^2)$???

- $\Theta(N)$ compares and array accesses.

The Merge Operation

One way to sort N items:

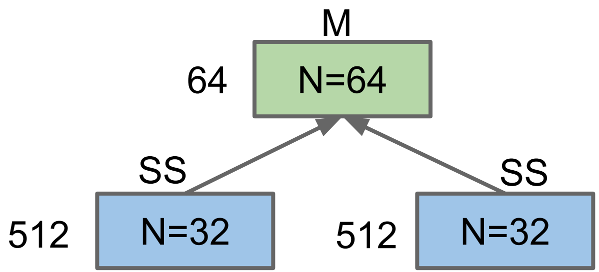
- Give half of the items away for sorting to one algorithm.
- Give the other half to some other algorithm.
- Merge the results: $\Theta(N)$ compares.



how do we do better?

Suppose the other two algs are selection sort.

- N=64:
  - Merge: ~64 compares.
  - Selection sort: ~512 each.



- Still $\Theta(N^2)$, but faster since $N+2*(N/2)^2 < N^2$
  - ~1088 vs. ~2048 compares for N=64.

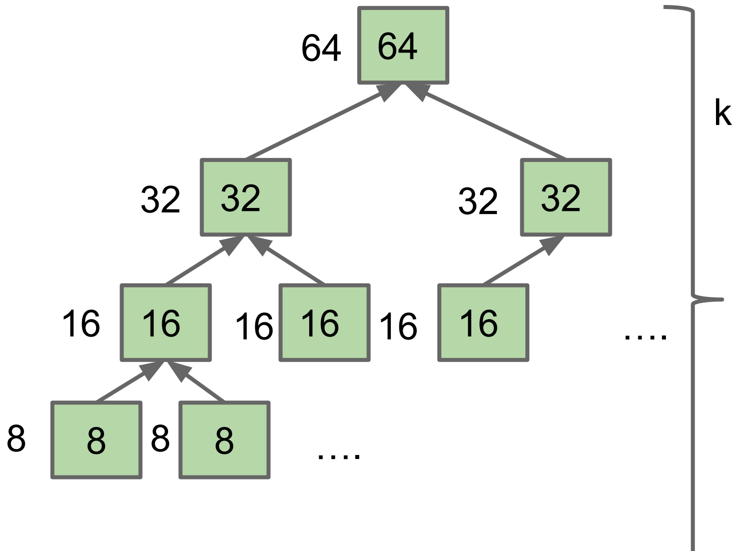
Example 5: Merge Sort

One way to sort N items:

- Give half of the items away for sorting to one algorithm.
- Give the other half to some other algorithm.
- Merge the results: Θ(N) compares.

Suppose they each use merge sort.

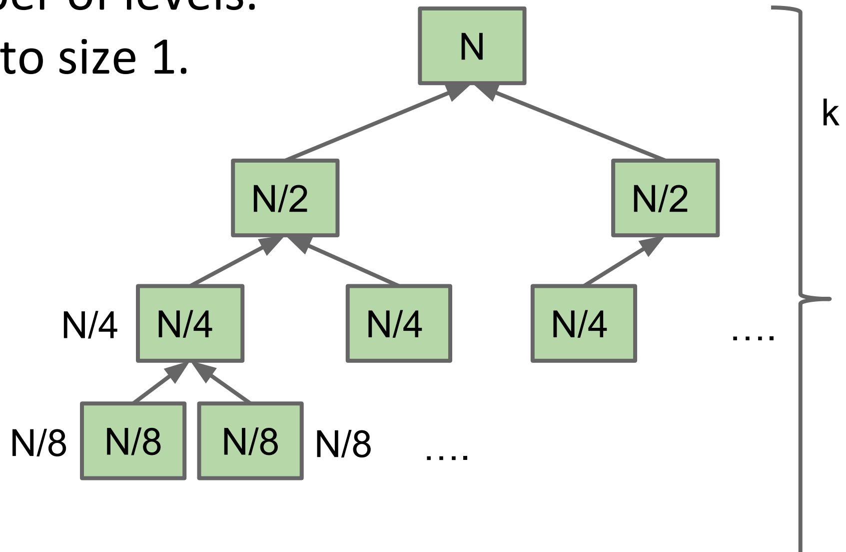
- N=64:
  - Top level: 64 compares
  - Next level: 64=32 + 32 compares.
  - Overall: 64*k compares.
  - k = ~lg(64), so ~384 compares.



Merge Sort: More General

Intuitive explanation:

- Every level does N work
  - Top level does N work.
  - Next level does N/2 + N/2 = N.
  - One more level down: N/4 + N/4 + N/4 + N/4 = N.
- Thus work is just Nk, where k is the number of levels.
  - How many levels? Goes until we get to size 1.
  - k = lg(N)
- Overall runtime is N log N.



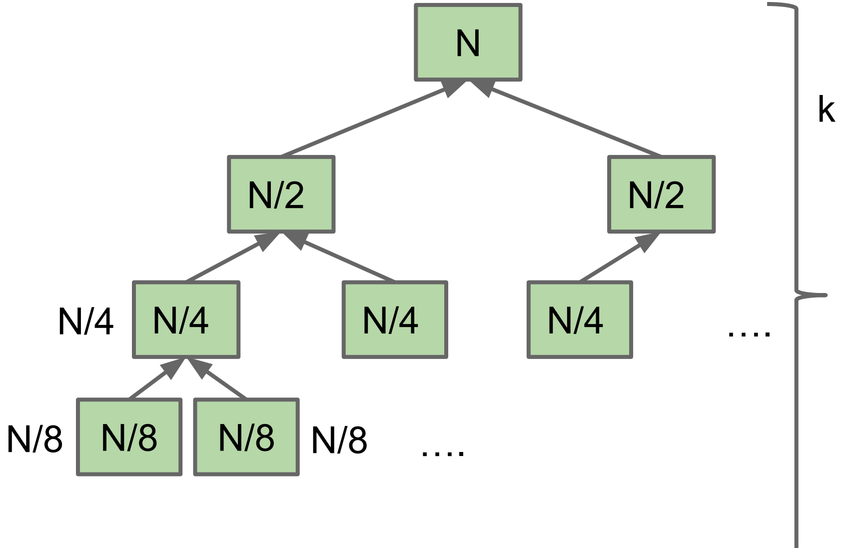
Merge Sort: Same Idea as Previous Slide, but Using Algebra

C(N): Number of items merged at each stage.

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

Only works for N=2^k. Can be generalized at the expense of some tedium (e.g. separately prove big O and big Omega)

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



Linear vs. Linearithmic (N log N)

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.

N log N is basically as good as N.

- Only a tiny bit slower. N = 1,000,000, and 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

Summary

Theoretical analysis of algorithm performance requires careful thought.

- Finding a simple expression for runtime is about finding patterns.
- Know the patterns we've learned today (more in HW and discussion).

Different solutions to the same problem may have radically different runtimes.
N^2 vs. N log N kind of a big deal.

Next time:

- Amortized analysis.
- Empirical measurement of program runtime.
- Sneak preview of complexity theory (extra).