Lecture 6: Chapter 2 Part 1

Divide-and-Conquer

Types of Algorithms

- There are many types of algorithms that solve problems by applying very specific strategies.
- These strategies work well for some problems types and poorly for others
  - The goal is to determine when to use which type of algorithm

In this course we will discuss:

- Divide-and-Conquer
- Greedy
- Dynamic Programming
- Backtracking
- Branch-and-Bound

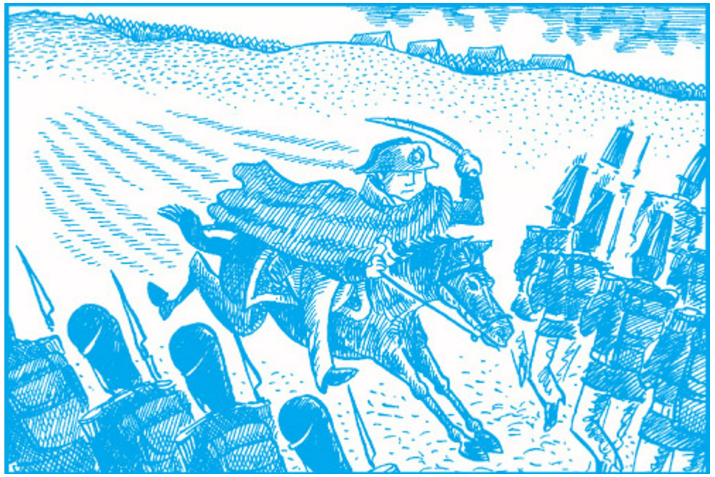
Divide-and-Conquer

On December 2, 1805, Napoleon's army was outnumbered by 15,000 soldiers.

The Austro-Russian army launched a massive attack on the French right flank.

Napoleon anticipated this and drive against the army's center, splitting it in two.

The two smaller armies were individually much more easily conquered.



Divide-and-Conquer

A Divide-and-Conquer algorithm has three general steps:

1. **Divide** a problem into smaller subproblems of the same kind.
2. **Conquer** (solve) these subproblems with recursion, using the same approach.
3. **Obtain** the solution from the recursive call(s).

We describe a divide-and-conquer algorithm in two different ways:

1. First, in a simple manner from a top-level perspective. We do not follow recursive steps or discuss implementation. We merely describe what the first procedure call does.
    - i.e. We might say: “Divide a list into two equal halves. Sort each recursively”
2. Second, show a much more in-depth example of the algorithm, demonstrating what occurs at each level of recursion.

Min-Max

Problem: Given an unordered list *A* of *n* values, find the minimum and maximum values

➢ A straightforward (i.e. not divide-and-conquer) algorithm:

```
void minMax (number &min, number &max)
     max = min = A[0]
     for (i = 1 to n - 1) do
          if (A[i] > max)
               max = A[i]
          if (A[i] < min)
               min = A[i]
```

What is the basic operation?

Min-Max

Problem: Given an unordered list *A* of *n* values, find the minimum and maximum values

➢ A straightforward (i.e. not divide-and-conquer) algorithm:

```
void minMax (number &min, number &max)
     max = min = A[0]
     for (i = 1 to n - 1) do
          if (A[i] > max)
               max = A[i]
          if (A[i] < min)
               min = A[i]
```

What is the basic operation?

There are 2: The comparison of `A[i]` with `max` and the comparison of `A[i]` with `min`.

➢ How many times do they occur?

Min-Max

Problem: Given an unordered list *A* of *n* values, find the minimum and maximum values

➢ A straightforward (i.e. not divide-and-conquer) algorithm:

```
void minMax (number &min, number &max)
     max = min = A[0]
     for (i = 1 to n - 1) do
          if (A[i] > max)
               max = A[i]
          if (A[i] < min)
               min = A[i]
```

What is the basic operation?

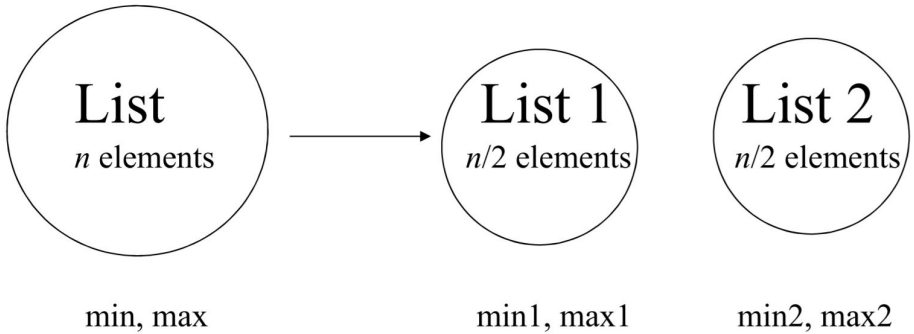
There are 2: The comparison of `A[i]` with `max` and the comparison of `A[i]` with `min`.

➢ How many times do they occur?

➢ Each one occurs n - 1 times, so $t_n = 2(n - 1)$

Min-Max Divide and Conquer

A top-level description of a divide and conquer approach to Min-Max



- **Divide**: the list into two sublists of equal size.
- **Conquer**: Recursively find the min and max of each sublist
- **Obtain**: The min and max values from both sublists. Determine the min and max values of the entire list with calls to simple `Min` and `Max` functions:

```
min = Min (min1, min2)
max = Max (max1, max2)
```

Min-Max Divide and Conquer Pseudocode

```
MinMax (index i, index j, &fmax, &fmin)
     if (i == j)                                  // base case 1
          fmin = fmax = A[i]
     else if (i == j - 1)                         // base case 2
          if (A[i] < A[j])
               fmax = A[j] && fmin = A[i]
          else
               fmin = A[j] && fmax = A[i]
     else
          int gmax, gmin, hmax, hmin = 0
          mid = ⌊(i + j) / 2⌋
          MinMax(i, mid, gmax, gmin)              // Find min and max of left half of array
          MinMax(mid + 1, j, hmax, hmin)          // Find min and max of right half of array
          fmin = Min(gmin, hmin)                  // Find min of both subarrays
          fmax = Max(gmax, hmax)                  // Find max of both subarrays
```

Note: `i` and `j` are index #s indicating which part of the array is being checked. A top-level call to `MinMax` with an array of size 10: `MinMax(1, 10, maxVal, minVal)`

Min-Max Divide and Conquer Pseudocode

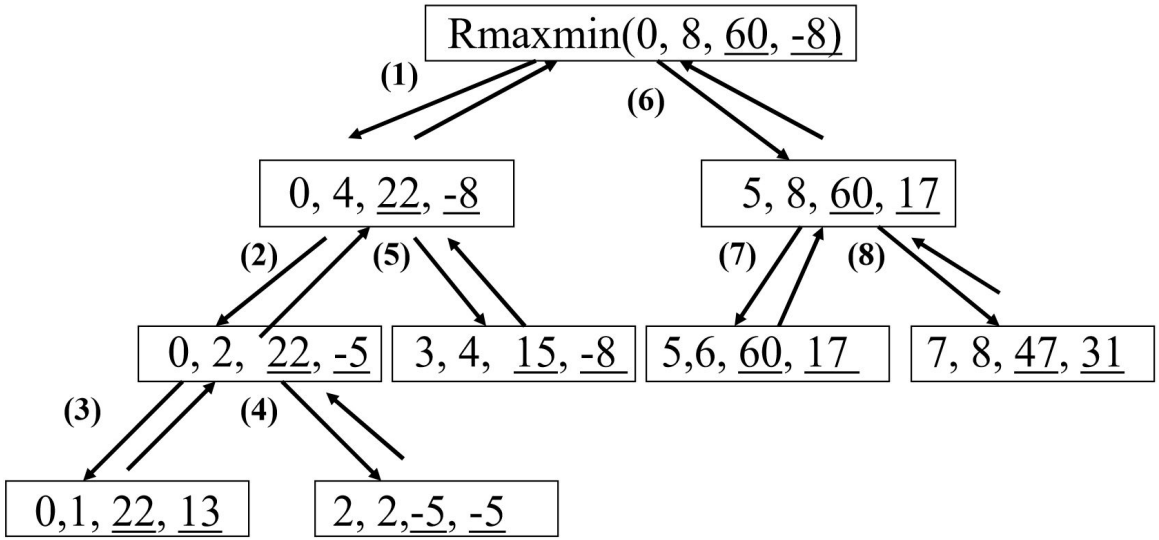
```
int maxVal = minVal = 0;
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
print maxVal;  // prints 10
print minVal;  // prints 0
```

Min-Max Divide and Conquer

Example: find *max* and *min* in the following array:

22, 13, -5, -8, 15, 60, 17, 31, 47 ($n = 9$)

Index:	0	1	2	3	4	5	6	7	8
Array:	22	13	-5	-8	15	60	17	31	47



Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:

Top Level

i = 1	gmax = 0
j = 6	gmin = 0
fmin = ?	hmax = 0
fmax = ?	hmin = 0
mid = 3	

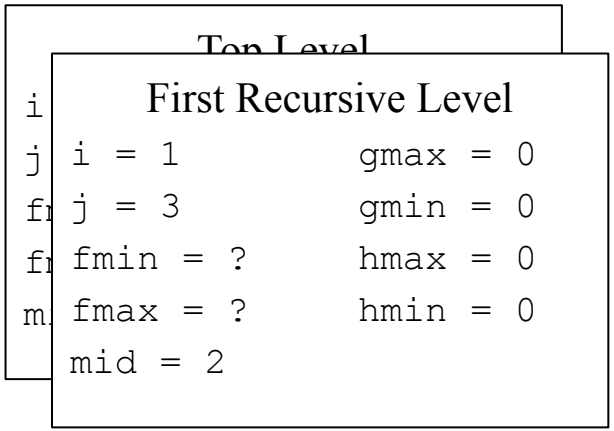
```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- In the top level call to `MinMax`, we are not at a base case, so we reach the `else` clause.
- We recursively call `MinMax` on the left half of the array (1 through 3)
  - `gmax` and `gmin` will hold the max and min of that half when control returns here.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:



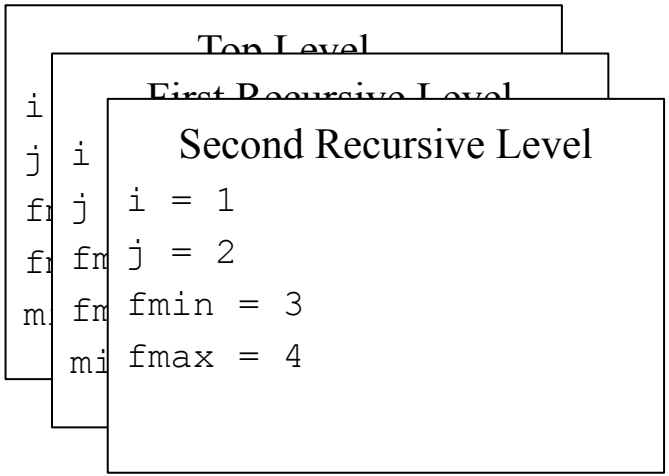

```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- In the first level of recursion we are not at a base case.
- We recursively call `MinMax` on the left half of this subarray (1 through 2)
  - `gmax` and `gmin` will hold the max and min of that half when control returns here.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:



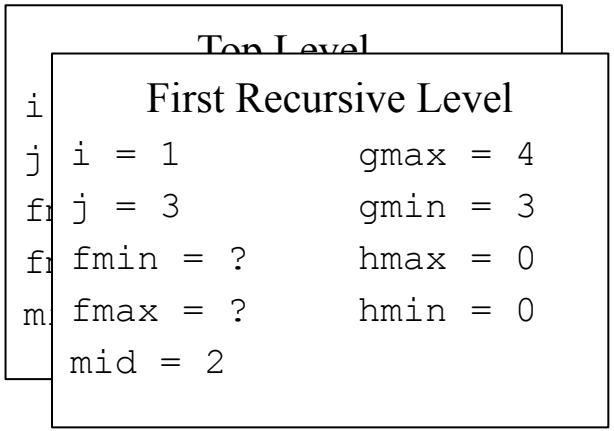

```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- In the second level of recursion, we are at a base case.
- `fmin` and `fmax` are set to the min and max of this subarray. This sets `gmin` and `gmax` in the first recursive level since they were passed by reference.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:



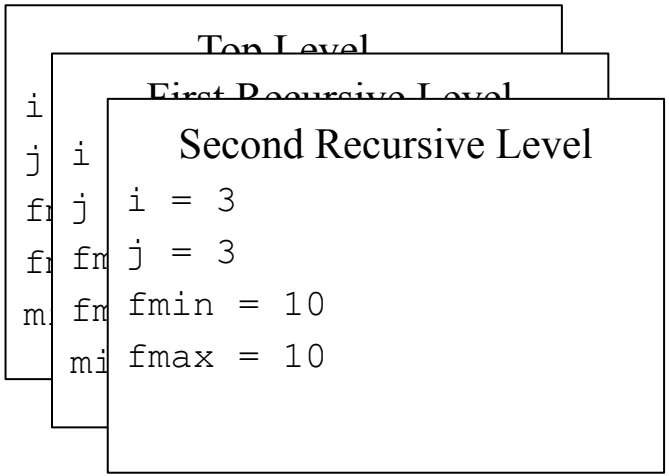

```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
→       MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- We return to the first level of recursion.
  - This level's first recursive call to `MinMax` has finished.
- We recursively call `MinMax` again, this time on the right side of this subarray. (3 through 3)

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:



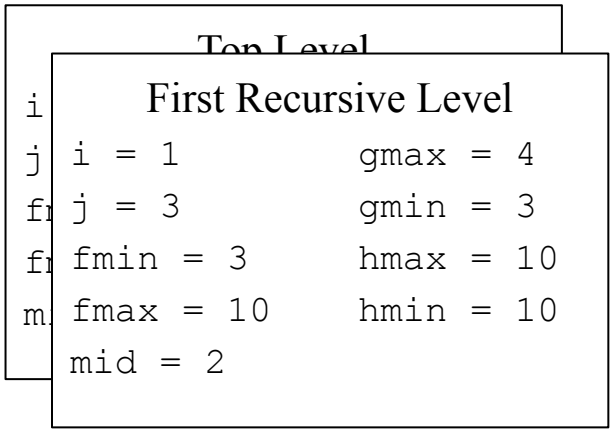

```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- In the second level of recursion, we are at a base case.
- `fmin` and `fmax` are set to the min and max of this subarray, setting `hmin` and `hmax` in the first recursive level.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:




```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
    → fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- We have returned to the first level of recursion.
- We determine the min and max of the current subarray.
  - When we set `fmin` and `fmax`, we are setting `gmax` and `gmin` in the top level.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:

Top Level

i = 1	gmax = 10
j = 6	gmin = 3
fmin = ?	hmax = 0
fmax = ?	hmin = 0
mid = 3	

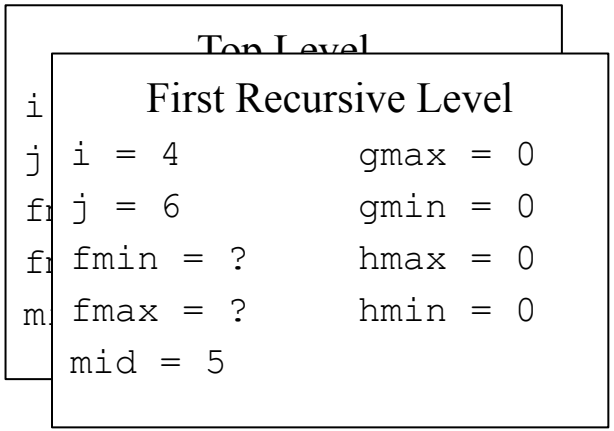
```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- We return to the top level (note that `gmax` and `gmin` are now set).
  - This level's first recursive call to `MinMax` has finished.
- We recursively call `MinMax` again, this time the right side of the array.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:



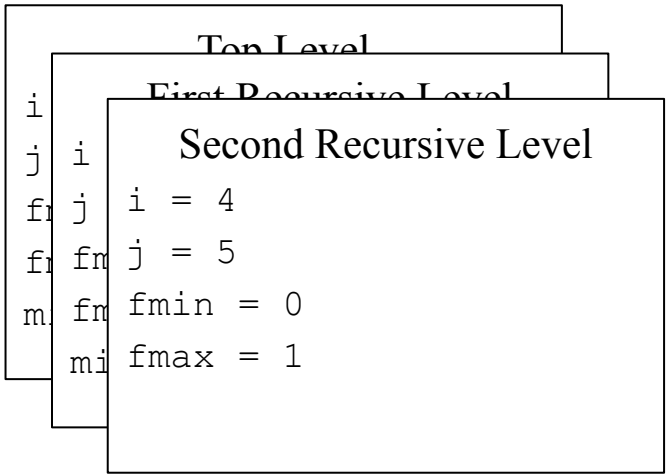

```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
→       MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- In the first level of recursion we are not at a base case.
- We recursively call `MinMax` on the left half of the current subarray (4 through 5)

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:



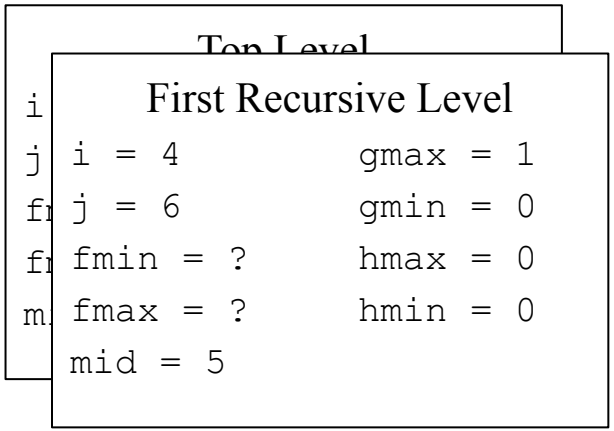

```
MinMax (index i, index j, &fmax, &fmin)
     if (i == j)
          fmin = fmax = A[i]
     else if (i == j - 1)
          if (A[i] < A[j])
               fmax = A[j] && fmin = A[i]
          else
               fmin = A[j] && fmax = A[i]
     else
          int gmax, gmin, hmax, hmin = 0
          mid = ⌊(i + j) / 2⌋
          MinMax(i, mid, gmax, gmin)
          MinMax(mid + 1, j, hmax, hmin)
          fmin = Min(gmin, hmin)
          fmax = Max(gmax, hmax)
```

- In the second level of recursion, we are at a base case.
- `fmin` and `fmax` are set to the min and max of this subarray, setting `gmin` and `gmax` in the first recursive level.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:



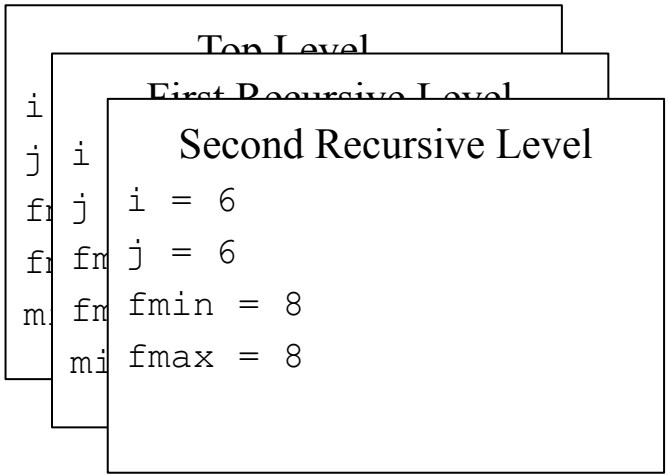

```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- We have returned to the first level of recursion.
  - This level's first recursive call to `MinMax` has finished.
- We recursively call `MinMax` again, this time the right side of the current subarray.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:



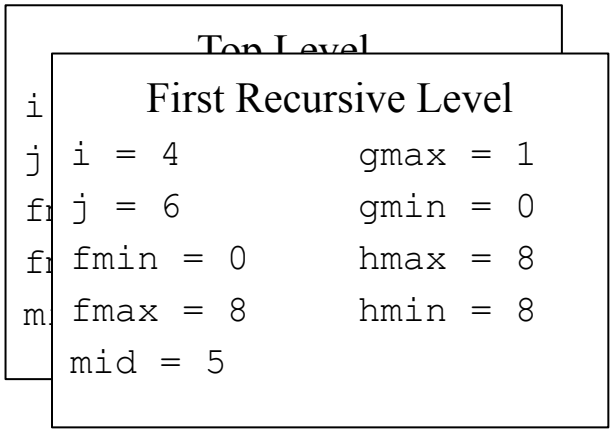

```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- In the second level of recursion, we are at a base case.
- `fmin` and `fmax` are set to the min and max of this subarray, setting `hmin` and `hmax` in the first recursive level.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:




```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
    → fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- We have returned to the first level of recursion.
- We determine the min and max of the current subarray, setting `hmax` and `hmin` in the top level.

Min-Max Example

```
A = { 4, 3, 10, 1, 0, 8 };
MinMax(1, 6, maxVal, minVal);
```

Activation Records:

Top Level

i = 1	gmax = 10
j = 6	gmin = 3
fmin = 0	hmax = 8
fmax = 10	hmin = 0
mid = 3	

```
MinMax (index i, index j, &fmax, &fmin)
    if (i == j)
        fmin = fmax = A[i]
    else if (i == j - 1)
        if (A[i] < A[j])
            fmax = A[j] && fmin = A[i]
        else
            fmin = A[j] && fmax = A[i]
    else
        int gmax, gmin, hmax, hmin = 0
        mid = ⌊(i + j) / 2⌋
        MinMax(i, mid, gmax, gmin)
        MinMax(mid + 1, j, hmax, hmin)
        fmin = Min(gmin, hmin)
        fmax = Max(gmax, hmax)
```

- We have returned to the top level (note that `hmax` and `hmin` are now set).
  - This level's second recursive call to `MinMax` has finished.
- We set `fmin` and `fmax`, setting `maxVal` and `minVal` in the initial function call.

Min-Max Analysis

Min-Max is recursive so we need a recurrence relation to find its time complexity.

- What is the basic operation?

- How many times do we perform this operation at the top level?

Min-Max Analysis

Min-Max is recursive so we need a recurrence relation to find its time complexity.

- What is the basic operation?
  - `Min` and `Max` function calls (or comparison at the base case)

- How many times do we perform this operation at the top level?

```
MinMax(i, mid, gmax, gmin)
MinMax(mid + 1, j, hmax, hmin)
min = MIN(gmin, hmin)
max = MAX(gmax, hmax)
```

The # of times we call `Min` and `Max` in both recursive calls and twice at the top level.

Min-Max Analysis

- Let t_n = total # of operations that occur with an input of size n
- $t_{n/2}$ is the # of operations that occur on half the input
- Total Calculations: $t_n = 2t_{n/2} + 2$ (i.e. 2 times the # of operations that occur on half the input + 2 at the top level)

Is the recurrence relation missing anything?

```
if (i == j)
     fmin = fmax = A[i]
if (i == j - 1)
     if (A[i] < A[j])
          fmax = A[j] && fmin = A[i]
     else
          fmin = A[j] && fmax = A[i]
```

Min-Max Analysis

- Let t_n = total # of operations that occur with an input of size n
- $t_{n/2}$ is the # of operations that occur on half the input
- Total Calculations: $t_n = 2t_{n/2} + 2$ (i.e. 2 times the # of operations that occur on half the input + 2 at the top level)

Is the recurrence relation missing anything? Initial condition(s)!

```
if (i == j)
     fmin = fmax = A[i]
if (i == j - 1)
     if (A[i] < A[j])
          fmax = A[j] && fmin = A[i]
     else
          fmin = A[j] && fmax = A[i]
```

Min-Max Analysis

- Let t_n = total # of operations that occur with an input of size n
- $t_{n/2}$ is the # of operations that occur on half the input
- Total Calculations: $t_n = 2t_{n/2} + 2$ (i.e. 2 times the # of operations that occur on half the input + 2 at the top level)

Is the recurrence relation missing anything?

```
if (i == j)
     fmin = fmax = A[i]    // When n = 1, 0 comparisons
if (i == j - 1)            // When n = 2, 1 comparison
     if (A[i] < A[j])
          fmax = A[j] && fmin = A[i]
     else
          fmin = A[j] && fmax = A[i]
```

We end up with the following recurrence relation:

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2T(\frac{n}{2}) + 2 & otherwise \end{cases}$$

Min-Max Analysis

- $t_n = 2t_{n/2} + 2$ is the total # of basic operations that are performed in a call to Min-Max.
- If we solve this recurrence relation, we can plug any number in for n to determine the total # of basic operations that will occur with that input size.
  - Without a solution, we must recursively determine the number of operations that take place which is less than ideal as n grows!

Min-Max Analysis

$t_n = 2t_{n/2} + 2$ is the total # of basic operations at the top level of recursion

- We will eventually solve this recurrence relation to determine how many total operations take place in Min-Max for any n.
- Let's first recursively see how many operations Min-Max performs when $n = 16$:

A = { 22, 13, -5, -8, 15, 60, 17, 31, 5, 7, -3, 10, 1, -2, 18, -12 }, $n = 16$

How many operations occur in a call to Min-Max in this instance?

Min-Max Analysis

$t_n = 2t_{n/2} + 2$ is the total # of basic operations at the top level of recursion

- We will eventually solve this recurrence relation to determine how many total operations take place in Min-Max for any n.
- Let's first recursively see how many operations Min-Max performs when $n = 16$:

A = { 22, 13, -5, -8, 15, 60, 17, 31, 5, 7, -3, 10, 1, -2, 18, -12 }, $n = 16$

How many operations occur in a call to Min-Max in this instance?

$t_{16} = 2t_{16/2} + 2$ or $2t_8 + 2$

Min-Max Analysis

$A = \{ 22, 13, -5, -8, 15, 60, 17, 31, 5, 7, -3, 10, 1, -2, 18, -12 \}$, $n = 16$

$t_{16} = 2t_8 + 2$

- At the top level call to Min-Max, the list has 16 elements.
- We recursively call Min-Max on two sublists of size 8 (hence $2t_8$).
- When we *obtain* the answer from these two recursive calls, we perform the basic operation two more times (hence + 2)

What is the next step?

Min-Max Analysis

$A = \{ 22, 13, -5, -8, 15, 60, 17, 31, 5, 7, -3, 10, 1, -2, 18, -12 \}$, $n = 16$

$t_{16} = 2t_8 + 2$

- At the top level call to Min-Max, the list has 16 elements.
- We recursively call Min-Max on two sublists of size 8 (hence $2t_8$).
- When we *obtain* the answer from these two recursive calls, we perform the basic operation two more times (hence + 2)

What is the next step?

- Calculate how many operations take place on a list of size 8

Min-Max Analysis

$A_1 = \{ 22, 13, -5, -8, 15, 60, 17, 31 \}$ $A_2 = \{ 5, 7, -3, 10, 1, -2, 18, -12 \}$ $n = 8$

$t_{16} = 2t_8 + 2$

- We have two second levels of recursion, each with sublists of size 8.
- We only need to calculate the # of basic operations that occur in one of these since it is equal for both (and we multiply that # by 2 in the above equation)

$t_8 = ?$

Min-Max Analysis

$A_1 = \{ 22, 13, -5, -8, 15, 60, 17, 31 \}$ $A_2 = \{ 5, 7, -3, 10, 1, -2, 18, -12 \}$ $n = 8$

$t_{16} = 2t_8 + 2$

- We have two second levels of recursion, each with sublists of size 8.
- We only need to calculate the # of basic operations that occur in one of these since it is equal for both (and we multiply that # by 2 in the above equation)

$t_8 = ?$ To solve this, plug 8 in to the recurrence relation $t_n = 2t_{n/2} + 2$:

$t_8 = 2t_{8/2} + 2$ or $2t_4 + 2$
$t_{16} = 2(2t_4 + 2) + 2$ or $4t_4 + 4 + 2$

Min-Max Analysis

$A_1 = \{ 22, 13, -5, -8 \}$ $A_2 = \{ 15, 60, 17, 31 \}$ $A_3 = \{ 5, 7, -3, 10 \}$ $A_4 = \{ 1, -2, 18, -12 \}$ $n = 4$

$t_{16} = 4t_4 + 4 + 2$

- We now know how many operations take place at the top level call, including how many take place in the first level of recursion.
- We have four second levels of recursion, each with $n = 4$. What next?

Min-Max Analysis

$A_1 = \{ 22, 13, -5, -8 \}$ $A_2 = \{ 15, 60, 17, 31 \}$ $A_3 = \{ 5, 7, -3, 10 \}$ $A_4 = \{ 1, -2, 18, -12 \}$ $n = 4$

$t_{16} = 4t_4 + 4 + 2$

- We now know how many operations take place at the top level call, including how many take place in the first level of recursion.
- We have four second levels of recursion, each with $n = 4$. What next?

$t_4 = ?$ To solve this, plug 4 in to the recurrence relation $t_n = 2t_{n/2} + 2$:

$t_4 = 2t_{4/2} + 2$ or $t_4 = 2t_2 + 2$

$t_{16} = 4(2t_2 + 2) + 4 + 2$ or $8t_2 + 8 + 4 + 2$

Min-Max Analysis

$A_1 = \{ 22, 13 \}$ $A_2 = \{ -5, -8 \}$ $A_3 = \{ 15, 60 \}$ $A_4 = \{ 17, 31 \}$
$A_5 = \{ 5, 7 \}$ $A_6 = \{ -3, 10 \}$ $A_7 = \{ 1, -2 \}$ $A_8 = \{ 18, -12 \}$ $n = 2$

$t_{16} = 8t_2 + 8 + 4 + 2$

- What is the next step?

Min-Max Analysis

$A_1 = \{ 22, 13 \}$ $A_2 = \{ -5, -8 \}$ $A_3 = \{ 15, 60 \}$ $A_4 = \{ 17, 31 \}$
$A_5 = \{ 5, 7 \}$ $A_6 = \{ -3, 10 \}$ $A_7 = \{ 1, -2 \}$ $A_8 = \{ 18, -12 \}$ $n = 2$

$t_{16} = 8t_2 + 8 + 4 + 2$

- What is the next step?
  - We have 8 base cases (for this recurrence relation, recursion stops at t_2).
  - Therefore, we can plug in 1 for t_2 since we know that is how many operations occur at case t_2

$t_{16} = 8(1) + 8 + 4 + 2 = \mathbf{22}$

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2T(\frac{n}{2}) + 2 & otherwise \end{cases}$$

Min-Max Analysis

$t_{16} = 8(1) + 8 + 4 + 2 = \mathbf{22}$

- We have shown that a total of 22 operations take place in Min-Max when $n = 16$.
  - However, different sizes of arrays can be passed to Min-Max.
- We want to find an equation that will instantly tell us how many operations occur for *any* size of n, not just 16.

Min-Max Analysis

$t_n = 2t_{n/2} + 2$ is the total # of basic operations at the top level of recursion

- Note that the result of t_n is dependent on how many basic operations occur at the second level of recursion, $t_{n/2}$.

- Based on the recurrence relation, we can determine what $t_{n/2}$ equals:

Min-Max Analysis

$t_n = 2t_{n/2} + 2$ is the total # of basic operations at the top level of recursion

- Note that the result of t_n is dependent on how many basic operations occur at the second level of recursion, $t_{n/2}$.

- Based on the recurrence relation, we can determine what $t_{n/2}$ equals:

  $t_{n/2} = 2t_{n/4} + 2$ is the # of basic operations in each second level of recursion

- We can plug this in for $t_{n/2}$ in the original equation:

$t_n = 2(2t_{n/4} + 2) + 2$ or $2^2(t_{n/4}) + 2^2 + 2$

Min-Max Analysis

$t_n = 2^2(t_{n/4}) + 2^2 + 2$

- Next, we can determine how many basic operations take place in each third level of recursion, $t_{n/4}$

  $t_{n/4} = 2t_{n/8} + 2$

- Plugging this in to the original equation gives us:

Min-Max Analysis

$t_n = 2^2(t_{n/4}) + 2^2 + 2$

- Next, we can determine how many basic operations take place in each third level of recursion, $t_{n/4}$

  $t_{n/4} = 2t_{n/8} + 2$

- Plugging this in to the original equation gives us:

  $2^2\ (2t_{n/8} + 2) + 2^2 + 2$ or $2^3(t_{n/8}) + 2^3 + 2^2 + 2$

Min-Max Analysis

$2^3(t_{n/8}) + 2^3 + 2^2 + 2$ or $2^3(t_{n/2^3}) + 2^3 + 2^2 + 2$

- Is there a pattern here? (we are in the 3rd level of recursion)

Min-Max Analysis

$2^3(t_{n/8}) + 2^3 + 2^2 + 2$ or $2^3(t_{n/2^3}) + 2^3 + 2^2 + 2$

- Is there a pattern here? (we are in the 3rd level of recursion)

$2^k(t_{n/2^k}) + 2^k + \ldots + 2^2 + 2$ is the total # of basic operations that occur when we have k levels of recursion.

- When do we stop recursion?

Min-Max Analysis

$2^3(t_{n/8}) + 2^3 + 2^2 + 2$ or $2^3(t_{n/2^3}) + 2^3 + 2^2 + 2$

- Is there a pattern here? (we are in the 3rd level of recursion)

$2^k(t_{n/2^k}) + 2^k + \ldots + 2^2 + 2$ is the total # of basic operations that occur when we have k levels of recursion.

- When do we stop recursion? At the **base case**!

The base case is reached when a subarray passed to `MinMax` has 2 elements.

- i.e. at case t_2 . Based on the equation above, we know this case occurs when $n/2^k = 2$

Min-Max Analysis

When $n/2^k = 2$ we know that $n = ?$

Min-Max Analysis

When $n/2^k = 2$ we know that $n = 2^{k+1}$

If $n = 2^{k+1}$, we can take the lg of both sides to remove the exponent:

- $\lg n = \lg(2^{k+1})$
  $= (k+1)\lg 2$
  $= k + 1$ (since $\lg 2 = 1$)

Therefore $k = \lg n - 1$

$2^k(t_{n/2^k}) + 2^k + \ldots + 2^2 + 2$

$2^{\lg n - 1}(t_2) + 2^{\lg n - 1} + \ldots + 2^2 + 2$

Min-Max Analysis

Important math!

- Given this equality:

$$n^{k-1} + \ldots + n^2 + n^1 + 1 = (n^k - 1) / (n - 1)$$

- We can transform a sequence missing the final 1 by adding and subtracting a 1:

$$n^{k-1} + \ldots + n^2 + n^1 = (n^{k-1} + \ldots + n^2 + n^1 + 1) - 1 = [(n^k - 1) / (n - 1)] - 1$$

- We can make the following transformation to our equation, also missing the final 1:

$$2^{\lg n - 1} + \ldots + 2^2 + 2 = [(2^{\lg n} - 1) / (2 - 1)] - 1$$
$$= (2^{\lg n} - 1) - 1$$
$$= (n^{\lg 2} - 1) - 1$$
$$= (n - 2)$$

Min-Max Analysis

$2^{\lg n - 1}(t_2) + 2^{\lg n - 1} + \ldots + 2^2 + 2 =$

$2^{\lg n - 1}(1) + 2^{\lg n - 1} + \ldots + 2^2 + 2$ (since $t_2 = 1$)

- $2^{\lg n - 1} + \ldots + 2^2 + 2 = n - 2$ (from the previous slide)
- $2^{\lg n - 1} = 2^{\lg n} \times 2^{-1} = n / 2$

We end with $n / 2 + n - 2 = 1.5n - 2$

The time complexity of the divide and conquer Min-Max algorithm is $1.5n - 2$

Recurrence Relation #2

Solve the following recurrence relation:

$$t_n = 2t_{n-1} + 1 \qquad n > 1$$
$$t_1 = 1$$

What is the first step?

Recurrence Relation #2

Solve the following recurrence relation:

$$t_n = 2t_{n-1} + 1 \qquad n > 1$$
$$t_1 = 1$$

What is the first step? Find how many operations take place in the 2nd level of recursion, t_{n-1}

$$t_{n-1} = 2t_{n-2} + 1$$

What's next?

Recurrence Relation #2

Solve the following recurrence relation:

$t_n = 2t_{n-1} + 1 \qquad n > 1$

$t_1 = 1$

What is the first step? Find how many operations take place in the 2nd level of recursion, t_{n-1}

$t_{n-1} = 2t_{n-2} + 1$

What's next? Plug the calculation for t_{n-1} in to the equation for t_n:

$t_n = 2(2t_{n-2} + 1) + 1 \qquad \text{or} \qquad 4t_{n-2} + 2 + 1$

Recurrence Relation #2

Solve the following recurrence relation:

$$t_n = 2t_{n-1} + 1 \qquad n > 1$$
$$t_1 = 1$$

$t_n = 4t_{n-2} + 2 + 1$ at the 2nd level.

$t_{n-2} = 2t_{n-3} + 1$, which we plug in to the above equation:

$t_n = 4(2t_{n-3} + 1) + 2 + 1$ or $8t_{n-3} + 4 + 2 + 1$

Recurrence Relation #2

Solve the following recurrence relation:

$$t_n = 2t_{n-1} + 1 \qquad n > 1$$
$$t_1 = 1$$

$t_n = 8t_{n-3} + 4 + 2 + 1$ with the 3rd level calculated.

Based on the level we have calculated, what pattern emerges?

Recurrence Relation #2

Solve the following recurrence relation:

$$t_n = 2t_{n-1} + 1 \qquad n > 1$$
$$t_1 = 1$$

$t_n = 8t_{n-3} + 4 + 2 + 1$ with the 3rd level calculated.

Based on the level we have calculated, what pattern emerges?

$2^k t_{n-k} + 2^{k-1} + \ldots + 2^1 + 2^0$

When do we stop recursion?

Recurrence Relation #2

Solve the following recurrence relation:

$t_n = 2t_{n-1} + 1 \qquad n > 1$

$\boldsymbol{t_1 = 1}$

$t_n = 8t_{n-3} + 4 + 2 + 1$ with the 3rd level calculated.

Based on the level we have calculated, what pattern emerges?

$2^k t_{n-k} + 2^{k-1} + \ldots + 2^1 + 2^0$

When do we stop recursion? At t_1

i.e. when $n - k = 1$ or $k = n - 1$

Recurrence Relation #2

$2^k t_{n-k} + 2^{k-1} + \ldots + 2^1 + 2^0$

When do we stop recursion? At t_1

i.e. when $n - k = 1$ or $k = n - 1$

We can now make the necessary substitutions:

$2^{n-1} + 2^{n-2} + \ldots + 2^1 + 2^0 = (2^n - 1) / (2 - 1)$

$= \mathbf{2^n - 1}$

In-Class Exercise

Solve the following recurrence relation:

$$t_n = 8t_{n/2} + 3n$$

$$t_1 = 1$$

Binary Search

- Binary search can be implemented as a recursive divide-and-conquer algorithm.

Problem: Is x in the *sorted* array S?

If x equals the middle item, return true. Otherwise:

1. *Divide* the array into two subarrays about half as large. If x is smaller than the middle item, choose the left subarray. If x is larger than the middle item, choose the right subarray
2. *Conquer* the subarray by determining whether x is in that subarray. We do this recursively.
3. *Obtain* the solution to the top-level call from the solution to the recursive call.

Binary Search Top-Level Description

If $x = 18$, and we have the following array, we first see if the middle element equals x

10 12 13 14 18 20 25 27 30 35 40 45 47

Binary Search Top-Level Description

If $x = 18$, and we have the following array, we first see if the middle element equals x

10 12 13 14 18 20 **25** 27 30 35 40 45 47

The middle item is 25, which is not what x equals.

Binary Search Top-Level Description

If $x = 18$, and we have the following array, we first see if the middle element equals x

10 12 13 14 18 20 25 **27 30 35 40 45 47**

- We then divide our array up into 2 subarrays, made up of the values to the left and the right of the middle item we checked.
- Since $18 < 25$, we want to *conquer* the left subarray by passing it to a recursive call to `BinarySearch`.

Binary Search Top-Level Description

If $x = 18$, and we have the following array, we first see if the middle element equals x

10 12 13 14 18 20 25 27 30 35 40 45 47

- ➢ We conquer the left subarray by determining if 18 is in it.
- ➢ Since 18 **is** in this left subarray, we return true to the top-level call to `BinarySearch`, which then returns true.

Note: Although `BinarySearch` would take another recursive step (18 isn't the middle item of the subarray), we only describe what happens at the top level (i.e. recursively call `BinarySearch` on a subarray and wait for a response).

Binary Search In-Depth Description

1. Check the middle value:

10 12 13 14 18 20 **25** 27 30 35 40 45 47

2. 18 != 25. Divide into 2 subarrays:

10 12 13 14 18 20 25 **27 30 35 40 45 47**

3. 18 < 25, perform BinarySearch on the *left* subarray. Check the middle value:

10 12 **13** 14 18 20 25 27 30 35 40 45 47

4. 18 != 13. Divide into 2 subarrays:

10 12 13 **14 18 20** 25 27 30 35 40 45 47

5. 18 > 13, perform BinarySearch on the *right* subarray. Check the middle value:

10 12 13 14 **18** 20 25 27 30 35 40 45 47

6. 18 = 18, value found! Return true.

Binary Search Pseudocode

Problem: Determine whether x is in the sorted array S of size n

Parameters: Positive integer n, sorted array of keys S indexed from 1 to n, a key x

Outputs: The location of x in S, or 0 if x is not in S.

```
index BinarySearch(index low, index high)     // low = 1 and high = n at top level
     if (low > high)
          return 0;                           // key not found in array
     else
          index middle = ⌊(low + high) / 2⌋
          if (x == S[middle])
               return middle
          else if (x < S[middle])
               return BinarySearch(low, middle - 1)
          return BinarySearch(middle + 1, high)
```

Binary Search Analysis

```
index middle = ⌊(low + high) / 2⌋
if (x == S[middle])
     return middle
else if (x < S[middle])
     return BinarySearch(low, middle - 1)
return BinarySearch(middle + 1, high)
```

What is the basic operation?

Binary Search Analysis

```
index middle = ⌊(low + high) / 2⌋
if (x == S[middle])
     return middle
else if (x < S[middle])
     return BinarySearch(low, middle - 1)
return BinarySearch(middle + 1, high)
```

What is the basic operation? Although we have `x == S[middle]` and `x < S[middle]`, we count this as **one** operation.

➢ Remember, we always assume the comparisons have been implemented as efficiently as possible. In other words, we assume that in assembly code this if/if-else statement only requires one comparison.

Binary Search

- The recursive version of `BinarySearch` employs **tail-recursion**.
  - No operations are done after the recursive call.
- For this reason, developing an iterative version of `BinarySearch` is relatively straightforward.
- We discussed a recursive version of this algorithm since it clearly illustrates the divide-and-conquer process of dividing an instance into smaller instances.
- However, in many languages (such as C++) it is beneficial to use an iterative approach over a recursive approach.
  - Why?

Binary Search

- The recursive version of `BinarySearch` employs **tail-recursion**.
  - No operations are done after the recursive call.
- For this reason, developing an iterative version of `BinarySearch` is relatively straightforward.
- We discussed a recursive version of this algorithm since it clearly illustrates the divide-and-conquer process of dividing an instance into smaller instances.
- However, in many languages (such as C++) it is beneficial to use an iterative approach over a recursive approach.
  - Why?
  - An activation record is pushed to the stack for each recursive call.
    - Removing the need for adding these records to the stack increases both efficiency and memory usage.

In-Class Exercise

1. The comparison in the `else` statement is the basic operation. Analyze `BinarySearch` by finding its recurrence relation and determining its best-case and worst-case order.

```
index BinarySearch (index low, index high)          // low = 1 and high = n at top level
     if (low > high)
          return 0;                                 // key not found in array
     else
          index middle = ⌊(low + high) / 2⌋
          if (x == S[middle])
               return middle
          else if (x < S[middle])
               return BinarySearch(low, middle - 1)
          return BinarySearch(middle + 1, high)
```

2. Design an iterative version of the `BinarySearch` algorithm.