Backtracking with Recursion
Sorting

Nov 10

Finding a data item

- Suppose you have an array (or ArrayList) of N items. How do you determine if the array contains a particular item?
  - Does the form of the array matter?
    - Unsorted
    - Sorted
    - Heap
  - What is the complexity of finding an item?

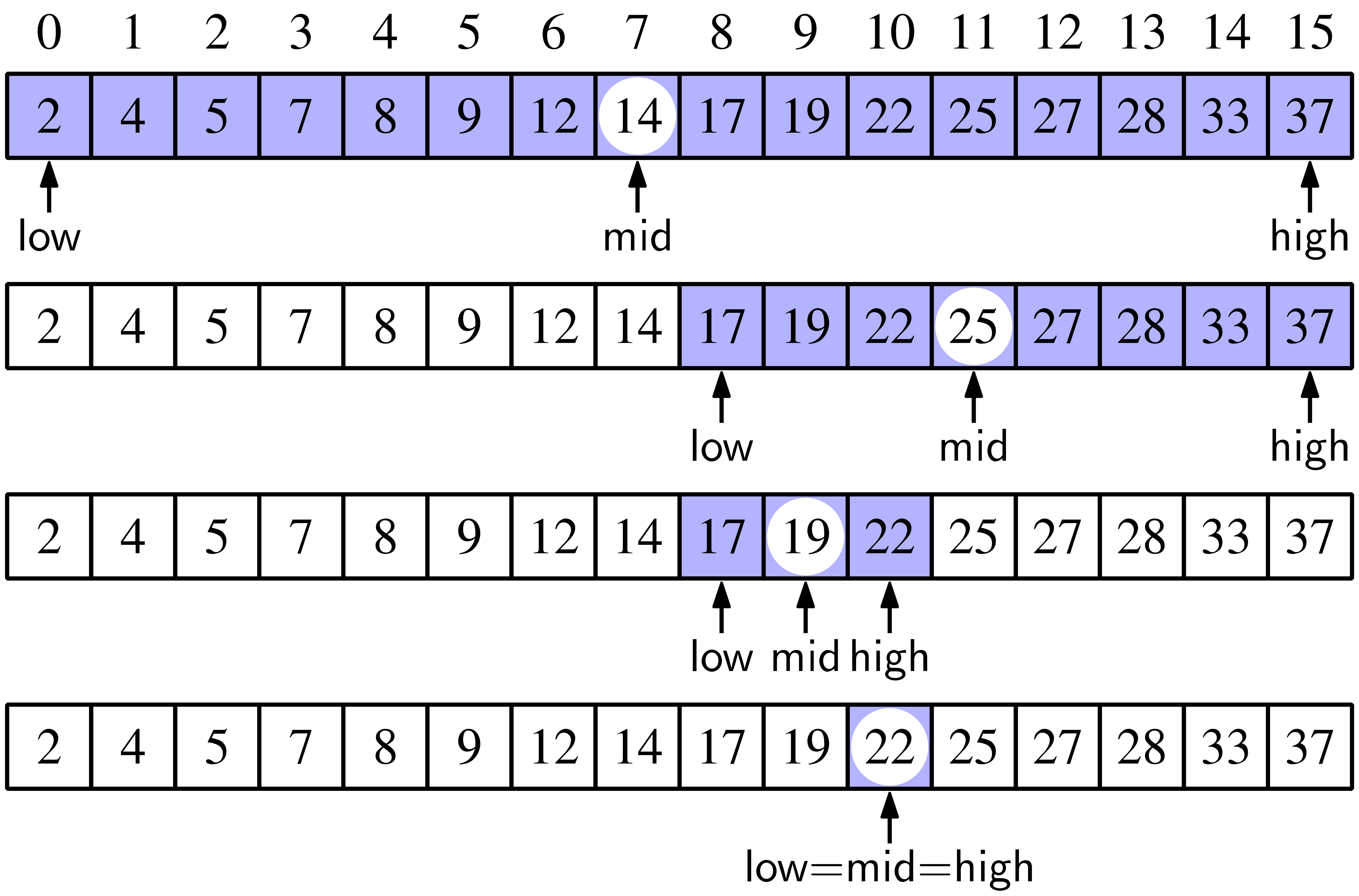
Binary Search

- Search for an integer (22) in an ordered list

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

- $mid = \left\lfloor \frac{low + high}{2} \right\rfloor = \left\lfloor \frac{0 + 15}{2} \right\rfloor = 7$
  - `target==data[mid]`, found
  - `target>data[mid]`, recur on second half
  - `target<data[mid]`, recur on first half

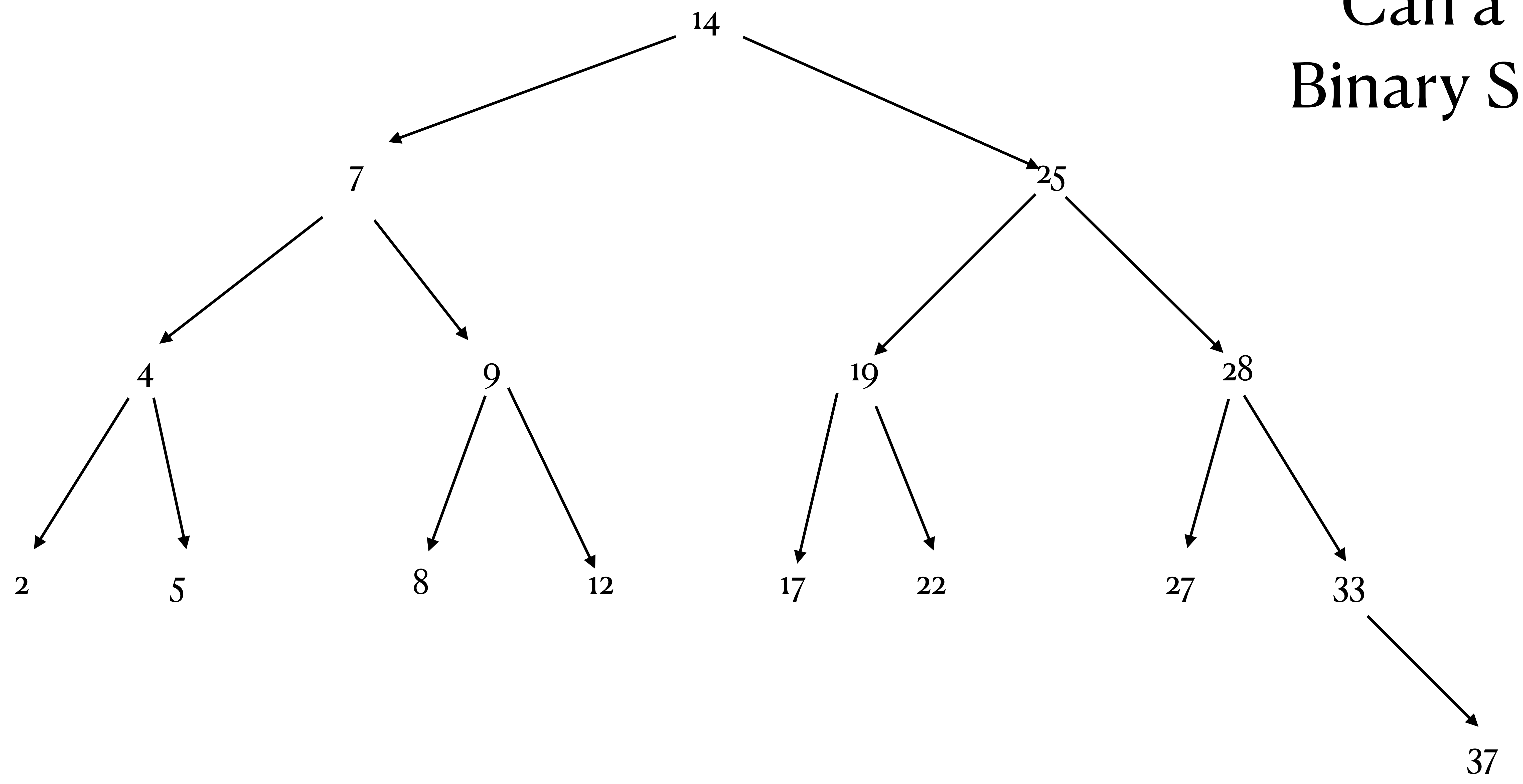
target = 22


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low=mid=high

View the data as a binary tree

"Binary Search Tree"

Is this a heap?
Can a heap be a
Binary Search Tree?



Binary Search Code

```java
/**
* The public facing call to array search
* The array to be searched is a private instance variable
* @param target the value being searched for
* @return true if the value is in known, false otherwise
*/
public boolean contains(int target) {
    if (data==null)
        return 0;
    return iSearch(target, 0, data.length-1, 0);
}
```

Suppose change instance variable data to ArrayList?

Binary Search Code

```
/**
  * Binary search, recursively on sorted internal array of ints
  * @param target the item to be found
  * @param lo the bottom of the range being searched
  * @param hi the top of the range being searched
  * @param steps the number of steps the search has taken
  * @return true if the target was found
  */
private boolean iSearch(int target, int lo, int hi, int steps) {
    if (lo>hi) return false;
    int mid = (lo+hi)/2;
    System.out.println(target + " " + data[mid] + " " + lo + " " + hi + " " + step
    if (data[mid]==target) return true;
    if (data[mid]<target)
        return iSearch(target, mid+1, hi, steps+1);
    else
        return iSearch(target, lo, mid-1, steps+1);
 }
```

Binary Search Analysis

- Each recursive call divides the array in half
- If the array is of size n, it divides (and searches) at most $\log_2 n$ times before the current half is of size 1
- $O(\log_2 n)$

Reimplement Binary search with iteration

What parameters does the iterative method need?
Does a separate private method even make sense?

Backtracking with Recursion

- Previous examples all progressed linearly to success/failure
- So consider doing binary like search on an unsorted array
  - Need to backtrack and try other directions on failure.
  - Backtracking is when recursion really shines

Recursion and Backtracking

- All examples progress steadily towards an answer.
- Consider a maze. Sometimes you need to backtrack.
  - Will work with maze code in lab.

- Problem: given an english word can you remove one letter and still have an english word.
  - Can you do this repeatedly until only a 2 letter word remains?
  - Consider the word "cored" .. core, ore, or!!!
  - Lets suppose:
    - boolean isInEnglish(String s)
      - return true iff s is an English word
    - String removeNchar(int n, String s)
      - removes the nth character of the string. So removeNchar(0, "dour") is "our"

Base Cases for Word reducer

Word Reducer

```
public class Worder {
    HashMap<String, Integer> words;
    public Worder() {
        words = new HashMap<>();
        try (BufferedReader br = new BufferedReader(new FileReader("/usr/share/dict/words"))) {
            String l;
            while (null != (l=br.readLine())) {
                words.put(l.trim(), 1);
            } }
        catch (Exception ee) {  ee.printStackTrace();   }
    }
    public boolean isEnglish(String s) {
        return words.containsKey(s);
    }
    String removeNchar(int n, String s) {
        if (n>0 && n<s.length()-1) {
            return s.substring(0,n)+s.substring(n+1);
        }
        if (n==0)
            return s.substring(1);
        return s.substring(0, s.length()-1);
    }
    public boolean reducable(String s) {
        System.out.println(s);
        if (s.length()<=2) { return isEnglish(s); }
        if (!isEnglish(s)) return false;
        for (int i=0; i<s.length(); i++) {
            if (reducable(removeNchar(i, s)))
                return true;
        }
        return false;
    }
```

Sorting

- public void sort(Comprable[] arra)
    - change the order of the items in arra
    - All examples will use integers but same statements apply to any Comparable object
    - ideally, do this “in place”.
        - That is do not use any extra memory

- First 3 sort techniques we have already discussed

Selection Sort

- Selection-sort:
  - select the min/max and swap with 0
- priority queue is implemented with an unsorted sequence
- Time:
  - Add: O(n)
  - Remove: O(n^2)

Example

Phase 1 — Inserting

(a)	7	(7)	
(b)	4	(7,4)	
….			
(g)	()		(7,4,8,2,5,3,9)

Phase 2 — Polling

(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion Sort

- Insertion-sort:
  - insert/swap the element into the correct sorted position
- Priority queue where the priority queue is implemented with a sorted sequence
- Time:
  - Add:O(n^2)
  - Remove: O(n)

Example

Phase 1 — Inserting

(a)	7	(7)
(b)	4	(4,7)
(c)	8	(4,7,8)
(d)	2	(2,4,7,8)
(e)	5	(2,4,5,7,8)
(f)	3	(2,3,4,5,7,8)
(g)	9	(2,3,4,5,7,8,9)

Phase 2 — polling

(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Heap Sort

- Heap-sort:
  - Insertion — no more than $\log_2(n)$ steps per insertion
  - Deletion — no more than $\log_2(n)$ steps per deletion
- priority queue is implemented with a heap

- Time:
  - Add:O(n * $\log_2(n)$) — under some assumptions O(n).
  - Remove: O(n * $\log_2(n)$)
- Note: with a **lot of work** can do this without an additional array.

Example

Phase 1 — Inserting

(a)	7	(7)
(b)	4	(4,7)
(c)	8	(4,7,8)
(d)	2	(2,4,8,7)
(e)	5	(2,4,8,7,5)
(f)	3	(2,4,3,7,5,8)
(g)	9	(2,4,3,7,5,8,9)

Phase 2 — polling

(a)	(2)	(3,4,7,5,8,9)
(b)	(2,3)	(4,5,7,9,8)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Timing

Table 1

size	selection	Insertion	Insertion	Heap
1000	16	15	11	2
2000	8	12	26	3
4000	24	23	20	5
8000	96	95	81	10
16000	370	378	315	17
32000	1585	1359	1218	36
64000	5771	5590	4605	77
128000	23087	21547	19849	161
256000				345
512000				1128
1024000				1973
2048000				3225
4096000				7577
8192000				18586

10000==1 second

Batch and Flow

- Another consideration is how the you get the data and when you produce the sort.
  - BATCH
    - you get the data all at once and have to produce a sorted list at the time (or later)
  - FLOW — the data come in over the course of time. At any time, you can be asked to produce a sort list of the data yo already have.

- Which sorts algorithms perform well / poorly in each situation
  - suppose receive M requests for sort and assume that new data — of size N — arrives unpredicatbly with respect to M.
    - Selection sort: insert=O(n), retrieve=O(M*N^2)
    - Insertion Sort: insert=O(N * N^2) retrieve = O(M)
    - Heap Sort: insert: O(N * logN) retrieve: O(M * N * logN)

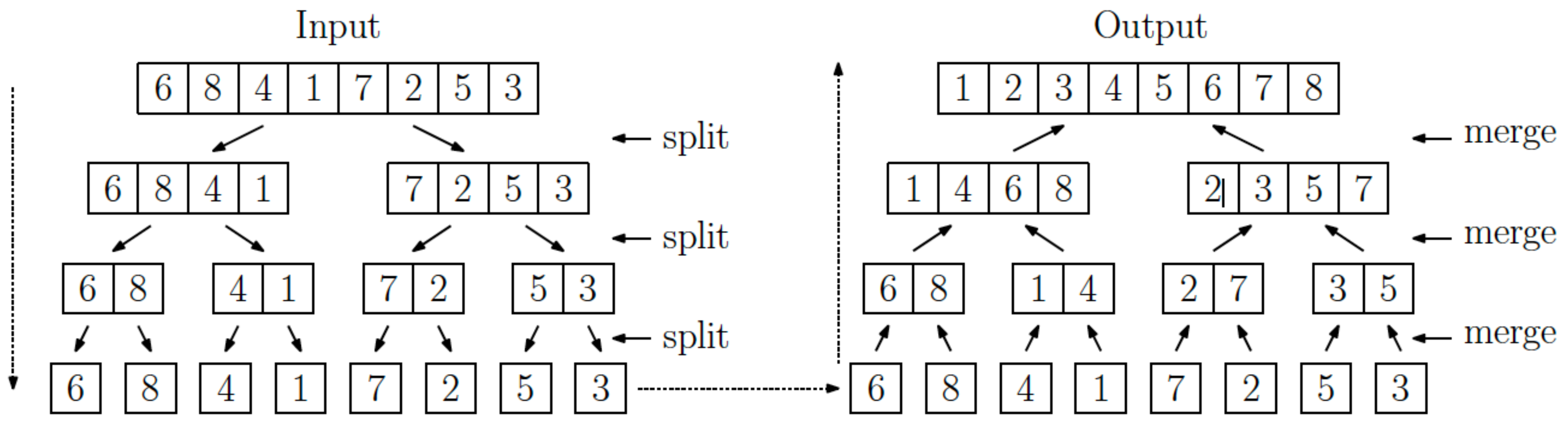
Divide-and-Conquer

- Divide – the problem (input) into smaller pieces
- Conquer – solve each piece individually, usually recursively
- Combine – the piecewise solutions into a global solution (if needed)
- Usually involves recursion

Merge Sort

- Sort a sequence of numbers A, $|A| = n$
- Base case: $|A| = 1$, then it's already sorted
- General
  - divide: split A into two halves, each of size $\frac{n}{2}$ ($\left\lfloor \frac{n}{2} \right\rfloor$ and $\left\lceil \frac{n}{2} \right\rceil$)
  - conquer: sort each half (by calling mergeSort recursively)
  - combine: merge the two sorted halves into a single sorted list

Example


Input
6 8 4 1 7 2 5 3
split
6 8 4 1
7 2 5 3
split
6 8
4 1
7 2
5 3
split
6
8
4
1
7
2
5
3
Output
1 2 3 4 5 6 7 8
merge
1 4 6 8
2 3 5 7
merge
6 8
1 4
2 7
3 5
merge
6
8
4
1
7
2
5
3

Algorithm

```
mergeSort(S):
  if S.size() <= 1 return
  else
    s1 = S[0,n/2]
    s2 = S[n/2+1,n-1]
    mergeSort(s1)
    mergeSort(s2)
    S = merge(s1, s2)
```

Merge Algorithm

- The key is the merging process
- How does one merge two sorted lists?
- Each element in $A \bigcup B$ is considered once
- $O(n)$

```
Algorithm merge(A, B)
  Input sorted A and B
  Output sorted A ∪ B
  S = empty sequence
  while(!A.isEmpty() and
        !B.isEmpty())
   if A.first() < B.first()
      S.addLast(A.removeFirst())
   els
      S.addLast(B.removeFirst())
 while (!A.isEmpty())
 S.addLast(A.removeFirst())
 while (!B.isEmpty())
 S.addLast(B.removeFirst())
 return S
```

Merge (in Java)

```java
private int[] domerge(int[] list1, int[] list2) {
    int[] rtn = new int[list1.length + list2.length];
    int locr=0, loc1=0, loc2=0;
    while (loc1<list1.length && loc2<list2.length)  {
        if (list1[loc1] < list2[loc2])
        {
        rtn[locr++]=list2[loc2++];
        }
        else
        rtn[locr++]=list1[loc1++];
    }
    for (int i=loc1; i<list1.length; i++)
        rtn[locr++]=list1[i];
    for (int i=loc2; i<list2.length; i++)
        rtn[locr++]=list2[i];
    return rtn;
    }
```

MergeSort

```
    public int[] mergesort(int[] list) {
        return doMergeSort(list, 0, list.length-1);
    }

    private int[] doMergeSort(int[] list, int strt, int eend)
    {
    if (eend==strt)
    {
        int[] tmp = new int[1];
        tmp[0]=list[strt];
        return tmp;
    }
    if (eend<strt)
        return new int[0];
    int mid = (strt+eend)/2;
    return domerge(mergesort(list, strt, mid), mergesort(list, mid+1, eend));
    }
```

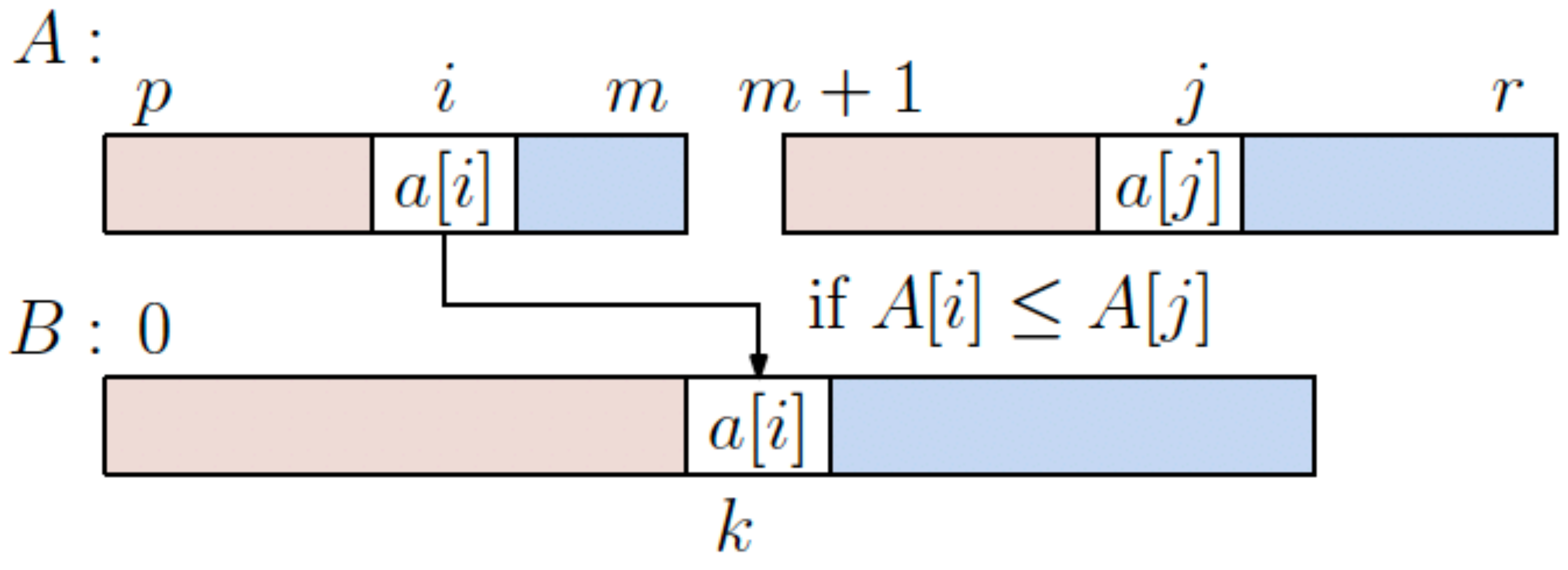
Timing

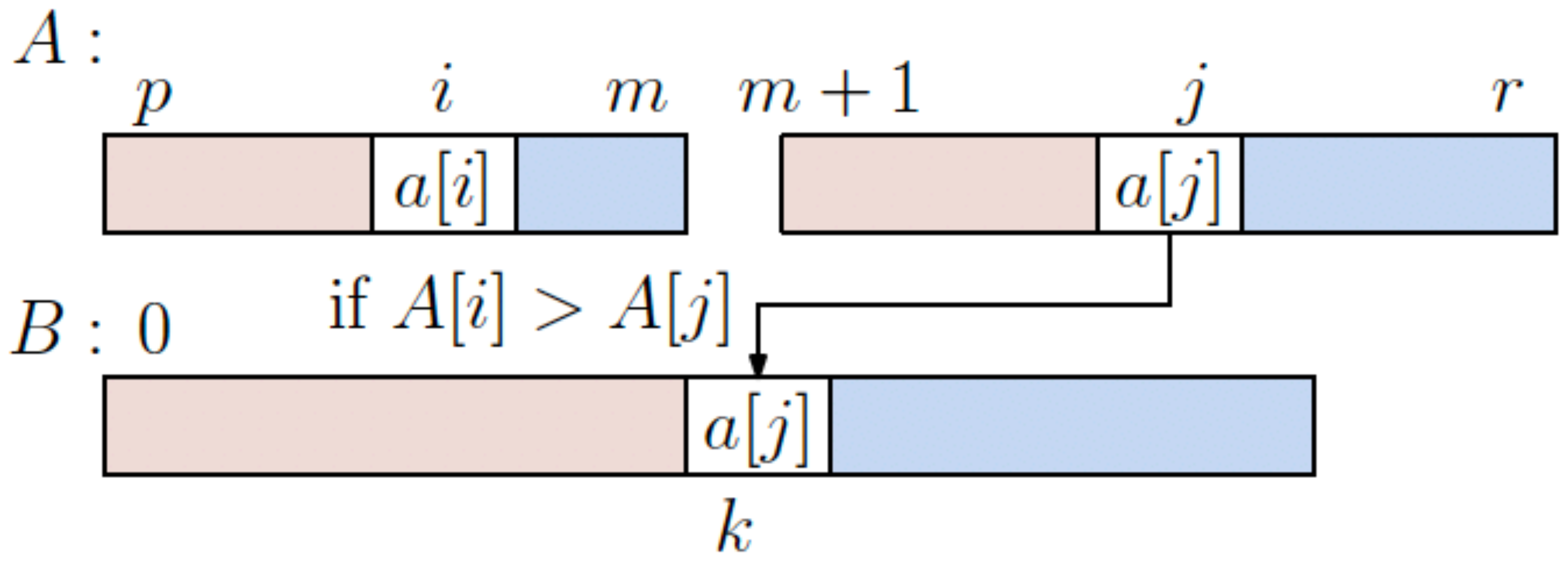
Table 1

size	selection	Insertion	Insertion	Heap	merge
1000	16	15	11	2	3
2000	8	12	26	3	3
4000	24	23	20	5	7
8000	96	95	81	10	13
16000	370	378	315	17	27
32000	1585	1359	1218	36	58
64000	5771	5590	4605	77	119
128000	23087	21547	19849	161	219
256000				345	372
512000				1128	776
1024000				1973	1631
2048000				3225	3822
4096000				7577	6772
8192000				18586	14159

In-place Merge

- Making new lists is slow!
- How does one merge two sorted lists `A[p,…,m]` and `A[m+1,…,r]`?
- Use a temp array `B` and maintain two indices `i` and `j`, one for each subarray





MergeSort using one temp array

```
    private int[] array;
    private int[] tempMergArr;
    private int length;
    public int[] mergesort3(int inputArr[]) {
        this.array = inputArr;
        this.length = inputArr.length;
        this.tempMergArr = new int[length];
        doMergeSort3(0, length - 1);
        return array;
    }

    private void doMergeSort3(int lowerIndex, int higherIndex) {

        if (lowerIndex < higherIndex) {
            int middle = lowerIndex + (higherIndex - lowerIndex) / 2;
            // Below step sorts the left side of the array
            doMergeSort3(lowerIndex, middle);
            // Below step sorts the right side of the array
            doMergeSort3(middle + 1, higherIndex);
            // Now merge both sides
            mergeParts3(lowerIndex, middle, higherIndex);
        }
    }
```

Merge with temp array

```
    private void mergeParts3(int lowerIndex, int middle, int higherIndex) {

        for (int i = lowerIndex; i <= higherIndex; i++) {
            tempMergArr[i] = array[i];
        }
        int i = lowerIndex;
        int j = middle + 1;
        int k = lowerIndex;
        while (i <= middle && j <= higherIndex) {
            if (tempMergArr[i] <= tempMergArr[j]) {
                array[k] = tempMergArr[i];
                i++;
            } else {
                array[k] = tempMergArr[j];
                j++;
            }
            k++;
        }
        while (i <= middle) {
            array[k] = tempMergArr[i];
            k++;
            i++;
        }
    }
```

Timing

Table 1

size	selection	Insertion	Insertion	Heap	merge	merge (improved)
1000	16	15	11	2	3	2
2000	8	12	26	3	3	3
4000	24	23	20	5	7	7
8000	96	95	81	10	13	9
16000	370	378	315	17	27	16
32000	1585	1359	1218	36	58	32
64000	5771	5590	4605	77	119	69
128000	23087	21547	19849	161	219	143
256000				345	372	294
512000				1128	776	563
1024000				1973	1631	1191
2048000				3225	3822	2412
4096000				7577	6772	5191
8192000				18586	14159	10282

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort		▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort		▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort		▪ fast ▪ in-place ▪ for large data sets (1K — 1M)
merge-sort		▪ fast ▪ sequential data access ▪ for huge data sets (> 1M)

Mini Homework

14, 6, 18, 2, 13, 7, 8, 9, 3, 17, 5, 10, 11, 12, 15, 19, 16, 0, 1, 4

For the data above, show all the steps of a merge sort, along the lines of slide 12.

Quick Sort

- Another divide-and conqueror sort
  - divide: pick a random element x (pivot) and partition into
    - L: $< x$
    - E: $= x$
    - G: $> x$
  - conquer: sort L and G
  - combine: join L, E and G

Pseudo Code

```
quickSort(S):
  if (S.size()<2) return
  p = S.last()                    // first as pivot
  L = E = G = new list()
  partition(S, p)
  quickSort(L)
  quickSort(G)

  S = L+E+G
```

QS in action

Partition (pink)

{7, 2, 9, 4, 3, 7, 1, 6}

Partition on 6

{2, 4, 3, 1}   6   {7, 7, 9}

Partition on 1   Partition on 9

{} 1 {2, 4, 3}   {7, 7} 9 {}

Partition on 3   Partition on 7

{2} 3 {4}   {7} 7 {}

Divide

Assemble (aqua)

{2,3,4}   {7,7}

{1,2,3,4}   {7,7,9}

{1,2,3,4,6,7,7,9}

Conquer

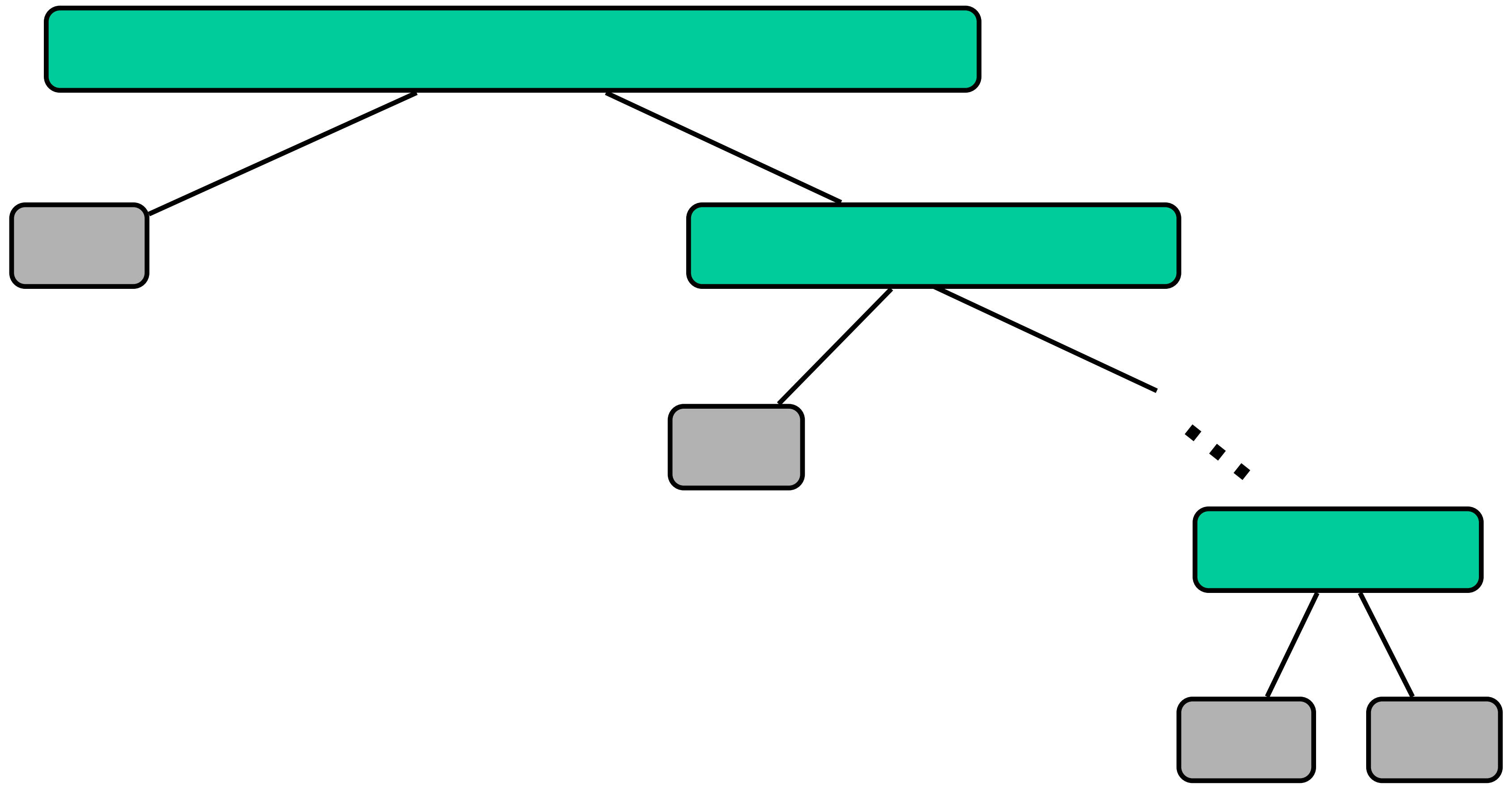
MergeSort, Quicksort, etc

- Quicksort does work on way down in recursion
  - Mergesort does work on way up
- Insertion sort does work on way down
  - Selection sort on way up
- Which one in faster Quick or Merge?

Worst-case Running Time

- When the pivot is the min or max
  - one of L or G has size $n-1$
  - $T(n) = n + (n-1) + \ldots + 2 + 1 = O(n^2)$

depth	time
0	$\boldsymbol{n}$
1	$\boldsymbol{n}-1$
…	…
$\boldsymbol{n}-1$	1



In-place Quick Sort

- instead of three lists partition rearranges the input list
  - $L: \left[0, l-1\right]$
  - $E: \left[l,\ r\right]$
  - $G: \left[r+1,\ n-1\right]$
- Recursive calls on $[0,\ l-1]$ and $[r+1,\ n-1]$

Partition

```
public int partition(int arr[], int begin, int end)  {
    int pivot = arr[end];
    int insertLoc = (begin-1);
    for (int j = begin; j < end; j++) {
        if (arr[j] <= pivot) {
            insertLoc++;
            int swapTemp = arr[insertLoc];
            arr[insertLoc] = arr[j];
            arr[j] = swapTemp;
        }
    }
    int swapTemp = arr[insertLoc+1];
    arr[insertLoc+1] = arr[end];
    arr[end] = swapTemp;
    return insertLoc+1;
}
```

QuickSort

```
private void quickSort(int arr[], int begin, int end)
{
    if (begin < end) {
        int partitionIndex = partition(arr, begin, end);

        quickSort(arr, begin, partitionIndex-1);
        quickSort(arr, partitionIndex+1, end);
    }
}
```

Speed

Table 1

size	Insertion	Heap	merge (improved)	Quick
1000	11	2	2	1
2000	26	3	3	1
4000	20	5	7	2
8000	81	10	9	5
16000	315	17	16	13
32000	1218	36	32	30
64000	4605	77	69	59
128000	19849	161	143	108
256000		345	294	219
512000		1128	563	464
1024000		1973	1191	955
2048000		3225	2412	1989
4096000		7577	5191	4148
8192000		18586	10282	10101
16384000				17614
32768000				37291

Quick and Merge

- Quicksort is reliably quicker than merge
- Quicksort does not need extra memory for auxiliary array

Mini Homework

14, 6, 18, 2, 13, 7, 8, 9, 3, 17, 5, 10, 11, 12, 15, 19, 16, 0, 1, 4

For the data above, show all the steps of a quick sort, following the pattern of slide 19. Always choose the last element as the partitioning element.