# Data Structures and Algorithms for Information Processing

Lecture 13: Sorting II

# Outline

- Brief review from last time
- Radix sorting and indexing
- Recursive sorting algorithms
- Quicksort

# Sorting Demonstration

http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html

# Intuitive Introduction

Main’s slides from Chapter 12

# Insertion Sort

```
void insertionSort(int A[]) {
   for(int i=1; i<A.length; i++)
      for(int j=i; j>0 && A[j]<A[j-1]; j--)
            swap(A[j],A[j-1]);

}
```

- Worst case runs in $O(n^2)$, where n = A.length.
- Best case, A is sorted already, runs in O(n).
- Use if you're in a hurry to code it , and speed is not an issue.

# What is the Average Disorder?

***Theorem***: The average disorder for a sequence of n items is $n(n-1)/4$

***Proof***: Assume all permutations of array A equally likely. If $A^R$ is the reverse of A, then disorder(A) + disorder($A^R$) = $n(n-1)/2$ because $A[i]<A[j]$ iff $A^R[i]>A^R[j]$. Thus the average disorder over all permutations is $n(n-1)/4$.

***Corollary***: The average running time of *any* sorting program that swaps only adjacent elements is $O(n^2)$.

***Proof***: It will have to do $n(n-1)/4$ swaps and may waste time in other ways.

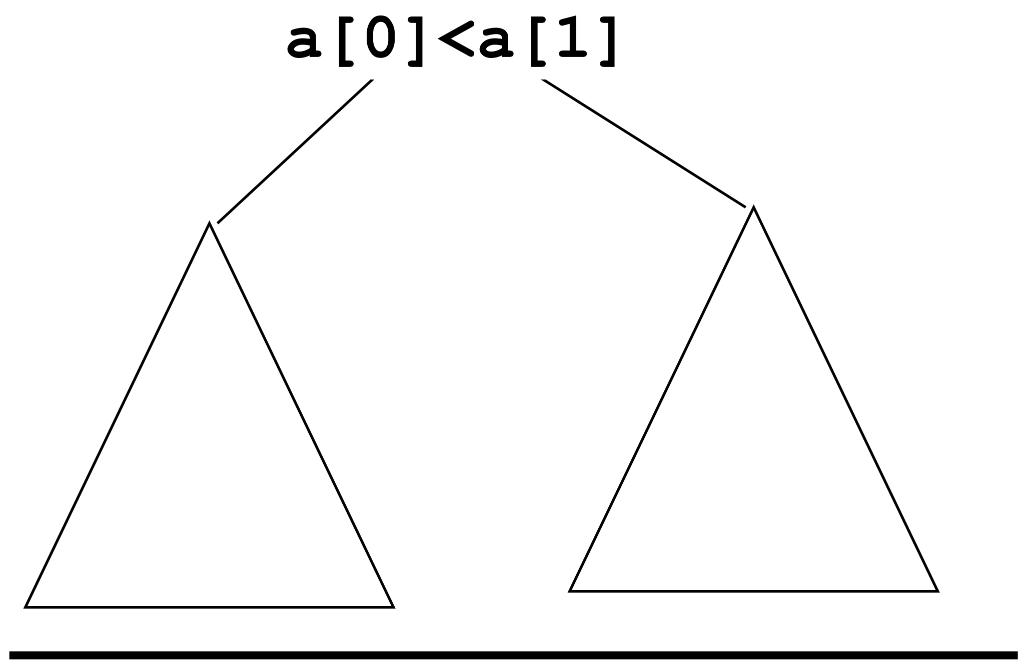
# To better $O(n^2)$ we must compare non-adjacent elements

Shell Sort: Swap elements n/2, n/4, … apart

Heap Sort: Swap A[i] with A[i/2]

QuickSort: Swap around "median"

# How many leaves must there for a sorting tree for n items?




n!, the number of different permutations.

So a tree with n! leaves has depth at least lg n!.

Notice that depth = the maximum number of tests one might have to perform.

$$\begin{aligned}
\lg n! &= \lg n(n-1)(n-2)\ldots 1 \\
&= \lg n + \lg n-1 + \lg n-2 + \ldots + \lg 1 \\
&\geq \lg n + \ldots + \lg(n/2) \\
&\geq (n/2) \lg(n/2) \\
&\geq (n/2) \lg n - n/2 \\
&= \Omega(n \lg n)
\end{aligned}$$

So *any* sort algorithm takes $\Omega$(n lg n) comparisons.

# Is there a way to sort without using binary comparisons?

Ternary comparisons, K-way comparisons.

The basic $\Omega$(n log n) result will still be true, because $\Omega(\log_2 x) = \Omega(\log_k x)$.

Useful speed-up heuristic: use your data as an index of an array.

# Consider sorting letters

```
int counts[26];
int j = 0;
for(int i=0; i<26; i++) counts=0;
for(j=0; j<clist.length; j++)
     count[clist[j]-'a']++;
j=0;
for(int i=0; i<26; i++)
  while(count[i]-- > 0) clist[j++]=i+'a';
```

# Sorting list of letters

```
int counts[26];
int j = 0;
for(int i=0; i<26; i++) counts=0;
for(j=0; j<clist.length; j++)
          count[clist[j]-'a']++;
j=0;
for(int i=0; i<26; i++)
   while(count[i]-- > 0) clist[j++]=i+'a';
```

if clist = "abbcabbdaf"

count = {3,4,1,1,0,1,0, …, 0}

and new  clist = "aaabbbbcdf"

Running time is O(26+clist.size()), i.e. ***linear***!

# Why does this beat n log n?

- The operation `count[clist[j]]++` is like a 26-way test; the outcome depends directly on the data.
- This is "cheating" because it won't work if the data range grows from 26 to $2^{32}$.
- Technique can still be useful — can break up range into "buckets" and use mergesort on each bucket

# Radix Sort

A way to exploit the data-driven idea for large data spaces.

Idea: Sort the numbers by their *lowest* digit. Then sort them by the next lowest digit, being careful to break ties properly. Continue to highest digit.

| | | | | | |
|---|---|---|---|---|---|
| 4567 | 3480 | 1908 | 2009 | 109 | |
| 2132 | 9241 | 109 | 109 | 456 | |
| 456 | 8721 | 2009 | 2132 | 1908 | |
| 1908 | 3521 | 8721 | 9241 | 2009 | |
| 3456 | 2132 | 3521 | 3297 | 2132 | |
| 9241 | 456 | 2132 | 456 | 3297 | |
| 109 | 3456 | 9241 | 3456 | 3456 | |
| 5789 | 4567 | 456 | 3480 | 3480 | |
| 3297 | 3297 | 3456 | 3521 | 3521 | |
| 2009 | 1908 | 4567 | 4567 | 4567 | |
| 8721 | 109 | 3480 | 8721 | 5789 | |
| 3521 | 5789 | 5789 | 5789 | 8721 | |
| 3480 | 2009 | 3297 | 1908 | 9241 | |
| | | | | | |
| | | | | | |
| | | | | | |

# Radix Sort

- Each sort must be ***stable***

  The relative order of equal keys is preserved
- In this way, the work done for earlier bits is not "undone"

# Radix Sort

Informal Algorithm:

To sort items A[i] with value 0…$2^{32}$-1 (= INT_MAX)

- Create a table of 256 buckets.
- {For every A[i] put it in bucket A[i] mod 256.
- Take all the items from the buckets 0,…, 255 in a FIFO manner, re-packing them into A.}
- Repeat using A[i]/256 mod 256
- Repeat using A[i]/$256^2$ mod 256
- Repeat using A[i]/$256^3$ mod 256
- This takes O(4*(256+A.length))

# Radix Sort using Counts

The Queues can be avoided by using counts.

```
void RadixSort(int a[], int b[], int N) {
   int i, j, pass, count[M];
   for (pass=0; pass < (w/m); pass++) {
      for (j=0; j < M; j++) count[j] = 0;
      for (i=1; i <= N; i++)
         count[a[i].bits(pass*m, m)]++;
      for (j=1; j < M; j++)
         count[j] = count[j-1] + count[j];
      for (i=N; i >= 1; i--)
         b[count[a[i].bits(pass*m,m)]--] =
                                          a[i];
      for (i=1; i <= N; i++) a[i] = b[i];
   }
}
```

# Radix Sort using Queues

```
const int BucketCount = 256;
void RadixSort(vector<int> &A) {
 vector<queue<int> > Table(BucketCount);
 int passes = ceil(log(INT_MAX)/log(BucketCount));
 int power = 1;
 for(int p=0; p<passes;p++) {
    int i;
    for(i=0; i<A.size(); i++) {
         int item = A[i];
         int bucket = (item/power) % BucketCount;
         Table[bucket].push(item);
    }
    i =0;
    for(int b=0; b<BucketCount; b++)
      while(!Table[b].empty()) {
         A[i++] = Table[b].front(); Table[b].pop();
      }
    power *= BucketCount;
 }}
```

# Radix Sort

In general it takes time

O( Passes*(NBuckets+A.length))

where Passes= $\lceil$log(INT_MAX)/log(NBuckets)$\rceil$

It needs O(A.length) in extra space.

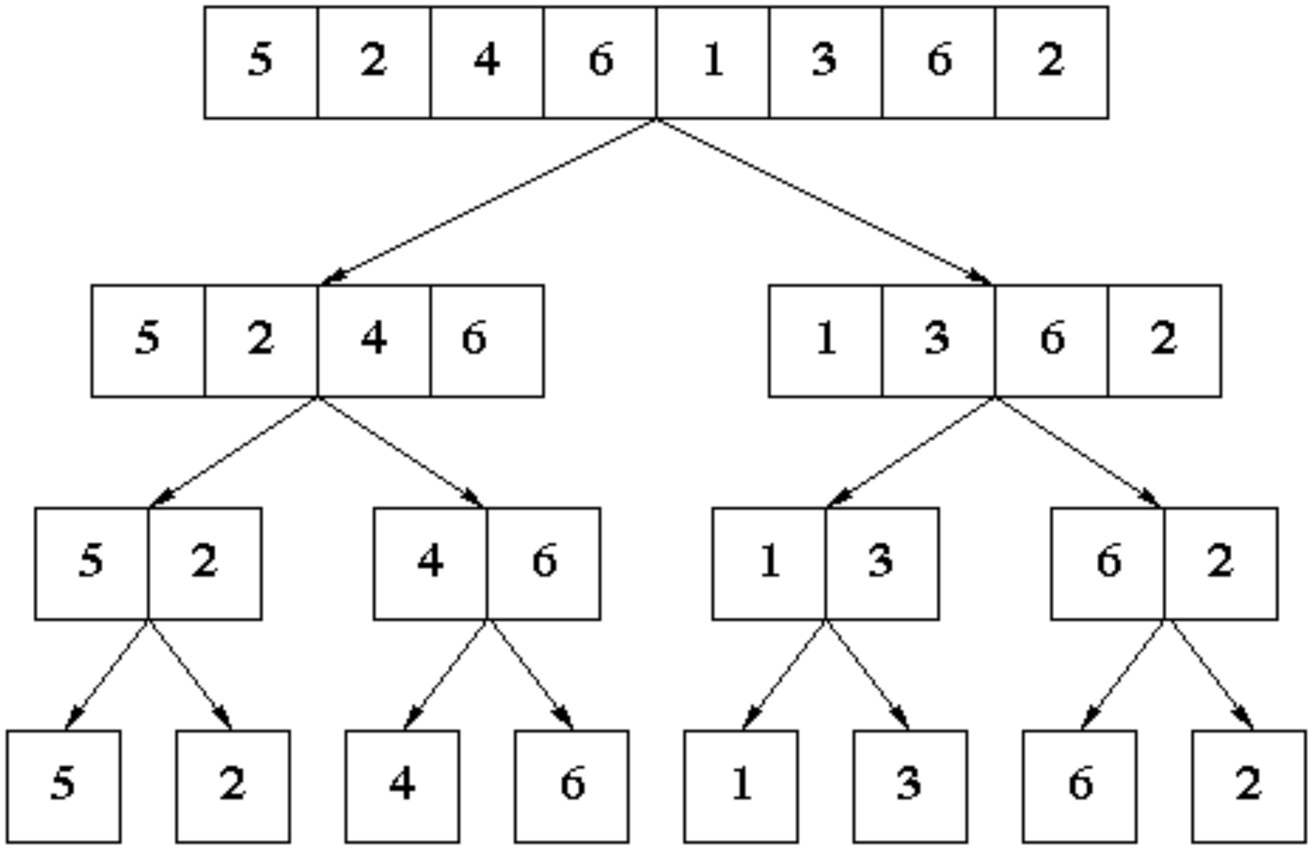
# Idea of Merge Sort

- Divide elements to be sorted into two groups of equal size
- Sort each half
- Merge the results using a simultaneous pass through each

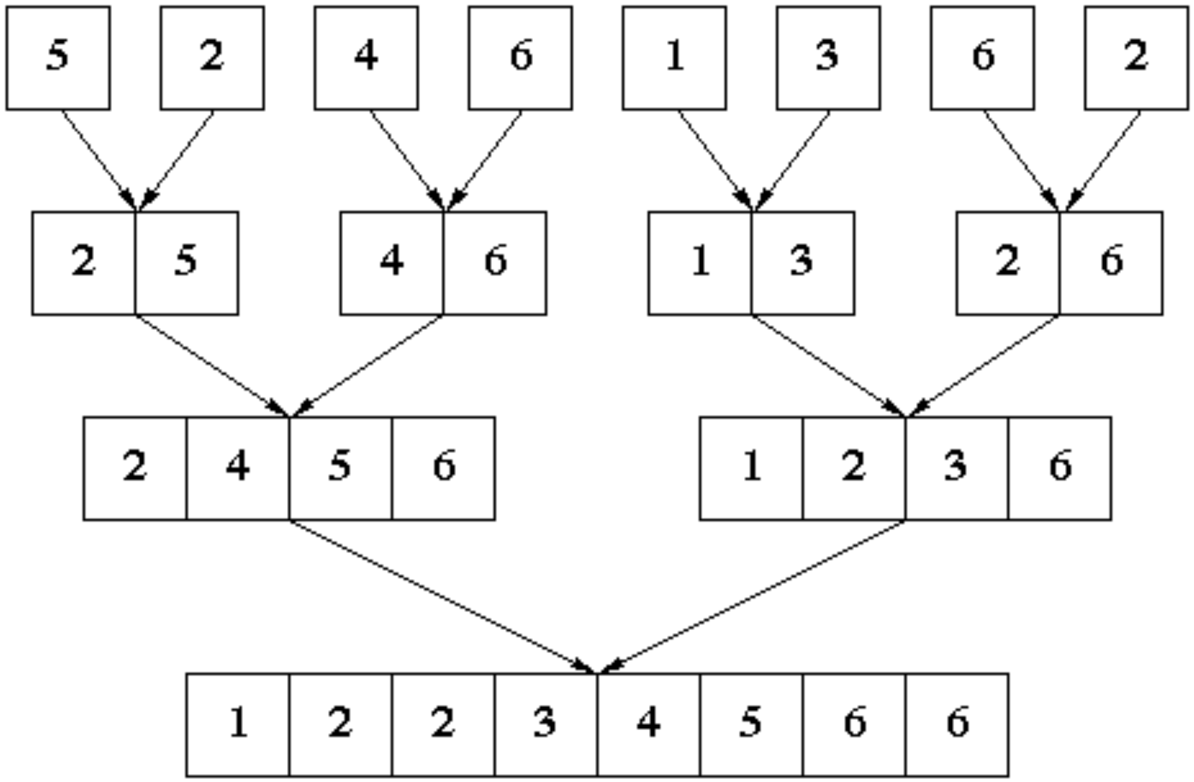
# Psuedocode for Merge Sort

```
void mergesort(int data[], int first, int n) {
  if (n > 1) {
      int n1 = n/2;
      int n2 = n - n1;
      mergesort(data, first, n1);
      mergesort(data, first+n1, n2);
      merge(data, first, n1, n2);
  }
}
```

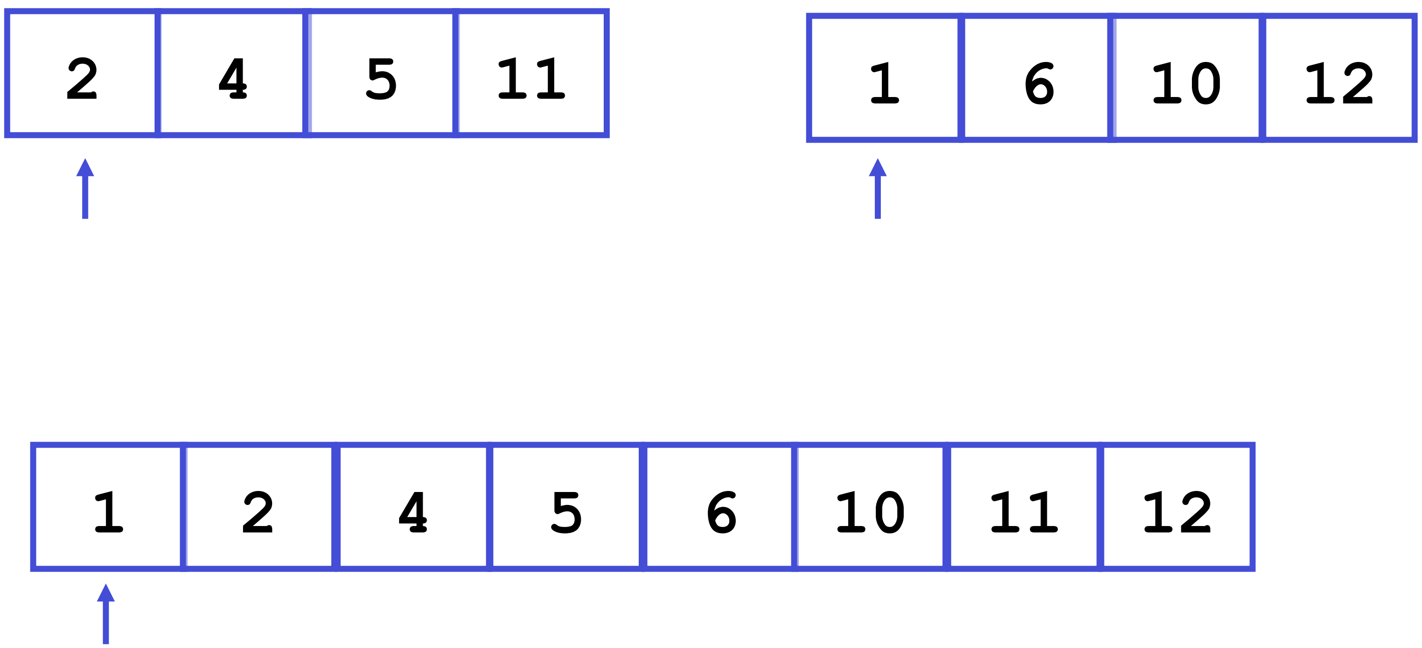
# Mergesort in Action



# Mergesort in Action



# The Merge Operation



# Mergesort Performance

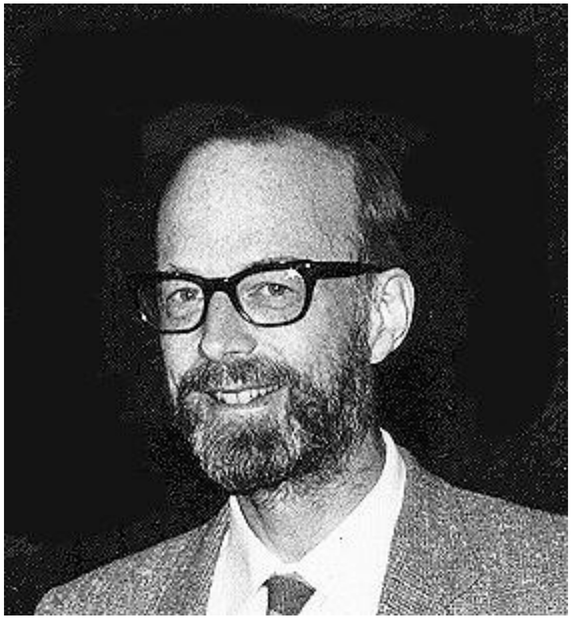
- The worst-case, average-case, and best-case running time for mergesort are all O(n log n)
- The basic idea:
  - by dividing in half we do O(log n) merges
  - Each merge requires linear time

# General Recursive Strategy to Sort List L

- If L has zero or one element, we' re finished
- Otherwise
  - divide L into two smaller lists L1, L2
  - recursively sort each of the smaller lists
  - combine L1 and L2
- So far have considered merge combination method
- Next we' ll consider "joining" the two lists

# Quicksort

- First devised by the computer scientist  C.A.R. Hoare
- One of the most effective algorithms in practice, though quadratic in the worst case



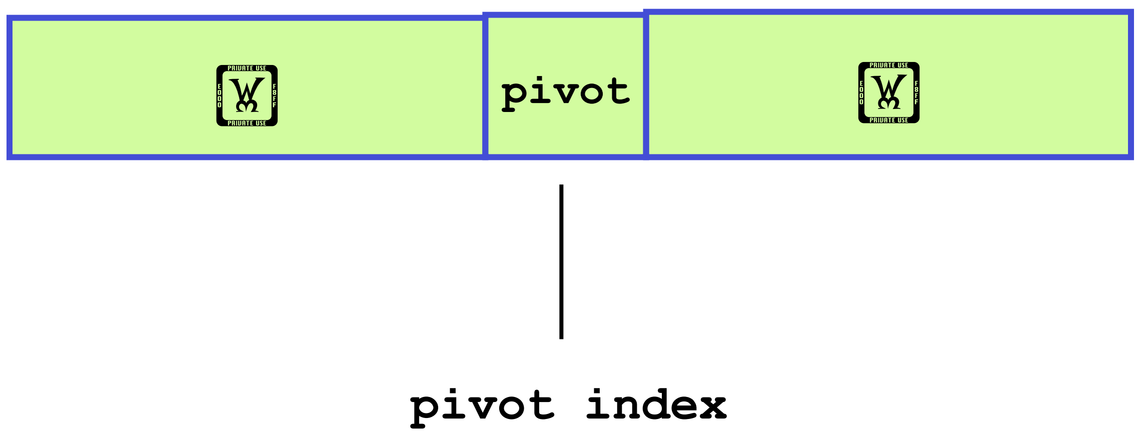
# Quicksort

- Has two phases:
  - partition phase, to break the array into two pieces
  - the sort phase, to recursively sort halves
- Most of the work goes into the partition phase
- After partitioning, the values in the left half

  are less than the values in the right half

# The Pivot

What is the invariant?



# Choosing a Pivot

- The key question: “How do we choose the pivot item?”
- Can affect performance dramatically
- Ideally, we should choose to pivot around the ***median***
- Was once thought that finding the median costs as much as sorting...But the median can be found in O(n)
- A deterministic algorithm might simply choose the *first* element as the pivot.
- A non-deterministic algorithm might choose the pivot element randomly. The worst case does not change.

# Partitioning

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 | 90 | 70 |

# Partitioning

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 | 90 | 70 |

# Partitioning

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 60 | 50 | 7 | 80 | 100 | 90 | 70 |

# Partitioning

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 60 | 50 | 7 | 80 | 100 | 90 | 70 |

# Partitioning

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 | 90 | 70 |

# Partitioning

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 | 90 | 70 |

# Partitioning

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 | 90 | 70 |

# Partitioning

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 20 | 10 | 30 | 40 | 50 | 60 | 80 | 100 | 90 | 70 |

# Partitioning

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 20 | 10 | 30 | 40 | 50 | 60 | 80 | 100 | 90 | 70 |

`pivotIndex = 4`

# Quicksort Implementation

```
int pivot = arr[pivot_loc];
swap(arr[pivot_loc], arr[0]);
int l = 1;
int r = n - 1;
while(l < r) {
      // INVARIANT: all left of l <= pivot,
      // and all right of r > pivot
      while(l < r && arr[l] <= pivot) l++;
      while(r > l && arr[r] > pivot) r--;
      if(l < r) {
            swap(arr[r], arr[l]);
            l++;
            r--;
      }
}
if (arr[l] <= pivot) swap(arr[0], arr[l]);
else swap(arr[0], arr[l - 1]);
```

# Rough Analysis

- If we divide the list in about half each time, we partition O(log n) times
- Finding the pivot index requires O(n) work
- So, we should expect the algorithm to take O(n log n) work if we find a good pivot

# Worst Case

- When do we get a bad split?
- If each value is larger than the pivot
- This happens if the array is already sorted!
- In this case runs in $O(n^2)$ time

# Ideas for Choosing Pivot

- Randomly choose an index
- Take the median of the first 3 elements
- Take the median of 3 random elements
- Median of random 2n+1 elements...

# Heapsort

- Worst-case and average case O(n log n)
- Uses heap data structure, pulling off max and re-heapifying
- [examples on the board]

# Radix Sort Question

What does the following list like after the first iteration of radix sort's outer loop?

```
class
leaks
every
other
refer
embed
array
```

# Radix Sort Question

What does the following list like after the first iteration of radix sort's outer loop?

| | |
|---|---|
| **class** | **embed** |
| **leaks** | **other** |
| **every** | **refer** |
| **other** | **class** |
| **refer** | **leaks** |
| **embed** | **every** |
| **array** | **array** |

# Mergesort Question

If we are using Mergesort, what will the following array look like right before the ***last*** merge?

35 57 53 26 50 15 22 21 25 14 11 2

# Mergesort Question

If we are using Mergesort, what will the following array look like right before the ***last*** merge?

35 57 53 26 50 15 22 21 25 14 11 2

15 26 35 50 53 57 2 11 15 21 22 25

# Quicksort Question

If we are using Quicksort, what will the result be if we pivot on 35?

35 57 53 26 50 15 22 21 25 14 11 2

# Quicksort Question

If we are using Quicksort, what will the result be if we pivot on 35?

35 57 53 26 50 15 22 21 25 14 11 2

25 2 11 26 14 15 22 21 35 50 53 57

# Heapsort Question

Heapify the following list, placing the maximum on top.

35 57 53 26 50 15 22 21 25 14 11 2

# Heapsort Question

Heapify the following list, placing the maximum on top.

35 57 53 26 50 15 22 21 25 14 11 2

57 50 53 26 35 15 22 21 25 14 11 2

# Heapsort Question

Beginning with the following array, what is the result of running the heapsort procedure (take max put it on the end of the heap, re-heapify) after four iterations?

57 50 53 26 35 15 22 21 25 14 11 2

# Heapsort Question

Beginning with the following array, what is the result of running the heapsort procedure (take max put it on the end of the heap, re-heapify) after four iterations?

57 50 53 26 35 15 22 21 25 14 11 2

26 25 22 21 14 15 2 11 35 50 53 57