Problem of the Day

Take as input a sequence of $2n$ real numbers. Design an $O(n \log n)$ algorithm that partitions the numbers into $n$ pairs, with the property that the partition minimizes the maximum sum of a pair.

For example, say we are given the numbers (1,3,5,9). The possible partitions are ((1,3),(5,9)), ((1,5),(3,9)), and ((1,9),(3,5)). The pair sums for these partitions are (4,14), (6,12), and (10,8). Thus the third partition has 10 as its maximum sum, which is the minimum over the three partitions.
Importance of Sorting

Why don’t CS profs ever stop talking about sorting?

1. Computers spend more time sorting than anything else, historically 25% on mainframes.

2. Sorting is the best studied problem in computer science, with a variety of different algorithms known.

3. Most of the interesting ideas we will encounter in the course can be taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds.

You should have seen most of the algorithms - we will concentrate on the analysis.
Efficiency of Sorting

Sorting is important because that once a set of items is sorted, many other problems become easy. Further, using $O(n \log n)$ sorting algorithms leads naturally to sub-quadratic algorithms for these problems.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2/4$</th>
<th>$n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>25</td>
<td>33</td>
</tr>
<tr>
<td>100</td>
<td>2,500</td>
<td>664</td>
</tr>
<tr>
<td>1,000</td>
<td>250,000</td>
<td>9,965</td>
</tr>
<tr>
<td>10,000</td>
<td>25,000,000</td>
<td>132,877</td>
</tr>
<tr>
<td>100,000</td>
<td>2,500,000,000</td>
<td>1,660,960</td>
</tr>
</tbody>
</table>

Large-scale data processing would be impossible if sorting took $\Omega(n^2)$ time.
Application of Sorting: Searching

Binary search lets you test whether an item is in a dictionary in $O(\lg n)$ time. Search preprocessing is perhaps the single most important application of sorting.
Application of Sorting: Closest pair

Given $n$ numbers, find the pair which are closest to each other. Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an $O(n)$ linear scan completes the job.
Application of Sorting: Element Uniqueness

Given a set of $n$ items, are they all unique or are there any duplicates?
Sort them and do a linear scan to check all adjacent pairs. This is a special case of closest pair above.
Application of Sorting: Mode

Given a set of $n$ items, which element occurs the largest number of times? More generally, compute the frequency distribution.
Sort them and do a linear scan to measure the length of all adjacent runs.
The number of instances of $k$ in a sorted array can be found in $O(\log n)$ time by using binary search to look for the positions of both $k - \epsilon$ and $k + \epsilon$. 
Application of Sorting: Median and Selection

What is the $k$th largest item in the set?

Once the keys are placed in sorted order in an array, the $k$th largest can be found in constant time by simply looking in the $k$th position of the array.

There is a linear time algorithm for this problem, but the idea comes from partial sorting.
Pragmatics of Sorting: Equal Elements

Elements with equal key values will all bunch together in any total order, but sometimes the relative order among these keys matters.

Sorting algorithms that always leave equal items in the same relative order as in the original permutation are called *stable*. Unfortunately very few fast algorithms are stable, but Stability can be achieved by adding the initial position as a secondary key.
Selection Sort

Selection sort scans through the entire array, repeatedly finding the smallest remaining element.

For $i = 1$ to $n$
A: Find the smallest of the first $n - i + 1$ items.
B: Pull it out of the array and put it first.

Selection sort takes $O(n(T(A) + T(B)))$ time.
The Data Structure Matters

Using arrays or unsorted linked lists as the data structure, operation $A$ takes $O(n)$ time and operation $B$ takes $O(1)$, for an $O(n^2)$ selection sort.

Using balanced search trees or heaps, both of these operations can be done within $O(\lg n)$ time, for an $O(n \log n)$ selection sort, balancing the work and achieving a better tradeoff.

Key question: “Can we use a different data structure?”
The heap property

- A node has the **heap property** if the value in the node is as large as or larger than the values in its children.

![Binary trees with heap property examples]

- All leaf nodes automatically have the heap property.
- A binary tree is a **heap** if *all* nodes in it have the heap property.
Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child. This is sometimes called *sifting up*. Notice that the child may have *lost* the heap property.
Constructing a heap (Step 1)

- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:

```
Add a new node here
```

```
Add a new node here
```

```
Add a new node here
```

```
Add a new node here
```
Constructing a heap (Step II)

• Each time we add a node, we may destroy the heap property of its parent node
• To fix this, we sift up
• But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of its parent node
• We repeat the sifting up process, moving up in the tree, until either
  – We reach nodes whose values don’t need to be swapped (because the parent is still larger than both children), or
  – We reach the root
Constructing a heap III

1. Initial tree with nodes 8, 10, 12.
2. Swapping 10 and 8.
3. Swapping 12 and 5.
4. Swapping 12 and 5.
Other children are not affected

- The node containing 8 is not affected because its parent gets larger, not smaller.
- The node containing 5 is not affected because its parent gets larger, not smaller.
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally.
A sample heap

• Here’s a sample binary tree after it has been heapified

• Notice that heapified does *not* mean sorted
• Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way
Removing the root (Step I)

- Notice that the largest number is now in the root
- Suppose we *discard* the root:

```
    11
   / \    /
  22   17 14
 / \    /  /  /
19 18 21 3  9
```

- Remove the rightmost leaf at the deepest level and use it for the new root
The reHeap method (Step II)

- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property

- We can `siftUp()` the root
- After doing this, one and only one of its children may have lost the heap property
The reHeap method (Step II)

- Now the left child of the root (still the number 11) lacks the heap property

- We can siftUp() this node

- After doing this, one and only one of its children may have lost the heap property
The reHeap method (Step II)

- Now the right child of the left child of the root (still the number 11) lacks the heap property:

- We can siftUp() this node
- After doing this, one and only one of its children may have lost the heap property — but it doesn’t, because it’s a leaf
The **reHeap** method (Step II)

- Our tree is once again a heap, because every node in it has the heap property

```
22
22
19
18
11
14
14
21
3
9
17
15
```

- Once again, the largest (or a largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest
• What do heaps have to do with sorting an array?
• Here’s the neat part:
  – Because the binary tree is balanced and left justified, it can be represented as an array
  – All our operations on binary trees can be represented as operations on arrays
  – To sort:
    heapify the array;
    while the array isn’t empty {
      remove and replace the root;
      reheap the new root node;
    }
Mapping into an array

- The left child of index \( i \) is at index \( 2i+1 \)
- The right child of index \( i \) is at index \( 2i+2 \)

Example: the children of node 3 (19) are 7 (18) and 8 (14)
Removing and replacing the root

- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...

...And pretend that the last element in the array no longer exists—that is, the “last index” is 11 (9)
Reheap and repeat

- Reheap the root node (index 0, containing 11)...  
  0 1 2 3 4 5 6 7 8 9 10 11 12  
  11 22 17 19 22 14 15 18 14 21 3 9 25  
  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  
  0 1 2 3 4 5 6 7 8 9 10 11 12  
  22 22 17 19 21 14 15 18 14 11 3 9 25  
  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  
  0 1 2 3 4 5 6 7 8 9 10 11 12  
  9 22 17 19 22 14 15 18 14 21 3 22 25  

- ...And again, remove and replace the root node  
- Remember, though, that the “last” array index is changed  
- Repeat until the last becomes first, and the array is sorted!
Analysis

• Here’s how the algorithm starts:
  
  heapify the array;

• Heapifying the array: we add each of $n$ nodes
  
  – Each node has to be sifted up, possibly as far as the root
    
    • Since the binary tree is perfectly balanced, sifting up a single node takes $O(\log n)$ time
  
  – Since we do this $n$ times, heapifying takes $n \times O(\log n)$ time, that is, $O(n \log n)$ time
Analysis

• Here’s the rest of the algorithm:

  while the array isn’t empty {
    remove and replace the root;
    reheap the new root node;
  }

• We do the while loop $n$ times (actually, $n-1$ times), because we remove one of the $n$ nodes each time

• Removing and replacing the root takes $O(1)$ time

• Therefore, the total time is $n$ times however long it takes the reheap method
Analysis

• To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
• The binary tree is perfectly balanced
• Therefore, this path is $O(\log n)$ long
  – And we only do $O(1)$ operations at each node
  – Therefore, reheaping takes $O(\log n)$ times
• Since we reheap inside a while loop that we do $n$ times, the total time for the while loop is $n*O(\log n)$, or $O(n \log n)$
Robert Floyd’s Improvement

Robert Floyd found a better way to build a heap, using a *merge* procedure called *heapify*.

Given two heaps and a fresh element, they can be merged into one by making the new one the root and bubbling down.

**Build-heap(A)**

```plaintext
n = |A|
For i = n/2 to 1 do
  Heapify(A, i)
```

**Heapify(A, i)**

```plaintext
le = left(i)
ri = right(i)
if (le<=heapsize) and (A[le]>A[i])
  largest = le else largest = i
if (ri<=heapsize) and (A[ri]>A[largest])
  largest = ri
if (largest != i) {
  exchange(A[i], A[largest])
  Heapify(A, largest) }
```
Analysis of Build-heap

In fact, Build-heap performs better than $O(n \log n)$, because most of the heaps we merge are extremely small. It follows exactly the same analysis as with dynamic arrays. In a full binary tree on $n$ nodes, there are at most $n/2^{h+1}$ nodes of height $h$, so the cost of building a heap is as follows.

<table>
<thead>
<tr>
<th>Height of heaps</th>
<th># of heaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$n/2$</td>
</tr>
<tr>
<td>2</td>
<td>$n/4$</td>
</tr>
<tr>
<td>3</td>
<td>$n/8$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\lg n$</td>
<td>1</td>
</tr>
</tbody>
</table>

Build-heap($A$)

$n = |A|$

For $i = n/2$ to 1 do

Heapify($A, i$)
Proof of Convergence (*)

The identify for the sum of a geometric series is

\[ \sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \]

If we take the derivative of both sides, . . .

\[ \sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1 - x)^2} \]

Multiplying both sides of the equation by \( x \) gives:

\[ \sum_{k=0}^{\infty} k x^k = \frac{x}{(1 - x)^2} \]

Substituting \( x = 1/2 \) gives a sum of 2, so Build-heap uses at most \( 2n \) comparisons and thus linear time.
Heapsort

Heapify can be used to construct a heap, using the observation that an isolated element forms a heap of size 1.

Heapsort(A)
   Build-heap(A)
   for \( i = n \) to 1 do
      swap(A[1],A[i])
      \( n = n - 1 \)
      Heapify(A,1)

Exchanging the maximum element with the last element and calling heapify repeatedly gives an \( O(n \log n) \) sorting algorithm. Why is it not \( O(n) \)?
Priority Queues

Priority queues are data structures which provide extra flexibility over sorting. This is important because jobs often enter a system at arbitrary intervals. It is more cost-effective to insert a new job into a priority queue than to re-sort everything on each new arrival.
Priority Queue Operations

The basic priority queue supports three primary operations:

- **Insert**($Q,x$): Given an item $x$ with key $k$, insert it into the priority queue $Q$.
- **Find-Minimum**($Q$) or **Find-Maximum**($Q$): Return a pointer to the item whose key value is smaller (larger) than any other key in the priority queue $Q$.
- **Delete-Minimum**($Q$) or **Delete-Maximum**($Q$) – Remove the item from the priority queue $Q$ whose key is minimum (maximum).

Each of these operations can be easily supported using heaps or balanced binary trees in $O(\log n)$. 

Applications of Priority Queues: Greedy Algorithms

In greedy algorithms, we always pick the next thing which locally maximizes our score. By placing all the things in a priority queue and pulling them off in order, we can improve performance over linear search or sorting, particularly if the weights change.

War Story: sequential strips in triangulations
Merge Sort

• Apply divide-and-conquer to sorting problem
• Problem: Given $n$ elements, sort elements into non-decreasing order
• Divide-and-Conquer:
  – If $n=1$ terminate (every one-element list is already sorted)
  – If $n>1$, partition elements into two or more sub-collections; sort each; combine into a single sorted list
• How do we partition?
Partitioning - Choice 1

• First n-1 elements into set A, last element set B
• Sort A using this partitioning scheme recursively
  – B already sorted
• Combine A and B using method Insert() (= insertion into sorted array)
• Leads to recursive version of InsertionSort()
  – Number of comparisons: $O(n^2)$
    • Best case = n-1
    • Worst case = 

\[ c \sum_{i=2}^{n} i = \frac{n(n-1)}{2} \]
Partitioning - Choice 2

• Put element with largest key in B, remaining elements in A
• Sort A recursively
• To combine sorted A and B, append B to sorted A
  – Use Max() to find largest element → recursive SelectionSort()
  – Use bubbling process to find and move largest element to right-most position → recursive BubbleSort()
• All $O(n^2)$
Partitioning - Choice 3

• Let’s try to achieve balanced partitioning
• A gets $n/2$ elements, B gets rest half
• Sort A and B recursively
• Combine sorted A and B using a process called *merge*, which combines two sorted lists into one
  – How? We will see soon
Example

- Partition into lists of size $n/2$

```
[10, 4, 6, 3, 8, 2, 5, 7]
```

```
[10, 4, 6, 3]  [8, 2, 5, 7]
```

```
[10, 4]  [6, 3]

[4] [10]

[3][6]

[8, 2]

[2][8]

[5][7]

[5, 7]  [8, 2]
```
Example Cont’d

• Merge

\[
\begin{array}{c}
[2, 3, 4, 5, 6, 7, 8, 10] \\
[3, 4, 6, 10] & [2, 5, 7, 8] \\
\end{array}
\]
Static Method mergeSort()

Public static void mergeSort(Comparable []a, int left, int right)
{
    // sort a[left:right]
    if (left < right)
    {
        // at least two elements
        int mid = (left+right)/2;  //midpoint
        mergeSort(a, left, mid);  //mergeSort(a, left, mid);
        mergeSort(a, mid + 1, right);  //mergeSort(a, mid + 1, right);
        merge(a, b, left, mid, right);  //merge from a to b
        copy(b, a, left, right);  //copy result back to a
    }
}
Evaluation

• Recurrence equation:
• Assume n is a power of 2

\[
T(n) = \begin{cases} 
  c_1 & \text{if } n=1 \\
  2T(n/2) + c_2n & \text{if } n>1, \ n=2^k 
\end{cases}
\]
Solution

By Substitution:

\[ T(n) = 2T(n/2) + c_2n \]
\[ T(n/2) = 2T(n/4) + c_2n/2 \]

\[ T(n) = 4T(n/4) + 2c_2n \]
\[ T(n) = 8T(n/8) + 3c_2n \]

\[ T(n) = 2^iT(n/2^i) + ic_2n \]

Assuming \( n = 2^k \), expansion halts when we get \( T(1) \) on right side; this happens when \( i=k \)  
\[ T(n) = 2^kT(1) + kc_2n \]

Since \( 2^k=n \), we know \( k=\log n \); since \( T(1) = c_1 \), we get  
\[ T(n) = c_1n + c_2n\log n \]

thus an upper bound for \( T_{\text{mergeSort}}(n) \) is \( O(n\log n) \)
Quicksort Algorithm

Given an array of \( n \) elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot*.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results
Example

We are given array of n integers to sort:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>20</td>
<td>10</td>
<td>80</td>
<td>60</td>
<td>50</td>
<td>7</td>
<td>30</td>
<td>100</td>
</tr>
</tbody>
</table>
Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements $\geq$ pivot
2. Another sub-array that contains elements $<$ pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.
pivot_index = 0

[40, 20, 10, 80, 60, 50, 7, 30, 100]

[0] [1] [2] [3] [4] [5] [6] [7] [8]

too_big_index too_small_index
1. While data[too_big_index] <= data[pivot]
    ++too_big_index
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
1. While \(\text{data}[\text{too_big_index}] \leq \text{data}[	ext{pivot}]\)
   \[++\text{too_big_index}\]
2. While \(\text{data}[\text{too_small_index}] > \text{data}[	ext{pivot}]\)
   \[--\text{too_small_index}\]
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]
1. While \text{data[too_big_index]} \leq \text{data[pivot]}
   \quad ++\text{too_big_index}
2. While \text{data[too_small_index]} > \text{data[pivot]}
   \quad --\text{too_small_index}
3. If \text{too_big_index} < \text{too_small_index}
   \quad swap \text{data[too_big_index]} and \text{data[too_small_index]}
4. If \text{too_small_index} > \text{too_big_index}, go to 1.
5. Swap \text{data[too_small_index]} and \text{data[pivot_index]}

pivot\_index = 4

\begin{array}{cccccccccc}
7 & 20 & 10 & 30 & 40 & 50 & 60 & 80 & 100 \\
\end{array}

[0] [1] [2] [3] [4] [5] [6] [7] [8]

\text{too_big_index} \quad \text{too_small_index}
Partition Result

<table>
<thead>
<tr>
<th>7</th>
<th>20</th>
<th>10</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
<td>[8]</td>
</tr>
</tbody>
</table>

<= data[pivot]  > data[pivot]

Recursive calls on two sides to get a sorted array.
Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• What is best case running time?
Quicksort Analysis

• Assume that keys are random, uniformly distributed.

• What is best case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays of size $n/2$
    2. Quicksort each sub-array
Quicksort Analysis

• Assume that keys are random, uniformly distributed.

• What is best case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  – Depth of recursion tree?
Quicksort Analysis

• Assume that keys are random, uniformly distributed.

• What is best case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  – Depth of recursion tree? $O(\log_2 n)$
Quicksort Analysis

• Assume that keys are random, uniformly distributed.

• What is best case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays of size \( n/2 \)
    2. Quicksort each sub-array
  – Depth of recursion tree? \( O(\log_2 n) \)
  – Number of accesses in partition?
Quicksort Analysis

• Assume that keys are random, uniformly distributed.

• What is best case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  – Depth of recursion tree? $O(\log_2 n)$
  – Number of accesses in partition? $O(n)$
Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: $O(n \log_2 n)$
• Worst case running time?
Quicksort Analysis

• Assume that keys are random, uniformly distributed.

• Best case running time: \( O(n \log_2 n) \)

• Worst case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays:
      • one sub-array of size 0
      • the other sub-array of size n-1
    2. Quicksort each sub-array
  – Depth of recursion tree?
Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: $O(n \log_2 n)$
• Worst case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays:
       • one sub-array of size 0
       • the other sub-array of size n-1
    2. Quicksort each sub-array
  – Depth of recursion tree? $O(n)$
Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: $O(n \log_2 n)$
• Worst case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays:
      • one sub-array of size 0
      • the other sub-array of size $n-1$
    2. Quicksort each sub-array
  – Depth of recursion tree? $O(n)$
  – Number of accesses per partition?
Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: $O(n \log_2 n)$
• Worst case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays:
       • one sub-array of size 0
       • the other sub-array of size n-1
    2. Quicksort each sub-array
  – Depth of recursion tree? $O(n)$
  – Number of accesses per partition? $O(n)$
Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: $O(n \log_2 n)$
• Worst case running time: $O(n^2)$!!!
QuickSort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: $O(n \log_2 n)$
• Worst case running time: $O(n^2)$!!!
• What can we do to avoid worst case?
  – Randomly pick a pivot
Quicksort Analysis

- Bad divide: \( T(n) = T(1) + T(n-1) \) \( \text{-- } O(n^2) \)
- Good divide: \( T(n) = T(n/2) + T(n/2) \) \( \text{-- } O(n \log_2 n) \)
- Random divide: Suppose on average one bad divide followed by one good divide.
  \[ T(n) = T(1) + T(n-1) = T(1) + 2T((n-1)/2) \]
  \[ T(n) = c + 2T((n-1)/2) \text{ is still } O(n \log_2 n) \]
Randomized Guarantees

• Randomization is a very important and useful idea. By either picking a random pivot or scrambling the permutation before sorting it, we can say:
  – “With high probability, randomized quicksort runs in $O(n \lg n)$ time.”

• Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.

• The worst-case is still there, but we almost certainly won’t see it.
Improved Pivot Selection

Pick median value of three elements from data array:
   data[0], data[n/2], and data[n-1].

Use this median value as pivot.

For large arrays, use the median of three medians from
   {data[0], data[1], data[2]}, {data[n/2-1], data[n/2], data[n/2+1]}, and {data[n-3], data[n-2], data[n-1]}. 
Improving Performance of Quicksort

• Improved selection of pivot.
• For sub-arrays of size 10 or less, apply brute force search, such as insert-sort.
  – Sub-array of size 1: trivial
  – Sub-array of size 2:
    • if(data[first] > data[second]) swap them
  – Sub-array of size 3: left as an exercise.
Improving Performance of Quicksort

• Improved selection of pivot.
• For sub-arrays of size 10 or less, apply brute force search, such as insert-sort.
• Test if the sub-array is already sorted before recursive calls.
• Don’t pick out the pivot (avoiding the last swap).