Our first sort: Selection Sort

• General Idea:

What is the invariant of this sort?
Selection Sort

• Let A be an array of n ints, and we wish to sort these keys in non-decreasing order.
• Algorithm:
  for i = 0 to n-2 do
    find j, i ≤ j ≤ n-1, such that A[j] ≤ A[k], ∀k, i ≤ k ≤ n-1.
    swap A[j] with A[i]
• This algorithm works in place, meaning it uses its own storage to perform the sort.
### Selection Sort Example

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>66</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>99</td>
<td>55</td>
<td>66</td>
<td>88</td>
<td>44</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>55</td>
<td>66</td>
<td>88</td>
<td>44</td>
<td>77</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>66</td>
<td>88</td>
<td>55</td>
<td>77</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>88</td>
<td>66</td>
<td>77</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>88</td>
<td>77</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>88</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>88</td>
<td>99</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Selection Sort

```java
public static void sort(int[] data, int n)
{
    int i,j,minLocation;
    for (i=0; i<=n-2; i++) {
        minLocation = i;
        for (j=i+1; j<=n-1; j++)
            if (data[j] < data[minLocation])
                minLocation = j;
        swap(data, minLocation, i);
    }
}
```
Run time analysis

- **Worst Case:**
  
  Search for 1\textsuperscript{st} min: \( n-1 \) comparisons
  
  Search for 2\textsuperscript{nd} min: \( n-2 \) comparisons
  
  ... 
  
  Search for 2\textsuperscript{nd}-to-last min: 1 comparison
  
  **Total comparisons:**
  
  \[(n-1) + (n-2) + ... + 1 = O(n^2)\]

- **Average Case and Best Case:**
  
  \( O(n^2) \) also! (Why?)
public static void sort(int[] data, int n)
{
    int i, j;
    for (i=0; i<=n-2; i++)
        for (j=i+1; j<=n-1; j++)
            if (data[j] < data[i])
                swap(data, i, j);
}

Is this any better?
Insertion Sort

- General Idea:

What is the invariant of this sort?
Insertion Sort

• Let A be an array of n ints, and we wish to sort these keys in non-decreasing order.

• Algorithm:
  for i = 1 to n-1 do
    item = A[i]
    let k be the smallest j above, or k=i if no shifts
    A[k] = item

• This algorithm also works in place.
### Insertion Sort Example

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>66</td>
<td>99</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>66</td>
<td>99</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>55</td>
<td>66</td>
<td>99</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>99</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>88</td>
<td>99</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>88</td>
<td>99</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>88</td>
<td>99</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>88</td>
<td>99</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
public static void sort(int[] data, int n) {
    int i, j, item;
    for (i = 1; i <= n - 1; i++) {
        item = data[i]; j = i;
        while (j > 0 && data[j - 1] > item) {
            data[j] = data[j - 1];
            j--;
        }
        data[j] = item;
    }
}
Is insertion sort any better?

• Worst Case:  
  Insert 2\textsuperscript{nd} item: 1 shift  
  Insert 3\textsuperscript{rd} item: 2 shifts  
  ...  
  Insert last item: n-1 shifts  
  Total shifts: 1+2+...+(n-1) = O(n^2)

• Average Case: O(n^2) also

• Best Case: O(n)  
  This occurs if the array is already sorted.  
  No shifts needed!
Bubble Sort (Exchange Sort)

What is the invariant of this sort?
# Bubble Sort Example

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>22</td>
<td>88</td>
<td>33</td>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>33</td>
<td>88</td>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>77</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>66</td>
<td>55</td>
<td>99</td>
<td>77</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>99</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>88</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>88</td>
<td>99</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
public static void sort(int[] data, int n) {
    int i, j;
    for (i=0; i <= n-2; i++)
        for (j=n-1; j > i; j--)
            if (data[j] < data[j-1])
                swap(data, j, j-1);
}
Bubble Sort Special cases

- Array is already sorted
- Array is partially sorted
- First element is out of place
- Last element is out of place
Quadratic Sorts

- Quadratic sorts have a worst-case order of complexity of $O(n^2)$
- Selection sort always performs poorly, even on a sequence of sorted keys!
- Insertion sort performs much better if the keys are sorted or nearly sorted.
- Bubble sort performs better on special cases.
Divide-and-Conquer Sorts

• Divide the elements to be sorted into two groups of approximately equal size.
• Sort each of these smaller groups.
• Combine the two sorted groups into one large sorted list.

Use recursion to sort the smaller groups.
Merge Sort

• Split the array into two “halves”.
• Sort each of the halves recursively using merge sort.
• Merge the two sorted halves to obtain a completely sorted array.

Example:

66   44   99   55   11   88   22   77   33

sort the halves recursively...

44   55   66   99   11   22   33   77   88
Then merge the two sorted halves into a new array:

```
44  55  66  99  11  22  33  77  88

___  ___  ___  ___  ___  ___  ___  ___  ___  ___  ___

11  ___  ___  ___  ___  ___  ___  ___  ___  ___  ___

11  22  ___  ___  ___  ___  ___  ___  ___  ___  ___

11  22  ___  ___  ___  ___  ___  ___  ___  ___  ___
```
<table>
<thead>
<tr>
<th>44</th>
<th>55</th>
<th>66</th>
<th>99</th>
<th>11</th>
<th>22</th>
<th>33</th>
<th>77</th>
<th>88</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>55</td>
<td>66</td>
<td>99</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>77</td>
<td>88</td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>55</td>
<td>66</td>
<td>99</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>77</td>
<td>88</td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Merge Sort (cont’d)

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>55</td>
<td>66</td>
<td><strong>99</strong></td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>77</td>
<td>88</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>55</td>
<td>66</td>
<td><strong>99</strong></td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>77</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>55</td>
<td>66</td>
<td><strong>99</strong></td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>77</td>
<td>88</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>88</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>55</td>
<td>66</td>
<td><strong>99</strong></td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>77</td>
<td>88</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Merge Sort (cont’d)

Once one of the halves has been merged into the new array, copy the remaining element(s) of the other half into the new array:

```
44  55  66  99  11  22  33  77  88
11  22  33  44  55  66  77  88  99
```

Another Example:

```
43  42  21  44  45  26  27  25  23  24
```

sort the halves recursively...

```
21  42  43  44  45  23  24  25  26  27
```

then merge the two sorted halves ....

```
21  23  24  25  26  27  42  43  44  45
```

• Merge sort does not sort in place.
public static void sort(int[] data, int first, int n) {
    int n1, n2;
    if (n > 1) {
        n1 = n / 2;  n2 = n - n1;
        sort(data, first, n1);
        sort(data, first+n1, n2);
        merge(data, first, n1, n1, n2);
    }
    // what is the stopping case?
}
Run-Time Analysis

- Let $T(N) =$ number of operations to sort $N$ elements using merge sort.
- How many operations does it take to sort half of the array? $T(N/2)$
- How many operations does it take to merge the two halves? $2N$
- $T(N) = 2^*T(N/2) + 2N$
Run-Time Analysis (cont’d)

• What is the stopping case?
  \[ T(1) = 0 \]
• \[ T(N) = O(N \log N) \]
• Note: \[ O(N \log N) < O(N^2) \]
Another way to look at it

\[ \log_2 N \]
Quick Sort (Partition Sort)

- Choose a pivot element of the array.
- Partition the array so that
  - the pivot element is in the correct position for the sorted array
  - all the elements to the left of the pivot are less than or equal to the pivot
  - all the elements to the right of the pivot are greater than the pivot
- Sort the subarray to the left of the pivot and the subarray to the right of the pivot recursively using quick sort
Partitioning the array

Arbitrarily choose the first element as the pivot.

66  44  99  55  11  88  22  77  33

Search from the left end for the first element that is greater than the pivot.

66  44  99  55  11  88  22  77  33

Search from the right end for the first element that is less than (or equal to) the pivot.

66  44  99  55  11  88  22  77  33

Now swap these two elements.

66  44  33  55  11  88  22  77  99
Partitioning the array (cont’d)

From the two elements just swapped, search again from the left and right ends for the next elements that are greater than and less than the pivot, respectively.

Swap these as well.

Continue this process until our searches from each end meet.
Partitioning the array (cont’d)

At this point, the array has been partitioned into two subarrays, one with elements less than (or equal to) the pivot, and the other with elements greater than the pivot.

```
66  44  33  55  11  22  88  77  99
```

Finally, swap the pivot with the last element in the first subarray section (the elements that are less than the pivot).

```
22  44  33  55  11  66  88  77  99
```

Now sort the two subarrays on either side of the pivot using quick sort recursively.
Quick Sort

```java
public static void sort(int[] data, int first, int n) {
    int n1, n2, pivotIndex;
    if (n > 1) {
        pivotIndex = partition(data, first, n);
        n1 = pivotIndex - first;
        n2 = n - n1 - 1;
        sort(data, first, n1);
        sort(data, pivotIndex+1, n2);
    }
}
```
Run-Time Analysis

- Assume the pivot ends up in the center position of the array.
- Then, quick sort runs in $O(N \log N)$ time just like merge sort.
- However, what if the pivot doesn’t end up in the center during partitioning?
  Example: Pivot is smallest element. Then we get two subarrays, one of size 0, and the other of size $n-1$ (instead of $n/2$ for each).
- Then, quick sort can perform as bad as $O(n^2)$.  
  *When does this occur?*
Some Improvements to Quick Sort

• Choose three values from the array, and use the middle element of the three as the pivot.

Of 11, 33, 66, use 33 as the pivot.

• As quick sort is called recursively, if a subarray is of “small size”, use insertion sort instead of quick sort to complete the sorting to reduce the number of recursive calls.
void quickSort(int[] data, int first, int last) {
    int left, right, pivot;
    if (first >= last)  return;
    left = first; right = last;
    pivot = data[(first + last)/2];
    do{
        while (data[left] < pivot)  left++;
        while (data[right] > pivot)  right--;
        if (left <= right)
            swap(data, left++, right--);
    } while (left <= right);
    quickSort(data, first, right);
    quickSort(data, left, last);
}
Heap Sort

- We can use a heap to sort data.
- Convert an array to a heap.
- Remove the root from the heap and store it in its proper position in the same array.

Example:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>61</td>
<td>83</td>
<td>53</td>
<td>39</td>
<td>72</td>
<td>16</td>
<td>24</td>
<td>48</td>
</tr>
</tbody>
</table>

**Heap**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>83</td>
<td>61</td>
<td>72</td>
<td>53</td>
<td>39</td>
<td>48</td>
<td>16</td>
<td>24</td>
<td>95</td>
</tr>
</tbody>
</table>

**SORTED ARRAY**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>61</td>
<td>48</td>
<td>53</td>
<td>39</td>
<td>24</td>
<td>16</td>
<td>83</td>
<td>95</td>
</tr>
</tbody>
</table>
Sorting using the heap

HEAP | SORTED ARRAY

remove root from heap

HEAP | SORTED ARRAY

reheapify!
Heap Sort

```java
public static void sort(int[] data, int n) {
    int lastHeapPosition;
    makeHeap(data, n);
    lastHeapPosition = n - 1;
    while (lastHeapPosition > 0) {
        swap(data, 0, lastHeapPosition);
        reheapify(data, lastHeapPosition);
        lastHeapPosition--;
    }
}
```

- Convert array to a heap.
- Move the root and fix the remaining heap.
Making the heap
```java
public static void makeHeap
        (int[] data, int n) {
    int i;
    int next;  // next element for heap
    for (i = 1; i <= n - 1; i++) {
        next = i;
        //insert next into heap from 0..i-1
        while (next != 0 &&
               data[next] > data[parent(next)]) {
            swap(data, next, parent(next));
            next = parent(next);
        }
    }
}
```

Assumes a parent() method to return index of parent of next.
Make Heap Example
(heap in **blue**)

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>44</td>
<td>66</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>55</td>
<td>66</td>
<td>44</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>55</td>
<td>66</td>
<td>44</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>55</td>
<td>88</td>
<td>44</td>
<td>11</td>
<td>66</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>55</td>
<td>88</td>
<td>44</td>
<td>11</td>
<td>66</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>77</td>
<td>88</td>
<td>55</td>
<td>11</td>
<td>66</td>
<td>22</td>
<td>44</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>77</td>
<td>88</td>
<td>55</td>
<td>11</td>
<td>66</td>
<td>22</td>
<td>44</td>
<td>33</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
private static void reheapify
   (int[] data, int heapsize) {
   int position = 0; int childPos;
   while (position*2 + 1 < heapsize) {
      childPos = position*2 + 1;
      if (childPos < heapsize-1 &&
         data[childPos+1] > data[childPos])
         childPos++;
   if (data[position]<data[childPos]) {
      swap(data, position, childPos);
      position = childPos;
   }
   else return;
   }
}
Make Heap - Another View

66  44  99  55  11  88  22  77  33
Make Heap - Another View

66  44  99  77  11  88  22  55  33
Make Heap - Another View

66  77  99  55  11  88  22  44  33
Make Heap - Another View

99  77  88  55  11  66  22  44  33

# Make Heap Example

*(organized heap in blue)*

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>77</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>44</td>
<td>99</td>
<td>77</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>55</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>44</td>
<td>99</td>
<td>77</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>55</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>44</td>
<td>99</td>
<td>55</td>
<td>11</td>
<td>88</td>
<td>22</td>
<td>44</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>77</td>
<td>88</td>
<td>55</td>
<td>11</td>
<td>66</td>
<td>22</td>
<td>44</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>
public static void efficientMakeHeap
         (int[] data, int n) {
    int i, child, position = 0;
    for (i = (n - 2) / 2; i >= 0; i--) {
        position = i;
        while (position * 2 + 1 < n) {
            child = position * 2 + 1;
            if (child < n - 1 &&
                data[child + 1] > data[child])
                child++;
            if (data[child] > data[position]) {
                swap(data, child, position);
                position = child;
            } else break;
        }
    } // for
} // efficientMakeHeap
Heap Sort Run-Time Analysis

- Number of operations to convert an array into a heap: $O(n)$
- Number of operations to remove each element from a heap: $O(\log n)$
- Number of operations to remove $n$ elements from the heap: $O(n \times \log n)$
- Total number of operations:
  \[ O(n) + O(n \times \log n) = O(n \times \log n) \]
Counting Sort (Linear Time)

- Input array has n integers in the range \([0, k-1]\)
- The idea is that for each number \(x\), how many elements are less than \(x\).
- For example, if there are 6 elements less than \(x\), then \(x\) belongs in the 7\(^{th}\) position.
- If there are several elements with the same value we place them one after the other.
- We use following arrays for the sort:
  - data \([0..n-1]\) – The input array
  - result\([0..n-1]\) – The output array
  - count\([0..k-1]\) & start\([0..k-1]\) – Temp arrays
## Counting Sort Example

<table>
<thead>
<tr>
<th>data</th>
<th>count</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 5 3 4 3 1 3 4 3 1</td>
<td>0 3 0 5 2 1</td>
<td>1 1 1 3 3 3 3 3 3 4 4 5</td>
</tr>
</tbody>
</table>

### Data

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3 1</td>
</tr>
</tbody>
</table>

### Count

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### Start

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

### Result

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4 5</td>
</tr>
</tbody>
</table>
Counting Sort Algorithm

for i = 0 to k-1 do  \( O(k) \)
    count[i] = 0

for i = 0 to n-1 do  \( O(n) \)
    count[data[i]]++

start[0]=0

for i = 1 to k-1 do  \( O(k) \)
    start[i] = count[i-1] + start[i-1]

for i = 0 to n-1 do  \( O(n) \)
    pos = start[data[i]]
    result[pos] = data[i]
    start[data[i]]++
Counting Sort (Cont’d)

- Run-Time Analysis
  - The first and third loops $O(k)$
  - The second and last loops $O(n)$
  - Total running time $O(n+k)$
  - If $k = O(n)$, then total time will be $O(n)$

- Counting sort is **stable**

- This algorithm **does not** work **in place**.