CS 106B, Lecture 25
Sorting
Plan for Today

• Analyze several algorithms to do the same task: sorting
  – Big-Oh in the real world
• **sorting**: Rearranging the values in a collection into a specific order.
  – *can be solved in many ways:*

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bogo sort</td>
<td>shuffle and pray</td>
</tr>
<tr>
<td>bubble sort</td>
<td>swap adjacent pairs that are out of order</td>
</tr>
<tr>
<td>selection sort</td>
<td>look for the smallest element, move to front</td>
</tr>
<tr>
<td>insertion sort</td>
<td>build an increasingly large sorted front portion</td>
</tr>
<tr>
<td>merge sort</td>
<td>recursively divide the data in half and sort it</td>
</tr>
<tr>
<td>heap sort</td>
<td>place the values into a binary heap then dequeue</td>
</tr>
<tr>
<td>quick sort</td>
<td>recursively &quot;partition&quot; data based on a pivot value</td>
</tr>
<tr>
<td>bucket sort</td>
<td>cluster elements into smaller groups, sort the groups</td>
</tr>
<tr>
<td>radix sort</td>
<td>sort integers by last digit, then 2nd to last, then ...</td>
</tr>
</tbody>
</table>
• **bogo sort**: Orders a list of values by repetitively shuffling them and checking if they are sorted.
  – name comes from the word "bogus"; a.k.a. "bogus sort"

The algorithm:
  – Scan the list, seeing if it is sorted. If so, stop.
  – Else, shuffle the values in the list and repeat.

• This sorting algorithm (obviously) has terrible performance!
  – What is its runtime?
Bogo sort code

// Places the elements of v into sorted order.
void bogoSort(Vector<int>& v) {
    while (!isSorted(v)) {
        shuffle(v);       // from shuffle.h
    }
}

// Returns true if v's elements are in sorted order.
bool isSorted(Vector<int>& v) {
    for (int i = 0; i < v.size() - 1; i++) {
        if (v[i] > v[i + 1]) {
            return false;
        }
    }
    return true;
}
Bogo sort runtime

• How long should we expect bogo sort to take?
  – related to probability of shuffling into sorted order
  – assuming shuffling code is fair, probability equals
    \( \frac{1}{\text{number of permutations of } N \text{ elements}} = \frac{1}{N!} \)
  – average case performance: \( O(N \times N!) \)
  – worst case performance: \( O(\infty) \)
  – What is the best case performance?
**Selection sort example**

- **Selection sort**: Repeatedly swap smallest unplaced value to front.

<table>
<thead>
<tr>
<th>index</th>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>22</td>
<td>18</td>
<td>12</td>
<td>-4</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>2</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>

- After 1st, 2nd, and 3rd passes:

<table>
<thead>
<tr>
<th>index</th>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>18</td>
<td>12</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>2</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>12</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>18</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>12</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>18</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>
Selection sort code

// Rearranges elements of v into sorted order.
void selectionSort(Vector<int>& v) {
    for (int i = 0; i < v.size() - 1; i++) {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < v.size(); j++) {
            if (v[j] < v[min]) {
                min = j;
            }
        }
        // swap smallest value to proper place, v[i]
        if (i != min) {
            int temp = v[i];
            v[i] = v[min];
            v[min] = temp;
        }
    }
}
Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert second item into sorted sublist, shifting first item if needed
  - insert third item into sorted sublist, shifting items 1-2 as needed
  - ...
  - repeat until all values have been inserted into their proper positions
  - How people line up when they have different arrival times!

- Runtime: $O(N^2)$.
  - Generally somewhat faster than selection sort for most inputs.
Insertion sort example

- Makes $N-1$ passes over the array.
- At the end of pass $i$, the elements that occupied $A[0]...A[i]$ originally are still in those spots and in sorted order.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>15</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 1</td>
<td>2</td>
<td>15</td>
<td>8</td>
<td>1</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 2</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>1</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 3</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 4</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 5</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>15</td>
<td>17</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 6</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>pass 7</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>17</td>
</tr>
</tbody>
</table>
// Rearranges the elements of v into sorted order.
void insertionSort(Vector<int>& v) {
    for (int i = 1; i < v.size(); i++) {
        int temp = v[i];

        // slide elements right to make room for v[i]
        int j = i;
        while (j >= 1 && v[j - 1] > temp) {
            v[j] = v[j - 1];
            j--;
        }
        v[j] = temp;
    }
}
Bucket/radix sort

- **bucket sort**: arrange items into buckets or bins repeatedly
- **radix sort**: sort integers by 1s, then 10s, then 100s, ...
  - $O(N)$ when used with data in a known fixed range (!)
Announcements

• MiniBrowser is due today, Calligraphy will be released later today
  – Multiple parts, please start early (2nd and 3rd parts are harder than the 1st part)
• Final is a **week from Saturday**, at 8:30AM
  – Practice exam will be released in the next few days
• Please give us feedback! cs198.stanford.edu
• Course feedback:
  – A note on LaIR/Piazza
• **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

The algorithm:
– Divide the list into two roughly equal halves.
– Sort the left half.
– Sort the right half.
– Merge the two sorted halves into one sorted list.

– Often implemented recursively.
– An example of a "divide and conquer" algorithm.
  • Invented by John von Neumann in 1945

– Runtime: $O(N \log N)$. Somewhat faster for asc/descending input.
Merge sort example

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>22</td>
<td>18</td>
<td>12</td>
<td>-4</td>
<td>58</td>
<td>7</td>
<td>31</td>
<td>42</td>
</tr>
</tbody>
</table>

```
22 18 12 -4
  12 18 42 22
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
  12 18 22 42
```
Merging sorted halves

<table>
<thead>
<tr>
<th>Subarrays</th>
<th>Next include</th>
<th>Merged array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>14</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>i</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>14 from left</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>23 from right</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>32 from left</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>41 from right</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>58 from right</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>67 from left</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>76 from left</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>85 from right</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
</tbody>
</table>
Merge sort code

// Rearranges the elements of v into sorted order using
// the merge sort algorithm.
void mergeSort(Vector<int>& v) {
    if (v.size() >= 2) {
        // split vector into two halves
        Vector<int> left = v.subList(0, v.size() / 2);
        Vector<int> right =
            v.subList(v.size() / 2 + 1, (v.size() - 1) / 2);

        // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        v.clear();
        merge(v, left, right);
    }
}
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
void merge(Vector<int>& result,
    Vector<int>& left, Vector<int>& right) {
    int leftIndex = 0;
    int rightIndex = 0;

    for (int i = 0; i < left.size() + right.size(); i++) {
        if (rightIndex >= right.size() ||
            (leftIndex < left.size() &&
             left[leftIndex] <= right[rightIndex])) {
            result += left[leftIndex];  // take from left
            leftIndex++;
        } else {
            result += right[rightIndex];  // take from right
            rightIndex++;
        }
    }
}
Runtime intuition

• Merge sort performs $O(N)$ operations on each level. (width)
  – Each level splits the data in 2, so there are $\log_2 N$ levels. (height)
  – Product of these = $N \times \log_2 N = O(N \log N)$. (area)
  – Example: $N = 32$. Performs $\sim \log_2 32 = 5$ levels of $N$ operations each:

```
32
16
8
4
2
1

width = N
height = \log_2 N
```
Quick sort

• **quick sort**: Orders a list of values by partitioning the list around one element called a *pivot*, then sorting each partition.
  – invented by British computer scientist C.A.R. Hoare in 1960

• Quick sort is another divide and conquer algorithm:
  – Choose one element in the list to be the pivot.
  – *Divide* the elements so that all elements less than the pivot are to its left and all greater (or equal) are to its right.
  – *Conquer* by applying quick sort (recursively) to both partitions.

• Runtime: \(O(N \log N)\) average, but \(O(N^2)\) worst case.
  – Generally somewhat faster than merge sort.
Choosing a "pivot"

• The algorithm will work correctly no matter which element you choose as the pivot.
  – A simple implementation can just use the first element.

• But for efficiency, it is better if the pivot divides up the array into roughly equal partitions.
  – What kind of value would be a good pivot? A bad one?

| index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| value | 8  | 18 | 12 | -4 | 27 | 30 | 36 | 50 | 7  | 68 | 91 | 56 | 2  | 85 | 42 | 98 | 25 |
Partitioning an array

• Swap the pivot to the last array slot, temporarily.
• Repeat until done partitioning (until $i,j$ meet):
  – Starting from $i = 0$, find an element $a[i] \geq$ pivot.
  – Starting from $j = N-1$, find an element $a[j] \leq$ pivot.
  – These elements are out of order, so swap $a[i]$ and $a[j]$.
• Swap the pivot back to index $i$ to place it between the partitions.

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

8 i → j 6
2  i → j 8
5  i → 9

2 1 4 5 0 3 6 8 7 9
**Quick sort example**

<table>
<thead>
<tr>
<th>index</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>65</td>
<td>23</td>
<td>81</td>
<td>43</td>
<td>92</td>
<td>39</td>
<td>57</td>
<td>16</td>
<td>75</td>
<td>32</td>
</tr>
</tbody>
</table>

- **choose pivot=65**
- **swap pivot (65) to end**
- **swap 81, 16**
- **swap 57, 92**
- **swap pivot back in**

**recursively quicksort each half**

<table>
<thead>
<tr>
<th>32</th>
<th>23</th>
<th>16</th>
<th>43</th>
<th>57</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>23</td>
<td>16</td>
<td>43</td>
<td>57</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>23</td>
<td>39</td>
<td>43</td>
<td>57</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>23</td>
<td>32</td>
<td>43</td>
<td>57</td>
<td>39</td>
</tr>
</tbody>
</table>

- **pivot=32**
- **swap to end**
- **swap 39, 16**
- **swap 32 back in**

<table>
<thead>
<tr>
<th>81</th>
<th>75</th>
<th>92</th>
</tr>
</thead>
<tbody>
<tr>
<td>92</td>
<td>75</td>
<td>81</td>
</tr>
<tr>
<td>75</td>
<td>92</td>
<td>81</td>
</tr>
<tr>
<td>75</td>
<td>81</td>
<td>92</td>
</tr>
</tbody>
</table>

- **pivot=81**
- **swap to end**
- **swap 92, 75**
- **swap 81 back in**
void quickSort(Vector<int>& v) {
    quickSortHelper(v, 0, v.size() - 1);
}

void quickSortHelper(Vector<int>& v, int min, int max) {
    if (min >= max) {
        // base case; no need to sort
        return;
    }

    // choose pivot; we'll use the first element (might be bad!)
    int pivot = v[min];
    swap(v, min, max); // move pivot to end

    // partition the two sides of the array
    int middle = partition(v, min, max - 1, pivot);

    swap(v, middle, max); // restore pivot to proper location

    // recursively sort the left and right partitions
    quickSortHelper(v, min, middle - 1);
    quickSortHelper(v, middle + 1, max);
}
// Partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
int partition(Vector<int>& v, int i, int j, int pivot) {
    while (i <= j) {
        // move index markers i,j toward center
        // until we find a pair of out-of-order elements
        while (i <= j && v[i] < pivot) { i++; }
        while (i <= j && v[j] > pivot) { j--; }

        if (i <= j) {
            swap(v, i++, j--);
        }
    }
    return i;
}

// Moves the value at index i to index j, and vice versa.
void swap(Vector<int>& v, int i, int j) {
    int temp = v[i]; v[i] = v[j]; v[j] = temp;
}
Choosing a better pivot

• Choosing the first element as the pivot leads to very poor performance on certain inputs (ascending, descending)
  – does not partition the array into roughly-equal size chunks

• Alternative methods of picking a pivot:
  – random: Pick a random index from $[\text{min} \ldots \text{max}]$
  – median-of-3: look at left/middle/right elements and pick the one with the medium value of the three:
    • $v[\text{min}]$, $v[(\text{max}+\text{min})/2]$, and $v[\text{max}]$
    • better performance than picking random numbers every time
    • provides near-optimal runtime for almost all input orderings

<table>
<thead>
<tr>
<th>index</th>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>8</td>
<td>18</td>
<td>91</td>
<td>-4</td>
<td>27</td>
<td>30</td>
<td>86</td>
<td>50</td>
<td>65</td>
<td>78</td>
<td>5</td>
<td>56</td>
<td>2</td>
<td>25</td>
<td>42</td>
<td>98</td>
<td>31</td>
</tr>
</tbody>
</table>