CSc 110, Spring 2017

Lecture 40: Sorting
Adapted from slides by Marty Stepp and Stuart Reges
Searching

• How many items are examined worse case for sequential search?

• How many items are examined worst case for binary search?

• An algorithm's efficiency can be expressed in terms of being proportional to its _________ size.

• Why is sequential search also known as linear search?
Linear vs. Logarithmic Growth
Sorting

• **sorting**: Rearranging the values in a list into a specific order (usually into their "natural ordering").
  
  • one of the fundamental problems in computer science
  • can be solved in many ways:
    • there are many sorting algorithms
    • some are faster/slower than others
    • some use more/less memory than others
    • some work better with specific kinds of data
    • some can utilize multiple computers / processors, ...

• **comparison-based sorting**: determining order by comparing pairs of elements:
  • <, >, ...
Sorting algorithms

• bogo sort: shuffle and pray
• selection sort: look for the smallest element, move to front
• bubble sort: swap adjacent pairs that are out of order
• insertion sort: build an increasingly large sorted front portion
• merge sort: recursively divide the list in half and sort it
• heap sort: place the values into a sorted tree structure
• quick sort: recursively partition list based on a middle value

other specialized sorting algorithms:
• bucket sort: cluster elements into smaller groups, sort them
• radix sort: sort integers by last digit, then 2nd to last, then ...
• ...
Bogo sort

• **bogo sort**: Orders a list of values by repetitively shuffling them and checking if they are sorted.
  • name comes from the word "bogus"

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!
Bogo sort code

# Places the elements of a into sorted order.
def bogo_sort(a):
    while (not is_sorted(a)):
        shuffle(a)

# Returns true if a's elements are in sorted order.
def is_sorted(a):
    for i in range(0, len(a) - 1):
        if (a[i] > a[i + 1]):
            return False
    return True
Selection sort

10  13  8  2  4  7
Selection sort

- **selection sort**: Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.

  The algorithm:
  - Look through the list to find the smallest value.
  - Swap it so that it is at index 0.
  - Look through the list to find the second-smallest value.
  - Swap it so that it is at index 1.
  ...

- Repeat until all values are in their proper places.
Selection sort example

- **Initial list:**

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | 22| 18| 12| -4| 27| 30| 36| 50| 7 | 68| 91 | 56  | 2  | 85 | 42 | 98 | 25 |

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

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | -4| 18| 12| 22| 27| 30| 36| 50| 7 | 68| 91 | 56  | 2  | 85 | 42 | 98 | 25 |

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | -4| 2 | 12| 22| 27| 30| 36| 50| 7 | 68| 91 | 56  | 18 | 85 | 42 | 98 | 25 |

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | -4| 2 | 7 | 22| 27| 30| 36| 50| 12| 68| 91 | 56  | 18 | 85 | 42 | 98 | 25 |
Selection sort code

# Rearranges the elements of a into sorted order using
# the selection sort algorithm.
def selection_sort(a):
    for i in range(0, len(a) - 1):
        # find index of smallest remaining value
        min = i
        for j in range(i + 1, len(a)):
            if (a[j] < a[min]):
                min = j
        # swap smallest value its proper place, a[i]
        swap(a, i, min)

def swap(a, i, j):
    if (i != j):
        temp = a[i]
        a[i] = a[j]
        a[j] = temp
Selection sort runtime

• How many comparisons does selection sort have to do?
  First round (N-1)
  Second round (N-2)
  Third round (N-3)
  ...
  2
  1

or

\( (N-1) + (N-2) + (N-3) + \ldots + 2 + 1 = N(N-1)/2 \)

• Selection sort examines a number of elements in proportional to \( N^2 \)
Similar algorithms

• **bubble sort**: Make repeated passes, swapping adjacent values
  • slower than selection sort (has to do more swaps)
Bubble sort

- **bubble sort**: Orders a list of values by repeatedly comparing adjacent values, swapping if the values are out of order.

The algorithm for a list of size N:
- Compare the first two adjacent values.
- Swap if the second is smaller than the first.
- Repeat until the the end of the list.
- Largest value is now at position N
- Decrement N by 1 and repeat.
Bubble sort runtime

• How many comparisons does selection sort have to do?

  First round (N-1)
  Second round (N-2)
  Third round (N-3)

  ...
  2
  1

or

  \[(N-1) + (N-2) + (N-3) + \ldots + 2 + 1 = \frac{N(N-1)}{2}\]

• Bubble sort examines a number of elements in proportional to \(N^2\)
Similar algorithms

<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>

- **insertion sort**: Shift each element into a sorted sub-list
  - faster than selection sort (examines fewer values)
Merge sort

- **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
Merge sort example
Merge halves code

# Merges the left/right elements into a sorted result.
# Precondition: left/right are sorted

def merge(result, left, right):
    i1 = 0  # index into left list
    i2 = 0  # index into right list

    for i in range(0, len(result)):
        if (i2 >= len(right) or (i1 < len(left) and left[i1] <= right[i2])):
            result[i] = left[i1]  # take from left
            i1 += 1
        else:
            result[i] = right[i2]  # take from right
            i2 += 1
Merge sort code

# Rearranges the elements of a into sorted order using
# the merge sort algorithm.
def merge_sort(a):
    if (len(a) >= 2):
        # split list into two halves
        left = a[0, len(a)//2]
        right = a[len(a)//2, len(a)]

        # sort the two halves
        merge_sort(left)
        merge_sort(right)

        # merge the sorted halves into a sorted whole
        merge(a, left, right)
Merge sort runtime

- How many comparisons does merge sort have to do?

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
</tr>
<tr>
<td>8000</td>
<td>0</td>
</tr>
<tr>
<td>16000</td>
<td>0</td>
</tr>
<tr>
<td>32000</td>
<td>15</td>
</tr>
<tr>
<td>64000</td>
<td>16</td>
</tr>
<tr>
<td>128000</td>
<td>47</td>
</tr>
<tr>
<td>256000</td>
<td>125</td>
</tr>
<tr>
<td>512000</td>
<td>250</td>
</tr>
<tr>
<td>1e6</td>
<td>532</td>
</tr>
<tr>
<td>2e6</td>
<td>1078</td>
</tr>
<tr>
<td>4e6</td>
<td>2265</td>
</tr>
<tr>
<td>8e6</td>
<td>4781</td>
</tr>
<tr>
<td>1.6e7</td>
<td>9828</td>
</tr>
<tr>
<td>3.3e7</td>
<td>20422</td>
</tr>
<tr>
<td>6.5e7</td>
<td>42406</td>
</tr>
<tr>
<td>1.3e8</td>
<td>88344</td>
</tr>
</tbody>
</table>