CMSC 132: Object-Oriented Programming II

Sorting
What Is Sorting?

• To arrange a collection of items in some specified order.
  • Numerical order
  • Lexicographical order

• **Input:** sequence $<a_1, a_2, \ldots, a_n>$ of numbers.
• **Output:** permutation $<a'_1, a'_2, \ldots, a'_n>$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

• Example
  
  - Start $\rightarrow$ 1 23 2 56 9 8 10 100
  - End $\rightarrow$ 1 2 8 9 10 23 56 100
Why Sort?

• A classic problem in computer science.
  • Data requested in sorted order
    • e.g., list students in increasing GPA order

• Searching
  • To find an element in an array of a million elements
    • Linear search: average 500,000 comparisons
    • Binary search: worst case 20 comparisons

• Database, Phone book
  • Eliminating duplicate copies in a collection of records
  • Finding a missing element, Max, Min
Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort
- Shell Sort
- \( T(n) = O(n^2) \) **Quadratic growth**
- In clock time

<table>
<thead>
<tr>
<th></th>
<th>10,000</th>
<th>20,000</th>
<th>50,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 sec</td>
<td>17 sec</td>
<td>5 min</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Double input -> 4X time
  - Feasible for small inputs, quickly unmanageable
- Halve input -> 1/4 time
  - Hmm... can recursion save the day?
  - If have two sorted halves, how to produce sorted full result?
Divide and Conquer

1. **Base case**: the problem is small enough, solve **directly**

2. **Divide** the problem into two or more **similar and smaller** subproblems

3. **Recursively** solve the subproblems

4. **Combine** solutions to the subproblems
Merge Sort

- Divide and conquer algorithm
- Worst case: $O(n \log n)$
- Stable
  - maintain the relative order of records with equal values
- Input: 12, 5, 8, 13, 8, 27
- Stable: 5, 8, 8, 12, 13, 27
- Not Stable: 5, 8, 8, 12, 13, 27
Merge Sort: Idea

Divide into two halves

A: FirstPart SecondPart

Recursively sort

FirstPart SecondPart

Merge

A is sorted!
Merge-Sort: Merge

A: Sorted
merge
L: Sorted
R: Sorted
Merge Example

A: 

L: 1 2 6 8

R: 3 4 5 7

i = 0

j = 0
Merge Example

A: 1

L: 1 2 6 8  

i = 1

R: 3 4 5 7  

j = 0
Merge Example

A:

1 2

L:
1 2 6 8

i = 2

R:
3 4 5 7

j = 0
Merge Example cont.

A:

1 2 3 4 5 6 7 8

L:

1 2 6 8

R:

3 4 5 7

i=4  k=8  j=4
Merge sort algorithm

MERGE-SORT A[1 . . n]

1. If n = 1, done.
2. Recursively sort A[ 1 . . ⌊n/2⌋ ] and A[ ⌊n/2⌋+1 . . n ].
3. “Merge” the 2 sorted lists.

Key subroutine: MERGE
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Merge sort (Example)
Analysis of merge sort

\[
\begin{align*}
T(n) & \quad \text{MERGE-SORT } A[1 \ldots n] \\
\Theta(1) & \quad 1. \text{ If } n = 1, \text{ done.} \\
2T(n/2) & \quad 2. \text{ Recursively sort } A[1 \ldots \lceil n/2 \rceil] \text{ and } A[\lfloor n/2 \rfloor + 1 \ldots n]. \\
\Theta(n) & \quad 3. \text{ “Merge” the 2 sorted lists.}
\end{align*}
\]
Analyzing merge sort

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1; \\
2T(n/2) + \Theta(n) & \text{if } n > 1.
\end{cases}
\]

\[
T(n) = \Theta(n \log n) \quad (n > 1)
\]
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$h = \log n$

$\Theta(1)$ #leaves = $n$

Total = $\Theta(n \log n)$
Memory Requirement

Needs additional $n$ locations because it is difficult to merge two sorted sets in place

A:

L: 1 2 6 8

R: 3 4 5 7
Merge Sort Conclusion

- Merge Sort: $O(n \log n)$
  - asymptotically beats insertion sort in the worst case
  - In practice, merge sort beats insertion sort for $n > 30$ or so
- Space requirement:
  - $O(n)$, not in-place
Heapsort

- Merge sort time is $O(n \log n)$ but still requires, temporarily, $n$ extra storage locations
- *Heapsort* does not require any additional storage
- As its name implies, heapsort uses a heap to store the array
Heapsort Algorithm

- When used as a priority queue, a heap maintains a smallest value at the top
- The following algorithm
  - places an array's data into a heap,
  - then removes each heap item \(O(n \log n)\) and moves it back into the array
- This version of the algorithm requires \(n\) extra storage locations

Heapsort Algorithm: First Version

1. Insert each value from the array to be sorted into a priority queue (heap).
2. Set \(i\) to 0
3. while the priority queue is not empty
4. Remove an item from the queue and insert it back into the array at position \(i\)
5. Increment \(i\)
Trace of Heapsort
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)

CMSC 132 Summer 2017
Trace of Heapsort (cont.)

![Heapsort Trace Diagram]
Trace of Heapsort (cont.)

76
/
/  
6   74
/
/
37 32 39
/
/
20 26 18 28 29
/
/
89

CMSC 132 Summer 2017
Trace of Heapsort (cont.)

[Diagram of a heap with nodes labeled 6, 20, 26, 18, 28, 29, 32, 37, 76, 74, 66, and 89]
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)

Diagram of a heap with root node 28, and nodes 37, 66, 26, 32, 39, 20, 6, 18, 74, 76, and 89.
Trace of Heapsort (cont.)

Continue until everything sorted
Revising the Heapsort Algorithm

- If we implement the heap as an array
  - each element removed will be placed at the end of the array, and
  - the heap part of the array decreases by one element

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
<td>76</td>
<td>74</td>
<td>37</td>
<td>32</td>
<td>39</td>
<td>66</td>
<td>20</td>
<td>26</td>
<td>18</td>
<td>28</td>
<td>29</td>
<td>6</td>
</tr>
</tbody>
</table>

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>37</td>
<td>74</td>
<td>26</td>
<td>32</td>
<td>39</td>
<td>66</td>
<td>20</td>
<td>6</td>
<td>18</td>
<td>28</td>
<td>29</td>
<td>89</td>
</tr>
</tbody>
</table>

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>37</td>
<td>66</td>
<td>26</td>
<td>32</td>
<td>39</td>
<td>29</td>
<td>20</td>
<td>6</td>
<td>18</td>
<td>28</td>
<td>76</td>
<td>89</td>
</tr>
</tbody>
</table>

<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>18</td>
<td>20</td>
<td>26</td>
<td>28</td>
<td>29</td>
<td>32</td>
<td>37</td>
<td>39</td>
<td>66</td>
<td>74</td>
<td>76</td>
<td>89</td>
</tr>
</tbody>
</table>
Algorithm for In-Place Heapsort

1. Build a heap by rearranging the elements in an unsorted array

2. While the heap is not empty

3. Remove the first item from the heap by swapping it with the last item in the heap and restoring the heap property
Analysis of Heapsort

- Because a heap is a complete binary tree, it has log $n$ levels
- Building a heap of size $n$ requires finding the correct location for an item in a heap with log $n$ levels
- Each insert (or remove) is $O(\log n)$
- With $n$ items, building a heap is $O(n \log n)$
- No extra storage is needed
QUICKSORT
Quicksort

- Developed in 1962
- Quicksort selects a specific value called a pivot and rearranges the array into two parts (called partitioning)
  - all the elements in the left subarray are less than or equal to the pivot
  - all the elements in the right subarray are larger than the pivot
  - The pivot is placed between the two subarrays
- The process is repeated until the array is sorted
Merge sort vs Quick Sort

### Merge sort

<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>13</td>
<td>89</td>
<td>46</td>
<td>22</td>
<td>57</td>
<td>76</td>
<td>98</td>
<td>34</td>
<td>66</td>
<td>83</td>
</tr>
</tbody>
</table>

**split**

<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>13</td>
<td>89</td>
<td>46</td>
<td>22</td>
<td>57</td>
<td>76</td>
<td>98</td>
<td>34</td>
<td>66</td>
<td>83</td>
</tr>
</tbody>
</table>

**sort recursively**

<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>13</td>
<td>22</td>
<td>46</td>
<td>57</td>
<td>89</td>
<td>76</td>
<td>98</td>
<td>34</td>
<td>66</td>
<td>83</td>
</tr>
</tbody>
</table>

**merge**

<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>13</td>
<td>22</td>
<td>34</td>
<td>46</td>
<td>57</td>
<td>66</td>
<td>76</td>
<td>83</td>
<td>89</td>
<td>98</td>
</tr>
</tbody>
</table>
Merge sort vs Quick Sort

| 13 | 89 | 46 | 22 | 57 | 76 | 98 | 34 | 66 | 83 |

Split (smart, extra work here)

| 13 | 46 | 22 | 57 | 34 | 89 | 76 | 98 | 66 | 83 |

sort recursively

| 13 | 22 | 34 | 46 | 57 | 66 | 76 | 83 | 89 | 98 |

Merge is not necessary
Trace of Quicksort

44  75  23  43  55  12  64  77  33
Trace of Quicksort (cont.)

Arbitrarily select the first element as the pivot
Trace of Quicksort (cont.)

Swap the pivot with the element in the middle
Trace of Quicksort (cont.)

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right.
Trace of Quicksort (cont.)

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right.
Quicksort Example (cont.)

44 is now in its correct position

12  33  23  43  44  55  64  77  75
Now apply quicksort recursively to the two subarrays
Trace of Quicksort (cont.)

Pivot value = 12
Trace of Quicksort (cont.)

Pivot value = 12
Trace of Quicksort (cont.)

Pivot value = 33

12 33 23 43 44 55 64 77 75
Trace of Quicksort (cont.)

Pivot value = 33

12  23  33  43  44  55  64  77  75
Trace of Quicksort (cont.)

Pivot value = 33

12  23  33  43  44  55  64  77  75
Trace of Quicksort (cont.)

Pivot value = 33

Left and right subarrays have single values; they are sorted
Trace of Quicksort (cont.)

Pivot value = 33

Left and right subarrays have single values; they are sorted
Trace of Quicksort (cont.)

Pivot value = 55

12 23 33 43 44 55 64 77 75
Trace of Quicksort (cont.)

Pivot value = 64

12 23 33 43 44 55 64 77 75
Trace of Quicksort (cont.)

Pivot value = 77

12 23 33 43 44 55 64 77 75
Trace of Quicksort (cont.)

Pivot value = 77
Trace of Quicksort (cont.)

Pivot value = 77
Trace of Quicksort (cont.)

Left subarray has single value; it is sorted
Trace of Quicksort (cont.)
Quick Sort Algorithm

/* quicksort the subarray from a[lo] to a[hi] */

void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
Partition

// partition the subarray a[lo..hi] so that a[lo..j-1] <= a[j] <= a[j+1..hi]
// and return the index j.
int partition(Comparable[] a, int lo, int hi) {
    int i = lo;
    int j = hi + 1;
    Comparable v = a[lo];
    while (true) {
        // find item on lo to swap
        while (less(a[++i], v))
            if (i == hi) break;
        /* find item on hi to swap */
        while (less(v, a[--j]))
            if (j == lo) break;
        // check if pointers cross
        if (i >= j) break;
        exch(a, i, j);
    }
    // put partitioning item v at a[j]
    exch(a, i, j);
    // now, a[lo .. j-1] <= a[j] <= a[j+1 .. hi]
    return j;
}
Analysis of Quicksort

- If the pivot value is a random value selected from the current subarray,
  - then statistically half of the items in the subarray will be less than the pivot and half will be greater
- If both subarrays have the same number of elements (best case), there will be $\log n$ levels of recursion
- At each recursion level, the partitioning process involves moving every element to its correct position—$n$ moves
- Quicksort is $O(n \log n)$, just like merge sort
Analysis of Quicksort (cont.)

- The array split may not be the best case, i.e. 50-50

- An exact analysis is difficult (and beyond the scope of this class), but, the running time will be bounded by a constant $x \cdot n \log n$
A quicksort will give very poor behavior if, each time the array is partitioned, a subarray is empty. In that case, the sort will be $O(n^2)$. Under these circumstances, the overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of quicksort a poor performer relative to the quadratic sorts.
Algorithm for Partitioning

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript \textit{first}. 

\begin{tabular}{|c|}
\hline
44 & 75 & 23 & 43 & 55 & 12 & 64 & 77 & 33 \\
\hline
\end{tabular}
If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript \text{first}.
Trace of Partitioning (cont.)

For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored light purple.
Trace of Partitioning (cont.)

For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored light purple.
Trace of Partitioning (cont.)

Search for the first value at the left end of the array that is greater than the pivot value

44  75  23  43  55  12  64  77  33
Search for the first value at the left end of the array that is greater than the pivot value.
Then search for the first value at the right end of the array that is less than or equal to the pivot value.
Then search for the first value at the right end of the array that is less than or equal to the pivot value.
Trace of Partitioning (cont.)

Exchange these values
Trace of Partitioning (cont.)

Exchange these values
Trace of Partitioning (cont.)

Repeat
Find first value at left end greater than pivot
Find first value at right end less than or equal to pivot
Trace of Partitioning (cont.)

Exchange

44  33  23  43  12  55  64  77  75
Trace of Partitioning (cont.)

Repeat
Find first element at left end greater than pivot
Trace of Partitioning (cont.)

Find first element at right end less than or equal to pivot
Since \textit{down} has "passed" \textit{up}, do not exchange
Trace of Partitioning (cont.)

Exchange the pivot value with the value at down
Trace of Partitioning (cont.)

Exchange the pivot value with the value at \textit{down}
The pivot value is in the correct position; return the value of \texttt{down} and assign it to the pivot index \texttt{pivIndex}
Code for partition when Pivot is the largest or smallest value
Revised Partition Algorithm

- Quicksort is $O(n^2)$ when each split yields one empty subarray, which is the case when the array is presorted.
- A better solution is to pick the pivot value in a way that is less likely to lead to a bad split.
  - Use three references: first, middle, last.
  - Select the median of these items as the pivot.
Trace of Revised Partitioning
Trace of Revised Partitioning (cont.)
Trace of Revised Partitioning (cont.)

Sort these values
Trace of Revised Partitioning (cont.)

Sort these values

first: 33, 75, 23, 43, 44, 12, 64, 77, 55
middle: 43, 44, 12
last: 77, 55
Trace of Revised Partitioning (cont.)

Exchange middle with first
Trace of Revised Partitioning (cont.)

Exchange middle with first
Run the partition algorithm using the first element as the pivot
## Sorting Algorithm Comparison

<table>
<thead>
<tr>
<th>Name</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Memory</th>
<th>Stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>$n$</td>
<td>$n^2$</td>
<td>$n^2$</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
<td>$n^2$</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$n$</td>
<td>$n^2$</td>
<td>$n^2$</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n$</td>
<td>yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n^2$</td>
<td>$\log n$</td>
<td>no</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>1</td>
<td>no</td>
</tr>
</tbody>
</table>