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 \(a'_1 \leq a'_2 \leq \cdots \leq a'_n\).

• 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>Input</th>
<th>Time</th>
<th>Input</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>3 sec</td>
<td>20,000</td>
<td>17 sec</td>
</tr>
<tr>
<td>50,000</td>
<td>77 sec</td>
<td>100,000</td>
<td>5 min</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
Stable Sort Example

\[(Dave, A)\]
\[(Alice, B)\]
\[(Ken, A)\]
\[(Eric, B)\]
\[(Carol, A)\]

Sort by name

Now, sort by section

\[(Carol, A)\]
\[(Dave, A)\]
\[(Ken, A)\]
\[(Eric, B)\]
\[(Alice, B)\]

Not Stable

\[(Alice, B)\]
\[(Carol, A)\]
\[(Dave, A)\]
\[(Eric, B)\]
\[(Ken, A)\]

Stable
Merge Sort: Idea

1. Divide into two halves

2. Recursively sort

3. Merge

4. A is sorted!
Merge-Sort: Merge

A: Sorted

merge

L: Sorted

R: Sorted
Merge Example

A:

L:

R:

i = 0

j = 0
Merge Example

\[ A: \begin{array}{cccccc}
1 & & & & & \\
\end{array} \]

\[ L: \begin{array}{cccc}
1 & 2 & 6 & 8 \\
\end{array} \]

\[ i=1 \]

\[ R: \begin{array}{cccccc}
3 & 4 & 5 & 7 \\
\end{array} \]

\[ j=0 \]
Merge Example

A:

L:

R:

\[ i=2 \]

\[ j=0 \]
Merge Example cont.

A:

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

L:

\[
\begin{array}{cccc}
1 & 2 & 6 & 8 \\
\end{array}
\]

R:

\[
\begin{array}{cccc}
3 & 4 & 5 & 7 \\
\end{array}
\]

\[i = 4\]

\[k = 8\]

\[j = 4\]
Merge sort algorithm

\[
\text{MERGE-SORT } A[1 \ldots n]
\]

1. If \( n = 1 \), done.
2. Recursively sort \( A[1 \ldots \lceil n/2 \rceil] \) and \( A[\lfloor n/2 \rfloor+1 \ldots n] \).
3. “Merge” the 2 sorted lists.

\textit{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

\[ T(n) \]
\[ \Theta(1) \]
\[ 2T(n/2) \]
\[ \Theta(n) \]

MERGE-SORT \[ A[1 \ldots n] \]

1. If \( n = 1 \), done.
2. Recursively sort \( A[1 \ldots \lceil n/2 \rceil] \) and \( A[\lceil n/2 \rceil+1 \ldots n] \).
3. “Merge” the 2 sorted lists.
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.

$h = \log n$

#leaves = $n$

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

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

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

1. Insert each value from the array to be sorted into a priority queue (min-heap).
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the sink() function on the list to sink the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.
Trace of Heapsort
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.)

```
76   37   74   26   32   39   66   20   6   18   28   29   89
```
Trace of Heapsort (cont.)

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

CMSC 132 Summer 2020
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)

```
20  6  18  28  76  89
26  37  32  39  66  29
  74
```
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)

Continue until everything sorted
Revising the Heapsort Algorithm

- Each element removed will be placed at the end of the array.
- The heap part of the array decreases by one element.
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

- 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

13 89 46 22 57 76 98 34 66 83

split

13 89 46 22 57 76 98 34 66 83

sort recursively

13 22 46 57 89 76 98 34 66 83

merge

13 22 34 46 57 66 76 83 89 98
Merge sort vs Quick Sort

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

Split (smart, extra work here)

<= 57
| 13 | 46 | 22 | 57 | 34 |

> 57
| 89 | 76 | 98 | 66 | 83 |

sort recursively

| 13 | 22 | 34 | 46 | 57 |

| 66 | 76 | 83 | 89 | 98 |

Merge is not necessary
Trace of Quicksort
Trace of Quicksort (cont.)

```
44  75  23  43  55  12  64  77  33
```

- Move $i$ if $a[i] > \text{pivot}$
- Move $j$ if $a[j] < \text{pivot}$
Trace of Quicksort (cont.)

pivot

Swap(a[i], a[j])

44 33 23 43 55 12 64 77 75

i

j
Trace of Quicksort (cont.)

Move i if \( a[i] > \text{pivot} \)
Move j if \( a[j] < \text{pivot} \)
Trace of Quicksort (cont.)

\[
\begin{array}{cccccccc}
44 & 33 & 23 & 43 & 12 & 55 & 64 & 77 & 75 \\
\end{array}
\]

\[\text{Swap}(a[i], a[j])\]
Trace of Quicksort (cont.)

```
44  33  23  43  12  55  64  77  75
```

pivot

Move i if \( a[i] > \) pivot
Move j if \( a[j] < \) pivot
Trace of Quicksort (cont.)

Break if $i \geq j$
Trace of Quicksort (cont.)

One iteration is done

Swap(pivot, a[j])
Recursively sort first and second subarray
Trace of Quicksort (cont.)

Move $i$ if $a[i] > \text{pivot}$
Move $j$ if $a[j] < \text{pivot}$
Trace of Quicksort (cont.)

Break if $i \geq j$
Trace of Quicksort (cont.)

Second iteration for left half is done
Trace of Quicksort (cont.)

Recursively sort second subarray
Trace of Quicksort (cont.)

Move $i$ if $a[i] > \text{pivot}$
Move $j$ if $a[j] < \text{pivot}$
Trace of Quicksort (cont.)

Break if $i \geq j$

33 23 43

pivot
Trace of Quicksort (cont.)

Another iteration is done

Swap(pivot, a[j])
Trace of Quicksort (cont.)

Subarray to sort
Trace of Quicksort (cont.)

Sorted

Recursively sort the second subarray
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, lo, 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.)

- 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.
If Pivot is the largest or smallest value
Revised Partition Algorithm

- 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

10  75  23  43  90  12  64  77  50
Trace of Revised Partitioning (cont.)

first

10

middle

75

23

43

90

12

64

last

77

50
Trace of Revised Partitioning (cont.)

Make the middle number 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>$\text{nlog}n$</td>
<td>$\text{nlog}n$</td>
<td>$\text{nlog}n$</td>
<td>$n$</td>
<td>yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$\text{nlog}n$</td>
<td>$\text{nlog}n$</td>
<td>$n^2$</td>
<td>$\log n$</td>
<td>no</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$\text{nlog}n$</td>
<td>$\text{nlog}n$</td>
<td>$\text{nlog}n$</td>
<td>1</td>
<td>no</td>
</tr>
</tbody>
</table>