CS250 Winter 2003

Week 10, Lecture 3:

Sorting:

review of sorting algorithms
and some extra ones
Sorting review: merge sort

mergeSort(A,p,r)
    if p<r then
        q = floor( (p+r)/2 )
        mergeSort(A,p,q)
        mergeSort(A,q+1,r)
        merge(A,p,q,r)

mergeSort(A,0,n-1) sorts A of length n. E.g.:

4, 2, 9, 1, 5, 3, 2, 8

Gets split into sequences of length 1:

4   2   9   1   5   3   2   8
merge  merge  merge  merge
2,4  1,9  3,5  2,8
merge  merge
1,2,4,9 2,3,5,8
merge
1,2,2,3,4,5,8,9

Merging metaphor: a deck of cards, split in sorted halves, is merged into one sorted pile.
Example of recursive calls and returns.

mergeSort([4,2,9,1,5,3,2,8])
    mergeSort([4,2,9,1])
        mergeSort([4,2])
            mergeSort([4])
            mergeSort([2])
            merge([4],[2]) = [2,4]
    mergeSort([9,1])
        mergeSort([9])
        mergeSort([1])
        merge([9],[1]) = [1,9]
    merge([2,4],[1,9]) = [1,2,4,9]
mergeSort([5,3,2,8])
    mergeSort([5,3])
        mergeSort([5])
        mergeSort([3])
        merge([5],[3]) = [3,5]
    mergeSort([2,8])
        mergeSort([2])
        mergeSort([8])
        merge([2],[8]) = [2,8]
    merge([3,5],[2,8]) = [2,3,5,8]
merge([1,2,4,9],[2,3,5,8]) = [1,2,2,3,4,5,8,9]
Sorting review: insertion sort

insertionSort(L)
    if L is not empty then
        a = pop(L)
        insertionSort(L)
        insert(a, L)  // a in position in sorted L

Run on: [5, [2, [4, [3, [1, [6, []]]]]]]
Gets split into lists of length 1:

    insert
[5] [2] [4] [3] [1, [6]]
    insert
[5] [2] [4] [1, [3, [6]]]
    insert
[5] [2] [1, [3, [4, [6]]]]
    insert
[5] [1, [2, [3, [4, [6]]]]]
    insert
[1, [2, [3, [4, [5, [6]]]]]]
Example of recursive calls and returns.

\[\text{insertionSort([5, [2, [4, [3, [1, [6]]]]]])}\]
\[a= 5 \text{ and insertionSort([2, [4, [3, [1, [6]]]]])}\]
\[a= 2 \text{ and insertionSort([4, [3, [1, [6]]]])}\]
\[a= 4 \text{ and insertionSort([3, [1, [6]]])}\]
\[a= 3 \text{ and insertionSort([1, [6]])}\]
\[a= 1 \text{ and insertionSort([6])}\]
\[a= 6 \text{ and insertionSort([])}\]
\[\text{insert(6, []): L = [6]}\]
\[\text{insert(1, [6]): L = [1, [6]}\]
\[\text{insert(3, [1, [6]]): L = [1, [3, [6]]]}\]
\[\text{insert(4, [1, [3, [6]]]): L = [1, [3, [4, [6]]]}\]
\[\text{insert(2, [1, [3, [6]]]): L = [1, [2, [3, [4, [6]]]}\]
\[\text{insert(5, [1, [3, [6]]]): L = [1, [2, [3, [4, [5, [6]]]}\]
Sorting review: selection sort

selectionSort(A)
    for i = 1 to n-1 do  // finds the ith smallest
        smallest = i
        for j = i to n do
            swap(A[j], A[smallest])

Run on: $A = [5, 2, 4, 6, 1, 3, 2, 6]$

i=1: j=1 to 8, finds smallest=5
    $A = [1, 2, 4, 6, 5, 3, 2, 6]$  

i=2: j=2 to 8, finds smallest=7
    $A = [1, 2, 4, 6, 5, 3, 2, 6]$  

i=3: j=3 to 8, finds smallest=7
    $A = [1, 2, 4, 6, 5, 3, 2, 6]$  

i=4: j=4 to 8, finds smallest=6
    $A = [1, 2, 2, 6, 5, 3, 4, 6]$  

i=5: j=5 to 8, finds smallest=7
    $A = [1, 2, 2, 3, 4, 6, 5, 6]$  

...
Algorithm for sorted array: binary search

`binarySearch(D, k, 0, n-1)` searches for `k` in `D` of length `n`.

```
binarySearch(D, k, low, high)
    if (low > high) return(FAILURE)
    else { mid = floor((low+high)/2)
        if (k = D[mid]) return(D[mid])
        else if (k < D[mid])
            return(binarySearch(D, k, low, mid-1))
        else if (k > D[mid])
            return(binarySearch(D, k, mid+1, high))
    }
```

Metaphor: the children game of high-low. Alice chooses a number between 1 and 100. Bob guesses a number between 1 and 100 and tells Alice. Alice tells Bob if the guess is higher or lower or right. If it is not right, Bob, recurses from the guessing point.
binarySearch for 14 on [2, 4, 5, 8, 9, 14, 22, 27, 28, 31]

[2, 4, 5, 8, 9, 14, 22, 27, 28, 31]
low mid high

[2, 4, 5, 8, 9, 14, 22, 27, 28, 31]
low mid high

[2, 4, 5, 8, 9, 14, 22, 27, 28, 31]
low high

Recursive calls (and only one return):

binarySearch([2, 4, 5, 8, 9, 14, 22, 27, 28, 31], 14, 0, 9)
mid = floor(9/2) = 4
// D[mid]=9 < 14, so search in upper half
binarySearch([14, 22, 27, 28, 31], 14, 5, 9)
mid = floor((5+9)/2) = 7
// D[mid]=27 > 14, so search in lower half
binarySearch([14, 22], 14, 5, 6)
mid = floor((5+6)/2) = 5
// found D[mid]=14
return mid=5
Sorting review: heap sort

heapSort(A) sorts an array $A$ of length $n$ using a priority queue on integers $Q$.

heapsort(A)
   for $i = 0$ to $n - 1$
      Q.insert(A[i])
   for $i = 0$ to $n - 1$
      A[i] = Q.removeMin()

Recall that a priority queue is an ADT that stores Objects and enables inserting and removing the minimum inserted Object. Note that priority queues have nothing to do with FIFO. A priority queue can be efficiently implemented by using a heap ADT. [Lecture 9-3]

The heap sort is the slowest of the $O(n \log n)$ sorting algorithms, but unlike the merge (and quick) sorts, it does not require massive recursion or multiple arrays to work. This makes it the most attractive option for very large data sets of millions of items.
Bubble sort

Bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass through the list without swapping any items.

This causes larger values to “bubble” to the end of the list while smaller values “sink” towards the beginning of the list.

bubbleSort(A) sorts array $A[1..n]$ of length $n$:

```plaintext
bubbleSort(A)
   for i = n to 1
      for j = 2 to i
         if (A[j-1] > A[j])
            swap(A[j-1], A[j])
```

The bubble sort is the oldest and simplest sort in use. It also is the slowest of the $O(n^2)$ sorts.
Quick sort

Quick sort is an **in-place**, divide-and-conquer, massively recursive sort. It is akin to a faster version of merge sort. The algorithm is simple in theory, but difficult to put into pseudo-code.

The recursive algorithm consists of four steps:

1. If there are one or less elements in the array to be sorted, return immediately.

2. Pick an element to serve as a “**pivot**” point. Usually, it is the left-most element.

3. Split the array in two: the elements larger than the pivot and the smaller ones.

4. Recursively repeat the algorithm for both halves of the original array.

Can you rephrase this in terms of a divide-and-conquer strategy?
Algorithm quickSort(A) sorts an array $A[r..l]$:

quickSort(A, l, r) {
    pivot = A[l]
    i = l
    j = r
    // grow the bottom and top regions w.r.t pivot
    while (1) {
        while (1) { // throw right to left first
            if (i == j) { A[i] = pivot
                break
            }
                break
            }
            else j--
        }
        while (1) { // now throw left to right
            if (i == j) { A[i] = pivot
                break
            }
            else if (A[i] > pivot) { A[j--] = A[i]
                break
            }
            else i++
        }
    }
    if (l < i) quickSort(A, l, i-1)
    if (r > i) quickSort(A, i+1, r)
}
Example of a run:

\[
\begin{align*}
[5,3,2,6,4,1,4,7] & \quad [4,3,2,6,4,1,4,7] \\
i & \quad i \\
[4,3,2,6,4,1,6,7] & \quad [4,3,2,1,4,1,6,7] \\
i & \quad i \\
i/j & \quad i/j \\
[1,3,2,1,4] 5 [6,7] & \quad [1,3,2] 4 4 5 [6,7] \\
i & \quad i/j \\
[1,3,2] 4 4 5 [6,7] & \quad [3,2] 4 4 5 [6,7] \\
i & \quad i/j \\
i & \quad i/j
\end{align*}
\]

The efficiency depends on which element is chosen as the pivot point. The worst-case efficiency of the quick sort, \(O(n^2)\), occurs when the list is sorted and the left-most element is chosen. If the pivot point is chosen randomly, quick sort runs in \(O(n \log n)\). (Proof omitted.)

Quick sort is the fastest of the common sorts. While it is possible to write a special-purpose sorting algorithm that can beat quick sort for some data sets, for general-case sorting there is nothing faster.

Should one use quick sort for everything? No! Why? But faster is better...? Why would one bother with heap sort or merge sort, for instance? In which cases?
Counting (or bucket) sort

All sorting algorithms so far share the property that the sorted order they determine is based only on comparisons between the input elements. Such sorting algorithms are called **comparison sorts**.

It can be shown that the running time of any comparison sort for sorting an \( n \)-element sequence is \( \Omega(n \log n) \) in the worst case. (Proof: Proposition 9.4 in Goodrich & Tamassia.)

Nevertheless, if we have access to more information about the sequence, linear-time sorting is achievable.

For instance, if each of the \( n \) input elements in \( A \) is an integer in the range 1 to \( k \), for some integer \( k = \mathcal{O}(n) \), then `countingSort` runs in \( \mathcal{O}(n) \) time.
The input is in $A[1..n]$, the sorted output is in $B[1..n]$, and $C[1..k]$ is for temporary storage.

countingSort(A, B, k) {
    for $i = 1$ to $k$
        $C[i] = 0$

    // make $C[i]$ contain the # of elements = $i$
    for $j = 1$ to $n$
        $C[A[j]] +=$

    // make $C[i]$ contain the # of elements <= $i$
    for $i = 2$ to $k$
        $C[i] += C[i-1]$

    for $j = n$ to 1 {
        $B[C[A[j]]] = A[j]$
        $C[A[j]] --$
    }
}

Example of a run:

\[
A = [3,6,4,1,3,4,1,4] \\
C = [2,0,2,3,0,1] \Rightarrow [2,2,4,7,7,8]
\]

\[
B = [ , , , , , ,4, ] \text{ and } C = [2,2,4,6,7,8] \\
B = [ ,1, , , , ,4, ] \text{ and } C = [1,2,4,6,7,8] \\
B = [ ,1, , , ,4,4, ] \text{ and } C = [1,2,4,5,7,8] \\
B = [ ,1, ,3, ,4,4, ] \text{ and } C = [1,2,3,5,7,8] \\
B = [1,1, ,3, ,4,4, ] \text{ and } C = [0,2,3,5,7,8] \\
B = [1,1, ,3,4,4,4, ] \text{ and } C = [0,2,3,4,7,8] \\
B = [1,1, ,3,4,4,4,6] \text{ and } C = [0,2,3,4,7,7] \\
B = [1,1,3,3,4,4,4,6] \text{ and } C = [0,2,2,4,7,7]
\]

An important property of countingSort is that it is **stable**: numbers with the same value in \( A \) appear in \( B \) in the same position as they appeared in \( A \). This is important when satellite data are carried around with the element being stored. Can you provide a proof of this?
Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Lectures</th>
<th>Running time</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge sort</td>
<td>3-2, 7-1</td>
<td>$O(n \log n)$</td>
<td>9.1</td>
</tr>
<tr>
<td>insertion sort</td>
<td>3-3, 7-1</td>
<td>$O(n^2)$</td>
<td>7.2.3</td>
</tr>
<tr>
<td>selection sort</td>
<td>4-3, 7-1</td>
<td>$O(n^2)$</td>
<td>7.2.3</td>
</tr>
<tr>
<td>heap sort</td>
<td>9-3</td>
<td>$O(n \log n)$</td>
<td>7.3.3</td>
</tr>
<tr>
<td>bubble sort</td>
<td>10-3</td>
<td>$O(n^2)$</td>
<td>5.4</td>
</tr>
<tr>
<td>quick sort</td>
<td>10-3</td>
<td>$O(n \log n)$</td>
<td>9.3</td>
</tr>
<tr>
<td>counting sort</td>
<td>10-3</td>
<td>$O(n)$</td>
<td>9.5</td>
</tr>
</tbody>
</table>

If two algorithms performing the same task have the same big-O complexity, how does one determine if one is better to use than another?

Are there cases when one would use an algorithm with a larger big-O complexity?