Sorting

Order in the court!
Importance of sorting

- Sorting a list of values is a fundamental task of computers - this task is one of the primary reasons why people use computers in the first place.
- Many sorting algorithms have been developed over the history of computer science.
Quadratic sorting algorithms

• Algorithms with order of magnitude $O(N^2)$ are known as quadratic algorithms.

• Such algorithms are particularly sensitive to the size of the data set; sorting times increase geometrically as the size of the data set grows.

• Their quadratic behavior is their chief disadvantage; however, they are easy to comprehend and code, and work reasonably well on relatively small data sets.
Quadratic sorting algorithms

• We will examine two examples of quadratic sorting algorithms:
  – Selectionsort
  – Insertionsort

• Each of these algorithms follows the same principle: sort a portion of the array, then progress through the unsorted portion, adding one element at a time to the sorted portion
Selectionsort

• Goal of the algorithm is to sort a list of values (for example, integers in an array) from smallest to largest

• The method employed comes directly from this statement of the problem
  – find smallest value and place at front of array
  – find next-smallest value and place in second position
  – find next-next-smallest and place in third position
  – and so on ...
Selectionsort

• The mechanics of the algorithm are simple: swap the smallest element with whatever is in the first position, then move to the second position and perform a similar swap, etc.

• In the process, a sorted subarray grows from the front, while the remaining unsorted subarray shrinks toward the back.
Sorting an Array of Integers

- The picture shows an array of six integers that we want to sort from smallest to largest.
The Selectionsort Algorithm

• Start by finding the **smallest** entry.
• Swap the smallest entry with the **first** entry.
The Selectionsort Algorithm

- Part of the array is now sorted.
The Selectionsort Algorithm

- Find the smallest element in the unsorted side.
The Selectionsort Algorithm

- Swap with the front of the unsorted side.
The Selectionsort Algorithm

- We have increased the size of the sorted side by one element.

![Bar chart showing the sorted and unsorted sides of the selection sort algorithm.](chart.png)
The Selectionsort Algorithm

- The process continues...

Sorted side

Unsorted side

Smallest from unsorted
The Selectionsort Algorithm

- The process continues...

![Diagram showing the Selectionsort Algorithm with an arrow indicating swap with front.]
The Selectionsort Algorithm

- The process continues...

Sorted side is bigger
The Selectionsort Algorithm

- The process keeps adding one more number to the sorted side.
- The sorted side has the smallest numbers, arranged from small to large.
The Selectionsort Algorithm

- We can stop when the unsorted side has just one number, since that number must be the largest number.
The Selectionsort Algorithm

- The array is now sorted.
- We repeatedly selected the smallest element, and moved this element to the front of the unsorted side.
Implementation of Selectionsort

```java
public void selectionSort (int data[]) {
    int mindex, tmp;
    for (int x = 0; x <= data.length-2; x++)
    {
        mindex = x;
        for (int y = x+1; y <= data.length-1; y++)
            if (data[y] < data[mindex])
                mindex = y;
        tmp = data[x];
        data[x] = data[mindex];
        data[mindex] = tmp;
    }
}
```
Perform selectionsort on the following set of numbers – show the array after each step

5, 29, 43, 17, 99, 56, 72, 38
Time Analysis of Selectionsort

• The driving element of the selectionsort function is the set of for loops, each of which is $O(N)$

• Both loops perform the same number of operations no matter what values are contained in the array (even if all are equal, for example) -- so worst-case, average-case and best-case are all $O(N^2)$
Insertionsort

- Although based on the same principle as Selectionsort (sorting a portion of the array, adding one element at a time to the sorted portion), Insertionsort takes a slightly different approach.
- Instead of selecting the smallest element from the unsorted side, Insertionsort simply takes the first element and inserts it in place on the sorted side so that the sorted side is always in order.
Insertionsort algorithm

- Designate first element as sorted
- Take first element from unsorted side and insert in correct location on sorted side:
  - copy new element
  - shift elements from end of sorted side to the right (as necessary) to make space for new element
Insertionsort algorithm

- Correct location for new element found when:
  - front of array is reached or
  - next element to shift is \( \leq \) new element
- Continue process until last element has been put into place
The Insertionsort Algorithm

- The Insertionsort algorithm also views the array as having a sorted side and an unsorted side.
The Insertionsort Algorithm

- The sorted side starts with just the first element, which is not necessarily the smallest element.
The Insertion Sort Algorithm

• The sorted side grows by taking the front element from the unsorted side...
The Insertionsort Algorithm

- ...and inserting it in the place that keeps the sorted side arranged from small to large.
The Insertionsort Algorithm

- In this example, the new element goes in front of the element that was already in the sorted side.
The Insertionsort Algorithm

- Sometimes we are lucky and the new inserted item doesn't need to move at all.
The Insertionsort Algorithm

• Sometimes we are lucky twice in a row.
How to Insert One Element

1. Copy the new element to a separate location.
How to Insert One Element

1. Shift elements in the sorted side, creating an open space for the new element.
How to Insert One Element

1. Shift elements in the sorted side, creating an open space for the new element.
How to Insert One Element

2. Continue shifting elements...
How to Insert One Element

Continue shifting elements...
How to Insert One Element

...until you reach the location for the new element.
How to Insert One Element

3 Copy the new element back into the array, at the correct location.
How to Insert One Element

• The last element must also be inserted. Start by copying it...
How to Insert One Element

• Four items are shifted.
• And then the element is copied back into the array.
Implementation of InsertionSort

```java
public void insertionSort (int [] array) {
    int x, y, tmp;
    for (x=1; x<array.length; x++) {
        tmp = array[x];
        for (y=x; y>0 && array[y-1] > tmp; y--)
            array[y] = array[y-1];
        array[y] = tmp;
    }
}
```
Perform insertionsort on the following set of numbers – show the array after each step

5, 29, 43, 17, 99, 56, 72, 38
Time analysis of Insertionsort

- In the worst case and in the average case, Insertionsort’s order of magnitude is $O(N^2)$.
- But in the best case (when the starting array is already sorted), the algorithm is $O(N)$ because the inner loop doesn’t go.
- If an array is nearly sorted, Insertionsort’s performance is nearly linear.
Recursive sorting algorithms

• Recursive algorithms are considerably more efficient than quadratic algorithms
• The downside is they are harder to comprehend and thus harder to code
• Two examples of recursive algorithms that use a divide and conquer paradigm are Mergesort and Quicksort
Divide and Conquer sorting paradigm

• Divide elements to be sorting into two (nearly) equal groups
• Sort each of these smaller groups (by recursive calls)
• Combine the two sorted groups into one large sorted list
Using pointer arithmetic to specify subarrays

• The divide and conquer paradigm requires division of the element list (e.g. an array) into two halves

• Pointer arithmetic facilitates the splitting task
  – for array with n elements, for any integer x from 0 to n, the expression (data+x) refers to the subarray that begins at data[x]
  – in the subarray, (data+x)[0] is the same as data[x], (data+x)[1] is the same as data[x+1], etc.
Mergesort

• Straightforward implementation of divide and conquer approach

• Strategy:
  – divide array at or near midpoint
  – sort half-arrays via recursive calls
  – merge the two halves
Mergesort

• Two functions will be implemented:
  – mergeSort: simpler of the two; divides the array and performs recursive calls, then calls merge function
  – merge: uses dynamic array to merge the subarrays, then copies result to original array
Mergesort in action

Original array

| 40 | 22 | 13 | 8 | 51 | 29 | 36 | 11 |

First split

| 40 | 22 | 13 | 8 | 51 | 29 | 36 | 11 |

Second split

| 40 | 22 | 13 | 8 | 51 | 29 | 36 | 11 |

Third split

| 40 | 22 | 13 | 8 | 51 | 29 | 36 | 11 |

Stopping case is reached for each array when array size is 1; once recursive calls have returned, merge function is called.
Mergesort method

public static void mergesort (int data[], int first, int n)
{
    int n1;
    int n2;
    if (n>1)
    {
        n1 = n/2;
        n2 = n-n1;
        mergesort(data, first, n1);
        mergesort(data, first+n1, n2);
        merge(data, first, n1, n2);
    }
}

Merge method

• Uses temporary array (temp) that copies items from data array so that items in temp are sorted

• Before returning, the method copies the sorted items back into data

• When merge is called, the two halves of data are already sorted, but not necessarily with respect to each other
Merge method in action

Two subarrays are assembled into a merged, sorted array with each call to the merge function; as each recursive call returns, a larger array is assembled.
How merge works

- As shown in the previous example, the first call to merge deals with several half-arrays of size 1; merge creates a temporary array large enough to hold all elements of both halves, and arranges these elements in order.
- Each subsequent call performs a similar operation on progressively larger, presorted half-arrays.
How merge works

• Each instance of merge contains the dynamic array temp, and three counters: one that keeps track of the total number of elements copied to temp, and one each to track the number of elements copied from each half-array

• The counters are used as indexes to the three arrays: the first half-array (data), the second half-array (data + n1) and the dynamic array (temp)
How merge works

- Elements are copied one-by-one from the two half-arrays, with the smallest elements from both copied first.
- When all elements of either half-array have been copied, the remaining elements of the other half-array are copied to the end of temp.
- Because the half-arrays are pre-sorted, these “leftover” elements will be the largest values in each half-array.
The copying procedure from merge (pseudocode)

while (both half-arrays have more elements to copy)
{
    if (next element from first <= next element from second)
    {
        copy element from first half to next spot in temp
        add 1 to temp and first half counters
    }
    else
    {
        copy element from second half to next spot in temp
        add 1 to temp and second half counters
    }
}
First refinement

// temp counter = c, first counter = c1, second counter = c2
while (both 1st & 2nd half-arrays have more elements to copy)
{
    if (data[c1] <= (data + n1)[c2])
    {
        temp[c] = data[c1];
        c++; c1++;
    }
    else
    {
        temp[c] = (data + n1)[c2]);
        c++; c2++;
    }
}
Final version -- whole algorithm in pseudocode

Initialize counters to zero
while (more to copy from both sides)
{
    if (data[c1] <= (data+n1)[c2])
        temp[c++] = data[c1++];
    else
        temp[c++] = (data+n1)[c2++];
}
Copy leftover elements from either 1st or 2nd half-array
Copy elements from temp back to data
public static void merge (int [] data, int first, int n1, int n2) {
    int [] temp = new int[n1 + n2];
    int copied = 0, copied1 = 0, copied2 = 0, i;
    while ((copied1 < n1) && (copied2 < n2)) {
        if (data[first + copied1] < data[first + n1 + copied2])
            temp[copied++] = data[first + (copied1++)];
        else
            temp[copied++] = data[first + n1 + (copied2++)];
    }
Code for merge

while (copied1 < n1)
    temp[copied++] = data[first + (copied1++)];
while (copied2 < n2)
    temp[copied++] = data[first + n1 + (copied2++)];

for (i=1; i < n1+n2; i++)
    data[first+1] = temp[i];
}
Perform mergesort on the following set of numbers – show the array after each step

5, 29, 43, 17, 99, 56, 72, 38
Time analysis of mergesort

• Start with array of N elements
• At top level, 2 recursive calls are made to sort subarrays of N/2 elements, ending up with one call to merge
• At the next level, each N/2 subarray is split into two N/4 subarrays, and 2 calls to merge are made
• At the next level, 4 N/8 subarrays are produced, and 4 calls to merge are made
• and so on ...
Time analysis of mergesort

• The pattern continues until no further subdivisions can be made

• Total work done by merging is \( O(N) \) (for progressively smaller sizes of \( N \) at each level)

• So total cost of mergesort may be presented by the formula:
  
  \[ \text{(some constant)} \times N \times \text{(number of levels)} \]
Time analysis of mergesort

• Since the size of the array fragments is halved at each step, the total number of levels is approximately equal to the number of times N can be divided by two.

• Given this, we can refine the formula:

\[(\text{some constant}) \times N \times \log_2 N\]

• Since the big-O form ignores constants, we can state that mergesort’s order of magnitude is \(O(N \log N)\) in the worst case.
## Comparing $O(N^2)$ to $O(N \log N)$

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N \log N$</th>
<th>$N^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>24</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>160</td>
<td>1024</td>
</tr>
<tr>
<td>128</td>
<td>896</td>
<td>16,384</td>
</tr>
<tr>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
</tr>
<tr>
<td>2,048</td>
<td>22,528</td>
<td>4,194,304</td>
</tr>
</tbody>
</table>
Application of Mergesort

• Although more efficient than a quadratic algorithm, mergesort is not the best choice for sorting arrays because of the need for a temporary dynamic array in the merge step.

• The algorithm is often used for sorting files.

• With each call to mergesort, the file is split into smaller and smaller pieces; once a piece is small enough to fit into an array, a more efficient sorting algorithm can be applied to it.
Quicksort

- Similar to Mergesort in its use of divide and conquer paradigm
- In Mergesort, the “divide” part was trivial; the array was simply halved. The complicated part was the merge
- In Quicksort, the opposite is true; combining the sorted pieces is trivial, but Quicksort uses a sophisticated division algorithm
Quicksort

• Select an element that belongs in the middle of the array (called the pivot), and place it at the midpoint

• Place all values less than the pivot before the pivot, and all elements greater than the pivot after the pivot
Quicksort

• At this point, the array is not sorted, but it is closer to being sorted -- the pivot is in the correct position, and all other elements are in the correct array segment

• Because the two segments are placed correctly with respect to the pivot, we know that no element needs to be moved from one to the other
Quicksort

• Since we now have the two segments to sort, can perform recursive calls on these segments

• Choice of pivot element important to efficiency of algorithm; unfortunately, there is no obvious way to do this

• For now, we will arbitrarily pick a value to use as pivot
public static void quicksort (int data[], int first, int n)
{
    int pivotIndex; // array index of pivot
    int n1, n2;     // number of elements before & after pivot
    if (n > 1)
    {
        pivotIndex = partition (data, first, n);
        n1 = pivotIndex - first;
        n2 = n - n1 - 1;
        quicksort (data, first, n1);
        quicksort ((data, pivotIndex+1, n2);
    }
}
Partition method

• Moves all values <= pivot toward first half of array
• Moves all values > pivot toward second half of array
• Pivot element belongs at boundary of the two resulting array segments
Partition method

• Starting at beginning of array, algorithm skips over elements smaller than pivot until it encounters element larger than pivot; this index is saved (too_big_index)

• Starting at the end of the array, the algorithm similarly seeks an element on this side that is smaller than pivot -- the index is saved (too_small_index)
Partition method

• Swapping these two elements will place them in the correct array segments

• The technique of locating and swapping incorrectly placed elements at the two ends of the array continues until the two segments meet

• This point is reached when the two indexes (too_big and too_small) cross -- in other words, too_small_index < too_big_index
Partition method: pseudocode

- Initialize values for pivot, indexes
- Repeat the following until the indexes cross:
  
  ```
  while (too_big_index < n && data[too_big_index] <= pivot)
      too_big_index ++
  
  while (data[too_small_index] > pivot)
      too_small_index --
  
  if (too_big_index < too_small_index)
      // both segments still have room to grow
      swap (data[too_big_index], data[too_small_index])
  ```
Partition method: pseudocode

• Finally, move the pivot element to its correct position:
  
  \[
  \text{pivot\_index} = \text{too\_small\_index};
  \]
  
  \[
  \text{swap}\ (\text{data}[0], \text{data}[\text{pivot\_index}]);
  \]
Time analysis of Quicksort

- In the worst case, Quicksort is quadratic
- But Quicksort is $O(N \log N)$ in both the best and the average cases
- Thus, Quicksort compares well with Mergesort and has the advantage of not needing dynamic memory allocated
Choosing a good pivot element

• The worst case of Quicksort comes about if the element chosen for pivot is at either extreme - in other words, the smallest or largest element -- this will occur if the array is already sorted!

• To decrease the likelihood of this occurring, pick three values at random from the array, and choose the middle of the three as the pivot
Perform quicksort on the following set of numbers – show the array after each step

5, 29, 43, 17, 99, 56, 72, 38