Sorting

Insertion Sort, Selection Sort, Quick Sort, Bubble Sort, Heapsort
Objectives

• Examine several sorting algorithms
  * **Insertion Sort**
  * **Selection Sort**
  * **Quick Sort**
  * **Bubble Sort**
  * **Heapsort**

• Analyze the time complexity of these algorithms
Sorting Problem

• Suppose we have an unordered list of objects that we wish to have sorted into ascending order
• We will discuss the implementation of several sort methods with a header of the form:

```java
public void someSort(UnorderedList list) {
    // precondition: list holds a sequence of objects in some random order
    // postcondition: list contains the same objects, now sorted into ascending order
}
```
Insertion Sort

- *Insertion Sort* orders a sequence of values by repetitively inserting the next value into a *sorted subset* of the sequence.

- More specifically:
  - Consider the first item to be a *sorted subsequence* of length 1.
  - Insert the second item into the *sorted subsequence*, now of length 2.
  - Repeat the process, always inserting the *first* item from the *unsorted portion* into the *sorted subsequence*, until the entire sequence is in order.
Discussion

• Is there a best case?
  • Yes: the items are already sorted, but in reverse order (largest to smallest)
  • What is the time complexity then?

• What is the worst case?
  • The items are already sorted, in the correct order!!
  • Why is this the worst case?
Insertion Sort

**In-Place:** the algorithm does not use auxiliary data structures.
Insertion Sort
Insertion Sort

sorted

5 8 2 6 9 4 6

2
Insertion Sort

sorted

5 \rightarrow 5 \rightarrow 8 \rightarrow 6 \rightarrow 9 \rightarrow 4 \rightarrow 6

2
Insertion Sort
Insertion Sort
Insertion Sort

sorted

2 5 6 8 9 4 6 4
Insertion Sort
Insertion Sort

sorted

2 4 5 6 6 → 8 → 9
**Algorithm** insertionSort (A,n)

**In:** Array A storing n values

**Out:** {Sort A in increasing order}

```plaintext
for i = 1 to n-1 do {
    // Insert A[i] in the sorted sub-array A[0..i-1]
    temp = A[i]
    j = i - 1
    while (j >= 0) and (A[j] > temp) do {
        j = j - 1
    }
    A[j+1] = temp
}
```
Selection Sort

- **Selection Sort** orders a sequence of values by repetitively putting a particular value into its *final* position.
- More specifically:
  - Find the *smallest value* in the sequence.
  - Switch it with the value in the *first position*.
  - Find the *next smallest value* in the sequence.
  - Switch it with the value in the *second position*.
  - Repeat until all values are in their proper places.
Selection Sort is an $O(n^2)$ algorithm
The analysis is similar to that of Insertion Sort. We will leave it as an exercise for you to analyze this algorithm.
Discussion

• Is there a best case?
  • No, we have to step through the entire remainder of the list looking for the next smallest item, no matter what the ordering

• Is there a worst case?
  • No
SelectionSort

Selection sort without using any additional data structures. Assume that the values to sort are stored in an array.

8 5 2 6 9 4 6
SelectionSort

First find the smallest value

smallest value
SelectionSort

Swap it with the element in the first position of the array.

smallest value

swap
SelectionSort

Swap it with the element in the first position of the array.
SelectionSort

sorted

2 5 8 6 9 4 6
SelectionSort

Now consider the rest of the array and again find the smallest value.
SelectionSort

Swap it with the element in the second position of the array, and so on.

sorted

swap

smallest value
SelectionSort

sorted

2  4  8  6  9  5  6
SelectionSort

sorted

smallest value
SelectionSort

sorted

swap

smallest value

2 4 8 6 9 5 6
SelectionSort

sorted

2 4 5 6 9 8 6
SelectionSort

sorted

2 4 5 6 6 8 9

smallest value
SelectionSort

2 4 5 6 6 8 9

sorted
Algorithm selectionSort (A,n)

In: Array A storing n values
Out: {Sort A in increasing order}

for i = 0 to n-2 do {
    // Find the smallest value in unsorted subarray A[i..n-1]
    smallest = i
    for j = i + 1 to n - 1 do {
            smallest = j
    }
    // Swap A[smallest] and A[i]
    temp = A[smallest]
    A[i] = temp
}
Quick Sort

- **Quick Sort** orders a sequence of values by *partitioning* the list around one element (called the *pivot* or *partition element*), then sorting each partition.

- More specifically:
  - Choose one element in the sequence to be the *pivot*
  - Organize the remaining elements into three groups (*partitions*): those *greater than* the pivot, those *less than* the pivot, and those *equal* to the pivot.
  - Then sort each of the first two partitions (recursively).
Quick Sort

*Partition element* or *pivot*:

- The choice of the *pivot* is arbitrary
- For efficiency, it would be nice if the pivot divided the sequence roughly in half
  - However, the algorithm will work in any case
Quick Sort

Approach to the problem:

• We put all the items to be sorted into a container (e.g. an array)
• We choose the pivot (partition element) as the first element from the container
• We use a container *smaller* to hold the items that are smaller than the pivot, a container *larger* to hold the items that are larger than the pivot, and a container *equal* to hold the items of the same value as the pivot
• We then *recursively* sort the items in the containers *smaller* and *larger*
• Finally, copy the elements from *smaller* back to the original container, followed by the elements from *equal*, and finally the ones from *larger*
Quicksort Algorithm

Numbers on yellow background give the order of recursive calls
Analysis of Quick Sort

• We will look at two cases for Quick Sort:
  
  • **Worst case**
    
    • When the pivot element is the *largest* or *smallest* item in the container (why is this the worst case?)
  
  • **Best case**
    
    • When the pivot element is the *middle* item (why is this the best case?)
Best Case Analysis

- The best case occurs when the pivot element is chosen so that the two new containers are as close as possible to having the same size.
- It is beyond the scope of this course to do the analysis, but it turns out that the best case time complexity for Quick Sort is $O(n \log_2 n)$.
- And it turns out that the average time complexity for Quick Sort is the same.
Bubble sort (for int's)

\[x = n - 1; \quad \text{sorted} = \text{false};\]

\[\text{while}(!\text{sorted})\{\]
    \[\text{sorted} = \text{true};\]
    \[x = x - 1;\]
    \[\text{for}(\text{int } y = 0; \quad y < x; \quad y++)\{\]
        \[\text{if}(\text{array}[y] > \text{array}[y + 1]) \{\]
            \[\text{swap(} \text{array}[y], \text{array}[y + 1])\]
            \[\text{sorted} = \text{false};\]
        \[\}\]
\[\}\]
Example

20 15 17 14 6 9 4 1 8
15 17 14 6 9 4 1 8 20
15 14 6 9 4 1 8 17 20
14 6 9 4 1 8 15 17 20
6 9 4 1 8 14 15 17 20
6 4 1 8 9 14 15 17 20
4 1 6 8 9 14 15 17 20
1 4 6 8 9 14 15 17 20
1 4 6 8 9 14 15 17 20
Cost analysis

- **Worst case**
  - n times through the main loop
  - \[ n + (n-1) + (n-2) + \ldots = O(n^2) \]
    comparisons and swaps

- **Best case**
  - the array is already sorted
  - \[ O(n) \]
    comparisons and swaps
Heap sort

- A heap is an almost complete binary tree:
- The tree is completely filled on all levels except possibly lowest, where it is filled in left to right order.
- The height of a binary tree with n nodes in it is $O(\log_2(n))$. 
Example

The heap with 9 nodes
The heap property

- When storing comparable's in a heap, ensure that:
  - the value of the element at a node is greater than, or equal to, the value of the elements at its two children
  - the value at a node is less than, or equal to, the value at its parent.
Building a heap

Add node in the next available space in the almost complete binary tree and "bubble" that node up to restore the heap property for the tree. Since this only involves operations from a leaf node along the path to the root node, there are at most $\log_2(n)$ operations required.
Example

Insert the values 9, 14, 10, 6, 17
Example

Insert the values 9, 14, 10, 6, 17
Example

Insert the values 9, 14, 10, 6, 17
Removing the largest element

- Swap the root and last node in the heap in array
- The heap is no longer a heap: bubble the new root down the heap:
  - If not a heap, swap it with the largest of its children
  - continue along the largest child node until you get a heap (will happen for sure if you get to a leaf node)
Black numbers are unsorted, Red numbers are sorted

Heap

Bubble Down

Heap

Bubble Down

Bubble Down
Heap

6

10

17

14


Bubble Down

6

9

10

14

17


Sorted Heap

6

9

10

14

17
Cost

for a heap with n elements:
  the height of the tree is $O(\log(n))$
  insert and remove move an element on a path from the root to a leaf
so both take $O(\log(n))$ operations
Data representation

The most common implementation uses an array of ...

For a heap with n element, indices range from 0 to n-1

Easy access to parent and children:

left child(i) = 2i+1
right child(i) = 2i+2
parent(i) = (i-1)/2
Heap sort

Simple algorithm:
create an empty heap
insert all elements
Repeat
  remove the largest
until the heap is empty

Cost for n elements: $O(n \log(n))$
Summary

• *Insertion Sort* is $O(n^2)$
• *Selection Sort* is $O(n^2)$
• *Quick Sort* is (in the average case) $O(n\log_2 n)$
• *Bubble Sort* is $O(n^2)$
• *Heapsort* is $O(n\log_2 n)$
• Which one would you choose?