CS 162
Intro to Programming II

Sorting I
Sorting

• A sorting algorithms rearranges the elements so that they are in ascending or descending order
• Many algorithms are similar to how we would sort a pile of papers
• Every item must be compared at least once
• Our goal is to minimize the number of operations- as efficiently as possible
Selection Sort

• Find the smallest element of the unsorted region
• Swap the smallest element with the first item of the unsorted region
Selection Sort

23 16 8 15 42 4

4 16 8 15 42 23

4 8 16 15 42 23

4 8 15 16 42 23

4 8 15 16 42 23

4 8 15 16 23 42
void selectSort(int a[], int size) {
    for( int i = 0; i < size-1; i++ ) {
        int minPos = minimumPosition(a, size, i);
        swap(a, minPos, i);
    }
}

int minimumPosition(int a[], int size, int from) {
    int minPos = from;
    for( int i = from + 1; i < size; i++ )
        if( a[i] < a[minPos] ) minPos = i;
    return minPos;
}

void swap(int a[], int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
Complexity

• We’ll count the number of operations selection sort takes
• Simplification: count how often an array element is visited
  – proportional to the number of operations
• Let $n = \# \text{ of elements in array}$
Complexity

[i=0] minimumPosition: visits n elements
  swap: visits 2 elements
[i=1] minimumPosition: visits n-1 elements
  swap: visits 2 elements
[i=2] minimumPosition: visits n-2 elements
  swap: visits 2 elements

...:
[i=n-2] minimumPosition: visits 2 elements
  swap: visits 2 elements
Complexity

Sum the numbers on the previous slide

\[
n + 2 + (n - 1) + 2 + \cdots + 2 + 2 \\
n + (n-1) + \cdots + 2 + (n-1) \times 2 \\
2 + \cdots + (n-1) + n + (n-1) \times 2 \\
\frac{n(n+1)}{2} - 1 + (n-1) \times 2 \\
\frac{1}{2}n + \frac{5}{2}n^2 - 3 = O(n^2) \\
\]

Note that \( 2 + \ldots + (n-1) = \frac{n(n+1)}{2} - 1 \)
Complexity of Selection Sort

Best Case- $O(n^2)$

Worst Case- $O(n^2)$

Average Case- $O(n^2)$
Insertion Sort

• Assume the initial sequence $a[0]$ $a[1]$ ... $a[k]$ is already sorted
• $k = 0$ when the algorithm starts
• Insert the array element at $a[k+1]$ into the proper location of the initial sequence
• Note that the insertion enlarges the initial (i.e. sorted) sequence
What

\[ \begin{array}{cccccc}
23 & 16 & 8 & 15 & 42 & 4 \\
\end{array} \]

\[ k(0) \]

\[ \begin{array}{cccccc}
16 & 23 & 8 & 15 & 42 & 4 \\
\end{array} \]

\[ k(1) \]
What

23 16 8 15 42 4

16 23 8 15 42 4

8 16 23 15 42 4

8 15 16 23 42 4

8 15 16 23 42 4

4 8 15 16 23 42
void insertSort(int a[], int size) {
    for( int i = 1; i < size; i++ ){
        int next = a[i];
        // Find the insertion location
        // Move all larger elements up
        int j = i;
        while( j > 0 && a[j-1] > next) {
            a[j] = a[j-1];
            j--;
        }
        // Insert the element
        a[j] = next;
    }
}
Complexity

• for loop runs for \((n-1)\) iterations
• On the \(k\)th iteration:
  – We have \(k\) elements that are already sorted
  – Need to insert into these \(k\) elements and move up the elements that are past the insertion point => \(k + 1\) elements visited
• Total # of visits is:

\[
2 + 3 + \ldots + n = \frac{n(n+1)}{2} = O(n^2)
\]
Complexity of Insertion Sort

Best Case-

already in order \( O(n) \)

Worst Case-

reversed \( O(n^2) \)

Average Case-

\( O(n^2) \)
Bubble Sort

• Compare adjacent elements. If the first is greater than the second, swap them.
• Do this for each pair of adjacent elements, starting with the first two and ending with the last two. At this point the last element should be the greatest.
• Repeat the steps for all elements except the last one.
• Keep repeating for one fewer element each time, until you have no more pairs to compare.
Bubble Sort

23 16 8 15 42 4

16 23 8 15 42 4

16 8 23 15 42 4

16 8 15 23 42 4

16 8 15 23 4 42
Bubble Sort

• The largest element is in the last element of the array. The rest of the array is still unsorted. We do it again but stop at n-1.
void bubbleSort(int a[], int size) {
    for (int i = (size-1); i >= 0; i--) {
        for (int j = 1; j<=i; j++){
            if (a[j-1] > a[j]){
                // Swap elements at j-1 and j
                int temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
Complexity

Best Case-
  already in order \( O(n^2) \)

Worst Case-
  reversed \( O(n^2) \)

Average Case-
  \( O(n^2) \)

There is a version with best case \( O(n) \)