

A decorative graphic on the right side of the page. It features three blue circles of varying sizes, each composed of concentric circles in different shades of blue. Two thin blue lines intersect at the top left and extend diagonally across the page, framing the circles.

Sorting

C++ Notes

This document contains descriptions of the algorithms, graphic representation, and sample code for both selection and insertion sort.

Why Sort?

Sometimes you have a list (array or vector) of information that you would like to put into order (either ascending or descending). This may be a list of numeric values. This may be a list of strings that you would like to sort lexicographically (remember strings don't quite follow the same alphabetic rules as the dictionary). For example, perhaps you want to alphabetize a list of students by last name. You might also want to order a list of students according to their GPA, so you can determine the top 10% of the graduating class. Learning how to implement a sorting algorithm (method of solving a problem) can help you do this in code.

Selection Sort

The Algorithm:

- Start at the beginning of the vector.
- Make a pass through the vector, looking for the smallest item.
 - Each time you visit a new item, if it is smaller than the smallest you've seen so far, designate it as the new smallest.
 - When you get to the end of the vector, swap the smallest item with the item at the start of the current sweep.
 - Move the start of the sweep one position ahead.
- Repeat passes through the vector until you have reached the last element in the vector as a starting point.
- Now the vector is sorted.



Figure 1. Selection Sort - Red represents the current starting position. Blue represents the smallest so far by end of pass (except where start is also smallest, then it is just shown in red). Only the state of the elements right before the swap (see code below) are depicted in this picture.

Example Selection Sort Function:

This function takes as an argument (by reference) a vector of strings.

```
void selectionSort(vector<string> &words)
{
    int smallestIndex;           //holds index of smallest item I've seen so far
    string smallest;            //holds "smallest" word I've seen so far
    string temp;                //used during swap so we don't lose data
                                //outer sweeps thru vector
    for(int i=0; i<words.size(); i++)
    {
        //assume the start position is smallest we will see
        smallestIndex=i;
        smallest=words[i];
        //see if we can do better further along in the vector
        for(int j=i; j<words.size(); j++)
        {
            //if we find a smaller word, save it's position and value
            if(words[j]<smallest)
            {
                smallest=words[j];
                smallestIndex=j;
            }
        }
        //swap smallest with current start position before we begin the next sweep
        temp=words[i];
        words[i]=smallest;
        words[smallestIndex]=temp;
    }
}
```

Following is an example of creation of a vector, allowing the user to enter words, and sorting the vector with our function before outputting the result:

```
vector<string>words(10);  
  
for(int i=0; i<words.size(); i++)  
{  
    cout<<"enter word";  
    cin>>words[i];  
}  
  
selectionSort(words);  
  
for(int i=0; i<words.size(); i++)  
{  
    cout<<words[i]<<endl;  
}
```

Weakness:

The algorithm goes through its iterations to examine every cell, even in the case where the vector is already in order.

Insertion Sort

The Algorithm:

- Start at the beginning of the vector.
- Pick up the current item.
 - Go backwards in the vector towards index 0 until you find the insertion point.
 - As you visit each position, if it is still larger than the current item, assign it to the next index.
 - For example: if you examine the element at index 2 and find it larger than your item, move the element from index 2 to index 3.
 - The insertion point will be either:
 - The point at which the next lowest index contains a lower element.
 - The point at which you have reached the 0 index of the vector.
- Repeat passes through the vector until you have reached the last element in the vector as a starting point.
- Now the vector is sorted.



Figure 2. Insertion Sort - Red represents the current starting position.

Example Insertion Sort Function:

This function takes as an argument (by reference) a vector of strings.

```
void insertionSort(vector<string> &words)
{
    //value of element removed for insertion
    string removedItem;
    //keep track of whether insertion pt found
    bool pointFound;
    //keep track of current position
    int j;

    //iterate over elements to remove and reinsert
    for(int i=1; i<words.size(); i++)
    {
        //set removed Value
        removedItem=words[i];
        //set found to false
        pointFound=false;
        //set starting j for this sweep
        j=i-1;

        //go backwards thru list looking for a better place
        while(!pointFound && j>=0)
        {
            //if next cell up is still bigger...
            if(words[j]>removedItem)
            {
                //shift it down
                words[j+1]=words[j];
                j--;
            }
            else
            {
                //we found the insertion point
                //break the loop
                pointFound=true;
            }

            //insert removed item
            words[j+1]=removedItem;
        }
    }
}
```

The call to the function would be identical to that of selection sort, except using this new function name.

Compared to Selection Sort:

Slightly more efficient (time-wise) because there are potentially fewer comparisons. This is due to the fact that the inner loop breaks once the insertion point is found. In a situation where the entire vector is already sorted, the vector will be swept through once, and will be markedly more efficient than selection sort (which will still process each individual position with nested loops).