2: List Processing

So we've programmed the easylist which used an array for its implementation. Now we have the background to discuss some of the common manipulations that are used on list and how efficient they are. Recall that the whole reason behind data structures is to arrange data in such a way that it can be manipulated efficiently.

What exactly is a list anyway? It is a set of values of the same type, and as we've seen, arrays are perfect for storing such kinds of data. As you may have noticed before, many data structures in computer science are based on methods of arrangement that occur in real life, and lists are not an exception. If you ever get confused, just visualize a physical list e.g. a shopping list like the one you'd use to buy groceries. Now we know how it looks like, what kind of operations/manipulations do we typically do with a list?

- Insert an item in the list
- delete an item from the list
- search the list
- sort the list

Insertion

This is a very basic operation for a list: We just insert an item at the end of the list. This operation doesn't depend on how many items (or pieces of data) are already in the list. Therefore its complexity in the big-O notation is *O(1)*.

Deletion

Deleting an item from a list is slightly more involved. We first have to search for the item that we have to delete, delete it, and then shift the remaining elements to cover up the space we just freed up.

Its complexity is *O(n)* because the process grows linearly with the size of data that you are dealing with. On average, we'll compare $\frac{n}{2}$ items, and shift $\frac{n}{2}$ items to the left.

Searching

To search through our list, we need three things: the list itself, its length, and the item for which we are going to search.

Typical search algorithms will return a -1 if the value for which we are searching is not found. If the value is found, it will return the index of the item.

There are different kinds of searching algorithms that you can employ on your list. The most basic is the **linear** or **sequential search.** It starts at the beginning of the array and searches until either we find the item, or we reach the end of the array.

Sequential/Linear Search

```
int LinearSearch(int [] list, int item)
{
      for(int i=0; i<list.length; i++)
      {
            if(list[i] == item)
                  return i;
      }
      return -1;
}
```

So what about the performance of this search? Imagine our list has 1000 items. If the item we are searching for is near the front, the search will be fast, but if its near the end of the list, the search will be long.

Its complexity is O(n). On average, we'll compare $\frac{n}{2}$ items.

So how can it be improved? Well if the list was ordered, then we wouldn't have to search all the way to the end to figure out that the key we were searching for wasn't in the list. We would know that a key isn't in the list as soon as we found a value larger (or smaller...depending on how your list is ordered) than the key we were looking for.

Ordered Sequential/Linear Search

```
int OrderedLinearSearch(int [] list, int item)
{
      for(int i=0; i<list.length; i++)
      {
            if(list[i] == item)
                  return i;
            if(list[i] > item)
                  return -1;
      }
      return -1;
}
```

If the list was ordered, we could also employ a binary search which is a slightly more complicated algorithm but would perform much faster.

Binary Search

```
int BinarySearch(int [] list, int item)
{
      int first = 0;
      int last = list.length -1;
      int mid;

      while(first <= last)
      {
            mid = (first + last) / 2;

            if(list[mid] == item)
                  return mid;
            if(list[mid] > item)
                  last = mid - 1;
```

```
            else
                  first = mid + 1;
      }
      return -1;
}
```

What is the complexity for a binary search? *O(log n)*

Just so you remember, the division to find the mid is integer division. For example, if the list had *100* elements (indices *0* to *99*), then mid would evaluate to $\frac{99}{2} = 49$.

Sorting

Another common operation for lists is sorting them. Why? Did you see how much faster a binary search is than a sequential search? If you are going to be doing a lot of searching on your list, it makes sense to sort them at the beginning so that you can save on time when it comes to searching. Alternatively, if you are going to be doing very little searching in your list, then maybe sorting would take up too much time.

So how do we sort? Its very easy for us as humans to do with small data sizes because we can see multiple pieces of data at the same time, do loads of comparisons in a short time, etc. But what about a computer? Is there a way that we can design an algorithm to sort the data in a sequence of steps that will always work regardless of how the data looks in the first place?

```
Let's see if you can sort the provided playing cards in non-
descending order. Now can you figure out a sequence of steps to
describe the process you just went through? A sequence that will
always work?
```

Bubble Sort.

Bubble Sort

```
void BubbleSort(int [] list)
{
      for(int i=1; i<list.length; i++)
      {
            for(int j=1; j<list.length; j++)
            {
                  if(list[j] < list[j-1])
                  {
                        int temp = list[j];
                        list[j] = list[j-1];
                        list[j-1] = temp;
                  }
            }
      }
}
```

Let's assume that we want to sort the list in increasing order. Bubble sort works by making successive swaps in such a way as to move the largest element to the end of the array. That way the larger values "bubble" to the end of the array.

The inner loop affects a single **pass** through the list, and controls how many comparisons are carried out in each pass.

The outer loop controls the number of **passes** through the array.

To sort a list of *n* elements, this algorithm will take *n-1* passes.

In each pass, it will make between *n-1* and *1* comparisons. This totals up to $\frac{n(n-1)}{2}$ comparisons. On average, only half of those comparisons will result in a swap, i.e. $\frac{n(n-1)}{4}$ swaps. This means that for a list of *1000* elements, bubble sort would require about *500,000* comparisons, and *250,000* swaps on average. This represents a complexity of $O(n^2)$.

Pass	Comparisons	List	Indices compared
		9 5 7 1 3	
1	4	5 7 1 3 9	0/1, 1/2, 2/3, 3/4
2	3	5 1 3 7 9	0/1, 1/2, 2/3
3	2	1 3 5 7 9	0/1, 1/2
4	1	1 3 5 7 9	0/1

One of the weaknesses with the bubble sort is that even if the input is already sorted, it will still go through all the passes before ending. Fortunately, there is a way to make it optimized.

Optimized Bubble Sort

```
void OptimizedBubbleSort(int list[])
{
      bool swapMade;

      for(int i=1; i<LIST_LENGTH; i++)
      {
            swapMade = false;

            for(int j=1; j<LIST_LENGTH; j++)
            {
                  if(list[j] < list[j-1])
                  {
                        int temp = list[j];
                        list[j] = list[j-1];
                        list[j-1] = temp;
                        swapMade = true;
                  }
            }
```

```
            if(!swapMade)
                  break;
      }
}
```

Selection Sort

Selection sort works by identifying the smallest element in the list, and placing it in the first position (using a single swap). It then starts at the second position and finds the next-smallest item and places it in the second position (also using a single swap). It does this process repeatedly: finding the smallest item in the unsorted part of the list, and placing it in its proper position in the sorted part of the list. The left side of the list is the sorted part, and the right hand side is the unsorted part.

```
Selection Sort

void SelectionSort(int list[])
{
      for(int i=0; i<LIST_LENGTH-1; i++)
      {
            int minIndex = i;

            for(int j=i+1; j<LIST_LENGTH; j++)
                  if(list[j] < list[minIndex])
                        minIndex = j;

            int temp = list[i];
            list[i] = list[minIndex];
            list[minIndex] = temp;
      }
}
```

On average, for a list of size *n*, it still requires $\frac{n(n-1)}{2}$ comparisons, but only $(n-1)$ swaps which is considerably less than the bubble sort that we looked at earlier. For example, a list of *1000* elements would require approximately *500,000* key comparisons, but only *1000* swaps.

Its complexity is still $O(n^2)$.

Its useful with small amounts of data, in particular when swapping is time consuming.

Pass	Comparisons	List
		9 5 7 1 3
1	4	1 5 7 9 3
2	3	1 3 7 9 5
3	2	1 3 5 9 7
4	1	1 3 5 7 9

Insertion Sort

Insertion sort is the kind of sorting that we typically use as humans. We assume that the first part of the list is sorted, and the latter part is unsorted. We then move one item at a time from the unsorted part to the sorted part making sore to put it in its right position in order to maintain the "sortedness" of the sorted part.

Insertion Sort

```
void InsertionSort(int list[])
{
      for(int i=1; i<LIST_LENGTH; i++)
      {
            if(list[i] < list[i-1])
            {
                  int j, temp = list[i];

                  for(j=i-1; j>=0 && list[j]>temp; j--)
                        list[j+1] = list[j];

                  list[j+1] = temp;
            }
      }
}
```

On average, for a list of size n, it takes $\frac{n(n-1)}{2}$ comparisons, and $\frac{n(n-1)}{4}$ swaps which amounts to a complexity of $O(n^2)$. However, with an almost sorted list, it takes almost $O(n)$. This means that it is the best algorithm to use for an almost sorted list.

Pass	List
	9 5 7 1 3
1	5 9 7 1 3
2	5 7 9 1 3
3	1 5 7 9 3
4	1 3 5 7 9

Recap

- For each sort, sorting *n* elements takes *n-1* passes.
- Each sort maintains a sorted and unsorted side.

- Bubble: unsorted = left, sorted = right
- Select: unsorted = right, sorted = left
- Insertion: unsorted = right, sorted = left

- A single element is trivially sorted.
- To differentiate between the different kinds of sorts
  - Bubble: on each pass, largest item in unsorted side "bubbles" to end of unsorted side. Requires many swaps, and many comparisons.
  - Select: on each pass, smallest item in unsorted side moves to beginning of unsorted side. Requires few swaps, but many comparisons.
  - Insertion: on each pass, first item in unsorted side moves to its proper place in sorted side. Some swaps, some comparisons.
- Sequential search on an ordered list can abort search early if current element is greater than the key we are searching for. On average, it takes $\frac{n}{2}$ key comparisons but is a bit better than sequential search on an un-ordered list. On the down side, insertion into an ordered list is more costly. Complexity of a sequential search is *O(n)*.
- Binary search can only be used when the list is sorted. It uses a divide and conquer technique. It is usually applied to array based lists since its easy to find the middle element ($\frac{first+last}{2}$). If the middle element is what we are searching for, then we're done. If its less than what we are searching for, we search in the list to the right, otherwise, we search in the list to the left. We keep on repeating this process till we find the key (or don't find it). Complexity of binary search is *O(lg n).*