Lecture 8.1

Sorting and Searching

Sorting with Bubble Sort

- Compares two values next to each other and exchanges them if necessary to put them in the right order.
- Many variations on the order in which the pairs are examined.

Algorithm Analysis

- The BIG – O - Notation
- Some algorithms may take very little computer time to compute while others may take a considerable amount of time
- Example

```
c= 0;
sum = 0;
cin>>num;
while (num !=-1)
{sum+=num;c++;cin>>num;}
average=sum/count;
cout<<“Average is”<<average;
```


The BIG – O - Notation

- The algorithm
  - has 3 operations before the while loop
  - 4 operations within the while loop
  - 2 operations after the while loop
- In total (if loop executes 10 times)
  - 10*4 + 3+2
- In total (if loop executes 100 times)
  - 100*4 + 3+2
- We can generalize it to 4n+5 (n = loops)
- For large values of n, the term 4n becomes the dominating term

The BIG – O - Notation

- Growth Rate of various functions
- Function grows as n (problem size grows)

n	$\log_2 n$	$n\log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536

Growth Rate

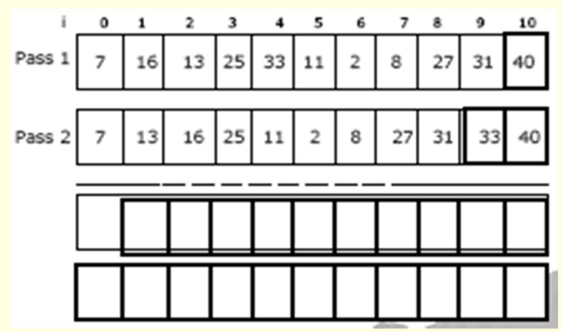
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n \log_2 n$	$f(n)=n^2$	$f(n)=2^n$
10	0.01µs	0.03µs	0.033µs	0.1µs	1µs

- Time for f(n) instructions on a computer that executes 1 billion instructions per second
- Definition – we say that f(n) is **Big-O** of g(n) written f(n) = O(g(n)) if there exists positive constants c and n_o such that f(n) <=cg(n) for all $n>=n_o$

Bubble Sort facts

- Efficient aspect of bubble sorts,
  - can quit early if the elements are almost sorted.
- Bubble sorts are $O(N^2)$ on the average,
- Can have an O(N) best case.

$O(N^2)$



BS-fixed number of passes

```
void bubbleSort1 ( int x [ ] , int n )
{
   for (int pass=1; pass<n; pass++){ //count
                                    //how many times
                                    //This next loop becomes
                                    //shorter and shorter
        for ( int i=0; i < n - pass; i++ ) {
                if ( x [ i ] > x [ i+1 ] ) {
                        //exchange elements
                        int temp = x [ i ];
                        x [ i ] = x [ i+1 ];
                        x [ i+1 ] = temp;}}}
}
```


BS-fixed number of passes facts

- Fixed number of passes =length of the array - 1
- Each inner loop is one shorter than the previous one
- Disadvantage:
  - Always makes n-1 passes over the array,
  - Can't stop early if the array is already sorted.

BS-stop when no exchanges

```
void bubbleSort2 ( int x [ ] , int n ) {
   bool exchanges;
   int temp;
   do {
        exchanges = false; // assume no exchanges
        for ( int i=0; i < n-1; i++ ) {
                if ( x [ i ] > x [ i+1 ] ) {
                temp = x[ i ];
                x [ i ] = x[ i+1 ];
                x [ i+1 ] = temp;
                exchanges = true; }}       // after
                                           //exchange,
                                           // must look again
   }while (exchanges);
```


BS—stop when no exchanges facts

- Continues making passes over the array as long as there were any exchanges.
- If the array already sorted, sort will stop after only one pass.
- Disadvantage:
  - Doesn't shorten the range each time by 1 as it could.

BS-stop when no exchanges, shorter each time

```
void bubbleSort3 ( int x [ ] , int n ) {
   bool exchanges;
   int temp;
   do {
        n--; //make loop smaller each time
        exchanges = false; // assume this is last pass over array
        for ( int i=0; i < n; i++ ) {
                if (x [ i ] > x [ i+1 ]) {
                temp = x[ i ];
                x [ i ] = x [ i+1 ];
                x [ i+1 ] = temp;
                exchanges = true; // after exchange must look again
        } } }
   while (exchanges); }
```


Linear Search

- Straightforward loop comparing every element in the array with the *key (*the item we are looking for*)*.
- As soon as an equal value is found, it returns
- If loop finishes without finding a match, a -1 is returned.
- Good solution for small arrays

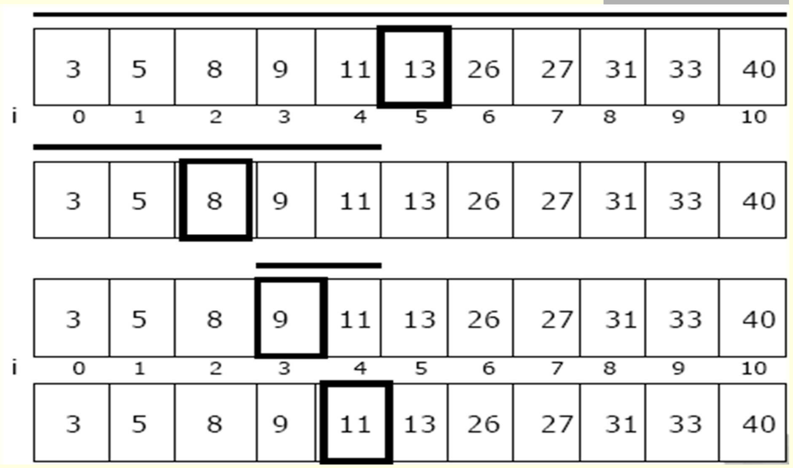
Linear Search code

```
int linearSearch ( int a [ ] , int first, int last, int key)
{
   for ( int i = first; i <= last; i++) {
        if ( key == a [ i ] )
                return i;
   }
   return -1; // failed to find key
}
```


Binary Search

- Fastest way to search a sorted array.
- Look at the element in the middle.
  - If the key is equal to that, search is finished.
  - If the key is less than the middle element, do a binary search on first half.
  - If it's greater, do a binary search of the second half.

Binary Search (cont)



Binary search code

```
int binarySearch ( int sortedArray [ ] , int first, int last, int key ) {
   while ( first <= last ) {
        int mid =( first + last ) / 2; //compute mid point.
        if ( key > sortedArray [ mid ] )
                first = mid + 1; // repeat search in top half.
        else if ( key < sortedArray [ mid ] )
                last = mid - 1; // repeat search in bottom half.
        else return mid; // found it. return position
   }
   return - ( first + 1 ); // failed to find key
}
```