Lecture 8.2

Recursive sorting and searching

Binary Search

	3	5	8	9	11	**13**	26	27	31	33	40
i	0	1	2	3	4	5	6	7	8	9	10

3	5	**8**	9	11	13	26	27	31	33	40

	3	5	8	**9**	11	13	26	27	31	33	40
i	0	1	2	3	4	5	6	7	8	9	10

3	5	8	9	**11**	13	26	27	31	33	40

Recursive Binary Search

```
int binarySearch (int array [ ] , int first, int last, int key)
{
   int middle;

   // base case
   if ( first > last)
        return - 1; // key was not found
```


Recursive Binary Search (cont)

```
else {
   middle = ( first + last) / 2;
   if ( key == array [ middle ] )
        return middle;
   else if ( key < array [middle ] )
        return binarySearch ( array , first, middle - 1, key );
   else
        return binarySearch ( array, middle + 1, last, key);
}
```

Quicksort

- One of the fastest and simplest sorting algorithms. It works recursively by a divide-and-conquer strategy.
- Divide the array of items to be sorted into two partitions and then call the quicksort procedure recursively to sort the two partitions,
- *ie* we *divide* the problem into two smaller ones and *conquer* by solving the smaller ones.

Quicksort Algorithm

1. pick one element in the array, which will be the *pivot*.
2. make one pass through the array, called a *partition step*, re-arranging the entries so that:
   - entries smaller than the pivot are to the left of the pivot.
   - entries larger than the pivot are to its right.
   - the pivot is in its proper place.
3. recursively apply quicksort to the part of the array that is to the left of the pivot, and to the part on its right.

How Quicksort works

Step 1	7	36	28	14	11	$\boxed{13}$	45	3	21	33	56
	0	1	2	3	4	5	6	7	8	9	10 j
Step 2	7	36	28	14	11	$\boxed{13}$	45	3	21	33	56
i	0	1	2	3	4	5	6	7	8	9	10
	7	**3**	**13**	**11**	**14**	**28**	45	**36**	21	33	56

How Quicksort works

Step 3

				4	5	6	7	8	9	10
7	**3**	**13**	**11**	**14**	**28**	45	**36**	21	33	56
0	1	2	3							

Step 1 again

7	$\boxed{3}$	**13**	**11**
0	1	2	3

Quicksort Implementation

```
void quicksort ( int array [ ], int lo, int hi ) {
   // lo is the lower index,
   // hi is the upper index
   // of the region of array a that is to be
   // sorted
   int i = lo, j = hi, temp;
   //step 1, x will be the pivot
   int x = array [ ( lo + hi ) / 2 ];
```


Quicksort Implementation (cont 2)

```
// partition //step
do {
while (array [ i ] < x ) //check until we find an
   //element bigger than the pivot in the
   i++; // lower part of the array.
while (array [ j ] > x ) // check until we find an
   //element smaller than the pivot in the
   j- - ; // higher part of the array.
```


Quicksort Implementation (cont 3)

```
    if (i <= j)
    {
      temp = array [ i ]; //swap
      array [ i ] = array [ j ];
      array [ j ] = temp;
      i++;
      j- - ;
    }
} while (i <= j);
```


Quicksort Implementation (cont 4)

```
   // recursion // step 3
   if ( lo < j ) quicksort ( array , lo, j );
   if ( i < hi ) quicksort ( array , i, hi );
}
```

Quicksort Analysis

- The *best-case* behavior of the Quicksort algorithm occurs when in each recursion step the partitioning produces two parts of equal length.
- In order to sort n elements, in this case the running time is in O(nlog(n)).
- This is because the recursion depth is log (n) and on each level there are n elements to be treated.

Quicksort Analysis (cont)

- The worst case occurs when in each recursion step an unbalanced partitioning is produced.
- When one part consists of only one element and the other part consists of the rest of the elements .
- Then the recursion depth is n-1 and Quicksort runs in time (n^2).