The Binary Search and Quick Sort algorithms

In the following section, we give a brief look at 3 important and **efficient** algorithms and their complexity analysis.

Example: The binary search algorithm

Write an algorithm to find the index of a given key in an array a, the array is sorted in **ascending** order, and find the complexity of your algorithm.

input parameters

a: an array to search in (sorted in ascending order)

l: an integer index to the first element of the array

r: an integer index to the last element of the array

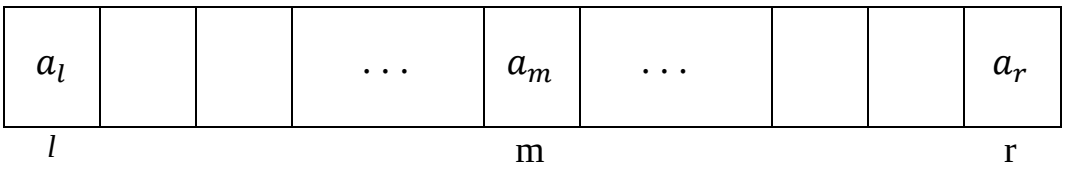
key: the element to search for

output through return

index: the location of the key in the array, or -1 if not found

Idea behind the binary search algorithm

We make use of the fact that the array is **sorted** by comparing the key with the element in the **middle** of the array: if they are equal we are done, else if the key < this element then we search in the **left half** of the array, else we search in the **right half** of the array. Binary search is one of the well-known **divide and conquer** algorithms.



```
int binarySearch ( a [ ],  l, r, key )
    if l <= r
        m = ( l + r ) / 2;
        if ( key = a_m )        return m;
        else if (key < a_m)          // search the left half of the array
                return binarySearch (a, l, m – 1, key);
        else            // key > a_m: search the right half of the array
                return binarySearch (a, m + 1, r, key);
    else
        return –1;              // key not found
```

Time complexity

Let $T(n)$ be the number of **comparisons** done to search an array of n elements in the **worst** case.

To simplify the math, we assume that $n \approx 2^k$.

It can be easily shown that $T(n)$ is governed by the following **recurrence relation**:

$$T(n) = 1 + T(\frac{n}{2}), T(1) = 1.$$

Its' solution is given by: $T(n) = 1 + \log n$.

This shows that the **binary search** algorithm is much more **efficient** than the **sequential** search algorithm whose complexity is O(n).

Example: The quick sort algorithm

Write an algorithm to sort an array a in **ascending** order, and find the complexity of your algorithm.

input parameters

a: an array to be sorted

l: an integer index to the first element of the array

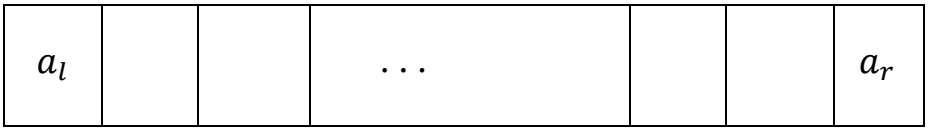
r: an integer index to the last element of the array

output parameter

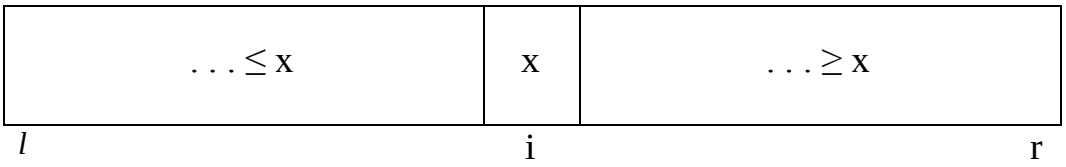
a: the array after sorting

Idea behind the quick sort algorithm

Quick sort applies **divide** and **conquer** by **dividing** the array into 2 sub arrays (hopefully of equal size) and then applies the same idea to the **left** and **right** sub arrays over and over until we have arrays of size 1, at which point we are finished.



To sort the array, we **partition** (**divide**) it so that **small** elements (compared to a pivotal element x) are moved to the **left** and **large** ones are moved to the **right** as shown next:



That is:

$$a_k \leq x, \ \forall \ k < i,$$

$$a_k \geq x, \ \forall \ k > \ i, \text{and}$$

$$a_i = x.$$

The element x is called the **pivot**. Now, to sort the array a, we proceed as follows:

```
void quick_sort ( a [ ], l, r )
    if l < r
        partition ( a, l, r, i );        // i is the index of the pivot
        quick_sort ( a, l, i-1 );        // sort the left sub array
        (quick_sort ( a, i+1, r);        // sort the right sub array
```

The **partition** algorithm, shown next, is the part which is doing the actual work of comparing and moving the array elements. Notice that the integer parameter i is a call by **reference** (output parameter).

```
void partition ( a [ ],  l, r, &i )
    i = l; j = r; x = a_i ;          // x is the pivot. Location a_i is now free
    while i < j
        while ( (a_j >= x) and ( j > i) ) j—; // looking for a small
                                              // element from the right
        if ( j > i )
            a_i = a_j;                        // a_j is now a free location
            i++;
        while ((a_i <= x) and (i < j) ) i++;  // looking for a large
                                              // element from the left
        if ( i < j )
            a_j = a_i;                        // a_i is now a free location
            j—;
    // now i=j, and a_i is free: put the pivot x into its' final location
     a_i = x;
```

This algorithm will do n-1 comparisons to **partition** an array of n elements.

Time complexity of the quick sort algorithm

Let $T(n)$ be the number of **comparisons** done to sort an array of n elements in the **best** case. The best case happens if we are so lucky that the partition algorithm will divide the array into 2 **equal** sub arrays.

To simplify the math, we assume that $n \approx 2^k$. It can be easily shown that $T(n)$ is governed by the following **recurrence relation**:

$$T(n) = n + 2T(\frac{n}{2}), T(1) = 0.$$

Its' solution is given by: $T(n) = n \log n$. It can also be shown that the **average** complexity is about: $1.4\, n \log n$. This shows that on the **average**, the **quick sort** algorithm is much more **efficient** than many other sorting algorithms such as the **selection** sort algorithm whose complexity is O(n^2).

Example: The merge algorithm

Write an algorithm to **merge** 2 **sorted** arrays a and b to form a new (**sorted**) array c, and find the complexity of your algorithm.

input parameters

a, b: 2 arrays, each of which is sorted in **increasing** order

na, nb: the number of elements in a and b

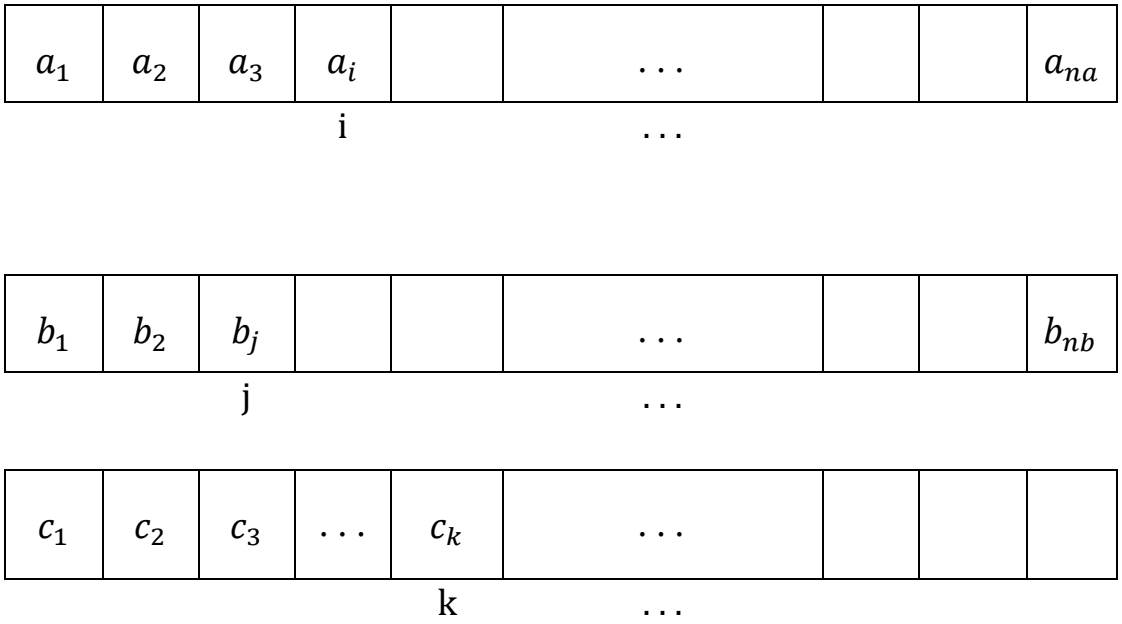
output parameters

c: an array, the result of the merge in **increasing** order

nc: the number of elements in c

Idea of the merge algorithm

In the general step, we compare a_i with b_j and copy the **smaller** element into c_k. If 2 elements are equal we copy one and only one of them to c_k.




```
void merge ( a [ ], b [ ], c [ ], na, nb,  & nc)
    i = 1; j = 1; k = 1;                // i, j, k are running indices through a, b, c
    while (i <= na and j <=nb)
        if (a[i] < b[j])
            c[k] = a[i];
            i ++;
            k ++;
        else if (b[j] < a[i])
            c[k] = b[j];
            j ++;
            k ++;
        else            // equal elements, we save one and only one of them
            c[k] = a[i];
            k++;
            i ++;
            j++;
    // end while
    // copy the remaining elements from and/or b, if any, into c
    for i = i to na
        c[k] = a[i];
        k++;
    for j = j to nb
        c[k] = b[j];
        k++;
    nc = k--;
    // end merge
```

Complexity of the merge algorithm

We find the number of **comparisons** done between the elements during the **merge** operation in the **worst** case. Each time we go through the **while** loop **one element** from a or b is copied into c depending on which is **smaller**. If 2 elements are **equal** one of them is copied into c. The worst case happens when there are no equal elements and the ends of the two arrays are reached at the same time inside the while loop. Hence the total number of comparisons = na+nb-1 = O(na+nb).