Searching and Sorting

Rossano Venturini[1]

1 **Computer Science Department, University of Pisa, Italy**
rossano.venturini@unipi.it

Notes for the course *"Competitive Programming and Contests"* at Department of Computer Science, University of Pisa.
Web page: `https://github.com/rossanoventurini/CompetitiveProgramming`

This document is still a draft. This means that there may be typos, errors, or imprecisions. Please, read it critically and report me any correction or suggestion that may help to improve it.

These notes sketch the content of the 3rd lecture. The topics of this lecture are treated in any introductory book on Algorithms. You may refer to Chapters 2.3, 7, and 8 of *Introduction to Algorithms, 3rd Edition*, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2009 for a more detailed explanation.

1 Divide&Conquer paradigm

Divide&Conquer is one of the most powerful algorithm design paradigm. The idea is to solve a problem by splitting it into subproblems of smaller sizes. Then, those subproblems are solved recursively and their solutions are combined to obtain the solution of the original problem.

More preciselly, a Divide&Conquer algorithm follows three steps.

1. **Divide** the instance of the problem into smaller subinstances of the same problem.
2. **Solve** recursively the subinstances. If a subinstance's size is small enough, solve it directly with a straightforward approach.
3. **Combine** solutions of subinstances to obtain the solution of the original instance.

Recursion stops as soon as the size of a subinstance is small enough to be easily solved with a straightforward (possibly inefficient) approach.

2 Binary search

Binary search is one of the most easy (and useful) applications of the paradigm. It is the most efficient algorithm to search for keys in a sorted array.

▶ **Problem 1** (Searching). *We are given a sorted array $A[0, n]$ of keys and key k, the goal is to find the position of k in A, if any.*

Divide&Conquer can be easily applied.

1. **Divide**: split the array into two halves of (roughly) equal size.
2. **Solve**: recursively search k only in the half that may contain it. Check explicitly if there are a constant (e.g., 1) number of elements in the subarray.
3. **Combine**: Do nothing. Just report the answer.

The pseudocode of the binary search is reported below.

```
BinSearch(A, i, j, k)
1.  if(i == j)
2.      if(A[i] == k)
3.          return i
4.      else
5.          return −1
6.  c = ⌊(i+j)/2⌋
7.  if(k ≤ A[c])
8.      return BinSearch(A, i, c)
9.  else // implies k > A[c]
10.     return BinSearch(A, c+1, j)
```

Binary search requires $\Theta(\log n)$ comparisons and, thus, takes $\Theta(\log n)$ time if any comparison takes constant time. This is true for integers, floats, doubles, ... but not for strings. As a comparison may require the entire scan of the string k we are searching for, the cost of binary searching an array of strings is $\Theta(m \log n)$, where m is the length of the string k. This time complexity can be reduced to $\Theta(m + \log n)$ by adding additional data structures to speed up the required comparisons.

Pseudocode above can be easily modified to solve other kinds of queries. For example, find the predecessor or successor of a given key k, find the range of keys between two keys k and k', and so on.

Exponential search is a variant of binary search which has two steps.

- Find the smallest prefix of A which contains k and has a length which is a power of two.
- Use binary search on this prefix to find k.

The first step is done by comparing k with the last element of every prefix of A which is a power of two until we find a larger element. We thus compare elements at positions $0, 1, 3, 7, 15, \ldots$ until we find the first element, say at position i, such that $A[i] \geq k$. Then, we conclude with the second step by binary searching the interval $[i/2, i]$.

Exponential search has two advantages: 1) it can be used even if n is unknown; 2) it requires $\Theta(\log j)$ comparisons, where j is the position of k in A.

There are cases in which the first point above is useful. Consider the following problem.

▶ **Problem 2.** *We are given a monotone funtion $f()$. The goal is to find the smallest integer i for which the function $f()$ is positive.*

The problem can be solved in $\Theta(\log i)$ time with exponential search.

3 Merge Sort

Both QuickSort and Merge Sort are based on Divide&Conquer paradigm. I invite you to review QuickSort on your own. In the following I'll briefly recall Merge Sort as it can be used to solve one of the problems of this lecture: Inversions count.

Merge Sort uses the following three steps.

1. **Divide** the array of n elements in 2 subarrays of (roughly) $n/2$ elements each.
2. **Solve** the two subproblems recursively. Stop if the size is 1.
3. **Combine** the two (now sorted) subarrays by merging them.

Combine step requires to merge two sorted subarrays. This can be easily done in linear time by using the following algorithm.

Merge(L, R)

```
1.  L[n + 1] = R[m + 1] = +∞ // Sentinel
2.  i = j = 1
3.  for k = 1 to n+m
4.      if(L[i] ≤ R[j])
5.          C[k] = L[i]
6.          i = i + 1
7.      else
8.          C[k] = R[j]
9.          j = j + 1
```

The pseudocode of Merge Sort is the following.

Merge Sort**(A, p, r)**

```
1.  if (p < r)
2.      q = ⌊(p+r)/2⌋
3.      Merge Sort(A, p, q)
4.      Merge Sort(A, q+1, r)
5.      Merge (A, p, q, r)
```

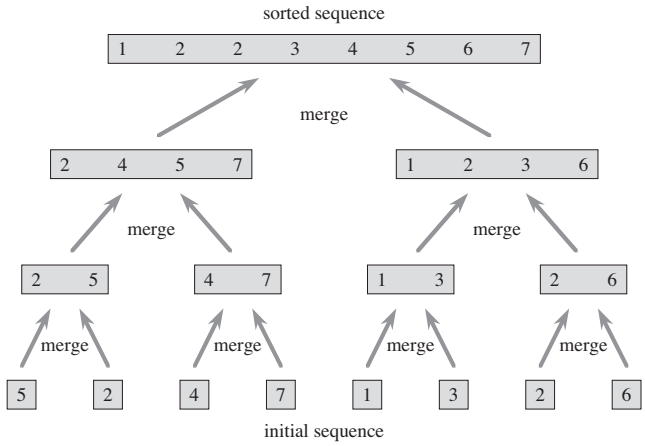


Figure 1 Merge Sort on the array $A = \{5, 2, 4, 7, 1, 3, 2, 6\}$

It should be clear that Merge Sort requires $\Theta(n \log n)$ comparisons.

Now, we would like to solve the *Inversions count* problem, which is as follows.

▶ **Problem 3** (Inversions count). *Given an array $A[0 \ldots n-1]$ of n distinct positive integers. If $i < j$ and $A[i] > A[j]$ then the pair (i, j) is called an inversion of A. The goal is to count the number of inversions of A.*

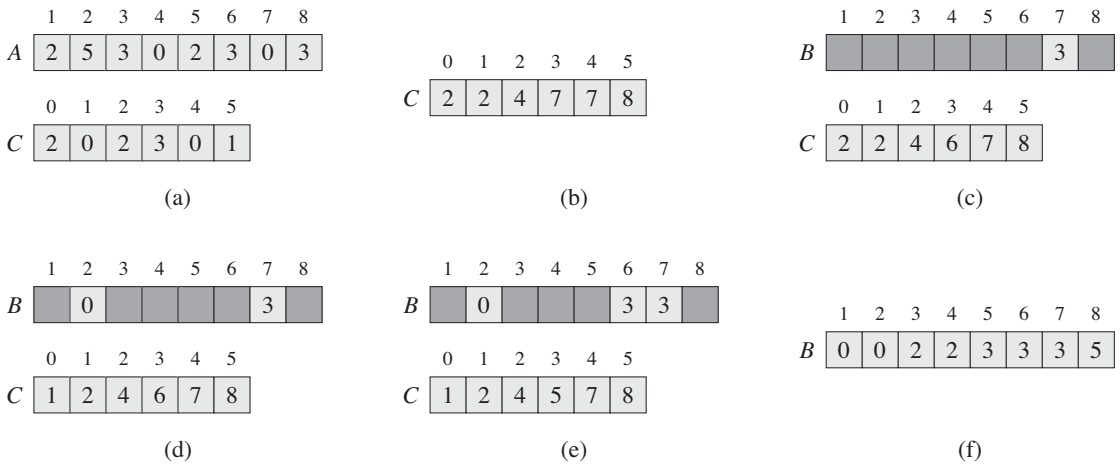
4 Sorting in linear time

Merge Sort requires $\Theta(n \log n)$ time, which is optimal time in the comparison model. This implies that we cannot sort faster, right? NOOOOO!

Counting Sort and Radix Sort are able to sort in linear time, under certain assumptions on the elements of the array. More precisely, let us assume that $A[1, n]$ contains only positive integers and that M is the largest element.

Counting Sort runs in $\Theta(M+n)$ time. Observe that this is linear time whenever $M = O(n)$. Instead, Radix Sort runs in linear time whenever $M = O(n^c)$, where c is a constant.

```
Counting Sort (A, B, max)
1.  for i = 1 to max
2.      C[i] = 0
3.  for j = 1 to n
4.      C[A[j]] = C[A[j]] + 1
5.  // C[i] stores the number of occurrences of i in A
6.  for i = 1 to max
7.      C[i] = C[i] + C[i-1]
8.  // C[i] stores the number of occurrences of elements smaller than or equal to i in A
9.  for j = n downto 1
10.     B[C[A[j]]] = A[j]
11.     C[A[j]] = C[A[j]]-1
```



Counting Sort is stable. A sorting algorithm is stable iff two equal elements in the array A preserve their relative order in the sorted array. This is a key property for one of its most important applications

Radix Sort is a sorting algorithm that uses Counting Sort to sort in linear time whenever $M = O(n^c)$, for any constant c.

Suppose we have to sort the following 3-digit numbers.

329
457
657
839
436
720
355

Probably you'd start sorting the numbers by their most significant digits.

329	329	329	329
457	355	355	355
657	457	437	437
839	437	457	457
436	657	657	657
720	720	720	720
355	839	839	839

Radix Sort instead starts from the less significant digits and uses stability to get the final array sorted.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	567	355	567
720	329	457	720
355	839	567	839

▶ **Theorem 4.** *Given n numbers of d digits each of which is a value in $[0, k-1]$, Radix Sort correctly sorts the numbers in $\Theta(d(n+k))$ time.*

Correctness of Radix Sort can be proven by indiction on the number of digits. Assume Radix Sort already sorted correctly the firt $d-1$ digits, what does it happen sorting the dth digit? If two numbers have the same value at the dth digit, do we need stability?

Time complexity follows by observing that at any of the d iterations we use Counting Sort to sort the digits in $\Theta(n+k)$.

What is the time complexity to sort n values with $M = O(n^c)$ with c constant?

Consider the binary representation of these values. Each of them fits in at most $c \log n$ bits. Suppose we split their representations in c digits of $\log n$ bits each.

This way, we obtain numbers with c digits in the range $[0,\ k = 2^{\log n} - 1 = n - 1]$. By the previous theorem we know that the time complexity is $\Theta(c(n+n)) = \Theta(n)$.