Algorithm and data structures

Roberto Antoniello

July 31, 2023

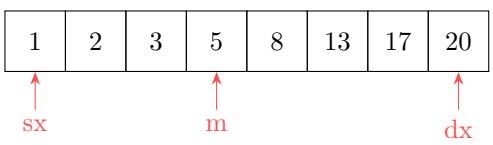
In this file I will resume all the concepts I liked while studying the course of Algorithm and data structures.

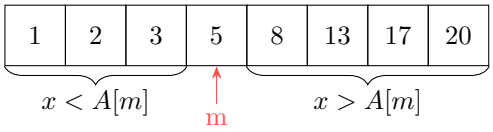
1 Binary search

This algorithm can be used only if you have a sorted array. Here how it works: BinarySearch take a sorted array as input and return an index as output. So it returns the index of the found element or -1 if not found.
When it starts execution, the algorithm saves three variables **sx**, **dx** and **m**. The **m** variable is the index in the middle of the array, **sx** and **dx** are the first and the last index. It asks if the element is less or more than the element in **m** position.
Basically, if the element is x the question is: $\mathbf{x} < \mathbf{A[m]}$ or $\mathbf{x} > \mathbf{A[m]}$? We are reducing the search space by 2 every time because if it's less, our **dx** becomes **m**, otherwise our **sx** becomes **m+1**.

At the first iteration the search space is n elements, at the second it is $\frac{n}{2}$, at the third one it is $\frac{n}{2^2}$ and so on.

At the i° iteration it will be $\frac{n}{2^i}$.
During the last iteration the size of our array A is 1. So:
$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$
We have just said the amount of steps are $\log_2 n \Rightarrow O(\log n)$
Here below there's an array as example with the initial value of sx,m and dx.





I let you read the pseudocode here below.

Listing 1:*Iterative version of the binary search algorithm.*

```
Algorithm BinarySearch(Array A[0,..,n-1]) --> index
   sx <-- 0
   dx <-- n
   index <-- -1
   while sx < dx do
      m <-- (sx+dx) / 2
      if x < A[m] then
         dx <-- m
      else if x = A[m] then
         index <-- m
      else
         sx <-- m+1
```

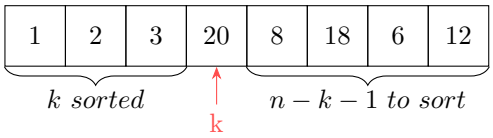
2 Selection Sort

So we know that to do a biinary search, we need a sorted array.
There are many solutions to sort an array with more or less complexity in time and space.
The first algorithm we're gonna see is the selection sort. This sorting algorithm is really simple, essentially it slides the array and when it find the minimum value it put it at the beginning in the right place. After this step we consider that element sorted and so on until we have the entire array sorted.
Let's see here below the pseudocode and then a quick and easy complexity analysis.

Listing 2:*Selection sort algorithm.*

```
Algorithm SelectionSort(Array A[0,..,n-1])
   for k <-- 1 to n-2 do
      //finding the index m of minimum value between k and n-1
      m <-- k
      for j <-- k+1 to n-1 do
         if A[j] < A[m] then
            m <-- j
      swap A[m] with A[k]
```

The first for cycle ends at **n-2** because the last element is already sorted if we remember how it works the algorithm.
The second for cycle finds the minimum value starting from m position.
So, how many comparisons have I to do?
If we look at our array A, we can consider that at the k iteration we have **k** sorted elements on the left and **n-k-1** to sort on the right.



Basically we are doing this sum:

$$\sum_{k=0}^{n-2}(n-k-1)$$

This is equal to sum every $n \in \mathcal{N}$ from 0 to n-2, so:

$$\sum_{k=0}^{n-2}(n-k-1) = \sum_{i=1}^{n+1} i = \frac{n \cdot (n-1)}{2}$$

This is the total of comparisons I have to do in this algorithm. The complexity is thus $\Theta(n^2)$.
We can say also that this amount of comparisons is always done because we do not set any condition to do these comparisons.

3 Bubble sort

The second sorting algorithm we're gonna see is Bubble sort.
This works by sliding the array and compare the current element with the previous one. The bigger element is moved in the next position. This make the bigger element go forward until there's another bigger than it. It's called bubble because these big values go forward like the bubbles in the water.
Let's do a simulation here below.

5	7	6	1	13	2	12

let's do the first iteration.

5	6	1	7	2	12	13

We compared 5 with 7 ⇒ already sorted.
Then 7 with 6 ⇒ swap.

Then 7 with 1 ⇒ swap.
Then 7 with 13 ⇒ already sorted.
Then 13 with 2 ⇒ swap.
Then 13 with 12 ⇒ swap.
In the next iteration I'll slide again until the entire array is sorted.

Now we consider the pseudocode.

Listing 3:*Bubble sort algorithm.*

```
Algorithm BubbleSort(Array A[0,..,n-1])
   i <-- 1
   do
      swapped <-- false
      for j <--1 to n-i do
         if A[j] < A[j-1] then
            swap A[j] with A[j-1]
            swapped <-- true
      i <-- i + 1
   while swapped and i < n
```

The for cycle ends at **n-i** because the last **i** elements are already sorted as we discussed how it work bubble sort. The external do while cycle ends when we did an iteration without any swap of elements or we reached the **n-1** iteration. The **i** < **n** condition saves us the last iteration without any swap sliding the already sorted array.
Let's analyse the comparisons. Inside the main for cycle we do **n-i** comparisons where **i** starts at **1** to **n-1**. We're doing this sum:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{n\cdot(n+1)}{2} = \Theta(n^2)$$

Again, the total amount of comparisons is in the order of n^2, but there's a difference between selection sort and bubble sort. That's because here we have this amount of comparisons only in the worst case(reverse sorted array), but in the best case(array already sorted) we can easily see that the amount of comparisons is in the order of $\Theta(n)$ because we just slide one time the array and then the algorithm ends doing **n-1** comparisons.

4 Merge sort

Let's consider the idea of having two already sorted arrays.
How can we build an array that contains all the elements sorted?
The first option we can think is to do a bubble sort and surely we can do it in n^2 steps.
Anyway there's a clever way to do this and it's called **merge**.

5	6	10	12

2	9	15	20

2	5	6	9	10	12	15	20

4.1 merge

The merge procedure works as this:
at every step I compare the first two numbers and the mininum go in the final array. When one of the sorted array ends, I append the remaining numbers of the other array without compare anything.
How much it costs? Well, every compare will put an element in the right place, so in the worst case I compare all the elements and the remaining is only one element in the other array. So the merge procedure has complexity in the order of $O(n)$ because I do **n-1** comparisons in the worst case.
Here below I drop the pseudocode of a simple version of the merge procedure, which it isn't perfectly optimized in memory.

Listing 4:*Merge algorithm.*

```
Algorithm Merge(Array B[0,..,lb-1], Array C[0,..,lc-1]) --> Array
   Be X[o,..,lb+lc-1] an array
   i1,i2,k <-- 0
   while i1 < lb and i2 < lc do
      if B[i1] <= C[i2] then
         X[k] <-- B[i1]
         i1 <-- i1+1
      else
         X[k] <-- C[i2]
         i2 <-- i2+1
      k <-- k +1
   //The remaining elements are in B
   if i1 < lb then
      for j <- i1 to lb-1 do
         X[k] <-- B[j]
         k <-- k+1
   else
      //The remaining elements are in C
      for j <--i2 to lc-1 do
         X[k] <-- C[j]
         k <-- k+1
   return x
```

Let's remember two corollaries we will use later:

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1$$

If you convert in binary, you can see that all the power of two are 1 with all zeros behind. If you sum those all you obtain 2^k.
Now the second corollary:
$\forall n \in \mathcal{N}\ \exists$ a power of two $N\ :\ n \leq N \leq 2n$

4.2 Using merge inside Merge sort

Let's define Merge sort recursively:
If $n \leq 1$ then A is already sorted, I do nothing.
Otherwise:

1. Divide A by two equal parts.
2. Sort them separately.
3. Merge them in a unique array(merge algorithm)

I'll show here below the pseudocode of the Merge sort, which use our merge algorithm written before. So this algorithm suffers too of the missing optimization in memory, but it will be good enough to do our complexity analysis.

Listing 5:*Merge algorithm.*

```
Algorithm Mergesort(Array A[0,..,n-1)
   If n > 1 then
      m <-- n / 2
      B <-- A[o,..,m-1]
      C <-- A[m,..,n-1]
      Mergesort(B)
      Mergesort(C)
   A <-- merge(B,C)
```

4.3 Amount of comparisons

The amount of comparisons made are:

1. Comparisons of Mergesort(B).
2. Comparisons of Mergesort(C).
3. Comparisons of merge(B,C).

The equation resulting from this observation is here below and we're going to solve it.

$$\begin{cases} 0 & if\ n \leq 1 \\ C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + C_{merge}(n) & otherwise \end{cases}$$

Let's consider our **n** is even. The equation becomes:

$$\begin{cases} 0 & if\ n \leq 1 \\ 2C\left(\frac{n}{2}\right) + C_{merge}(n) & otherwise \end{cases}$$

By solving this equation we're going to demonstrate the amount of comparisons Mergesort do until we have a sorted array in the end. We'll solve by substitution.

$$C(n) = 2C\left(\frac{n}{2}\right) + n - 1 =$$
$$2\left[2C\left(\frac{n}{2^2}\right) + \frac{n}{2} - 1\right] + n - 1 =$$
$$2^2 \cdot C\left(\frac{n}{2^2}\right) + n - 2 + n - 1 =$$
$$2^2 \cdot \left[2C\left(\frac{n}{2^3}\right) + \frac{n}{2^2} - 1\right] + n - 2n - 1 =$$
$$2^3 C\left(\frac{n}{2^3}\right) + n - 2^3 + n - 2^1 + n - 2^0 =$$

$$2^3 C\left(\frac{n}{2^3}\right) + 3n - \sum_{i=0}^{3-1} 2^i =$$

$$2^k C\left(\frac{n}{2^k}\right) + kn - \sum_{i=0}^{k-1} 2^i =$$

We know that $\sum_{i=0}^{k-1} 2^i = 2^k - 1$
Now let's manipulate to the base case:
$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2(n)$ Let's substitute now:
$C(n) = n \cdot C(1) + n \cdot \log_2 n - n + 1$

We know that $C(1) = 0$, so the amount of comparisons is in the order of

$$\Theta(n \cdot \log(n))$$

What about the recursion stack of memory?
If we still consider n as even, then the equation is:

$$H(n) = 1 + H\left(\frac{n}{2}\right)$$

If we do similar manipulation as made a moment ago we can reach this result:

$$H(n) = \Theta(\log(n))$$

5 Quick sort

The recursive definition of Quick sort is pretty similar to the Merge sort one. This is because both algorithm are using the Divide et impera technique.
The difference between these two sortin algorithm is quite simple. Inside Mergesort the complex part was the final merge, instead here we're going to see that the complex part will be the first one of dividing the problem in more problem of minor size.

44	55	12	42	94	9	18	64

We choose a pivot element, 42 for example. if $n \leq pivot$ then those elements go on the left of pivot, otherwise they go on the right.

9	12	18	42	44	55	64	94

Pivot

How to do this? We use an algorithm we will call partition. We define two index **sx** and **dx**. sx starts at the beginning of array while dx starts from the end.

1. dx starts sliding until it finds an element $\leq$ of pivot.
2. sx then starts sliding until it finfd an element $>$ of pivot.
3. Then swap sx with dx.
4. If sx $<$ dx, then swap dx with the pivot.

Let's give a look to the pseudocode of partition here below.

Listing 6:*Partition algorithm.*

```
Algorithm Partition(Array A[0,..,n-1],index i,index f) --> index
  pivot <-- A[i]
  sx <-- i
  dx <-- f
  while sx < dx do
    do
      dx <-- dx-1
    while A[dx] > pivot
    do
      sx <-- sx+1
    while sx < dx and A[sx] <= pivot
    if sx < dx then
      swap A[sx] with A[dx]
  swap A[i] with A[dx]
  return dx
```

The amount of comparisons is in the order of $O(n)$.