Exercise of Getting Started

Grabur

Feb 2025

1 Chapter 1: Foundations

1.1 The Role of Algorithms in Computing

I didn't make the exercise of this section because I didn't find them useful.

1.2 Getting Started

Exercise 1.2-1):
It could be an application like booking. When you search a hotel close to the airport, it gets involved algorithms as searching the hotels close to that airport and it should be searched in a short time period.

Exercise 1.2-2):

$$8n^2 < 64n \cdot \log_2 n \quad \rightarrow \quad n < 8 \cdot \log_2 n$$

Try values until this inequality is false. To $n \lessapprox 43$, insertion sort runs faster than merge sort.

Exercise 1.2-3):

$$100n^2 < 2^n$$

Trying values, for $n \lessapprox 15$, 2^n runs faster than $100n^2$.

Exercise 1.2-4):

View photo of the exercise on the next page.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$	∞	∞	∞	∞	∞	∞	∞
$\sqrt{n}$	10^{12}	$3.6 \cdot 10^{15}$	$1.3 \cdot 10^{19}$	$7.47 \cdot 10^{21}$	$6.91 \cdot 10^{24}$	$9.95 \cdot 10^{26}$	$9.95 \cdot 10^{30}$
n	10^6	$6 \cdot 10^7$	$3.6 \cdot 10^9$	$8.64 \cdot 10^{10}$	$2.63 \cdot 10^{12}$	$3.16 \cdot 10^{13}$	$3.16 \cdot 10^{15}$
$n \lg n$	62746	$2.8 \cdot 10^6$	$1.33 \cdot 10^8$	$2.76 \cdot 10^9$	$7.29 \cdot 10^{10}$	$7.99 \cdot 10^{11}$	$6.87 \cdot 10^{13}$
n^2	1000	7746 (aprox)	60000	293939	$1.62 \cdot 10^6$	$5.62 \cdot 10^6$	$5.61 \cdot 10^7$
n^3	100	391 (aprox)	1532	4420	13803	31601	146679
2^n	20 (aprox)	26 (aprox)	32	36	41	44	51
$n!$	9	11	12	13	15	16	17

Exercise 2.1-1):
Note: Resolved using the logic of C, C++, Java, etc. while iterating over an array on a for loop. Also the number that appears in green, is the number being checked. The number or numbers that appears in red are the numbers being moved.

i	Array
1)	[31, 41, 59, 26, 41, 58]
2)	[31, 41, 59, 26, 41, 58]
3)	[26, 31, 41, 59, 41, 58]
4)	[26, 31, 41, 41, 59, 58]
5)	[26, 31, 41, 41, 58, 59]

Exercise 2.1-2):
Initialization: The loop start getting the first number in the array. In spite of that, it has initialized to 0 the variable sum where the total sum will be stored. Due to that, the invariant holds the first number that will be added to sum.

Maintenance: On each iteration, the loop will hold only the index of the number that will be added, after add it, i will be incremented by 1, holding the next number (i + 1).

Termination: The loop will terminate when the 'n' elements of the array are added. In conclusion, sum it's equivalent of say that $sum = \sum_{i=1}^{n} A[i]$.

Exercise 2.1-3):
click this link to see the resolution → *resolution*

Exercise 2.1-4):

Algorithm 1 Linear Search

function LINEAR-SEARCH(A, n, x)
  for $i \leftarrow 1$ **to** n **do**
    if $A[i] == x$ **then return** i
  return *NIL*

Initialization: The loop start getting the first element of the array.

Maintenance: On each iteration, the loop takes the next element (i + 1) and compare it with the value being search (x). If it's found return i, else, continue searching that value.

Termination: When all values are read, if x wasn't found in the array, it returns NIL to indicate that no value was found on all the array.

Exercise 2.1-5):

Algorithm 2 ADD-BINARY-INTEGERS

```
function ADD-BINARY-INTEGERS(A, B, n)
    //Initialize array C with n values
    carry ← 0
    for i ← 1 to n do
        c ← A[i] + B[i]
        C[i] ← c mod 2
        carry ← c ÷ 2    //Integer division

    C[n] ← carry
    return C    //Return the array C with the values
```

Initialization: The loop starts with value of carry to 0, and getting the first bits of A and B.

Maintenance: On each iteration, the loop takes the next bits values of A and B. Add these values and calculate the value to insert into C and the carry that could exists.

Termination: All values were added and store, now C[0:n - 1] with the result of the sum. To reach the n-th value, adds the last carry value on the position n.

Exercise 2.2-1):
Like the book says, Θ notation is like saying "roughly proportional to n^2 (for example), when n is large." In this case, we remove constants, so the remaining expression is $n^3 + n^2 + n + 3$. The term with the highest exponent is n^3, so at any moment: $n^3 \gg n^2 \gg n$.
Solution: $\Theta(n^3)$.

Exercise 2.2-2):

Algorithm 3 SELECTION-SORT

```
function SELECTION-SORT(A, n)
    for i ← 1 to n − 1 do
        ind_small_elm ← i
        for j ← i + 1 to n do
            if A[ind_small_elm] > A[j] then
                ind_small_elm ← j
        SWAP(A[i], A[ind_small_elm])
```

The invairant is that on each iteration of extern for, it only takes 1 by 1 element. In the inner for, also take all elemnts from i to n, and compare the value of the outter for against the inner for to take the smaller elemnt.

When the algorithm arrives to the last element, all swaps ocurred and the last element will be in the correct place.

The worst case happens when it must iterate on the outher for and also with al the elements from i to n in the inner for. So it's: $\frac{n*(n-1)}{2}$. Thats mean that avoiding all constants values, the solution is: $\Theta(n^2)$.

The best case is not better because you have to check all values in the if, the only instruction that is avoided is the instruction inside the if because the if won't be evaluated to true. But that instruction is insignificant if n it's too big.

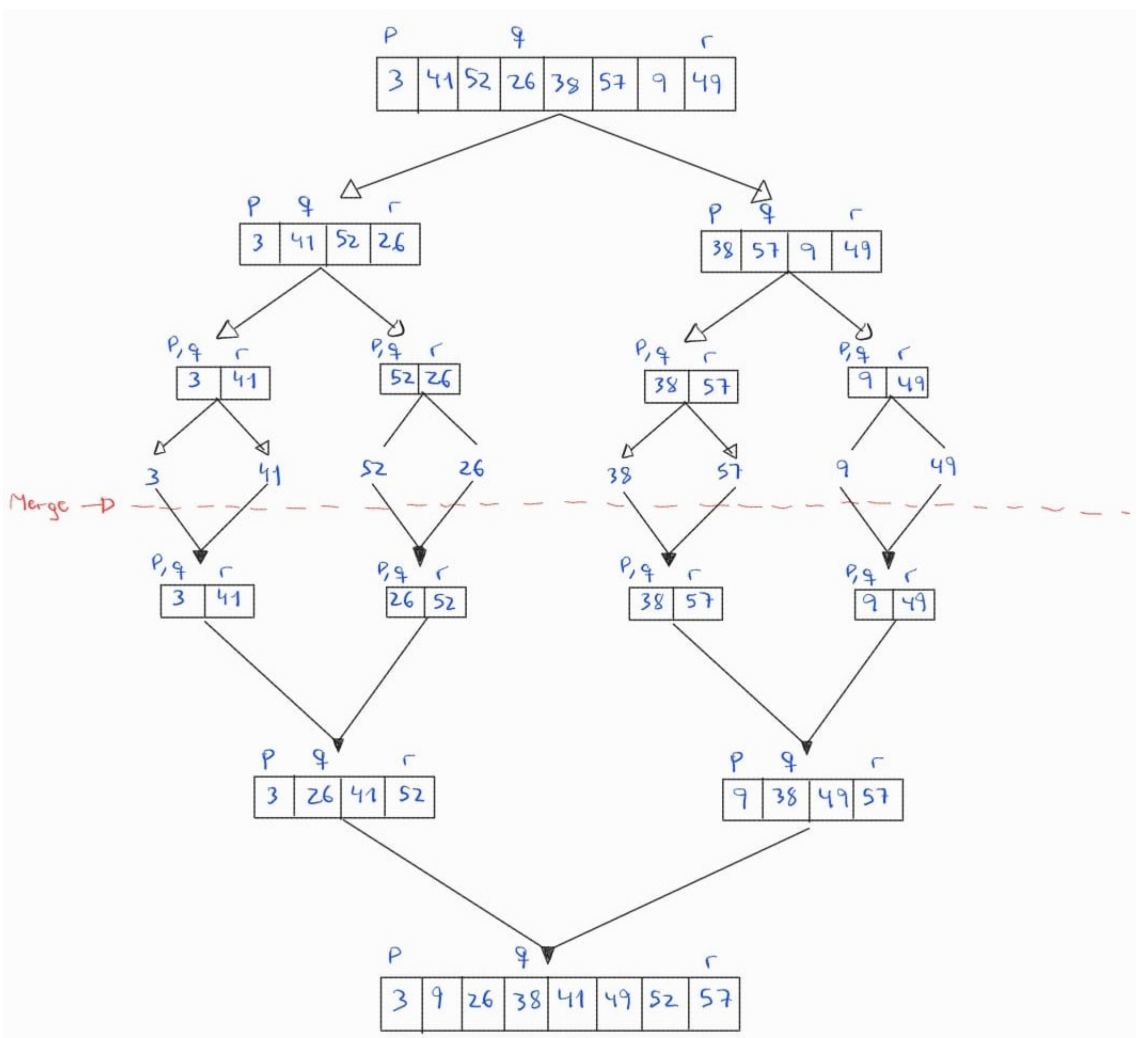
Exercise 2.2-3):
Depends on the value where is storage, if the x value is storage at the first position it will take a constant value to search it. However, if the value is in the last element (worst case) it will spend $constant * n$ time to find that value.

Averege case is suposing that it's in the middle of the array. The averegage is $\frac{n}{2} = n$ if n it's too big.

Worst case as mentioned before is $\Theta(n)$.

Exercise 2.2-3):
The only thing you could do is a preprocessing step to check if it's alredy sorted or nearly to be sorted and then apply the algorithm who best fits when the best case was achieved.

Exercise 2.3-1):



Exercise 2.3-2):
The "***if*** $p \neq r$" is not useful because $p = 1$ and $q > 1$, so the if will be evaluated to true, and the return (the termination of recursion) will execute without doing any recursion step.

Exercise 2.3-3):
Initialization: The loop starts obtaning the first element to insert it in the sorted array.

Maintenance: On each iteration, the loop insert 1 element of the left or right array, then increment k by 1, to insert on the next iteration the next value 1 by 1. The values that will be inserted with the first loop in the best case is *n - 1* where n is the lenght of the subarray/array being sorted.

Termination: In one of the last 2 whiles, it will be inserted all remaining values (could be values in the left or right array) until reach the nth values in the sorted array.

Exercise 2.3-4):
n = 2

$$T(2) = 2$$

The solution given says that:

$$T(2) = 2 * \log_2 2 = 2 \cdot 1 = 2$$

Induction hypotesis:

$$T(k) = k \cdot \log_2 k$$

Where k is a power of 2. We want to demostrate that *n = 2k*. Use recurrence to calculate $T(2k)$:

$$T(2k) = 2T(k) + 2k \quad \rightarrow \quad T(2k) = 2 \cdot (k * \log_2 k) + 2k \quad \rightarrow \quad T(2k) = 2k \cdot (\log_2 k + 1) \quad \rightarrow \quad T(2k) = 2k \cdot \log_2 2k$$

As we mentiones before, replacing *n = 2k*, we conclude that the induction hypotesis is correct $n \cdot \log_2 n$

Exercise 2.3-5):

Algorithm 4 INSERTION-SORT-RECURSIVE

function INSERTION-SORT-RECURSIVE(A, n)
    if $n == 0$ **then return**
    INSERTION-SORT-RECURSIVE$(A, n - 1)$
    key $\leftarrow$ *A[n]*
    $i \leftarrow n - 1$
    while $i \geq 0$ **and** $A[i] > key$ **do**
        $A[i + 1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i + 1] \leftarrow key$

Exercise 2.3-6):

Algorithm 5 BINARY-SEARCH

```
function BinarySearch(A, x, l, n)
    if l > n then return
    mid ← l + n/2
    if A[mid] > x then
        BinarySearch(A, x, l, mid − 1)
    else if A[mid] < x then
        BinarySearch(A, x, mid + 1, r)
    else   return mid
```

Exercise 2.3-7): You can't use binary search because while you are sorting most part of the time, it won't be that value in the sorted array. Maybe you should make a variation of binary search to find out in which direction has less difference betwen the number being search and then you should move all values 1 position to right and insert in the new position.

On worst case, imagine that the worst value is at the least position, you should move n items to the right + n / 2 that cost to search all the. So it's $n \cdot \frac{n}{2} = \Theta(n^2)$

Exercise 2.3-8):

Algorithm 6 FIND-SUM-TWO-ELEMENTS

```
function FindSumTwoElements(S, n, x)
    MergeSort(S, 0, n - 1)    //Θ(n ∗ logn)
    i ← 0
    j ← n − 1
    while i < j do    //Θ(n)
        if S[i] + S[j] = x then
            return i, j
        else if S[i] + S[j] < x then
            i ← i + 1
        else
            j ← j − 1
    return No pair found
```