

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):

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

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

Exercise 1.2-3):

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

Trying values, for $n \lesssim 15$, 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 (approx)	60000	293939	$1.62 \cdot 10^6$	$5.62 \cdot 10^6$	$5.61 \cdot 10^7$
n^3	100	391 (approx)	1532	4420	13803	31601	146679
2^n	20 (approx)	26 (approx)	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):

Exercise resolved in the file resolution

Exercise 2.1-4):

Algorithm 1 Linear Search

```

1: function LINEAR-SEARCH( $A, n, x$ )
2:   for  $i \leftarrow 1$  to  $n$  do
3:     if  $A[i] == x$  then return  $i$ 
4:   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

```
1: function ADD-BINARY-INTEGERS( $A, B, n$ )
2:   //Initialize array  $C$  with  $n$  values
3:    $carry \leftarrow 0$ 
4:   for  $i \leftarrow 1$  to  $n$  do
5:      $c \leftarrow A[i] + B[i]$ 
6:      $C[i] \leftarrow c \bmod 2$ 
7:      $carry \leftarrow c \div 2$  //Integer division

8:    $C[n] \leftarrow carry$ 
9:   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

```
1: function SELECTION-SORT( $A, n$ )
2:   for  $i \leftarrow 1$  to  $n - 1$  do
3:      $ind\_small\_elm \leftarrow i$ 
4:     for  $j \leftarrow i + 1$  to  $n$  do
5:       if  $A[ind\_small\_elm] > A[j]$  then
6:          $ind\_small\_elm \leftarrow j$ 
7:     SWAP( $A[i], A[ind\_small\_elm]$ )
```

The invariant 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 outer for and also with all the elements from i to n in the inner for. So it's: $\frac{n*(n-1)}{2}$. That means that avoiding all constant 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 is 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.

Average case is supposing that it's in the middle of the array. The average is $\frac{n}{2} = n$ if n is 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 already sorted or nearly to be sorted and then apply the algorithm who best fits when the best case was achieved.