BT RPA –Answers

Algorithm

Estimated time: 30 min.

Actual time: 1h:15min.

Documentation:

/*

This function has an array of integer numbers and an index as input. It sorts the array in ascending order until it has reached the specified index and it returns the element from that index from the sorted array.

Input parameters: **a-**array containing integer numbers. Type: int[]

b-index/position from the sorted array from which the user wants the element. Type: int

Output parameters: **a[e]-**the element from the sorted array at the given index.

Type: int

Time Complexity: θ**(n^2).**

*/

How I figured out what does the algorithm do: I started with some basic and simple examples such as a=[2,4,5,3,6] and b=4, a=[2,4,5,3,6] and b=1, a=[2,3,4,5,6] and b=98 etc. I will reproduce down below some of these cases such as following:

Test case a)

We take into consideration the following values for a and b:

a=[2,4,5,3,6]

b=4

alg1

a=[2,4,5,3,6];

b=4;

c=0;

d=4;

4 >= 0? YES → e = alg2(a,0,4,0);

alg2

a=[2,4,5,3,6];

b=0;

c=4;

d=0;

e=2;

f=0;

alg3(a,0,4);

alg3

0 != 4 ? YES → a[4]=(a[0]=(a[4]=a[0]^a[4])^a[0])^a[4]

a[4] = 2^6 = 0010 XOR 0110 = 0100 = 4

a[0] = 4^2 = 0100 XOR 0010 = 0110 = 6

a[4] = 6^4 = 0110 XOR 0100 = 0010 = 2

➔ a = [6,4,5,3,2]

go back to alg2

for 0<=i<4

//i=0 → 6 < 2 ? NO→i++;

//i=1 → 4< 2? NO → i++;

//i=2 → 5<2? NO → i++;

//i=3 → 3<2? NO → i++;

end of for loop

alg3(a,4,0)

alg3

4 != 0 ? YES → a[0]=(a[4]=(a[0]=a[4]^a[0])^a[4])^a[0]

a[0]=2^6=4

a[4]=4^2=6

a[0]=6^4=2

→ a=[2,4,5,3,6]

go back to alg2

return f → **go back to alg1** → e=alg2(a,0,4,0)=0;

0==4 ? NO → 0 < 4? YES → c=0+1=1

4 >= 1? YES e = alg2(a,1,4,1);

alg2

a=[2,4,5,3,6]

b=1

c=4

d=1

e=4

f=1

alg3(a,1,4);

alg3

1 != 4 ? YES → a[4]=(a[1]=(a[4]=a[1]^a[4])^a[1])^a[4]

a[4] = 4 ^ 6 = 2

a[1] = 2 ^ 4 = 6

a[4] = 6 ^ 2 = 4

→ a=[2,6,5,3,4]

go back to alg2

for 1 <= i < 4

//i=1 → 6 < 4 ? NO →i++;

//i=2 → 5 < 4 ? NO → i++;

//i=3 → 3 < 4 ? YES alg3(a,3,1) and f++→ f=2;

alg3

3 != 1 ? YES → a[1]=(a[3]=(a[1]=a[3]^a[1])^a[3])^a[1]

a[1] = 3 ^ 6 = 5

a[3] = 5 ^ 3 = 6

a[1] = 6 ^ 5 = 3

→ a=[2,3,5,6,4]

go back to alg2

end of for loop

alg3(a,4,2)

alg3

4 != 2 ? YES a[2]=(a[4]=(a[2]=a[4]^a[2])^a[4])^a[2]

a[2] = 4 ^ 5 = 1

a[4] = 1 ^ 4 = 5

a[2] = 5 ^ 1 = 4

➔ a=[2,3,4,6,5]

go back to alg2

return f → **go back to alg1** → e = alg2(a,1,4,1) = 2;

2==4 ? NO → 2 < 4 ? YES → c=2+1=3

4 >= 3? YES → e = alg2(a,3,4,3);

alg2

a=[2,3,4,6,5]

b=3

c=4

d=3

e=6

f=3

alg3(a,3,4)

alg3

3 != 4 ? YES → a[4]=(a[3]=(a[4]=a[3]^a[4])^a[3])^a[4]

a[4] = 6 ^ 5 = 3

a[3] = 3 ^ 6 = 5

a[4] = 5 ^ 3 = 6

➔ a=[2,3,4,5,6]

go back to alg2

for 3<=i<4 → // i=3 → 5 < 6 ? YES → alg3(a,3,3) but here 3 != 3 ? NO → f++=4;

end of for loop

alg3(a,4,4)

4 != 4 ? NO → return f → **go back to alg1** → e = alg2(a,3,4,3) = 4;

4 == 4? YES → return a[4] = 6;

Test case b)

We take into consideration the following values for a and b:

a=[2,4,5,3,6]

b=1

alg1

a=[2,4,5,3,6]

b=1

c=0

d=4

4 >= 0 ? YES → e=alg2(a,0,4,0);

alg2

a=[2,4,5,3,6]

b=0

c=4

d=0

e=2

f=0

alg3(a,0,4)

alg3

0 != 4 ? YES → a[4]=(a[0]=(a[4]=a[0]^a[4])^a[0])^a[4]

a[4] = 2 ^ 6 = 4

a[0] = 4 ^ 2 = 6

a[4] = 6 ^ 4 = 2

→ a=[6,4,5,3,2]

go back to alg2

for 0 <= i < 4

//i=0 → 6 < 2 ? NO → i++;

//i=1 → 4 < 2 ? NO → i++;

//i=2 → 5 < 2 ? NO → i++;

//i=3 → 3 < 2 ? NO → i++

end of for loop

alg3(a,4,0)

alg3

4 != 0 → a[0]=(a[4]=(a[0]=a[4]^a[0])^a[4])^a[0]

a[0] = 2 ^ 6 = 4

a[4] = 4 ^ 2 = 6

a[0] = 6 ^ 4 = 2

➔ a=[2,4,5,3,6]

go back to alg2

return f → go back to alg1 → e = alg2(a,0,4,0) = 0;

0 == 1 ? NO → 0 < 1 ? YES → c = 0 + 1 = 1

4 >= 1 ? YES → e = alg2(a,1,4,1);

alg2

a=[2,4,5,3,6]

b=1

c=4

d=1

e=4

f=1

alg3(a,1,4)

alg3

1 != 4 ? YES → a[4]=(a[1]=(a[4]=a[1]^a[4])^a[1])^a[4]

a[4] = 4 ^ 6 = 2

a[1] = 2 ^ 4 = 6

a[4] = 6 ^ 2 = 4

➔ a=[2,6,5,3,4]

go back to alg2

for 1 <= i < 4

//i = 1 → 6 < 4 ? NO → i++;

//i = 2 → 5 < 4 ? NO → i++;

//i = 3 → 3 < 4 ? YES → alg3(a,3,1) and f++=2;

alg3

3 != 1 ? YES → a[1]=(a[3]=(a[1]=a[3]^a[1])^a[3])^a[1]

a[1] = 3 ^ 6 = 5

a[3] = 5 ^ 3 = 6

a[1] = 6 ^ 5 = 3

➔ a=[2,3,5,6,4]

go back to alg2

end of for loop

alg3(a,4,2)

alg3

4 != 2 → a[2]=(a[4]=(a[2]=a[4]^a[2])^a[4])^a[2]

a[2] = 4 ^ 5 = 1

a[4] = 1 ^ 4 = 5

a[2] = 5 ^ 1 = 4

→a=[2,3,4,6,5]

Go back to alg2

return f → **go back to alg1** → e = alg2(a,1,4,1) = 2;

2 == 1 ? NO → 2 < 1 ? NO → d = 2-1=1

1 >= 0 ? YES → e = alg2(a,1,1,1);

alg2

a = [2,3,4,6,5]

b = 1

c = 1

d = 1

e = 3

f = 1

alg3(a,1,1)

0 != 1 ? NO → **go back to alg2**

alg2

for 1<=i<1 end of for loop

alg3(a,1,1)
alg3

1 != 1 ? NO → **go back to alg2**

alg2

return f → **go back to alg1** → e = alg2(a,1,1,1) = 1

1 == 1 ? YES → return a[1] = 3

From the previous examples I came to the conclusion that the **alg1** function returns the element from the given index b from the sorted array up to the index b (the array is sorted in an ascending manner from 0 up to the b index).

The complexity analysis of **alg1**:

Consider n→∞, so n is a very large number, there are 2 loops: while (from alg1) and for (from alg2 that is called in alg1), so the formula will be something like: Best_Case = Average_Case = Worst_Case = $\sum_0^{n-1}\sum_0^{n-1} 1$ = $\sum_0^{n-1} n$ = $\frac{n(n-1)}{2}$ = $\frac{n^2-n}{2}$ => $\theta(n^2)$.

Random Number Generation

Estimated time: 30 min.

Actual time: 30 min.

The implementation for this can be find in the git repository, link here:

https://github.com/RaluMV/Raspuns-Test-BT/tree/master/BT%20Random%20Number%20Generation

Minimum retrieving

Estimated time: 30 min.

Actual time: 20 min.

Consider n→∞, so n is a very large number.

The **Best_Case** is considered when the first element of the array is also the minimum element from the entire array, so this way, with the initial assigning of min=A[0] there is no need to perform any updates on the min variable (0 updates).

The **Worst_Case** is considered when the minimum element from the entire array is also the last element in the array. So, given the fact that min=A[0] there will be n-1 updates needed to be performed in order to have in the min variable the last element from the array.

The **Average_Case** is considered when the minimum element is somewhere in the middle of the array. Then, the number of updates needed to be performed is equal to $\frac{n}{2}$.

SOLID and OOP

Estimated time: 40 min.

Actual time: 1h: 35 min.

From my experience so far, I have studied and used the OOP and SOLID principles in the Clean Architecture architectural design pattern when implementing the AgriApp's Glia solution for my master thesis.

This architecture is based on the SOLID Principles (**S**- The **S**ingle Responsibility Principle, **O**- The **O**pen-Closed Principle, **L**-The **L**iskov Substitution Principle, **I**- The **I**nterface Segregation Principle and **D**- The **D**ependency Inversion Principle) which tell us how to group and arrange the functions and data structures into classes (groupings) and how to interconnect those classes (groupings). The OOP principles (Encapsulation, Abstraction, Inheritance and Polymorphism) help us, programmers, to model the real world.

Starting with encapsulation. This mechanism draws a line around cohesive sets of data and functions and outside this line, the data is hidden and only some of the functions that handle that data are known. I used this in the Domain layer from the solution's architecture. There are the model's classes which have private instance variables and public accessing methods.

Moving on to abstraction. It means a concept or an idea that is not necessarily associated with any particular instance. Using abstract classes/interfaces we express the inherit of the class rather than the actual implementation. In a way, one class should not know the inner details of another in order to use it, just knowing the interface should work just as fine. This came in use when I declared interfaces in the Application Layer of my solution for external services like the ClimaCell (a public API for weather forecast) and the actual implementation for the methods declared in the interface are located in the corresponding Service from the Infrastructure Layer. This way, when calling the methods from the ClimaCell’s interface in the corresponding Controller the unnecessarily details about the implementation of those methods are hidden.

Continuing with inheritance. Basically, inheritance is simply the redeclaration of a group of variables and functions within an enclosing scope this way expressing a “is-a” or “has-a” relationship between two objects and reusing the code of existing parent class in its child class. I used inheritance to inherit the VerifyScopes() method from the ApiControllerBase class into my custom Controllers.

Last, but not least, polymorphism. It means that one object or entity can take many other forms. It is usually of two types: static (achieved by using method overloading) and dynamic (achieved by using method overriding).Objects of different types can be accessed through the same interface and each type can provide its own independent implementation of this interface. A case where I used polymorphism in my solution was when I override the Handle function from the IRequest interface in my custom made commands classes from Application layer.

Resuming back to the SOLID principles, as I said before, they are the base of the Clean Architecture pattern. The Single Responsibility Principle states that a class should only have one reason to change. In other words, it should depend on only one user/stakeholder or use case. This is why every use case/actor from my solution has its own class or sets of classes grouped into folders named accordingly.

The Open-Closed Principle states that a software artifact should be open for extension but closed for modification, this means that it should be extensible without having to modify it (without changing the existent code of the functions, classes and modules that are contained in that artifact). This is why I have partitioned my solution’s into components that are arranged into a dependency hierarchy that protects higher-level components from changes in lower-level components.

The Liskov Substitution Principle states that in order to build a software system made of reusable and interchangeable components, those components need to adhere to a contract that allows those parts to be substituted one for another.

Objects of sub-classes type should be able to be interchanged with objects of their super-classes without any problems.

The Interface Segregation Principle states that classes should not depend on interfaces they don’t use. This way, by splitting the functions and the interfaces that have too many responsibilities into smaller ones the system will not get crowded.

The Dependency Inversion Principle states that high-level modules should not depend on the code that implements low-level details.

Combining all these principles into one architectural design pattern, my solution illustrated these principles in action. It has 4 layers, the core of the solution is made from the Domain layer and the Application layer (which contains the implementation of the use cases). The other two layers have dependencies to the core but the core (which has the highest degree of abstraction and it rarely changes) does not have dependencies on the outside layers. The Infrastructure layer and the WebUI layer only depend on the Core and not on each other.

By taking into account the SOLID principles, the final product will be one that tolerates change, is independent of external frameworks and data providers, is easy to debug and maintain, testable, independent of the database and UI and its components can be interchanged across multiple software systems. For example, in the future maybe the Client (which is now an Angular 11 app) needs to be changed or the ClimaCell external provider brakes and needs to be changed the whole application would not just crash and those changes can be made pretty easily and quite fast. The solution actually went through a phase or redesign and because I had taken into consideration the SOLID Principles and the OOP Principles the refactoring went very smooth and nothing broke.

Use Case

Estimated time: 30 min.

Actual time: 15 min.

I will give an answer that somehow incorporates all the questions. I don't know how to optimize the in-memory search engine part, but regarding the queries I would make sure that there are non-cluster indexes on the database table columns that are highly interrogated (for example a non-cluster index for the column that holds the car's brand/name <e.g. Audi a4> and another one in the column that holds the status of the car <e.g. for sale> from the table Cars or if there are two different tables, one that stores cars and another that has records about cars that are for sale then an non-cluster index in both of those tables for the column that has the car's brand/name.).

Another way is to group the queries that modify data into one module in the backend and there to be a dedicated database that stores those entries that only get modified (I'm talking here about write operations like INSERT, UPDATE and DELETE) and on the other hand, in a separated module, to be grouped only the queries that retrieve data (only GET operations) with a dedicated database for the records that are only selected to be retrieved.

I think the use of Elastic Search engine can improve the performance as well.