Elements of Solution

NTNU 2302 / Algorithms and Data Structures

Session II • May 12, 2022

I present below some elements of solution. Bear in mind that other approaches may very well be valid. These are the solution I come up with. These are *not* necessarily the "minimum" solutions that would give a full score, neither the best solution. Let me know if you find any mistakes or if you disagree.

1 Basic Knowledge

1.1 Testing

Question Can "testing" establish the correctness of an algorithm? Why?

Solution Testing can only establish the presence of defects (i.e., bugs) not their absence. If we run a test and we observe an unexpected result, we know we have found a defect, but if a test passes, we know nothing about other test cases. In general, establishing correctness would require running an infinity of tests.

1.2 Recursion

Question Consider the following Java function. Explain (in words) what would happen (and why) if one invokes it with $n = -3$ as argument.

```
int factorial (int n) {
  if (n==0) return 1;
  return n * factorial(n-1);
}
```

Solution This program yields an *infinite recursion*. The invocation `factorial(-3)` induces the invocation `factorial(-4)`, which in turn induces `factorial(-5)` and so on and so forth. In practice however, the execution will stop when the *call stack* has exhausted its maximum size (i.e., a `StackOverflowError` in Java).

1.3 Array vs. Linked-lists

Question What is an "array" . How does it differ from a linked-list?

Solution An array is a continuous memory fragment, which is allocated to store n elements of the same size (and often of the same type t). Arrays enable a direct access (i.e., $O(1)$) to the i-th element because its address is given by:

$$address(i) = address(array) + i * size(t)$$

By contrast a linked list is made of separate blocks of memory, linked together by pointers. Accessing the i-th element therefore requires a sequential access the i first links (i.e., $O(n)$).

1.4 Quick Sort

Question Quick-sort runs in $O(n \log n)$. Is it correct to say that it also runs in $O(n^2)$. Explain your reasoning.

Solution This statement is correct. If a function f admits an upper bound g, then it admits any other function h which is itself an upper bound of g. In the case of quick sort $O(n \log n)$ is a *tight* bound, whereas $O(n^2)$ is not.

1.5 Depth-first Traversal

Question Consider the graph shown below on Figure 1. In what order will the nodes be reached during a depth-first traversal, starting from Node A, and processing neighbor nodes in alphabetical order?

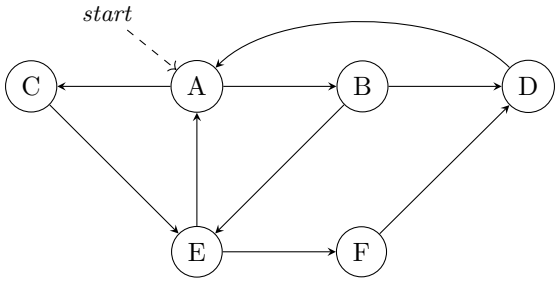

Figure 1: Sample directed graph, with 6 nodes.

Solution Starting from A and traversing children nodes in alphabetical order, a depth-first traversal reaches the nodes in the following order: A, B, D, E, F, C.

2 Algorithm Analysis

The "Caesar cipher" is an old encryption algorithm used during the Roman empire. The idea is to *shift* all letters by a fixed number along the alphabet,

either forward or backward. To encode and decode, we must know the secret key, which is the length of this shift. Here are a few examples assuming a 26 letters alphabet from 'a' to 'z',

- The word "abc" gets encoded as "bcd" when the key/shift is 1. The character 'a' becomes 'b' when shifted by 1, the character 'b' becomes 'c', and, 'c' becomes 'd'.
- The word "zoo" shifted by 2 gets encoded as "bqq". The second letter after 'z' is 'b' (we assume that the alphabet forms a circle). The second letter after 'o' is 'q'.
- The word "algorithm" gets encoded as "dojrulwkp" if the key/shift is 3. The third character after 'a' is 'd', the third character after 'l' is 'o' and so on and so forth.

The following Java function is one possible implementation of the Caesar cipher. It uses the ASCII character encoding, which preserves the alphabet ordering. For example, in ASCII, 'a' is encoded as 97, 'b' as 98, 'c' as 99, etc. In Java, converting a "char" to an "int" yields an ASCII code.

```
1   static char[] caesarCipher(char[] givenWord, int shift) {
2     char[] result = new char[givenWord.length];
3     for(int index=0 ; index<givenWord.length ; index++) {
4       int asciiCode = (int) givenWord[index];
5       int encoded = asciiCode + shift;
6       if (encoded > (int) 'z') {
7         encoded -= 26;
8       }
9       result[index] = (char) encoded;
10    }
11    return result;
12  }
```

2.1 Encoding "zone"

Question Explain how the execution unfolds given the word "zone" and a shift of 4.

Solution This algorithm iterates over the letters of the given word, that is "zone". First it allocates an array for the results. Then, it converts each letter in ASCII code, adds the given shift of 4, and if the value is beyond the ASCII code for 'z' (122 in ASCII), it subtracts 26. That gives us:

1. 'z' maps to 122 in ASCII. We add the shift and we obtain 126, which is bigger than 122 so we subtract 26 and get 100, which stands for 'd'.

2. 'o' maps to 111 in ASCII, so we obtain 115 after adding the shift. This remains below 122 and translates to 's'.

3. 'n' maps to 110 in ASCII so we obtain 114 after adding the shift of 4. This remains below 122 and translates to 'r'.

4. 'e' maps to 101 in ASCII so we obtain 105 after adding the shift of 4. This remains below 122 and translates to 'i'

As a result, the word "zone" is encoded into "dsri".

2.2 Problem Size

Question What is the *size of the problem*, that is, what drives the runtime and memory consumption.

Solution The resources needed to encode (resp. decode) result from the length of the given word. The longer the word, the more memory we need to allocate, and the more characters we need to process.

2.3 Best case

Question What is the best case scenario? What is the execution time efficiency/complexity. Explain your reasoning.

Solution The *best case* scenario demands the least resources, for a given problem size. In this case, even if we know the length of the given word (say ℓ), the total time depends on how many times we will have to subtract 26 because the shift went beyond 'z'. So the best-case is defined by words whose characters are *never* shifted beyond 'z'.

Table 1 details how the time is spent in the cipher algorithm. We assume a unit cost for assignments as well as arithmetic and logical operations. Note that in the best case, where no character is shifted further than 'z', Line 7 is never executed. In total, we see that the best case would be $time(\ell) = 8\ell + 3$, which is linear, by definition.

2.4 Worst Case

Question What is the worst case scenario? What is the execution time efficiency/complexity. Explain your reasoning.

Solution The worst case here implies that every character is shifted beyond z, that is, Line 7 is always executed. That is the only difference with the best case: The frequency count associated with Line 7 is ℓ. In total, in the worst case we obtain: $time(\ell) = 10\ell + 3$, which is also linear.

Line	Fragment	Freq.	U. Cost	T. Cost
2	`result = new char[...]`	1	1	1
3	`index = 0`	1	1	1
3	`index < givenWord.length`	$\ell + 1$	1	$\ell + 1$
3	`index++`	ℓ	2	2ℓ
4	`asciiCode = (int) givenWord[index]`	ℓ	1	ℓ
5	`encoded = asciiCode + shift`	ℓ	2	2ℓ
6	`if (encoded > (int) 'z')`	ℓ	1	ℓ
7	`encoded -= 26`	0	2	0
9	`result[index] = (char) encoded`	ℓ	1	ℓ
			Grand Total	$8\ell + 3$

Table 1: Time spent in the Ceasar cipher for the best case, with frequency counts (Freq), unit costs (U. Cost) and total cost (T. Cost). We account only for assignements, as well as arithmetic and logic operations.

2.5 Average Case

Question What is the average case scenario? What is the execution time. Explain your reasoning. (Assume that every letter is equally probable).

Solution The average case captures what we should expect if we have neither the best or the worst case. The simple answer is linear as well, because, intuitively the average between linear and linear is linear.

A more detailed answer requires making assumptions about the number of times Line 7 is executed, that is the number of characters whose shift get passed 'z'. If we denote this number by K then our model becomes:

$$time(\ell, K) = 8\ell + 3 + 2K$$

Since we do not know the value of K, we have to model it using a random variable, where each possible value is given a probability $\mathbb{P}[K = k]$. Here k ranges from 0 (the best case), to ℓ in the worst case. The average case is therefore the expected value associated with our time function for all possible value of K, that is:

$$\mathbb{E}[time(\ell, K)] = \sum_{k=0}^{k=\ell} \mathbb{P}[K = k] \cdot time(\ell, k) \tag{1}$$

For the sake of simplicity, we can assume that every value is equally probable. A more realistic approach would requires counting how many words can contain exactly k characters that are shifted. With our simplified assumption we get:

$$\mathbb{P}[K = k] = \frac{1}{\ell + 1}$$

We can now substitute this definition into Eq. 1.

$$\begin{aligned}
\mathbb{E}[time(\ell, K)] &= \sum_{k=0}^{\ell} \mathbb{P}[K = k] \cdot time(\ell, k) \\
&= \sum_{k=0}^{\ell} \frac{1}{\ell + 1} \cdot (8\ell + 2k + 3) \\
&= \frac{1}{\ell + 1} \cdot \sum_{k=0}^{\ell} 8\ell + 2k + 3 \\
&= \frac{1}{\ell + 1} \cdot \left(\sum_{k=0}^{\ell} 2k + \sum_{k=0}^{\ell} 8\ell + 3 \right) \\
&= \frac{1}{\ell + 1} \cdot \left([\ell\,(\ell + 1)] + \ [(\ell + 1)\,(8\ell + 3)] \ \right) \\
&= \ell + (8\ell + 3) \\
\mathbb{E}[time(\ell, K)] &= 9\ell + 3
\end{aligned}$$

For the record, a more accurate probability of having a sequence of characters with K whose get shifted is given by the formula below, where s denotes the shift. This however leads to more complicated calculation, beyond the scope of this exam. But this would better capture the fact that the longer the shift, the closer we move towards the worst case.

$$\mathbb{P}[K = k] = \frac{1}{26^{\ell}} \left((26 - s)^{\ell - k} \cdot s^{k} \cdot \binom{\ell}{k} \right)$$

3 Algorithm Design

Consider linked-list as shown in Figure 2, where each node points to the next one. As we build such list, it is possible to build "loops" as shown on Figure 3. We would like to design a procedure to check whether the pointers that make up the list forms a loop or not.

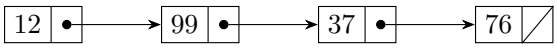

Figure 2: A "regular" linked-list.

For the sake of simplicity, we will assume the existence of the following procedure that helps manipulate the nodes of these lists:

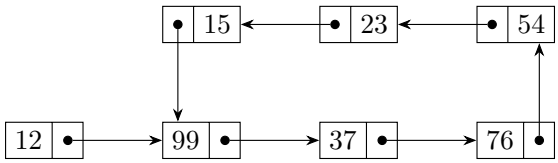

Figure 3: A "invalid" linked-list that includes a loop.

- `Node getNext (Node n) {...}` returns the "next" Node or null if it is not defined.

This task is about a procedure `boolean hasLoop (Node first) {...}`, which returns true only if the given list contains a loop.

3.1 Iterative Algorithm

Question Propose an "iterative" algorithm to detect such a loop. Feel free to add information to the node structure, if you feel it helps.

Solution One way to solve this problem is to use a hash table to store the nodes that we have already seen. We traverse the nodes in order and for each node, we check in this hash table if we have already seen it, if not, we add it, and we continue with the next node.

```
1   boolean hasLoop(Node first) {
2     var visited = new HashSet<Node>();
3     var current = first;
4     while (current != null) {
5       if (visited.contains(current)) return true;
6       visited.add(current);
7       current = getNext(current);
8     }
9     return false;
10  }
```

3.2 Time Complexity

Question What is the time efficiency of your algorithm? Explain your reasoning.

Solution The problem size is defined the length of the list n. The longer the list the more node have to be checked.

We assume a unit cost for arithmetic operations, logic operations and assignments. Since our algorithm relies on a hash table, we know that insertion and search both takes $O(1)$. Table 2 details the unit cost, frequency count of each line of our algorithm. We obtain a total of $time(n) = 4n + 3$.

Line	Fragment	Freq.	U. Cost	T. Cost
2	`visited = new HashSet()`	1	1	1
3	`current = first`	1	1	1
4	`current != null`	$n+1$	1	$n+1$
5	`visited.contains(current)`	n	1	n
6	`visited.add(current)`	n	1	n
7	`current = getNext(current)`	n	1	n
			Grand Total	$4n+3$

Table 2: Time spent in the iterative algorithm, with frequency counts (Freq), unit costs (U. Cost) and total cost (T. Cost). We account only for assignments, as well as arithmetic and logic operations.

3.3 Space Complexity

Question What is the space efficiency of your algorithm? Explain your reasoning.

Solution To estimate the memory consumption we can identify the place in the program, where we *allocate* memory. There is only one, on Line 2 `visited = new HashSet<>()`. Basically, the hash table will contain all the node of our list, whether it forms a loop or not. So we can conclude that $space(n) = n$, which is linear.

3.4 Recursive Algorithnm

Question Convert your "iterative" algorithm into a "recursive" one.

Solution We can create a "recursive" algorithm by observing that processing a node is the same as processing a list of node: We process the first element and then remaining list. We however have to pass the hash table in every recursive calls.

```
1   boolean hasLoop(Node first) {
2     return doHasLoop(first, new HashSet<Node>());
3   }
4
5   boolean doHasLoop(Node current, Set<Node> visited) {
6     if (current == null) return false;
7     if (visited.contains(current)) return true;
8     visited.add(current);
9     return doHasLoop(getNext(current), visited);
10  }
```

3.5 Recursive Space Efficiency

Question Recursive algorithms leverage the "call stack" to store parameters of each active calls. What is the space efficiency of your recursive solution? Explain your reasoning.

Solution The recursive algorithm above also uses a hash table, which is carried along the recursive calls. So the call stack has to allocated at least two "cells" per call, one to store the `current` node, and one for the hash table. Besides, there will be one call for each entry in the list, so that given a total of $space(n) = 3n$, which is linear as well.

We note that function is tail-recursive, so a compiler could possibly suppress the call stack, which would yield the time complexity than the "iterative" algorithm.

4 Problem Solving

We would like to implement a container data-structure, which is defined by an *absract data type* (ADT). This structure, denoted by $\mathbf{s}$, can contain an arbitrary number of elements of type T T can be any thing—think of a generic type in Java for example. T does not make a difference for the rest of this exercise. Our ADT defines the following operations:

1. Insertion. We should be able to add an element x to any given structure $\mathbf{s}$, but only once. In other words, our structure forbids duplicates. Formally, we could write:

$$\begin{aligned} &insert : S \times T \to S \\ &\quad insert\,(\mathbf{s}, x) = \mathbf{s} \cup \{x\} \end{aligned}$$

2. Deletion. We should be able to remove an element x from any given structure $\mathbf{s}$. Formally, we could specify that:

$$\begin{aligned} &delete : S \times T \to S \\ &\quad delete\,(\mathbf{s}, x) = \mathbf{s} \ - \{x\} \end{aligned}$$

3. Range queries. We want to find all the elements from $\mathbf{s}$ that lays in between the two given elements x and y: Formally, we could write:

$$\begin{aligned} &between : S \times T \times T \to S \\ &\quad between\,(\mathbf{s}, x, y) = \{ z \mid \ z \in \mathbf{s} \ \ \wedge x \leq z \leq y \ \} \end{aligned}$$

4.1 Choosing Data Structure

Question What data-structure would you choose to implement this ADT?

Solution I would choose a binary search tree (BST): It preserves the ordering of items and enable insertion, access and deletion in logarithmic time (in average). BST works well also to ensure the set property and avoid duplicates.

The downside of BST is that they require more space than a sorted array for example. Here I favor speed over space, but this is my own choice.

4.2 Insertion

Question Propose an algorithm to implement the "insert" procedure.

Solution The insertion is basically the same as in a binary search tree. I present here excerpt of a Java class that stores arbitrary data type using a generic type `Item` but the principles would be the same if I was to store only integers.

```
1   public class Tree<Item extends Comparable<Item>>  {
2
3     private final Item item;
4     private Tree<Item> left;
5     private Tree<Item> right;
6
7     // ... Constructor and other methods
8
9     public Tree<Item> insert(Item givenItem) {
10      int difference = this.item.compareTo(givenItem);
11      if (difference > 0)  {
12        if (left == null) {
13          this.left.insert(givenItem);
14
15        } else {
16          this.left = new Tree(givenItem);
17
18        }
19        return this;
20
21      } else if (difference < 0) {
22        if (right == null) {
23          this.right.insert(givenItem);
24
25        } else {
26          this.right = new Tree(givenItem);
27        }
28        return this;
29
30      } else {
31          throw new RuntimeException("Duplicated item " + item);
```

```
32          }
33        }
34
35      }
```

4.3 Time Efficiency of Insertion

Question What is the time efficiency of your insertion procedure? Explain your reasoning?

Solution The insertion algorithm given above is a recursive function. There are two cases. Either the tree has no children yet and the insertion is immediate $O(1)$, or the tree has children and the insertion is delegated to the children, in which case the time depends on the depth of the tree. In the worst case, the tree looks just like a linked list and time would be $O(n)$. In average, however, the tree is fairly balanced and its depth of the tree implies a runtime in $O(\log n)$.

4.4 Range Queries

Question Propose an algorithm to evaluate "range queries", that is to find all the elements in between two values.

Solution A possible solution to the "range query" problem is to traverse the BST and to only select those nodes that fall within the selected range. In the following, I represent the range by its lower and upper bounds. All the items that fits are placed into the `bag` collections given as input.

```
1   public void collectRange(Item lowerBound,
2                            Item upperBound,
3                            Collection<Item> bag)
4   {
5     var diffWithLower = item.compareTo(lowerBound);
6     var diffWithUpper = item.compareTo(upperBound);
7     if (diffWithLower < 0 || diffWithUpper > 0 ) {
8       return;
9     } else {
10      bag.add(item);
11      if (left != null)
12        left.collectRange(lowerBound, upperBound, bag);
13      if (right != null)
14        right.collectRange(lowerBound, upperBound, bag);
15    }
16  }
```

4.5 Time Efficiency of Range Query

Question What is the time efficiency of your "range queries" procedure? Explain your reasoning.

Solution As for the time-efficiency of the above algorithm, in the worst-case, we never enter the first conditional statement (Line 7). This occurs when the whole tree lays within the given bound. In that case, the algorithm has to traverse every single node and the time complexity is therefore $O(n)$ where n is the number of items in the tree. In the best case, the whole tree is outside of the selected interval and the algorithm takes constant time (i.e., $O(1)$). In average it depends on how many elements from the tree lays within the selected interval.