Final Examination 2021—Solution

Algorithms & Data Structures

IDATA 2302

Dec. 2021

1 Basic Knowledge

1.1 Binary Search

Question Explain why does a binary search runs in logarithmic time (i.e, in $O(\log n)$ where n represents the length of the array with search in).

Solution The binary search proceeds with halving the array until it finds either the desired value or an empty array. The number of time one can split an array of length n in two is given by $\log_2 n$.

1.2 Quadratic Runtime

Question An algorithm takes 1 ms to solve a problem of size 100. What is the biggest problem size that can be solved in 1 s if its runtime efficiency is $\Theta(n^2)$?

Solution If the algorithm is in $\Theta(n)$, by definition, we know that there exist a constant c, such that $time(n) = c \cdot n^2$. Since we know that it takes 1 ms to solve a problem of size 100, we thus derive that:

$$1\,ms = c \cdot 100^2$$
$$\frac{0.001s}{100^2} = c$$

We can apply the same definition when the algorithm works for 1 s, that is:

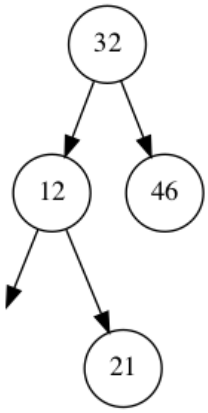
$$
\begin{aligned}
1 &= c \cdot n^2 \\
1 &= \frac{0.001}{100^2} n^2 \\
\frac{100^2}{0.001} &= n^2 \\
3162 &\approx n
\end{aligned}
$$

1.3 Binary Search Tree

Question What binary search tree results from the following. Which element will end at the root of the tree? Why?

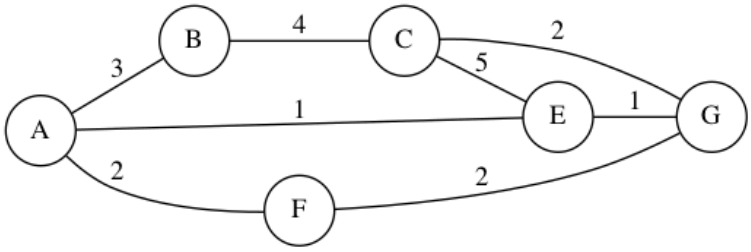
```
var tree = new BinarySearchTree();
tree.insert(25);
tree.insert(45);
tree.insert(46);
tree.insert(32);
tree.delete(25);
tree.insert(12);
tree.delete(45);
tree.insert(21);
```

Solution The picture below shows the tree that results from the sequence of operations. The first insertion creates a unique node with value 25 (the root). 45 then comes as its right child. 46 as the right child of 45, whereas 32 comes as its left child. When we delete 25, it *gets replaced by its successor*, which is 32. We then insert 12, and it comes in as the left child of 32. When we delete 45, 46 becomes the right child of 36. Finally, when we insert 21 it gets inserted as the right child of 12.



1.4 Minimum Spanning Tree

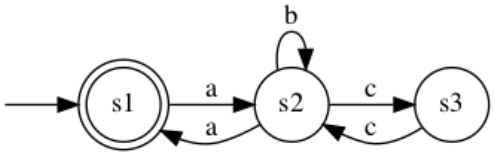
Question Looking at the following graph, find one minimum spanning tree. Describe the tree as a list the edges which you would include, e.g, $(A, B), (B, C)$ and (C, D).



Solution A minimum spanning tree is a tree that includes all the graph vertices and, which minimizes the total weight of its branches. We can build it using Prim's algorithm, which chooses the edge with the lowest edge that expand sour tree until all vertices are included. One solution is $(A, E), (E, G), (G, F), (G, C), (A, B)$.

1.5 Automata

Question What is the transition table underpinning the following finite state automaton?



Solution The transition table is given by the following transition table with current state, input and resulting state.

$$s_1, a \rightarrow s_2$$
$$s_2, a \rightarrow s_1$$
$$s_2, b \rightarrow s_2$$
$$s_2, c \rightarrow s_3$$
$$s_3, c \rightarrow s_2$$

2 Algorithm Analysis

The LOGO language aims at providing children with a first "programming" experience. LOGO programs run on a turtle-like robot, which obeys basic instructions such as `FORWARD`, `BACKWARD`, `LEFT`, `RIGHT`, etc. Children can thus write short programs and see visually whether their turtle moves as they expect. The robot turtle also embeds a pen which can touch the ground and therefore draw a line as the turtle moves forward.

We look here at a program written in Java against a possible LOGO API. This API defines a `Robot` class equipped with the following methods:

- `forward(int distance)` The robot moves forward by the given distance (in cm). It takes a time proportional to the given distance, denoted by $time_F(d) = d$;
- `backward(int distance)` The robot moves backward by the given distance (in cm). It takes a time proportional to the given distance, denoted by $time_B(d) = d$.
- `turnLeft(int angle)` The robot spins on the left by given angle (from 0 to 360). This takes a time proportional to the given angle, denoted by $time_L(a) = a$.
- `turnRight(int angle)` The robot spins on the right by the given angle. This takes a time proportional to the given angle denoted by $time_R(a) = a$;

- `penUp()` The robot lifts the pen up so that it does not touch the ground anymore. This has no effect if the pen is already up and takes a constant amount of time denoted by $time_U$.
- `penDown()` The robot puts down its pen so that it marks the ground. This has no effect if the pen is already down and takes a constant time $time_D$.

A friend suggests the following program to draw regular polygons (i.e., triangle, square, pentagon, etc.) using the previous API. This exercise looks at the efficiency of this algorithm—regardless of its imperfection. Because the time spent performing arithmetic operations is negligible compared to the time spent moving the robot, we assume that arithmetic operations, logical operations and memory accesses all take 0 time.

```
1 void drawPolygon(Robot turtle, int sideCount, int sideLength) {
2     var angle = 360 / sideCount;
3     turtle.penDown();
4     for (int side=0 ; side <= sideCount ; side++) {
5        turtle.forward(sideLength);
6        turtle.turnLeft(angle);
7     }
8     turtle.penUp();
9 }
```

2.1 Square of length 10

Question How much time does it takes for this program to draw a square with 10 cm sides (i.e., invocation `drawPolygon(turtle, 4, 10)`).

Solution We only need to account the methods listed in the given API, since the other take zero time. That leaves us with:

- `penDown` line 3, runs only once and take $time_D$.
- `turtle.forward(10)`, line 5, takes $time_F(10) = 10$ and runs 5 times.
- `turtle.turnLeft(90)`, line 6, takes $time_L(90) = 90$ and runs 5 times
- `turtle.penUp()`, line 8, takes $time_U$ and run once.

That gives us a total $time(4, 10)$ of:

$$\begin{aligned} time(4, 10) &= time_D + 5 \times (90 + 10) + time_U \\ &= time_D + 500 + time_U \end{aligned}$$

The factor 5 indicates an possible improvement. This algorithm actually draws 5 sides when it paints a square (only four are needed). See the loop conditional which could (and should) include a strict comparison operator.

2.2 Problem size

Question Which parameters govern the time the robot takes to draw any given polygon, and why?

Solution The parameters that affect the time the robot takes to draw an arbitrary polygon are the number of side and length of the side. The number of side impact the duration, because the more sides, the longer is the perimeter of the polygon. The length of the side affects the duration, because the longer the length the longer it takes for the robot to cover the distance (the time of `forward` is proportional to the distance).

2.3 Time estimation

Question Estimate the time the robot needs to draw a given polygon whose number of side is s, and the side length is ℓ. Could you suggest an improvement?

Solution We can expand the solution we found for the duration of the square of length 10, as follows:

- `penDown` line 3, runs only once and takes $time_D$.
- `turtle.forward(10)`, line 5, takes $time_F(\ell)$ and runs $s+1$ times
- `turtle.turnLeft(90)`, line 6, takes $time_L(\frac{360}{/}s) = \frac{360}{s}$ and runs $s+1$ times
- `turtle.penUp()`, line 8, takes $time_U$ and runs once.

That gives us a total $time(s, \ell)$ of:

$$time(s, \ell) = time_D + (s+1) \times (\frac{360}{s} + \ell) + time_U$$

Our analysis in the first question revealed a possible improvement: Draw only s segments instead of $s+1$, which reduces the total time to

$$
\begin{aligned}
time(s, \ell) &= time_D + s \times (\frac{360}{s} + \ell) + time_U \\
&= time_D + 360 + s \cdot \ell + time_U
\end{aligned}
$$

2.4 Big-O Notation

Question Prove that your time estimate admits an upper bound such that $time(s, \ell) \in O(s \times \ell)$.

Solution From the previous algebraic expression, if we discard the constant factors, we are left with

$$
time(s, \ell) = s \cdot \ell
$$

Which is, by definition, $O(\ell \times s)$.

2.5 Tighter Bounds

Question Given the API of this robot, do you think there exists an alternative algorithm that would admit a tighter upper bound? Why?

Solution I don't think there is a faster algorithm, because I do not see how one could draw a polygon without drawing every side. Besides, the motion of the robot seems optimal: Drawing the sides in any other order would incur additional time to move from one segment to the next.

3 Algorithm Design

We now turn to the "rod cutting problem" where we are given a long piece of wood (the rod) which we must chop into segments. The particularity is that segments of different length are sold at different prices, and we would like to maximize our revenue. The problem is thus the following: Given the rod of a specific length, and given the selling prices of various length, how should we cut this rod to make as much money as possible?

Let us consider for example that we are given a rod of 4 meters, and that we can sell segments for the prices shown in the table below. Given these length and prices, we could cut the rod as follows:

- 4 segments of 1 meter, which would yield a profit of $4 \times 1.5 = 6$ kr.
- 2 segments of 1 meter, and 1 segment of 2 meter, which would yield $2 \times 1.5 + 4.5 = 7.5$ kr.
- 2 segments of 2 meters, which would yield a revenue of $2 \times 4.5 = 9$ kr.
- 1 segment of 3 meters and 1 segment of 1 meter, which would yield a revenue $6.5 + 1.5 = 8$ kr.
- 1 segment of 4 meters for 7 kr.

Length (m)	1	2	3	4
Selling price (kr)	1.5	4.5	6.5	7

3.1 Greedy Approach

Question The following listing uses a "greedy" approach: It always maximizes the price per meter. In the previous example, we would always choose the segment of length 2, and then one segment of length 1 if the given length permits it. Show that this approach does not always yield the highest possible revenue, by giving a scenario (prices and total length) where it actually fails.

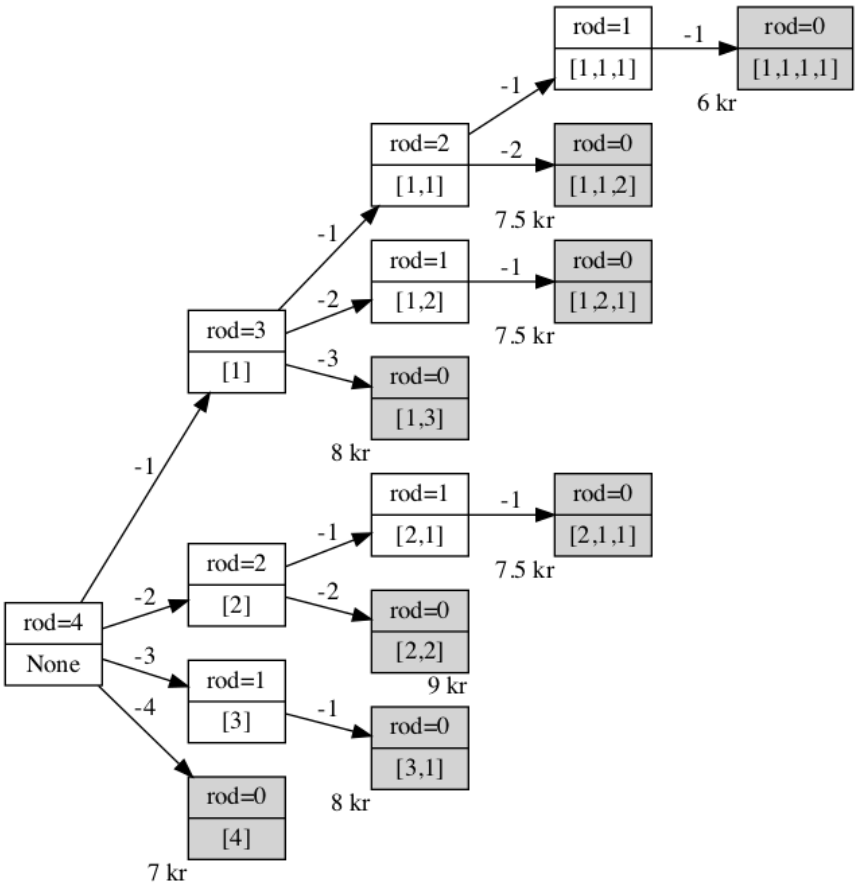
```
List<Integer> greedySolution(int rodLength, int[] prices) {
    List<Integer> cuts = new ArrayList<Integer>();
    var remaining = rodLength;
    while (remaining > 0) {
       var newCut = findHighestPricePerMeter(prices, remaining);
        cuts.add(newCut);
        remaining -= newCut;
    }
    returns cuts;
}
```

Solution The greedy approach fails when choosing the highest price per length leads to a "bad" situation. For example, given following prices and a rod of length 10 meters, the best solution is to cut three segment of 3 meters for a revenue of 9 kr. The greedy approach would however chooses to cut 2 segments of 4 meters, for a value 8.5 kr, but would be left with 2 meters of rod that do not worth anything.

Length (m)	1	2	3	4
Selling price (kr)	0	0	3	4.25
Price / meter	0	0	1	1,06

3.2 Combinatorial Search

We propose to use a *combinatorial search* to enumerate all possible ways to cut the given rod, and find the most profitable one. We start with the original rod, and we explore all the possible segments we can cut. The following illustration portrays the associated search tree where each box denotes a configuration with nnits remaining rod length on top and the list of segments cut so far below.



Question What strategy would you use to prune this search tree, and in turn, find the most profitable strategy faster?

Solution We could use *branch-and-bound* to prune the search tree and avoid engaging in branches that are not promising. To do that we could define a heuristic that would estimate the best possible outcome given the remaining length of rod. Given this estimate we could then skip branches for which the estimate is lower than the best solution we found so far.

3.3 Self-similar Sub Problems

Question Can you see any *self-similar sub problems* in this tree? If yes, where and why?

The tree do contain self-similar sub problems. For example, while finding the best way to divide a rod of 4 meters, once we have chopped a 1 meter segment, we need to find the best way to divide the remaining rod. This is the very same problem, but with a smaller rod.

3.4 Recursive formulation

Question Give a recursive algorithm to solve this *rod-cutting problem*. The recurrence formula is enough.

Solution Following the sub problem that we have identified previously, we can formulate the solution as follows. Finding the best cut for a given rod r, is selecting among all possible cuts, the one whose price added to the highest revenue yielded by the remaining rod is maximal.

$$rodCut(r, S) = \begin{cases} 0 & \text{if } r \leq 0 \\ \max_{s \in S} price(s) + rodCut(r - s, S) & \text{if } r > 0 \end{cases}$$

where r represents the length of the rod to cut, S the various segments size sold on the market, and $price(s)$ the revenue yielded by selling a segment of size s.

3.5 Dynamic Programming

Question How would you improve the runtime efficiency of your recursive solution? Sketch a solution and motivate your decision.

Rod Length	0	1	2	...	r
Best Revenue	0				

Solution One way to improve the performance of such a recursive algorithm is to use dynamic programming. The idea of dynamic programming

is to replace the recursion with a table, which we fill starting from the base case(s) of the recursive formulation.

The following Java-like listing illustrates this idea.

```
int rodCut(int length, int [] prices) {
    int[] revenue = new int[length+1];
    revenue[0] = 0;
    for (int i=1 ; i<length ; i++) {
        int bestRevenue = 0;
        for (int j=0 ; j<prices.length ; j++) {
            var remaining = i - j;
            if (remaining >= 0) {
                newRevenue = prices[j] + revenue[remaining];
                if (bestRevenue < newRevenue) {
                    bestRevenue = newRevenue;
                }
           }
        }
        solution[i] = bestRevenue;
    }
    return revenue[length];
}
```

That would give us a solution in $O(\ell \times |P|)$ where ℓ is the length of the rod and $|P|$ the length of the price table.

4 Data Structure Design

Finally, let us look at the *two-sum* problem. Given an array of integer values, find the pair of values that sums up to a given value. For example, provided with the array $[1, 3, 6, 4, 2]$ and 10 as a target, the algorithm outputs $(6, 4)$. By contrast, provided with $[4, 2, 3, 9, 8]$ and 19, the algorithm raises an error since there is no pair that sums up to 19.

4.1 Brute Force

Question Propose an algorithm that uses a "brute force" approach.

Solution A brute force approach consists in building all possible pairs of element available in the given array and checking if any pair sums up to the desired number. The following Java-like code illustrates this idea:

```java
int findPair(int[] array, int target) {
    for (int i=0 ; i<array.length ; i++) {
       for (int j=0 ; i<array.length ; j++) {
          if (array[i] + array[j] == target) {
              System.out.println(array[i] + " + " array[j]);
              return;
          }
       }
    }
    System.out.println("No solution.");
}
```

4.2 Worst-case Complexity

Question What is its worst-case runtime efficiency? Explain your reasoning.

Solution If n is the length of the given array, there are basically n^2 possible pairs of elements. In the worst case, none of these pairs adds up to the desired number and the algorithm has to check every one of these n^2 pairs. That is therefore $O(n^2)$.

4.3 Data structure

Question What data structure can help outperform this brute-force approach, and why?

Solution One could use a *hash table* to remember the values that have already checked. Thus, when we come to a new value, we could simply check that the difference between the target and the current value is not in the hash table.

4.4 Better Algorithm

Question Describe an algorithm that leverages this data structure.

Solution The following Java-like code gives an idea of the solution

```java
int findPairFaster(int[] array, int target) {
    var table = new HashSet<Integer>();
    for (int i=0 ; i<array.length ; i++) {
       if (table.containsKey(target-array[i])) {
          System.out.println(i + " + " + target-array[i]);
          return;
       }
```

```
            table.add(array[i]);
        }
        System.out.println("No solution.");
    }
```

4.5 Runtime Efficiency

Question What runtime efficiency do you obtain? Explain your reasoning.

Solution Using a hashtable as shown above, we only need to traverse the array once, which yields a linear runtime efficiency. $O(n)$.

End of the examination