University of Bergen

INF237

Algorithm Engineering

Personal notes from the problem solving

Author:
Oskar Leirvåg (OLE006)

April 19, 2020

Contents

Abstract

This document is meant to give an insight in how the algorithms were designed. The intention is to give a deeper explanation to the thought process in the hopes of helping others understand while also helping myself to remember how it was done. It is important to mention that this course does require an oral test at the end of the semester and this text will help me in the preparation for this.
This course uses Kattis as a tool to publish the weekly challenges, and also test submissions. The algorithms that are submidted are thurougly tested over many hidden tests to affirm that they do fulfill all requirements. In order to pass this course a given number of problems must be completed each week, and they usually hold a theme in common.

Assignment 1: Ad-hoc

The first week was obviusly only a few simple problems as to get to know the Kattis tool. I used this time to experiment with some different programming languages like Java, C++ and Haskell.

Taks one: Oddities

Programing language: Java *Difficulty: Beginner* *Est: 0.5 hrs.*

This problem was really straight foreward. Given the input of any random number n of a given size, write if the given number is *odd* or *even*. So I guess there is little to say about the logic here. Could possibly have been solved even faster in Haskell than java.

Taks two: Different Problem

Programing language: Haskell *Difficulty: Beginner* *Est: 0.5 hrs.*

This problem simply gives you two numbers a and b and wants the output of the **absolute distance**. This is solved quite simply in Haskell with the recursive function

```
solve :: [Integer] -> [Integer]
solve [] = []
solve (a:b:xs) = abs(a - b):(solve xs)
```

Taks three: Guess the number

Programing language: C++ *Difficulty: Easy* *Est: 2 hrs.*

This is a simple simulation of the old guessing game. The computer decides a number $1 \leq n \leq 1000$, and your algorithm should be able to guess the correct number within 10 questions given an honest responce of higher/lower/correct. This has to be done by a binary search.

Taks four: Turtle master

Programing language: C++ *Difficulty: Easy* *Est: 3 hrs.*

In this problem we are given a game-board with a robot, a goal and some obsticles. Then we are given inputs (commands) for the robot to follow, and wee shall simply simulate the result. Then if all criteria is met we should return either *Diamond!* if correct, or *Bug!* if any error/illegal move occurs.

The short solution to this problem was simply a full on object oriented approach. I simply made a Position object and read from the map if any new positions was valid. Now finally if the robot was standing on the goal in the end, print *"Diamond!"*. Now this took a lot of extra time due to me not knowing C++ and how to use pointers.

Assignment 2: Graphs 1

Reversing Roads

Programing language: Java Difficulty: Medium Est: 5 hrs.

This problem could be really easy if we should only test if it was valid or not. Unfortunately we also need to test if it can be solved by flipping one single road.

To store the graph i made an adjacency-matrix, but since those usually take a lot of memory it had to be compressed, that's why I used BitSet. I read the input and built the matrix. Right after we can do shorter validations to remove some simpler solutions like if there exist only one house, or if there are not enough roads to be mathematically feasible.

Now using a DFS i test the solution. If it's valid then output *"valid"* and terminate, if not we iterate over all roads, flip them, and try again, if now valid then output the road flipped and terminate. If none is valid output *"invalid"* and terminate.

The Maze Makers

Programing language: Java Difficulty: Medium Est: 5 hrs.

Solving this took quite a while. The question is overly complicated and confusing with the binary part resulting in a lot of programming.

I solved the problem by using a DFS algorithm visiting all possible nodes counting them as we go starting from the first hole found in the wall. If any was already visited it needed to store that. Due to the problem having a priority list of reported errors; it was necessary to store errors occurring and not terminate.

To traverse the graph it was necessary to interpret the different symbols. Since they had a binary translation their char values could be converted to the integer value, from which we could read the walls.

Assignment 3: Sliding, searching and sorting

Free Weights

Programing language: C++ *Difficulty: Easy* *Est: 3 hrs.*

Now this problem can be done by simply removing weights smaller than a given weight X and testing if the solution is valid. The problem with this is that it would take quite a while if there are many different weights.

To combat the issue of many different weights we can use the same solution, but instead of as linear search, we do a binary search.

Now this solution is not necessarily the optimal due to the use of *std::copy_if* to filter, which could be skipped and just added to the loop instead.

Room painting

Programing language: C++ *Difficulty: Easy* *Est: 5 hrs.*

This problem is really easy and I used a lot of unnecessary time due to a simple mistake of using a 32 bit integer instead of a 64 bit resulting in failure.

To solve this, one could sort the shop-list, iterate over all the wanted-list, and binary search for the closest bucket. For each iteration add the difference of the closest bucket found.

Now an even faster solution here is to sort both lists and use a linear search while storing the position of the last found.

```
for (int i = 0; i < m; i++)
   for (int j = lastTaken; j < n && n > 0; j++)
     if (wanted[i] <= possible[j]) {
       waste += possible[j] - wanted[i];
       lastTaken = j;
       break;
     }
```

Assignment 4: Dynamic programming 1

Train sorting

Programing language: C++ *Difficulty: Medium* *Est: 8 hrs.*

This problem took a while due to trying an invalid solution. The problem can be solved by using *Longest Decreasing Subsequence* plus *Longest Increasing Subsequence*. Hence in a single loop we find both *lis* and *lds*, then we iterate over all positions of the trains and find the maximal solution -1 so we dont add the middle cart twice.

Now it's important that we find both the *lis* and *lds* backwards, since we will pick up the carts or drop them, as we cannot go backwards.

```
int res = 1;
 Arrays.fill(lds, 1);
 Arrays.fill(lis, 1);
for (int i = a.length - 2; i >= 0; i--)
   for (int k = a.length - 1; k > i; k--) {
     if (a[k] < a[i] && lds[i] < lds[k] + 1)
       lds[i] = lds[k] + 1;
     else if (a[k] > a[i] && lis[i] < lis[k] + 1)
       lis[i] = lis[k] + 1;

     res = Math.max(res, (lis[i] + lds[i]) - 1);
   }
 return res;
```

Chemist vow

Programing language: C++ *Difficulty: Medium* *Est: 5 hrs.*

The most annoying part of this was to order the possible elements and add them into the lists. Although the problem is not too easy itself, but can be solved efficiently with a recursive branching algorithm.

It must be stated that a non optimal solution can still test all known words from the English alphabet in reasonable time, but this specific problem requires an efficient solution which can complete longer tests.

To solve this one iterates over each letter and tests if there exist a valid solution for either *letter[i]* or *letter[i] + letter[++i]* recursively. The important part here is for a branch to store if a given position is invalid so to stop later branches from continuing from that point.

Assignment 5: Graphs 2

Gruesome Cave

Programing language: Java Difficulty: Hard Est: 10 hrs.

This problem can be solved by finding the minimum path from entry to the diamond, if we use the cost as the probability of the monster being inside that given node.

Calculating the probability is possibly the most challenging part of the entire problem. By statistics we can say that since it's a connected graph we can say that each cells probability of the monster standing in it (or its cost) can be calculated by its degree (number of connected edges) divided by the total risk.

Landline Telephone Network

Programing language: C++ Difficulty: Easy Est: 3 hrs.

This problem asks for what is essentially a minimum spanning tree or *MST*. The main difference here is that there are certain nodes required to be leaf-nodes.

To solve this I used a Kruskal approach, but before connecting the nodes I sort the nodes (or rather edges) into either *"good"*, *"bad"* and *"superbad"*; where the nodes that connect two bad nodes are just removed.

Now since Kruskal works on edges and with Union Find we can just create an *MST* from the good nodes. Now if at this point the unconnected houses is a number greater than bad houses there is no solution. Else we continue by adding the bad edges to the pool and continue Kruskal. Now we have a ternary output depending on the size of the union being one.

Assignment 6: Segment trees

Supercomputer

Programing language: Java Difficulty: Easy Est: 0.5 hrs.

This problem is intended to be solved with a Segment Tree, but since it consist of only binary numbers it's possible to just use a single BitSet at the cost of efficiency.

```
import java.util.BitSet;
public class Supercomputer {
   public static void main(String[] args) {
      Kattio kattio = new Kattio(System.in, System.out);
      int n = kattio.getInt(), k = kattio.getInt();
      BitSet maybeMagic = new BitSet(n);
      for (int i = 0; i < k; i++)
            if (kattio.getWord().equals("F"))
               maybeMagic.flip(kattio.getInt());
            else System.out.println(maybeMagic.get(kattio.getInt(),
               kattio.getInt() +      1).cardinality());
   }
}
```

Turbo

Programing language: C++ Difficulty: Easy Est: 2 hrs.

Turbo is almost the same problem as Supercomputer, but had to be solved properly with a Segment tree due to time constraints. Here the problem specifies where it should flip bits and what to query.

Also this is slightly tricky as one needs to keep track of some pointers going back and forth from max to min.

```
for (int i = 0, min = 0, max = 0; i < n; i++) {
   int moveTo = i % 2 == 0 ? 0 : n - 1;
   int moveFrom = i % 2 == 0 ? nums[min++] : nums[n - 1 - max++];
   st.update(moveFrom, 0);
   var res = st.query(Math.min(moveTo, moveFrom), Math.max(moveTo,
      moveFrom), 0);
   System.out.println(res);
 }
```

Assignment 7: Geometry 1

Counting Triangles

Programing language: C++ Difficulty: Medium Est: 5 hrs.

This problem has some fascinating aspects which can be greatly simplified using linear algebra. As we need to use an almost DFS algorithm to iterate over all unique permutations of three line-segments given, and test if they all intersect. This itself is rather easy, but the method required to test if two line-segments intersects can be complex.

First we store the line-segments as a vector v and a point p. To test if they intersect we can solve the linear equation given by this equation and test if x and y is both between 0 and 1. Where a and b are the vector-points of the line-segment and c are the vectors between the points.

$$\begin{vmatrix} a_1x + b_1y = c_1 \\ a_2x + b_2y = c_2 \end{vmatrix}$$

Since this equation set is of the format $Ax = b$ it can be solved simpler by using Cramer's rule. Giving the following function.

```
bool interSects(LineSegment* other) {
    double M = (-other->v->dx * this->v->dy + this->v->dx * other->v->dy);

    double s = (-this->v->dy * (this->p->x - other->p->x)
    + this->v->dx * (this->p->y - other->p->y)) / M;
    double t = (other->v->dx * (this->p->y - other->p->y)
    - other->v->dy * (this->p->x - other->p->x)) / M;

    return 0 <= s && s <= 1 && 0 <= t && t <= 1;
}
```

Jabuke

Programing language: C++ Difficulty: Easy Est: 2 hrs.

This problem is a wider know problem and is solved quite fast with math (geometry). We can calculate the angle of a vector AB to a pint P by a new vector AP. But with linear algebra we can create the simpler function.

```
public boolean pointIsLeftOrOn(Point test) {
    return (b.x - a.x) * (test.y - a.y) >= (b.y - a.y) * (test.x - a.x);
}
```

Now we can know it the point is to the left of any vector we simply test if the point is to the left of all vectors forming a triangle. The main problem now is that the points can be given in a reverse order, where we need to reverse the triangle or test *pointIsRightOrOn()*. To circumvent this we can simply use the given formula for the areal. As that would be negative if reverted. Now we can rotate the triangle if necessary.

Assignment 8: Exponential time algorithms

Coloring Graphs

Programing language: C++ Difficulty: Hard Est: 8 hrs.

Graph coloring is a well known NP-hard problem. This task is solved by testing all possible coloring combinations, resulting in a k^n runtime. This can be greatly improved by using dynamic programming and backtracking to a runtime of 2^n. This is solved with recursive branching testing all possible solutions and aborting if invalid.

Assignment 9: Dynamic programming 2

Rebel Portals

Programing language: Java Difficulty: Hard Est: 8 hrs.

Rebel portals is a famous problem in algorithms called *The travelling salesperson problem* (another NP-hard problem) but the possibility of using portals adds an entire new complexity. The problem is a fully connected graph, and finding the optimal solution requires brute-force of $n!$. This problem can be solved faster with dynamic programming by using backtracking. The following formula shows how this is done;

$$g(i, S) = \min_{k \in S}\{c_{i \to k} + g(k, S - \{k\})\}$$

In my solution I use a bottom up recursive branching backtracking algorithm, branching from a planet to all other planets. This is done by passing a the S subset of unvisited planets further down, and branching to all planets in S at each stage.

To solve the problem of the portals, we also need to pass the portal status of all planets down the branching, and if the portal is not open on the current planet we need to branch to all other unvisited planets with an open portal as well. Finally all leaf nodes should simply return to the home planet 0 and we can return the distance cost of either the distance home, or 0 if wee can use a portal.

Now keeping track of open portals and each subset will require a harsh data structure using either a lot of memory, or being slow. To solve this, I chose to represent these as binary arrays in integers. Accessing bits can be done by testing if $\{(S \oplus (1 << i)) > 0\}$

To fully make this dynamic we also need to memoize. We will visit a state with the corresponding planet, subset and portal-state multiple times, and this can be stored. The problem is that the combination of both the subsets and portals is 2^n or $(1 << n)$ resulting in $n * 2^{2n}$ cells for the memo. Luckily we only need to store the portal-state on the current planet so the required size becomes $2n * 2^n$ which is a difference in gigabytes for only 18 planets.

The Citrus Intern

Programing language: Java Difficulty: Medium Est: 4 hrs.

This problem can be solved by finding the graphs *Dominating set*. This is another NP-hard problem that can be simplified with dynamic programming as long as the graph is a tree which it luckily is. Again by recursive branching we can visit all nodes and calculate their respective values bottom up. Then the top node holds the final value.

The twist of this problem is that two adjacent vertexes cannot be connected. This is solved by never evaluating the of holding the current node in the set while a child node is also in the set.

Another small issue is that we need to know which node is the top-node of the tree. This is done by topological sorting the graph by a DFS and choosing the one with the largest number of reachable nodes.