1 Part 1 Theory

Task 1 Dynamic programming

a) Tree of recursive calls for fib(5)

When we analyse the runtime of the recursion for calculating the 5th Fibonacci number by drawing up a tree, we notice the pattern that it has to analyse each number down to the the 1st or 2nd Fibonacci number for each recursive call. This means it has several duplicate subtrees. We also notice that the runtime of the algorithm will be $O(2^n)$.

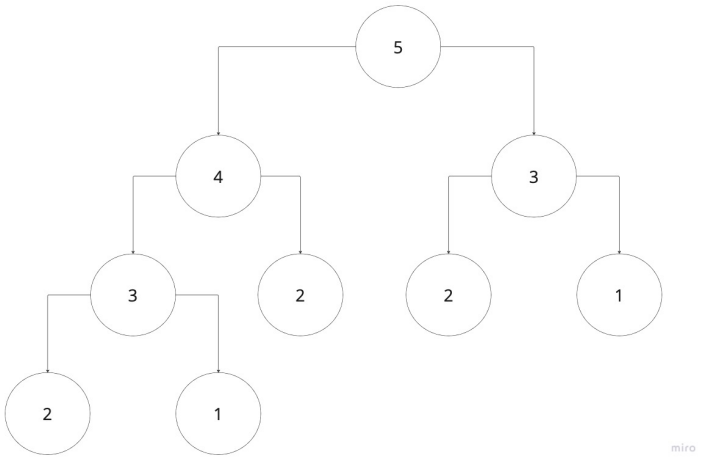

Figure 1: Recursive calls for fib(5)

b) Fibonacci algorithm in linear time

Dynamic programming practices will solve the runtime issue of the original algorithm to make sure we don't have to calculate duplicate subtrees over and over. By, for example, implementing memoization will make the tree of recursive calls look like figure 2 instead.

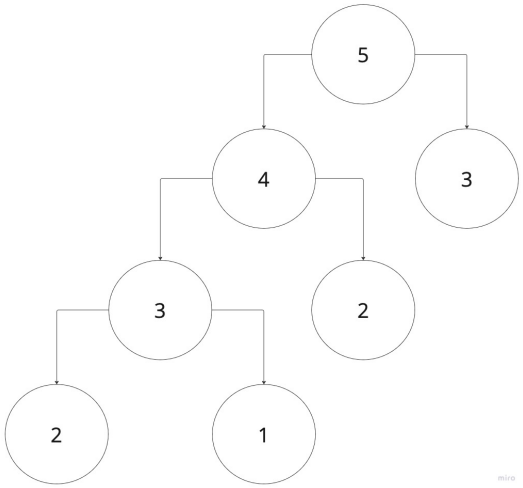

Figure 2: Recursive calls for fib(5) with memoization

c) Properties needed for dynamic programming

In order for a problem to be solved by dynamic programming it needs have two properties: optimal substructure and overlapping sub-problems. If the problem does not have those two properties, it can't be solved with dynamic programming. Dynamic programming is similar to the divide-and-conquer approach as it also combines solutions to sub-problems. In dynamic programming the solution to a sub-problem is needed repeatedly, so the solutions are stored in a table so that they don't need to be recomputed. For this to be useful, the sub-problems need to overlap.

Task 2 Smoothie Algorithm

It would be easy to adding the functionality so that the algorithm gives you the ingredients for making the best-tasting smoothie. Since the algorithm is based on dynamic programming, you would only need to extend the algorithm so that it keeps track of the optimal solution as well as the value by maintaining an additional array. This array S contains the solutions so that $S[i]$ contains an optimal set of intervals among $\{1, 2, ..., i\}$.
Naively enhancing the code to maintain the solutions in the array S, would however increase the runtime by $O(n)$. If we rather save the optimal values from the original algorithm down in array S, we are not explicitly maintaining the array, and the runtime is only increased by $O(1)$.

Task 3 Shortest-path

Bellman Ford's algorithm The Bellman Ford algorithm computed the shortest paths in a bottom-up manner like other dynamic programming problems also does. First it calculates the shortest distances that have at-most one edge in the path. Then it calculates the shortest paths with at-most 2 edges and so on. After the i-th iteration of outer loop. the shortest path with at most i edges are calculated. Assuming that there is no negative weight cycle if we have calculated shortest paths

with at most i edges, then an iteration over all edges guarantees to give the shortest path with at-most $(i + 1)$ edges.

Djikstra's algorithm With Djikstra's algorithm we compute a minimum spanning tree with a given source as a root. We maintain two sets, where one contains the nodes included in the shortest path tree, and the other contains nodes that are not yet included in the shortest path three. At each iteration of the algorithm we find a node in the second set of nodes that are not yet included, that has a minimum distance from the source.

Differences between the two algorithms Both Bellman Ford's algorithm and Djikstra's algorithm are single-source shortest path problems, meaning they both compute the shortest distance between each node in a graph from a single source node. However, they differ from each other in their approach for solving the problem. Bellman Ford's algorithm follow the dynamic programming approach, while Djikstra's algorithm follow the greedy algorithm approach.

Task 4 Grid travelling

a) Given a $n \times m$ grid where you are only allowed to move to the right or downwards from one cell to another. In how many ways can you travel from the top left corner (S) to the bottom right corner (F)?

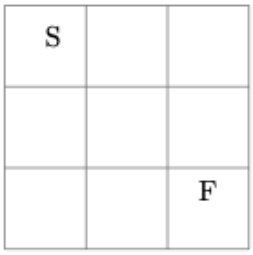

Figure 3: 3×3 grid

i. When we have a 3×3 grid we can draw up all possible ways to travel like we have in figure 4. We then find that we have 6 different ways of travel.

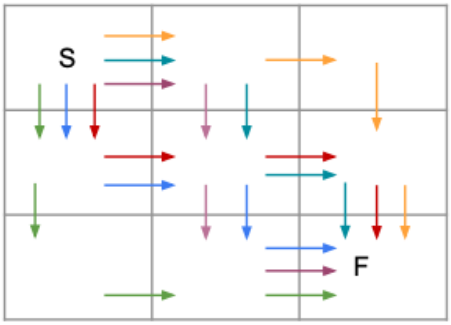

Figure 4: 3×3 grid

ii. With a 2×3 grid we find that we have 3 ways of traveling from S to F.

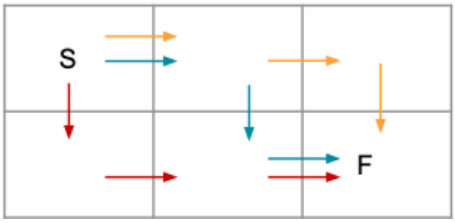

Figure 5: 2×3 grid

iii. With a 1×3 grid we find that we have 1 way of traveling from S to F.

Figure 6: 1×3 grid

Task 5 Sequence Alignment

a) What does α_{pq} and δ represent and how do they relate to the cost?

The sequence alignment problem is motivated by the notion of "lining up" two strings X and Y. An alignment gives a way of lining up the two strings, by telling us which pairs of positions will be lined up with one another.
The parameter δ corresponds to the gap penalty. This gives a penalty at the cost of δ if a gap is inserted between the string. For each pair of letters p, q in our alphabet, we have a mismatch cost of α_{pq} for lining up p with q. There is no mismatch cost to line up a letter with another copy of itself.
This means that the cost for each alignment matching M is the sum of its gap an mismatch points, and we seek an alignment of minimum cost.

b) Which alignments of PALETTE and PALATE has the lowest cost?

We assume that $\delta = 2$ and $\alpha_{pq} = 1$ if p is not equal to q, and $\alpha_{pq} = 0$ if p is equal to q.

```
# alignment 1
PALETTE
PALATE-
```

From alignment 1 we see that we have one gap and two mismatches which adds up to a cost of $M = 2\alpha_{pq} + \delta = 2 + 2 = 4$.

```
# alignment 2
PALETTE
PALAT-E
```

From alignment 2 we see that we have one mismatch and one gap which adds up to the cost of $M = \alpha_{pq} + \delta = 1 + 2 = 3$. This is the alignment with the lowest cost.

```
# alignment 3
P-ALETTE
-PALAT-E
```

From alignment 3 we see that we have one mismatch and and three gaps which adds up to the cost of $M = \alpha_{pq} + 3\delta = 1 + 6 = 7$.