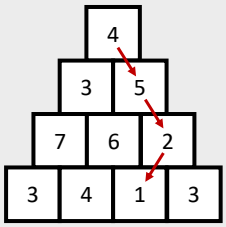
Algorithms Example Sheet 2: Problems

1 Dynamic programming

Exercise 2.P.1 [Pyramid of numbers] You are given a pyramid of boxes each of which contains a number. You start at one at the top and at each step you move to the level below in one of the adjacent boxes. Your goal is to find the path with minimum sum of entries. For example, in the image below the largest sum is achieved by the path shown:



(a) Design an algorithm to efficiently solve this problem.

(b) What is the time and space complexity of your approach?

(c) Write pseudocode for the top-down and the bottom-up approach.

(d) **(I)** Implement this algorithm. (You may want to submit your solution to **[LeetCode 120]**)

Exercise 2.P.2 [Longest Common Substring] The *longest common substring* between two strings is defined as the longest (continuous) string s that appears in both strings. For example, the longest common substring of `abcarexample` and `simplexactcat` is $s =$ mple. Design an algorithm to retrieve the longest common substring. What is the time and space complexity of your algorithm?

Exercise 2.P.3 [Bounded knapsack] In the bounded knapsack problem each item can be used at most c_i times.

(a) How would you solve the knapsack problem by reducing it to the 0/1 knapsack? What is the time complexity for this?

(b) (optional +) Read this article about speeding this up.

Exercise 2.P.4 [Unbounded knapsack]

(a) Extend your solution for 0/1 knapsack to the case where you have an infinite supply of each item.

(b) What is the time and space complexity of your approach?

(c) **(I)** Implement the algorithm and test your solution on **[GeeksForGeeks Unbounded knapsack]**.

(d) Explain how you can modify your algorithm to count the number of optimal subsets.

Exercise 2.P.5 [Knapsack variants]

(a) Given an array of positive integers $\leq U$, determine if it is possible to partition the array into two sets so that they have the same sum. **[LeetCode 416]**

(b) You are given an array of binary strings `strs` and two integers m and n. Return the size of the largest subset of `strs` such that there are at most m 0's and n 1's in the subset. **[LeetCode 474]**

(c) You are given an integer n and you are asked to break it into the sum of k positive integers, where $k \geq 2$, so as to maximise the product of those integers. **[LeetCode 343]**

Exercise 2.P.6 [Change-making]

(a) Recall the change-making problem (define in FoCS). Design a DP algorithm to determine if it is possible to make change for value C. How does the running time compare to that of the FoCS solution?

(b) **(I)** Implement the algorithm for finding the

(c) How would you find the way to make a change w using the fewest number of coins?

(d) Design an algorithm for counting the number of ways of making change.

(e) **(I)** Implement the algorithm for counting the ways of making change. You can test your implementation on **[LeetCode 518]**.

Exercise 2.P.7 [Seam Carving]

(a) Watch this video of seam carving in action (or if you are very interested, this extended one).

(b) Attempt Problem 15.8 in CLRS to see how it is implemented.

Exercise 2.P.8 Attempt **[2019P1Q7]**.

Exercise 2.P.9 [Max-cost independent set on trees] (optional) Attempt problem 15.6 in CLRS.

Exercise 2.P.10 [Longest Increasing Subsequence] The *longest increasing subsequence* of sequence $a_1, \ldots, a_n$ is the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that $i_1 \leq i_2 \leq \ldots \leq i_k$ and $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_k}$. For example, in the sequence $[4; 2; 3; 7; 5; 1; 2; 9]$ the longest increasing subsequence is $[2; 3; 5; 9]$.

(a) Design an algorithm to solve this problem.

(b) **(I)** Implement the LIS algorithm. You may want to test your implementation on **[LeetCode 300]**.

(c) Explain how you could retrieve a longest increasing subsequence.

(d) (+) Think how you could use binary search to improve the time complexity of your algorithm to $\mathcal{O}(n \log n)$.

Exercise 2.P.11 [Longest Palindromic Subsequence] A palindrome is a string that reads the same way forwards and backwards, e.g. `abcba`. The *longest palindromic subsequence* of $a_1, \ldots, a_n$ is the longest subsequence that is a palindrome. For example, $s =$ `anotherexample` then LPS is `aerea`. Design an algorithm that efficiently solves this problem. (If you prefer you can attempt **[2013P1Q6]**)

Exercise 2.P.12 [Bring your own problem I] Find at least one problem solvable using dynamic programming and attempt to solve it (or just bring it to the supervision and be ready to discuss it). (One possible source is the end of Chapter 15 in CLRS)

Greedy algorithms

Exercise 2.P.13 [Minimising waiting time at the queue] There are n customers waiting at a supermarket queue. You know that the i-th person will take t_i minutes to be served. You want to find the order to serve the n customers, which minimises the total waiting time. For example, if $t_1 = 5$, $t_2 = 7$ and $t_3 = 2$, then serving customer 2 (waits 0 minutes), then 3 (waits 7 minutes) and 1 (waits $7+2$ minutes) gives a total of 16 minutes (which is not optimal).

(a) Design an algorithm to find the optimal order to serve the customers.

(b) Prove that your algorithm is optimal.

Exercise 2.P.14 [Rearrangement inequality] You are given positive values $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$. Design an algorithm that pairs them up (say using an bijection $f : [n] \to [n]$) so that $\sum_{i=1}^{n} a_i b_{f(i)}$ is minimised. Prove that your algorithm is optimal.

Exercise 2.P.15 [Fractional knapsack] You are given a knapsack of capacity W. There are n different types of cheese with weight x_i and total value c_i. You can choose to include a fractional amount of cheese in your knapsack. Design an algorithm to maximise the value of cheese in your knapsack. (You can submit your solution **[GeeksForGeeks FractionalKnapsack]**).

Exercise 2.P.16 Attempt **[2015P1Q8]**.

Exercise 2.P.17 [Supervision scheduling] I have n supervisions to schedule. Supervision i starts at time a_i and ends at time b_i.

(a) Help me find the minimum number of supervisions that I will have to reschedule so that I do not have any overlapping ones.

[Source: Hayk Saribekyan]

(b) **(I)** Implement your approach. You may test your implementation on **[LeetCode 435]**.

(c) What if the tasks were not supervisions and they were meetings; and each meeting has a different weight of importance. How do you maximise the total weight of the meetings you attend?

(d) **(I)** Implement your approach. You may test your implementation on **[LeetCode 1235]**.

Exercise 2.P.18 [Invigilator scheduling] We want to find invigilators to cover the time interval $[S, T]$. We have received several offers from person i to invigilate in the interval $[a_i, b_i]$. Find the minimum number of invigilators needed to cover the time interval $[S, T]$.
If you implement your algorithm, you can test your implementation on **[LeetCode 1326]**.

Exercise 2.P.19 [Necklace] You have a necklace with n beads $b_1, \ldots, b_n$. There are k types of beads (for convenience we number these between 0 and $k-1$). We want to find the smallest (continuous) part of the necklace that contains all k different types (if such exists). (Aim for an $\mathcal{O}(n)$ algorithm)
If you want you can test your implementation on **[LeetCode 76]**.

Exercise 2.P.20 [Facility location in 1D] Given n points on the x-axis with coordinates x_i, find the optimal position x^* so as to minimise

$$f(x^*) = \sum_{i=1}^{n} |x^* - x_i|.$$

[Hint: How does $f(x^*)$ change as we move x^* from $-\infty$ to ∞?]

Exercise 2.P.21 [Bring your own problem II] Find at least one problem solvable using a greedy approach and attempt to solve it (or just bring it to the supervision and be ready to discuss it).

Divide and conquer

Exercise 2.P.22 [Counting inversions] Modify merge sort to count the number of pairs $i < j$ such that $A[i] > A[j]$ in $\mathcal{O}(n \log n)$ time. How does this help us count the number of swaps performed by insertion sort?
If you want you can test your implementation on **[GeeksForGeeks Count Inversions]**.

Exercise 2.P.23 [Closest pair of points] (+) In the *closest-pair-of-points* problem you are given a collection of N points $\{(x_i, y_i)\}_{i=1}^{N}$ on the 2D (Euclidean plane). The goal is to find two points with the smallest distance. This exercise guides you through the analysis of the algorithm presented here. It will probably not be clear why scanning finds the smallest distance.

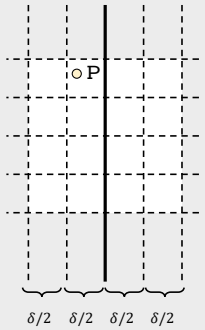
(a) Argue that you can implement the time complexities shown on the right of the algorithm.

(b) Argue that the time complexity of the algorithm is $\mathcal{O}\left(n \log^2 n\right)$.

(c) In the scanning step, why do we consider only points within a distance of δ from the median?

(d) Argue that two points in a square of side length $\delta/2$ have distance less than δ.

(e) (+) By considering a point P within a distance of δ from the median, use the following diagram (or otherwise) to argue that we need to check at most the next 8 points from the other side (in the y-sorted sequence of the points in the strip).



(f) Argue that you can implement sorting by y coordinates in $\mathcal{O}(n)$ time (in each recursive call). Show that the new algorithm takes $\mathcal{O}(n \log n)$ time.

Exercise 2.P.24 [Range sum queries (I)] In the *range sum query* problem, we are given an array with n elements and operations of the form:

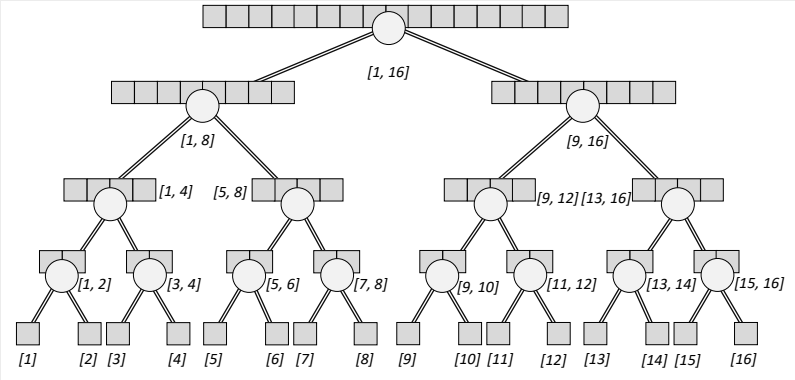
1. Update index i to value v.
2. Query the sum of the entries from index i to index j.

(a) Give a simple algorithm that solves the problem in $\mathcal{O}(1)$ per update and $\mathcal{O}(n)$ per query.

(b) Give an algorithm that solves the problem in $\mathcal{O}(n)$ per update and $\mathcal{O}(1)$ per query.

(c) (+) Split the array into sub-arrays of size roughly $\sqrt{n}$. By keeping the sum for each sub-array separately, argue (in detail) that you can support both operations in $\mathcal{O}(\sqrt{n})$ time.

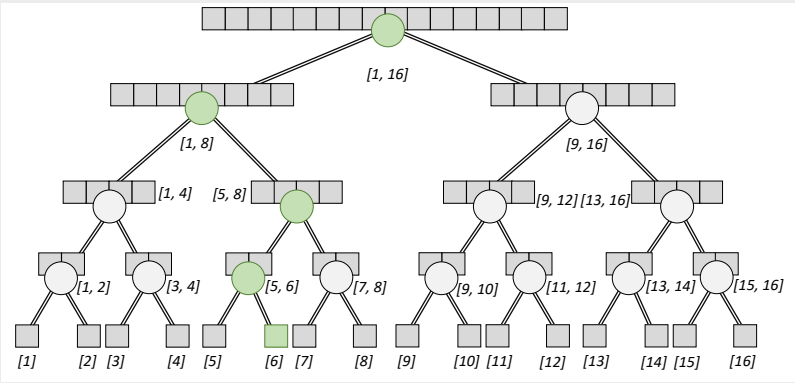
Exercise 2.P.25 [Range sum queries (II)] In this exercise, we will examine a slightly different approach to the range sum query problem than the one we took in Exercise 24. We will store a binary tree whose root maintains the sum of elements in $[1, n]$. This root will have two children the left one storing the sum of elements in $[1, n/2]$ and the right one storing the sum for $[n/2 + 1, n]$. The following diagram shows the case for $n = 16$.



(a) How many nodes are there in this binary tree? What is the height of the tree?

(b) Think how you would implement the update operation.

(c) (+) In order to update element i we start from the root and follow the sub-interval containing i. What is the worst-case time complexity of this operation? (See figure below for updating index 6)



(d) Think how you would implement the query operation.

(e) (++) In order to query the sum from elements $[i, j]$, we start from the root and we recursively consider the following cases (assuming $[u, v]$ is the interval of the current node):

- If $[u, v] \subset [i, j]$, then return the sum of the node without exploring its children.
- If $[u, v] \cap [i, j] = \emptyset$, then return 0.
- Otherwise, recursively call both children and sum their answers.

What is the time complexity for this operation? (See figure below for querying the sum in $[4, 14]$)