Homework 2

Due: 24 October, 2025 6pm PT

Unless stated otherwise, you should always prove the correctness of your algorithms and analyze their runtimes.

Each problem is graded on the coarse scale of $\checkmark^+$, $\checkmark$, $\checkmark^-$ and no $\checkmark$. It is also assigned a multiplier, denoting the relative importance of the problem. Both correctness and presentation are grading criteria.

Please read and make sure you understand the collaboration policy on the course missive. Extra credit problems are clearly marked below (see course missive for the details of grade calculations).

Remember to prove all your (non-elementary and not shown in class) mathematical claims, unless stated otherwise.

Each group of students should submit only 1 pdf to the corresponding Gradescope assignment.

Guidelines for writing dynamic programming solutions

Please try to keep your algorithm description as simple as possible—the main thing should be a single recurrence relation, expressed cleanly using mathematical operators like max or min. You should not be turning in 30 lines of code, since it will be way harder to prove the correctness of complicated code.

Your solution and proof of correctness of your dynamic programming algorithm **must** have the following ingredients:

1. Meaning of the table: how is the dynamic programming table indexed, how big is each dimension, and most importantly, what is the precise mathematical definition and **meaning** of each table entry? As an example, for the longest common subsequence problem in class, table entry $T[i, j]$ means "the length of the longest common subsequence between the first i entries $A[1..i]$ in string A and the first j entries $B[1..j]$ in string B".
2. Explicitly write out the recurrence that your dynamic programming algorithm will use. As an example, for longest common subsequence, the recurrence presented in class is: $T[i, j] = \max(T[i, j-1], T[i-1, j])$ if $A[i] \neq B[j]$, else $1+T[i-1, j-1]$ if $A[i] = B[j]$, else 0 if $i = 0$ or $j = 0$.
3. Write out how your algorithm will fill out the table. This is where you should present code (or pseudocode).
4. To prove correctness, first argue (perhaps less formally, without any induction, for the purposes of this course) how, by the time you fill an entry $T[i, j]$ (in our example), the entries it depends on in the recurrence must already have been computed.
5. The most important part of the proof is to show that the recurrence is correct. This is where many online resources and even textbooks handwave away, despite being the core and gist of the entire argument. Typically, to show the correctness of the recurrence, we argue that the recurrence "covers all possible cases/forms of a valid/optimal solution". Sentences like "a solution must be of the form bla" can be helpful.

6. Wrapping up: having (somewhat informally) shown that the table is filled correctly according to the specific meaning you gave to each table entry, conclude by showing how to find the overall solution to the problem from your table.

Problem 1

(5 ✓s total)

1. (2 ✓s) Given c different coins with positive integer denominations $d_1, \ldots, d_c$, where you only have one of each coin, give an $O(cn)$ time algorithm to determine if it is possible to make exactly n cents.

2. (3 ✓s) Given c different types of coins with positive integer denominations $d_1, \ldots, d_c$ (there are infinitely many of each type of coin), give an $O(cn)$ time algorithm to find the smallest number of coins needed to make n cents. You may assume that there is at least one way to make each (non-negative integer) value with the coins (though it is not difficult to handle sets of denominations where some values are not achievable).

Problem 2

(5 ✓s)

A.A. Milne is typing up a brand new Winnie the Pooh story. He finds that his word processor's greedy approach to word wrapping is not aesthetically pleasing enough for him. An easy way to do word wrapping is: as you go along, greedily put as many words as possible on the current line until you hit the margin, then go to the next line. However, perhaps a more aesthetically pleasing result might be to minimize the sum for every line *except the last line* of the square of the distance between the end of the line and the margin, *allowing the text to go over the margin*. However, the last line cannot go over the margin. For example, if A.A. Milne wants to write "Life is a journey to be experienced, not a problem to be solved" with margins that allow 16 characters between them, the greedy solution would do this:

```
|Life is a       |    : 9 characters, 7 characters remaining (7^2 = 49 penalty)
|journey to be   |    : 13 characters, 3 characters remaining (3^2 = 9 penalty)
|experienced, not|    : 16 characters, no penalty
|a problem to be |    : 15 characters, 1 characters remaining (1^2 = 1 penalty)
|solved.         |    : 7 characters, 9 characters remaining (0 penalty — last line)
```

This has a total penalty of $49 + 9 + 0 + 1 = 59$. However, the optimal solution has only 5 penalty:

```
|Life is a journey   |    : 17 characters, 1 character over (1^2 = 1 penalty)
|to be experienced,  |    : 18 characters, 2 characters over (2^2 = 4 penalty)
|not a problem to    |    : 16 characters, no penalty
|be solved.          |    : 10 characters, 6 characters remaining (0 penalty — last line)
```

1. Design a dynamic programming algorithm to achieve this beauty in documents. Your algorithm should both find the minimum possible penalty as well as the line splitting scheme that

yields this penalty. As a first step, or for partial credit, consider using the same penalty for the last line as for all the others.

2. Prove the correctness of your algorithm.
3. What is the runtime of your algorithm? Justify your answer.

Submit at most 1 of the 2 following extra credit problems, though you are welcome to work on both in collab hours

Problem 3

(**Extra credit, Difficult,** 5 ✓s)

Both divide+conquer algorithms and dynamic programming algorithms are often useful when processing trees. In this problem we will work through an example that combines tools from divide+conquer and dynamic programming.

Given a tree with n vertices that has weighted edges, and a positive integer $k \leq n$, we would like to find a path (that does *not* use any vertex more than once) of length k in the tree, whose sum of edge weights is maximal. Recall that the length of a path is the number of edges in that path.

Over the course of the following steps, we will devise an $O(n \log n)$ algorithm for this problem, that uses a mix of divide-and-conquer and dynamic programming. (There is a simpler algorithm that takes time $O(nk)$, but k might be big, so this is not what we are aiming for.) For each of the following parts that asks for an algorithm, a proof of correctness and runtime is also needed.

1. (1 ✓) Suppose you have a rooted tree with n vertices. Design an $O(n)$ algorithm that finds the best (that is, the maximal weight) path of length k that *starts at the root*, and goes down from there. (Hint: dynamic programming.)
2. (1 ✓) Suppose **for this part only** that the root of the tree has *exactly* two children. Modify your algorithm above so that it now finds the best path of length k that passes *through* the root. (It can start or end at the root, or pass through it as an intermediate node.) This algorithm should also take $O(n)$ time.
3. (1 ✓) Now solve the previous part *without* restricting the number of children of the root. The algorithm should still take $O(n)$ running time. (You cannot just reuse part 2 on all pairs of children of the root, as this could take quadratic time or k memory-per-child-of-the-root, which would be too slow.)

   Warning or hint: Make sure your path does *not* use any vertices more than once; check that the starting vertex and the ending vertex of the path do not belong to the same subtree of the root.
4. (1 ✓) For any given tree with n vertices, there is a vertex that, if we remove it, splits the tree into several smaller trees where each of these remaining trees has at most $\frac{n}{2}$ vertices. To prove this, show an $O(n)$ algorithm that finds such a vertex. (Hint: dynamic programming)
5. (1 ✓) Use the previous two algorithms you have constructed to design an $O(n \log n)$ *divide-and-conquer* algorithm that solves our problem, finding the maximal-weight path of length k in the tree.

Problem 4

(**Extra credit, Difficult**, 5 ✓s)

We are trying to ship products to Alaska. We have n_1 CDs that each weigh w_1, n_2 cassettes that each weigh w_2, and n_3 vinyl records that each weigh w_3, where $0 \leq w_1, w_2, w_3 \leq 1$. The goal is to ship everything using as few boxes as possible, given that each box can hold total weight at most 1. Find an algorithm to compute the number of boxes we need, in time $O(n_1 n_2 n_3)$.

(**Hint**: Try having a dynamic programming table where each table entry is a *pair* of numbers instead of a single number. Be very careful about the precise *meaning* of the table entry.)