Algorithms for Applications	HW2: *Dynamic Programming and Randomized Algorithms*
CS 5112 Fall 2025	**Due: October 16, 11:59pm ET**

Homework Instructions: The homework contains mainly three types of tasks:

- **Coding.** You can find a template for all the coding problems inside the `problems.zip` in the module. Please use these templates, to ensure the files and functions have the same names that the autograder will check for.

  Inside the `problems` directory, you will find the `tests` directory, which contains some sample test cases for the different problems. These test cases are intended to help you test and debug your solution. However, note that Gradescope will run a more comprehensive test suite. Feel free to add additional test cases to further test your code.

  In order to run the tests for a particular problem n, navegate into `problems/tests` and run: `python3 test_problem_n.py`

- **Algorithm Description.** When a questions requires designing an algorithm, you should describe what your algorithm does in the writeup, in clear concise prose. You can cite algorithms covered in class to help your description. You should also argue for your algorithm's asymptotic run time or, in some cases when indicated in the problem, for the run time bounds of your implementation.

- **Empirical Performance Analysis.** Some questions may ask you to do an empirical performance analysis of one or more algorithms' runtime under different values of n. For these questions, you should generate at least ten test cases for the implementation, for various values of n (include both small and large instances). Then, measure the performance locally on your own system, by taking the median runtime of the implementation over ten or more iterations. Graph the resulting median run times. The x-axis should be instance size and the y-axis should be median run time. You can use any plotting library of your choice. If you have never plotted on Python before, this matplotlib turorial has some examples.

  If several subquestions in a problem ask you to provide performance analysis, please do them all in the same environment (same system and configuration), in order to have a more accurrate comparison between algorithms. You can include graphs for a same problem in the same plot, as long as the different graphs are properly labeled.

  For these types of questions, you will not need to submit the tests you generate or your code for benchmarking or plotting the algorithms. However, you should include the graphs you generate into the write-up. You should also write about what you observe from the analysis and potentially compare it to the complexity analysis. Finally, also include a description of the environment (CPU, operating system and version, amount of memory) you did the testing on.

Submission Instructions: Hand in your solutions electronically on Gradescope. There are two active assignments for this problem set on Gradescope: one with the autograder, where you submit your code, and another one where you submit the write-up.

Coding Submission: You only need to submit the code for the specific functions we ask you

to implement. While you will write additional code to generate the performance analysis graphs when required, you do not need to submit that code.

To submit to the autograder, please zip the `problems` directory we provided you with. Therefore, your zipped file should have the following structure:

```
problems
├── problem_1
│   ├── p1_a.py
│   ├── p1_b.py
│   └── ...
├── problem_2
│   └── p2.py
└── ...
```

In order to ensure your code runs correctly, do not change the names of the files or functions we have provide you. Additionally, do not import external python libraries in these files, you don't need any libraries to solve the coding assignments. Finally, if you write any helper functions for your solutions, please include them in the same files as the functions that call them.

We have some public tests in the autograder to verify that all the required files are included, so you will be able to verify on Gradescope that your work is properly formatted shortly after submitting. We strongly encourage checking well ahead of the due date that your solutions work on the autograder, and seeking out assistance from the TAs should it not. We cannot guarantee being responsive the night the assignment is due.

Write-up Submission: Your write-up should be a nicely formatted PDF prepared using the LaTeX template on Canvas, where you can type in your answer in the `main.tex` file. If you do not have previous experience using LaTeX, we recommend using Overleaf. It is an online LaTeX editor, where you can upload and edit the template we provide. For additional advice on typesetting your work, please refer to the *resources* directory on the course's website.

Academic Integrity: You may use online sources or tools (such as code generation tools), but any tools you use should be explicitly acknowledged and you must explain how you used them. You are responsible for the correctness of submitted solutions, and we expect you to understand them and be able to explain them when asked by teaching staff.

Collaboration Policy: Collaboration (in groups of up to three students) is encouraged while solving the problems, but:

1. list the netids of those in your group;
2. you may discuss ideas and approaches, but you should not write a detailed argument or code outline together;
3. notes of your discussions should be limited to drawings and a few keywords; you must **write up the solutions and write code on your own**.

Problem 1. In this problem, you will compute the maximum sum of non-adjacent elements, maxsum(x) $= \max_S \sum_{s \in S} x_s$ where no two elements in S are adjacent, in two settings: a list and a tree. The definition of adjacent will be specific to each setting. Additionally, for each setting you must first write out the subproblem and then implement the full dynamic programming algorithm.

(a) Given a sequence of integers $x = x_1, x_2, \ldots, x_n$, your goal is to compute the largest sum of non-adjacent elements of x, maxsum_list(x). Adjacent elements of the sequences are next to each other: x_i and x_{i+1}.

(*2 points*) Identify the dynamic programming subproblem and write its solution here.

(*8 points*) Implement maxsum_list in `problem_1/p1_a.py`.

Code input and output format. The input is a python list, and the output should be an integer. Example: maxsum_list($[2, 1, 3, 4]$) $= 6$.

(b) Given a tree with nodes that contain integers, your goal is to compute the largest sum of non-adjacent elements. Trees are represented as a set of vertices $v_i \in V$ and edges $(v_i, v_j) \in E \subseteq V \times V$. There are no loops and every vertex has at most a single parent. Adjacent elements of the tree are those that have an edge between nodes, i.e. $(v_i, v_j) \in E$. For convenience, define the children of a node to be $c(v_i) = \{v_j | (v_i, v_j) \in E\}$.

(*2 points*) Identify the dynamic programming subproblem and write its solution here.

(*8 points*) Implement the algorithm in `problem_1/p1_b.py`.

Code input and output format. The input is a tree represented as an adjacency list. The adjacency list is a pair of dictionaries: the first dictionary maps node names to node values, and the second dictionary maps each node name to a list of child node names. The output should be an integer. Example: maxsum_tree(values $= \{2:2, 1:1, 3:3, 4:4\}$, adjacency $= \{2:[1], 1:[3], 3:[4]\}$) $= 6$.

Problem 2. Given n points in a 2D plane, your goal is to find the two points that are closest together according to the Euclidean distance $d(x, y) = \sum_i (x_i - y_i)^2$.

(a) (*5 points*) Implement a $O(n^2)$ brute-force method in `problem_2/p2_a.py`.

(b) (*5 points*) Implement the $O(n \log n)$ divide-and-conquer approach, described in Section 5.4 of Kleinberg and Tardos and the 9/19 lecture in `problem_2/p2_b.py`.

(c) (*10 points*) Implement the $O(n)$ randomized approach, described in Section 13.7 of Kleinberg and Tardos in `problem_2/p2_c.py`.

(d) (*5 points*) Perform an empirical analysis of the three approaches, plotting their runtime performance for a series of problem sizes.

Code input and output format. The input is a list of tuples of floats corresponding to points in the plane. The output is a list of the two closest points. Example: closest_points$([(0, 0), (1, 1), (5, 5)]) = [(0, 0), (1, 1)]$

Problem 3. Skip lists are randomized datastructures that allow efficient (in expectation) insertion, search, and deletion operations in a sorted list. We assume that list elements are always ints.

You will implement skip lists in this problem, as well as a linked list baseline. We supply separate Node classes in each file `problem_3/*.py`, which will be the backbone of each implementation. Both implementions will have the following interface:

- insert(list, x: int) $\rightarrow$ None: Add a new int to our list.
- search(list, x: int) $\rightarrow$ Node: Return the node in our list that has the value x.
- delete(list, x: int) $\rightarrow$ None: the node in our list that has have value x.

For simplicity, we assume that no repeated elements will be inserted into the list.

Please read the following resource on skip lists for more helpful information.

(a) (*5 points*) As a baseline, implement a linked list class. Give the runtime complexity of search, insertion, and deletion. The linked list must always be in sorted order and all elements will be ints. Include your implementation in `problem_3/p3_a.py`.

(b) (*5 points*) Implement insertion for skip lists, which should take $O(\log n)$ expected time. All list elements will be ints. Perform an empirical analysis of insertion in skip lists versus the linked list baseline. Include your implementation in `problem_3/p3_b.py`.

(c) (*5 points*) Implement search for skip lists, which should take $O(\log n)$ expected time. Give the expected and worst-case runtime complexity. All list elements will be ints. Perform an empirical analysis of search in skip lists versus the linked list baseline. Include your implementation in `problem_3/p3_b.py`.

(d) (*5 points*) Implement deletion for skip lists, which should take $O(\log n)$ expected time. All list elements will be ints. Perform an empirical analysis of deletion in skip lists versus the linked list baseline. Include your implementation in `problem_3/p3_b.py`.

Note: A deterministic alternative to skip lists are binary trees. However, in order for binary trees to remain performant, they must be balanced. Balanced binary tree implementations such as AVL and Red-Black trees have relatively complicated implementations compared to skip lists.

Algorithms for Applications	HW2: Challenge Problem
CS 5112 Fall 2025	**Due: October 16, 11:59pm ET**

Instructions: Challenge problems are, as the term indicates, *challenging*. They do not count for the homework score (90% of your course grade); instead, they are considered separately as extra credit over your course grade (additional 15% in total, 3.75% per assignment).

Questions about challenge problems will have lowest priority in office hours, and we do not provide assistance beyond a few hints to help you know whether you are on the right track.

Submission Instructions: You can choose not to hand a submission, but we encourage everyone to attempt the challenge problem. If you solve it, please hand in your answers (both, the coding and the write-up) through Gradescope, along with the rest of your assignment.

Academic Integrity and Collaboration Policy: The same guidelines apply to challenge problems as for the regular homework problems.

The number partition problem (NPP)

Given a set of n integers S, where all elements are unique, our goal is to find a partitioning of this set into $A \subset S$ and $B \subset S$ in order to minimize their difference $|\sum_{a \in A} a - \sum_{b \in B} b|$. Please include all of your implementations in `challenge_1/partition.py`.

(a) (*5 points*) Implement a brute-force approach that operates in exponential time.

(b) (*5 points*) Describe and implement a greedy approximation algorithm for this problem.

(c) (*5 points*) Describe and implement an exact dynamic programming approach.

(d) (*5 points*) Perform an empirical analysis of the approaches, reporting their runtime for different problem sizes as well as their accuracy for problem sizes where exact solutions are feasible.

Code input and output format. The input $x = x_1 \dots x_n$ is a list of n integers. The output should be a vector of 0's and 1's indicating the partitioning of the input into A and B.

Algorithms for Applications	HW2: Challenge Problem
CS 5112 Fall 2025	**Due: October 16, 11:59pm ET**

Instructions: Challenge problems are, as the term indicates, *challenging*. They do not count for the homework score (90% of your course grade); instead, they are considered separately as extra credit over your course grade (additional 15% in total, 3.75% per assignment).

Questions about challenge problems will have lowest priority in office hours, and we do not provide assistance beyond a few hints to help you know whether you are on the right track.

Submission Instructions: You can choose not to hand a submission, but we encourage everyone to attempt the challenge problem. If you solve it, please hand in your answers (both, the coding and the write-up) through Gradescope, along with the rest of your assignment.

Academic Integrity and Collaboration Policy: The same guidelines apply to challenge problems as for the regular homework problems.

The sequence alignment problem

In this challenge problem we will solve the *sequence alignment problem*. The approach we will implement is discussed in detail in Kleinberg and Tardos Sections 6.6 and 6.7.

In the sequence alignment problem, we are given two sequences $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$, as well as a cost function $c(x_i, y_j)$ that measures element-wise similarity. We denote δ as the cost for not aligning an element to anything, or aligning an element to a gap. The goal is to match up or align elements of x to elements of y that are similar. This is formalized as a minimum cost alignment between x and y, where a lower total cost implies that x and y are similar.

Formally, an alignment A is a set of ordered pairs that denote which elements $x_i \in x$ are matched with elements $y_j \in y$. Alignments have two properties: each element in x and y is aligned to at most one other element in the other sequence and there are no crossing pairs: if $(i, j), (i', j') \in A$ and $i < i'$, then $j < j'$. The cost of an alignment is the cost of each of the aligned elements.

For example,

```
stop-
-tops
```

corresponds to the alignment $[(2, 1), (3, 2), (4, 3)]$ and has cost

$$\delta + c(t, t) + c(o, o) + c(p, p) + \delta.$$

Elements aligned to gaps - are not included in the final list-of-tuples representation of the alignment.

(a) (*5 points*) Implement the $O(nm)$ dynamic programming algorithm described on page 282 of Kleinberg and Tardos in `challenge_2/seq_alignment_a.py`. This algorithm is known as the Needleman-Wunsch algorithm.

(b) (*15 points*) The above approach utilized $O(nm)$ space. Utilize a divide-and-conquer approach to reduce the space to $O(n + m)$ while keeping the running time complexity of $O(nm)$, following section 6.7 of Kleinberg and Tardos. This algorithm is known as Hirschberg's algorithm. Include your implementation in `challenge_2/seq_alignment_b.py`.

Code input and output format. The input is two strings, the cost function c, and gap penalty δ align(x, y, c, δ). For example, $x =$ "abcd", $y =$ "bcde",

```
def c(a, b):
    return 0 if a == b else 1
```

and $\delta = 1$. The output should be the list of tuples corresponding to the alignment and the cost of the alignment: $[(2, 1), (3, 2), (4, 3)]$ and cost 2.

While sequence alignment of two sequences is relatively solved, the problem of aligning multiple sequences is not. This is referred to as multiple sequence alignment (MSA), and is under active research.