CPSC 320 2025S: Assignment 4

This assignment is due **Friday, August 1 at 7 PM**. Late submissions will not be accepted. Please follow these guidelines:

- Prepare your solution using LaTeXand submit a pdf file. Easiest will be to submit using the .tex file provided. For questions where you need to select a circle, you can simply change `\fillinMCmath` to `\fillinMCmathsoln` .
- Enclose each paragraph of your solution with `\begin{soln}Your solution here...\end{soln}`. Your solution will then appear in dark blue, making it a lot easier for TAs to find what you wrote.
- Clearly label your answer to each question.
- Submit the assignment via GradeScope at `https://gradescope.ca`. Your group must make a **single** submission via one group member's account, marking all other group members in that submission **using GradeScope's interface**.
- After uploading to Gradescope, link each question with the page of your pdf containing your solution.

Before we begin, a few notes on pseudocode throughout CPSC 320: Your pseudocode should communicate your algorithm clearly, concisely, correctly, and without irrelevant detail. Reasonable use of plain English is fine in such pseudocode. You should envision your audience as a capable CPSC 320 student unfamiliar with the problem you are solving. You may **neither** include what we consider to be irrelevant coding details **nor** assume that we understand the particular language you chose. (So, for example, do not write `#include <iostream>` at the start of your pseudocode, and avoid language-specific notation like C/C++/Java's ternary (question-mark-colon) operator.)

Remember also to **justify/explain your answers**. We understand that gauging how much justification to provide can be tricky. Inevitably, judgment is applied by both student and grader as to how much is enough, or too much, and it's frustrating for all concerned when judgments don't align. Justifications/explanations need not be long or formal, but should be clear and specific (referring back to lines of pseudocode, for example). Proofs should be a bit more formal.

On the plus side, if you choose an incorrect answer when selecting an option but your reasoning shows partial understanding, you might get more marks than if no justification is provided. And the effort you expend in writing down the justification will hopefully help you gain deeper understanding and may well help you converge to the right selection :).

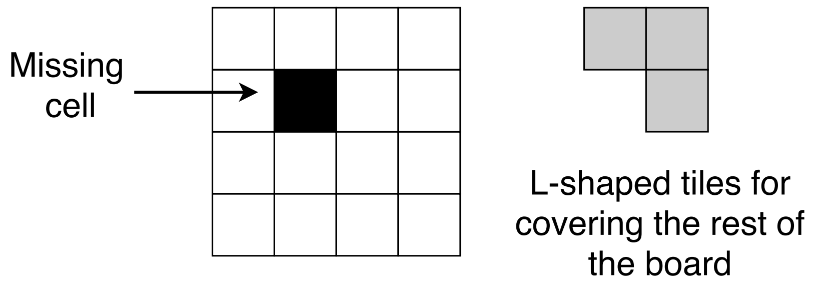
Ok, time to get started...

Group Members

Please list the CWLs of all group members here (even if you are submitting by yourself). We will deduct a mark if this is incorrect or missing.

1 Tiles and Tribulations

You're given a grid of size $n \times n$, where $n = 2^k$ for some $k \geq 1$, with one cell missing. Your job is to cover the board with *L-tiles*. An L-tile is three squares that form an L shape (i.e., a 2×2 square with one square missing). Below is a 4×4 board, with a missing cell at position $(2, 3)$ (using 1-based indexing from bottom left).



The L-tiles:

- Must cover every white square of the board.
- Must NOT cover the single black square on the board.
- Are not allowed to overlap.

1. [5 points] Prove that, for any $2^k \times 2^k$ board with an arbitrary missing cell, we can come up with a way to cover the board with L-tiles. (Hint: induction on k works well.)

   For a $2^1 \times 2^1$ board we can cover the board using one $L - tile$ by removing any arbitrary square, thus our base case $n = 1$ holds.

   Assume that we can cover any $2^k \times 2^k$ board using L-tiles when a square is missing with $k \geq 1$.

   Now consider a $2^{k+1} \times 2^{k+1}$ board with an arbitrary missing square.

   We partition this $2^{k+1} \times 2^{k+1}$ board into four $2^k \times 2^k$ quadrants, each of which maintains the structure of a square. We label them Q_1, Q_2, Q_3, Q_4 using traditional convention that Q_1 is top-right quadrant and so forth.

   The removed cell must appear in at least one of these sections. By the inductive hypothesis, we can cover it using $L - tiles$.

   WLOG, assume the missing cell is in top-right quadrant, Q_1. If not, we can rotate our board and relabel the quadrants accordingly.

   Then observe that the bottom-right square of Q_2, the top-right square of Q_3, and the top-left square of Q_4 forms an $L - tile$.

   Remove these cells from the respective quadrants and we get that by the inductive hypothesis we can cover those qudrants using $L - tiles$.

   Return the cells using the described $L - tile$ and we get that we can cover the $2^{k+1} \times 2^{k+1}$ board using $L - tiles$ with a square missing.

2. [5 points] Design a divide-and-conquer algorithm to place L-tiles to cover a $2^k \times 2^k$ board with a single missing cell.

B can be thought of as an $n \times n$ matrix where $n = 2^k$ for some integer $k \geq 1$. The matrix represents a square board.

For ease of notation we may also say $B = \{(i,j) : 1 \leq i, j \leq n\}$.

The coordinate $(1,1)$ refers to the bottom-left corner of the board, in particualar it's the origin.

For (i,j), first coordinate denotes horizontal position, second coordinate denotes vertical position.

The solution S is defined as a collection of sets of coordinates $\{L_1, L_2, \ldots, L_m\}$, where each L_i is a set of three coordinates that together form an L-shape.

We say that L_i is an $L-shape$ if each of its coordinates are adjacent to each other and they do not form a $3-cell$ line.

Additionally, the sets must satisfy the following conditions, where (x,y) is the missing cell:

$$\bigcap_{i=1}^{m} L_i = \varnothing \quad \text{(pairwise disjoint)}, \quad \text{and} \quad \bigcup_{i=1}^{m} L_i = B \setminus \{(x,y)\} \quad \text{(fully covers the board)}.$$

```
1: procedure Cover-Board($B$, $n$, $(x, y)$)
2:     if $n = 2$ then
3:         return $B \setminus \{(x,y)\}$
4:     end if
5:     $B_1 \leftarrow$ First Qaudrant of $B$
6:     $B_2 \leftarrow$ Second Qaudrant of $B$
7:     $B_3 \leftarrow$ Third Qaudrant of $B$
8:     $B_4 \leftarrow$ Fourth Qaudrant of $B$
9:     $L \leftarrow$ Middle square coordinates
10:    Identify $l \in \{1,2,3,4\}$ such that $(x,y) \in B_l$
11:    $L := L \setminus B_l$
12:    $L' \leftarrow L \cup \{(x,y)\}$
13:    $B_1 =$ Cover-Board($B_1$, $\frac{n}{2}$, $B_1 \cap L$)
14:    $B_2 =$ Cover-Board($B_2$, $\frac{n}{2}$, $B_2 \cap L$)
15:    $B_3 =$ Cover-Board($B_3$, $\frac{n}{2}$, $B_3 \cap L$)
16:    $B_4 =$ Cover-Board($B_4$, $\frac{n}{2}$, $B_4 \cap L$)
17:    return $B_1 \cup B_2 \cup B_3 \cup B_4 \cup L$
18: end procedure
```

```
1: procedure Wrapper-Call($B$, $n$, $(x, y)$)
2:     return Cover-Board($B$, $n$, $(x, y)$)
3: end procedure
```

3. [2 points] Give and briefly justify a good asymptotic bound on the runtime of your algorithm in terms of n (the dimension of the board).

We can model this using a recurrence relation.

We see that $T(2) = O(1)$, since we are just returning a set without a particular element.

Then to create each of B_1, B_2, B_3, B_4 this will take $O(n^2)$ time since we have to iterate through the entire oringal matrix to create them.

Determining the set L will take $O(1)$ time because we can query it using the passed in parameter n.

Each of the recursive calls will take $T(\frac{n}{2})$ time, and the return statement takes $O(\frac{n^2}{3})$ because we are supposed to merge together sets of size 3, which form partitions of B.

But we have $\frac{n^2-1}{3}$ of those sets.

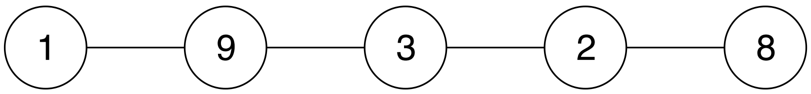
Thus, we can model this with

$$T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + O(n^2), & n > 2 \\ O(1), & n = 2 \end{cases}$$

We see that $n^2 \in \Theta(n^{\log_2(4)} \log^0(n))$, thus the master theorem says $T(n) \in \Theta(n^2 \log(n))$.

2 The Path to Victory

Suppose we have an undirected graph G that is a *path*: namely, its nodes can be written as $v_1, v_2, \ldots v_n$ with an edge between v_i and v_j if and only if i and j are consecutive numbers. Each node v_i has a positive weight w_i. For example, in the following path, the weights are shown as numbers drawn inside the nodes:



In this problem, we want to find the *maximum independent set.* An independent set is a subset of nodes such that no two of them share and edge, and the maximum independent set is the independent set with largest total weight. For example, in the graph above, the largest-weight independent set is v_2 and v_5, with total weight 17.

Assume you're given an n-dimensional array V, where $V[i]$ contains the weight of the node v_i (note that we are assuming 1-based indexing).

1. [3 points] Give a recurrence $M(i)$ (for $0 \leq i \leq n$) that defines the weight of the maximum independent set of the first i nodes in G.

$$M(i) = \begin{cases} \max(M(i-1), M(i-2) + V[i]), & i \geq 3 \\ \max(V[2], V[1]), & i = 2 \\ V[1], & i = 1 \\ 0, & i < 1 \end{cases}$$

2. [5 points] Give pseudocode for a recursive or iterative dynamic programming solution to find the weight of the maximum independent set in G.

```
procedure MAX-WEIGHT($V, n$)
    if $n = 1$ then
        return $V[1]$
    end if
    $w_1 = V[1]$
    $w_2 = \max(V[1], V[2])$
    for $i = 3, 4, \ldots n$ do
        $w_{max} = \max(w_2, w_1 + V[i])$
        $w_1 = w_2$
        $w_2 = w_{max}$
    end for
    return $w_2$
end procedure
```

```
procedure Max-Weight($V, W, n$)
    if $n < 1$ then
        return 0
    end if
    if $W[n] = -1$ then
        $W[n] = \max(\text{Max-Weight}(V, W, n-1), \text{Max-Weight}(V, W, n-2) + V[n])$
    end if
    return $W[n]$
end procedure
```

```
procedure WEIGHT-MAX-INDEPENDENT-SET($V, n$)
    Define $W[1..n]$ such that $W[i] = -1$ for each $i = 1, 2, \ldots, n$
    return Max-Weight($V, W, n$)
end procedure
```

3. [4 points] Write a function that takes the table from your dynamic programming solution and the array V and returns the indices (in 1-based indexing) of the actual nodes in the maximum independent set (i.e., write an "explain" function like we've done in class).

```
procedure EXPLAIN($V, W, n$)
    Define $R := \varnothing$
    Define $i := n$
    while $i > 3$ do
        if $W[i] == W[i-2] + V[i]$ then
            Add $i$ to $R$
            $i := i - 1$
        end if
        $i := i - 1$
    end while
    if $i = 1$ then
        Add $i$ to $R$
    end if
    return $R$
end procedure
```

3 Greed Is Good, but Sloth Is Better

Your boss is sending you to attend a conference to network with other people in your industry. The conference consists of various events (which may be seminars, lunches, teambuilding events, etc.), with each event having a start and end time. There can be multiple events going on at once.

The catch is that you don't really like conferences and networking, so would prefer to attend as few events as possible. However, you don't want your boss to catch on that you're trying to attend fewer events, so you'd like to attend a set of events that conflicts with every other event that you could be attending – that way, if your boss asks you why you didn't attend more events, you'll be able to tell her that you couldn't attend any other events due to scheduling conflicts. You also can't attend more than one event at a time, so you need to make sure that the events you select don't overlap with each other.

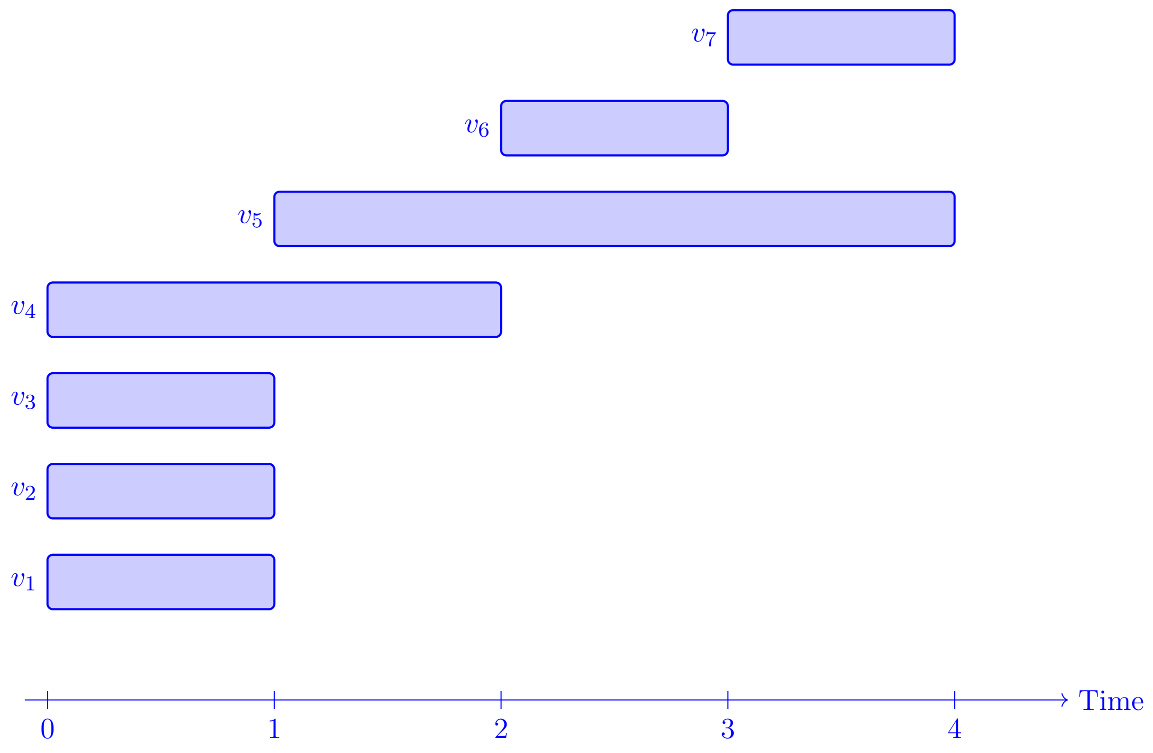
Formally, the conference events are given as an input list V, and each event v_i is defined by a start time and finish time (s_i, f_i). Your goal is to select the smallest subset of **non-overlapping** events such that every event in V overlaps with at least one of your chosen events. An event $v_i = (1, 4)$ overlaps with the event $v_j = (3, 5)$ but not with the event $v_k = (4, 6)$.

1. [3 points] Your co-worker took CPSC 320, and he's noticed that this problem is **awfully** similar to one he encountered in an assignment on greedy algorithms... He therefore proposes the following greedy algorithm (slightly modified from the solutions for A2) for choosing conference events to attend:

   If there are no events, return None. Otherwise, consider the event v' with the earliest end time. Of the events that intersect with v', add to the solution the event v with the latest end time. Remove all events that intersect with v and recurse on the remaining events.

   Give and briefly explain a counterexample in which this algorithm does not return an optimal solution.

   Consider the instance



   Using the proposed algorithm we would look at v_1 and determine that v_4 has the latest finish time such that it still intersects with v_1.

   Thus, after removing all the events that intersect with v_4 we are left with v_6 and v_7 that are disjoint, thus we would have to attned 3 events. In particular, we attned v_4, v_6, v_7.

However, if we select v_1 and v_5 we are able to intersect all of the events but by choosing one less event than proposed greedy would have.

2. [4 points] Assume the events in V are sorted by start time, and let $ME(i)$ denote the minimum number of events needed to cover events i to n (we say that an event is “covered” by a solution if it is either in the solution or overlaps with an event in the solution). Write a recurrence to define $ME(i)$.

   You can use the following helper function in your recurrence. For any index i, let $N(i)$ denote the index of the first event whose start time is greater than or equal to the end time of event i. You may assume that $N(i)$ is already implemented and that $N(i)$ returns ∞ if there are no events that begin after event i ends.

   At event n, being the latest in terms of start time we have no choice but to cover it using itself, so the minimum must be $ME(n) = 1$.

   If we consider some $i < n$, either the optimal solution for $(i, i+1, \ldots, n)$ includes i or not.

   If it includes i, then we must include into the optimal soltution the events that start after its finish time.

   Thus, we have that $ME(i) = 1 + ME(N(i))$.

   In the case where where i is not included, then there is some later finishing event who intersects it and is part of the optimal solution. Without further information, its at least the next latest finishing event.

   Thus, $ME(i) = ME(i+1)$.

$$ME(i) = \begin{cases} 1 & \text{if } i = n \\ \min(1 + ME(N(i)), ME(i+1)) & \text{if } i < n \end{cases}$$

3. [4 points] Write pseudocode for an iterative dynamic programming algorithm to compute $ME(i)$ for all i in $[1, \ldots, n]$.

```
procedure MIN-EVENTS($V, n, N$)
    Define $ME[1 \ldots n]$
    $ME[n] = 1$
    Define $i = n - 1$
    while $i \geq 1$ do
        $ME[i] = \min(1 + ME[N[i]], ME[i+1])$
        $i := i - 1$
    end while
    return $ME[1]$
end procedure
```

4 Constrained Shortest Paths

A company is leasing communication bandwidth from a national telecom provider to establish a new connection between its office in Vancouver and its office in Newark.

The telecom provider's network is represented by a graph $G = (V, E)$ and each edge (u, v) has an advertised price $p(u, v)$. In addition, each edge has a transit delay $d(u, v)$ due to node equipment. Suppose a new connection is to be established between two given nodes $s, t \in V$ and we require that the total delay between the two nodes be no more than D. Assume that all prices and delays are greater than 0. The company would therefore like to find the cheapest st-path which obeys this constraint. In other words, the problem to solve is:

$$\min_{P \text{ an } st\text{-path}} p(P) \text{ such that } d(P) \leq D$$

Here $p(P)$ denotes the price of the path P – that is $\sum_{(u,v)\in P} p(u, v)$ – and $d(P)$ denotes the total delay of the path P – that is, $\sum_{(u,v)\in P} d(u, v)$.

1. [5 points] For each $v \in V$, define $P(\delta, v)$ to be the minimum cost of a path from s to v with total communication delay at most δ. Write a recurrence to define $P(\delta, v)$.

   I care about making the least expensive path from s, t such that $p(P_{s-t}) = price$ is minimized as much as possible, but also that $d(P) \leq D$ where this means the delay.

   First, we define a base case that the path from s to itself will have cost zero, so long as the delay is non-negative.

   In the case where we look at a case where delay less then zero, we will give it a cost of ininfity.

   Then in the other cases, we look at the last node in the Path, and look at candidate nodes that precede it, say it's u.

   If such a node is going to precede it, then the delay between that node and v plus the delay from the best path s to u should be δ.

   We also should have that the cost of the path from s to u plus the cost of the edge from (u, v) should be minimized across all possible candidates.

$$P(\delta, v) = \begin{cases} 0 & \text{if } s = v \text{ and } \delta \geq 0 \\ \infty & \text{if } \delta < 0 \\ \min_{(u,v)\in E}(P(\delta - d(u, v), u) + p(u, v)) & \text{otherwise} \end{cases}$$

2. [7 points] Give a recursive memoized algorithm to solve for $P(D, v)$ for a given maximum delay D and node v.

   Assume that all relevant functions and arrays are defined.

   I.e. weight functions for delays d, cost weight functions p, adjacency lists, edge lists.

```
1: procedure FIND-BEST-PATH-WRAPPER($D, v$)
2:     Define $P[(\delta, u)]$ = min cost to reach node $u$ from $s$ with delay at most $\delta$ using a dictionary
3:     return Find-Best-Path($D, v, P$)
4: end procedure
```

```
procedure FIND-BEST-PATH($\delta, u, P$)
    if $s = u$ and $\delta \geq 0$ then
        return $0$
    end if
    if $\delta < 0$ then
        return $\infty$
    end if
    if $P[(\delta, u)] = None$ then
        $P[(\delta, u)] = \min_{(u', u) \in E}(\text{Find-Best-Path}(\delta - d(u', u), u', P) + p(u', u))$
    end if
    return $P[(\delta, u)]$
end procedure
```