CPSC 320 2022W1: Assignment 4

This assignment is due **Thursday November 17, 2022 at 22:00 Pacific Time**. Assignments submitted **within 24 hours after the deadline** will be accepted, but a penalty of up to 15% will be applied. Please follow these guidelines:

- Prepare your solution using LaTeX, and submit a pdf file. Easiest will be to use 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.
- Start each problem on a new page, using the same numbering and ordering as this assignment handout.
- 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. There are instructions for doing this on the CPSC 121 website, see `https://www.students.cs.ubc.ca/~cs-320/2022W1/index.php?page=assignments&menu=1&submenu=0`.

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. If you choose to use actual code, note that you may **neither** include what we consider to be irrelevant detail **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...

1 Statement on Collaboration and Use of Resources

To develop good practices in doing homeworks, citing resources and acknowledging input from others, please complete the following. This question is worth 2 marks.

1. All group members have read and followed the guidelines for groupwork on assignments in CPSC 320 (see `https://www.students.cs.ubc.ca/~cs-320/2022W1/index.php?page=assignments&menu=1&submenu=3`).

   ◯ Yes ◯ No

2. We used the following resources (list books, online sources, etc. that you consulted):

3. One or more of us consulted with course staff during office hours.

   ◯ Yes ◯ No

4. One or more of us collaborated with other CPSC 320 students; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ◯ Yes ◯ No

   If yes, please list their name(s) here:

5. One or more of us collaborated with or consulted others outside of CPSC 320; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ◯ Yes ◯ No

   If yes, please list their name(s) here:

2 Easily tired singers

You have been charged with the task of scheduling concerts for the famous rock band FosNot. They can give a concert every two days, and each concert takes place in a venue that is categorized as either *small* or *large*. There is a profit associated with each concert they give, which varies depending on the location, venue rental fee, expected attendance, etc. Their lead singer Wovid Abide gets tired easily however, so before giving a concert in a large venue the band needs to take a day off (that's in addition to the regular day off: so if they play in a large venue on day i, then their previous concert would be on day $i - 4$). This mini-vacation allows Wovid to gain the energy they'll need for the following concert.

You are given, for each day i in $\{2, 4, 6, \ldots, 2n\}$, an array A_i that contains a list $A_i[0]$, $A_i[1]$, $\ldots A_i[c_i]$ where element $A_i[j]$ includes the following information:

- $A_i[j]$.large: True if concert $A_i[j]$ would be in a large venue.
- $A_i[j]$.profit: The profit made by giving concert $A_i[j]$.

plus a lot of additional information that's not relevant to your task (the name of the city, the venue, etc).

Given this input, you want to produce a *plan*: a list that specifies for each day i in $\{2, 4, 6, \ldots, 2n\}$, either the index j_i of the concert the band will give on that day or an entry specifying the band will rest. The *profit* of the plan is determined in the natural way: for each i, you add $A_i[j_i]$.profit (unless the band rests on day i).

The problem: Given the input $A_2, A_4, \ldots, A_{2n}$, find the plan with maximum profit (such a plan will be called *optimal*).

1. [2 points] Let $NC[i]$, $SC[i]$ and $LC[i]$ denote the values of the best plans for the first $2i$ days that have no concert, a concert in a small venue, or a concert in a large venue respectively on day $2i$. Recall from the quiz that the following recurrence relation describes the value $SC[i]$:

$$SC[i] = S_i + \max\{NC[i-1], SC[i-1], LC[i-1]\}$$

where S_i is the largest profit for a small venue in A_{2i}, with $S_i = 0$ if there is no small venue in A_{2i}.

Write a recurrence relation for $NC[i]$.

Solution: $NC[i] = \max\{NC[i-1], SC[i-1], LC[i-1]\}$

2. [2 points] Now write a recurrence relation for $LC[i]$:

Solution: Let L_i be the largest profit for a large venue in A_{2i} (with $L_i = 0$ if there is no large venue in A_{2i}). The recurrence relation for $LC[i]$ is:

$$LC[i] = L_i + NC[i-1]$$

3. [4 points] Using the recurrence relations for $NC[i]$, $SC[i]$ and $LC[i]$, complete the following algorithm that returns the profit of an optimal plan for the band.

Solution: We simply implement our recurrences, building solutions in an order guaranteed to ensure that subproblem solutions are available by the time we need them:

```
Algorithm bestConcertSequence(A, n)
  $NC[0]$ ← 0
  $SC[0]$ ← 0
  $LC[0]$ ← 0

  for i ← 1 to n:
    //
    // Find both the small venue concert with the maximum profit on day i and the
    // large venue concert with the maximum profit on that day.
    //
    S[i] ← 0
    L[i] ← 0
    for j ← 1 to length(A[i]) do
      if A[i][j].large and A[i][j].profit > L[i] then
        L[i] ← A[i][j]
      if not A[i][j].large and A[i][j].profit > S[i] then
        S[i] ← A[i][j]

    //
    // Now compute profit for the three possible choices on day i.
    //
    $NC[i]$ ← max($NC[i-1]$, $SC[i-1]$, $LC[i-1]$)
    $SC[i]$ ← S[i].profit + max($NC[i-1]$, $SC[i-1]$, $LC[i-1]$)
    $LC[i]$ ← L[i].profit + $NC[i-1]$

  return max($NC[n]$, $SC[n]$, $LC[n]$)
```

4. [4 points] Write an algorithm `planConcerts` that uses the arrays your answer to question 3 constructed to construct an optimal plan for the band.

   Solution:

```
Algorithm planConcerts(A, n)
  choice ← "best" // The only other possibility is "none".
  for n ← length[A] down to 1:
    if choice = "best":
      if $LC[n]$ > $SC[n]$ and $LC[n]$ > $NC[n]$ then
        C[n] ← L[i]
        choice ← "none"
      else
        C[n] ← S[i]
    else
      C[n] ← "And Wovid Abide rested."
      choice ← "best"

   return C
```

5. [2 points] Analyze the time and space complexity of your algorithm from questions 3 and 4.

   Solution: There are $n+1$ subproblems stored in each of our table. So, the algorithm uses $O(n)$ space. Computing each table entry takes $\Theta(1)$ time, as does each iteration of the loop in the answer to question 4, and so the algorithm takes $\Theta(n)$ time.

3 Watching performances on Granville Street

On a clear, sunny Saturday, you get together with a group of friends to collect JockeyMon badges at the Granville Street JockeyMon badges festival. We will make the following assumptions about this festival:

- The participating stores are equally spaced, and the position of each participating store is an integer. Walking from the store at position i to the store at position $i+1$ takes exactly 5 minutes.
- There are n available badges; one badge is available every 5 minutes, in exactly one store. Badge i is available at the store at position p_i, exactly $5i$ minutes after the beginning of the festival (and only at that specific time).
- Collecting a badge is instantaneous, but you have to be directly in front of the store to get it.
- Your starting point is in front of the store at position 0.

You do not expect to be able to collect every badge, because consecutive badges may be available in stores that are not at successive positions. For instance, badge 4 may be available at the store at position 7 ($p_4 = 7$), and badge 5 may be at the store at position 2 ($p_5 = 2$). However you absolutely insist on collecting badge number n, which means you will need to plan your meandering along the street accordingly. Given this constraint, you want to collect as many badges as possible.

The problem: given the locations $p_1, p_2, \ldots, p_n$ where each of the n badges will be available, find a subset of badges of maximum size that you are able to collect, subject to the requirement that it should contain badge n. Such a solution will be called *optimal*.

1. [3 points] Give a recurrence relation for the maximum number of badges you will be able to collect from time 0 to time $5k$, assuming that you **must** be able to collect the badge on offer at time $5k$.

Solution: Let Q_k be the set $\{j \mid j < k \text{ and } |p_k - p_j| \leq k - j\}$

$$Counts[k] = \begin{cases} 0 & \text{if } |p_k| > k \\ \max_{j \in Q_k}\{1 + Counts[j]\} & \text{otherwise} \end{cases}$$

2. [4 points] Complete the following algorithm that returns the largest number of badges you can collect from time 0 to time $5n$. The parameter P is an array with the coordinate positions p_i of the store from which badge i will be available.

Solution:

```
Algorithm bestBadgesToCollect($p_1$, $p_2$, ..., $p_n$)
  count[0] ← 0
  for k ← 1 to n do
    if |$p_k$| > k then
      count[k] ← 0   // can never collect badge k
    else
      count[k] ← 1
      for j ← 1 to k-1 do
        if |$p_k$ - $p_j$| ≤ k - j and 1 + count[j] > count[k] then
        count[k] ← 1 + count[j]
  return count[n]
```

3. [4 points] Write an algorithm `listBadges` that uses the array your answer to question 2 constructed to output a list of the numbers of the badges that you will be able to collect in the optimal solution.

Solution: As usual, we revisit the *reasons* for our recurrence in the domain and build a solution on that basis. Whichever previous badge gave us the count we're using, that's the badge in our list before the current one (the badge from which we walked to reach the store where that badge is available).

```
define listBadges($p_1$, $p_2$, ..., $p_n$)
  if count[n] = 0 then
      return "Error: unable to collect the last badge"

  k ← n
  while k > 0 do
    print "Collect badge " + k
    //
    // Exit as soon as we've printed the last (first) badge number.
    //
    if count[k] = 1 then
      return

    //
    // Find the previous badge
    //
    for j ← k-1 downto 0 do
      if abs(P[j] - P[k]) ≤ k - j and count[k] = 1 + count[j] then
        k ← j
        exit for loop
```

4. [2 points] Analyze the time and space complexity of your algorithm.

   Solution: There are $n+1$ subproblems stored in our solution table (badge 0 up to and including badge n). So, the algorithm takes $O(n)$ space.

   However, it takes $O(k)$ time to compute the value for badge k because we need to iterate through all previous badges. Our runtime is therefore proportional to a sum like $\sum_{i=1}^{n} i$, which is in $O(n^2)$.

   Thus, the algorithm takes $O(n^2)$ time and $O(n)$ space.

4 Solving a Texla dealer's inventory problem

You have been hired to help a new Texla dealer with an inventory problem. Predictions tell them the quantity of sales to expect over the next n months. Let c_i denote the number of sales they expect in month i. To keep things simple, we will assume that all sales happen at the beginning of the month, and that cars that are not sold are *stored* until the beginning of the next month. The dealer has no cars in stock at the beginning of the first month.

The dealer has room to keep at most S cars, and it costs C to store a single car for a month. They receive shipments of cars from Texla by placing orders from them, and there is a fixed fee of F each time they place an order, regardless of the number of cars they order. We assume the cars arrive on the morning of the first day of the month (before the first sale for that month) and that the order is paid for on that day.

Let us call a sequence of orders that satisfies all the demands c_i and minimizes the total cost an *optimal plan*. In summary:

- There are two parts to the cost: (1) storage — it costs C for every car on hand that is not needed that month; (2) ordering fees — it costs F each time an order is placed, independently of the size of the order.
- In each month, the dealer needs enough cars to satisfy the demand c_i, but the number left over after satisfying the demand for the month should not exceed the inventory limit S.

The problem is to design an algorithm to compute an optimal plan.

1. [1 point] One possible way to find an optimal plan would be to define a recurrence for $\mathit{Cost}[m, x]$: the minimum cost for the first m months assuming you want to be left with x cars at the end of month m. What would be the space complexity of a dynamic programming algorithm that uses this recurrence?

   Solution: The table would have size Sn.

2. [3 points] There is another dynamic programming algorithm that uses a different recurrence relation, and has a much lower space complexity. This algorithm relies on the following theorem:

   > An optimal plan that places an order for one or more cars for month m does not have any cars remaining in stock at the end of month $m-1$.

   Prove this theorem.

   Solution: We use a proof by contradiction. Let P be an optimal plan. Suppose that, in P, the dealer places an order for x cars in month m, and that they had $y > 0$ cars left in stock at the end of month $m-1$. Let $m' \leq m-1$ be the month where the previous order (the order that occurs before the order at month m) was placed.

   We construct a plan P' has follows: we reduce the number of cars ordered in month m' by y, and increase the number of cars ordered in month m by the same number. The plans P and P' are thus identical before month m', and from month m onwards; the only difference between them is that P has y more cars in stock at the end of months m', $m'+1, \ldots m-1$. Therefore $\mathit{Cost}(P) = \mathit{Cost}(P') + yC(m-m')$. Since y, C and $m-m'$ are all positive, this contradicts the optimality of P.

3. [3 points] Let $Cost[m]$ denote the cost of an optimal plan for the first m months. Derive a recurrence relation for $Cost[m]$, and explain why it is correct.

Solution: First, we observe that an optimal plan for the first m months will leave no cars in stock at the end of month m (the proof is essentially the same as the proof of the statement in question 2). Given a positive integer $m > 1$ (a month number), let p_m be the earliest month during which the dealer could order enough cars to satisfy demand for months p_m, $p_m + 1$, ..., m without exceeding the inventory limit S. That is,

$$p_m = \min k \text{ such that } 1 \leq k \leq m \text{ and } \sum_{i=k+1}^{m} c_i \leq S$$

because at the end of month p_m, the number of cars in stock will need to satify demand for the remaining months $p_m + 1$, $p_m + 2$, ..., m.

The choice we need to make to have an optimal plan for the first m months is when to order the cars that will be sold in month m. If we order these cars in month k, then the cost of the plan will be

$$Cost[k-1] + F + C \sum_{i=k+1}^{m} c_i(i-k)$$

because the cars needed in month i will need to be in stock at the end of months k, $k+1$, ..., $m-1$. We want the plan with minimal cost that respect the inventory limit, and so

$$Cost[m] = \begin{cases} \min_{p_m \leq k \leq m}\{Cost[k-1] + F + C\sum_{i=k+1}^{m} c_i(i-k)\} & \text{if } m > 0 \text{ and } c_m > 0 \\ Cost[m-1] & \text{if } m > 0 \text{ and } c_m = 0 \\ 0 & \text{if } m = 0 \end{cases}$$

4. [3 points] Write a dynamic programming algorithm that computes $Cost[m]$ for every value of m.

Solution:

```
Algorithm computeMinCost($c_1$, ..., $c_n$)
  //
  // First precompute $\sum_{i=1}^{k} c_i$ and $\sum_{i=1}^{k} ic_i$ for each value of k. These sums will be
  // used further down to avoid looping repeatedly.
  //
  sum[0] ← 0
  isum[0] ← 0
  for m ← 1 to n do
    sum[m] ← sum[m-1] + $c_m$
    isum[m] ← isum[m-1] + m * $c_m$

  //
  //
  //
  Cost [0] ← 0
  for m ← 1 to n do
    if $c_m$ = 0 then
      Cost [m] ← Cost [m-1]
    else
    Cost[m] ← +∞
    k ← m
    total ← 0
    while k ≥ 1 and total ≤ S do
      cost ← Cost[k - 1] + F + C (isum[m] - isum[k] - k * (sum[m] - sum[k]))
      if cost < Cost[m] then
        Cost[m] ← cost
      total ← total + $c_k$
      k ← k - 1
```

5. [3 points] Write an algorithm **planPurchases** that takes as input any array that your answer to question 4 may have constructed, and returns a list of the numbers of the cars to buy in each of the n months.

Solution:

```
Algorithm planPurchases($c_1$, ..., $c_n$)
  m = n
  while m ≥ 1 do
    if $c_m$ = 0 then
      m ← m - 1
    else
      //
      // Find out which k gave the minimum cost for Cost[m]
      //
      k ← m
      while k ≥ 1 do
        cost ← Cost[k - 1] + F + C (isum[m] - isum[k] - k * (sum[m] - sum[k]))
        if cost = Cost[m] then
          break out of inner loop
        k ← k - 1

      print "Buy " + (sum[m] - sum[k]) + " cars in month " + k
      m = k - 1
```

6. [2 points] Analyze the worst-case running time and the space requirements of your algorithm from question 4 and 5.

Solution: The algorithm runs in $O(n^2)$ time and uses $O(n)$ space.