CPSC 320 Final Examination
December 12th, 2020, 12:00 to 14:30

- You have 150 minutes to write the 7 questions on this examination. A total of 89 marks are available.
- **Justify all of your answers.**
- You are allowed to access the course web site, Canvas, your notes, the textbook and (only to ask a question if you believe you have found an error on the exam) private posts on Piazza, but not other web sites. You are not allowed to contact people other than the course staff.
- Keep your answers short. If you run out of space for a question, you have written too much.
- The number in square brackets to the left of the question number indicates the number of marks allocated for that question. Use these to help you determine how much time you should spend on each question.
- Use the back of the pages for your rough work.
- **Good luck!**

UNIVERSITY REGULATIONS:

- Each candidate should be prepared to produce, upon request, his/her UBC card.
- CAUTION: candidates guilty of any of the following, or similar, dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action.

    1. Having at the place of writing, or making use of, any books, papers or memoranda, electronic equipment, or other memory aid or communication devices, other than those authorised by the examiners.
    2. Speaking or communicating with other candidates.

1 Short Answer (warmup) questions

1. **[3 marks]** Which Θ bound will the Master Theorem give for the following recurrence relation?

$$T(n) = \begin{cases} 9T(n/3) + n^2 \log n & \text{if } n \geq 3 \\ \Theta(1) & \text{if } n \leq 2 \end{cases}$$

   a. ◯ n^2
   b. ◯ $n^2 \log n$
   c. ◯ $n^2 \log^2 n$
   d. ◯ $n^{\log_3 9}$
   e. ◯ The Master theorem can not be used to obtain a tight bound on $T(n)$.

2. **[3 marks]** Which Θ bound will the Master Theorem give for the following recurrence relation?

$$T(n) = \begin{cases} \sqrt{n}T(n/\sqrt{n}) + n & \text{if } n \geq 2 \\ \Theta(1) & \text{if } n \leq 1 \end{cases}$$

   a. ◯ $\sqrt{n} \log n$
   b. ◯ n
   c. ◯ $n \log n$
   d. ◯ $n \log^2 n$
   e. ◯ The Master theorem can not be used to obtain a tight bound on $T(n)$.

3. **[4 marks]** When we use the Gale-Shapley algorithm to solve the employer/applicant problem discussed in class, we may obtain different solutions depending on who is proposing (employers or applicants). However not every stable matching may be obtainable using Gale-Shapley. Give a small example of a single input to the stable matching problem, and three **different** stable matchings for that input:

   - The stable matching returned by the Gale-Shapley algorithm when employers propose.
   - The stable matching returned by the Gale-Shapley algorithm when applicants propose.
   - A stable matching that is different from the previous two.

   Hint: consider combining a pair of instances, each with two employers/applicants.

2 I have a (recurring) dream

1. **[8 marks]** Consider the recurrence relation:

$$T(n) = \begin{cases} aT(n-b) + c^n & \text{if } n \geq b \\ 1 & \text{if } n < b \end{cases}$$

Give a Θ bound for $T(n)$ in the case where $a < c^b$. To simplify the algebra, you may assume that n is a multiple of b. Hint: use a recursion tree.

3 Making Change Revisited

In this question, we revisit the problem of making change, which we worked on in lectures/worksheets. (However, our notation here might be different, so be sure to use the notation in this question.) Briefly, the problem is, given a nonnegative integer n and k types of coins, each with nonnegative integer value x_i, where $i = 1, \ldots, k$, find the smallest number of coins whose values add up to exactly n. For this question, we will set $k = 3$, and always have $x_1 = 1$. We will also assume that $x_1 < x_2 < x_3$. For example, if $n = 17$, and $x_2 = 5$, and $x_3 = 10$, then the best solution uses 4 coins (1 of x_3, and 1 of x_2, and 2 of x_1).

In class, we used this problem to learn memoization and dynamic programming. But in real life, we are typically greedy, always using the largest coin that is less than or equal to the amount leftover. E.g., returning to the example above, to make change for the $n = 17$ case, we use one of x_3 because $x_3 \leq 17$, and then we are left to make change for $17 - x_3 = 7$ cents. Now, $x_3 \not\leq 7$, but $x_2 \leq 7$, so we use one of x_2, leaving $7 - x_2 = 2$ cents. And then since $x_3 \not\leq 2$ and $x_2 \not\leq 2$, we use x_1 twice. A natural question is to wonder when is it optimal to be greedy?

1. **[2 marks]** Here's some pseudo-code for the greedy algorithm. Prove that this does **not** always compute the correct, minimum number of coins needed. Note that this question is for the general case, where $x_1 = 1$, but x_2 and x_3 are arbitrary positive integers with $x_1 < x_2 < x_3$, i.e., we are **not** continuing the example where $x_2 = 5$ and $x_3 = 10$. (Hint: To disprove something, all you need is a concrete counterexample, with a brief explanation. Excessively verbose or vague answers will not get credit!)

```
1:  Let ncoins = 0;                          ▷ ncoins counts the number of coins used.
2:  while x_3 ≤ n do                         ▷ Use as many x_3 as possible.
3:      Let n = n − x_3.
4:      Let ncoins = ncoins + 1.
5:  end while
6:  while x_2 ≤ n do                         ▷ Next, use as many x_2 as possible.
7:      Let n = n − x_2.
8:      Let ncoins = ncoins + 1.
9:  end while
10: while x_1 ≤ n do                         ▷ Finally, use x_1 = 1 until done.
11:     Let n = n − x_2.
12:     Let ncoins = ncoins + 1.
13: end while
14: return ncoins
```

2. **[4 marks]** Here's a recursive function to solve the problem correctly. However, it runs in worst-case time exponential in n. Memoize the code so that it runs in worst-case time linear in n. You may assume that there is already an array `solution[0..n]` and a wrapper function that has pre-initialized the array so that all entries are initially -1 when this function is called, so all you have to do is make the simple changes *within* this function to memoize it. (Excessively convoluted code will not receive credit! You may copy the code with your changes, or simply mark your changes on the code, or describe your edits (e.g., insert "`printf("Hello, World.\n");`" betweens line 7 and 8), as long as it's clear, unambiguous, and easy-to-read.)

```
1:  function NCOINS(n)
2:      if n < 0 then
3:          return ∞;
4:      end if
5:      if n = 0 then
6:          return 0;
7:      end if
8:      Let a = ncoins(n − x3)          ▷ Try using x3 first.
9:      Let b = ncoins(n − x2)          ▷ Try using x2 first.
10:     Let c = ncoins(n − x1)          ▷ Try using x1 first.
11:     return min(a, b, c) + 1         ▷ Best of those solutions plus 1 for the coin used.
12: end function
```

3. **[6 marks]** Now, consider this code that mixes a bit of greedy into the recursive function:

```
1:  function NCOINS(n)
2:      if n < 0 then
3:          return ∞;
4:      end if
5:      if n = 0 then
6:          return 0;
7:      end if
8:      Let g = 0                          ▷ g counts how many greedy steps we do.
9:      while x_2 x_3 ≤ n do               ▷ When n is big, use x_3.
10:         Let n = n − x_3.
11:         Let g = g + 1.
12:     end while
13:     Let a = ncoins(n − x_3)            ▷ Try using x_3 first.
14:     Let b = ncoins(n − x_2)            ▷ Try using x_2 first.
15:     Let c = ncoins(n − x_1)            ▷ Try using x_1 first.
16:     return g + min(a, b, c) + 1        ▷ Best of those solutions plus 1 for the coin used.
17: end function
```

In particular, if n is greater or equal to the product x_2x_3, then the algorithm takes greedy steps of using x_3. Formally prove that this code computes the correct solution. You may assume the recursive calls work correctly, so you need to prove only that the greedy steps do not make the algorithm give an incorrect solution. You should use a greedy-stays-ahead or exchange-argument proof, as described in our textbook.

4. **[1 marks]** Notice that the code in lines 8–12 in part 3 are basically doing division by x_3, so we can optimize the code into:

```
1:  function NCOINS(n)
2:      if n < 0 then
3:          return ∞;
4:      end if
5:      if n = 0 then
6:          return 0;
7:      end if
8:      Let g = ⌊(n − x2x3)/x3⌋
9:      Let n = n − gx3
10:     Let a = ncoins(n − x3)          ▷ Try using x3 first.
11:     Let b = ncoins(n − x2)          ▷ Try using x2 first.
12:     Let c = ncoins(n − x1)          ▷ Try using x1 first.
13:     return g + min(a, b, c) + 1     ▷ Best of those solutions plus 1 for the coin used.
14: end function
```

If we assume arithmetic operations take constant time, and **if we assume x_1, x_2, and x_3 are constants** (so that only n is considered to grow asymptotically), then give a tight big-*Theta* bound (in n) on the worst-case runtime of this algorithm.

4 Evaluating Movie Ranking Engines

1. **[13 marks]** Imagine that you are working for a major media streaming company. Your company has announced a public machine learning challenge competition, and you need to develop an efficient algorithm for evaluating the entries.

   Specifically, there is a set of n movies, which we can consider to be numbered with the integers from $1, \ldots, n$. The company has an internal list `truelist[1,...,n]` that lists the movies in decreasing order of the CEO's preference, so e.g., `truelist[1]` is the CEO's favourite movie, and `truelist[n]` is the CEO's least favourite movie. To make your job easier, the company has also pre-computed a second array `truerank[1,...,n]`, which you can think of as `truelist` inverted, i.e., `truerank[i]` tells you the rank of movie i, and so `truelist[truerank[i]]=i` for all i. These two arrays are not made public.

   A competitor submits a list `complist[1,...,n]` that also lists the n movies, trying to guess the CEO's preference list. The competition rules define a *mistake* to be a pair of ranks i and j in the competitor's list, such that $i < j$, but `truerank[complist[i]]` $>$ `truerank[complist[j]]`. In other words, the competitor guesses that the CEO likes movie i more than movie j, but actually, the CEO likes movie i less than movie j.

   (Note: This next example just illustrates the definitions above. If the mathematical definitions are clear, you may safely skip the example.) For example, if $n = 4$, we might have the following four movies with their numbers:

Number	Title
1	*101 Dalmatians*
2	*2 Fast 2 Furious*
3	*The Third Man*
4	*Ip Man 4*

   and suppose the CEO likes newer action movies, so they would rank *Ip Man 4* first, then *2 Fast 2 Furious*, then *101 Dalmatians*, and last *The Third Man*. We would represent these preferences in the arrays:

`truelist`				
Array Position	1	2	3	4
Value	4	2	1	3

`truerank`				
Array Position	1	2	3	4
Value	3	2	4	1

   If a competitor submitted this list:

`complist`				
Array Position	1	2	3	4
Value	1	2	3	4

   (which ranks the movies in numerical order), then we compute that there are 4 mistakes (corresponding to $(i, j) = (1, 2)$, $(1, 4)$, $(2, 4)$, and $(3, 4)$).

   It's easy to compute the number of mistakes by directly following the definition:

```
1: Let m = 0
2: for i = 1, ..., n − 1 do
3:     for j = i + 1, ..., n do
4:         if truerank[complist[i]] > truerank[complist[j]] then
5:             Let m = m + 1
6:         end if
7:     end for
```

8: **end for**
9: return m

Unfortunately, this runs in time $\Theta(n^2)$, which is too slow when n is large.

Fortunately, the brilliant company CEO left you a note, "You can do this with divide-and-conquer, just like QuickSort, and run in average-case $O(n \log n)$ time," and some pseudocode, before disappearing for a year of yoga in a yurt in the Yukon. Unfortunately, the pseudocode is illegible in a few key places. The pseudocode is shown here, and your task for this question is to fill in the missing code so that the algorithm runs in average-case time $O(n \log n)$ (just like QuickSort) and computes the correct number of mistakes.

```
1:  function CountMistakes(complist)
2:      Let n = complist.length.        ▷ For any list, assume .length gets the length in O(1) time
3:      if n ≤ 1 then
4:          return [                    ]
5:      end if
6:      Let betterpart be an empty list.
7:      Let worsepart be an empty list.
8:      Let p = complist[1].                                          ▷ Like the QuickSort pivot
9:      Let x = 0.                                  ▷ x counts the mistakes we lose when we partition.
10:     for j = 2, ..., n do          ▷ Split complist into the movies that rank better or worse than p.
11:         if truerank[complist[j]] < truerank[p] then
12:             Append complist[j] to betterpart.                 ▷ Assume this line takes O(1) time
13:             [                                                 ]
14:         else
15:             Append complist[j] to worsepart.                  ▷ Assume this line takes O(1) time
16:             [                                                 ]
17:         end if
18:     end for
19:     Let b = CountMistakes(betterpart).
20:     Let w = CountMistakes(betterpart).
21:     return [                    ]
22: end function
```

Insert the appropriate code in each of the boxes. If there is no code to be inserted, write "N/A" in that box.

5 The Magic of Dynamic Programming

An Oxford medieval studies Faculty member has found a manuscript that contains several lists of instructions claiming to be magic rituals. Each ritual is a list of instructions of the form

Pick a direction, walk s_i steps and then state the magic words x_i outloud.

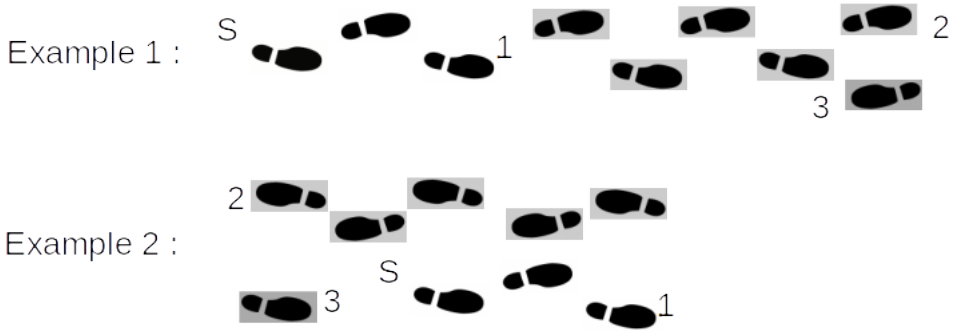
where the values s_i and words x_i are the only variables. For instance, the list of instructions for the *substitutiary locomotion* ritual is:

1. Pick a direction, walk 8 steps and then state the magic word "Treguna".
2. Pick a direction, walk 3 steps and then state the magic word "Mekoides".
3. Pick a direction, walk 14 steps and then state the magic word "Tracorum".
4. Pick a direction, walk 4 steps and then state the magic word "Satis".
5. Pick a direction, walk 9 steps and then state the magic word "Dee".

The steps do not need to all be in the same direction; walking back and forth is acceptable as long as the right number of steps are taken each time.

As this Faculty member does not want to become the department's laughing stock, they plan to conduct the rituals in the narrow hallway (of length L steps) in front of their office, in the middle of the night, to see if anything happens. This means that in order to conduct a ritual, the entire sequence of steps, from an arbitrary starting point in the hallway to the last step of the ritual, must fit within that length L. We assume without loss of generality that the starting point (and all other points where the Faculty member will stop while following the instructions of the ritual) are an integer number of steps from the beginning of the hallway.

Here are two examples of a three stage ritual with 3, 5 and 1 steps respectively, in a hallway of length 8. The starting point is denoted by S, and the numbers 1, 2 and 3 correspond to where the Faculty member will be standing to state the first, second and third magic words. In the first example, the Faculty member walks towards the right for the first two stages, and left for the third stage. In the second example, the Faculty member walks right, then left, then right again. Observe that the two examples start at different points along the hallway.



As the Faculty member knows nothing about computer science, you have been hired to write an algorithm that takes as input an array A containing the number of steps of each stage of a ritual, and the length L of the hallway. The example from the figure corresponds to the input $A = [3, 5, 1]$, $L = 8$. We will use n to denote the length of the array A.

Let

$P[i, t]$

be `True` if it is possible to fit the first i stages of the ritual in the hallway, so that the Faculty member will be at position t after the i^{th} stage, and `False` otherwise.

1. **[6 marks]** Complete the following recurrence relation that defines the value $P[i,t]$. Hint: if the Faculty member is at position t at the end of the i^{th} stage of the ritual, where might they have been at the end of the $i-1^{st}$ stage?

   Note: you can complete the other questions in this section without the answer to this one.

$$P[i,t] = \begin{cases} \text{False} & \text{if } t < 0 \text{ or } t > L \\ \boxed{} & \text{if } t \geq 0 \text{ and } t \leq L \text{ and } i \boxed{} \\ \boxed{} & \text{if } t \geq 0 \text{ and } t \leq L \text{ and } i \boxed{} \end{cases}$$

2. **[7 marks]** Write pseudo-code for a dynamic programming algorithm that computes $P[n,t]$ for every value of t. If you were unable to find a recurrence relation in question 1, use the recurrence

$$P[i,t] = \begin{cases} \text{False} & \text{if } t < 0 \text{ or } t > L \\ \text{True} & \text{if } t \geq 0 \text{ and } t \leq L \text{ and } i \leq 2 \\ P[i-1,t] \oplus (P[i-2,t-3] \wedge P[i-2,t+5] & \text{if } t \geq 0 \text{ and } t \leq L \text{ and } i \geq 3 \end{cases}$$

3. **[2 marks]** What is the running time of your algorithm from question 2 as a function of n and L? Note: describing a trivial (and totally irrelevant) algorithm in question 2 may result in not getting full marks for question 3.

4. **[2 marks]** Once you have computed the table, how do you determine whether or not the Faculty member can attempt the ritual in the hallway in front of their office?

5. **[3 marks]** Explain briefly how, in the case where the hallway is long enough for the ritual to be attempted, you would determine the starting point where the Faculty member should stand, and in which direction they should walk for each stage of the ritual.

6 Twinkle, twinkle, little star

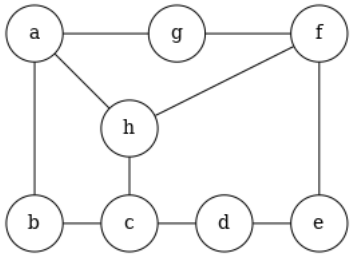
In this problem, we consider two problems: the *Independent Set* problem

> Given a graph $G = (V, E)$ and a positive integer K, is there a subset V' of V containing at least K vertices, no two of which are joined by an edge in G.

which is known to be NP-complete, and the *K-star* problem

> Given a graph $G = (V, E)$ and a positive integer K, does G contain a K-star: a subset V' of V such that $|V'| = K$, and that contains a vertex v connected to every other vertex of V', where no vertex of V' other than v is connected to any other vertex of V'.

For instance, in the following graph, there are many independent sets with 3 vertices, for instance: $\{a, d, f\}$, $\{b, d, f\}$ and $\{b, d, g\}$ (there are more). This graph also contains several 4-stars, for instance: $\{a, b, g, h\}$, $\{a, c, f, h\}$ and $\{b, c, d, h\}$.



We will prove that the *K-star* problem is also NP-complete.

1. **[2 marks]** Once we have proved that the K-star problem is NP-complete, what will we know about this problem?

   a. ◯ No algorithm can solve the K-star problem in polynomial time.
   b. ◯ There is an algorithm that solves the K-star problem in polynomial time.
   c. ◯ If there is an algorithm that solves the K-star problem in polynomial time, then the 3SAT and Vertex Cover problems can be solved in polynomial time.
   d. ◯ None of the above.

2. **[2 marks]** Which of the following will we need to do in order to prove that the K-star problem is NP-complete (choose all answers that apply)?

   a. ◯ Prove that if the answer to the problem is Yes, then there is a "proof" ("certificate") of this fact that can be verified in polynomial time.
   b. ◯ Prove that if the answer to the problem is No, then there is a "proof" ("certificate") of this fact that can be verified in polynomial time.
   c. ◯ Show how to reduce a known NP-complete problem (for instance, Independent Set) to the K-star problem in polynomial time.
   d. ◯ Show how to reduce the K-star problem to a known NP-complete problem (for instance, Independent Set) in polynomial time.
   e. ◯ None of the above.

3. **[3 marks]** Describe **one** of the following:

   (a) A possible "proof" (certificate) for a Yes answer to the K-star problem.

   (b) A possible "proof" (certificate) for a No answer to the K-star problem.

   Explain briefly how to check this certificate in polynomial time.

4. **[4 marks]** Show how to reduce an arbitrary instance of the Independent Set problem to an instance of the K-star problem with the same answer. (Note: We are asking for only one direction on this test (the reduction from Independent Set to K-star) because it's simpler than the reduction in the other direction. We are not implying that this direction is needed for the proof of NP-completeness — it might be; it might not be. Do not assume that the question structure here gives away answers on part 2 of this question.)

   Hint: you will want to add exactly one vertex to the graph in the instance of Independent Set.

5. **[5 marks]** Prove that the answer to the instance of the Independent Set problem is Yes if and only if the answer to the instance of the K-star problem resulting from your reduction is Yes.