Due November 3 (11:59 p.m.)

Instructions

- **Answers should be written in the boxes by modifying the provided Latex template or some other method answers are clearly marked and legible. Submit via Gradescope.**
- **Honors questions are optional. They will not count towards your grade in the course. However you are encouraged to submit your solutions to these problems to receive feedback on your attempts.**
- **You must enter the names of your collaborators or other sources (including use of LLMs) as a response to Question 0. Do NOT leave this blank; if you worked on the homework entirely on your own, please write "None" here. Even though collaborations in groups of up to 3 people are encouraged, you are required to write your own solution.**
- **For the late submission policy, see the website.**
- Acknowledgment - Thank you to Amir, Daji, Oded, Peter and Rotem for help in setting the problem set.

1-0 List all your collaborators and sources: ($-\infty$ points if left blank)

tutor, TX, chatgpt

Problem 7-1 – Huffman Coding (10 Points)

Suppose our alphabet has symbols $1, \ldots, n$, and we construct a Huffman code as shown in class. Give an example of frequencies $f_1, \ldots, f_n$ for these symbols, for which we will get the following tree:

1. (2 points) A perfectly-balanced tree, where all n leafs have the same depth. (Assume that n is a power of 2.)

Solution: The frequencies for $f_1, \ldots, f_n$ could all be equal to yield a perfectly balanced tree. All the frequencies could be 1, for example.

2. (4 points) A tree where $n-1$ leafs have the same depth, and one leaf has a different depth from the others. (Assume that n is of the form $2^i + 1$ for some $i \geq 1$.)

> **Solution:** We could give equal frequencies to 2^i symbols and then give a smaller frequ. to the last one so that the huffman algo merges it last. So an example: set i = 2, so $n = 2^2 + 1 = 5$.The frequencies would be $f_1 \to f_4 = 10$ and $f_5 = 1$.

3. (4 points) A tree with exactly one leaf at each depth $1, \ldots, n-2$ and two leafs at depth $n-1$.

> **Solution:** Impt to note that the two smallest frequencies will be merged first and the expectation is that their sum will be smaller than the next frequency, so we put the two smallest leaves at the bottom, n-1. Let's say $n = 6$, which implies one leaf at depth levels 1-4 and then two leaves at depth 6. The frequences would be $f_1 = 1, f_2 = 2, f_3 = 4, f_4 = 8, f_5 = 16, f_6 = 32$. This should result in one leaf per level and then two leaves at the deepest.

Honors problem: (***) Prove Kraft's inequality, which asserts that for any collection of lengths $\ell_1, \ldots, \ell_n$, there is a (binary) prefix code where the encodings have lengths $\ell_1, \ldots, \ell_n$ if and only if $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$.

Solution:

Problem 7-2 – Optimal Prefix Trees (30 Points)

Let S be an alphabet with frequencies $\{f_x\}_{x \in S}$, and let T be a prefix code represented as a tree, and let $\mathsf{depth}_T(x)$ denote the depth of symbol x in the tree T. Recall that the ABL of tree T is defined as follows:

$$\mathrm{ABL}(T) = \sum_{x \in S} f_x \cdot \mathsf{depth}_T(x).$$

Suppose that T is an *optimal code*, i.e., it has minimal ABL among all prefix codes representing S with the given frequencies. Formally prove the following claims, which were presented in class.

(Hint: prove these claims *by contradiction.*)

1. (10 points) For any two symbols $x, y \in S$, if $f_x < f_y$, then $\mathsf{depth}_T(x) \geq \mathsf{depth}_T(y)$.

   Solution: Assume that $f_x < f_y$ but that $depth_T(x) < depth_T(y)$. Take the tree that results from T by swapping the symbols placed at the two leaves that hold x and y (leaves are swapped, structure unchanged). Only the two terms for x and y in the ABL change. If we take the assumption that $(f_x - f_y) < 0$ and $depth_T(y) - depth_T(x) > 0$, the product will be negative. Therefore, ABL(resulting tree from T) $< ABL(T)$, which contradicts that T is optimal. This proves that the given statement is true.

2. (10 points) T does not contain any inner vertex that has only one child.

   Solution: Take v as this internal vertext of T which actually has only one child, u. Let's also say that $S_u \subseteq S$ be the leaves in the subtree rooted at u. If you were to delete v and replace it with u, then every leaf in S_u will move up one level, decreasing their depths throughout by 1. Assuming the frequencies aren't negative and that some leaf exists in S_u, then $\sum_{x \in S_u} f_x > 0$, which results in ABL(resulting tree from T) < ABL(T). This wouldn't make T optimal.

3. (10 points) T has two leafs that are siblings (i.e., have the same parent).

Solution: Not sure if we need a contradiction here. If you just take one such deepest leaf on the tree, the assumption is that its parent will have two children and the depth of the other child can't exceed the depth of the deepest leaf that we picked. So it has to be equal and a child of the parent. Therefore, T contains two leaves that are siblings.

Problem 7-3 – Huffman's Greedy Choice Property (30 points)

Prove the Greedy Choice Property of Huffman Coding:

Let S be an alphabet with frequencies $\{f_x\}_{x \in S}$, and let $y, z \in S$ be two symbols with the smallest frequencies in S. Then there exists an *optimal code* T (represented as a tree) where y, z are sibling leafs.

(Hint: review the structure of the proof of the greedy choice property for Interval Scheduling.)

Solution: Let T' be any optimal prefix-code tree, minimum ABL. If they're already siblings, there's nothing to do. Otherwise, pick any pair of sibling leaves in T'. Among all the sibling pairs, we can take one (a,b) with symbols (u,v) which has a maximum depth (a deepest sibling pair). And we can just say their depth (because they're at the same level) is d. The assumption is that such a deepest sibling pair exists because the tree is going to end at some point. Now form a new tree from T' which is the same in terms of shape but we swap the labels so that 'a' has label y and 'b' has label z. Only the two terms for the affected leaves change. Because y and z are the two least frequent symbols in S, we have $f_y \leq f_x$ and $f_z \leq f_x$ for every symbol x and $f_y \leq f_u$ and $f_z \leq f_v$. Adding these inequalities results in $ABL(T) \leq ABL(T')$. Since T' was optimal, there has to be equality, meaning T is also optimal but just that T has z and y as siblings at depth d. Therefore, there exists an optimal tree with y/z as siblings.

Problem 7-4 – Baseball! (30 Points)

You are the manager of a baseball team and want to decide which of your m pitchers are going to pitch which innings (one pitcher per inning). Your pitchers follow the following rules:

- Each pitcher may pitch multiple innings if they are able.
- Each pitcher j has starting effectiveness e_j, where $e_j \in \mathbb{R}^+$ (i.e., all starting effectiveness are positive).
- The first inning a pitcher pitches, they have effectiveness e_j. Every time thereafter, though, their effectiveness is reduced by t_j (where this constant may vary for different pitchers). This means that the effectiveness for pitcher j in its i-th inning pitched is given by:

$$e_{j,i} = e_j - (i-1) \cdot t_j$$

- A pitcher may not pitch if their effectiveness for that inning would be nonpositive (i.e., 0 or negative).

As an example, if a pitcher has $e_j = 100$ and $t_j = 10$, their effectiveness for their 1st inning is 100, their 2nd is 90, their 3rd is 80, and so on. They would be able to pitch up to 10 innings, but not 11 (as in the eleventh inning their effectiveness would be 0). Assume there exists a feasible solution (i.e., we can pick pitchers for all m innings such that all innings will have a pitcher with positive effectiveness). Your goal is to maximize the total effectiveness summed across all innings.

1. (10 points) Assume that you can put pitchers into the game as many times as you like. The pitchers do not have enough time to rest (i.e., their effectiveness rating does not go back up and depends only on the total number of innings they have played). Write an efficient algorithm to determine who should pitch at each inning.

Solution: Input: e[1..n], t[1..n], m, implies n pitchers, m innings (assume feasible)
Output: P[1..m], implies P[k] = index of pitcher for inning k

for j = 1..n: count[j] = 0, val = e[j] - count[j] * t[j] and if val > 0: push maxheap (val,j)
for inning = 1..m: (val,j) = popMax(maxheap), implies largest current effectiveness
P[inning] = j
count[j] = count[j] + 1
nextval = e[j] - count[j] * t[j], implies next efficacy for this pitcher
if nextval > 0: push maxheap (nextval, j)
return P

2. (4 points) In your answer above, do you need to know how many innings will be played? (Recall we assume that you are guaranteed to have effective pitchers for the whole game) Why or why not?

Solution: Yes we need to knw the number of innings played because it'll determine how many times to repeat selection process. We need it to basically decide when to stop. The algo overall is picking the top m effectiveness values across all possible innings from all pitchers, so it'll keep picking the best available pitcher until all the innings are filled, at which pt it should stop.

3. (6 points) We now assume that the pitchers cannot return to the game once they have left. They can still pitch for as many consecutive innings as you specify, but once you take them out, they are done. Suppose there is a fixed number of innings n that you know ahead of time. Specify an efficient algorithm that determines the best possible pitching lineup. (Note: You just need to determine the set of (pitcher, consecutive innings pitched) pairs that sum to n total innings. The total effectiveness is the sum of these individual contributions, regardless of the order they appear in the game).

Solution: This means that for any fixed j that the marginals $m_{j,1}, m_{j,2}$, the marginals are either decreasing or remaining constant. That means that any selection that takes some marginals for a given pitcher is going to have to take a prefix of that pitcher's marginals.

function BestLineup(e[1..p], t[1..p], n):
for j = 1..p: count[j] = 0
heap = empty max-heap, implies the max heap keyed by current marginal value, (value, pitcher)
for j = 1..p: if e[j] > 0: push(heap, (e[j], j)), assumes mj,1 = ej
for inning = 1..n: (allocates innings by repeatedly taking largest marg)
if heap is empty: throw error but otheriwse:
popMax(heap) = (val,j)
count[j] + 1 = count[j]
push next marginal if pos and does not exceed Kj
e[j] - (count[j] * t[j]) = next-val, this is mj,count[j]+1
if next-val > 0: push(heap, (nextVal,j))
now count[j] is number of consecutive innings to assign to pitcher j
build list of (pitcher, consecutive innings) pairs:
empty list = pairs
for j = 1..p: if count[j] > 0: append(j, count[j]) to pairs, return pairs

4. (10 points) Now assume the number of innings is not known ahead of time (as is often the case in real baseball games), but you still cannot put pitchers back in. Will an inning-by-inning greedy algorithm (e.g., "always make the move that provides the highest effectiveness for the *current* inning") be guaranteed to produce the maximum possible total effectiveness, no matter when the game ends? If yes, prove it, if not, provide a counter example.

Solution: No, it's not guaranteed to produce the maximum possible total effectiveness regardless of when the game ends. If we use an inning-by-inning greedy algo, it would be picking the pitcher with the highest current effectiveness that could throw off the optimal total if the game suddenly ends at an unoptimal time. I don't think you can do a universal guarantee of sorts because instances exist where the prefix-optimal choice for horizon k will differ from the prefix-optimal choice for horizon (k+1), which we don't account for since we're doing it inning-by-inning. It's why an inning-by-inning algo cannot be guaranteed to be optimal for every possible stopping time of the game.

Counterexample:
Pitcher with per-inning efficacy sequence:
A: 11, 10, 9
B: 10, 1, 0
C: 9, 9, 9

Now take two possible strats for the first two innings:
1. Greedy: picking highest current each inning. Inning 1 picks 9. For inning 2, A gives 10, B = 1, and C = 9. So we pick B which gives us 10, and so on, giving us a sum of 11+10+9 = 30. Greedy picks A(11) then sees B(10) and may switch to B producing later loss; but optimal for some fixed stopping time may have been A,A... to produce a higher effectiveness. If we had stuck with A throughout, the total (30) would be equal (for small n in this case) but if you adjust for tie-breaking or consider an early end to the game at just 2 innings, switching to B early can strictly reduce the total (11+10=21).