CS/ECE 374 Sec A ✧ Spring 2023

Homework 7

Due Wednesday, March 22, 2023 at 10am

---

1. A sequence $A = a_1, a_2, \ldots, a_n$ is a *supersequence* of a sequence $B = b_1, b_2, \ldots, b_m$ if and only if B is a *subsequence* of A. More formally, A is a supersequence of B if there exist indices $1 \leq i_1 < i_2 \ldots < i_m \leq n$ such that $a_{i_1} = b_1, a_{i_2} = b_2, \ldots, a_{i_m} = b_m$. A sequence $A = a_1, a_2, \ldots, a_n$ is *palindromic* if A and its reverse are the same sequence, that is, $a_{n-i+1} = a_i$ for $i = 1, 2, \ldots, n$. Examples of palindromic strings (recall strings are sequences over an alphabet) are A, MALAYALAM, KAYAK, 374473, 47374.

   - Describe an efficient algorithm that outputs the length of the shortest palindromic supersequence of a given sequence A. For example if SUPERSEQUENCE is the input string (viewed as a seqeunce), your algorithm should return 21 (the length of SUPECNRSEQUQESRNCEPUS). Assume A is given as an array of length n. For full credit explain how to compute the optimum value in $O(n)$ space.
   - Let $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$ be two sequences. Describe an efficient algorithm to compute the length of the shortest *common palindromic supersequence* of X and Y. That is, if the answer is h then there is a palindromic sequence Z of length h such that Z is a supersequence of X and Z is a supersequence of Y. For example if X is the string 374 and Y is the string 473 then the answer should be 5 (the length of 37473 or that of 47374).

2. Given a graph $G = (V, E)$ a matching is a subset of edges in G that do not *intersect*. More formally $M \subseteq E$ is a matching if every vertex $v \in V$ is incident to at most one edge in M. Matchings are of fundamental importance in combinatorial optimization and have many applications. Given G and non-negative weights $w(e), e \in E$ on the edges one can find the maximum weight matching in a graph in polynomial time but the algorithm requires advanced machinery and is beyond the scope of this course. However, finding the maximum weight matching in a tree is easier via dynamic programming.

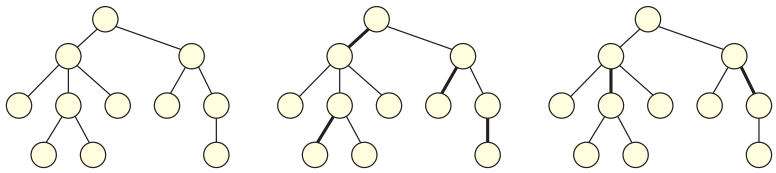

Figure 1. A tree and two examples of matchings. Edges in the matching are shown in bold.

   - **Not to submit:** Given a tree $T = (V, E)$ and non-negative weights $w(e), e \in E$, describe an efficient algorithm to find the weight of a maximum weight matching in T.
   - Solve the previous part even though it is not required to be submitted for grading. It will help you think about this part.

(a) Given a tree $T = (V, E)$ describe an efficient algorithm to *count* the number of distinct matchings in T. Two matchings M_1 and M_2 are distinct if they are not identical as sets of edges. Unlike the maximum weight matching problem, the problem of couting matchings is known to be hard in general graphs, but trees are easier.

(b) Write a recurrence for the exact number of matchings in a path on n nodes. For the base case of a single node tree assume that the answer is 1 since an empty set is also a valid matching. Would the answer for a path with $n = 500$ fit in a 64-bit integer word? Briefly justify your answer.

(c) How would you implement your counting algorithm from part (a), more carefully, to run on a 64 bit machine? Accounting for this more careful implementation, what is the running time of your algorithm?

(d) Now consider the following generalization of the maximum weight matching problem. Given T and integer k describe an efficient algorithm that find the weight of a maximum weight matching with at most k edges in it. For example if the tree is a path with three edges with weights $2, 3, 2$ respectively then the maximum weight matching is 4 (we take the first and third edges) while the maximum weight matching with $k = 1$ has weight 3 (consisting of the second edge).

3. **Not to submit:** This is a cute problem from our CA Alex.

After surviving the farmers, Mr. Fox is off on a United States tour, stretching from Seattle, Washington to Orlando, Florida, with a whole list of cities to visit in order. At certain cities he will simply say "Ring" and move on, while at others, he will say "Ding" and spend a night in the city. Because he is so famous, some cities will pay him to stay, while at other cities he has to rent a hotel room. However, Mr. Fox cannot say "Ring" in too many cities in a row, because he will then get tired. Note that Mr. Fox must say "Ding" in Orlando, Florida, his last stop. Given an array A of length n, where $A[i]$ denotes the cost for spending the night in city i (it can be negative, indicating Mr. Fox gets paid), and an integer k denoting the maximum number of cities in a row that he can say "Ring" in, return the maximum profit he can make on this trip. Express your runtime as a function of n and k. Full credit for a solution that uses $O(n)$ space and time.

4. **Not to submit:** You are given n jobs each of which has a start time s_i and an end time t_i. To complete a job it needs to be worked on by a server continuously from its start time to its end time without interruption. Completing job i earns you a profit p_i. You have two servers available. Describe an efficient algorithm to find the maximum profit that you can earn by using these servers.

Solved Problems

5. A string w of parentheses **(** and **)** and brackets **[** and **]** is ***balanced*** if it is generated by the following context-free grammar:

$$S \to \varepsilon \mid \mathbf{(}S\mathbf{)} \mid \mathbf{[}S\mathbf{]} \mid SS$$

For example, the string $w = \mathbf{([()][][])[()()]()}$ is balanced, because $w = xy$, where

$$x = \mathbf{(\,[()]\,[]\,()\,)} \qquad \text{and} \qquad y = \mathbf{[\,()\,()\,]\,()}.$$

Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets. Your input is an array $A[1..n]$, where $A[i] \in \{\mathbf{(}, \mathbf{)}, \mathbf{[}, \mathbf{]}\}$ for every index i.

Solution: Suppose $A[1..n]$ is the input string. For all indices i and j, we write $\boldsymbol{A[i] \sim A[j]}$ to indicate that $A[i]$ and $A[j]$ are matching delimiters: Either $A[i] = \mathbf{(}$ and $A[j] = \mathbf{)}$ or $A[i] = \mathbf{[}$ and $A[j] = \mathbf{]}$.

For all indices i and j, let $\boldsymbol{LBS(i,j)}$ denote the length of the longest balanced subsequence of the substring $A[i..j]$. We need to compute $LBS(1,n)$. This function obeys the following recurrence:

$$LBS(i,j) = \begin{cases} 0 & \text{if } i \geq j \\ \max\left\{ \begin{array}{c} 2 + LBS(i+1, j-1) \\ \max\limits_{k=1}^{j-1} \big(LBS(i,k) + LBS(k+1,j)\big) \end{array} \right\} & \text{if } A[i] \sim A[j] \\ \max\limits_{k=1}^{j-1} \big(LBS(i,k) + LBS(k+1,j)\big) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $LBS[1..n, 1..n]$. Since every entry $LBS[i,j]$ depends only on entries in later rows or earlier columns (or both), we can evaluate this array row-by-row from bottom up in the outer loop, scanning each row from left to right in the inner loop. The resulting algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

```
LongestBalancedSubsequence(A[1..n]):
  for i ← n down to 1
      LBS[i,i] ← 0
      for j ← i+1 to n
          if A[i] ~ A[j]
              LBS[i,j] ← LBS[i+1,j-1] + 2
          else
              LBS[i,j] ← 0
          for k ← i to j-1
              LBS[i,j] ← max{LBS[i,j], LBS[i,k] + LBS[k+1,j]}
  return LBS[1,n]
```

■

Rubric: 10 points, standard dynamic programming rubric

6. Oh, no! You've just been appointed as the new organizer of Giggle, Inc.'s annual mandatory holiday party! The employees at Giggle are organized into a strict hierarchy, that is, a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how "fun" the employee is. In order to keep things social, there is one restriction on the guest list: An employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all.

   Describe an algorithm that makes a guest list for the party that maximizes the sum of the "fun" ratings of the guests. The input to your algorithm is a rooted tree T describing the company hierarchy, where each node v has a field $v.fun$ storing the "fun" rating of the corresponding employee.

   Solution (two functions): We define two functions over the nodes of T.

   - *MaxFunYes*(v) is the maximum total "fun" of a legal party among the descendants of v, where v is definitely invited.
   - *MaxFunNo*(v) is the maximum total "fun" of a legal party among the descendants of v, where v is definitely not invited.

   We need to compute *MaxFunYes*$(root)$. These two functions obey the following mutual recurrences:

$$\textit{MaxFunYes}(v) = v.\textit{fun} + \sum_{\text{children } w \text{ of } v} \textit{MaxFunNo}(w)$$

$$\textit{MaxFunNo}(v) = \sum_{\text{children } w \text{ of } v} \max\{\textit{MaxFunYes}(w), \textit{MaxFunNo}(w)\}$$

   (These recurrences do not require separate base cases, because $\sum \varnothing = 0$.) We can memoize these functions by adding two additional fields $v.yes$ and $v.no$ to each node v in the tree. The values at each node depend only on the vales at its children, so we can compute all $2n$ values using a postorder traversal of T.

```
BestParty(T):
  ComputeMaxFun(T.root)
  return T.root.yes
```

```
ComputeMaxFun(v):
  v.yes ← v.fun
  v.no ← 0
  for all children w of v
      ComputeMaxFun(w)
      v.yes ← v.yes + w.no
      v.no ← v.no + max{w.yes, w.no}
```

   (Yes, this is still dynamic programming; we're only traversing the tree recursively because that's the most natural way to traverse trees![1]) The algorithm spends $O(1)$ time at each node, and therefore runs in $\boldsymbol{O(n)}$ ***time*** altogether. ■

[1] A naïve recursive implementation would run in $O(\phi^n)$ time in the worst case, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio. The worst-case tree is a path—every non-leaf node has exactly one child.

Solution (one function): For each node v in the input tree T, let $\mathit{MaxFun}(v)$ denote the maximum total "fun" of a legal party among the descendants of v, where v may or may not be invited.

The president of the company must be invited, so none of the president's "children" in T can be invited. Thus, the value we need to compute is

$$\mathit{root.fun} + \sum_{\text{grandchildren } w \text{ of } \mathit{root}} \mathit{MaxFun}(w).$$

The function *MaxFun* obeys the following recurrence:

$$\mathit{MaxFun}(v) = \max \left\{ \begin{array}{l} v.\mathit{fun} + \displaystyle\sum_{\text{grandchildren } x \text{ of } v} \mathit{MaxFun}(x) \\ \displaystyle\sum_{\text{children } w \text{ of } v} \mathit{MaxFun}(w) \end{array} \right\}$$

(This recurrence does not require a separate base case, because $\sum \varnothing = 0$.) We can memoize this function by adding an additional field $v.\mathit{maxFun}$ to each node v in the tree. The value at each node depends only on the values at its children and grandchildren, so we can compute all values using a postorder traversal of T.

```
BestParty(T):
  ComputeMaxFun(T.root)
  party ← T.root.fun
  for all children w of T.root
      for all children x of w
          party ← party + x.maxFun
  return party
```

```
ComputeMaxFun(v):
  yes ← v.fun
  no ← 0
  for all children w of v
      ComputeMaxFun(w)
      no ← no + w.maxFun
      for all children x of w
          yes ← yes + x.maxFun
  v.maxFun ← max{yes, no}
```

(Yes, this is still dynamic programming; we're only traversing the tree recursively because that's the most natural way to traverse trees![2])

The algorithm spends $O(1)$ time at each node (because each node has exactly one parent and one grandparent) and therefore runs in $\boldsymbol{O(n)}$ ***time*** altogether. ■

Rubric: 10 points: standard dynamic programming rubric. These are not the only correct solutions.

Standard dynamic programming rubric. 10 points divided as follows

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - -1 for naming the function "OPT" or "DP" or any single letter.
  - – No credit if the description is inconsistent with the recurrence.
  - – No credit if the description does not explicitly describe how the function value depends on the named input parameters.

[2] Like the previous solution, a direct recursive implementation would run in $O(\phi^n)$ time in the worst case, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio.

  - No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns, not (here) an explanation of *how* that value is computed.

4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.

  1 for base case(s). $-½$ for one *minor* bug, like a typo or an off-by-one error.

  3 for recursive case(s). -1 for each *minor* bug, like a typo or an off-by-one error.

  -2 for greedy optimizations without proof, even if they are correct.

  - **No credit for the rest of the problem if the recursive case(s) are incorrect.**

3 points for iterative details

  + 1 for describing an appropriate memoization data structure
  + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order.
  + 1 for correct time analysis. (It is not necessary to state a space bound.)

- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative pseudocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10.***
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points ($=$ 70% of 3, rounded).