HW 6: Extra problems

Instructors: Har-Peled

CS/ECE 374A: Intro. Algorithms & Models of Computation, Fall 2022 Version: **1.0**

No solutions for the following problems will NOT be provided but you can discuss them on Piazza.

1 Given an array of n unsorted integers A and k ranks $i_1 < i_2 < \ldots < i_k$ describe an algorithm that outputs the elements in A with these given k ranks. Your algorithm should run in $O(n \log k)$ time. One can easily do this via sorting in $O(n \log n)$ time. There is also an $O(nk)$ time algorithm (how?).

2 Problems in Jeff's notes on dynamic programming. In particular, Probs 1, 2, 3, 5, 6.

3 Problems in Dasgupta etal book Chapter 6. In particular Probs 1, 2

4 Problems in Kleinberg-Tardos book Chapter 6. Problems 1, 2, 7.

5 Let $w \in \Sigma^*$ be a string. We say that $u_1, u_2, \ldots, u_h$ where each $u_i \in \Sigma^*$ is a valid split of w iff $w = u_1 u_2 \ldots u_h$ (the concatenation of $u_1, u_2, \ldots, u_h$). Given a valid split $u_1, u_2, \ldots, u_h$ of w we define its ℓ_3 measure as $\sum_{i=1}^{h} |u_i|^3$.

Given a language $L \subseteq \Sigma^*$ a string $w \in L^*$ iff there is a valid split $u_1, u_2, \ldots, u_h$ of w such that each $u_i \in L$; we call such a split an L-valid split of w. Assume you have access to a subroutine IsStringInL(x) which outputs whether the input string x is in L or not. To evaluate the running time of your solution you can assume that each call to IsStringInL() takes constant time.

Describe an efficient algorithm that given a string w and access to a language L via IsStringInL(x) outputs an L-valid split of w with minimum ℓ_3 measure if one exists.

6 Recall that a *palindrome* is any string that is exactly the same as its reversal, like *I*, or *DEED*, or *RACECAR*, or *AMANAPLANACATACANALPANAMA*.

Any string can be decomposed into a sequence of palindrome substrings. For example, the string *BUBBASEESABANANA* ("Bubba sees a banana.") can be broken into palindromes in the following ways (among many others):

$$BUB \bullet BASEESAB \bullet ANANA$$
$$B \bullet U \bullet BB \bullet A \bullet SEES \bullet ABA \bullet NAN \bullet A$$
$$B \bullet U \bullet BB \bullet A \bullet SEES \bullet A \bullet B \bullet ANANA$$
$$B \bullet U \bullet B \bullet B \bullet A \bullet S \bullet E \bullet E \bullet S \bullet A \bullet B \bullet ANA \bullet N \bullet A$$

Describe and analyze an efficient algorithm that given a string w and an integer k decides whether w can be split into palindromes each of which is of length at least k. For example, given the input string *BUBBASEESABANANA* and 3 your algorithm would answer yes because one can find a split *BUB* • *BASEESAB* • *ANANA*. The answer should be no if we set $k = 4$. Note that the answer is always yes for $k = 1$.

7 The McKing chain wants to open several restaurants along Red street in Shampoo-Banana. The possible locations are at $L_1, L_2, \ldots, L_n$ where L_i is at distance m_i meters from the start of Red street. Assume that the street is a straight line and the locations are in increasing order of distance from the starting point (thus $0 \leq m_1 < m_2 < \ldots < m_n$). McKing has collected some data indicating that opening a restaurant at location L_i will yield a profit of p_i independent of where the other restaurants are located. However, the city of Shampoo-Banana has a zoning law which requires that any two McKing locations should be D or more meters apart. Describe an algorithm that McKing can use to figure out the maximum profit it can obtain by opening restaurants while satisfying the city's zoning law.

Solved Problem

8 A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string $BANANAANANAS$ is a shuffle of the strings `BANANA` and $ANANAS$ in several different ways.

$$\underline{BANANA}ANANAS \qquad \underline{BAN}ANA\underline{ANA}NAS \qquad \underline{B}AN\underline{AN}A\underline{A}NA\underline{NA}S$$

Similarly, the strings $PRODGYRNAMAMMIINCG$ and $DYPRONGARMAMMICING$ are both shuffles of `DYNAMIC` and $PROGRAMMING$:

$$PRO\underline{D}G\underline{Y}R\underline{NAM}AMMI\underline{I}N\underline{C}G \qquad \underline{DY}PRO\underline{N}G\underline{A}R\underline{M}AMM\underline{IC}ING$$

Given three strings $A[1\,..\,m]$, $B[1\,..\,n]$, and $C[1\,..\,m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

Solution: We define a boolean function *Shuf*(i, j), which is TRUE if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. This function satisfies the following recurrence:

$$Shuf(i,j) = \begin{cases} \text{TRUE} & \text{if } i = j = 0 \\ Shuf(0, j-1) \wedge (B[j] = C[j]) & \text{if } i = 0 \text{ and } j > 0 \\ Shuf(i-1, 0) \wedge (A[i] = C[i]) & \text{if } i > 0 \text{ and } j = 0 \\ \big(Shuf(i-1, j) \wedge (A[i] = C[i+j])\big) & \\ \qquad \vee \big(Shuf(i, j-1) \wedge (B[j] = C[i+j])\big) & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

We need to compute *Shuf*(m, n).
We can memoize all function values into a two-dimensional array *Shuf*$[0..m][0..n]$. Each array entry *Shuf*$[i, j]$ depends only on the entries immediately below and immediately to the right: *Shuf*$[i-1, j]$ and *Shuf*$[i, j-1]$. Thus, we can fill the array in standard row-major order. The original recurrence gives us the following pseudocode:

```
Shuffle?($A[1..m]$, $B[1..n]$, $C[1..m+n]$):
$Shuf[0,0] \leftarrow$ TRUE
for $j \leftarrow 1$ to $n$
   $Shuf[0,j] \leftarrow Shuf[0,j-1] \wedge (B[j] = C[j])$
for $i \leftarrow 1$ to $n$
   $Shuf[i,0] \leftarrow Shuf[i-1,0] \wedge (A[i] = B[i])$
   for $j \leftarrow 1$ to $n$
      $Shuf[i,j] \leftarrow$ FALSE
      if $A[i] = C[i+j]$
         $Shuf[i,j] \leftarrow Shuf[i,j] \vee Shuf[i-1,j]$
      if $B[i] = C[i+j]$
         $Shuf[i,j] \leftarrow Shuf[i,j] \vee Shuf[i,j-1]$
return $Shuf[m,n]$
```

The algorithm runs in $\boldsymbol{O(mn)}$ ***time***.

Rubric: Max 10 points: Standard dynamic programming rubric. No proofs required. Max 7 points for a slower polynomial-time algorithm; scale partial credit accordingly.

9 (100 PTS.) Feline indiscretion advised.

Real cats plan out their day by drawing action cards from a deck. Each time they draw a card, they must either perform the action indicated by the card, or else discard it. The possible action cards are Nap, Yawn, Eat, Stretch, and Climb. There are a few rules governing the sequences of actions they perform:

- A Nap can only be followed by a Yawn or a Stretch.
- Eat can only follow a Climb.

Other than these rules, cats are free to choose any *subsequence* of action cards from the deck of cards they are given.

For example, given the following deck of cards

$$N, C, Y, S, C, E, E$$

The subsequence N, C, E would not be compatible with the rules (since Nap was followed by something other than Yawn or Stretch. One longest acceptable subsequence would be N, Y, S, C, E.

9.A. (50 PTS.) Given a deck of n action cards, give a backtracking algorithm for computing the length of the longest sequence of actions compatible with the rules above. Describe the asymptotic running time as a function of n.

9.B. (50 PTS.) Given a deck of n action cards, give a dynamic programming algorithm for computing the length of the longest sequence of actions compatible with the rules above. Describe the asymptotic running time as a function of n. The running time and space used by your algorithm should be as small as possible.

10 (100 PTS.) Cutting strings.

Assume you have an oracle that can query assess the "quality" of strings: $\mathbf{q} : \Sigma^* \to \mathbb{N}$. Computing $\mathbf{q}$ using the oracle takes $O(1)$ time regardless of the size of the string.

The following is an example $\mathbf{q}$ function (please note this is just an example, your algorithms must work for *any* $\mathbf{q}$ function):

$$\mathbf{q}(CATDOG) = 10, \qquad \mathbf{q}(CAT) = 9, \qquad \mathbf{q}(DOG) = 10, \qquad \mathbf{q}(ATDO) = 15,$$

where $\mathbf{q}(\text{everything else}) = 0$. For the given input string $x[1 \ldots n]$, you may invoke the $\mathbf{q}$ function by passing indexes. Thus, $\mathbf{q}(i, j)$ is a shorthand for $\mathbf{q}(x[i...j])$.

For each of the following, it is sufficient to describe how to compute the value realizing the desired optimal solution.

10.A. (20 PTS.) Give an algorithm to compute the highest quality substring of x.

For example, for the $\mathbf{q}$ function defined above, the highest quality substring of the string $CATDOG$ is 15 (because the highest quality substring is $ATDO$). Analyze the performance of your algorithm. For full credit, you need to *prove* that your algorithm achieves the best possible asymptotic efficiency.

10.B. (40 PTS.) A *decomposition* of string x is a sequence of **non-empty** substrings $x_1, x_2, ..., x_k$ such that $x = x_1 x_2 ... x_k$. The *quality* of such a decomposition is

$$\mathbf{Q}(x_1, \ldots, x_k) = \min_i \mathbf{q}(x_i).$$

Namely, the quality is determined by the worst substring the decomposition uses.

Give a dynamic programming algorithm (and analyze its performance) to compute the decomposition that **maximizes** the quality $\mathbf{Q}$ of the decomposition (yeh, this is a bit confusing).

For example, if the string is $CATDOG$, then the best possible min-quality is 9. (CAT, DOG is a decomposition that achieves this).

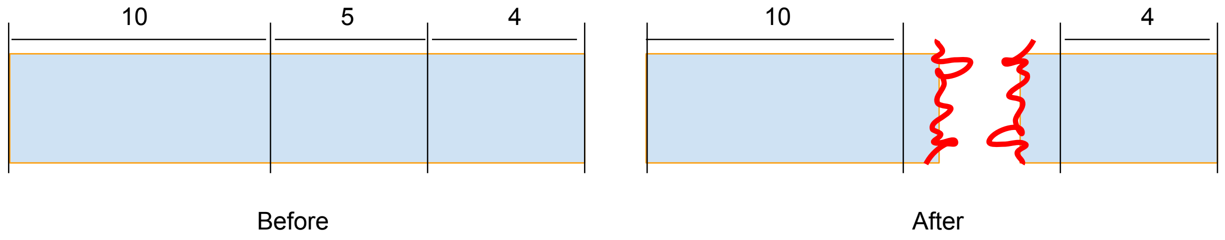
10.C. (40 PTS.) Give a dynamic programming algorithm (and analyze its performance) to find the best possible *average* quality of the **non-empty** substrings in a decomposition (that is, the sum quality of the substrings in a decomposition divided by the number of substrings in the decomposition).

For example, continuing the above examples, the best average quality decomposition for the string $CATDOG$ is 10 (the decomposition $CATDOG$ achieves this).

11 (100 PTS.) Laser Cuttery

You are in charge of programming the laser cutter at the local Maker Space. A member of the Maker Space needs your help to cut wood for a project. You're given a piece of wood to cut into smaller pieces. The wood is already marked according to positive integer-length segments, the lengths of which are given to you as a sequence.

Your laser cutter isn't tuned all that well, such that the only way to cut a piece of wood into two smaller pieces is to burn through one of the marked segments, destroying it. For example, the following illustrates the result of cutting a piece of wood marked as $[10, 5, 4]$ into two smaller pieces $[10]$ and $[4]$.



The cost of each cut is proportional to the *total length* of the piece being cut. Thus the cut depicted above would cost \$19. (Assume after cutting into two smaller pieces, you can file the ends down for free).

It is also possible to cut off a segment at the end, so for example you could cut $[5, 4]$ into $[5]$, for a cost of \$9.

The project that hired you can only make use of pieces of wood length 5 or smaller. Anything larger is unusable. For example, pieces $[2, 3], [1], [5]$ would all be usable, but $[5, 4]$ would have to be thrown out or cut down further.

11.A. (40 PTS.) Suppose you need to produce usable wood pieces with a *total combined length* as large as possible. Cost is no object. Give an algorithm to compute the largest total combined length you could achieve. Analyze its asymptotic efficiency in terms of n, the number of marked segments in the initial piece of wood.

11.B. (40 PTS.) Suppose you get paid t dollars for each usable piece you produce, for some pre-specified t. Your net profits are the difference between the amount you get paid and the total cost of the cuts you make. Give an algorithm, as efficient as possible, to compute the best possible achievable net profit. Analyze its asymptotic efficiency in terms of n, the number of marked segments in the initial piece of wood.

11.C. (20 PTS.) Modify your algorithm for the previous part so that it outputs not just the lowest cost, but also the sequence of cuts necessary to achieve that cost.

*Rubric:*Standard dynamic programming rubric For problems worth 10 poins:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you are *trying* to do.) **Automatic zero if the English description is missing.**

  + 1 point for stating how to call your function to get the final answer.
  + 1 point for base case(s). $-1/2$ for one *minor* bug, like a typo or an off-by-one error.
  + 3 points for recursive case(s). -1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**

- 4 points for details of the dynamic programming algorithm
  + 1 point for describing the memoization data structure
  + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  + 1 point for 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 says otherwise.

- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative psuedocode 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 still need to describe the underlying recursive function in English.)

- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but did not work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).