CS 170
Fall 2017

Efficient Algorithms and Intractable Problems
Prasad Raghavendra and Sanjam Garg

Homework 6

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.
- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.
- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Wednesday, October 18, at 4:59pm

This homework emphasizes dynamic programming problems. In your solutions, note the following:

- When you give the main idea for a dynamic programming solution, you need to explicitly write out a recurrence relation and explain its interpretation.
- When you give the pseudocode for a dynamic programming solution, it is not sufficient to simply state "solve using memoization" or the like; your pseudocode should explain how results are stored.

0. Who did you work with?

List all your collaborators on this homework. If you have no collaborators, please list "none".

Solution: N/A

1. (★★★ level) Longest Palindrome Subsequence

A subsequence is *palindromic* if it is the same whether read left to right or right to left. For example, "bob" and "racecar" are palindromes, but "cat" is not. Devise an algorithm that takes a sequence $x[1..n]$ and returns the length of the longest palindromic subsequence. Its running time should be $O(n^2)$. (Recall that a subsequence need not be consecutive, e.g., "aba" is the longest palindromic subsequence of "anbma".)

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

Solution:

Main Idea:

Let $x = x_1 \dots x_n$ be the string, and let $P[i,j]$ be the length of the longest palindrome subsequence of $x_i \dots x_j$. Then we have the following recurrence:

$$P[i,j] = \begin{cases} P[i+1,j-1]+2 & \text{if } x_i == x_j \\ \max\{P[i+1,j], P[i,j-1]\} & \text{otherwise} \end{cases}$$

With base cases:

$$P[i,i] = 1 \qquad \forall i = 1 \dots n$$
$$P[i,i+1] = (x_i == x_{i+1}) \qquad \forall i = 1 \dots n-1$$

Once we have computed $P[i,j]$ for $i = 1 \dots n, j = i \dots n$, we can then output $P[1,n]$ – this is the length of the longest palindromic subsequence. One thing to be careful with is to make sure to compute $P[i,j]$ in the right order so that the entries $P[i+1,j-1], P[i+1,j], P[i,j-1]$ are computed before $P[i,j]$. They are several ways to do this; below we order based on decreasing order of i and increasing order of j respectively.

Pseudocode:

---

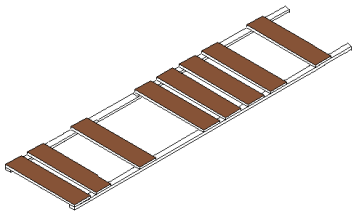
procedure LONGESTPALINDROMICSUBSEQUENCE($x[1 \dots n]$)
  $P \leftarrow$ matrix of size n by n.
  $P[i,i] \leftarrow 1$ for $i = 1 \dots n$.
  $P[i,i+1] \leftarrow (x_i == x_{i+1})$ for $i = 1 \dots n-1$.
  for $i = n-2 \dots 1$ **do**
    for $j = i+2 \dots n$ **do**
      if $x_i == x_j$ **then**
        $P[i,j] \leftarrow P[i+1,j-1]+2$.
      else
        $P[i,j] \leftarrow \max\{P[i+1,j], P[i,j-1]\}$
  return $P[1,n]$.

Runtime:

There are $\frac{1}{2}n(n+1) = O(n^2)$ entries to be filled, each takes constant time. The overall runtime is $O(n^2)$.

2. (★★★ level) Bridge Hop

You notice a bridge constructed of a single row of planks. Originally there had been n planks; unfortunately, some of them are now missing, and you're no longer sure if you can make it to the other side. For convenience, you define an array $V[1..n]$ so that $V[i]$ = TRUE iff the ith plank is present. You're at one side of the bridge, standing still; in other words, your *hop length* is 0 planks. Your bridge-hopping skills are as follows: with each hop, you can increase or decrease your hop length by 1, or keep it constant.



For example, the image above has planks at indices [1, 2, 4, 7, 8, 9, 10, 12], and you could get to the other side with the following hops: [0, 1, 2, 4, 7, 10, 12, 14].

You start at location 0, just before the first plank. Arriving at any location greater than n means you've successfully crossed. Due to your winged shoes, there is no maximum hop length. But you can only hop forward (hop length cannot be negative).

Devise an efficient algorithm to determine whether or not you can make it to the other side.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

Solution:

Main idea:

As in the previous problems, we have to decide what information to include in the subproblem definitions to make the recurrence easy to compute. We decide to define subproblems as $P(i,h)$, which is TRUE if we can get to location i with our last hop of length h and FALSE otherwise. Then, the recurrence is

$$P(i,h) = V[i] \wedge [P(i-h,h) \vee P(i-h,h-1) \vee P(i-h,h+1)]$$

because we can only land at location i with hop length h if there's a plank there, and our hop length on the previous step was $h-1$, h, or $h+1$.

Note that because our hop length can only increase by 1 at each step, the maximum hop length is n.

Note: It's also possible to do this problem in the other direction, with the recurrence answering the question "can we get to the other side from this location at this speed?"

Pseudocode:

procedure canHop($V[1..n]$):
1. Set $V[n+1..2n]$ to TRUE.
2. Set $P(i,h)$ to FALSE in the range $(-n..-1, 0..n+1)$, $(0, 1..n+1)$, $(1..n, 0)$, and $(0..n, n+1)$.
3. Set $P(0,0)$ to TRUE.
4. For $i := 1, \ldots, 2n$:
5. For $h := 1, \ldots, n$:
6. Set $P(i,h) := V[i] \wedge [P(i-h,h) \vee P(i-h,h-1) \vee P(i-h,h+1)]$
7. If $i > n$ and $P(i,h)$ return TRUE
8. return FALSE

The bounds of our iteration are pretty loose, in the sense that we're checking some values of $P(\cdot)$ we know will be FALSE, like $P(1,2)$, or in general $P(i,h)$ where $h > i$, for example. It's possible to add a bit of additional logic and speed up the algorithm by a constant factor, but the asymptotic runtime will not change.

Proof of correctness:

Base cases:

First, we always start at location 0, just before the bridge starts, at hop length 0, so we initialize this to TRUE.

Because we chose to write a simple recurrence, we have to ensure that impossible states are set to FALSE, so we don't lookup a missing value. Thus, we enforce min and max values on the hop length: we never need to stop after we get started (the only thing to do at that point would be to hop length back up to 1), so we can set hop lengths of 0 to FALSE after the starting point. Also, we need some upper bound so that the recurrence does not error when checking $P(i-h, h+1)$ at the max value of s, so we'll set hop lengths of $n+1$ to FALSE (we can never even reach hop length n, so $n+1$ can be safely set to FALSE). We also have to initialize $P(i,h)$ to FALSE at locations all the way back to $-n$, because our naive recurrence will consult them. Finally, we'll be looking at locations after the end of the bridge in our iteration, so we set $V[\cdot]$ to TRUE in all such plausible locations. This is fine as there's solid ground everywhere after the bridge ends.

Recurrence:

Having done all this legwork ensuring the recurrence never errors out, now we can confirm our recurrence is correct. By the definition of the problem, you can only speed up or slow down by at most one unit with each step, so that if you want to reach location i with hop length h you must have gotten to $i-h$ with a hop length equal to or one more/less than h. And there must be a plank at i. We set $P(i,h)$ to TRUE iff both of these conditions hold.

Now, let's consider when we return TRUE. We want to see if it's possible to cross the bridge, which means we must be able to reach some location larger than n at any hop length. It's impossible to speed up to faster than hop length n, so it's impossible for your first step after the bridge ends to fall after location $2n$. Thus, we can safely stop looking there. Thus, we simply return TRUE if we can successfully reach any location $i > n$.

Note that just checking location $n+1$ would not have been enough in all cases.

Running time:

The running time is $O(n^2)$, because we iterate through this number of cases in the nested for-loop in constant time for each case, and there are also $O(n^2)$ base cases.

3. (★★★ level) A Sisyphean Task

Suppose that you have n boulders, each with a positive integer weight w_i. You'd like to determine if there is any set of boulders that together weight exactly k pounds. You may want to review the solution to the Knapsack Problem for inspiration.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

(a) Design an algorithm to do this.

Solution:

Main idea: This is very much like the knapsack problem, except we want our items to sum to *exactly* k instead of being less than or equal to k.

Let $S(i,k')$ be true if and only if there is some subset of $W[1..i]$ that sums to k'. Then, we have the following recurrence relation:

$$S(i,k') = S(i-1,k') \vee S(i-1,k'-w_i)$$

Intuitively, this comes from the observation that a subset of $W[1..i]$ summing to k' either includes a_i or it doesn't. If it includes a_i, then we are left looking for a subset of $W[1..i-1]$ summing to $k'-w_i$. If it doesn't include a_i, then this means we need a subset of $W[1..i-1]$ that sums to k'. Thus, $S(i,k')$ is true if and only if at least one of $S(i-1,k')$ and $S(i-1,k'-w_i)$ is true. Our base cases are:

- $S(0,0) = true$ because the empty set sums to 0.
- $S(0,k') = false$ for $k' > 0$.
- $S(i,k') = false$ for $k' < 0$.

The final answer we are looking for is $S(n,k)$. This gives us nk subproblems, each of which takes cosntant time to solve.

Pseudocode:

```
1'.  Set S[0,0] = true and S[0,k'] = false for 0 < k' ≤ k.
2'.  For i := 1,...,n:
3'.      For k' := 0,...,k:
4'.          If S[i-1,k']:
5'.              S[i,k'] = true.
6'.          Else if k' ≥ w_i and S[i-1,k'-w_i]:        // Base case: false if k'-w_i < 0.
7'.              S[i,k'] = true.
8'.          Else:
9'.              S[i,k'] = false.
10'. Return S[n,k].
```

Proof of correctness: The recurrence relation as described in the main idea is correct. To show this, we observe that if $S(i,k')$ is true, meaning there is some subset I of $W[1..i]$ that sums to k', then I either contains w_i or it doesn't. If $w_i \in I$, then removing w_i from I yields a I', which is a subset of $W[1..i-1]$ and must sum to $k'-w_i$. In this case, $S(i-1,k'-w_i)$ must be true. If $w_i \notin I$, then I is a subset of $W[1..i-1]$ that sums to k', so $S(i-1,k')$ must be true. To show the other direction, we observe that if $S(i-1,k'-w_i)$ is true, then $S(i,k')$ must be true because if I' is a subset of $W[1..i-1]$ that sums to $k'-w_i$, then $I = I' \cup \{w_i\}$ is a subset of $W[1..i]$ that sums to k'. If $S(i-1,k')$ is true, then $S(i,k')$ must be true because if I' is a subset of $W[1..i-1]$ that sums to k', then I' is also a subset of $W[1..i]$ that sums to k'.

We observe that the algorithm correctly computes this recurrence relation, using the fact that each w_i must be strictly positive, so each iteration of the loop only relies on previously computed values. By definition $S(n,k)$ gives us the answer we want.

Running time analysis: This runs in $O(nk)$. There are nk iterations of the inner for loop, each of which takes a constant time.

(b) Is your algorithm polynomial in the *size* of the input? Remember that size is in terms of how many bits we need.

Solution:

No. This algorithm runs in time proportional to k, but to represent k in the input, we only need $\log_2 k$ bits. This may seem like a technicality, but it's actually very important – it means that it wouldn't take very long for somebody to write down an input that would make this algorithm run for a very long time. For instance, writing down the number 2^{100} takes just 100 bits, but would make our running time proportional to 2^{100}, which is more than the age of the universe in seconds (intractably large). This algorithm is what is known as *pseudopolynomial*, meaning it is polynomial in the input *values*, not the input *size*.

4. (★★★★ level) Non-Prefix Code

As we have learned in lecture, the Huffman code satisfies the *Prefix Property*, which states that the bit string representing each symbol is not a prefix of the bit string representing any other symbol. One nice property of such codes is that, given a bit string, there is at most one way to decode it back to a sequence of symbols. However, this is not true anymore once we are working with codes that do not satisfy the Prefix Property. For example, consider the code that maps A to 1, B to 01 and C to 101. A bit string 101 can be interpreted in two ways: as C or as AB. Your task is to, given a bit string s, determine how many ways one can interpret s. The mapping from symbols to bit strings of the code will be given to you as a dictionary d (e.g., in the example, $d = \{A : 1, B : 01, C : 101\}$); you may assume that you can access each symbol in the dictionary in constant time. Your algorithm should run in time at most $O(nm\ell)$ where n is the length of the input bit string s, m is the number of symbols, and ℓ is an upper bound on the length of the bit strings representing symbols.

Please turn in a four-part algorithm solution for this problem.

Solution: Main Idea: We define our subproblems as follows: let $A[i]$ be the number of ways of interpreting the string $s[:i]$. We can then compute $A[i]$ using the values of $A[j], j < i$ via the following recurrence relation:

$$A[i] = \sum_{\substack{\text{symbol } a \text{ in } d \\ s[i-\text{length}(d[a])+1:i]=d[a]}} A[i - \text{length}(d[a])].$$

Note here that we set $A[0] = 1$. Our algorithm simply computes the above formula this in a trivial manner.

Proof of Correctness: We can show this via a simple induction argument.

Base Case. When $i = 0$, there is only one way to interpret $s[:0]$ (the empty string). Hence, $A[0] = 1$.

Inductive Step. Suppose that $A[0], \ldots, A[i-1]$ contains the right value. We will show that the above recurrence relation gives the right value for $A[i]$. To do this, we partition interpretations of $s[:i]$ as a sequence of symbols $a_1 \ldots a_k$ based on the ending symbol a_k. For $a_k = a$, if the suffix of $s[:i]$ coincides with $d[a]$, every interpretation $a_1 \ldots a_k$ has a one-to-one correspondence with an interpretation $a_1 \ldots a_{k-1}$ of $s[:i - \text{length}(d[a])]$. From our inductive hypothesis, there are exactly $A[i - \text{length}(d[a])]$ of the latter. On the other hand, if the suffix of $s[:i]$ differs from $d[a]$, then there is no interpretation of $s[:i]$ ending with symbol a. Summing this up over all symbols a's implies that our recurrence relation yields the right value for $A[i]$.

Finally, note that our program below implements this recurrence in a straightforward way, so the output of our program is indeed $A[n]$, the number of ways to interpret s.

Pseudocode:

```
procedure TRANSLATE(s):
    Create an array A of length n+1 and initialize all entries with zeros.
    Let A[0] = 1
    for i := 1 to n do
        for each symbol a in d do
            if i ≥ length(d[a]) and d[a] = s[i − length(d[a]) + 1 : i] then
                A[i] += A[i − length(d[a])]
    return A[n]
```

Runtime Analysis: There are n iterations of the outer for loop and m iterations of the inner for loop. Inside each of these loops, checking that the two strings are equal take $O(\text{length}(d[a])) \leq O(\ell)$ time. Hence, the total running time is $O(nm\ell)$.

Note that it is possible to speed up the algorithm running time to $O((n+m)\ell)$ using a trie instead of reconstructing the string every time, but this is not required to receive full credit for the problem.

5. (★★★★ level) Lexicographically smallest subsequence

You are given a string S of length n, and an input k. Give the main idea and runtime (including recurrence relation) for a dynamic programming algorithm to find the smallest (by lexicographical ordering) subsequence of length exactly k. Note that a subsequence must retain characters in the same ordering as in S, but need not be contiguous. For example: in the word "rocket", the smallest subsequence of length 3 is "cet".

Please turn in a four-part algorithm solution for this problem. You may assume that $n > k$, and that concatenating strings together takes constant time. But comparing two strings of length k takes $\Theta(k)$ time.

Extra Practice Question: Can you find a solution that only takes $O(nk)$ time? If you have finished all the other questions, we encourage you to think about this question.

Solution 1:

Main Idea

Let $D[i,t]$ be the smallest subsequence of length t over $S[1\ldots i]$. Then for all $1 \le i \le n$ and $1 \le t \le \min\{i,k\}$,

$$D[i,t] = \begin{cases} S[1\ldots i], & \text{if } i = t, \\ \min\{D[i-1,t], D[i-1,t-1]+S[i]\}, & \text{otherwise,} \end{cases}$$

where the $+$ means string concatenation.

Base cases: Let ε be the empty string, then $D[i,0] = \varepsilon$ for $0 \le i \le k$.

Interpretation: at each step of the recurrence, in order to find the smallest subsequence of length t, you can either use the same smallest subsequence as before, or include the new character $S[i]$ together with a previous smallest subsequence of length $t-1$.

Pseudocode:

```
procedure LexSubsequence(S, k):
    Let n be the length of S.
    Create an array D of dimension (n+1) × (k+1) and initialize all entries with the empty string.
    for i := 1 to n do
        for t := 1 to min{i, k} do
            if i = t then
                D[i,t] = S[1...i]
            else
                D[i,t] = min{D[i-1,t], D[i-1,t-1] + S[i]}
    return D[n,k]
```

Proof of correctness: We prove the correctness of the recurrence by induction on i.

Base cases $i = 1$*:* when $t = 0$, we are only looking for an empty string, which we initialized as ε.

Inductive step $i > 1$*:* suppose that $D[i-1,t]$ is the smallest subsequence of length t over $S[1\ldots i-1]$ for all $t \le \min\{i-1,k\}$, we will show that $D[i,t]$ will be the smallest subsequence of length t over $S[1\ldots i]$ for all $1 \le t \le \min\{i,k\}$.

First of all, if $t = i$, then there is only 1 subsequence of length i over $S[1\ldots i]$, which is just $S[1\ldots i]$ itself. For $t < i$, there are two cases:

- if the smallest subsequence does not include the new character $S[i]$, then it means the smallest subsequence is entirely supported on $S[1\ldots(i-1)]$, thus we have $D[i,t] = D[i-1,t]$;

- if the smallest subsequence includes the new character $S[i]$, then it means the first $t-1$ characters are from $S[1\dots(i-1)]$. To achieve lexicographically smallest, these $t-1$ characters must be the lexicographically smallest subsequence from $S[1\dots(i-1)]$, thus we have $D[i,t] = D[i-1,t-1]+S[i]$;

Runtime: each step of the algorithm takes time $O(k)$ and there are nk total sub-problems to solve, therefore the total runtime will be $O(nk^2)$.

Solution 2: This is an improvement to the previous one, based on two new observations.

Main Idea: Let $D[i,t]$ be the smallest subsequence of length t over $S[1\dots i]$ as in the first solution. We will show that it suffices to carry out the above recurrence by only keeping track of $D[i,k]$.

Let T be a string of length $t+1$. Consider the following sub-routine $\mathsf{DeleteOne}(T)$ of removing one letter from T: Let j be the smallest index such that $1 \le j \le t$ and $T[j] > T[j+1]$. If there is no such index j, then $\mathsf{DeleteOne}(T) = T[1\dots t]$. Otherwise $\mathsf{DeleteOne}(T) = T[1\dots(j-1)]+T[(j+1)\dots(t+1)]$.

Observation 1 *If T is a string of length $t+1$, then the shortest subsequence of length t is* $\mathsf{DeleteOne}(T)$.

Observation 2 *For all* $1 \le i \le n, 1 \le t \le \min\{i,k\}$, $D[i,t-1] = \mathsf{DeleteOne}(D[i,t])$.

Note that this observation would follow from Observation 1 if $D[i,t-1]$ is a subsequence of $D[i,t]$, which we will show later.

Recall that $D[i,k] = \min\{D[i-1,k], D[i-1,k-1]+S[i]\}$. The above observation says that even if we don't know $D[i-1,k-1]$, we can compute $D[i-1,k-1]$ by deleting one letter from $D[i-1,k]$. Therefore we could carry out the recurrence as before without keeping track of $D[i,t]$ for $t < k$.

Let $C[i] = D[i,k]$, which is the smallest subsequence of length k over $S[1\dots i]$. Then for $i > k$,

$$C[i] = \min\{C[i-1], \mathsf{DeleteOne}(C[i-1])+S[i]\},$$

where the $+$ means string concatenation.

Base case: $C[k] = S[1\dots k]$.

Interpretation: at each step of the recurrence, you can either use the same smallest subsequence as before, or include this new character $S[i]$ to replace one of the previous ones. The index j is picking out which letter to remove from the previously calculated smallest subsequence of length k over $S[1\dots(i-1)]$.

As a side note: it is possible to compute the recurrence in time $O(1)$ by further observing that $C[i] = \mathsf{DeleteOne}(C[i-1]+S[i])$, and $\mathsf{DeleteOne}(\cdot)$ can be computed in $O(1)$ time: notice that as i increases, the position of the character being removed is strictly increasing, so one can keep track of the last position being removed to ensure that each character only needs an amortized of $O(1)$ comparisons. We omit the details here because this is a bit ad hoc, and there are other ways to arrive at linear time solutions below.

Proof of correctness: We start by proving Observation 1. To get the shortest length k subsequence out of a length $k+1$ string, we need and only need to remove one letter. Let the optimal subsequence of length k be R, and let j be the first index on which $R[j] \neq T[j]$ and $1 \le j \le k$. It means that the deleted letter is to the left of $T[j]$, or the deleted letter is $T[j]$ itself. Without loss of generality, we can assume it is the letter $T[j]$ itself, because if it were before $T[j]$, then they must be the same letter. Therefore $R[j] = T[j+1]$. And we must have $R[j] < T[j]$ otherwise we could replace $R[j]$ with $T[j]$ and get a better subsequence. Thus, j is such that $T[j] > T[j+1]$. And it must be the smallest such j, otherwise we could get a better subsequence.

We next show Observation 2: $D[i,t-1]$ is always a subsequence of $D[i,t]$ for all $1 \le i \le n, 1 \le t \le \min\{i,k\}$ by induction on i.

Base case $i = 1$: $D[1,0] = \varepsilon$ is a subsequence of $D[1,1]$.

Inductive step for $i \ge 2$: suppose that $D[i-1,t-1]$ is a subsequence of $D[i-1,t]$ for all $1 \le t \le \min\{i-1,k\}$, we will show that $D[i,t-1]$ is a subsequence of $D[i,t]$ for all $1 \le t \le \min\{i,k\}$. Recall that $D[i,t] = \min\{D[i-1,t], D[i-1,t-1]+S[i]\}$, and $D[i,t-1] = \min\{D[i-1,t-1], D[i-1,t-2]+S[i]\}$. By our induction hypothesis, the only possibility for $D[i,t-1]$ to not be a subsequence of $D[i,t]$ is if $D[i,t] = D[i-1,t]$ but $D[i,t-1] = D[i-1,t-2]+S[i]$. We argue that this cannot happen. Suppose that $D[i,t] = D[i-1,t]$, by Observation 1, the string $D[i-1,t]+S[i]$ must be non-decreasing (in other words, the first letter is no greater than the second letter, the second letter is no greater than the third, etc.). Therefore its string $D[i-1,t-1]+S[i]$ is also non-decreasing. So $D[i,t-1] = D[i-1,t-1]$ by Observation 1. This concludes the proof.

As a remark, Observation 2 is not an obvious consequence of the lexicographical order, because even if $a \le b$, it's possible that $\mathsf{DeleteOne}(a) > \mathsf{DeleteOne}(b)$, thus $\mathsf{DeleteOne}(\cdot)$ does not preserve lexicographical order in general.

Runtime: each step of the algorithm takes time $O(k)$ and there are $n-k$ total sub-problems to solve, therefore the total runtime will be $O(nk)$.

Solution 3: So far we have defined subproblems from the left of the string. We will see an example of defining subproblems from the right of the string, which leads to a drastically simpler solution.

Main Idea Consider $C[i,t]$ as the smallest subsequence from $S[i \dots n]$ of length t. Let $j^* = \arg\min_{i \le j \le n-t+1} S[j]$, where we break ties by choosing the smallest j^*, then

$$C[i,t] = S[j^*] + C[j^*+1, t-1].$$

Base case: $C[n,0] = \varepsilon, C[n,1] = S[n]$.

This is a tail recursion, thus we can unroll the recursion into the following loop-based algorithm: find the lexicographically smallest character among $S[1 \dots (n-k+1)]$, break ties by choosing the first of such, denoted by $S[j_1]$, then find the smallest among $S[(j_1+1) \dots (n-k+2)]$, break ties by choosing the fist of such, denoted by $S[j_2]$, and so on.

As a side note, it is possible to speed up the above procedure by maintaining a priority queue of the candidate characters; initially, make a list of tuple $\{(S[1],1),(S[2],2),\cdots\}$ for the characters in $S[1 \dots (n-k+1)]$; we prioritize first over the smallest character, and then the smallest position; in step i, we dequeue from the priority queue to get the smallest character (may need to dequeue multiple times to find one with a position that is after the current position), and then en-queue $S[n-k+i+1]$ if $i < k$; this will have a running time of $O(n \log n)$ because every character is enqueued once and dequeued at most once. We omit the details here as it is a bit ad hoc.

Proof of correctness: We prove the recurrence is correct by induction on i.

Base case $i = n$: $C[n,0] = \varepsilon, C[n,1] = S[n]$ is correct by definition.

Inductive step $i < n$: suppose that $C[j,t]$ is the smallest subsequence from $S[j \dots n]$ of length t for all $j > i$ and all $t : n-j+1 \le t \le k$, we show that $C[i,t]$ is computed correctly. Recall that $j^* = \arg\min_{i \le j \le n-t+1} S[j]$, where we break ties by choosing the smallest j^*. Suppose the contrary, there is a smaller string $T < C[i,t]$ of length t, then either $T[1] < S[j^*]$, or the substring $T[2 \dots t] < C[j^*+1, t-1]$. The first case is not possible, because we defined j^* as the smallest. For the second case, since j^* is the smallest index such that

$T[1] = S[j^*]$, so $T[2 \dots t]$ must be a substring of $S[(j^*+1) \dots n]$, therefore by the induction hypothesis we should have $T[2 \dots t] \geq C[j^*+1, t-1]$, thus the second case is also not possible. This concludes the proof.

Runtime: To find one character, it takes $O(n)$ time, we have to find k characters, so the total running time will be $O(nk)$.

Solution 4: This can be seen as a repeated application of Observation 2 from Solution 2. Alternatively this can also be seen as an attempt to speed up Solution 3: instead of finding only one character in every step, we try to find as many characters as possible.

As an intuition, in every step, we keep track of the smallest subsequence (of length possibly shorter than k), such that it can later be extended to a length k string.

Main Idea Given a string A, we write $|A|$ for the length of A. For notational convenience, we will first extend the definition of lexicographical order to strings of unequal length as follows. For strings A, B, let $t = \min\{|A|, |B|\}$, then we say $A < B$ if $A[1 \dots t] < B[1 \dots t]$. It is worth noting however, with this definition, $A \leq B$ and $B \leq A$ only means that either A is a prefix of B, or B is a prefix of A. Moreover, for a string A of at least 2 letters, $\mathsf{DeleteOne}(A) < A$ if and only if there exists j such that $A[j] > A[j+1]$.

Let $D[i,t]$ be the smallest subsequence of length t over $S[1 \dots i]$ as in the first solution. We consider the following subproblem

$$C[i] = \min_{\substack{t \leq k \\ t \geq \max\{1, k-(n-i)\}}} D[i,t],$$

where we break ties by choosing the *longest* subsequence.

We will show that $C[i]$ can be obtained from $C[i-1]$ as follows: Given $S[i]$, let j be the largest index such that $C[i-1][j] \leq S[i]$, and $k-(n-i)-1 \leq j \leq |C[i-1]|$ (in other words, from right to left find the first such j). If no such index j exists, then $C[i] = C[i-1][1 \dots (k-(n-i)-1)] + S[i]$; otherwise, let $T = C[i-1][j] + S[i]$, if $|T| > k$, then $C[i] = C[i-1]$; otherwise $C[i] = T$.

As a side note, $C[i]$ can also be characterized as follows: Let $T = C[i-1] + S[i]$. First check that if $|T| > k$, then $T = \mathsf{DeleteOne}(T)$. Next keep applying $T = \mathsf{DeleteOne}(T)$ until T becomes too short, or the letters in T are already in non-decreasing order (in other words, $T[1] \leq T[2] \leq T[3] \leq \cdots$).

Interpretation: $C[i]$ is the lexicographically smallest string of length at least $\max\{1, k-(n-i)\}$ and at most k. And in each step, try to find the longest among the smallest subsequence of $C[i-1] + S[i]$ of length at least $\max\{1, k-(n-i)\}$ and at most k.

Proof of correctness: Define

$$\mathsf{Opt}[i] = \min_{\substack{t \leq k \\ t \geq \max\{1, k-(n-i)\}}} D[i,t],$$

where we break ties by choosing the longest subsequence.

We first show that $\mathsf{Opt}[i]$ restricted to the first $i-1$ characters, must be a prefix of $\mathsf{Opt}[i-1]$ for $i > 1$.

Let r be the length of $\mathsf{Opt}[i]$, and R be $\mathsf{Opt}[i]$ restricted to the first $i-1$ characters (in other words, if $\mathsf{Opt}[i]$ does not contain $S[i]$ then we let $R = \mathsf{Opt}[i]$, otherwise we let $R = \mathsf{Opt}[i][1 \dots (r-1)]$). We claim that $\mathsf{Opt}[i-1] \leq R$. Since $\mathsf{Opt}[i]$ is of length at least $k-(n-i)$, then R is of length at least $k-(n-i)-1$. And R is only supported on the first $i-1$ characters, so if $R < \mathsf{Opt}[i-1]$, we get a contradiction to the optimality of $\mathsf{Opt}[i-1]$. Similarly we must have $R \leq \mathsf{Opt}[i-1]$, otherwise we could substitute $\mathsf{Opt}[i-1]$ into $\mathsf{Opt}[i]$ and get a better $\mathsf{Opt}[i]$, a contradiction. This implies that either R is a prefix of $\mathsf{Opt}[i-1]$ or $\mathsf{Opt}[i-1]$ is a prefix of R. Since we break ties by choosing the longest subsequence when defining $\mathsf{Opt}[i-1]$, we conclude that R is a prefix of $\mathsf{Opt}[i-1]$.

We then argue that either the length of $\mathsf{Opt}[i]$ is $k-(n-i)$, or $\mathsf{Opt}[i]$ is in non-decreasing order (that is, $\mathsf{Opt}[i][1] \le \mathsf{Opt}[i][2] \le \mathsf{Opt}[i][3] \le \cdots$). In other words, if the length of $\mathsf{Opt}[i]$ is greater than $k-(n-i)$, then $\mathsf{Opt}[i]$ is in non-decreasing order. If the length of $\mathsf{Opt}[i]$ is strictly greater than $k-(n-i)$, then $\mathsf{DeleteOne}(\mathsf{Opt}[i])$ is of length at least $k-(n-i)$. Suppose for the sake of contradiction that $\mathsf{Opt}[i]$ is not in non-decreasing order, that is, there exists j such that $\mathsf{Opt}[i][j] > \mathsf{Opt}[i][j+1]$. Then we have $\mathsf{DeleteOne}(\mathsf{Opt}[i]) < \mathsf{Opt}[i]$, contradicting the optimality of $\mathsf{Opt}[i]$.

Now we are ready to prove $C[i] = \mathsf{Opt}[i]$ by induction.

Base case $i = 1$: $C[1] = D[1,1] = \mathsf{Opt}[1]$.

Inductive step: we want to show $C[i] = \mathsf{Opt}[i]$ from $C[i-1] = \mathsf{Opt}[i-1]$. Let t be the length of $\mathsf{Opt}[i-1]$.

- If $t = k-(n-i)-1$, then because $\mathsf{Opt}[i]$ has to be of length $k-(n-i)$, and $\mathsf{Opt}[i]$ restricted to the first $i-1$ characters is a prefix of $\mathsf{Opt}[i-1]$, so $\mathsf{Opt}[i] = \mathsf{Opt}[i-1] + S[i] = C[i]$.
- If $t > k-(n-i)-1$, then $\mathsf{Opt}[i-1]$ must be in non-decreasing order. Recall that $\mathsf{Opt}[i]$ restricted to the first $i-1$ characters is a prefix of $\mathsf{Opt}[i-1]$, $\mathsf{Opt}[i]$ must be a subsequence of $\mathsf{Opt}[i-1]+S[i]$. Let $T = \mathsf{Opt}[i-1]+S[i]$. Notice that, as long as T is not in non-decreasing order, we will have $\mathsf{DeleteOne}(T) < T$. Thus to find the best subsequence in $\mathsf{Opt}[i-1]+S[i]$, recall from Observation 1 and Observation 2 that we can repeatedly apply $\mathsf{DeleteOne}(\cdot)$ to find the next best subsequence. Notice that since $\mathsf{Opt}[i-1]$ is already in non-decreasing order, so all the out-of-order pairs in $\mathsf{DeleteOne}(\cdot)$ only involve $S[i]$. In effects, repeated application of $\mathsf{DeleteOne}(\cdot)$ to $\mathsf{Opt}[i-1]+S[i]$ works just as in our algorithm: we scan from right to left on $\mathsf{Opt}[i-1]$, and find the first j such that $\mathsf{Opt}[i-1][j] \le S[i]$, and then take $\mathsf{Opt}[i-1][j]+S[i]$. We stop if and only if either the subsequence has become too short, or it has become non-decreasing order (since we want to find the longest among the ties).
  - Now if $|T| > k$, it means that the above procedure stopped because $\mathsf{Opt}[i-1]+S[i]$ is in non-decreasing order to begin with. So the best length k subsequence will be $\mathsf{Opt}[i-1]$, which is exactly the same in our algorithm, and we have $\mathsf{Opt}[i] = \mathsf{Opt}[i-1] = C[i-1] = C[i]$.
  - Otherwise if $|T| \le k$, T is the longest among all the smallest subsequence, so we have $\mathsf{Opt}[i] = T = C[i]$.

Runtime: Consider for each character, it can be added to the string at most once, and can be removed at most once, and the number of comparisons can be amortized to 2 (once for the first time of being added to $C[i]$, once just before being removed from $C[i]$), there are n total sub-problems to solve, therefore the total runtime will be $O(n)$. As a remark, without an amortized analysis one may only conclude $O(nk)$ time for this solution.

6. (??? **level**) **(Optional) Redemption for Homework 5**

Submit your *redemption file* for Homework 5 on Gradescope. If you looked at the solutions and took notes on what you learned or what you got wrong, include them in the redemption file.

Solution: N/A