CS 170 Efficient Algorithms and Intractable Problems
Spring 2016 Alessandro Chiesa and Umesh Vazirani **HW 7**

Due March 7, 11:59pm

1. (20 pts.) The Quest Begins

You're on a quest to visit the magical city of Pimentel, and the only way to get there is through a narrow path across a mountain. A dragon lives on the mountaintop, and eats all those who dare intrude upon his mountain – unless they pay a sufficiently high bribe. You'll need to earn as many gold coins as you can in the n days before you have to face the dragon.

There are two nearby villages where you can earn some money doing odd jobs. The villages publish lists $A[1..n]$ and $B[1..n]$, where $N[i]$ is the nonnegative integer number of gold coins available to be earned by working at village N on day i. (You're confident that you're a hard enough worker that you can always earn all $N[i]$ coins available.)

The travel time between the two villages is a day, so that if you're working at A on day d and decide to switch to B, you can start working there on day $d+2$. You can start at either village on day 1.

Design an efficient algorithm to find the highest number of gold coins you could earn in the next n days.
Solution 1:

Main Idea:

The key insight is that our subproblems should keep track of state information about your location (village A or B) at the start of day i. Define $V_A[i]$ to be the highest number of gold coins you could earn for days $i..n$, if start day i in village A. Similarly define $V_B[i]$.

With these definitions it is easy to write recurrences:

$$V_A[i] = \max\{A[i] + V_A[i+1], V_B[i+1]\}$$
$$V_B[i] = \max\{B[i] + V_B[i+1], V_A[i+1]\}$$

with base cases $V_A[n+1] = V_B[n+1] = 0$.

Filling in each entry takes $O(1)$ time for a total of $O(n)$.

This is in contrast with simply defining the subproblems as "the value of the optimum schedule for days $i..n$" We still have $O(n)$ subproblems, but in our solution the subproblems also store a crucial bit of information (where you are), rather than just the partial value of the schedule.

This could also have been done backwards, too, in which case $V_A[i]$ would depend on $V_A[i-1]$ and $V_B[i-1]$, and the iteration in the `for` loop would go from 1 to n.

Pseudocode:

1. Set $V_A[n+1] := 0$ and $V_B[n+1] := 0$.
2. For $i := n, n-1, \ldots, 1$:
3. Set $V_A[i] := \max(A[i] + V_A[i+1], V_B[i+1])$ and $V_B[i] := \max(B[i] + V_B[i+1], V_A[i+1])$.
4. Return $\max(V_A[1], V_B[1])$.

Correctness: The expression for $V_A[i]$ follows from a case analysis: in the ith day, our two options are to either stay at A or to travel to B. In the former case, we will get $A[i]$ coins on the ith day, and then the maximum amount of coins we can get in the remaining days is given by $V_A[i+1]$ (you'll be at village A at the beginning of the $i+1$st day, since you stayed there on the ith day). In the latter case, we get no coins on the ith day while traveling to village B, and then the maximum amount of coins we can get in the remaining hours is given by $V_B[i+1]$. So in the former case, we can get $A[i]+V_A[i+1]$ seconds of computation done on hours $i..n$, and in the latter case, we can get $V_B[i+1]$ seconds of computation done on hours $i..n$. We should choose whichever one is larger, which is exactly what our algorithm does.

The correctness of the expression for $V_B[i]$ follows from a very similar argument.

Finally, any schedule for days $1..n$ must start at either village A or village B on the first day, so its value must be given by $V_A[1]$ or $V_B[1]$, respectively. We can choose whichever is larger, so the value of the optimal overall schedule is given by $\max(V_A[1], V_B[1])$.

Running time: There are $2n = O(n)$ subproblems, and each problem takes $O(1)$ time to solve, so the total running time is $O(n)$. Put another way, there are n iterations of the loop in lines 2–3 of the pseudocode, and each iteration takes $O(1)$ time, so the total running time is $O(n)$.

Comments: It is also possible to reconstruct the optimum schedule itself in $O(n)$. One way is to use previous-pointers, like in the shortest paths algorithms we saw earlier. That involves augmenting the pseudocode to get something like this:

1'. Set $V_A[n+1] := 0$ and $V_B[n+1] := 0$.
2'. For $i := n, n-1, \ldots, 1$:
3'. If $a_i + V_A[i+1] > V_B[i+1]$ then set $V_A[i] := a_i + V_A[i+1]$ and $\text{next}_A(i) :=$ A, otherwise set $V_A[i] := V_B[i+1]$ and $\text{next}_A(i) :=$"move".
4'. If $b_i + V_B[i+1] > V_A[i+1]$ then set $V_B[i] := b_i + V_B[i+1]$ and $\text{next}_B(i) :=$ B, otherwise set $V_B[i] := V_A[i+1]$ and $\text{next}_B(i) :=$"move".
5'. If $V_A[1] > V_B[1]$ then set $t :=$ A, otherwise set $t :=$ B.
6'. For $i := 1, 2, \ldots, n$:
7'. Print t.
8'. Set $t := \text{next}_t(i)$.

Solution 2: Another approach is to reduce this to a longest-paths problem. We define a directed graph $G = (V, E)$ with $2n$ vertices, namely $V = \{(A,i) : i = 1, \ldots, n\} \cup \{(B,i) : i = 1, \ldots, n\}$. The graph has $4(n-1)$ edges, namely, one of each of the following, for each $i \in \{1, 2, \ldots, n-1\}$:

- An edge from (A,i) to $(A,i+1)$, with length $A[i]$, corresponding to working on village A on day i.
- An edge from (A,i) to $(B,i+1)$, with length 0, corresponding to traveling from village A to village B on day i.
- And, similarly, an edge from (B,i) to $(B,i+1)$, with length $B[i]$, and an edge from (B,i) to $(A,i+1)$, with length 0.

The longest path in the graph then corresponds to the optimal schedule. To turn this into a single-source longest-paths problem, we then add a single source vertex s, and add edges of length 0 from s to $(A,1)$ and from s to $(B,1)$. Then the longest path in G starting from s corresponds to the optimal schedule.

Note that G is a dag. The textbook contains a (dynamic-programming) algorithm for computing the longest path in a dag in $O(|V| + |E|)$ time, so we can use that. The running time is $O(n)$, since $|V| = O(n)$ and $|E| = O(n)$.

You didn't need to prove the longest-paths algorithm correct, since it comes from the textbook.

2. (25 pts.) Giant's Riddle

Having successfully bribed the dragon, you come upon a bridge over a huge chasm; you can just see the gates of Pimentel on the other side. A huge giant armed with a club blocks the entrance of the bridge. He asks you the following riddle, and promises to let you pass if you can solve it.

(a) **(20 pts.)** Consider the "longest common increasing subsequence" problem: given two sequences $a = a_1, a_2, \ldots, a_n$, and $b = b_1, b_2, \ldots, b_m$, find the longest possible subsequence that is both (1) a subsequence of both a and b and (2) is strictly increasing.

For example, the longest common increasing subsequence of $(1,2,3,5,6,2,4)$ and $(2,1,3,0,7,4,5,6)$ is $(1,3,5,6)$. We can verify that there is no longer subsequence of both sequences that is also strictly increasing.

Devise an efficient algorithm to return the length of the longest common increasing subsequence (algorithms can optionally find and return the subsequence as well).

Input: Two sequences of integers a, b.

Output: A single integer, denoting the length of the longest common increasing subsequence.

The runtime of your algorithm should be at most $O(nm(n+m))$.

(*The giant taunts you, telling you it's possible to solve this even in $O(nm)$ time.*)

(b) **(5 pts.)** The giant wants to double-check you truly understand your answer. Calculate the longest common increasing subsequence (not just its length) on the following two lists; show your work by writing the list/table of subproblems you solved to find the answer.

(*YOUR CAL ID*) and (3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8).

Solution:

Solutions to part (b) will vary; the solution to part (a) follows.

Solution 1. Main idea:

The key insight in out subproblem definition is that we'll insist that the last element of b in our prefix (b_j) be in our partial LCIS.

Let $L(i,j)$ denote the longest common increasing subsequence of the sequences $a_1,\ldots,a_i$, and $b_1,\ldots,b_j$ that includes b_j (note we don't necessarily have to include a_i). If no such subsequence exists that ends at b_j, we take $L(i,j) = -\infty$. Our final answer will be $\max(0, \max_{1\le j\le m} L(n,j))$.

Then, we can write the recurrence $L(i,j) = \max(L(i-1,j), 1+\max_{k<j \wedge b_k<b_j}(0, L(i-1,k))$ if $a_i = b_j$)

with base cases $L(1,x) = \begin{cases} -\infty \text{ if } a_1 \neq b_x \\ 1 \end{cases}$ for all valid x.

Proof of correctness:

We know that the common subsequence has to end at some point. So, taking the max of $L(n,j)$ will correctly account for all the cases, so the final answer computation is correct. We now need to show that the $L(i,j)$ values are computed correctly. First, we can verify that our base cases are correct, which is trivial, since a consists of only a single character, so we make sure that it matches with the last character.

For our recursive call, there are two cases, we either use a_i or we don't. If we do use a_i, we will get the correct answer from $L(i-1,j)$. However, if we do use a_i, we need to find the previous element of our common increasing subsequence. We iterate through all possible k that we could have ended before (or zero, if this is the first element), thus this case is correctly taken care of. Thus, we've shown that this algorithm will compute the correct length of the longest common increasing subsequence.

Pseudocode: procedure getLongest(a,b):

```
1.  For x := 1,...,m:
2.      If a[1] == b[x]:
3.          L[1][x] = 1
4.      Else:
5.          L[1][x] = -∞
6.
7.  For i := 2,...,n:
8.      For j := 1,...,m:
9.          L[i][j] = L[i-1][j]
10.         If a[i] == b[j]:
11.             for k := 0,...,j-1:
12.                 If b[k] < b[j]:
13.                     L[i][j] = max(L[i][j], L[i-1][k]+1)
14. ret = 0
15. For j := 1,...,m:
16.     ret = max(ret, L[n][j])
17. return ret
```

Running time

There are three nested for loops and the inside computation takes constant time, so this runs in $O(nm^2)$, which is in $O(nm(n+m))$, so this is sufficient. Another way we can analyze runtime is to look at the number of subproblems, and multiply by the time it takes to evaluate each subproblem. There are $O(nm)$ subproblems, and each subproblem takes $O(m)$ time to evaluate, thus, we get $O(nm^2)$ again. (note that $O(n^2m)$ is also acceptable).

How to speed this up to $O(nm)$ for the curious:

We'll keep the same problem definition. There are a few things to note in the pseudocode in the unoptimized version. First, when we iterate through the inner loop $a[i]$ stays constant. Another thing we can note is that we don't need to compute the max all the way from the beginning; rather we can keep track of a running max. Thus, we can only do constant time per subproblem. It's important to note that this was made much easier if we analyzed our dynamic program bottom up. If we had taken a memoized approach, this would have been much harder to find.

Thus, we can get the pseudocode below.

procedure getLongest(a,b):

1. For $x := 1, \ldots, m$:
2. If $a[1] == b[x]$:
3. $L[1][x] = 1$
4. Else:
5. $L[1][x] = -\infty$
6.
7. For $i := 2, \ldots, n$:
8. $m = 0$
9. For $j := 1, \ldots, m$:
10. $L[i][j] = L[i-1][j]$
11. If $b[j] < a[i]$ and $L[i-1][j] > m$:
12. $m = L[i-1][j]$
13. If $b[j] == a[i]$ and $m+1 > L[i][j]$:
14. $L[i][j] = m+1$
15. $ret = 0$
16. For $j := 1, \ldots, m$:
17. $ret = \max(ret, L[n][j])$
18. return ret

Here, the variable m is the running max, so we don't need to recompute it every time.

Solution 2:

Main idea:

In this problem, as in the last one, the challenge is how to define subproblems in such a way that they store enough information that the recurrence is simple and efficient to compute. One approach is to keep track not only of prefixes of a and b but also the value of the last (maximum) element of the common subsequence. Then, it'll be easy for us to "extend" LCISs as we iterate through the prefixes.

We'll define $G(i,j,k)$ as the longest common increasing subsequence of the sequences $a_1,\ldots,a_i$ and $b_1,\ldots,b_j$, given that the largest element of the common subsequence is at most k. Because the largest number in the sequences could be indefinitely large, *nmk* subproblems would be *exponential* in the size of the input. (Make sure you understand why; see the textbook discussion of KNAPSACK). Therefore, we'll remap our integers so they are between 1 and $n+m$ inclusive. This is possible because there are at most $n+m$ distinct integers in the sequences. So, we have the recurrence

$$G(i,j,k) = \max(G(i-1,j,k), G(i,j-1,k), G(i,j,k-1), G(i-1,j-1,k-1) + equals(a_i, b_j, k))$$

where $equals(x,y,z)$ returns 1 if and only if $x = y = z$.

Pseudocode

procedure getLongest(a,b):

1. Sort $a \cup b$, and set the ith distinct element in the sorted list to i.
2. Set $G(i,j,k) = 0$ for the following ranges: $(0, 1..m, 1..n+m)$, $(1..n, 0, 1..n+m)$, and $(1..n, 1..m, 0)$.
3. For $k := 1,\ldots,n+m$:
4. For $i := 1,\ldots,n$:
5. For $j := 1,\ldots,m$:
6. $G(i,j,k) = \max(G(i-1,j,k), G(i,j-1,k), G(i,j,k-1), G(i-1,j-1,k-1) + equals(a_i, b_j, k))$
7. return $G(n,m,k)$.

Correctness

The first line remaps the numbers in the range 1 to $n+m$ without changing any of their relative ordering.

Correctness flows from the validity of our recurrence relation: either a_i is not in our LCIS, b_j is not in our LCIS, we have a LCIS where the largest element is at most $k-1$, or $a_i = b_j = k$ is part of our LCIS. Our base cases ($k = 0$, which is less than our minimum element value of 1 in the remapped integers, or i or j are empty) are set to 0, and by the time we compute $G(i,j,k)$ we've already computed all the entries it depends on.

Running time

The running time is $O(nmk) = O(nm(n+m))$. $k = O(n+m)$ because of our remapping explained above, so there are $nm(n+m)$ states in our dp, and each state takes constant time to evaluate.

Common wrong solutions: One common wrong answer is do the LCS, then find the LIS of that sequence. However, this doesn't work as on this example here: a = [1,2,3,5,4,3,2,1], and b = [5,4,3,2,1,2,3]. Here, the LCS is [5,4,3,2,1], but the LCIS is [1,2,3], which would not be considered. It turns out any approach that only looks at LIS then the LCS is also incorrect, even if you look at the LIS ending at each sequence. Here's another counter example a = [7,9,10,11,4,5,6,8] and b = [7,12,13,14,1,2,3,8]. Here, the LCIS is [7,8], but [7,8] is not the LIS ending or beginning at any element of either sequence.

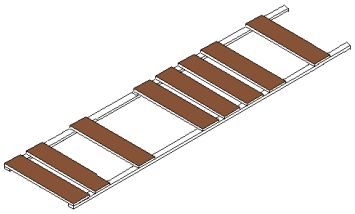
Another common wrong approach is if we take the subproblem to be $L(i,j)$ to be the longest common increasing subsequence of $a_1,\ldots,a_i$ and $b_1,\ldots,b_j$, and letting the recurrence be something like $L(i,j) =$

$\max(L(p,q)+1)$, where $p<i, a_p<a_i$ and $q<j, b_q<b_j$. This is too slow, since there are $O(nm)$ subproblems, each of which takes $O(nm)$ to evaluate, leading to a runtime of $O((nm)^2)$.

Also, greedy strategies where the dp only looks at the rightmost element that is less than a_i, or the largest element less than a_i that is before a_i don't quite work. The reason they fail is the same reason why the greedy version of longest increasing subsequence doesn't work.

3. (15 pts.) Bridge Hop

Having outwitted the giant's defenses, it's now time to cross the bridge. You notice the bridge is 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].

Clarifications: 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 you can make it to the other side.

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

4. (20 pts.) A Sisyphean Task?

You've crossed the bridge and arrived at the gates of Pimentel. They're locked, but you notice a giant-sized scale in front of them, with the positive integer k written on it. You deduce that the gates only unlock when the giant, who weighs k pounds, steps on the scale.

You look around for anything that might help, and you notice n boulders conveniently arrayed nearby, each boulder i helpfully labeled with its positive integer weight w_i. You jot down the list of weights $W[1..n]$. You'd like to determine if there is any set of boulders that together weigh exactly k pounds, so that you can place them on the scale to impersonate the giant, unlock the gates, and enter the city.

(a) **(18 pts.)** 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) **(2 pts.)** Is your algorithm polynomial in the *size* of the input?

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*.

5. (20 pts.) City Tour

You're finally inside, and you can't wait to tour Pimentel and all of its wondrous attractions! Its layout can be represented as an undirected graph with positive lengths on all of the edges, where the length of an edge represents the time it takes to travel along that edge. You have an ordered sequence of m attractions which you would like to visit. Each attraction is a vertex in the graph. There may be vertices in the graph that are not attractions.

Due to time constraints, you can only visit k of these attractions, but you still want to visit them in the same order as originally specified. What is the minimum travel time you incur?

Design an efficient algorithm for the following problem:

Input: a graph $G = (V, E)$, with lengths $\ell : E \to \mathbb{R}_{>0}$, attractions $a_1, \ldots, a_m \in V$, and $k \in \mathbb{N}$

Output: The length of the shortest path that visits k of the attractions, in the specified order

In other words, we want to find a subsequence of $a_1, \ldots, a_m$ that contains k attractions, and find a path that visits each of those k attractions in that order—while minimizing the total length of that path. More precisely, we want to find the length of the shortest path of the form $a_{i_1} \rightsquigarrow a_{i_2} \rightsquigarrow \cdots \rightsquigarrow a_{i_k}$, where the indices $i_1, \ldots, i_k$ can be freely chosen as long as they satisfy $1 \leq i_1 < i_2 < \cdots < i_k \leq m$. You don't need to output the path or the subsequence—just the length is enough.

Clarification: Suppose the shortest path from a_1 to a_2 goes through some other attraction, say a_5. That's OK. If the subsequence starts $a_1, a_2, \ldots$, it is permissible for the path from a_1 to a_2 to go through a_5 along the way to a_2, if this is the shortest way to get from a_1 to a_2 (this doesn't violate the ordering requirement, and a_5 doesn't count towards the k attractions).

Main idea: First, we compute all-pairs shortest paths for the m attractions in the graph. This can be done by m invocations of Dijkstra, starting once from each possible attraction (which takes $O(m(|V| + |E|) \log |V|)$ time), or the Floyd-Warshall algorithm (which takes $O(|V^3|)$ time).

This gives us $\texttt{dist}(u, v)$, the length of the shortest path between attractions u and v, for each pair of attractions u, v. Let $f(i, j)$ denote the length of the shortest path that contains exactly i attractions from the list of attractions $a_1, \ldots, a_j$, and that ends at the attraction a_j. We get a recursive equation for f, as follows:

$$f(i, j) = \min_{1 \leq u < j} f(i-1, u) + \texttt{dist}(a_u, a_j),$$

if $i \leq j$ and $i > 1$. We also have the base cases $f(1, j) = 0$ if $j \geq 1$ and $f(i, j) = \infty$ if $i > j$.

The problem statement wants us to output the length of the shortest path that visits k of the m candidate attractions. We achieve this by returning $\min(f(k,k), f(k,k+1), \ldots, f(k,m))$.

Pseudocode:

1. Set $\texttt{dist}$:= Floyd-Warshall(G).
2. For $j := 1, \ldots, m$:
3. Set $f[1][j] := 0$.
4. For $i := 2, \ldots, k$:
5. For $j := 1, 2, \ldots, i-1$:
6. Set $f[i][j] = \infty$.
7. For $j := i, i+1, \ldots, m$:
8. Set $f[i][j] := \min\{f[i-1][u] + \texttt{dist}[a_u][a_j] : u = 1, 2, \ldots, j-1\}$.
9. Return $\min(f[k][k], f[k][k+1], \ldots, f[k][m])$.

Proof of correctness: Consider the general problem of computing the shortest path containing exactly i attractions from the list of attractions $a_1, \ldots, a_j$ and ending at a_j. Let's assume we have correctly computed optimal solutions to all subproblems involving either fewer than i required attractions or fewer than j candidate attractions.

Consider the shortest path visiting i attractions that ends at a_j. Let a_u be the attraction that it visits immediately before a_j (if $i > 1$, such an attraction must exist). Since the path visits attractions in order and doesn't visit any attraction twice, we know $u < j$. Then our path must consist of a $i-1$-attraction path ending at a_u, followed by a path from a_u to a_j. The length of the first part must be $f(i-1,u)$ [that's the shortest possible length], and the length of the second is $\texttt{dist}(a_u, a_j)$. Thus, the length of this path is correctly computed by the recursive formula above.

We justify our base cases as follows:

- If $i > j$: We cannot choose more than j attractions from j candidate attractions. We penalize this subproblem by assigning it a value of ∞.
- If $i \leq j$ and $i = 1$: A path of length 0 (start at a_1 and don't go anywhere) suffices to visit one attraction.

Finally, the shortest i-attraction path must end at some attraction; we can find it by taking the minimum of all of the $f(k, \cdot)$ values.

Running time: To compute f, we need to solve mk subproblems (the loops in lines 4,5,7), each of which requires $O(m)$ amount of work (line 8). Therefore, computing f requires a total running time of $O(m^2k)$. To compute `dist` in line 1, if you chose to do m invocations of Dijkstra to compute the all-pairs shortest paths, then the total running time is $O(m^2k + m(|V| + |E|)\log|V|)$. Alternatively, if you chose Floyd-Warshall, the total running time is $O(m^2k + |V|^3)$. Either is acceptable.

6. (* pts.) Programming Assignment: Finding Counterexamples to Greedy Algorithms

*Grading: This problem will be worth 40 points total, with 20 points coming from Homework 6 and 20 points coming from homework 7. So if you get 30 points on this problem then you will get 15 points for it on Homework 6 and 15 points on Homework 7. Thus, the other problems on this homework will contribute to 80% of your score for it, with the other 20% coming from the programming assignment.

Please see HW 6 for the problem text and submission instructions.

Due to the nature of the assignment, we are not allowing resubmissions on this question.