CS 170 Algorithms Fall 2014 David Wagner Sol 8

1. (20 pts.) Subsequence

Design an efficient algorithm to check whether the string $A[1..n]$ is a subsequence of $B[1..m]$, for $n \leq m$. The running time of your solution should be at most $O(nm)$.

Note: the elements in a subsequence do not need to be consecutive (subsequence $\neq$ substring).

Solution 1: We can solve this problem with dynamic programming. Define the predicate $f(i, j)$ to be true if the string $A[1 \ldots i]$ is a subsequence of $B[1 \ldots j]$. We obtain a recursive formula for $f(i, j)$:

$$f(i,j) = \begin{cases} \text{True} & \text{if } i = 0 \wedge j \geq 0 \\ \text{False} & \text{if } j < i \\ f(i-1, j-1) & \text{if } A[i] = B[j] \\ f(i, j-1) & \text{otherwise} \end{cases}$$

Pseudocode:

```
1. For j := 0, 1, ..., m:
2.     Set f[0][j] := True.
3. For i := 1, 2, ..., n:
4.     For j := 1, 2, ..., m:
5.         If j < i:
6.             Set f[i][j] = False.
7.         else if A[i] = B[j]:
8.             Set f[i][j] := f[i-1][j-1].
7.         else:
8.             Set f[i][j] := f[i][j-1].
9. Return f(n, m).
```

Correctness: The recursive expression for $f(i, j)$ follows from a case analysis:

- if $A[i] = B[j]$: Since $B[j]$ matches with $A[i]$, $A[1 \ldots i]$ is a subsequence of $B[1 \ldots j]$ iff $A[1 \ldots i-1]$ is a subsequence of $B[1 \ldots j-1]$.
- if $A[i] \neq B[j]$: Since $B[j]$ doesn't match with $A[i]$, $A[1 \ldots i]$ is a subsequence of $B[1 \ldots j]$ iff $A[1 \ldots i]$ is a subsequence of $B[1 \ldots j-1]$.

We justify the base cases as follows:

- $i = 0 \wedge j \geq 0$: There are no remaining characters within A to match with B. $A[1 \ldots 0]$ is trivially a subsequence of $B[1 \ldots j]$. Therefore, $f(i, j)$ is True.
- $j < i$: Since $A[1 \ldots i]$ is a larger string than $B[1 \ldots j]$, $A[1 \ldots i]$ cannot be a subsequence of $B[1 \ldots j]$. Therefore, $f(i, j)$ is False.

Running time: There are mn subproblems, and each problem takes $O(1)$ time (lines 5–8). Therefore, the total running time is $O(nm)$.

Solution 2: A greedy linear scan through the two arrays solves this problem in linear time.

Pseudocode:

1. Set $i := 0$ and $j := 0$.
2. @loop invariant: $A[1..i]$ is a subsequence of $B[1..j]$.
3. While $j < m$ and $i < n$:
4. If $B[j+1] = A[i+1]$:
5. Set $i := i+1$.
6. Set $j := j+1$.
7. Return true if $i = n$, false otherwise.

This algorithm has a running time of $O(m+n)$. The while loop maintains the invariant that $A[1..i]$ is a subsequence of $B[1..j]$. [1]

2. (20 pts.) Another scheduling problem

You're running a massive physical simulation, which can only be run on a supercomputer. You've got access to two (identical) supercomputers, but unfortunately you have a fairly low priority on these machines, so you can only get time slots on them when they'd otherwise be idle. You've been given information about how much idle computing power is available on each supercomputer for each of the next n one-hour time slots: you can get a_i seconds of computation done on supercomputer A in the ith hour if your job is running on A at that point, or b_i seconds of computation if it running on supercomputer B at that point.

During each hour your job can be scheduled on only one of the two supercomputers. You can move your job from one supercomputer to another at any point, but it takes an hour to transfer the accumulated data between supercomputers before your job can begin running on the new supercomputer, so a one-hour time slot will be wasted where you make no progress.

So you need to come up with a schedule, where for each one-hour time slot your job either runs on supercomputer A, runs on supercomputer B, or "moves" (it switches which supercomputer it will use in the next time slot). If your job is running on supercomputer A for the $i-1$th hour, then for the ith hour your only two options are to continue running on A or to "move." The value of a schedule is the total number of seconds of computation that you get done during the n hours.

You want to find a schedule of maximal value. Design an efficient algorithm to find the value of the optimal schedule, given $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$.

Solution 1: We will construct two arrays, $A[1..n+1]$ and $B[1..n+1]$. $A[i]$ will contain the value of the optimum schedule for hours $i..n$, if the job is ready to run on supercomputer A at the beginning of hour i. $B[i]$ will hold the value of the optimum schedule for hours $i..n$, if the job is ready to run on supercomputer B at the beginning of hour i. Each array entry holds the solution to a subproblem, so there are $2n$ subproblems in all. Then we have the following recursive expression for these arrays:

$$A[i] = \max(a_i + A[i+1], B[i+1])$$
$$B[i] = \max(b_i + B[i+1], A[i+1])$$

along with the base cases $A[n+1] = B[n+1] = 0$.

[1] Note that $A[1 \ldots 0]$ can be defined to be the empty sequence, and $A[i \ldots i]$ is a sequence of length 1 containing $A[i]$.

Pseudocode:

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

Correctness: The expression for $A[i]$ follows from a case analysis: in the ith hour, our two options are to either continue running on A or to "move". In the former case, we will get a_i seconds of computation done in the ith hour, and then the maximum amount of computation we can get done in the remaining hours is given by $A[i+1]$ (the job is ready to run on supercomputer A at the beginning of the $i+1$st hour, since it was running on supercomputer A during the ith hour). In the latter case, we get no computation done in the ith hour while transferring the data to supercomputer B, and then the maximum amount of computation we can get done in the remaining hours is given by $B[i+1]$ (the job is ready to run on supercomputer B at the beginning of the $i+1$st hour, after the transfer). So in the former case, we can get $a_i + A[i+1]$ seconds of computation done on hours $i..n$, and in the latter case, we can get $B[i+1]$ 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 expresiion for $B[i]$ follows from a very similar argument.

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

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

Another approach is to stick with the original pseudocode, and use the $A[]$ and $B[]$ arrays to work out the best schedule. We can tell which machine the best schedule starts at by looking to see whether $A[1] > B[1]$ or not. Then we apply the following observation iteratively: if we know that the best schedule leaves the job ready to run on supercomputer A at the beginning of hour i, then we should stay on supercomputer A if $A[i] = a_i + A[i+1]$ (in which case the best schedule leaves the job ready to run on A at the beginning of hour $i+1$), or we should "move" if $A[i] = B[i+1]$ (in which case the job will be ready to run on B at the beginning of hour $i+1$); and similarly if the best schedule leaves the job ready to run on supercomputer B at the beginning of hour i. This is basically the same idea as the pseudocode above; it just avoids the need for some extra storage.

Solution 2: In Solution 1, each subproblem corresponds to a suffix of the a / b lists. You also could have solved it by making each subproblem corresponds to a prefix of the a / b lists. For instance, let $A^*[i]$ denote the value of the best possible schedule for hours $1..i$ that leaves the job ready to run on supercomputer A at the beginning of hour $i+1$, and similarly for $B^*[i]$; then solve them in the order $i = 1, 2, \ldots, n$ (the reverse of the order in Solution 1).

Solution 3: 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 running the job on supercomputer A in hour i.
- An edge from (A, i) to $(B, i+1)$, with length 0, corresponding to "moving" from supercomputer A to supercomputer B in hour i.
- An edge from (B, i) to $(B, i+1)$, with length b_i, corresponding to running the job on supercomputer B in hour i.
- An edge from (B, i) to $(A, i+1)$, with length 0, corresponding to "moving" from supercomputer B to supercomputer A in hour i.

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.

Comments: You might have noticed that this looks pretty similar to Problem 5 on HW5. The similarity is that, if we imagine looking at the optimal schedule at any point partway through its execution, there is some "state" that represents everything we need to know about the past. In fact that "state" is just one bit, indicating which supercomputer the data currently resides on (i.e., where the job is ready to run in the next hour). So we can build a dag where each vertex represents the current time and the one-bit state, and take it from there.

In general, if you can characterize a problem as a sequence of transitions through some statespace (of limited size), then you might try applying a similar approach.

3. (20 pts.) Park tours

You're trying to arrange a tour of a national park. The park 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 a 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 $\text{dist}(u, v)$, the length of the shortest path between attractions u and v, for each pair of attractions u, v. We introduce a function $f(i, j)$ which denotes the shortest path containing exactly i attractions from the list of attractions $a_1, \ldots, a_j$. We get a recursive equation for f, as follows:

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

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 $f(k, m)$.

Pseudocode:

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

Proof of correctness: Consider the general problem of computing the shortest path containing exactly i attractions from the list of attractions $a_1, \ldots, a_j$. 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. Given an attraction a_j, the optimal solution either visits it or avoids it. For each case, we prove that our recursive formulation returns the optimal solution:

- If the optimal solution visits attraction a_j: Any i-attraction path to a_j must pass through a predecessor of a_j, and the path to the predecessor must visit $i-1$ attractions. Therefore, the shortest path to a_j passes through the predecessor a_u that minimizes the sum of $\text{dist}(a_u, a_j)$ and $f(i-1, a_u)$.
- If the optimal solution avoids attraction a_j: Since we must visit i attractions from $a_1, \ldots, a_j$ and we are avoiding a_j, we must choose the shortest path containing i attractions from attractions $a_1, \ldots, a_{j-1}$.

We justify our base cases as follows:

- If $i > j$: We cannot choose $> j$ 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.

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.

4. (20 pts.) Optimal binary search trees

Suppose we know the frequency with which keywords occur in programs of a certain language.

We want to organize them in a binary search tree, so that the keyword in the root is alphabetically bigger than all keywords in the left subtree and smaller than all keywords in the right subtree (and this holds for all nodes).

The figure has a nicely-balanced example on the left. In this case, when a keyword is being looked up (for compilation perhaps), the number of comparisons needed is at most three; for instance, in finding "while", only the three nodes "end", "then", and "while" get examined. But since we know the frequency with which keywords are accessed, we can use an even more fine-tuned cost function, the *average number of comparisons* to look up a word, For the search tree on the left, it is

$$cost = 1(0.04) + 2(0.40 + 0.10) + 3(0.05 + 0.08 + 0.10 + 0.23) = 2.42$$

By this measure, the best search tree is the one on the right, which has a cost of 2.18.

Give an efficient algorithm for the following task.

Input: n words (in sorted order); frequencies of these words: $p_1, p_2, \ldots, p_n$.

Output: The binary search tree of lowest cost (defined above as the expected number of comparisons in looking up a word).

Solution:

Let $S(i, j)$ be the cost of the cheapest tree formed by words i to j, for $1 \leq i \leq j \leq n$. Also, initialize $S(i, j)$ to 0 if $i > j$. Then we can compute $S(i, j)$ by looking at all choices of which word to use as the root of the tree; the root could be word k, for any k where $i \leq k \leq j$. If word k is at the root, the cost of the left subtree will be $S(i, k-1)$ and the cost of the right subtree will be $S(k+1, j)$. Moreover, all words will need to pay one comparison at the root node and the cost of the words to the left or right of the root can be determined recursively, so if we place word k at the root, the total cost of the tree will be

$$\sum_{t=i}^{j} p_t + S(i, k-1) + S(k+1, j).$$

Hence:

$$S(i, j) = \min_{i \leq k \leq j} \{\sum_{t=i}^{j} p_t + S(i, k-1) + S(k+1, j)\}$$

Finally, the cost of the optimal tree is given by $S(1, n)$.

To reconstruct the tree, it suffices to keep track of which root k minimized the expression in the recursion for each subproblem and backtrack from $S(1, n)$.

Pseudocode: We implement this algorithm using recursive calls with memoization. Observe that line 1 in procedure $S(i, j)$ implements memoization. We invoke **findbest** to compute the cost of the optimal binary search tree.

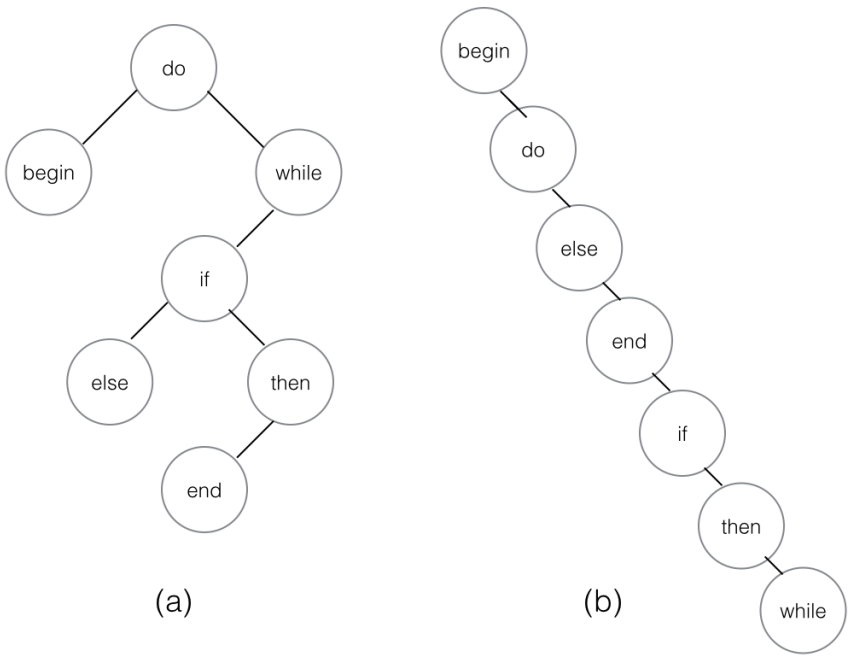

Figure 1: Binary search trees produced by the greedy strategy. (b) shows a suboptimal solution.

procedure findbest():
1. For $i := 1, \ldots, n$:
2. For $j := 1, \ldots, n$:
3. If $i > j$, set $T[i][j] := 0$, otherwise set $T[i][j] :=$ invalid.
4. Return $S(1,n)$.

procedure S(i,j):
1. If $T[i][j] =$ invalid:
2. Set $m := \infty$.
3. For $k := i, i+1, \ldots, j$:
4. Set $m := \min(m, \sum_{t=i}^{j} p_t + S(i,k-1) + S(k+1,j))$.
5. Set $T[i][j] := m$.
6. Return $T[i][j]$.

Running time: The running time for this algorithm is $O(n^3)$. There are $O(n^2)$ subproblems (one for each possible i, j pair), and each subproblem takes $O(n)$ to compute, since it requires taking a min over $O(n)$ values (the values of $\sum_{t=i}^{j} p_t$ can be memoized along with the values of $S(i, j)$).

Comment: It is important to note that greedy approaches lead to suboptimal solutions for this problem. One potential greedy approach is to choose the keyword with the highest probability and insert it into the binary search tree. We remove the chosen keyword from the input list, and repeat. For the example input

$$\{\text{begin: } 0.05, \text{ do: } 0.4, \text{ else: } 0.08, \text{ end: } 0.04, \text{ if: } 0.10, \text{ then: } 0.10, \text{ while: } 0.23\},$$

this greedy strategy produces the optimal solution of cost 2.18, as shown in Figure 1(a). However, consider the input

$$\{\text{begin: } 0.18, \text{ do: } 0.17, \text{ else: } 0.15, \text{ end: } 0.14, \text{ if: } 0.13, \text{ then: } 0.12, \text{ while: } 0.11\}.$$

The greedy strategy produces the solution shown in Figure 1(b). This solution is suboptimal: its cost is

$$cost = 1(0.18) + 2(0.0.17) + 3(0.15) + 4(0.14) + 5(0.13) + 6(0.12) + 7(0.11) = 3.67$$

On the other hand, the dynamic programming solution listed above gives us the optimal cost of 2.43 on this input.

Comment: The dynamic programming algorithm shown above has $O(n^3)$ running time, but with a slight tweak, it can be optimized to run in $O(n^2)$ time. In addition to keeping track of the cost $S(i,j)$ of the best tree for words $i..j$, also keep track of which word to use as the root for this tree: let $R(i,j)$ denote the index k of the word to use as the root for the best tree for words $i..j$. Keeping track of $R(i,j)$ is straightforward. Now, we get a better recursive formula for S:

$$S(i,j) = \min_{R(i,j-1) \le k \le R(i+1,j)} \{\sum_{t=i}^{j} p_t + S(i,k-1) + S(k+1,j)\}.$$

The only difference is that we let k range over $R(i,j-1) \ldots R(i+1,j)$ instead of $i \ldots j$, which reduces the number of possibilities for k that we must consider. Here's the intuition behind why this modified formula is correct: if we know the optimal root for words $i..j-1$, and then we add a word to the right (namely, word j), then the optimal root can only move to the right, but it cannot move left.

It turns out that, with this modification to the recursive formula, we get a $O(n^2)$ time algorithm. The proof is a bit tricky, but here is the intuition. Fix a length ℓ for the range of subwords (so i,j range over values such that $j-i=\ell$). Then the recursive formula for $S(0,\ell)$ lets k range over the interval $R(0,\ell-1) \ldots R(1,\ell)$; the formula for $S(1,\ell+1)$ lets k range over the interval $R(1,\ell) \ldots R(2,\ell+1)$; the formula for $S(2,\ell+2)$ lets k range over the interval $R(2,\ell+1) \ldots R(3,\ell+2)$, and so on. Here we notice that the right endpoint of each interval is the same as the left endpoint of the next interval. This means that for a fixed length ℓ, we only look at a total of $O(n)$ candidate values for the root in total (the total among computing $S(0,\ell), S(1,\ell+1), \ldots, S(n-\ell+1,n)$). Also, there are $O(n)$ different lengths. Therefore, the total running time is $O(n^2)$. I find this amazing and beautiful.

You didn't need to find a $O(n^2)$ time solution. The $O(n^3)$ time dynamic programming algorithm above was sufficient for full credit.

5. (20 pts.) Beat Inference

We have an audio track for some song, represented as an array $A[1..n]$, where $A[i]$ represents the loudness at the ith sample. We know the tempo of the song: the time between beats is approximately d samples.

Our goal is to infer the times at which drum beats occur. Of course, human musicians are not perfect at following the tempo; sometimes the number of samples between two beats will be a bit above d, sometimes a bit below d, but it'll generally be close. We do know that the samples when a drum beat occurs tend to be louder than average.

Based on this, we define a cost function

$$\text{cost}(t_1, \ldots, t_m) = \sum_{i=2}^{m} (t_i - t_{i-1} - d)^2 - \sum_{i=1}^{m} A[t_i].$$

Experiments suggest that if we find the indices $t_1, \ldots, t_m$ that minimize $\text{cost}(t_1, \ldots, t_m)$, then those represent the times at which the drum beats occurred.

Design an efficient algorithm for the following problem:

Input: $A[1..n]$, d, and m.

Output: $t_1, \ldots, t_m$ that minimize $\text{cost}(t_1, \ldots, t_m)$, subject to the requirement that $1 \leq t_1 \leq t_2 \leq \cdots \leq t_m \leq n$.

The running time of your algorithm should be at most $O(n^2 m)$.

Solution: Define $f(i, j)$ to be the lowest possible cost for any sequence of i beats $t_1, \ldots, t_i$ chosen out of $A[1..j]$, subject to the requirement that $t_i = j$. In other words, $f(i, j)$ is the lowest cost of any way to select i beats ending at $A[j]$, out of the first j samples of the song (out of $A[1..j]$).

It is helpful to observe that

$$\text{cost}(t_1, \ldots, t_{i-1}, t_i) = \text{cost}(t_1, \ldots, t_{i-1}) + (t_i - t_{i-1} - d)^2 - A[t_i].$$

This observation is useful for dynamic programming because it relates the cost of a i-length sequence of beats to the cost of a $(i-1)$-length subsequence of beats.

Using this observation, we obtain the following recursive formula for $f(i, j)$:

$$f(i, j) = \min_{i-1 \leq u \leq j-1} \{f(i-1, u) + (j - u - d)^2 - A[j]\}.$$

We also have base cases $f(1, j) = -A[j]$.

Then, the lowest attainable cost for the original problem (find m beat times from $A[1..n]$) is given by $\min(f(m, m), f(m, m+1), \ldots, f(m, n))$. While this tells us the minimum cost, we also want to know what specific sequence of beat times attain this cost. In order to reconstruct the sequence, we record prev pointers for each cell to remember which choice minimized the expression in the recursion. After computing the solution to all mn subproblems, we traverse the prev pointers to reconstruct the sequence.

Pseudocode:

```
1.  For j := 1, ..., n:
2.      Set f[1][j] := -A[j].
3.  For i := 2, ..., m:
4.      For j := i, i+1, ..., n:
5.          Set f[i][j] = ∞.
6.          For u := i-1, i, i+1, ..., j-1:
7.              Let x := f[i-1][u] + (j-u-d)^2 - A[j].
8.              If x < f[i][j]:
9.                  Set f[i][j] = x and prev[i][j] := u.
10. Set t_m := arg min_{m≤j≤n} f(m, j).
11. For i := m-1, m-2, ..., 1:
12.     Set t_i := prev[i+1][t_{i+1}].
13. Return t_1, ..., t_m.
```

Proof of correctness: Consider the general problem of choosing i beats from a list of j samples $A[1, \ldots, j]$, ending with $t_i = j$. Let's assume we have correctly computed optimal solutions to all subproblems involving either fewer than i beats or fewer than j candidate samples. Since the last beat must be at time j, consider the next-to-last beat: it must be at some time in the range $i-1, \ldots, j-1$ (e.g., it must be earlier than time j). The cost of beats $t_1, \ldots, t_i$ where $t_{i-1} = u$ and $t_i = j$ is

$$\text{cost}(t_1, \ldots, t_{i-1}, t_i) = \text{cost}(t_1, \ldots, t_{i-1}) + (t_i - t_{i-1} - d)^2 - A[t_i] = \text{cost}(t_1, \ldots, t_{i-1}) + (j - u - d)^2 - A[j].$$

The recursive formula for $f(i, j)$ follows immediately, as $f(i, j)$ is the lowest possible value for $\text{cost}(t_1, \ldots, t_{i-1}, t_i)$ such that $t_i = j$, and $f(i-1, u)$ is the lowest possible value for $\text{cost}(t_1, \ldots, t_{i-1})$ such that $t_{i-1} = u$.

Running time: We solve mn subproblems. Solving each subproblem takes $O(n)$ time (lines 5–9). Therefore, the total running time is $O(mn^2)$.

Comment: This approach is apparently effective in practice. See, e.g., `http://www.ee.columbia.edu/~dpwe/pubs/Ellis07-beattrack.pdf` for an academic publication on this algorithm.

Comment: We can greatly reduce the amount of space needed by storing only $f[i-1][\cdot]$ and $f[i][\cdot]$ (the previous and current row of f). This lets us compute the cost of the best beat times in $O(mn^2)$ time and $O(n)$ space.

What about recovering the beat times? At first glance, it looks like we'll need $O(mn)$ space to store all the prev pointers. However, it turns out it is possible to recover the beat times in $O(mn^2)$ time and $O(n)$ space, if we're very clever. The trick: along with each cost $f(m,j)$ (for a sequence of m beats ending at $t_m = j$), we'll remember the value of $t_{m/2}$ that attains this cost. This is easy to do by storing this value in a separate field during the iteration where $i = m/2$, and then copying this from $i-1$ to i each time we execute line 9. So, in $O(mn^2)$ time and $O(n)$ space, we've computed the cost of the best solution and the values of $t_{m/2}$ and t_m for the best solution. Now we can recurse (using divide-and-conquer): we find the best $m/2$ beat times $t_1,\ldots,t_{m/2}$ for $A[1..t_{m/2}]$, and the best $m/2$ beat times $t_{m/2+1},\ldots,t_m$ for $A[t_{m/2}+1,\ldots,n]$, recursively, using the same algorithm. Heuristically, we can expect $t_{m/2} \approx n/2$. Therefore, the running time has recurrence $T(m,n) \approx 2T(m/2,n/2)+O(mn^2)$, whose solution is $T(m,n) = O(mn^2)$. (This argument can be made more formal by noting that each level of the recursion takes less than half the time of the previous level, so the total running time is $O(mn^2) \times (1+\frac{1}{2}+\frac{1}{4}+\cdots) = O(mn^2)$.) The space needed is $O(n)$. However, you didn't need to do any of this: the basic solution, with $O(mn^2)$ time and $O(mn)$ space, is sufficient for full credit.

6. (5 pts.) Optional bonus problem: Image re-sizing

In this problem you will explore an application of dynamic programming to automatic re-sizing of images. We are given a (greyscale) image $I[1..m][1..n]$ with m rows of pixels and n columns; $I[i,j]$ contains the intensity (greyscale level) of the pixel in the ith row and jth column. We want to shrink this to an image with m rows and $n-1$ columns, i.e., one column narrower.

We will do this by deleting a *tear* from I. A *tear* is a sequence of pixels that follows a path from the top of the image to the bottom of the image. More precisely, a tear T is a sequence of pixels $T = ((1,x_1),(2,x_2),\ldots,(m,x_m))$, such that: (1) for each i, $1 \le x_i \le n$; and, (2) for each i, $|x_{i+1}-x_i| \le 1$. To delete a tear, we delete each pixel in the tear from the image I, so removing a single tear shrinks a $m \times n$-pixel image to a $m \times (n-1)$-pixel image.

We'd like to choose a tear whose removal will be least noticeable to the human eye. Intuitively, one reasonable idea is to choose pixels whose intensity is similar to that of their neighbors (avoiding removal of sharp edges or other kinds of intricate detail). With this idea in mind, we will consider the cost of deleting pixel $I[i,j]$ to be

$$\text{cost}(i,j) = \big|I[i,j-1]-I[i,j]\big| + \big|I[i,j]-I[i,j+1]\big|,$$

and the cost of a tear $T = ((1,x_1),\ldots,(m,x_m))$ to be

$$\text{cost}(T) = \text{cost}(1,x_1)+\cdots+\text{cost}(m,x_m).$$

Design an efficient algorithm to find a minimal-cost tear, given the image $I[1..m,1..n]$. Your algorithm should run in $O(nm)$ time.

Main idea: We will keep a matrix, where $M[i,j]$ represents the cost of the best tear that covers only rows 1 though i and that ends at the point (i,j). In other words, of all the partial tears of the form $(1,x_1),(2,x_2),\ldots,(i,x_i)$, we're looking at the cheapest one that satisfies $x_i = j$. Then we have the recursive expression

$$M[i,j] = \text{cost}(i,j) + \min(M[i-1,j-1],M[i-1,j],M[i-1,j+1]),$$

along with base cases $M[1,j] = \text{cost}(1,j)$.

Details and justification: This corresponds to *mn* subproblems, one for each pixel in the image. To find the best tear (and not just its cost), we will also build a second matrix $P[\cdot,\cdot]$, where $P[i,j]$ holds the predecessor (on row $i-1$) of the last element of the partial tear ending at (i,j). Let's see how we build up these two matrices.

What is the value of $M[i,j]$? Because it is a tear, the predecessor on row $i-1$ could have been at position $j-1$, j or $j+1$ only. Moreover, the tear up to the predecessor has to be optimal, otherwise the current tear would not be. Hence, we have the following update rules:

$$M[i,j] = \text{cost}(i,j) + \min(M[i-1,j-1], M[i-1,j], M[i-1,j+1])$$

and $P[i,j] = j-1$, j or $j+1$ according to which of the three terms achieves the minimum.

In order to reconstruct the minimum tear, we find the smallest of $M[m,1], M[m,2], \ldots, M[m,n]$, which tells us where in the last row the minimal-cost tear ends. Then we follow the predecessor trail P for the rows above to reconstruct a complete tear.

Running time: There are *mn* subproblems and each of them takes $O(1)$ time (because we only look at 3 neighbors), hence a total running time of $O(mn)$ (including $O(m+n)$ for reconstruction of the tear itself by following the entries in P).