Introduction to Algorithms **Solution Sketches for Problem Set 4**
CS 4820, Spring 2014

(1) *(10 points)* If P is a convex polygon in the plane, a *triangulation* of P is a way of dividing it up into triangles whose corners are vertices of P. More precisely, a triangulation of P consists of a set of line segments $\mathbf{L} = \{L_1, \ldots, L_r\}$ and a set of triangles $\mathbf{T} = \{T_1, \ldots, T_s\}$ satisfying the following properties.

1. For each segment $L_i \in \mathbf{L}$, the endpoints of L_i are vertices of P.
2. For each triangle $T_j \in \mathbf{T}$, the sides of T_j are segments in $\mathbf{L}$.
3. No two segments in $\mathbf{L}$ cross each other in the plane. In other words, any two segments $L_i, L_j \in \mathbf{L}$ are either disjoint or they have a unique point of intersection that is an endpoint of both segments.
4. Every side of the polygon P is a side of exactly one triangle $T_j \in \mathbf{T}$.
5. Every segment $L_i \in \mathbf{L}$ that is not a side of P is a side of exactly two triangles $T_j, T_k \in \mathbf{T}$.

For example, Figure 1 illustrates two different triangulations of a seven-sided polygon.

Suppose that P is a convex polygon in the plane and that you are given an input consisting of the following information:

- a list of the vertices $v_1, v_2, \ldots, v_n$ of P, in clockwise order;
- a positive integer cost $c(i, j)$ that denotes the cost of creating a line segment from v_i to v_j. These costs are defined whenever $1 \leq i < j \leq n$.

For any triangulation of P, the cost of the triangulation is defined to be the sum of the costs of the line segments included in the set $\mathbf{L}$. Design an algorithm to compute the minimum cost of a triangulation of P.

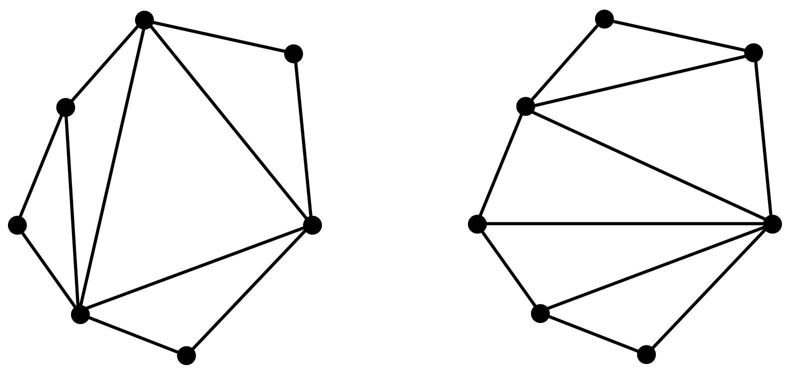

Figure 1: Two different triangulations of a seven-sided polygon.

Solution Sketch

Basically, the dynamic program works by finding the min-cost triangulation of every polygon formed from a consecutively-numbered subset of vertices $\{v_1, \dots, v_n\}$. So define $T[i,j]$ to be the minimum cost of a triangulation of the polygon formed by $\{v_i, \dots, v_j\}$. The value of $T[i,j]$ is well-defined as long as $1 \le i < i+2 \le j \le n$. In the min-cost triangulation of $\{v_i, \dots, v_j\}$, the side joining v_i to v_j must participate in a triangle with some other vertex v_k. If $k = i+1$ or $k = j-1$, these are special cases where the removal of this triangle leaves a single polygon that must be triangulated. If $i+1 < k < j-1$, then the removal of the triangle containing v_i, v_k, v_j leaves two polygons each of which must be triangulated. This justifies the recurrence

$$T[i,j] = \min \left\{ \begin{array}{c} c(i,i+1) + T[i+1,j] + c(i,j), \\ T[i,j-1] + c(j-1,j) + c(i,j), \\ \min_k \{T[i,k] + T[k,j] + c(i,j)\} \end{array} \right\} \tag{1}$$

where the internal minimization is over $k = i+2, ..., j-2$. If we fill in the table entries in order of increasing $|j-i|$, then when we evalute (1) for a given i, j, the formula only refers to table entries that have already been computed in earlier steps. This justifies the following algorithm.

```
1: for $i = 1, \dots, n-2$ do
2:     $T[i, i+2] = c(i, i+1) + c(i+1, i+2) + c(i, i+2)$
3: end for
4: for $d = 3, 4, \dots, n-1$ do
5:     for $i = 1, 2, \dots, n-d$ do
6:         Compute $T[i, i+d]$ using (1).
7:     end for
8: end for
```

The running time is $O(n^3)$ because we have to evaluate (1) $O(n^2)$ times, and it takes $O(n)$ work each time. The correctness of the algorithm is justified by the paragraphs preceding the pseudocode, formalized using induction over the dynamic programming table entries, in the order they were filled in.

General-Purpose Hints For Dynamic Programming

The way that I break down the thought process of designing dynamic programming algorithms is:

1. What's the last decision the algorithm will need to make?
2. What information does it need in order to make that decision? The answer to this question tells us how the dynamic programming table should be formatted.
3. Assuming it had that information, what rule (i.e. recurrence) would it use to calculate its last decision?

4. Is there an ordering of the entries of the dynamic programming table, that allows us to calculate each new entry using only the values of previously-calculated entries?

For people who don't like this "working backwards" thought process, the alternate line of attack is:

1. What dimensionality should the dynamic programming table have?
2. What is the meaning of the (i, j) entry?
3. What formula can we use to calculate it using previous entries.

The problem with this alternate line of attack is that there's no way to answer the first two questions, other than trial-and-error or inspired guesswork.

Proofs of correctness. The proof of correctness is usually one of the easiest parts of solving a dynamic program problem, in refreshing contrast to the case of greedy algorithms. Basically, the proof of correctness is always induction over the entries of the dynamic programming table, in the order that the algorithm fills them in. The following steps need to be addressed.

1. State the induction hypothesis: it should describe the intended meaning of each entry of the dynamic programming table, and it should assert that when the algorithm fills in an entry of the table, the value that it fills in matches the intended meaning.
2. Prove the base case: the table gets initialized correctly.
3. Prove the induction step. This involves justifying that the recursive formula for filling in an entry of the table is correct, assuming previously computed entries contain the correct value. There are two things that need to be done here.
   (a) When the recursive formula refers to another entry of the table, you must prove that that entry of the table has already been computed in an earlier step of the algorithm. Often this is a trivial consequence of the order in which the loops are written, but occasionally it requires an actual proof.
   (b) Justify that the recursive formula is correct. For example, if it is a maximization or minimization problem, argue that the formula involves maximizing or minimizing over a list of cases that is exhaustive.

Memoization. Some people prefer to write pseudocode containing a recursive procedure with memoization rather than explicit loops. That's fine, but they should make sure that the pseudocode is detailed enough that we can see that memoization is taking place. In particular, the following parts should be addressed in any solution that uses memoization.

1. The data structure that stores the results of previous function calls should be explicitly named, and the pseudocode should make it clear how this data structure is indexed. (For example, if it's an array then the pseudocode should indicate the dimensionality of the array and how the variables that index the array entries are related to the arguments of the recursive function call.)

2. The pseudocode should clearly indicate when and how the algorithm queries the data structure to check for previously-stored values, and it should clearly indicate when and how new entries are stored in this data structure.

3. The proof of correctness should address the issue of termination: how do you know that the algorithm won't get into an infinite loop of pushing function calls onto a recursive stack that grows without bound?

On acceptable running times for dynamic programs. Finally, people should not be afraid to design algorithms with slow (but polynomial) running times. Most dynamic programming algorithms are slow: n^2, n^3, sometimes much worse. I read a paper one time that was designing a dynamic programming algorithm with a running time of $O(n^{12})$. The size of the table was n^7, and computing a single entry of the table required n^5 time.

(2) *(10 points)* Solve Chapter 5, Exercise 7 in Kleinberg & Tardos.

Solution Sketch

We begin with the following observation: if a grid graph contains an interior node at which the value is less than the minimum value on the boundary, then there is an interior node which is a local minimum. The proof of the observation is almost trivial: the global minimum value cannot lie on the boundary, so it lies in the interior, and the node containing the global minimum value is obviously a local minimum.

So here's the divide and conquer algorithm. Subdivide the n-by-n grid into four sub-grids of size $n/2$ by $n/2$. Let S be the set of nodes on the boundaries of these four sub-grids. For every node in S, probe the node and all of its neighbors. If one of the nodes in S is a local minimum, output that node. Otherwise, let v be the node of S whose label is minimum. Since v is not a local minimum, it has a neighbor (lying in the interior of one of the sub-grids) whose label is even smaller. Recurse on that sub-grid.

The set S has size $O(n)$, so the number of probes performed by the algorithm satisfies $T(n) = T(n/2) + O(n)$, whose solution is $T(n) = O(n)$.

The proof of correctness follows from the observation above. Every time the algorithm recurses on a sub-grid, the observation guarantees that there is a local minimum contained in that sub-grid.