Honors Algorithms Lecture: Dynamic Programming

Julian Panetta (julian.panetta@gmail.com), based on material from Alan Siegel

September 26, 2011

Note: We covered up through 6.3 in class. I fixed the errors we caught, but let me know if you see more!

1 Greatest Common Subsequence (GCS)

Recall the introductory dynamic programming problem, GCS:

- Given strings $A[1..m],\ B[1..n]$
- Find longest string, S^*, appearing as a subsequence of both A and B. Remember, subsequences are not necessarily consecutive:

$$S = A[i_1]A[i_2]...A[i_k], \quad 1 \le i_1 < i_2 < \cdots < i_k \le m$$

- Simplification: find *length* of S^*.
- Recursive Solution: "Pac-man style" (bite letters off the right end)
  - Let $Len(i,j)$ = length of $S^*_{i,j}$, the GCS of $A[1..i],\ B[1..j]$
  - It can takes some experience to anticipate the dimensionality/type of subproblems (2D in this case)
  - Notice that $Len(i,j)$ defines a program invariant: for all i, j, $Len(i,j)$ gives the greatest length for the subproblem
  - Simply need an "upgrade rule" to compute $Len(i,j)$ from smaller-index solutions (subproblems) and the last letters, $A[i]$ and $B[j]$.
  - Add in the trivial initial conditions, and we are done:

$$Len(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \text{ else} \\ Len(i-1, j-1) + 1 & \text{if } A[i] = B[j] \text{ else} \\ max\left(Len(i-1,j), Len(i,j-1)\right) & \end{cases}$$

Analogy/Interlude: Towers of Hanoi

- We saw how to "upgrade" the $(n-1)$-ring solution to an n-ring solution
- Program invariant: TH(n, A, B, C) moves the n rings on pole A to pole B. This allows us to easily prove correctness by induction.
- This technique is indispensable for reasoning about harder problems, like the "world's slowest solver"
  - World's Slowest Solver: move the n disks on pole A to pole B while running through every possible configuration exactly once.
  - **Insight**: set of all configurations can be partitioned into three parts

  * All configurations of top $n-1$ disks with disk n on pole A
  * All configurations of top $n-1$ disks with disk n on pole C
  * All configurations of top $n-1$ disks with disk n on pole B

- Thus, if we assume (program invariant!) $SloTH(n-1, A, B, C)$ moves $n-1$ disks from pole A to pole C and reaches every configuration exactly once, we can easily extend it:

Algorithm 1 SLOTH(n, A, B, C)

```
1: if n ≥ 1 then
2:     SLOTH(n − 1, A, B, C)
3:     Move top ring, n, from A to C
4:     SLOTH(n − 1, B, A, C)
5:     Move top ring, n, from C to B
6:     SLOTH(n − 1, A, B, C)
7: end if
```

- A correctness proof by induction follows directly from the program invariant and the insight above.

2 DP Procedure

There is an methodical approach common to all dynamic programming solutions:

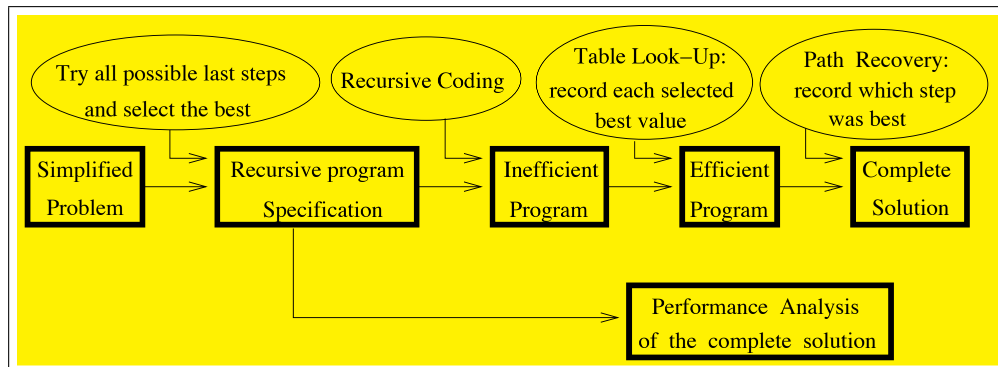


Figure 1: A dynamic programming schema where all of the problem solving is in the first balloon. (From textbook p. 511)

1. Simplify problem (find *cost* of best solution instead of the solution itself)
2. Write recursive specification (Try 'em all!) This is usually all the homework asks for, as all future steps are primarily mechanical
3. Write inefficient program (re-solves subproblems)
4. Transform into efficient program: use a look-up table to store sub-problems as they are computed
5. Find complete solution: use path recovery to get solution instead of cost

Before step 2, you should write in a few words what your cursive specification means. Ex: what is $Len(i, j)$?

3 DP Performance Advantage

Problems go from exponential to polynomial when a look-up table is introduced:

- In GCS, note there are exponentially many subsequences of A and B (every letter can be either in or out). Yikes.
- As written, recursive solution is still exponential: traverses all paths from (m, n) to $(0, 0)$ in an $(m + 1 \times n + 1)$ grid.
- After look-up table transformation it's $\theta(mn)$
  - Visualize the lookup table as a graph where each of the $\theta(mn)$ cells is a vertex.
  - Visualize the look-ups as directed edges from a cell's vertex to the vertices that cell's computation reads.
  - Since there is a constant amount of work done at each cell (up to 2 table reads + 2 letter reads + 1 arithmetic op) it only takes $\theta(mn)$ to fill out the table!
- In general, you can view your look-up table as a graph as described above, and operation count is proportional to the total number of edges

4 Another DP problem: WNWNWKW cutting problem

We already saw an example of an "alligator pac-man problem," the Waste Not Want Not Wood Knot Walnut cutting problem.

- Input: an n inch board and $Price[i, j]$ holding the retail price of an $j - i$-inch piece of wood produced by cutting at the i and j inch marks.
- Find: the cut locations yielding pieces with the largest total price.
- Simplification: find the largest total price
- Recursive Solution
  - **Insight**: we can view any collection of cuts as a single rightmost cut giving one *finished* piece and a one-smaller collection of other cuts.
  - Let $Best(j)$ be the optimal total price of the wood from mark 0 to mark j. (Program invariant!)
  - This gives the recursive specification:

$$Best(j) = \begin{cases} 0 & \text{if } j = 0 \text{ else} \\ \max_{0 \le k < j} \left(Best(k) + Price[k, j] \right) & \end{cases}$$

  $k = 0$ means a single large (j-inch) piece of wood, and $k = j - 1$ means a 1-inch piece of wood.

- Performance Analysis
  - Without table lookup:

$$T(0) = 0$$
$$T(n) = n + T(n - 1) + T(n - 2) + \cdots + T(1) + T(0)$$

This can't be solved as-is with recursion trees, but we can manipulate:

$$\begin{aligned} T(n) &= n + T(n-1) + T(n-2) + \cdots + T(1) + T(0) \\ T(n-1) &= n-1 \qquad\quad + T(n-2) + \cdots + T(1) + T(0) \end{aligned}$$

Subtracting,

$$\begin{aligned} T(n) - T(n-1) &= 1 + T(n-1) \Longrightarrow \\ T(n) &= 1 + 2T(n-1) \end{aligned}$$

This is clearly $\theta(2^n)$.

- With table lookup:
  - Location n depends on all n other values
  - Total work to fill in the table is $\theta(1 + 2 + \cdots + n) = \theta(n^2)$.
  - Can also view as a graph and count edges as we did for GCS

- Path Recovery
  - Let $Decision(j)$ store the position of the optimal solution's rightmost cut on the j-inch board from 0 to j. Update with k computed by max in the specification.
  - Following the path to retrieve the cuts is easy:

Algorithm 2 RecoverCutPath(Decisions, n)

```
1: repeat
2:     n ← Decisions(n)
3: until n = 0
```

5 Real-world Extension: LaTeX line breaking!

Given a paragraph of words, find the line breaks that make the resulting printed lines look as good as possible on a page.

What does best-looking mean? We want the text to be justified (all but the last line of each paragraph should run to the right margin). Basic version: ignore hyphenation and other rules that add work but don't change the basic challenge.

- Given: n consecutive words to break into some unknown number of lines

- Two penalty ("badness") functions (because the last line is special)
  - $NL(i, j)$: normal line badness of the words $i..j$ placed on a single justified line
  - $LL(k)$: last line badness of the words $k..n$. For those curious, a simple example is:

$$LL(k) = \begin{cases} 0 & \text{if the natural length for printing words } k..n \text{ is} < \text{page width} \\ NL(k, n) & \text{if words must be compressed to fit within the margins.} \end{cases}$$

- What do we ask? Where is the last cut? Where is the next to last?
  - This is a "Two alligator" problem, where the LL aligator bites off the last line using the LL penalty, then the NL alligator recursively bites of the remaining lines.
  - Basically the wood cutting problem with a twist: bite at end is different.

- Recursive Solution:

$$TotalCost = \min_{0 \le k < n} (RegularPenalty(k) + LL(k+1))$$

$$RegularPenaty(k) = \begin{cases} 0 & \text{if } k = 0 \\ \min_{0 \le h < n}(RegularPenalty(h) + NL(h+1, k)) & \text{otherwise.} \end{cases}$$

  - $k < n$ because there must be a last line, $k = 0$ allowed because there might not be any others be any others
  - $h < n$ because the line can't be empty and so we make progress, $h = 0$ allowed for termination

- Careful with details: consecutive lines are word ranges (k_{old}, h), $(h+1, k)$ instead of (i, j), (j, k) in the wood cutting problem. This is because indices correspond to elements (words) instead of separators.

6 Goodness Trees

The previous problems all have "flat" structure without hierarchy: there is no dependence on the order in which cuts are computed. Now we turn to some with tree structure. Typically, the tree hierarchy is implicit, but we'll start with explicit tree structures first.

These are all divide-and-conquer type problems, but unlike true divide-and-conquer there are many different places where the splitting can occur. We must try all possible splittings to know which is best.

We begin with artificial problems that are less complex and easier to understand/adapt to practice exercises.

- Let T be a tree where each vertex, v stores a *signed* numerical value, $v.dat$

- Define the goodness to be:

$$Goodness(T) = \sum_{v \in T} 2^{depth(v)} * v.dat$$

  where $depth(\text{root}) = 0$, $depth(\text{root's children}) = 1$, etc.

- Notice the definition is recursive:

$$Goodness(T) = T.dat + 2 \sum_{w \in children(T)} Goodness(w)$$

6.1 In-order Binary Goodness Trees

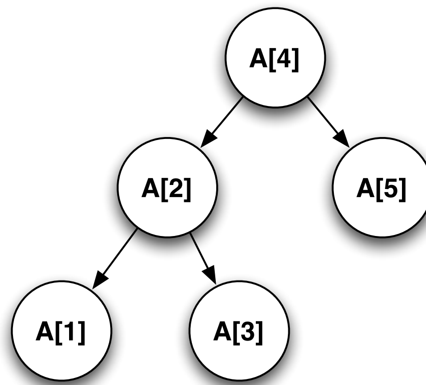


Figure 2: Example in-order binary tree for $n = 4$

- Let $A[1..n]$ be an array of signed numbers.
- We wish to construct a binary tree T such an in-order printing of vertex values reproduces sequence $A[1], A[2], \ldots$ will be reproduced. For example: $A[1]$ is stored in the left-most path.
- We want the tree with the maximum goodness:

$$Goodness(A,T) = Goodness(T) \text{ when } A \text{ is stored in } T$$

- Simplified Problem: find

$$\max_{T \in \{n\text{-vertex binary trees}\}} Goodness(A,T)$$

- Recursive Solution
  - What do we need to know to split into sub-problems? Answer: which number in $A[i..j]$ is the root? Try 'em all!
  - Once we choose a root, k, $A[i..k-1]$ goes in the left subtree and $A[k+1..j]$ goes in the right.
  - Let $Best(i,j)$ give the optimal binary tree goodness for the sequence $A[i..j]$.

$$Best(i,j) = \begin{cases} 0 & \text{if } i > j \\ \max_{i \le k \le j}(2Best(i,k-1) + A[k] + 2Best(k+1,j)) & \text{otherwise.} \end{cases}$$

  - Factors of 2 are because the subproblems "think" they are trees instead of subtrees (and all their vertex depths must be increased by 1).
  - k bounds: any number in the sequence gets a chance to be the root.
  - Initial condition: empty trees have 0 goodness. Initial conditions can be tricky, so you should do this after solving the problem.

- Performance analysis:
  - The look-up table is half an $n \times n$ array. Each location does $j - i + 1$ table reads.
  - We can easily bound this approximately by $O(n * n * n) = O(n^3)$
  - We can do it exactly by computing:

$$\sum_{1 \le i \le n} \sum_{j \ge i} \sum_{i \le k \le j} 1$$

  - We can even approximate it with a continuous computation:

$$\int_{x=0}^{n} \int_{y=x}^{n} \int_{z=x}^{y} dzdydx = \int_{0<x<z<y<n} 1dzdydx = \frac{1}{3!} \int_{x,y,z \in [0,n]} dzdydx = n^3/6.$$

6.2 Pre-order Binary Goodness Trees

The problem doesn't change much when you look at pre-order trees.

- Now you know what element goes in the root ($A[1]$)
- What's the correct question now? We need to know where to split into left and right subtrees.
- $A[2]$ is in left subtree root... which element is stored in right subtree root?

- Recursive Solution:

$$Best(i,j) = \begin{cases} 0 & \text{if } i > j \\ \max_{i+1 \le k \le j+1}(A[i] + 2Best(i,k-1) + 2Best(k,j)) & otherwise. \end{cases}$$

  - k is the pre-order id of the subproblem root's right child. Convince yourself of bounds!
  - It's true due to symmetry it isn't necessary to allow either the left *or* the right subtree to be empty, but our objective is to write general solutions that will hold for a broader class of problems.

6.3 Arbitrary Goodness Trees

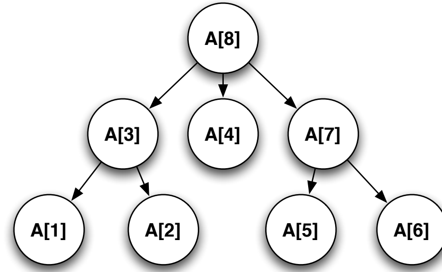


Figure 3: Example post-order arbitrary tree for $n = 8$

Now we generalize from binary trees to arbitrary trees. We'll look at a post-order goodness arbitrary tree problem (so $A[n]$ will be written in the root).

- This problem is noticeably more difficult. The naïve approach of choosing an arbitrary number of children is intractable.
- We need a recursive definition of a tree. There are two natural ones:
  - A tree is either a single root, or a tree with a subtree rooted by the rightmost child
  - A tree is either a single root, or a tree with a subtree rooted by the leftmost child
- In other words, you can decompose a T into trees S and R, where R is the rightmost subtree of T and S is everything else.
- Or you can decompose a T into trees S and L, where L is the leftmost subtree of T and S is everything else.
- One of these is better than the other! The "worse solution" will be likely inefficient, and will be much harder to implement.
- For post-order, if we bite off the rightmost subtree we see:
  - We fill the right subtree with data $A[k..n-1]$
  - The remaining tree subproblem gets the data $A[1..k-1]$ and $A[n]$ for the root. Uh-oh!
  - Future bites will leave additional pieces, and the problem gets worse.
- If we bite of the leftmost subtree:
  - We fill the left subtree with $A[1..k]$

- The remaining tree gets $A[k+1..n]$.
  - There is no fracturing!

- Recursive Solution:

$$Best(i,j) = \begin{cases} A[i] & \text{if } i = j \\ \max_{i \le k < j}(Best(i,k) + 2Best(k+1,j)) & otherwise \end{cases}$$

- Note: all root nodes are handled/all data is stored in initial conditions.

- Moral: don't fracture the data!

6.4 Summary of Tree Processing

- There is a hierarchy b/c there is just one root of the entire solution tree and different local roots have different weightings.

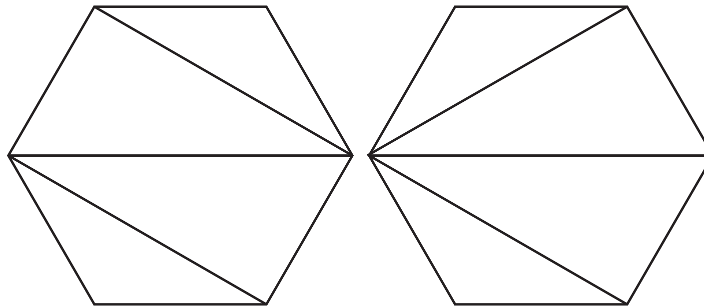
- Although the data is $1D$, the subproblems are accessed by $2D$ lookup table.

- Solution for $Best(i,j)$ is independent of the structure of the rest of the tree (i.e. it is independent of where that subtree is placed in the full tree)

- Run time of all these type of problems we've seen is $\approx \frac{n^3}{6} = \theta(n^3)$

7 Implicit Tree Structure: Convex Polygon Triangulation

This example highlights the danger of data fracturing. Our solution imposes a tree structure that isn't really forced by the problem and yields another $\theta(n^3)$ solution. There are more efficient solutions, but the problem of improving the following solution was open for more than a decade before it was answered.



- Convex Polygon: no "indentations," holes, or self crossings. If we place a rubber band around points, it forms a convex polygon. Alternative definition: a line connecting any two points of a convex polygon will lie entirely within the polygon.

- Triangulation: the insertion of non-intersecting, "slicing" edges between vertices so that the polygon is split into triangles.

- Basic fact: a convex polygon with n vertices requires $n-3$ new slicing edges to be triangulated.

- Input: Let $(i_1, j_1), (i_2, j_2), \ldots (i_n, j_n)$ form a convex n-vertex polygon, P, when the vertices are connected in the order as listed.

- Problem: find the triangulation where the sum of the $n-3$ slicing edge lengths is as small as possible.
- What do we ask? Notice we can draw an edge between any non-neighboring vertices to split the problem into independent subproblems–good!
- Try to define the subproblems:
  - Consider any subset of vertices. A sub-polygon can be formed from these specific vertices and no others by triangulating edges
  - Proof: draw sub-polygon. All of sides are either sides of P, or belong to a valid set of $n-3$ triangulating slices.
  - BAD: exponentially many sub-problems (subsets of vertices).
- The Fix: avoid fracturing data
  - We organize the slices so that each subpolygon is formed from consecutive vertices from the original sequence + a single "closing-off" edge that connects two non-consecutive vertices
  - How? Slice out triangles where one of the edges is a "closing-off" edge (choose a third vertex).
  - Sub-polygons formed this way will *always* have just one "closing-off" edge

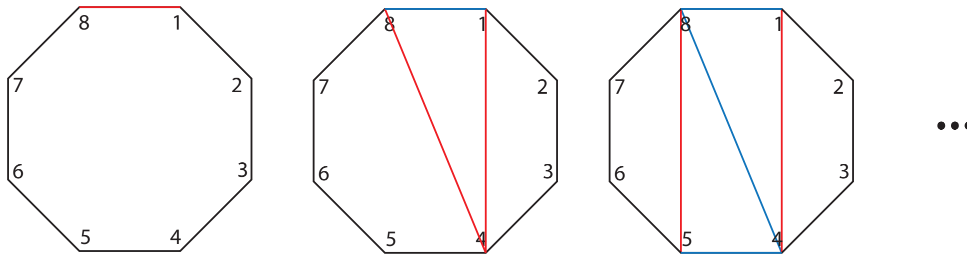


Figure 4: First triangulation steps using the fracture-avoiding technique. Active "closing-off" edges are shown in red, and black edges show the yet-to-be triangulated sub-polygons.

  - This is fully general because every edge edge that is part of a sub-polygon must belong to a triangle formed with another vertex in that sub-polygon.
- Recursive solution:
  - Let $Best(\ell, k)$ be the sum of the lengths of slices that triangulate vertices $\ell,\ \ell+1,\ \ldots,\ k$. We will require that $\ell < k$.
  - Note: the "closing-off" edge will always connect vertices ℓ and k.
  - For $a < b$, we define a distance function that only counts slicing edges:
$$dist(a,b) = \begin{cases} \sqrt{(i_a - i_b)^2 + (j_a - j_b)^2} & \text{if } b > a+1 \\ 0 & \text{if } b = a+1. \end{cases}$$
  - Solution is:
$$Best(\ell,k) = \begin{cases} 0 & \text{if } k \leq \ell + 2 \\ \min_{\ell<h<k} Best(\ell,h) + dist(\ell,h) + dist(h,k) + Best(h,k) \end{cases}$$
  - Explanation of bounds:
    * if $k = \ell + 1$, then the edge is original and not "closing-off"
    * if $k = \ell + 2$, then the sub-polygon is already just a triangle
    * h runs over all other sub-polygon vertices.