Exercises 4

Good problems

DPV6.2 We define our subproblems as follows. D(i) is the minimum penalty when our last stop is at hotel a_i. We have that D(0) = 0: we start with penalty 0. Then we have the recurrence

$D(i)$= min { D(k) + (200 - $(a_i - a_k)$)^2}, for $1 \leq k < i$

In other words, if we stop at hotel a_i, we check all possible previous stopping points. Our result is D(n), since we must stop at hotel a_n.

Runtime: We have n subproblems. It takes O(n) time to compute each subproblem. Hence our algorithm runs in O(n^2) time.

Correctness: We prove correctness of our algorithm by induction on n. Base case: D(0) = 0. We have 0 penalty at the very start of our trip. Inductive step: Assume D(k) is correct for all k < i. Consider D(i). If we stop at hotel i, our previous stopping location could be any hotel in [1, i-1]. We check all of these. By the inductive hypothesis, we know D(k) is correct for all [1, i-1]. The penalty if we stop at hotel a_i and our last stop was hotel a_k is D(k) plus the penalty for this day, which is (200 - $(a_i - a_k)$)^2. This is exactly what our algorithm calculates, so D(i) must be correct.

DPV6.4 A) We define our subproblems as follows. D(i) is true if s[$1 \dots i$] is a sequence of valid words. We have that D(0) = true, since this is the empty string. Then we have the recurrence

$D(i) = \vee_{1 \leq k < i} \{ dict(s[k \dots i]) \wedge D(k-1) \}$

In other words, D(i) is a word if there exists some k < i where s[k…i] is a word and s[1…k-1] is a valid sequence of words. Then our final result is D(n): whether s[1…n] is a valid sequence of words.

Runtime: There are n subproblems. Each subproblem takes O(n) time to compute. Hence our algorithm runs in O(n^2) time.

Correctness: We prove correctness of our algorithm by induction on n. Base case: D(0) = 0; the empty string is valid. Inductive step: Assume D(j) is correct for all k < i. Consider D(i). It must have at least one word, which ends at i. The starting index of this word can be anywhere in [1, i-1]. By the inductive hypothesis, we know D(k) is correct for all [1, i-1]. For a last word starting index k, D(i) is valid if s[k…i] is a word and s[1…k-1] is a valid sequence of words. Our algorithm checks all of these, so D(i) must be correct.

B) To actually print out the sequence of words, for each D(i), we record which starting index k made our equation $dict(s[k \dots i]) \wedge D(k-1)$ true. Then we can follow this sequence of indices starting from n to find the breakpoints of each word and reconstruct the sequence.

DPV6.7

We define our subproblems as follows. P(j, k) = true if the substring of s from index j to k is a palindrome. Every P(j, j) = true. For all j < k, our recurrence is

$$P(j,k) = (s[j] == s[k]) \wedge ((k-j \leq 2) \vee P(j+1, k-1))$$

Then our result is the largest k-j for all P(j, k) = true. In other words, P(j, k) is true if the characters at j and k are the same and the substring between characters j and k is a palindrome.

Runtime: There are O(n^2) subproblems, since j, k are both in the range [1, n]. Each subproblem takes O(1) time to compute, so we have that the runtime is O(n^2).

DPV6.9

We define our subproblems as follows. D(j, k) is the minimum cost to make all cuts between cut[j] and cut[k]. We add two dummy cuts to our cuts array: 0 at the beginning and n at the end to represent the beginning and end of the string. We have that for all consecutive j, k, D(j, k) = 0. We have the recurrence

$$D(j,k) = cuts[k] - cuts[j] + \min\{ D(j,l) + D(l,k)\} for\ j < l < k$$

Our solution is then D(1, m+1), the minimum cost to make all the cuts between the beginning and end of the string.

Runtime: There are O(m^2) subproblems, since j, k are both in the range [1, m]. Each subproblem takes O(m) time to compute, so we have that the runtime is O(m^3).

Correctness: We prove correctness of this algorithm by induction on k-j. Base case: k-j=1. Then these are two consecutive cuts, so there are no cuts between these two cuts and D(j,k) = 0. Inductive step: Assume D(x, y) is correct for all y-x < i. Consider D(j, k) where k-j=i. At this point, there are multiple cuts we can choose from between cut[j] and cut[k]. If we choose to execute some cut[l], it takes cuts[k]-cut[j] time to execute this first cut, plus the time it takes to execute cuts between j and l, and l and k (D(j,l) + D(l,k)). By the inductive hypothesis, for all j < l < k, D(j, l) and D(l, k) are computed correctly. Our algorithm checks all possible values of l and takes the minimum of these. Hence D(j,k) is computed correctly.

DPV6.21

We pick some vertex to be the root. We define our subproblems as follows. D(v) is the weight of the minimum vertex cover for the tree rooted at v. If v is a leaf, D(v) = 0. If v is not a leaf, we can either include v in the vertex cover or not include it. Let's call the minimum vertex cover for the subtree M.

- If v is in M, M - v is a minimum weight vertex cover for the forest consisting of all v's children. Then the cost of M is 1+ $\sum_{u\ child\ of\ v} D(u)$
- If v is not in M, then all of v's children must be in M, since we must cover each edge from v to its children. Then the cost of M is $\sum_{u\ child\ of\ v} 1 + \sum_{w\ grandchild\ of\ v} D(w)$

We can compute this by traversing every vertex of T once and using memoization. Our result is D(root).

Runtime: There are |V| subproblems. Each subproblem requires (#children + #grandchildren) lookups, which is

$$\sum_{v\ in\ V} \#children + \#grandchildren = \sum_{v\ in\ V} \#parent + \#grandparent \leq \sum_{v\ in\ V} 2 = 2$$

Hence the time complexity is O(|V|).

Correctness: We use induction on the vertices v to prove the correctness of our algorithm. Base case: the algorithm is correct for leaves, since leaves have no children. Inductive step: Assume the algorithm computes D(u) correctly for all vertices u with subtree smaller than v. Then there are only 2 cases as we described above: v is in the vertex cover or v is not in the vertex cover. By the inductive hypothesis and our discussion above, we have that our algorithm must compute D(v) correctly.

DPV6.26 We define our subproblems as follows. D(i, j) is the maximum score of the alignment of x[1…i] and y[1…j]. We have that D(0, 0) = 0. Then we have the recurrence

D(i, j) = max $\{\delta(x[i], y[i]) + D(i-1, j-1), \delta(x[i], -) + D(i-1, j), \delta(-, y[j]) + D(i, j-1)\}$

Our result is then D(n, m).

Runtime: There are O(nm) subproblems. Each subproblem does 3 computations. Hence the runtime of our algorithm is O(nm).

Correctness: We prove the correctness of our algorithm using induction on i+j. Base case: i=0 and j=0. D(0, 0) = 0, since the score of two empty strings is 0. Inductive step: Assume D(a, b) is true for all a+b < i+j. Consider D(i, j). There are 3 cases when we are at index i in string x and index j in string y.

- We match x[i] with y[i]. Then the score is $\delta(x[i], y[i]) + D(i-1, j-1)$
- We match x[i] with $-$. Then the score is $\delta(x[i], -) + D(i-1, j)$
- We match y[i] with $-$. Then the score is $\delta(-, y[j]) + D(i, j-1)$

By the inductive hypothesis, D(i-1, j-1), D(i-1, j), and D(i, j-1) are all correct. Our result is the best choice among these 3 cases, which is exactly what our algorithm computes. Hence we compute D(i, j) correctly.

CLRS15-4 Assume the length of each word is less than or equal to M. We define our subproblems as follows. D(j) is the minimum sum of squares of number of extra spaces for the first j words. Then D(1) = $(M - l_1)^2$. We have the recurrence

D(j) = min { D(i-1) + $(M - j + i - \sum_{k=i}^{j} l_k)^2$ }, for all $0 < i < j$ if $\sum_{k=i}^{j} l_k \leq$ M

D(n) = min { D(i-1) }, for all $0 < i < n$ if $\sum_{k=i}^{n} l_k \leq$ M

In other words, since word j is the last word, it is on the last line. We compute all possible starting points for this line and calculate their minimum sum of squares of number of extra spaces. We have a special case for D(n) since we don't care about the last line. The minimum sum of squares of extra spaces for all words is D(n) - using all the words. To obtain the actual line break locations, for each D(j) we keep track of which i we used as the line break. Then to obtain the line breaks, we start at D(n), find the last line break, and so on.

Runtime: There are n subproblems. For each subproblem, we iterate at most M times, since M is the maximum number of characters per line. The summation in the recurrence equation can be calculated in O(1) time in each iteration by keeping a running sum as we iterate starting

from j. Hence our runtime is O(nM).

Correctness: We prove the correctness of our algorithm on using induction on j. Base case: j= 1. Since we only have one word, we only have one line. The square of number of extra spaces is D(1) = $(M - l_1)^2$, which is what our algorithm calculates. Inductive step: Assume D(k) is correct for all 0 < k < j. Since word j is the last word, it is on the last line. We iterate through all possible starting locations for this last line. The minimum sum of squares of number of extra spaces for the previous lines is D(i-1), where i is the start index of the last line. Hence by the recurrence relation, our algorithm calculates D(j) correctly. Hence our algorithm is correct.

Interesting problems

DPV6.10

We define our subproblems as follows. D(i, j) is the probability of obtaining j heads when tossing the first i coins. D(0, 0) = 1, since when tossing 0 coins we obtain 0 heads. D(0, j) = 0 for all j > 0, since we cannot obtain more than 0 heads when tossing 0 coins. We have the recurrence

$$D(i,j) = D(i-1,j-1) \cdot p_i + D(i-1,j) \cdot (1-p_i)$$

Our final result is then D(n, k), the probability of obtaining k heads when tossing all n coins.

Runtime: There are O(nk) subproblems. For each subproblem, we have 2 lookups each taking O(1) time. Hence the runtime of our algorithm is O(nk), which is in O(n^2) since $k \leq n$.

Correctness: We prove the correctness of our algorithm by induction on n. Base case: i=0. D(0, 0) = 1, since when tossing 0 coins we obtain 0 heads. D(0, j) = 0 for all j > 0, since we cannot obtain more than 0 heads when tossing 0 coins. This is exactly what our algorithm calculates. Inductive step: Assume D(i-1, j) is correct for all j. Consider D(i, j). To get j heads, we need to either get a head on coin i and have j-1 heads for all the previous coins, or get a tail on coin i and have j heads for all the previous coins. The probability of the former is $D(i-1,j-1) \cdot p_i$, and the probability of the latter is $D(i-1,j) \cdot (1-p_i)$. The sum of the two is D(i, j), which is exactly what our algorithm calculates. Hence our algorithm calculates D(i, j) correctly.

DPV6.12

We first define d_{ij} as the length of a diagonal between vertices i and j, calculated as $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ if $j - i \geq 2$, 0 otherwise

We define our subproblems as follows. A(i, j) is the minimum cost triangulation of the polygon spanned by vertices i to j. A(i, i) = 0 for all i. We have the recurrence

$$A(i,j) = \min\{A(i,k) + A(k,j) + d_{ik} + d_{kj}\} \; for \; i \leq k \leq j$$

Our final solution is A(1, n), the minimum triangulation of the polygon spanned by all the vertices.

Runtime: There are O(n^2) subproblems. Computation of each subproblem takes O(n) time. Hence our algorithm runs in O(n^3) time.

Correctness: We prove correctness of our algorithm by induction on $j - i$. Base case: $j - i = 0$. D(i,i) = 0 for all i, since we can form 0 diagonals on a single vertex. Inductive step: Assume D(a, b) is computed correctly for all $b - a < j - i$. Consider D(i, j). We can triangulate at any

vertex $i \leq k \leq j$, and the cost is then the diagonals ik and kj, which is d_{ik} and d_{kj} respectively, as well as the cost to triangulate the polygon spanning vertice i,k and k,j, which is $A(i,k)$ and $A(k,j)$, respectively. Then the total cost of triangulating at vertex k is the sum of these. We try every possible vertex k and take the minimum cost option, so D(i, j) is correct.

DPV6.16 This is like traveling salesman except we want to consider all subsets, since we don't need every garage to be on our path. We define our subproblems as follows. D(i, S) is the best path ending at i, with vertices in S on the path. We have that $D(i,\{i\}) = g_i - d_{0i}$ for all $1 \leq i \leq n$. We have the recurrence

$$D(i,S) = \min\{ D(j, S - \{i\}) + g_i - d_{ji}\} \, for \, all \, j \in S - \{i\}$$

In other words, to compute D(i, S), we try all possibilities for the second-last vertex j in the path. Our final solution is $\min\{ D(i,S) - d_{i0}\}$ for all i: all possible tours of subsets, remembering to subtract the final leg d_{i0} back to home.

Runtime: There are O($n2^n$) subproblems, each requiring O(n) time to compute. Hence the runtime of our algorithm is O($n^2 2^n$).

Correctness: We prove correctness of our algorithm by induction on |S|. Base case: |S| = 1: $D(i,\{i\}) = g_i - d_{0i}$, since for a path ending at one garage, we just have the value of the garage minus the cost to get there. Inductive step: Assume D(j, T) is correct for all |T| = k-1. Consider D(i, S). The last garage on the path is i. The second-last vertex on the path must be some vertex in $S - \{i\}$. Let j be some vertex in $S - \{i\}$. The cost if the second-last vertex is j is the cost where j is the last vertex, plus the garage value of i minus the cost to get from j to i. $D(j, S - \{i\})$ is correct by the inductive hypothesis. The minimum value among all possible j is thus the minimum cost option. This is exactly what our algorithm calculates. Hence we calculate D(i, S) correctly.

DPV6.25 We define our subproblems as follows. D(j, x, y, z) is whether it is possible for the sums of elements in I, J, K to be x, y, z respectively by partitioning elements $a_1 \ldots a_j$. We have that D(0, 0, 0, 0) = true, since the sum of 0 elements is 0. D(0, x, y, z) = false for all other x, y, z, since we cannot use 0 elements to get a nonzero sum. We have the recurrence

$$D(j,x,y,z) = D(j-1, x-a_j, y, z) \vee D(j-1, x, y-a_j, z) \vee D(j-1, x, y, z-a_j)$$

Let S = $1/3 \sum_{i=1}^{n} a_i$. Then our final solution is D(n, S, S, S), whether it is possible to partition all the elements to get a sum of S for each partition.

Runtime: There are O(n$(\sum_{i=1}^{n} a_i)^3$) subproblems. Each subproblem does 3 lookups in O(1) time. Hence the runtime of our algorithm is O(n$(\sum_{i=1}^{n} a_i)^3$).

Correctness: We prove correctness of our algorithm by induction on n. Base case: D(0, 0, 0, 0) = true, since the sum of 0 elements is 0. D(0, x, y, z) = false for all other x, y, z, since we cannot use 0 elements to get a nonzero sum. Inductive step: Assume D(j-1, a, b, c) is true for all a, b, c. Consider D(j, x, y, z). To use element a_j, we must place it in one of the partitions. To get sum x, if we place it in partition I, we must have that the sum without a_j is $x - a_j$. Whether such a sum is possible is $D(j-1, x-a_j, y, z)$. It is symmetric for placing it in partitions J and K. Hence if any one of these is possible, then D(j, x, y, z) is true. This is exactly what our algorithm computes, so we compute D(j, x, y, z) correctly.

CLRS15-3 Let $d_{i,j}$ be the Euclidean distance between points i and j.

We first sort all points by x-coordinate. We model this problem as two people walking disjoint paths from left to right, where the union of their paths cover all vertices. We define our subproblems as follows. D(i, j) is the sum of lengths of the 2 paths, where person 1 has reached vertex i and person 2 has reached vertex j. This means all vertices $1 \dots \max(i, j)$ have been covered. Since $D(i, j) = D(j, i)$, assume $i \leq j$ from now on. We have that D(0, 0) = 0: if we have 0 vertices we have 0 distance. We have the recurrence

$$D(i,j) = \begin{cases} D(i, j-1) + d_{j-1,j} & \text{, if } i < j-1 \\ \min\{D(i,k) + d_{k,j}\} \text{ for } 1 \leq k < j & \text{, otherwise (if } i = j-1 \text{ or } i = j) \end{cases}$$

Our final solution is D(n, n), when both people have reached the end.

Runtime: There are O(n^2) subproblems that fall under case 1, each of which takes an O(1) computation. There are O(n) subproblems that call under case 2, each of which takes O(n) time to compute. Hence our algorithm runs in O(n^2) time.

Correctness: We prove correctness of our algorithm by induction on j. Base case: We have that D(0, 0) = 0: if we have 0 vertices we have 0 distance. Inductive step: Assume D(i, k) is correct for all k < j. Consider D(i, j). There are two cases:

- If $i < j - 1$, then the path ending in j must also visit j-1, since the other path cannot visit j-1 and then backtrack to i. The second path increments to j, and we have that the distance is the distance to get to i,j-1, plus the distance from j-1 to j: $D(i, j-1) + d_{j-1,j}$
- Otherwise (if $i = j - 1$ or $i = j$), then the optimal solution must have a path which ends in j and comes from some vertex k < j. We select the best of these paths, which computes to $\min\{D(i,k) + d_{k,j}\}$ for $1 \leq k < j$

By the inductive hypothesis, $D(i, j-1)$ and $D(i, k)$ is correct for all other k < j. This is exactly what our algorithm computes, so we compute D(i, j) correctly.

Challenging problems

DPV6.30 A) WIP