

CSE 101 Homework 4
Dynamic Programming
Due Thursday, March 10
100 points total = 10 %

Gizmos (20 points) Consider the following problem. You wish to purchase (at least) n identical gizmos. Gizmos come in packages of different sizes and different prices. You can buy any number of packages of each size, as long as the total number is at least n . You wish to find the minimum total price of such a set of packages.

The input is given as n and an array $Packages[1..m]$, where each $Package[i]$ has a positive integer field $Package[i].size$ and a positive real field $Package[i].price$ giving the number of gizmos in the package and the price of the package.

A recursive algorithm to solve this problem is:

$BestPrice[n : positiveinteger, Packages[1..m] : array of pairs (size: integer, price: real).]$

1. $MinPrice \leftarrow \text{inf}$;
2. For $d = 1$ to m do:
3. begin;
4. IF $Packages[d].size \geq n$ THEN $TempPrice \leftarrow Packages[d].price$
5. ELSE $TempPrice \leftarrow Packages[d].price + BestPrice(n - Packages[d].size, Packages)$;
6. IF $TempPrice < MinPrice$ THEN $MinPrice \leftarrow TempPrice$;
7. end;
8. Return $MinPrice$.

Part 1: 2 points Show the recursion tree of the above algorithm on the following input: $n = 6$, packages: buy 5 for \$ 12, 3 for \$8 or 2 for \$6.

Case 1: Buy package of 5. Cost: $12 + BestPrice(1, Packages)$

Case 1a: Buy package of 5 \wedge 1. Cost:12

Case 1b: Buy package of 3 \wedge 1. Cost:8

Case 1c: Buy package of 2 \wedge 1. Cost:6

Minimum: 6. Case 1 returns $12 + 6 = 18$.

Case 2: Buy package of 3. Cost: $8 + BestPrice(3, Packages)$

Case 2a: Buy package of 5 \wedge 3. Cost:12

Case 2b: Buy package of 3 =3. Cost:8

Case 2c: Buy package of 2. Cost: $6 + BestPrice(1, Packages)$

Case 2cI: Buy package of 5 \wedge 1. Cost:12

Case 2cII: Buy package of 3 \wedge 1. Cost:8

Case 2cIII: Buy package of 2 & 1. Cost:6
 Minimum=6. Case 2c returns 6+6 =12. Minimum for Case 2 is 8.
 Case 2 returns 8+8=16
 Case 3: Buy package of 2. Cost: 6+BestPrice(4,Packages)
 Case 3a: Buy package of 5 & 4. Cost:12
 Case 3b: Buy package of 3 . Cost:8+BestPrice(1,Packages)
 Case 3bI: Buy package of 5 & 1. Cost:12
 Case 3bII: Buy package of 3 & 1. Cost:8
 Case 3bIII: Buy package of 2 & 1. Cost:6
 Minimum=6. Case 3b returns 8+6 =14.
 Case 3c: Buy package of 2. Cost:6+BestPrice(2,Packages)
 Case 3cI: Buy package of 5 & 2. Cost:12
 Case 3cII: Buy package of 3 & 2. Cost:8
 Case 3cIII: Buy package of 2 =2 . Cost:6
 Minimum=6. Case 3b returns 6+6 =12.
 Case 3 minimum is 12. Case 3 returns 6+12 =18.
 Overall minimum is Case 2, 16, which is returned by the main procedure.

Part 2: 3 points Give a bound on the worst-case number of recursive calls the recursive algorithm could make in terms of n and m .

There are at most m recursive calls, and each reduces the value of n by at least 1. This gives a tree of fan-out m and depth at most n , so a total of $O(m^n)$ recursive calls.

Assume all packages have distinct sizes. (If not, we could just delete all but the least expensive package of a given size.) Then we make in terms of n , $T(n) = T(n - size_1) + T(n - size_2) + \dots T(n - size_m) \leq T(n-1) + T(n-2) + \dots$. We can use induction to prove $T(n) \in O(2^n)$. (See last answer key.)

Part 3: 10 points Give a dynamic programming version of the recurrence.

Note that only the value of n , not the set of Packages, changes in recursive calls, and that takes on values from $1 \dots n$. Also, n decreases in each recursive call. So we should fill in an array of one dimension of size n , in increasing order of n . This leads to:

DPBestPrice[n : positiveinteger, Packages[1.. m] : array of pairs ($size$: integer, $price$: real).]

1. Initialize $MP[1..n]$.
2. FOR $N = 1$ to n do:
3. $MinPrice \leftarrow \text{inf}$;

4. FOR $d = 1$ to m do:
5. IF $Packages[d].size \geq N$ THEN $TempPrice \leftarrow Packages[d].price$
6. ELSE $TempPrice \leftarrow Packages[d].price + MP[N - Packages[d].size]$;
7. IF $TempPrice < MinPrice$ THEN $MinPrice \leftarrow TempPrice$;
8. $MP[N] \leftarrow MinPrice$.
9. Return $MP[n]$.

Part 4: 3 points Give a time analysis of this dynamic programming algorithm, in terms of n and m .

There are two nested loops, one going from 1 to n , the other from 1 to m , which gives a total time of $O(nm)$.

Part 5: 2 points Show the array that your algorithm produces on the above example.

1. $6 = \min(6, 8, 12)$
2. $6 = \min(6, 8, 12)$
3. $8 = \min(6 + MP[1]=12, 8, 12)$
4. $12 = \min(6 + MP[2]=12, 8 + MP[1]=14, 12)$
5. $12 = \min(6 + MP[3]=14, 8 + MP[2]=14, 12)$
6. $16 = \min(6 + MP[4]=18, 8 + MP[3]=16, 12 + MP[1]=18)$

For each of the following problems, describe the fastest dynamic programming algorithm you can find, and give a time analysis (in terms on any of the given parameters).

One Dimensional Clustering: 20 pts You are given n real numbers r_1, r_2, \dots, r_n and an integer $1 \leq k \leq n$. You want to find k disjoint intervals $I_1 = [a_1, b_1], I_2 = [a_2, b_2], \dots, I_k = [a_k, b_k]$ so that each $r_i \in I_j$ for some j , in a way that minimizes the sum of the squares of the length of the intervals, $\sum_{j=1}^k (b_j - a_j)^2$.

Give an efficient algorithm for this problem. Our best time is $O(n^2k)$.

Assume the input is sorted. (This takes $O(n \log n)$ time). The first interval will always begin at r_1 , since otherwise we can shorten it and reduce the costs. A backtracking approach is based on answering the question: Where should the first interval end? Similarly, it must end at some r_i , so the possible answers are: at r_1, \dots at r_{n-k+1} (Since otherwise we'll have fewer future points than intervals.) If we are down to one interval, it must be $[r_1, r_n]$. This gives the following recursive algorithm:

$BT1DC[r_1..r_n]$: sorted array of reals, k : integer between 1 and n]: best cost of a clustering into k intervals.

1. IF $k = 1$ return $(r_n - r_1)^2$.

2. $BestCluster \leftarrow infinity$.
3. FOR $I = 1$ to $n - k + 1$ do:
 4. $ThisCase \leftarrow (r_I - r_1)^2 + BestCluster(r_{I+1}..r_n; k - 1)$.
 5. IF $ThisCase < BestCluster$ THEN $BestCluster \leftarrow ThisCase$.
6. Return $BestCluster$.

We can see that this always makes recursive calls of the form $BestCluster(r_i..r_n, k')$ for $1 \leq i \leq n$ and $1 \leq k' \leq k$. So we can use dynamic programming, setting up an array $bc[1..n][1..k]$ to store answers, with $bc[i][k']$ intended to store $BestCluster(r_i..r_n, k')$. Since in top-down order k' always decreases by one, we fill in this array from $k' = 1$ to k , with $k' = 1$ being the base case. Also note that we always have $i \leq n - k' + 1$, in order to have at least one point per cluster. This gives the dynamic programming version: $DP1DC[r_1..r_n]$: sorted array of reals, k : integer between 1 and n]: best cost of a clustering into k intervals.

1. Initialize $bc[1..n][1..k]$.
2. FOR $J = 1$ to n do:
 3. $bc[J][1] \leftarrow (r_n - r_J)^2$.
 4. FOR $K = 2$ to k do:
 5. FOR $J = 1$ to $n - K + 1$ do:
 6. $BestCluster \leftarrow infinity$.
 7. FOR $I = J$ to $n - k + 1$ do:
 8. $ThisCase \leftarrow (r_I - r_J)^2 + bc[I + 1][K - 1]$.
 9. IF $ThisCase < BestCluster$ THEN $BestCluster \leftarrow ThisCase$.
 10. $bc[J][K] \leftarrow BestCluster$.
11. Return $bc[1][k]$.

The running time is dominated by the three nested loops, taking time k , n and n respectively, for a total time of $O(n^2 k)$. This also dominates the pre-processing time for sorting.

Descending partitions-20pts A descending partition of positive integer N is a sequence of positive integers $A_1 > A_2 > \dots > A_k$ with $\sum_{i=1}^k A_i = N$. Give an efficient (poly-time in N) algorithm that, given N , computes the NUMBER of decreasing partitions of N . For example, if $N=6$, the decreasing partitions are: (6); (5, 1); (4, 2); (3, 2, 1) so your algorithm, on input 6 should return 4. (14 points correct poly-time algorithm, 6 pts. efficiency, e.g. N^2 vs. N^3 time)

A back-tracking, recursive algorithm is as follows: We pick the first number in our sequence, and then recurse, but we need to remember the number we picked, so that we do not try possibilities larger than this number. So the general problem is: Compute the number of decreasing partitions of N with elements all less than or equal to M . This gives the following recursion: $Count(N, 1) = 0$ if $N > 1$, $Count(0, M) = 1$, $Count(1, M) = 1$, and $Count(N, M) = \sum_{A=2}^{A=N} Count(N - A, A - 1)$ for $N, M > 1$. (If the first number in the partition is A , the rest sum to $N - A$ and all elements are less than $A - 1$.)

We can use this recursion in a dynamic programming algorithm, since only N^2 possible inputs to $Count$ are called. The recursion always calls values with smaller N , so bottom-up order can be in increasing order for N .

- PartitionNumber(N);
- Initialize $count(0..N, 1..N)$.
- FOR $I = 2$ TO N do: $count(I, 1) \leftarrow 0$
- FOR $J = 1$ TO N do: $count(0, J) \leftarrow 1$
- FOR $J = 1$ TO N do: $count(1, J) \leftarrow 1$
- FOR $I = 2$ TO N do: FOR $J = 2$ TO N do:
 - $sum \leftarrow 0$
 - FOR $K = 1$ TO J do: $sum \leftarrow sum + count(I - K, K - 1)$;
 - $count(I, J) \leftarrow sum$;
- Return $count(N, N)$.

This takes time $O(N^3)$, from the three nested loops.

A better method is based on a recursion branching on the least element, rather than the largest. If $a_1 > a_2 > \dots > a_k$ is a partition of N , then $a_1 - a_k > a_2 - a_k > \dots > a_{k-1} - a_k$ is a partition of $N - ka_k$. Since this mapping is invertible, it preserves the number of partitions of length k that sum to N and end in a_k . So if we let $Count2(N, K)$ stand for the number of partitions of N of size K , we get the recurrence: $Count2(N, 1) = 1$, $Count2(1, K) = 0$ if $K > 1$, and otherwise $Count2(N, K) = \sum_{1 \leq A \leq N/K} Count2(N - AK, K - 1)$. Using this in a dp algorithm gives:

1. PartitionNumber2(N);
2. Initialize $count2(1..N, 1..N)$;
3. FOR $I = 1$ TO N do: $count2(I, 1) \leftarrow 1$;
4. FOR $J=2$ TO N do: $count2(1, J) \leftarrow 0$;
5. FOR $J=2$ TO N do: FOR $I=2$ TO N do:

6. $sum \leftarrow 0$
7. FOR $A = 1$ TO $\lfloor I/J \rfloor$ do: $sum \leftarrow sum + count(I - AJ, J - 1)$;
8. $count(I, J) \leftarrow sum$;
9. Return $\sum_{1 \leq J \leq N} count(N, J)$.

The time here is a bit more complicated. For each J , let's upper bound the inside loop by N/J , so the inside two loops are bounded by N^2/J . Then the total time is bounded by $\sum_{J=1}^{J=N} N^2/J = N^2 \sum_{J=1}^{J=N} 1/J$. The inside sum is the N 'th harmonic number, which approaches $\ln N$, so the total time is $O(N^2 \log N)$, which dominates the rest of the algorithm.

Library storage-20pts A library has n books that must be stored in alphabetical order on adjustable height shelves. Each book has a height and a thickness. The width of the shelf is fixed at W , and the sum of the thicknesses of books on a single shelf must be at most W . The next shelf will be placed on top, at a height equal to the maximum height of a book in the shelf. You are given the list of books in alphabetical order, $b_i = (h_i, t_i)$, where h_i is the height and t_i is the thickness, and must organize the books in that order. Last homework, you gave a backtracking algorithm that minimizes the total height of shelves used to store all the books. This time, give an efficient dynamic programming algorithm for the same problem.

We could come up with a recursive formula, based on decisions where to end the first shelf: for each i so that $\sum_{1 \leq j \leq i} w_j \leq W$, we solve the problem l_{i+1}, \dots, l_n recursively (the sub-problem if we end the line at word i) and add the penalty for the first shelf $\max_{1 \leq j \leq i} h_j$ to the total penalty for the recursive problem. We take the minimum such total penalty. Note that we only solve suffixes of the original sequence, so this is a prime dynamic programming possibility. The following just computes the minimum penalty; it is easy to modify to keep track of how to achieve that penalty.

DPShelves(W : positive integer; *Books*[1.. n]: array of pairs of integers)

1. Initialize *Penalty*[1.. $n+1$]
2. IF *Book*[i].width $> W$ for any i return "infinite"
3. *Penalty*[$n + 1$] $\leftarrow 0$; (no penalty for an empty set of books)
4. For $J = n$ downto 1 do:
5. begin;
6. *Last* $\leftarrow J$
7. *Sum* \leftarrow *Book*[*Last*].width
8. *ShelfHeight* \leftarrow *Book*[*Last*].height
9. *best* = "infinity"

10. Until $Last > n$ or $Sum > W$ do:
 11. $best \leftarrow MIN(best, ShelfHeight + Penalty[Last + 1])$
 12. $Last \leftarrow Last + 1;$
 13. $Sum \leftarrow Sum + Book[Last].width$
 14. $ShelfHeight \leftarrow max(ShelfHeight, Book[Last].height)$
 15. $Penalty[j] \leftarrow best;$
 16. end;
 17. return $Penalty[1]$

The time of the until is at most the smaller of W and n . So the total time is the minimum of $O(n^2)$ and $O(Wn)$.

Protein Bonding : 20 pts Let Σ be a finite set of amino acids, and let $w = w_1 \dots w_n$ be a sequence of acids from Σ . For $\sigma, \sigma' \in \Sigma$, let $b(\sigma, \sigma')$ be the strength of a bond between the two types of acids, a non-negative real number. A *bonding* of the sequence is a partial matching between positions in the word so that matched pairs can be connected with lines drawn below the word without lines crossing. Equivalently, it should satisfy : there are no two bonded pairs i_1, j_1 and i_2, j_2 with $i_1 \leq i_2 \leq j_1 \leq j_2$. The *total bond strength* is the sum over all bonded positions i, j of the bond strength $b(w_i, w_j)$. Give as efficient as possible algorithm to find the bonding of a proteing sequence that maximizes the total bond strength. (We know an $O(n^3)$ algorithm.)

Identify a bonding with the set of pairs of positions matched, each pair listed in increasing order. Let B be a bonding of $w_1 \dots w_n$. If B does not match position 1 with anything, it is also a bonding on positions 2, ..., n . Conversely, any bonding on positions 2... n is also a bonding on 1... n

If B matches position 1 with position i , then every match (j, k) in B must either have $2 \leq j < k \leq i - 1$ or $i + 1 \leq j < k \leq n$, since if $j < i < k$, we have $1 \leq j < i < k$, which violates our constraints. Let B_{inside} be the matching on 2.. $i - 1$ of edges in B of the first type, and $B_{outside}$ of the second type. B_{inside} is a bonding on 2... $i - 1$, and $B_{outside}$ is a bonding on $i + 1..n$, and $Totalstrength(B) = Totalstrength(B_{inside}) + Totalstrength(B_{outside}) + strength(w_1, w_i)$. Conversely, let B_I be any bonding on 2.. $i - 1$, and B_O any bonding on $i + 1,..n$. Then $B = (1, i) \cup B_O \cup B_I$ is a bonding on 1... i of total strength $Totalstrenght(B_O) + Totalstrength(B_I) + strength(w_1, w_i)$.

This gives us the following recursive backtracking algorithm:

$TS(w_1 \dots w_n \in \Sigma^n, strength[\Sigma][\Sigma]$: array of non-negative real numbers): non-negative real number.

1. IF $n \leq 1$ return 0; {need two positions to bond }

2. $Non - bond \leftarrow TS(w_2...w_n, strength)$.
3. $Best \leftarrow Non - bond$.
4. FOR $i = 2$ to n do:
5. begin;
6. $CurrentCase \leftarrow TS(w_2...w_{i-1}, strength) + TS(w_{i+1}...w_n) + strength(w_1, w_i)$
7. IF $Best < CurrentCase$ THEN $Best \leftarrow CurrentCase$.
8. end;
9. Return Best.

Then we note that this recursive procedure never changes *strength* and only calls itself on consecutive subsequences $w_i...w_j$ of the original sequence. The value of i is always greater in a sub-call, so bottom-up can be: in decreasing order of i . This allows us to create the following dp version of the above recursion: $DPTS(w_1...w_n \in \Sigma^n, strength[\Sigma][\Sigma])$: array of non-negative real numbers): non-negative real number.

1. Initialize array $total[1..n][1..n]$ to 0's. {includes base case}
2. FOR $I = n - 1$ to 1 do :
3. FOR $J = I + 1$ to n do:
4. begin;{compute strength for $w_I...w_J$ }
5. $Non - bond \leftarrow total(I + 1, J)$
6. $Best \leftarrow Non - bond$.
7. FOR $K = I + 1$ to J do:
8. begin;
9. $Curr \leftarrow total(I + 1, K - 1) + total(K + 1, J) + strength(w_I, w_K)$
10. IF $Best < Curr$ THEN $Best \leftarrow Curr$.
11. end;
12. $total[I, J] \leftarrow Best$
13. end;
14. Return $total[1][n]$.

The total time is $O(n^3)$ since we have three nested loops, each with up to n iterations.