Theory of Algorithms Homework 5

Evan Dreher

October 2022

1 Problem 1

Show the unsorted buckets that result from the following problem in bucket sort.

$$\{0.6, 0.5, 0.24, 0.8, 0.25, 0.73, 0.1, 0.26\}$$

Bucket	0	1	2	3	4	5	6	7
	0.1	0.24	0.26		0.5	0.73	0.8	
			0.25		0.6			

2 Problem 2

Show the dynamic programming table that results for the following rebar-cutting problem out to length 10.

Length	1	2	3	4	5	6	7	8	9	10
Value	4	9	11	16	21	28	30	33	34	–
Best	4	9	13	18	22	28	32	37	41	46

3 Problem 3

Construct an example of the rebar-cutting problem to show that the following greedy strategy is not necessarily optimal. "Always choose the cut with the best price-per-inch ratio (that's no longer than your remaining piece)."

Consider the following for rebar-cutting,

Length	1	2	3
Value	2	7	10
Value/foot	2	3.5	3.33
Greedy Best	2	7	9

To calculate the best for a rebar of length 3 it sees that the value/foot of a 2 length bar is better than that of a 3 length bar so it makes a cut for a total of 9$ (the price of 1 bar of length 2 and 1 each). This is clearly not the optimal case since a bar of length 3 is worth 10$ when no cuts are made, thus this greedy approach is not optimal.

4 Problem 4

Draw the Levenshtein distance table for the strings CGATC and GATT.

–	–	G	A	T	T
–	0	1	2	3	4
C	1	1	2	3	4
G	2	1	2	3	4
A	3	2	1	2	3
T	4	3	2	1	2
C	5	4	3	2	2

5 Problem 5

Revisit the table from problem 2. Imagine that each cut you make costs 1$. State a revised dynamic programming calculation that will enable you to solve this problem and show the solution for the problems out to size 10.

Length	1	2	3	4	5	6	7	8	9	10
Value	4	9	11	16	21	28	30	33	34	–
Best	4	9	12	17	21	28	31	36	39	44

Let $B(i)$ denote the most money you can get from a piece of rebar of length i.

$$B(i) = max(V[i], B(i-1) + B(1) - 1, B(i-2) + B(2) - 1, ...B(\lceil \frac{i}{2} \rceil) + B(\lfloor \frac{i}{2} \rfloor) - 1)$$

- $B(1) = max(V[1]) = max(4) = 4$
- $B(2) = max(V[2], B(1) + B(1) - 1) = max(9, 8) = 9$
- $B(3) = max(V[3], B(2) + B(1) - 1) = max(11, 12) = 12$
- $B(4) = max(V[4], B(3) + B(1) - 1, B(2) + B(2) - 1) = max(16, 15, 17) = 17$
- $B(5) = max(V[5], B(4) + B(1) - 1, B(3) + B(2) - 1) = max(21, 20, 20) = 20$
- $B(6) = max(V[6], B(5)+B(1)-1, B(4)+B(2)-1, B(3)+B(3)-1) = max(28, 24, 25, 23) = 28$
- $B(7) = max(V[7], B(6)+B(1)-1, B(5)+B(2)-1, B(4)+B(3)-1) = max(30, 31, 29, 28) = 31$
- $B(8) = max(V[8], B(7) + B(1) - 1, B(6) + B(2) - 1, B(5) + B(3) - 1, B(4) + B(4) - 1) = max(33, 34, 36, 32, 33) = 36$
- $B(9) = max(V[9], B(8) + B(1) - 1, B(7) + B(2) - 1, B(6) + B(3) - 1, B(5) + B(4) - 1) = max(34, 39, 39, 39, 37) = 39$
- $B(10) = max(B(9)+B(1)-1, B(8)+B(2)-1, B(7)+B(3)-1, B(6)+B(4)-1, B(5)+B(5)-1 = max(42, 44, 42, 44, 41) = 44$

6 Problem 6

Revisit the table from problem 2 again. Imagine now that you can make a total of only k cuts, at maximum. You decide to solve this problem similarly to the approach we discussed in class: by making a separate dynamic row for each value of "maximum value using up to i cuts," with i ranging from 0 to k. (The 0 row, of course, will be just the Value row over again.) Write a short pseudocode algorithm for your approach and show the result for problem 2 out to length 10, assuming $k = 3$. Comment briefly on the time requirements of your algorithm.

Length	1	2	3	4	5	6	7	8	9	10
Value	4	9	11	16	21	28	30	33	34	–
0 cuts	4	9	11	16	21	28	30	33	34	–
1 cuts	4	9	13	18	21	28	32	37	39	44
2 cuts	4	9	13	18	22	28	32	37	41	46
3 cuts	4	9	13	18	22	28	32	37	41	46

Assumptions

1. "*Best*" is a 2D array of memoized calls to the functions.
2. "Best[i][j]" indicates the most money you can get for a piece of length j using no more than i cuts.
3. If finding the most money you can get for a piece of length j with no more than i cuts has not yet been solved then $Best[i][j] = 0$.
4. The 0th row of $Best$ (i.e the one where we aren't allowed to make any cuts) has been initialized such that $Best[0][i] = Value[i]$.
5. Division will be Java's integer division (i.e rounded down).

Algorithm 1 Rebar-Cutting(length, cuts)

if `Best[cuts][length] does not equal 0` **then**
  return $Best[cuts][length]$
if `cuts equals 0` **then**
  return $Best[0][length]$
$possibleBests = newArray$
`add to` $possiblebests$
$possibleBests.add(Rebar - Cutting(length, cuts - 1)$
for `i=1 to length - 1` **do**
  $possibleBests.add(RebarCutting(i, \frac{cuts}{2}) + RebarCutting(length - 1 - i, \frac{cuts-1}{2})$
$Best[cuts][length] = max(possibleBests)$
return $Best[cuts][length]$

7 Problem 7

Suppose that you have a collection of d dice, each of which has f faces (numbered 1-f), and you're interested in rolling dice so that they total up to some number n. You're interested in designing a dynamic programming problem to answer the question, "How many ways can I roll d dice with f faces so that they total to n?" Call this problem Roll(d,f,n).

(a) Write a recursive solution for Roll(d,f,n); that is, write an expression "Roll(d,f,n) = ..." that defines the value in terms of its sub-problems. At least some of your variables will change values in the sub-problems; they may not *all* change.

for `i = n-f-1 to n-1` **do**
    $Roll(d-1, f, i)$

(b) At some point, of course, the problem would stop recursing. What are your base cases? How many are there? What value would they have?

- If $d = n$ return 1.
- If $d = 1$ and $n \leq f$ return 1.
- If $n < d$ return 0.

(c) Now picture solving this in a bottom-up, dynamic-programming way. How many sub-problems would there be? How much work would it be to solve one sub-problem? How much total work would that be?

To solve Roll(d,f,n) you have to solve every problem from [1][1] to [d][n] which is a total od $d \times n$ problems. Each problem sums f entries for a total of f work. Thus, the total work is $d \times n \times f$.

(d) Following your formulas above, show the table for Roll(3,4,8).

In this 2D array, the row indicates the number of dice being rolled, and the column is the sum. Thus [2][3] = 2 means there are 2 ways to roll 2 dice such that they sum to 3.

Dice/Sum	1	2	3	4	5	6	7	8
1	1	1	1	1	0	0	0	0
2	0	1	2	3	4	3	2	1
3	0	0	1	3	6	10	12	12