Nalongsone Danddank Student ID : 14958950 StarID: jf3893pd

Email: [nalongsone.danddank@my.metrostate.edu](mailto:nalongsone.danddank@my.metrostate.edu)\

Homework 3 (15 pts)[[1]](#footnote-0)

1. [9 pts]: Suppose that you have *n* dice (normal, six-sided dice with sides labeled 1, 2, 3, 4, 5, 6). Calculate the number of ways to get the dice to sum to *d* – that is, to show a total of *s* dots - when you roll them all at once.

Part A: [3 pts] Suppose numDots(n,d) is the number of ways to show *d* dots when you show *n* dice. Write a recurrence that expresses the number of ways to show *n* dots on *d* dice as a function of the number of dots shown on *n­*-1 dice. For the purpose of this equation, you can assume that d > 6. Noe that as in the Edit Distance problem, this will have multiple cases.

***Solution:***

Step 1: Characterize the structure of an optimal solution:

As we have n dice with six face [1,2,3,4,5,6], and we want to calculate the number of ways to get the dice to sum to d.

First we take a simple way Like:

If we have n = 1 with target d = 6, numDots(1,6); we have result is 1 way is (6)

If we have n = 2 dice with target d = 7, numDots(2,7) :

We have possible of result like (1+6) (2+5) (3+4) (4+3) (5+2) (6+1) total 6 ways of sum target d = 7, numDots(2,7) = 6 and so on;

However when the target is d < 6, there will 0 way of sum. sumDots(n, 5) = 0.

Step 2: Recursive solution:

1. Base case is sumDots(1,6) = 1 when n = 1;
2. The total of number of ways to sum of the dice of possible is:

Recusive for n, n-1, n-2… 1 dice:

sumDots(n, d) = sumDots(n-1, d-1) + sumDots(n-1, d-2)+…+ sumDots(n-1, d-6);

On the other hand, we can draw the tree graph like below:

sumDots(n,d)

sumDots(n-1,d-6)

sumDots(n-1,d-5)

sumDots(n-1,d-4)

sumDots(n-1,d-3)

sumDots(n-1,d-2)

sumDots(n-1,d-1)

Part B: : [3 pts] Develop your recurrence into pseudocode. Note that this is pseudocode, it doesn’t need to be syntactically correct Java or any other language.

***Solution:***

Recurrence Pseudocode by Java syntactic:

/\* @params: n - int - number of dice with 6 sides 1 to 6 d - int - target of the

\* number of sum

\* @return: int - the number of ways to get sum 'd' with 'n' \*/

**int** sumDots(**int** n, **int** d) {

// set the base case for recursive.

**if** (n == 0) {

// when n == 0 and d == 0 return 1 to method

**if** (d == 0) **return** 1;

// when only n==0 return 0 to method

**return** 0;

}

// initialize 0 to target sum of the ways we need to return

**int** s = 0;

**for** (**int** i = 1; i <= 6; i++)

// sum up the number of way that we need.

s += *sumDots*(n - 1, d - i);

**return** s;

}

Computing the optional costs:

As we see the graph on the Part A, that is a tree graph each parent have 6 children because each dice has six-sided 1 to 6. Then the high level of tree depend on the number of dice, n. In addition, we can see the pseudocode on the Part B that the recursive is inside the for loop of length 6 (i : 1 to 6).

So the Time complexity should be exponential O(6n)

Part C: : [3 pts] Modify the pseudocode to use bottom-up dynamic programming so that you don’t make duplicate calls to handle overlapping subproblems.

***Solution:***

Up on the Part B we see that time complexity is too large O(6n). So we need modify the pseudocode to use botton-up dynamic programming by create the 2D array to remember each solution of number of the ways sum of dice dots, and we don’t have to make duplicate calls to handle overlapping subproblems like the pseudocode of recusive above.

/\*@params: n - int - number of dice with 6 sides 1 to 6 d - int - target of the

\* number of sum. @return: int - the number of ways to get sum 'd' with 'n' \*/

**int** sumDots\_bottomUp(**int** n, **int** d) {

// for preventing the target sum below then we need.

**if** (d > 6 \* n) **return** 0;

// initialize a Table to save the previous subproblems.

**int**[][] diceTable = **new** **int**[n + 1][d + 1];

diceTable[0][0] = 1;

**for** (**int** i = 1; i <= n; i++) {

**for** (**int** j = 0; j <= d; j++) {

**for** (**int** k = 1; k <= 6; k++) {

// [j-k] needs to be in the range of Table and big then 0

**if** (j - k < 0) **continue**;

diceTable[i][j] += diceTable[i - 1][j - k];

}

}

}

**return** diceTable[n][d];

}

Computing the optional costs: the time complexity is Θ(6\*n\*d),

so we can say that time complexity is a polynomial O(n2)

1. [4 pts]: Suppose that you have a set of jobs you can do for pay. Assume that you work *h* hours per day, and each job *j* takes time *tj* and gives you revenue *rj*. Your goal is to maximize the amount of revenue that you can earn in one day.

Write the recurrence that describes the maximum possible revenue recursively. Call the method maxrev(J), where J is the set of jobs J = {j1, j2, …, jn}.

***Solution:***

Assume that working h hours per day so h = t1 + t2 + t3 +… tn.

Then, J = {j1, j2, …, jn}

Example:

j1 => t1 = 1.2, => r1 = $20

j2 => t2 = 2.2, => r2 = $34

…

jn => tn = h, rn = $

So the recurrence should be,

maxrev(J[n]):

if n == 0 :

Return 0;

r\_max = - infinity;

for i = 1 to n :

r\_max = max(r\_max, J[i] + maxrev(J[n-I]));

return r\_max;

The time complexity is O(2n)[4 pts] Given a text consisting entirely of letters with no punctuation or spacing, divide that text into words such that there are no extraneous letters left over. For instance, this is Lincoln’s Gettysburg address with all spaces and punctuation and capitalization removed.

fourscoreandsevenyearsagoourfathersbroughtforthonthiscontinentanewnationconceivedinlibertyanddedicatedtothepropositionthatallmenarecreatedequalnowweareengagedinagreatcivilwartestingwhetherthatnationoranynationsoconceivedandsodedicatedcanlongendurewearemetonagreatbattlefieldofthatwarwehavecometodedicateaportionofthatfieldasafinalrestingplaceforthosewhoheregavetheirlivesthatthatnationmightliveitisaltogetherfittingandproperthatweshoulddothisbutinalargersensewecannotdedicatewecannotconsecratewecannothallowthisgroundthebravemenlivinganddeadwhostruggledherehaveconsecrateditfaraboveourpoorpowertoaddordetracttheworldwilllittlenotenorlongrememberwhatwesayherebutitcanneverforgetwhattheydidhereitisforusthelivingrathertobededicatedheretotheunfinishedworkwhichtheywhofoughtherehavethusfarsonoblyadvanceditisratherforustobeherededicatedtothegreattaskremainingbeforeusthatfromthesehonoreddeadwetakeincreaseddevotiontothatcauseforwhichtheygavethelastfullmeasureofdevotionthatweherehighlyresolvethatthesedeadshallnothavediedinvainthatthisnationundergodshallhaveanewbirthoffreedomandthatgovernmentofthepeoplebythepeopleforthepeopleshallnotperishfromtheearth

Converting this into real English text is a good job for a computer. But it’s not quite as simple as saying “find the first word, set it off, then continue”. If you tried that with the Gettysburg address, you’d have a problem because you would start out:

fourscoreandseven 🡪 four scoreandseven 🡪 four score andseven 🡪 four score an dseven

But although “an” is a word, it’s not the *right* word for this text (which is “and”). The problem is that some words are prefixes of other words. There is no word that will start with “dseven…”. One way to handle this is to use dynamic programming.

Suppose that L[1…n] is an array of letters. We want to divide these letters up into an array of words W (the size of the array isn’t known in advance) so that each element of the array of words is an English language word, and there are no extraneous letters.

Define variables and write the recursive formula that uses dynamic programming to figure out an array W such that each element of W is an English language word, with no letters left over. Assume that you have a Boolean function isWord() available to you that will tell if you any given string of letters is or is not a word (but you won’t need this as part of your formulation). You also know that the longest word in the dictionary has length *k* (it’s dictionary-dependent).

You can express this as an equation, with pseudocode, or in English Language prose, whichever you prefer.

***Solution:***

Assume that we have English dictionary and we have function isWord() to test characters.

String text = …// (Lincoln’s Gettysburg address text with no space);

String result = “”; // empty string to get the result.

textToWords(text, n, result): // n is length of text

for i = 1 to n: // loop all the text to find the right words.

string prefix = text [0 : i];

If isWord(prefix) is true:

result += prefix;

return; // solve the subproblem then return.

// then run the recursive until all correct.

textToWords(text[i : n - i], n - i, (result + prefix + “ “));

The time complexity is O(nn).

1. Note that 17 points are possible, so basically this homework includes a little extra credit. [↑](#footnote-ref-0)