CSE 565: Algorithm Design and Analysis **Due 2022-10-21T23:59**

Problem Set 3 Solutions

Instructor: Dr. Antonio Blanca *Release Date:* 2022-10-07
TA: Jeremy Huang

Notice: Type your answers using LaTeX and make sure to upload the answer file on Gradescope before the deadline. Recall that for any problem or part of a problem, you can use the "I'll take 20%" option. For more details and the instructions read the syllabus.

Problem 1. Too Different

Recall in class we use dynamic programming to calculate the edit distance between two strings. Given two strings A and B of the same length n, and a threshold $k, 0<k<n$, design an algorithm to check whether the edit distance between A and B is less than k. Your algorithm should run in $O(nk)$ time.

Solution

Instead of calculating the value for all the cells in the $n \times n$ dynamic programming matrix M, here, we only need to calculate the value in the diagonal with the width of $2k-1$ using exactly the same recursion we introduced during lecture. This is because cells more than k cells away from the main (i,i) diagonal must have values greater than k because they involve a minimum of k insertions/deletions. So we can treat all cells outside that digonal as having value ∞. Therefore by the same proof from lecture, $M[n,n]<k$ if and only if the edit distance between A and B is less than k. We can actualize this while only using $O(nk)$ space and time by using a default value hash table to store the cells, and by calculating cells in the following order: (i,i) first, then $(i,i+1)$ through $(i,i+k)$, and finally $(i+1,i)$ through $(i+k,i)$ for i from 0 to $n-1$.

Problem 2. LCS

Let $A=a_1a_2\cdots a_n$ and $B=b_1b_2\cdots b_m$ be two strings. Design a dynamic programming algorithm to compute the longest common subsequence between A and B, i.e., to find the largest k and indices $1 \leq i_1<i_2<\cdots<i_k \leq n$ and $1 \leq j_1<j_2<\cdots<j_k \leq m$ such that $A\left[i_1\right]=B\left[j_1\right], A\left[i_2\right]=B\left[j_2\right], \cdots, A\left[i_k\right]=B\left[j_k\right]$. Your algorithm should run in $O(\mathrm{mn})$ time.

Solution

Define $\mathrm{LCS}[i,j]$ as the length of the longest common subsequences between $a_1a_2\cdots a_i$ and $b_1b_2\cdots b_j$. We have the recursion formula below:

$$LCS[i,j]=\begin{cases}\mathrm{LCSS}[i-1,j-1]+1 & \text{if} \quad a_i=b_j \\ \max\{LCS[i-1,j], LCS[i,j-1]\} & \text{if} \quad a_i \neq b_j\end{cases}$$

The pseudo-code, which includes three steps, is as follows.

Initialization:

$$\mathrm{LCS}[i,0]=0, \text{ for all } i \text{ in } 1,2,\ldots n$$
$$LCS[0,j]=0, \text{ for all } j \text{ in } 1,2,\ldots m$$

Iteration:

```
for $i = 1 \cdots n$ :
    for $j = 1 \cdots m$ :
        if $a_i = b_j$: $LCS[i,j] = LCS[i-1,j-1]+1$
        else: $LCS[i,j] = max\{LCS[i-1,j], LCS[i,j-1]\}$
    endfor
endfor
```

Termination: $LCS[n,m]$ gives the length of the longest common subsequences between A and B. (The corresponding indices can be fetched through introducing backtracing pointers.)

Running time: The initialization of $LCS[i,j]$ takes $O(m+n)$ time. The iteration step have two embedded for loops, and therefore takes $O(mn)$ time. The total running time of the algorithm is $O(mn)$.

Problem 3. Mind the Gap

In lecture, we learned an $O(\text{mn})$ dynamic programming (DP) algorithm to compute edit distance of two strings, aka sequence alignment. Similar algorithms are widely used in bioinformatics where scientists often compare similarity of two given DNA sequences.

Here you are required to design a DP algorithm to compute the edit penalty of the optimal alignment of two given strings, but with a higher gap penalty. The biological implication is that mutations (i.e. mismatches) happens much more frequently than insertions/deletions (i.e. gaps) in a DNA sequence. Also, extending a gap (i.e. a longer gap) is more favored than opening a new gap (i.e. two short gaps). More specifically, in the edit distance problem, the penalty (i.e., the edit distance) is incremented by 1 for a mismatch or a gap between S_1 and S_2. In this problem with refined gap model, the penalty is still incremented by 1 for a mismatch, but by a when opening a new gap, and by b when extending an existing gap ($a > b \geq 3$). For example, the total penalty for alignment AA_CC $\times$ AAGGCT is $(a+b+1)$ where a is for opening the gap at position 3, b for extending the gap at position 4 , and 1 for a mismatch at position 6 .

Problem: Sequence alignment with gap penalty.

Input: Two strings $S_1[1\cdots n]$ and $S_2[1\cdots m]$ over alphabet $\Sigma = \{A,C,G,T\}$, and a,b with $a > b \geq 3$.

Output: the alignment between S_1 and S_2 with minimized total penalty.

Design a dynamic programming algorithm for the above problem. Show correctness and complexity analysis of your algorithm. Your algorithm should run in $O(\text{mn})$ time.

Hint: Use three DP tables. One for ending with a gap in S_1, one for ending with a gap in S_2, one for match/mismatch only.

Solution

Define $F(i,j)$ as the edit distance between $S_1[1\cdots i]$ and $S_2[1\cdots j]$ ending with a match or mismatch at $S_1[i]$ and $S_2[j]$.

Define $X(i,j)$ as the edit distance between $S_1[1\cdots i]$ and $S_2[1\cdots j]$ ending with a gap at S_1.

Define $Y(i,j)$ as the edit distance between $S_1[1\cdots i]$ and $S_2[1\cdots j]$ ending with a gap at S_2.

Define mismatch penalty function $\delta(x,y) = \begin{cases} 0, x = y \\ 1, x \neq y \end{cases}$.

Hence, we have the following recurrence:

$$F[i,j] = \delta\left(S_1[i], S_2[j]\right) + \min \begin{cases} F[i-1,j-1] \\ X[i-1,j-1] \\ Y[i-1,j-1] \end{cases}$$

$$X[i,j] = \min \begin{cases} F[i,j-1]+a & // \text{ opening a gap} \\ X[i,j-1]+b & // \text{ extending a gap} \\ Y[i,j-1]+a & // \text{ actually won't use} \end{cases}$$

$$Y[i,j] = \min \begin{cases} F[i-1,j]+a & //\text{opening a gap} \\ X[i-1,j]+a & //\text{actually won't use} \\ Y[i-1,j]+b & //\text{extending a gap} \end{cases}$$

Use P_f, P_x, P_y to store the traceback directions to get F, X, Y respectively.

```
function Alignment-affine-penalty ($S_1[1 \cdots n], S_2[1 \cdots m]$)
    initiate array $F, X, Y, P_f, P_x, P_y$ of size $(n+1) \times (m+1)$
    $F(0,0) = 0, X(0,0) = 0, Y(0,0) = 0$
    $F(i,0) = -\infty, X(i,0) = -\infty, Y(i,0) = a + (i-1) \times b, 1 \le i \le n$
    $F(0,j) = -\infty, X(0,j) = a + (j-1) \times b, Y(0,j) = -\infty, 1 \le j \le m$
    $P_f[i,0] = -1, P_x[i,0] = -1, P_y[i,0] = -1, 0 \le i \le n$
    $P_f[0,j] = -1, P_x[0,j] = -1, P_y[0,j] = -1, 0 \le j \le m$
    for $i = 1$ to $n$
        for $j = 1$ to $m$
            $k = min \begin{cases} F[i-1,j-1] \\ X[i-1,j-1] \\ Y[i-1,j-1] \end{cases}$
            $F[i,j] = \delta(S_1[i], S_2[j]) + k$
            $X[i,j] = min \begin{cases} F[i,j-1]+a \\ X[i,j-1]+b \\ Y[i,j-1]+a \end{cases}$
            $Y[i,j] = min \begin{cases} F[i-1,j]+a \\ X[i-1,j]+a \\ Y[i-1,j]+b \end{cases}$
            $P_f[i,j] = \begin{cases} 1, & \text{if } F[i-1,j-1] = k \\ 2, & \text{if } X[i-1,j-1] = k \\ 3, & \text{if } Y[i-1,j-1] = k \end{cases}$
            $P_x[i,j] = \begin{cases} 1, & \text{if } X[i,j] = F[i,j-1]+a \\ 2, & \text{if } X[i,j] = X[i,j-1]+b \\ 3, & \text{if } X[i,j] = Y[i,j-1]+a \end{cases}$
            $P_y[i,j] = \begin{cases} 1, & \text{if } Y[i,j] = F[i-1,j]+a \\ 2, & \text{if } Y[i,j] = X[i-1,j]+a \\ 3, & \text{if } Y[i,j] = Y[i-1,j]+b \end{cases}$
        End for
    End for
```

```
// trace back using P
        $max\_score = max(F[n,m], X[n,m], Y[n,m])$
        $P' = \begin{cases} P_f, \text{ if } max\_score = F[n,m] \\ P_x, \text{ if } max\_score = X[n,m] \\ P_y, \text{ if } max\_score = Y[n,m] \end{cases}$
        $p_0 = P'[n,m]$
        $i = n,\ j = m$
        // make alignment from end to start
        while $p_0 ! = -1$
            if $P' = P_f$: consume $S_1[i]$ and $S_2[j]$ for alignment, $i--$, $j--$
            if $P' = P_x$: consume a gap in $S_1$ and $S_2[j]$ for alignment, $j--$
            if $P' = P_y$: consume $S_1[i]$ and a gap in $S_2$ for alignment, $i--$
            $P' = \begin{cases} P_f, \text{ if } p_o = 1 \\ P_x, \text{ if } p_o = 2 \\ P_y, \text{ if } p_o = 3 \end{cases}$
            $p_0 = P'[i,j]$
        End while
        if $i \neq 0$ or $j \neq 0$: consume any remaining chars of $S_1$ or $S_2$ to align with gaps.
    End **function**
```

Correctness: To fill in F, X, and Y, they should be build upon the minimum of previous penalty plus the new penalty. For F the new penalty is $\delta(S_1[i], S_2[j])$. For X, the penalty is a for using optimal cases from F and Y, and the penalty is b for using optimal cases from X. Since the gap penalty is affine penalty (a linear function $a + (x-1) \times b$), we can safely use $F[i-1, j]$ and $X[i-1, j]$ for computing $X[i, j]$ without considering the length of the gap. (If the penalty function is a general one, for example log function, we have to consider the length of the gap and consequently the complexity becomes $O(n^3)$). The traceback step works similar to that when using a linear penalty, but we need to switch between three DP tables. Then the alignment can be constructed from end to the start.

Kudos: Why you actually will never compute $X[i, j]$ directly from $Y[,]$ or vice versa? Recall that it is an affine penalty where opening a gap costs a and extending a gap costs $b(a > b \geq 3)$. Suppose you compute $X[i, j]$ from $Y[i, j-1]$. In other words, the last position of the alignment is $S_2[j]$ and a gap in S_1 while the 2nd last position is $S_1[i]$ and a gap in S_2 (recall it is computed from Y meaning you end a gap in S_2). For example, XXXXAG_ $\times$ XXXXA_C. You can always eliminate the two adjacent gaps in opposite strings by making them a mismatch, because its penalty is always less than two gaps, i.e. $1 < a + b$ or $1 < a + a$. For example, XXXXAG_ $\times$ XXXXA_C $\Rightarrow$ XXXXAG $\times$ XXXXAC. So in practice, you will not have an alignment which has two adjacent gaps in opposite strings.

Complexity: All tables F, X, Y, P_f, P_x, P_y are of size $(n+1) \times (m+1)$, and filling them takes constant time, so filling all the tables take $O(mn)$ time.

In the traceback procedure, at least one of i or j is decremented by 1 , so in total the while loop can have at most $(m+n+1)$ iterations. Each iteration takes constant time.

Hence the total time complexity is $O(mn)$.

Problem 4. Line Fill

Consider a typesetting software (such as TEX) that converts a paragraph of text into PDF documents (say). The input text is a sequence of n words of width $w_1, \ldots, w_n$. The software needs to somehow minimize the amount of extra spaces, as follows. Every line has width W, and if it contains words i through j, then amount of extra spaces on this line is $W - j + i - \sum_{i \leq k \leq j} w_k$, because we leave one unit of space between words. The amount of extra space on each line must be nonnegative to avoid overflowing of words. Our goal is to minimize the sum of squares of extra spaces on all lines except the last. Give a dynamic programming algorithm for this problem that runs in $\mathcal{O}(n^2)$. Correctness proof is optional for this question.

Solution

We first solve a variant of the problem, where we have to typeset the first k words $w_1, \dots, w_k$ from the list, and the goal is to minimize the sum of squares of extra spaces on all lines (including the last). This variant is easier to solve, since we need not worry about not discounting the last line. Let $f(k)$ be the optimal answer to this new problem.

When typesetting word k, we need to decide what other words go with it on the same line. Let j be the first word lying on the same line as word k, and let $S(j,k) = W - k + j - \sum_{i=j}^{k} w_i$ be the amount of extra spaces on this line. This leads to the recurrence

$$f(k) = \min_{1 \le j \le k} \{ f(j-1) + S(j,k)^2 \mid S(j,k) \ge 0 \}.$$

The base case is $f(0) = 0$, since typesetting no words cost us nothing.

Finally, the answer to the original problem (where the last line is not counted) is

$$\min_{1 \le j \le n} \{ f(j-1) \mid S(j,n) \ge 0 \}.$$

In resolving the recurrence, we need to repeatedly compute consecutive width $\sum_{i=j}^{k} w_i$ from words i through j. One way to speed up the algorithm is to precompute all the cumulative widths $U(k) = \sum_{i=1}^{k} w_i$ in $O(n)$ time. Given $U(\cdot)$, we can compute any consecutive width $\sum_{i=j}^{k} w_i$ as $U(k) - U(j-1)$ in $O(1)$ time. Therefore the algorithm can be implemented in $\mathcal{O}(n^2)$ time.

Problem 5. Re-pea-pea-pea-pea-peat

Given a string S of length n and a positive integer k, find the longest continuous repeat of a substring which length is k. For example, when $S =$ ACGACGCGCGACG, if $k = 2$, the longest continuous repeat would be *CGCGCG* composed of *CG* repeating 3 times. And if $k = 3$, the longest continuous repeat would be *ACGACG* composed of *ACG* repeating twice. Design a dynamic programming algorithm to solve this problem and analyze its running time.

Solution

Define $f[i]$ as the length of repeating character at position i with distance k. For example, $f[i] = 3$ represents $f[i] = f[i-k] = f[i-2k] \neq f[i-3k]$.

We have the recursion formula below:

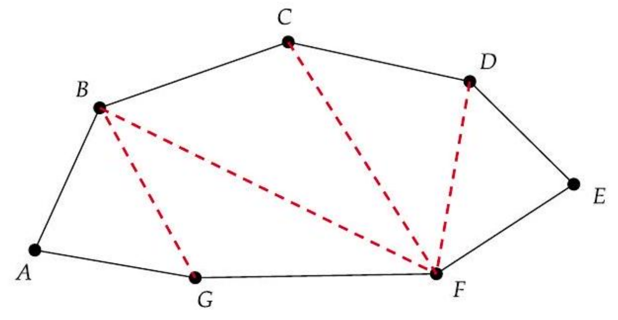
$$f[i] = \begin{cases} f[i-k]+1 & \text{if} \quad S[i] = S[i-k] \\ 1 & \text{if} \quad S[i] \neq S[i-k] \end{cases}$$

After computing $f[i]$ for every position in the array, we can use a sliding window to get the answer. Since for any substring $S[i, i+1, \cdots i+k-1]$ of length k, the repeating time of it is $\min(f[i], f[i+1], f[i+2] \cdots, f[i+\ k-1])$. Therefore we only need to use a sliding window of length k moving from beginning of the string to the end. Every time we find the minimum value of $f[i]$ for all the indices inside the sliding window. And the maximum value of all the values we get from each sliding window will be our answer. The running time is $O(nk)$.

Problem 6. Short Poly

You have a wood plank in a shape of a convex polygon with n edges, represented as the (circular) list of the 2D coordinates of its n vertices. You want to cut it into $(n-2)$ triangles using $(n-3)$ non-crossing diagonals, such that the total length of the cutting trace (i.e., total length of the picked diagonals) is minimized. Design a dynamic programming algorithm runs in $O\left(n^3\right)$ time to solve this problem. Your solution should include definition of subproblems, a recursion, how to calculate the minimized total length (i.e., the termination step), and an explanation of why your algorithm runs in $O\left(n^3\right)$ time.

For example, in the convex polygon given below, one possible way to cut it into triangles is illustrated with red dashed lines, and in this case the total length of cutting trace is $\overline{BG}+\overline{BF}+\overline{CF}+\overline{DF}$.



Solution

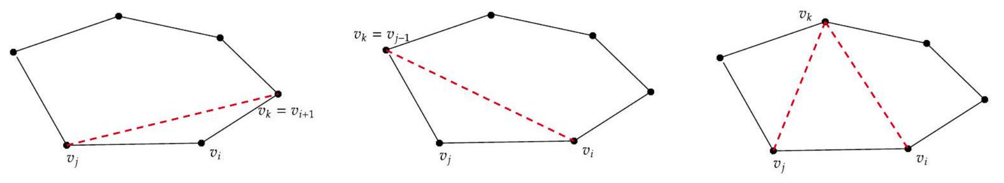
Note: this problem requires defining subproblems over intervals.

1. Subproblem: consider the convex polygon from vertex v_i to vertex v_j following couter-closewise order closed by a (possibly virtual) edge (v_j, v_i); define $M[i,j]$ as the minimized total cutting trace of this polygon. Notice that the subproblems we defined span all intervals of the given polygon in a circular manner; it could be that $i > j$.

2. Recursion. Consider the last edge (v_j, v_i) of above defined polygon: this edge must be in a triangle, and we enumerate the other vertex of this triange. Assume it's v_k. If v_k is next to v_i, i.e., $k = i+1$ (in circular manner, i.e., $\bmod n$), then we only need to cut $\overline{v_k v_j}$, and this leaves a smaller polygon from v_k to v_j. If v_k is next to v_j, i.e., $k = j-1$ (in circular manner, i.e., $\bmod n$), then we only need to cut $\overline{v_k v_i}$, and this leaves a smaller polygon from v_i to v_k. If v_k is not next to either one i.e., $i+1 < k < j-1$ (in circular manner, i.e., $\bmod n$), then we need to cut both $\overline{v_k v_i}$ and $\overline{v_k v_j}$, and this leaves two smaller polygons, one from v_i to v_k and one from v_k to v_j. See figure below. Formally,

$$M[i,j] = \min \begin{cases} \overline{v_{i+1} v_j} + M[i+1,j] \\ \overline{v_i v_{j-1}} + M[i,j-1] \\ \min_{i+1<k<j-1} \left(\overline{v_i v_k} + \overline{v_k v_j} + M[i,k] + M[k,j] \right) \end{cases}$$

Note that we define $M[i,j] = 0$ for any $i+2 \le j (\bmod n)$, i.e., any polygon with at most 3 vertices is free of cutting.



4. Termination: $M[1, n-1]$ answers the original question. 4. Time complexity: we define n^2 subproblems, and solving each takes $\Theta(n)$ time; hence the time complexity is $O\left(n^3\right)$

Problem 7. Push Shapes

There are n dominos of height h located at positions $x_1, \ldots, x_n$ on a line. All the positions are distinct. When a domino falls down, it either falls left with probability p or to the right with probability $1-p$. If the domino hits another domino in its way down, that domino will fall in the same direction as the domino that hit it. A domino can hit another domino if and only if the distance between them is strictly less than h. For instance, suppose there are 4 dominos located at positions 1, 3, 5 and 8 and $h = 3$. The domino at position 1 falls right. It hits the domino at position 3 and it starts to fall too. In it's turn it hits the domino at position 5 and it also starts to fall. The distance between 8 and 5 is exactly 3, so the domino at position 8 will not fall.

While there are dominos standing, a kid will select either the leftmost standing domino with probability q or the rightmost standing domino with probability $1 - q$ and will make it fall (it will fall to the left with probability p or to the right with probability $1 - p$). We would like to know the expected total length of the line covered with fallen dominos after all of them are pushed over.

Write a DP algorithm to solve this problem. The running time should be $O(n^2)$. Hint: make sure to work out a few small examples first.

Solution

First let us reindex the dominos by sorting them by x-coordinate. Let $f(i, j, b_1, b_2, i', j')$ where we would like to consider the problem of if we only have dominos $i \cdots j$ standing where $b_1 = 1$ indicates that domino $i-1$ fell right and $b_1 = 0$ if it fell left and $b_2 = 1$ indicates that domino $j+1$ fell right and $b_2 = 0$ if it fell left.

We start with the case that they choose the left domino to be knocked over to the right. The plan is to calculate the expected length in this scenario and multiply by the chance of this case occurring, which is $q(1-p)$. We can easily calculate what is the farthest right domino that falls as a result of this and call it w_i, and we can calculate the segment covered by the fallen dominos and call it $[i, y_{i,R}]$.

Then if $w_i \geq j$ this means the entire segment falls, from which the length of the ground covered by dominos is $L = |x_j - x_i + h|$. If we also have $b_2 = 0$, there may be overlapping from domino $j+1$ falling left; this occurs if $x_{j+1} - x_j > 2h$. If this happens, then we subtract the overlap $(x_j + h) - (x_{j+1} - h)$ from L. We populate the cell with $f(i, j, b_1, b_2) = L$.

If $w_i < j$, then we just consider adding the length of ground covered by dominos $i \cdots w_i$ falling right $L = |w_i - x_i + h|$ and add to the value of the sub-problem $f(w_{i+1}, j, 1, b_2)$, so $f(i, j, b_1, b_2) = L + f(w_{i+1}, j, 1, b_2)$.

There is another interesting case when a kid chooses the left domino and it falls left. In this case we calculate the expected length and multiply by the chance of this occurring, which is qp. The expected length of ground covered by the dominos here is just the length contributed by domino i falling left, which is $min(x_i - x_{i-1}, h)$ when $b_1 = 0$ and $min(x_i - (x_{i-1} + h), h)$ when $b_1 = 1$, plus the value of sub-problem $f(i+1, j, 0, b_2)$.

Cases where the right domino are chosen are analogous by symmetry.

Doing this naively would take $\mathcal{O}(n^3)$ time but this can be lowered to $\mathcal{O}(n^2)$ by pre-computing the furthest domino w_i toppled when i falls left or right (quadratic time) so that computing one cell of f only takes constant time ($O(n^2)$ many cells).

Problem 8. Price Spread

Cécile has a copper pipe of length n inches and an array of nonnegative integers that contains prices of all pieces of size smaller than n. She wants to find the maximum value she can make by cutting up the pipe and selling the pieces. For example, if length of the pipe is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6).

length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

Give a dynamic programming algorithm so Cécile can find the maximum obtainable value given any pipe length and set of prices. Clearly describe your algorithm, prove its correctness and runtime.

Solution

DP Algorithm

We create a recursive formula, where for each subproblem of length k we choose the cut-length i such that $Price(i) + Value(k-i)$ is maximized. Here $Price(i)$ is the price of selling the full pipe of length i and $Value(k-i)$ is the amount obtained after optimally cutting the pipe of length $k-i$.

```
def cutPipe(price, n):
  val = [0 for x in range(n+1)]
  val[0] = 0

  # Build the table val[] in bottom up manner and return
  # the last entry from the table
  for i from 1 to n:
    max_val = 0
    for j from 0 to i-1:
      max_val = max(max_val, price[j] + val[i-j-1])
    val[i] = max_val

  return val[n]
```

Correctness

An inductive proof on the length of the pipe will show that our solution is correct. We let $cutPipe(n)$ represent the optimal solution for a pipe of length n. Base case: If the pipe is length 1, $cutPipe(1) = Price(1) = Val(1)$. Inductive: Assume the optimal price is found for all pipes of length less than or equal to k. If the first cut the algorithm makes x_1 is not optimal, then there is an x_1' such that $Val((k+1)-x_1)+Price(x_1) < Val((k+1)-x_1')+Price(x_1')$. By the induction hypothesis, this implies that $cutPipe((k+1)-x_1)+Price(x_1) < cutPipe((k+1)-x_1')+Price(x_1')$. So the algorithm must have chosen x_1' instead of x_1, by construction (a contradiction). Therefore, by induction $cutPipe(n) = Val(n)$ for all $n > 0$.

Time Complexity

The algorithm contains two nested for-loops resulting in a run-time of $\mathcal{O}(n^2)$.

Problem 9. Apples of My Tote

You decide to go apple picking at the Natural Apple Orchard, where the trees grow apples whose mass is always a (positive) natural number of grams. You pay for a bag that can hold up to m grams of apples and set out to pick as many grams of apples as you can. You were told that there are n unique masses $a_1..a_n$ among the remaining apples. Assuming there are d_i many apples that weigh a_i grams in the orchard, what is the maximum amount of apples that can you pick? Find an $O(nm\log m)$ time DP algorithm that answers this question. Please also give the space complexity of your algorithm.

Hint: Think about the binary representation of d_i.

Solution

Idea & Reduction

We reduce this problem to the knapsack problem (with value and weight being equal). One correct reduction would be to just make each apple in the orchard an item that could go in our size m knapsack (so we have d_1 items of $w, v = a_1$ etc), but that is too many items to achieve the required running time. We can reduce the total number of items by having each item represent one bit of d_i instead of representing a single apple.

Reduction: First we upper bound each d_i by m since we can fit at most m apples in our bag. Let $b_i = \lfloor \log d_i \rfloor$ denote the number of bits in d_i. Then we create all our knapsack items by making up to $\log m$ items for each apple mass:

$$I = \langle 2^j \cdot a_i | j \in [b_i - 1], 1 \le i \le n \rangle ^\frown \langle (d_i - (2^{b_i} - 1)) \cdot a_i | 1 \le i \le n \rangle$$

We are using sequences instead of sets above to allow duplicate values. The right side sequence is for handling the case where d_i is not a power of 2 (see correctness proof). Note that we are representing our items with a single number since their weight and value are equal. We keep the maximum capacity of our knapsack as m.

Correctness

All selections of items in the original problem can be represented in the reduced problem: this is given by the construction of the reduced problem from the original problem.

All selections of items in the reduced problem correspond to a valid selection of items in the original problem. It is sufficient to show that all possible selections in the reduced problem correspond to a selection of $p_i \in \mathbb{Z}$ apples of mass a_i where $0 \leq p_i \leq d_i$.

- Reduced problem can pick at most d_i apples of mass a_i: Fix an arbitrary $1 \leq i \leq n$. The sum of all the terms corresponding to the i^{th} apple weight are $(d_i - (2^{b_i} - 1) + \sum_{0 \leq i \leq b_i - 1} 2^i) \cdot a_i = (d_i - (2^{b_i} - 1) + 2^{b_i} - 1) \cdot a_i = d_i a_i$.
- Reduced problem cannot pick fewer than 0 apples of mass a_i: all items in I have non-negative value and weight.
- Reduced problem only picks natural multiples of apples with mass a_i: all items in I have value and weight that are natural multiples of a_i. Sums of natural multiples of a_i are still natural multiples of a_i.
- p_i can take every integral value between 0 and d_i: Clearly p_i can be any integral value between 0 and $2^{b_i} - 1$ or between $d_i - (2^{b_i} - 1)$ and d_i. There is no integer value between $2^{b_i} - 1$ and $d_i - (2^{b_i} - 1)$ since otherwise $d_i - 2(2^{b_i} - 1) \geq 2$ so $d_i \geq 2^{b_i + 1}$ which contradicts the definition of b_i.

All selections of items have the same value, cost, and cost limit in the original and reduced problems, so by the statements above, an optimal selection in the original problem is an optimal selection in the reduced problem and vice versa. □

Complexity

Time Complexity: we have m total weight and $O(n \log m)$ items since we have one item per bit of d_i for each apple mass and there are $\lfloor \log m \rfloor$ bits of d_i and n unique apple masses. So using the knapsack algorithm from lecture gives us a time complexity of $O(nm \log m)$.

Space Complexity: we fill out an $n \log m \times m$ table so the complexity is $O(nm \log m)$.