Michael H. Goldwasser
Saint Louis University

Contest Problem Spoilers

This handout explores the contest problems from the previous class, providing both a high-level explanation of the dynamic programming strategy for each problem, as well as actual code that successfully solves the contest problem.

RIPOFF

For the sake of notation, we define space 0 to be the start; spaces 1 through N to be those given in the problem, with space i having a value v_i which is paid when landing on that spot; and space $N+1$ as the end, with implicit value $v_{N+1} = 0$.

We define subproblems which represented shortened versions of the original game, parameterized by both the index that is considered the end of the board and the number of turns that may be used. Formally, we define notation $P_{j,k}$ to be the maximum profit that can be collected in a shortened game that ends at space j using at most k turns, where $P_{j,k} = -\infty$ if it is impossible to reach j using at most k turns. By this definition, the maximum profit in the full game is denoted as $P_{N+1,T}$. We claim that $P_{j,k}$ can be defined recursively as follows:

$$P_{j,k} = \begin{cases} 0 & \text{if } j = 0 \\ -\infty & \text{if } j \neq 0 \text{ and } k = 0 \\ \max\limits_{\max(0,j-S) \leq i < j} (P_{i,k-1} + v_j) & \text{if } j > 0 \text{ and } k > 0 \end{cases}$$

We justify this formula as follows. If the game ends at the start, then zero profit is achieved, regardless of the number of turns that are allowed. If however, the end of the game is different from the start and 0 turns are allowed, it is impossible to finish the game, and therefore $P_{j,0} = -\infty$. In the most general case, any path ending on space j must have its final move come from some space i such that $max(0, j - S) \leq i < j$, given that a player may move at most S spaces on a turn. If a solution ends with a final move from such i to j, the maximum profit that can be achieved would be the value v_j plus the maximum profit that could have been achieved in a game ending at i using at most $k-1$ turns (since one turn is reserved to move from i to j). Although it is not immediately evident which i is optimal, taking the maximum of $(P_{i,k-1} + v_j)$ over all possible i values assures the optimality for $P_{j,k}$.

This recursive formula for $P_{j,k}$ can be used as the basis for a dynamic programming algorithm to compute $P_{N,T}$. Note that there are $\Theta(NT)$ subproblems by our definition. Since subproblems with parameter k depend only on known values with parameter $k-1$, we can perform a bottom-up dynamic program starting with base case $k = 0$, and then progressing to $k = 1$, and so on. Each individual value $P_{j,k}$ can be computed in time $O(S)$, given the bound on the number of choices within the max-term. Therefore, the original problem is solved in time $O(N \cdot T \cdot S)$; note that for the programming contest, $N \cdot T \cdot S \leq 400{,}000$, which is a reasonable number of computations to perform.

While there are $\Theta(NT)$ subproblems to compute, we need not store an entire $\Theta(NT)$ table in memory, since we only need the values for $k-1$ to compute the values for k, so two tables of length $\Theta(N)$ suffice. In fact, if careful we can use a single table, because we know $P_{j,k} \geq P_{j,k-1}$ as we define $P_{j,k}$ to use *at most* k terms and thus a solution using less turns is feasible. So this means we can simply use the table for $k-1$ as the initial table for k, though we then need to make sure to process updates from largest j to smallest so that we don't inadvertently use a recently updated value in the same round of updates.

```
1  while True:
2      pieces = [int(k) for k in input( ).split( )]
3      if pieces[0] == 0: break
4      N,S,T = pieces
5      board = [ ]
6      while len(board) < N:
7          board.extend(int(k) for k in input( ).split( ))
8      board = [0] + board + [0]    # convention with start=0 and end=N+1
9
10     # done parsing input. Time to solve the problem
11     profit = [0] + [float('-inf')]*(N+1)      # base case values with 0 turns allowed
12     for t in range(T):
13         for j in range(N+1, 0, -1):    # update from biggest to smallest!
14             for i in range(max(0,j-S), j):
15                 profit[j] = max(profit[j], profit[i]+board[j])
16     print(profit[N+1])
```

Pascal's Travels

This problem is a typical combinatorial example in which we must count the number of ways something can be accomplish, and which we use dynamic programming to derive the total count even without taking the time to individually consider each such solution that contributes to the count. In particular, this problem is a specific example of counting the number of paths from a source to a sink in a directed acyclic graph (DAG). In this case, the graph is implicit with edges moving from one square to another based on the rules of the game.

For notation, we consider the $n \times n$ board with rows and columns numbered 0 to $n-1$ inclusive, with $(0,0)$ the start at the top-left and $(n-1, n-1)$ the goal at the bottom-right. For notation, we let $v_{(x,y)}$ denote the value given at position (x,y) of the original board. We approach this problem by considering every square (x,y), and for each, consider $P_{(x,y)}$ to be the number of distinct paths that can be traced from that (x,y) to the goal[1] at $(n-1, n-1)$. We claim the following recursive definition of $P_{(x,y)}$ holds:

$$P_{(x,y)} \quad = \quad \begin{cases} 1 & \text{if } (x,y) = (n-1, n-1) \\ 0 & \text{if } v_{(x,y)} = 0 \text{ and } (x,y) \neq (n-1, n-1) \\ 0 & \text{if } x > n-1 \text{ or } y > n-1 \\ P_{(x+v_{(x,y)},\ y)} + P_{(x,\ y+v_{(x,y)})} & \text{otherwise} \end{cases}$$

As a base-case, we define $P_{(n-1,n-1)}$ to conventionally equal 1, since the one unique solution from $(n-1, n-1)$ is to remain there. We note that if you ever reach a position other than the end with $v_{(x,y)} = 0$, then you cannot move and thus there are no solutions from said position. Similarly, if x or y is strictly greater than $n-1$ there can be no solutions, since you may never move leftward or upward. Finally, for a general starting position (x,y), any solution that starts with a downward move must be distinct from any solution that starts with a rightward move. Therefore, the solutions counted within $P_{(x+v_{(x,y)},\ y)}$ and $P_{(x,\ y+v_{(x,y)})}$ are distinct from each other, and constitute all possible solutions when starting at (x,y), since the first move must be to either $(x + v_{(x,y)},\ y)$ or $(x,\ y + v_{(x,y)})$.

With knowledge of this recurrence, we see that dynamic programming can be used to solve the problem in $O(n^2)$ time and $O(n^2)$ space. We will create an $n \times n$ table to store the $P_{(x,y)}$ values for $0 \leq x, y \leq n-1$. In bottom-up fashion, we can compute all values for $x = n-1$, then $x = n-2$ and so on until $x = 0$, with each such row computed in decreasing order of y. Each individual value can be computed in $O(1)$ time based on the recurrence. The desired output for the problem is the quantity $P_{(0,0)}$.

[1] We could have otherwise considered the count of the number of distinct paths from $(0,0)$ to (x,y), though the recursion would then require a loop over all possible squares that might be the launch point to reach (x,y). Our formulation of where might you go to next has only two possible options.

```
while True:
    n = int(input( ))
    if n == −1: break
    board = [ [int(c) for c in input( )] for r in range(n) ]

    paths = [ [0]∗n for r in range(n) ]
    for x in range(n−1,−1,−1):
        for y in range(n−1,−1,−1):
            if (x,y) == (n−1,n−1):
                paths[x][y] = 1             # special case for goal
            elif board[x][y] != 0:
                right = x+board[x][y]       # new X if we move rightward
                if right < n:               # stepping right lands in bounds
                    paths[x][y] += paths[right][y]
                down = y+board[x][y]        # new Y if we move downward
                if down < n:                # stepping down lands in bounds
                    paths[x][y] += paths[x][down]
    print(paths[0][0])
```

Nine Packs

This problem is a spin on a classic problem known as *Subset Sum* where the goal is to determine whether there exists any subset of a sequence of numbers that sum to a specific value. For this problem, we begin by separately analyzing the hot dogs, not only to determine what quantity of total hot dogs are possible, but also what is the fewest number of packs needed to achieve such a total. (That is, we might want to know that we can achieve 24 hot dogs, and also that it is better to do so with packs 12+12 rather than 8+8+8.)

Actually testing all subsets of n given numbers would require 2^n steps and thus be infeasible for this problem which allows up to 100 such numbers. Dynamic programming can be used to solve subset sum, or this variant, as follows. Starting with hot dogs, let's assume that values $p_1, \ldots, p_H$ are the sizes of the H packs that are available to us. We are interested to know, for all positive values t, whether a total t is achievable, and if so, the minimum number of packs needed to achieve this.

The dynamic program is based on adding an additional parameter $1 \leq k \leq H$, and computing how t might be achieved if we are only allowed to use a subset of the first k values in the sequence. We let notation $m(t, k)$ denote the minimum number of packs needed to achieve total exactly t using a subset of the first k packs in our sequence, with

$m(t,k) = \infty$ if it is infeasible. We claim the following recursive definition

$$m(t,k) \;=\; \begin{cases} 0 & \text{if } t = 0 \text{ and } k = 0 \\ \infty & \text{if } t \neq 0 \text{ and } k = 0 \\ \min\left(m(t,k-1), 1 + m(t-p_k, k-1)\right) & \text{if } k \geq 1 \end{cases}$$

In short, when $k = 0$ we can trivially achieve total 0 without any packs, but cannot achieve any other total. Inductively, when we allow ourselves the additional option of using p_k, we can either use the pack and get the remaining $t - p_k$ from an earlier subset, or not use p_k and get the entire total t from a subset of the first $k - 1$ values.

In terms of implementation, there are $\Theta(NT)$ subproblems for a sequence of N packs with total sum T, and each subproblem requires constant time to compute. As we have seen in other dynamic programs, we can avoid storing a table of size $\Theta(NT)$ and instead use only a table of size $\Theta(T)$ because the entries for parameter k depend only on the entries for parameter $k - 1$.

As for the contest problem, we compute $m(t,n)$ for all possible t for the hot dogs, and then repeat the process for the buns. To conclude, we then consider the total number of combined packs that would be needed for any quantity that is achievable by both hot dogs and buns.

```
1  INF = float('inf')
2
3  def compute_table(V):    # utility to analyze a sequence of values
4      M = sum(V)
5      opt = [0] + [INF] * M   # can achieve 0 with no items
6      subtotal = 0
7      for v in V:
8          subtotal += v
9          for j in range(subtotal,v-1,-1): # loop from big to small to avoid "double update"
10             opt[j] = min(opt[j], 1 + opt[j-v])
11     return opt[1: ]        # return table without the 0 entry
12
13 H = [ int(k) for k in input( ).split( )[1:] ]
14 B = [ int(k) for k in input( ).split( )[1:] ]
15
16 if len(H)>0 and len(B)>0:
17     minH = compute_table(H)
18     minB = compute_table(B)
19     ans = min(a+b for a,b in zip(minH,minB))        # consider a[j]+b[j] for each j
20     print('impossible' if ans == INF else ans)
21 else:
22     print('impossible')
```

Robot Challenge

For modeling the robot challenge, we conventionally treat the starting location, $(0, 0)$, as target 0, and the ending location, $(100, 100)$, as target $n + 1$. For $1 \leq i \leq n$, we let (x_i, y_i) denote the location of target i, and p_i be the penalty incurred if skipping target i. (We are not allowed to skip targets 0 or $n + 1$.)

In analyzing this problem, we let quantity C_i, for $0 \leq i \leq n + 1$, denote the minimal possible "cost" of completing the challenge once leaving target i, where the cost includes the sum of travel time, waiting time, and incurred penalties. We also let $d(j, k)$ denote the Euclidean distance between targets j and k, computed as $\sqrt{(x_k - x_j)^2 + (y_k - y_j)^2}$. The quantity C_i can be defined recursively as follows:

$$C_i = \begin{cases} 0 & \text{if } i = n + 1 \\ \min\limits_{i+1 \leq k \leq n+1} \left(C_k + 1 + d(i, k) + \sum\limits_{i+1 \leq j \leq k-1} p_j \right) & \text{otherwise} \end{cases}$$

We justify this formula as follows. If you are ready to "leave" target $n + 1$, then you have already waited off the mandatory second of time and there is nothing left to do, thus the remaining cost is zero. If you are leaving any other target, your next stop must be at a higher numbered target (or the end, which we model as target $n+1$). That choice is captured by the minimization over parameter k for $i + 1 \leq k \leq n + 1$. If a choice is made to go directly from i to some k, there will be a cost of $d(i, k)$ for travel, a cost of 1 for the waiting time at k before you may leave, a cost of $\sum_{i+1 \leq j \leq k-1} p_j$ as penalties for all of the targets strictly between i and k that have been missed, and finally a cost of C_k for optimally completing the challenge once leaving k.

We claim that this recursively formula can be used with dynamic programming to result in an $O(n^2)$ algorithm to compute the optimal cost C_0. There are a total of $n + 2$ subproblems, which we can compute in bottom up fashion ordered as C_{n+1}, C_n, C_{n-1}, $\ldots$, given that each C_i depends only on values C_k for $k > i$. We next claim that we can compute a particular C_i value in $O(n)$ time. Naively, the recursive formula suggests $O(n^2)$ time, because of the linear computation of the penalty sum nested within the linear minimization over k. However, the penalty sum can be computed more efficiently by storing the value of that sum from one choice of k to the next. That is, if we let $P_{i,k}$ denote the sum $\sum_{i+1 \leq j \leq k-1} p_j$, then $P_{i,k+1} = P_{i,k} + p_k$. Therefore, we can update the cumulative penalty for each successive k in $O(1)$ time. So the overall time to compute each C_i is $O(n)$ and the total algorithm time is $O(n^2)$.

```
while True:
    N = int(input( ))
    if N == 0: break
    targets = [ [int(k) for k in input( ).split( )] for j in range(N) ]
    targets.insert(0, (0,0,0))            # start target
    targets.append( (100,100,0) )         # end target

    cost = [0] * (2+N)                    # note that cost[N+1] is truly zero
    for i in range(N,-1,-1):              # compute cost[i] from larger to smaller
        best = float('inf')               # score to beat
        penaltySubtotal = 0               # penalties for skipping from i to k
        for k in range(i+1,N+2):
            dist = ((targets[i][0]-targets[k][0])**2 + (targets[i][1]-targets[k][1])**2)**0.5
            best = min(best, cost[k] + 1 + dist + penaltySubtotal)
            penaltySubtotal += targets[k][2] # skipping to larger target incurs this penalty
        cost[i] = best

    print(f'{cost[0]:.3f}')
```

Narrow Art Gallery

For sake of notation, we let values ℓ_i and r_i denote the left and right room values, respectively, in row i for $1 \leq i \leq N$. To build a dynamic program we consider subproblems of the following form. For $0 \leq i \leq n$ and $0 \leq j \leq k$, we define $L(i,j)$ as the maximum possible value of rooms in the first i rows that can be left open under the constraint that the left room of row i be open yet j rooms within the first i rows be closed. Similarly, we define $R(i,j)$ as such problem in which the right room in row i be left open, and $B(i,j)$ as the problem in which both rooms of row i be left open. We claim the following mutually recursive definitions for $L(i,j)$, $R(i,j)$ and $B(i,j)$.

$$B(i,j) = \begin{cases} -\infty & \text{if } i = 0 \text{ and } j \geq 1 \\ 0 & \text{if } i = 0 \text{ and } j \leq 0 \\ \ell_i + r_i + \max\left(L(i-1,j), R(i-1,j), B(i-1,j)\right) & \text{if } i >= 1 \end{cases}$$

$$L(i,j) = \begin{cases} -\infty & \text{if } i = 0 \text{ and } j \geq 1 \\ 0 & \text{if } i = 0 \text{ and } j \leq 0 \\ \max\left(\ell_i + \max\left(L(i-1,j-1), B(i-1,j-1)\right), \right. & \\ \qquad \left. \ell_i + r_i + \max\left(L(i-1,j), R(i-1,j), B(i-1,j)\right)\right) & \text{if } i >= 1 \end{cases}$$

$$R(i,j) = \begin{cases} -\infty & \text{if } i = 0 \text{ and } j \geq 1 \\ 0 & \text{if } i = 0 \text{ and } j \leq 0 \\ \max\left(r_i + \max\left(R(i-1,j-1), B(i-1,j-1)\right), \right. & \\ \qquad \left. \ell_i + r_i + \max\left(L(i-1,j), R(i-1,j), B(i-1,j)\right)\right) & \text{if } i >= 1 \end{cases}$$

To understand these formulae, note that as a base case, if no rows are being considered, the problem is infeasible (thus $-\infty$ value) if you are expected to close one or more rooms, and you gain cost 0 if you are not required to close any rooms. More generally, for $B(i,j)$ since both left and right rooms in row i must be kept open, your museum gets credit for value $\ell_i + r_i$ but you have no choice but to close j rooms in the remaining $i-1$ rows. Also, since both rooms in row i are open, we can use a solution in which any one or two rooms are open in the preceding row. For $L(i,j)$, we have a choice of leaving both left and right rooms open in row i or in closing the right room. But if closing the right room, we must make sure that the left room (or both rooms) in row i remains open for travel. The case of $R(i,j)$ is symmetric.

Overall, we note that there are $O(nk)$ subproblems and each can be computed in $O(1)$ time in bottom-up fashion, leading to a solution that can compute $L(n,k)$, $R(n,k)$ and $B(n,k)$ in $O(nk)$ time, with the global optimal being the best of those three. We can do the computation using only $O(k)$ auxiliary space, as the expressions for a fixed i depend only on subproblems for $i-1$.

```
1  while True:
2      N,K = [int(a) for a in input( ).split( )]
3      if N == 0: break
4      rooms = [ [int(a) for a in input( ).split( )] for i in range(N) ]
5
6      L = R = B = [0] + [float('-inf')]*K    # base case values
7      for i in range(N):
8          newB = [ sum(rooms[i]) + max(L[j],R[j],B[j]) for j in range(1+K) ]
9          newL = list(newB)   # use newB values as defaults to beat
10         for j in range(1+K):
11             newL[j] = max(newL[j], rooms[i][0] + max(L[j-1],B[j-1]))
12         newR = list(newB)   # use newB values as defaults to beat
13         for j in range(1+K):
14             newR[j] = max(newR[j], rooms[i][1] + max(R[j-1],B[j-1]))
15         L,R,B = newL,newR,newB
16
17     print(max(L[K],R[K],B[K]))
```

Lawrence of Arabia

At a high-level, this program has great structural similarity to RIPOFF, in that there is a sequence of n items, and you effectively want to break that sequence into at most m pieces ("turns" in RIPOFF, "attacks" in this problem). We will approach this in similar fashion by considering a subproblem in which we have one less attack. However, this problem is a tad more difficult because care must also be taken to be efficient in computing the "strategic value" of components (as the natural definition of that would require quadratic time in the size of the component).

For this problem, there is no reason not to use all attacks (as an attack on a component clearly reduces the value of what remains). To manage the number of subproblems, we can effectively consider "guessing" where to make the last such attack. Let's define the depots $0 \leq j \leq n-1$ and their values as v_j. We will also use notation $SV(j,k)$ to be the Strategic Value of the consecutive group of depots $j \ldots k$, as defined by the problem. The subproblems we consider will be parameterized as $OPT(d,a)$ which is the minimum possible strategic value achievable by using a attacks on the railroad prefix containing only the first d depots.

We claim the following recursive formula.

$$OPT(d,a) \;=\; \begin{cases} 0 & \text{if } d = 0 \\ SV(0,d-1) & \text{if } a = 0 \\ \min\limits_{0 \leq j < d} OPT(j,a-1) + SV(j+1,d) & \text{otherwise} \end{cases}$$

In short, the first rule is that if there are no depots, then they have no value. The second rule is that if you are out of attacks, then the strategic value of the connected depots 1 through d remains. The final statement is justified by noting that the rightmost attack must separate a pair of depots j and $j+1$. Since it is the rightmost attack, the depots $j+1 \ldots d$ remain connected and have $SV(j+1,d$, while the remaining $a-1$ attacks can be deployed on the first j depots.

The only remaining implementation detail is the computation of the SV term when trying all possible j values. If each of those were computed from scratch, the result would be quadratic computation for each subproblem. However, we can easily compute $SV(j,d)$ if we have previously computed $SV(j+1,d)$ and the sum $\sum_{i=j+1}^{d} v_i$ by noting that

$$SV(j,d) = SV(j+1,d) + v_j \cdot \sum_{i=j+1}^{d} v_i.$$

Overall, we have $\Theta(nm)$ subproblems to compute, and with the above optimization, each subproblem can be computed in $O(n)$ time, and thus the overall runtime is $O(n^2 m)$. We can achieve this with only $O(n)$ space, by computing for all $a = 0$, then $a = 1$ and so forth.

```
1  while True:
2      N,M = [int(v) for v in input( ).split( )]
3      if N==0: break
4      depot = [int(v) for v in input( ).split( )]
5
6      OPT = [0] * (N+1)
7      subtotal = 0
8      for d in range(N):
9          OPT[1+d] = OPT[d] + subtotal * depot[d]
10         subtotal += depot[d]
11     for a in range(M):
12         for d in range(N,-1,-1):   # intentionally from biggest to smallest
13             subtotal = 0
14             SV = 0
15             for j in range(d-1,0,-1):
16                 SV += depot[j]*subtotal
17                 subtotal += depot[j]
18                 OPT[d] = min(OPT[d], OPT[j] + SV)
19     print(OPT[N])
```

Note: This Python implementation produces correct answers, but it would not be fast enough for the contest for problems with $n = 1000$. The same approach in C++/Java would suffice; see next page.

```
1  #include <iostream>
2  using namespace std;
3
4  int depot[1000];          // numbered from 0 to 999
5  long opt[1001];           // opt[n] is optimal cost for first n depots (using a attacks)
6
7  int main( ) {
8    while (true) {
9      int N,M;
10     cin >> N >> M;
11     if (N == 0) return 0;
12     for (int i=0; i<N; i++)
13       cin >> depot[i];
14
15     opt[0] = 0;
16     int subtotal = 0;
17     for (int d=0; d<N; d++) {
18       opt[1+d] = opt[d] + subtotal * depot[d];
19       subtotal += depot[d];
20     }
21     for (int a=0; a<M; a++) {
22       for (int d=N; d >= 0; d--) {   // intentionally from biggest to smallest
23         int subtotal = 0;
24         int SV = 0;
25         for (int j=d-1; j > 0; j--) {
26           SV += depot[j]*subtotal;
27           subtotal += depot[j];
28           opt[d] = min(opt[d], opt[j] + SV);
29         }
30       }
31     }
32     cout << opt[N] << endl;
33   }
34 }
```