CS 341, Winter 2019

Trevor Brown, Semih Salihoglu, Doug Stinson

# Assignment 3 (due Friday, March 1, 6:00pm)

**Instructions:**

- Hand in your assignment using Crowdmark. Detailed instructions are on the course website.
- Give complete legible solutions to all questions.
- Your answers will be marked for clarity as well as correctness.
- For any algorithm you present, you should justify its correctness (if it is not obvious) and analyze the complexity.

1. [*10 marks*] *Greedy — Regenerating Robots.*
   A delivery robot is trying to travel in a straight line to a destination $D$ meters away. Along this straight line, there are several robot docking stations at distances $d_1, d_2, \ldots, d_n$ (expressed in meters from the robot's starting position) where the robot can stop and *regenerate* before continuing. The robot can travel at most $k$ meters before it must regenerate. The problem is to identify *which* docking stations the robot should stop at in order to *minimize the number of stops*. For simplicity, you may assume it is always possible to reach the destination.

   **Input:** Variables $D, k$ and $d_1 < d_2 < \ldots < d_n$.
   **Output:** A feasible subsequence of $d_1, d_2, \ldots, d_n$ with the *minimum* total travel time.

   (a) [*4 marks*] Design an greedy algorithm that solves the problem.
   **Solution:**

```
GreedyAlgorithm(D, k, d_1, d_2, ..., d_n):
  // assume: it is possible to travel D meters
  // assume: d_i > 0 for all i
  // assume: D > 0
  stops := []
  mLastStop := 0   // meters from start to our last stop
  mCurrent := 0   // meters currently travelled from start
  for i := 1..n
    if D ≤ mCurrent + k then break   // can reach destination – exit the for loop
    if mCurrent + d_i ≤ mLastStop + k then
      mCurrent := d_i
    else
      mCurrent := d_i
      mLastStop := d_i
      stops.append(d_i)
  return stops   // note: output specification does not ask for final destination D to be returned
```

(b) [*1 marks*] What is the asymptotic runtime complexity of your algorithm?

**Solution:** $O(n)$ steps (assuming we can append to $stops$ in $O(1)$ time)

(c) [*5 marks*] Prove that your algorithm is correct (feasible and optimal).

**Solution:** The answer produced by the algorithm is **feasible** because all stops output by the algorithm are values of $mLastStop$, which is always set to $mCurrent$, which is always within distance $k$ of the previous stop $mLastStop$.

The **optimality** proof is a "greedy stays ahead" style inductive proof. Consider any arbitrary input. Let $G = [g_1, g_2, ..., g_k]$ be the greedy solution for this input, and $O = [o_1, o_2, ..., o_m]$ be an *optimal* solution. We want to show $G$ is optimal (i.e., $k = m$).

**Inductive hypothesis:** $g_i \geq o_i$ for each $i$. (In other words, the greedy solution travels at least as far as the optimal solution, by its $i$th stop, for each $i$.)

**Base case:** The first stop $g_1$ in $G$ is the largest (furthest) possible stop smaller than or equal to $k$, and $o_1$ cannot pick any stop larger than $k$, so $g_1 \geq o_1$.

**Inductive step:** Suppose $g_{i-1} \geq o_{i-1}$ for any $i > 1$. We show $g_i \geq o_i$. Note that $g_i$ is the largest (furthest) stop after $g_{i-1}$ within distance $k$. In other words, every stop greater than $g_i$ is more than distance $k$ from $g_{i-1}$. By the inductive hypothesis, $o_{i-1} \leq g_{i-1}$, so every stop greater than $g_i$ is also more than distance $k$ from $o_{i-1}$. Therefore, the stop $o_i$ must be $g_i$ or a smaller (closer) stop. This proves the inductive hypothesis.

**Concluding the proof:** Since $g_i \geq o_i$ for all $i$, if $o_k \geq D$ (so the optimal solution has reached the destination) then $g_k \geq D$ (so the greedy solution has also). So $O$ has at least as many stops as $G$ (so $G$ is optimal). QED.

2. [*12 marks*] *Greedy — Parade Planning.*

As the head of the parade planning committee, you are responsible for ensuring that every decorated parade float has an appropriately sized truck to carry it.

**Input:** Distinct *parade float sizes* $p_1, p_2, ..., p_n$ and distinct *truck sizes* $t_1, t_2, ..., t_n$. You can assume $t_1, ..., t_n$ are given in increasing order.

**Output:** A permutation $\pi = \pi(1), \pi(2), ..., \pi(n)$ that *minimizes* **"error"** $\sum_{k=1}^{n}(t_k - p_{\pi(k)})^2$.

(a) [*4 marks*] Consider the following algorithm: for $i = 1...n$, match the truck with size $t_i$ to the unmatched parade float with the closest size $p_j$ to $t_i$. Prove that this algorithm is not optimal.

**Solution:** Consider the following input: two trucks with sizes $t_1 = 10$ and $t_2 = 20$, and two parade floats with sizes $p_1 = 8$ and $p_2 = 11$. The suggested algorithm will match $t_1$ to $p_2$, and $t_2$ to $p_1$, so we get error $\sum_{k=1}^{n}(t_k - p_{\pi(k)})^2 = 1^2 + 12^2 = 145$. However, matching $t_1$ to $p_1$, and $t_2$ to $p_2$, we get error $\sum_{k=1}^{n}(t_k - p_{\pi(k)})^2 = 2^2 + 9^2 = 85$. Thus, the suggested algorithm is not optimal.

(b) [*8 marks*] Design a greedy algorithm for this problem and prove it is correct.

*Hint: to prove optimality, it may be helpful to fix an arbitrary input, and consider the output of the greedy algorithm and the output of an optimal algorithm, and suppose they differ. Try to show that the "optimal" solution can be improved to obtain a contradiction.*

**Solution:**

```
GreedyAlgorithm(p_1, p_2, ..., p_n, t_1, t_2, ..., t_n):
    // Sort parade float sizes in increasing order, while remembering the original index of each
    pairs := new Array large enough to hold n Pairs
    for i := 1..n
        pairs.append(new Pair(p_i, i))
    sort pairs in increasing order by first component (by p_i values)

    // Conceptually, we match each truck t_i to the float at index i in pairs.
    // Return the original index of each parade float.
    result := new Array of n integers
    for i := 1..n
        result.append(pairs[i].second)
    return result
```

To show *feasibility*, we need only prove that *result* is a permutation of $1..n$. The second coordinates of the pairs in the *pairs* array initially form a permutation, and sorting the pairs does not change this.

To show *optimality*, we make a greedy exchange argument. Consider any arbitrary input, and let $G = (g_1, ..., g_n)$ be a solution produced by our greedy algorithm, and $O = (o_1, ..., o_n)$ be a solution produced by an optimal algorithm. If $G$ and $O$ are the same, then $G$ is optimal, and we are done. So, to obtain a contradiction, suppose they differ. Consider the first index $i$ where they differ. Note $g_j = o_j$ for all $j < i$.

By the sort order, parade float size $p_{g_i}$ is *smaller* than float size $p_{o_i}$. Let $t_k$ be the truck that is matched with float size $p_{g_i}$ in $O$. Since $g_j = o_j$ for all $j < i$, and $g_i \neq o_i$, we must have $k > i$. So, by the input order, $t_k > t_i$.

In $O$, matching $t_k$ with float size $p_{g_i}$ contributes term $(t_k - p_{g_i})^2$ to the error. Matching $t_i$ with $p_{o_i}$ contributes term $(t_i - p_{o_i})^2$ to the error.

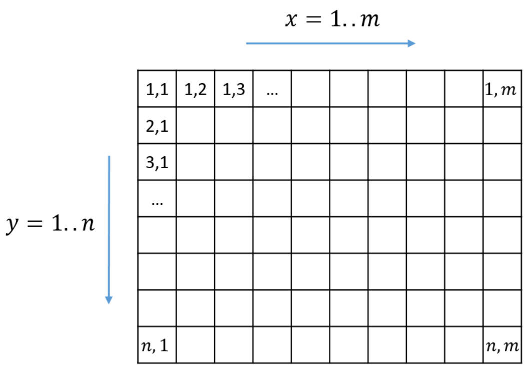
Consider a solution $O'$ that is constructed from $O$ by *exchanging* $g_i$ and $o_i$. The error for $O'$ is the same as $O$, except that the term $(t_k - p_{g_i})^2$ for the $k$th truck becomes $(t_k - p_{o_i})^2$, and the term $(t_i - p_{o_i})^2$ for the $i$th truck becomes $(t_i - p_{g_i})^2$. We prove that the error for $O'$ is *smaller* than the error for $O$, which will contradict the optimality of $O$ (proving our assumption that $G \neq O$ must be false).

Suppose the error for $O'$ is smaller than the error for $O$. This is equivalent to $(t_k - p_{g_i})^2 + (t_i - p_{o_i})^2 < (t_k - p_{o_i})^2 + (t_i - p_{g_i})^2$. Expanding and simplifying, we obtain $t_i p_{g_i} + t_k p_{o_i} > t_i p_{o_i} + t_k p_{g_i}$. We can then rearrange to $t_i(p_{g_i} - p_{o_i}) + t_k(p_{o_i} - p_{g_i}) > 0$, and again to $(p_{o_i} - p_{g_i})(t_k - t_i) > 0$. We have argued above that $p_{o_i} > p_{g_i}$ and $t_k > t_i$, so both of the parenthesized terms in the previous relation are positive, so the relation must hold. QED.

3. [*12 marks*] *Dynamic programming — Multipath Metropolis.*
Consider a city containing infinitely long horizontal streets with $y$-coordinates $1, 2, ..., n$ and infinitely long vertical streets with $x$-coordinates $1, 2, ..., m$. Note that the city contains $nm$ intersections forming a regular 2D grid. Currently, some of these intersections, denoted $(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)$, are under construction and *cannot* be driven through. Your task is to determine *how many different paths* there are from the top left intersection $(1, 1)$ to the bottom right intersection $(m, n)$, if you can only drive the right ($+x$) and down ($+y$).

(a) [*8 marks*] Design an $O(nm)$ dynamic programming algorithm to solve this problem. Write the recurrence for your solution, and provide pseudocode for your algorithm.

**Solution:** There are some inconsistencies in the coordinate system in the problem description. So, we accept solutions that use *any sensible and self-consistent coordinate system*. In this solution, we assume the following coordinate system. Cells in the diagram represent intersections. We write coordinate pairs $(y,x)$ with the *y-coordinate listed first*.




This coordinate system is consistent with the question description, except that the bottom-right intersection is $(n,m)$.

Let $N[y,x]$ represent the number of paths from $(y,x)$ to $(n,m)$. Note that $N[1,1]$ represents the solution to the entire problem. From any cell $(y,x)$ we can move to the right, or down, unless we are blocked (by construction or because we are at the edge of the world). If we move to the right, then there are $N[y,x+1]$ ways to reach the destination. If we move down, then there are $N[y+1,x]$ ways.

Base cases: we start by setting $N[n,m]=1$.[1] If an intersection is blocked, then there are *zero* ways to reach the destination from that cell, so $N[y,x]=0$ for $(y,x)\in\{(y_1,x_1),(y_2,x_2),\ldots,(y_k,x_k)\}$. ~~If $y=n$ or $x=m$, then there is only *one* way to reach the destination—a straight line, so $N[n,x]=1$ for all $x$ and $N[y,m]=1$ for all $y$ (as long as the intersections are *not blocked*).~~[2] We assume there is one **extra** row and column to simplify base cases: $N[y,x]=0$ when $y>n$ or $x>m$.

General case: For all **other** cells, we define $N[y,x]=N[y+1,x]+N[y,x+1]$.

We *implement* this recurrence via dynamic programming by creating an array $N[1..n+1,1..m+1]$ and filling in all of its values, then returning $N[1,1]$. We first fill in the base cases, then we fill in the rest of the array, skipping cells that are already filled in.

Pseudocode:

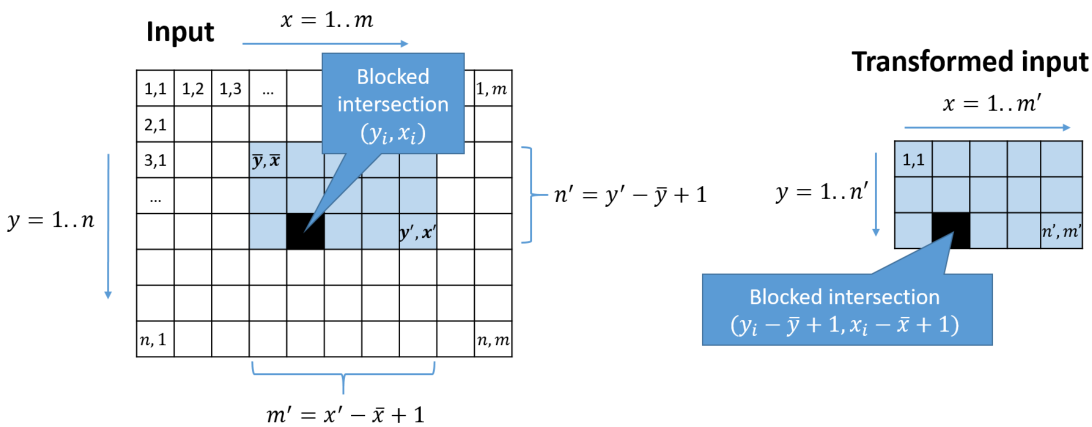
[1] This crucial base case was missing in the Mar 20th version of this solution, causing the entire array to contain either 0 or -1 in each cell. The mistakes made in this solution are a pretty good illustration of why it's a good idea to test DP solutions on small inputs to see if they contain obvious missing cases.

[2] The crossed-out base case previously had a subtle error: if an intersection $(y',m)$ in the rightmost column was blocked, then all cells $(y,m)$ above this cell (satisfying $y<y'$) should have satisfied $N[y,m]=0$, since there is no path from any of those intersections to the final intersection! Similarly, blocked intersections in the bottommost column should have caused cells to the left to be zero. The mistake was that these cells were blindly set to 1. The new solution simplifies the base cases by adding an extra row and column, and setting those cells to zero (since we know trivially there there is no path to the destination from any of those cells). Why does this help? Because adding that extra row and column eliminates the need for a special formula for the rightmost column and bottommost row to avoid going out-of-bounds. Using extra rows/columns often helps simplify base cases like this, making mistakes less likely.

```
DPAlgorithm(n, m, x_1, x_2, ..., x_k, y_1, y_2, ..., y_k):
  N[1..n+1, 1..m+1] := new Array with each cell containing −1 (representing "not yet filled in")
  // Base cases
  N[n,m] = 1  // there is one way to (n,m) from (n,m), namely staying there [added since last solution]
  for y := 1..n+1 do N[y, m+1] = 0  // "out-of-bounds" vertical street
  for x := 1..m+1 do N[n+1, x] = 0  // "out-of-bounds" horizontal street
  for i := 1..k do N[y_i, x_i] = 0  // blocked intersection
  // General cases
  for y := n..1
    for x = m..1
      if N[y,x] ≠ −1 goto next loop iteration // already filled in as a blocked intersection
      N[y,x] = N[y+1,x] + N[y,x+1]
  return N[1,1]
```

(b) [*4 marks*] Suppose you want to solve a new variant of this problem that asks how many different paths there are from one arbitrary intersection $(\bar{y},\bar{x})$ to another $(y',x')$ where $\bar{x} \le x'$ and $\bar{y} \le y'$. Describe how you could use your solution to part (a) as a black box to solve this new problem variant. (In other words, give a reduction from this new problem variant to the original problem.)

**Solution:** The input to this problem contains everything in the input to the original problem (in part (a)) **as well as** a pair of intersections $(\bar{y},\bar{x})$ and $(y',x')$. Our goal is to show how to **transform this input** so that (1) it can be fed to the DPAlgorithm procedure we wrote above, and (2) the return value of DPAlgorithm will be the number of paths from $(\bar{y},\bar{x})$ to $(y',x')$. This transformation is illustrated below.




**We transform the input as follows.** We start by computing new problem dimensions $n' = y' - \bar{y} + 1$ and $m' = x' - \bar{x} + 1$. We then *discard* all blocked intersections that are not inside the rectangle from $(\bar{y},\bar{x})$ to $(y',x')$ (inclusive). For each remaining blocked intersection $(y_i,x_i)$, we adjust its coordinates so that $(\bar{y},\bar{x})$ represents $(1,1)$ by subtracting $(\bar{y}-1)$ from $y_i$ and subtracting $(\bar{x}-1)$ from $x_i$.

**To use this transformed input**, we pass it to DPAlgorithm and return the result.

**Bonus: C++ code for Q3(a).**

```
#include <iostream>
#include <cstdlib>
#include <cassert>
using namespace std;

#define FOR(x,a,b) for(int (x)=(a);(x)<=(b);++(x))
#define FOR_DOWNTO(x,a,b) for(int (x)=(a);(x)>=(b);--(x))
#define MAXN 100
#define MAXM 100
#define MAXK 100

// allocate larger arrays & leave first slot empty in all dimensions
// to avoid changing coordinate systems from, e.g., 1..n to 0..(n-1)
int N[MAXN+2][MAXM+2], X[MAXK+1], Y[MAXK+1];

int main(void) {
    // input
    int n, m, k=1;
    cin>>n>>m;
    while (cin>>X[k]) {
        cin>>Y[k];
        assert(X[k]>=1 && X[k]<=MAXM && Y[k]>=1 && Y[k]<=MAXN);
        ++k;
    }
    assert(n<=MAXN && m<=MAXM && k<=MAXK);

    // "not yet filled in"
    FOR(y,1,n+1) FOR(x,1,m+1) N[y][x] = -1;

    // base cases
    N[n][m] = 1;
    FOR(y,1,n+1) N[y][m+1] = 0;
    FOR(x,1,m+1) N[n+1][x] = 0;
    FOR(i,1,k) N[Y[i]][X[i]] = 0;

    // general cases
    FOR_DOWNTO(y,n,1) FOR_DOWNTO(x,m,1) {
        if (N[y][x] != -1) continue;
        N[y][x] = N[y+1][x] + N[y][x+1];
    }

    // output
    FOR(y,1,n) FOR(x,1,n) printf("%5d%s", N[y][x], (x==n?"\n":" "));
    return 0;
}
```

4. [*16 marks*] *Dynamic programming — Hungry Hungry Hippos.*

Two hippos, Alice and Bob, play a turn-based game involving a stack of pancakes. Each hippo's goal is to eat a larger *volume* of pancake than the other hippo. Pancake volumes are measured in cubic inches. In each turn, a hippo can take one pancake from either the top **or** the bottom of the stack of pancakes, and eat it. Alice starts the game first, and the two hippos alternate turns until there are no pancakes left. Assume that Alice and Bob each play *optimally*. (That is, in each of Alice's turns, she takes the action that will yield the best result, under the assumption that Bob will do the same in each of his turns.)

**Input:** A sequence of pancake volumes $v_1, v_2, ..., v_n$ ($v_1$ is the top, $v_n$ the bottom).
**Output:** The total volume Alice will eat, assuming both hippos play optimally.

Design an $O(n^2)$ dynamic programming algorithm to solve the problem. Write the recurrence(s) for your solution and provide pseudocode for your algorithm.

(Hint: Consider using two recurrences, defined in terms of each another, to simulate Alice and Bob taking alternating turns.)

**Solution:** Let $A[i, j]$ = volume eaten by Alice's if it is her turn and only pancakes $i..j$ remain, and $B[i, j]$ = volume eaten by Alice if it is Bob's turn and only pancakes $i..j$ remain.

Base case: note that if $j < i$, then $A[i, j]$ and $B[i, j]$ are *undefined*. For this base case, it is reasonable to choose $A[i, j] = 0$ and $B[i, j] = 0$, since there are no pancakes to eat.

Base case: when $j = i$, there is only one pancake—the $i$th one. If it is Alice's turn, she eats it, so $A[i, j] = v_i$. But if it is Bob's turn, he eats it, and Alice eats *nothing* in range $i..j$, so $B[i, j] = 0$.

General case: if it is Alice's turn and pancakes $i..j$ remain, she can eat the $i$th or $j$th pancake (and $v_i$ or $v_j$ will be added to her score). The next turn will be Bob's, so the best she can do with the *remaining* pancakes will be $B[i+1, j]$ if she takes the $i$th one, and $B[i, j-1]$ if she takes the $j$th one. She maximizes her own score. Thus, $A[i, j] = max\{v_i + B[i+1, j], v_j + B[i, j-1]\}$.

If it is Bob's turn and pancakes $i..j$ remain, he can eat the $i$th or $j$th pancake (*neither* of which is added to Alice's score). The next turn will be Alice's, and the best she can do with the *remaining* pancakes will be $A[i+1, j]$ if Bob takes the $i$th one, and $A[i, j-1]$ if Bob takes the $j$th one. Bob minimizes Alice's score. Thus, $B[i, j] = min\{A[i+1, j], A[i, j-1]\}$.

We *implement* these recurrences via dynamic programming by creating arrays $A[1..n, 1..n]$ and $B[1..n, 1..n]$, filling in all of their values (being careful to use an order that will satisfy all data dependencies), then returning $A[1, n]$.

Pseudocode:

```
DPAlgorithm(v_1, v_2, ..., v_n) :
  A[1..n, 1..n] := new Array
  B[1..n, 1..n] := new Array

  // Base case: j < i (no pancakes)
  for i := 2..n
    for j = 1..i-1
      A[i, j] := 0
      B[i, j] := 0

  // Base case: j = i (one pancake)
  for i := 1..n
    A[i,i] := v_i  // Alice eats it
    B[i,i] := 0  // Bob eats it

  // General case: j > i (two or more pancakes)
  for i := (n-1)..1  // largest i first to satisfy data dependencies on B[i+1, j] and A[i+1, j]
    for j = (i+1)..n  // smallest j first to satisfy data dependencies on B[i, j-1] and A[i, j-1]
      A[i, j] := max{v_i + B[i+1, j], v_j + B[i, j-1]}
      B[i, j] := min{A[i+1, j], A[i, j-1]}
  return A[1,n]

  // Observe that B[i+1, j] and B[i, j-1] are computed before A[i, j].
  // Similarly A[i+1, j] and A[i, j-1] are computed before B[i, j].
```

5. [*16 marks*] *Practical programming problem*

   This assignment includes an implementation question (C++ or Python), which will be submitted *separately* via Marmoset. See Piazza for further instructions. The deadline for this implementation question is the same as for the written assignment.

   **Solution:**

The following is a **dynamic programming** implementation in C++.

It passes 1000 test cases in 12 seconds and uses 0.5MiB of memory.

```cpp
#include <iostream>
#include <limits>
using namespace std;

#define FOR(x,a,b)          for (int (x) = (a); (x) <= (b); ++(x))
#define FOR_DOWNTO(x,b,a)   for (int (x) = (b); (x) >= (a); --(x))
#define K                   (10)
#define M                   (100)
int N[K+2][M+2][M+2];

int main(void) {
    // initialize N to zeros (all cases not explicitly handled below)
    FOR(b,0,K+1) FOR(l,0,M+1) FOR(r,0,M+1) N[b][l][r] = 0;

    // special case N[b,l,r] where b = 1:
    FOR(l,1,M) FOR(r,l,M) N[1][l][r] = r*(r+1)/2 - (l-1)*l/2;

    // general case N[b,l,r] where b > 1:
    FOR(b,2,K) FOR(r,1,M) FOR_DOWNTO(l,r,1) {
        N[b][l][r] = numeric_limits<int>::max();
        FOR(i,l,r) N[b][l][r] = min(N[b][l][r], max(i+N[b-1][l][i-1], i+N[b][i+1][r]));
    }

    // process input, then reconstruct and output the sequence of explosive tests
    int k, m;
    cin>>k>>m;
    cout<<N[k][1][m]<<" = sum {";
    int b = k, l = 1, r = m;
    while (N[b][l][r] > 0) {
        FOR(i,l,r) {
            if (N[b][l][r] == max(i+N[b-1][l][i-1], i+N[b][i+1][r])) {
                cout<<" "<<i;
                if (N[b-1][l][i-1] > N[b][i+1][r]) {
                    b = b-1;
                    r = i-1;
                } else {
                    l = i+1;
                }
            }
        }
    }
    cout<<" }"<<endl;
    return 0;
}
```

The following is a **memoization** implementation in C++.

It passes 1000 test cases in 8 seconds and uses 0.5MiB of memory.

```cpp
#include <iostream>
#include <limits>
using namespace std;

#define FOR(x,a,b)      for (int (x) = (a); (x) <= (b); ++(x))
#define K               (10)
#define M               (100)
int memos[K+2][M+2][M+2];

int N(int b, int l, int r) {
    // check if already computed
    if (memos[b][l][r] >= 0) return memos[b][l][r];

    // special cases
    if (l > r) return (memos[b][l][r] = 0);
    if (b == 1) return (memos[b][l][r] = r*(r+1)/2 - (l-1)*l/2);

    // general case
    int result = numeric_limits<int>::max();
    for (int i=l; i<=r; ++i) {
        result = min(result, max(i+N(b-1,l,i-1), i+N(b,i+1,r)));
    }
    return (memos[b][l][r] = result);
}

int main(void) {
    // initialize all memos to -1 (indicates slot is not initialized yet)
    FOR(b,0,K+1) FOR(l,0,M+1) FOR(r,0,M+1) memos[b][l][r] = -1;

    // process input, then reconstruct and output the sequence of explosive tests
    int k, m;
    cin>>k>>m;
    cout<<N(k,1,m)<<" = sum {"; // recursive memoized call
    int b = k, l = 1, r = m;
    while (N(b,l,r) > 0) {
        FOR(i,l,r) {
            if (N(b,l,r) == max(i+N(b-1,l,i-1), i+N(b,i+1,r))) {
                cout<<" "<<i;
                if (N(b-1,l,i-1) > N(b,i+1,r)) {
                    b = b-1;
                    r = i-1;
                } else {
                    l = i+1;
                }
            }
        }
    }
    cout<<" }"<<endl;
    return 0;
}
```