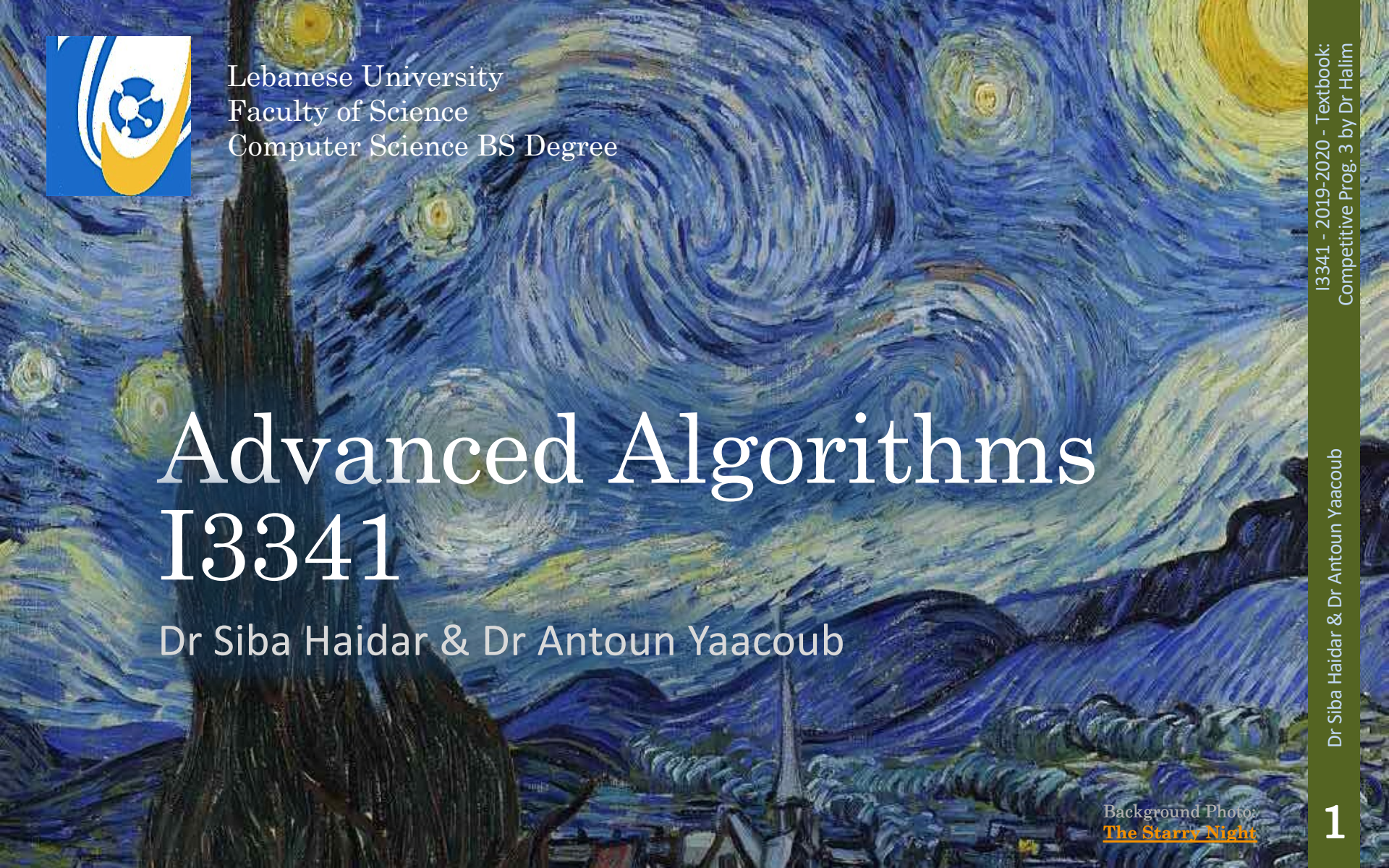


Lebanese University
Faculty of Science
Computer Science BS Degree

Advanced Algorithms I3341

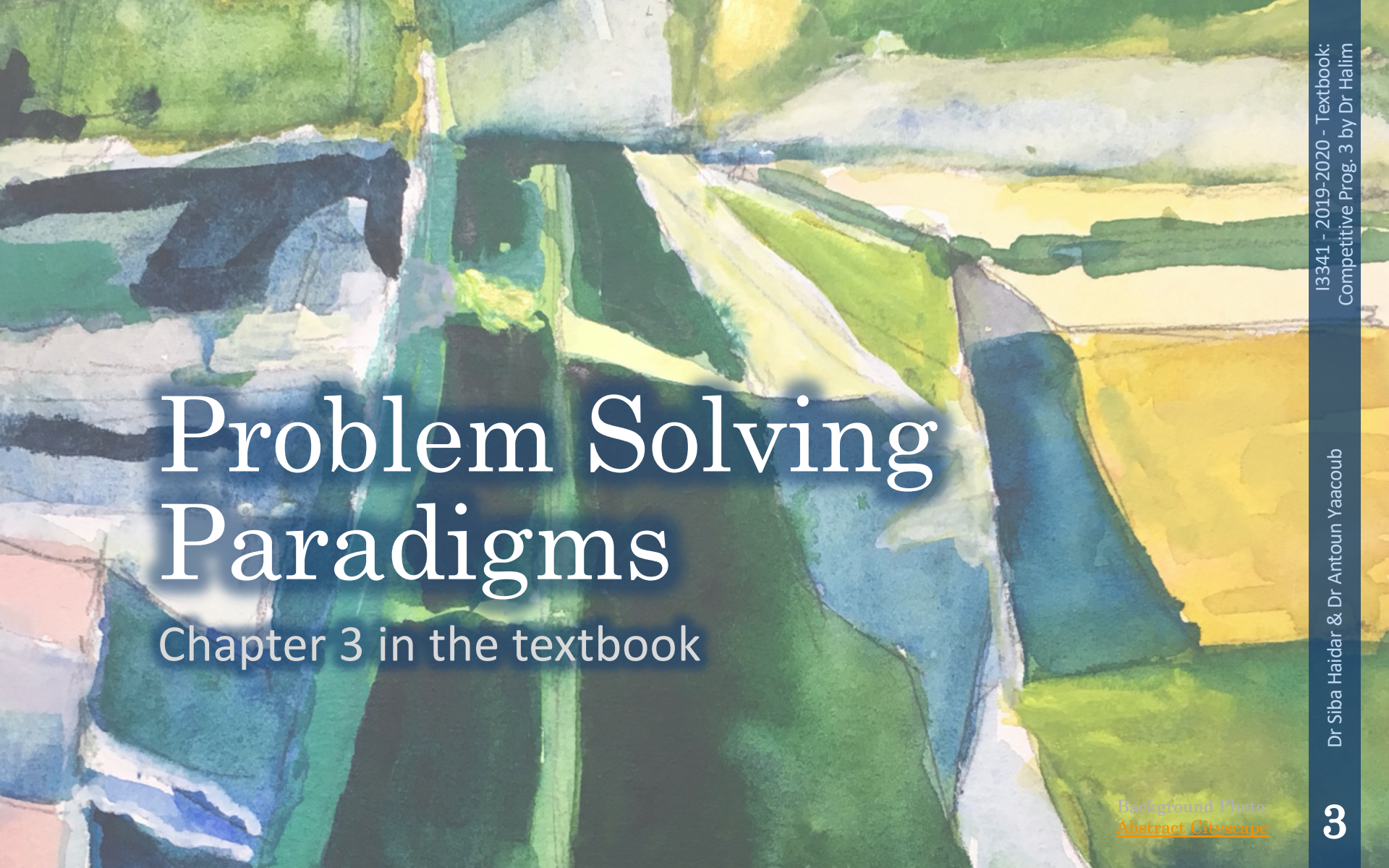
Dr Siba Haidar & Dr Antoun Yaacoub

Background Photo:
The Starry Night

Course Chapters

as per the textbook

1. Introduction
2. Data Structures and Libraries
3. Problem Solving Paradigms
4. Graph
5. Mathematics
6. String Processing
7. Computational Geometry



Problem Solving Paradigms

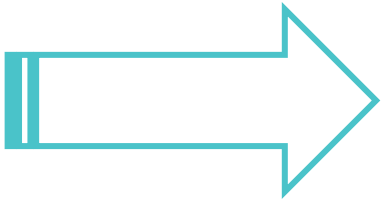
Chapter 3 in the textbook

Background Photo:
Abstract Cityscape

Chapter Three Outline

in the textbook

1. Complete Search = Brute Force
   a. Iterative Complete Search
   b. Recursive Complete Search
   c. Tips
2. Divide and Conquer
   a. Interesting Usages of Binary Search
3. Greedy
   a. Examples
4. Dynamic Programming
   a. DP Illustration
   b. Classical Examples
   c. Non-Classical Examples



Dynamic Programming

The key skills that you have to develop in order to master DP are the abilities to determine the problem states and to determine the relationships or transitions between current problems and their sub-problems.

Background Photo:
La Vague

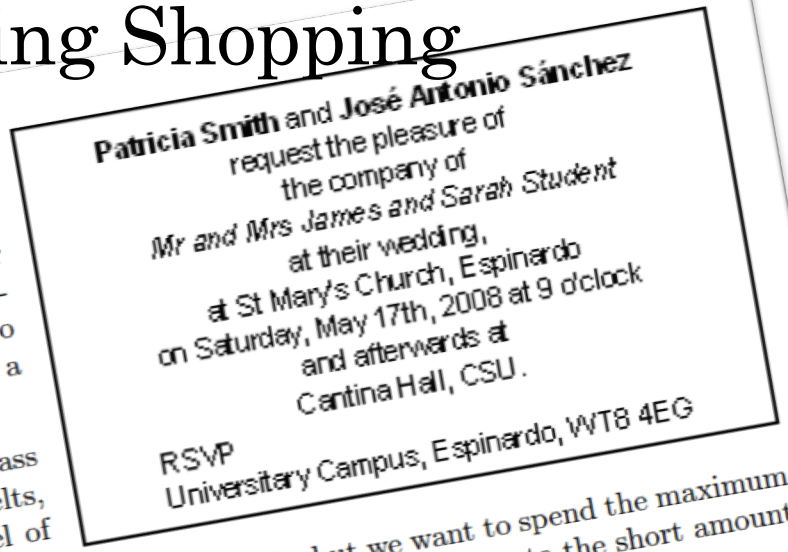
Dynamic Programming

- most challenging problem-solving technique among 4
- lots of recursion & recurrence relations!
- DP problems with small input size constraints may already be solvable with recursive backtracking
- top-down DP is a kind of intelligent or faster recursive backtracking
- DP is primarily used to solve optimization problems and counting problems
- if you encounter a problem that says
  - "minimize this" or
  - "maximize that" or
  - "count the ways to do that",
  - → high chance DP problem
- most DP problems in programming contests
  - only ask for optimal/total value
  - not for optimal solution itself
  - easier to solve by removing need to backtrack and produce solution
- however, some harder DP problems
  - also require optimal solution to be returned in some fashion

DP Illustration

Background Photo:
Bridge

UVa 11450 - Wedding Shopping



One of our best friends is getting married and we all are nervous because he is the first of us who is doing something similar. In fact, we have never assisted to a wedding, so we have no clothes or accessories, and to solve the problem we are going to a famous department store of our city to buy all we need: a shirt, a belt, some shoes, a tie, etcetera.

We are offered different models for each class of garment (for example, three shirts, two belts, four shoes, ...). We have to buy one model of each class of garment, and just one.

As our budget is limited, we cannot spend more money than it, but we want to spend the maximum possible. It's possible that we cannot buy one model of each class of garment due to the short amount of money we have.

Input

The first line of the input contains an integer, N, indicating the number of test cases. For each test case, some lines appear, the first one contains two integers, M and C, separated by blanks ($1 \leq M \leq 200$, and $1 \leq C \leq 20$), where M is the available amount of money and C is the number of garments you have to buy. Following this line, there are C lines, each one with some integers separated by blanks; in each of these lines the first integer, K ($1 \leq K \leq 20$), indicates the number of different models for each garment and it is followed by K integers indicating the price of each model of that garment.

...teger indicating the maximum amount of money ...initial amount of money. If there

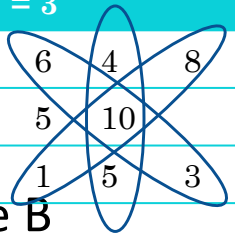
UVa 11450 - Wedding Shopping

- abridged problem statement:
  - given different options for each garment
    - 3 shirt models
    - 2 belt models
    - 4 shoe models . . .
  - & limited budget
  - task = buy 1 model of each garment
    - cannot spend more money than the given budget, but
    - we want to spend the maximum possible amount
- input
  - 2 integers M & C
    - M: budget $1 \le M \le 200$
    - C: #garments to buy $1 \le C \le 20$
  - followed by info about the C garments
    - for the garment $g \in [0..C-1]$
    - receive an integer $1 \le K \le 20$
    - # models for garment g
    - K int prices
- output
  - 1 int: max amount we can spend
  - if no solution due to small budget → "no solution"

UVa 11450 - Wedding Shopping

- test case A

M = 20 C = 3				
g0 = 3	6	4	8	
g1 = 2	5	10		
g2 = 4	1	5	3	5



19

- test case B

M = 9 C = 3				
g0 = 3	6	4	8	
g1 = 2	5	10		
g2 = 4	1	5	3	5

- no solution

- test case A
  - → answer = 19
  - 8+10+1 | 6+10+3 | 4+10+5
- test case B
  - → answer "no solution"
  - cheapest models total price
  - 4+5+1 = 10 > M= 9

to appreciate usefulness of DP:
→ first explore how far other approaches will get

Approach 1: Greedy (WA)

- test case A

M = 20 C = 3				
g0 = 3	6	4	8	
g1 = 2	5	10		
g2 = 4	1	5	3	5

19 ✓

- test case B

M = 9 C = 3				
g0 = 3	6	4	8	
g1 = 2	5	10		
g2 = 4	1	5	3	5

- test case C

no solution ✓

M = 12 C = 3				
g0 = 3	6	4	8	
g1 = 2	5	10		
g2 = 4	1	5	3	5

no solution ✗

- maximize the budget spent
- take most expensive model for each g which still fits our budget
- test case A
  - → take 8 (rest 12) then 10 (rest 2) then 1 → works
- test case B
  - → also works
- runs very fast:
  - 20+20+. . .+ 20 for a total of 20 times = 400 operations in the worst case
- test case C
  - incorrectly report "no solution"
  - 12 = 4+5+3
  - 12 = 6+5+1

Approach 2: Divide and Conquer (WA)

- not solvable using D&C paradigm
- because sub-problems are not independent

Approach 3: Complete Search (TL)

- recursive backtracking
  - function shop(money, g) 2 params:
  - current money & current garment g
  - (money, g) → state of this problem
    - order of params does not matter
  - return value: money spent
- start
  - money = M & garment g = 0
  - then try all possible models in garment g = 0 (max 20 models)
  - model i chosen
    - subtract price from money
    - repeat recursively g = 1 …
- stop at last garment g = C-1
- if money < 0 before end
  - prune infeasible solution
- among valid combinations
  - pick smallest >=0 money
  - max money spent = max return value
- CS recurrences: transitions: shop(money, g) =
  - money < 0 → $-\infty$
  - g = C → M - money
  - general case→ max(shop(money – price [g][model], g+1)) ∀ model ∈ [1..K] of current garment g
- slow! = 20^{20} >> 3s!

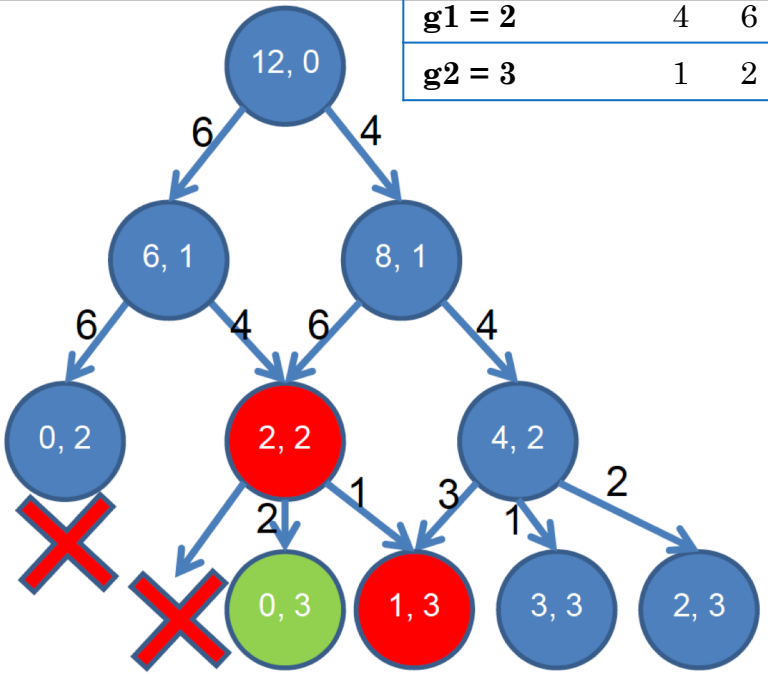
	0	1	2	…	19
g0					
g1					
g2					
…					
g19					

Is there Overlapping?

- case 1:
  - suppose that there are 2 models in a certain garment g with the same price p
  - CS same sub-problem shop(money - p, g + 1)
- case 2:
  - if at a garment g
  - some combination
  - money1 - p1 = money2 - p2
- search space !as big as 20^{20} because many sub-problems are overlapping!
- inefficient state of affairs

- example:
  - M=12

model garment	0	1	2
g0 = 2	6	4	
g1 = 2	4	6	
g2 = 3	1	2	3



Two Prerequisites Dynamic Programming

- ✓ 1. optimal sub structure
  - solution for sub-problem is part of solution of original problem
  - similar to Greedy!
- 2. overlapping sub problems
  - key characteristic of DP!
  - ✓ distinct sub problems are few
  - but they are repeatedly computed
    - ➢ different from Divide and Conquer

Dr Siba Haidar & Dr Antoun Yaacoub

Analyzing a basic DP solution

- how many distinct states (money, g)?
  - M = 201 x 20 = 4020
  - 201 possible values for money (0 - 200)
  - 20 possible values for the garment g
- each state just needs to be computed once → solve faster
- to compute each state = specific (money, garment)
  - iterate at most 20 models
  - K = 20
- time complexity:
  - M x K = 4020 x 20 = 80400 ops
  - per test case
  - a very manageable calculation

- easy
- memory space → O(M)
  - for M distinct states
- time complexity → O(kM)
  - if 1 state requires O(k) steps

DP Solution – Implementation

- There are two ways to implement DP:
  - Top-Down (td)
  - Bottom-Up (bu)

Dr Siba Haidar & Dr Antoun Yaacoub

Approach 4: Top-Down DP (AC)

- recursive backtracking + 2 steps:
  - 1. DP memo table
    - initialize with dummy values not used (-1)
    - table dimensions ≈ M = problem states
  - 2. recursive function
    - ask first: is this state computed before?
    - (a) yes
      - return value in DP memo table
      - O(1) memoization
    - (b) no
      - perform computation
      - & store value in DP memo table
      - then return
- code very similar to recursive backtracking:

```
const int MAX_gm = 30; const int MAX_M = 210;
int M, C, price[MAX_gm][MAX_gm];
int memo[MAX_gm][MAX_M];
int dp(int g, int money) {
  if (money < 0) return -1e9;
  if (g == C) return M-money;
  if (memo[g][money] != -1) return memo[g][money];
  int ans = -1;
  for (int k = 1; k <= price[g][0]; ++k)
    ans = max(ans, dp(g+1, money-price[g][k]));
  return memo[g][money] = ans;
}
int main() {
  int TC; scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (int g = 0; g < C; ++g) {
      scanf("%d", &price[g][0]);
      for (int k = 1; k <= price[g][0]; ++k)
        scanf("%d", &price[g][k]);
    }
    memset(memo, -1, sizeof memo);
    if (dp(0, M) < 0) printf("no solution\n");
    else              printf("%d\n", dp(0, M));
  }
  return 0;
}
```

1. initialize DP memo table

Dr Siba Haidar & Dr Antoun Yaacoub

```
const int MAX_gm = 30; const int MAX_M = 210;
int M, C, price[MAX_gm][MAX_gm];
int memo[MAX_gm][MAX_M];
int dp(int g, int money) {
  if (money < 0) return -1e9;
  if (g == C) return M-money;
  if (memo[g][money] != -1) return memo[g][money];
  int ans = -1;
  for (int k = 1; k <= price[g][0]; ++k)
    ans = max(ans, dp(g+1, money-price[g][k]));
  return memo[g][money] = ans;
}
int main() {
  int TC; scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (int g = 0; g < C; ++g) {
      scanf("%d", &price[g][0]);
      for (int k = 1; k <= price[g][0]; ++k)
        scanf("%d", &price[g][k]);
    }
    memset(memo, -1, sizeof memo);
    if (dp(0, M) < 0) printf("no solution\n");
    else              printf("%d\n", dp(0, M));
  }
  return 0;
}
```

2. recursive function

Dr Siba Haidar & Dr Antoun Yaacoub

```
const int MAX_gm = 30; const int MAX_M = 210;
int M, C, price[MAX_gm][MAX_gm];
int memo[MAX_gm][MAX_M];
int dp(int g, int money) {
  if (money < 0) return -1e9;
  if (g == C) return M-money;
  if (memo[g][money] != -1) return memo[g][money];
  int ans = -1;
  for (int k = 1; k <= price[g][0]; ++k)
    ans = max(ans, dp(g+1, money-price[g][k]));
  return memo[g][money] = ans;
}
int main() {
  int TC; scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (int g = 0; g < C; ++g) {
      scanf("%d", &price[g][0]);
      for (int k = 1; k <= price[g][0]; ++k)
        scanf("%d", &price[g][k]);
    }
    memset(memo, -1, sizeof memo);
    if (dp(0, M) < 0) printf("no solution\n");
    else              printf("%d\n", dp(0, M));
  }
  return 0;
}
```

recursive backtracking

memoization

Dr Siba Haidar & Dr Antoun Yaacoub

```
const int MAX_gm = 30; const int MAX_M = 210;
int M, C, price[MAX_gm][MAX_gm];
int memo[MAX_gm][MAX_M];
int dp(int g, int money) {
  if (money < 0) return -1e9;
  if (g == C) return M-money;
  int &ans = memo[g][money];
  if (ans != -1) return ans;
  for (int k = 1; k <= price[g][0]; ++k)
    ans = max(ans, dp(g+1, money-price[g][k]));
  return ans;]
}
int main() {
  int TC; scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (int g = 0; g < C; ++g) {
      scanf("%d", &price[g][0]);
      for (int k = 1; k <= price[g][0]; ++k)
        scanf("%d", &price[g][k]);
    }
    memset(memo, -1, sizeof memo);
    if (dp(0, M) < 0) printf("no solution\n");
    else              printf("%d\n", dp(0, M));
  }
  return 0;
}
```

can use local reference

Displaying the Optimal Solution(s)

```
void print_dp(int g, money) {
  if (g == C || money < 0) return;
  for (int k = 1; k <= price[g][0]; ++k)
    if (dp(g+1, money-price[g][k]) == memo[g][money]) {
      printf("%d - ", price[g][k]);
      print_dp(g+1, money-price[g][k]);
      break;
    }
}
```

same state as dp

find memo match

remove break to take all possible solutions

Approach 5: Bottom-Up DP (AC)

- basic steps:
- 1. determine params: state
- 2. if N params
  - prepare N-D DP table with 1 entry per state
  - ≈ memo table in td DP + differences:
    - in bu DP initialize → only some cells with known initial values: base cases
    - in td DP initialize → completely with dummy values to indicate not yet computed
- 3. determine cells/states that can be filled next
  - transitions
  - repeat until DP table complete
  - iterations = loops

- 1. state = cur g & cur money
  - reverse order make g first
  - → row indices of DP table
  - take advantage of cache-friendly row-major traversal in 2D
- 2. initialize 2D table 20X201
  - true if reachable[g][money]
  - only cells/states reachable by buying any of models of first garment g = 0 are set to true
  - → first row
- 3. use info of current row g to update values at next row g+1

Example Bottom-Up DP

- test case A

M = 20 C = 3				
g0 = 3	6	4	8	
g1 = 2	5	10		
g2 = 4	1	5	3	5

- cols 21 - 200 not shown
- init
  - g=0
    - 20-6 = 14 & 20-4 = 16 &20-8= 12
    - set to true

money =>

g = v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Dr Siba Haidar & Dr Antoun Yaacoub

Example Bottom-Up DP

- test case A

M = 20 C = 3				
g0 = 3	6	4	8	
g1 = 2	5	10		
g2 = 4	1	5	3	5

- cols 21 - 200 not shown
- init
  - g=0
    - 20-6 = 14 & 20-4 = 16 &20-8= 12
    - set to true

money =>

g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Dr Siba Haidar & Dr Antoun Yaacoub

Example Bottom-Up DP

- test case A

M = 20 C = 3				
g0 = 3	6	4	8	
g1 = 2	5	10		
g2 = 4	1	5	3	5

- cols 21 - 200 not shown
- init
  - g=0
    - 20-6 = 14 & 20-4 = 16 &20-8= 12
    - set to true

money =>

g = v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

Dr Siba Haidar & Dr Antoun Yaacoub

```
const int MAX_gm = 30;
const int MAX_M = 210;
int price[MAX_gm][MAX_gm];

bool reachable[MAX_gm][MAX_M];
int main() {
    int TC; scanf("%d", &TC);
    while (TC--) {
        int M, C; scanf("%d %d", &M, &C);
        for (int g = 0; g < C; ++g) {
            scanf("%d", &price[g][0]);
            for (int k = 1; k <= price[g][0]; ++k)
                scanf("%d", &price[g][k]);
        }
        memset(reachable, false, sizeof reachable);
        for (int k = 1; k <= price[0][0]; ++k)
            if (M-price[0][k] >= 0)
                reachable[0][M-price[0][k]] = true;
        int money;

        for (int g = 1; g < C; ++g)
            for (money = 0; money < M; money++)
                if (reachable[g-1][money])
                    for (int k = 1; k <= price[g][0]; ++k)
                        if (money-price[g][k] >= 0)
                            reachable[g][money-price[g][k]] = true;
        for (money = 0; money <= M && !reachable[C-1][money]; ++money);



        if (money == M+1) printf("no solution\n");
        else              printf("%d\n", M-money);
    }
    return 0;}
```

1. state (g, money)
2. initialize DP memo table

3. use info of row g-1 to update values at row g

Dr Siba Haidar & Dr Antoun Yaacoub

Space Saving Technique

- at each iteration
  - you only need two rows of the table:
    - previous g-1
    - current g
  - you can discard older ones
- just use a table with two rows
  - that you alteranate between iterations!
- code becomes

```
const int MAX_gm = 30;
const int MAX_M = 210;
int price[MAX_gm][MAX_gm];

bool reachable[2][MAX_M];
int main() {
    int TC; scanf("%d", &TC);
    while (TC--) {
        int M, C; scanf("%d %d", &M, &C);
        for (int g = 0; g < C; ++g) {
            scanf("%d", &price[g][0]);
            for (int k = 1; k <= price[g][0]; ++k)
                scanf("%d", &price[g][k]);
        }
        memset(reachable, false, sizeof reachable);
        for (int k = 1; k <= price[0][0]; ++k)
            if (M-price[0][k] >= 0)
                reachable[0][M-price[0][k]] = true;
        int money;
        int cur = 1;
        for (int g = 1; g < C; ++g) {
          memset(reachable[cur], false, sizeof reachable[cur]);
          for (money = 0; money < M; ++money)
            if (reachable[!cur][money])
              for (int k = 1; k <= price[g][0]; ++k)
                if (money-price[g][k] >= 0)
                  reachable[cur][money-price[g][k]] = true;
          cur = 1-cur;    }
        for (money = 0; money <= M && !reachable[!cur][money]; ++money);

        if (money == M+1) printf("no solution\n");
        else              printf("%d\n", M-money);
    }
    return 0;}
```

Displaying the Optimal Solution

- many DP problems request only for the value of the optimal solution
- required to print the optimal solution? 2 ways
- way 1: top down DP
  - already seen in this slide
- way 2: bottom-up DP approach
  - still applicable for top-down DPs
  - store the predecessor information at each state
  - if more than 1 optimal predecessors & have to output all
    - store in a list
  - once optimal final state
    - do backtracking and follow optimal transition(s) recorded at each state
    - until reach one of the base cases
- most problem authors usually set additional output criteria so that selected optimal solution is unique for easier judging

Top-Down or Bottom-Up?

common
both use table
tabular method: computation
technique involving a table

Top-Down

- pro
  - natural transformation from normal recursion
  - only compute sub problem when necessary
    - sometimes faster
- cons
  - slower if there are many sub problems due to recursive calls overhead
  - use exactly O(states) table size
    - ? ML = Memory Limit Exceeded

Bottom-Up

- pro
  - true form of DP
  - faster if many sub problems are visited: no recursive calls!
  - can save memory space (seen)
  - table filling order is topological order of the implicit DAG (later)
    - proper sequencing of nested loops
- cons
  - maybe not intuitive to those not inclined to recursions?
  - if there are M states, bu DP visits/fills value of all M states

Classical Examples

Classical problems with efficient DP solutions (states and transitions) should be mastered by every contestant.

Background Photo:
Bridge

DP Classical Examples

1. Max 1D Range Sum
   a. UVa 507 - Jill Rides Again
2. Max 2D Range Sum
   a. UVa 108 - Maximum Sum
3. Longest Increasing Subsequence (LIS)
4. 0-1 Knapsack| Subset Sum problem
5. Coin Change (CC) - The General Version
6. Traveling Salesman Problem (TSP)

DP Classical Examples

1. Max 1D Range Sum
   a. UVa 507 - Jill Rides Again
2. Max 2D Range Sum
   a. UVa 108 - Maximum Sum
3. Longest Increasing Subsequence (LIS)
4. 0-1 Knapsack| Subset Sum problem
5. Coin Change (CC) - The General Version
6. Traveling Salesman Problem (TSP)

Max 1D Range Sum: UVa 507 - Jill Rides Again

Jill likes to ride her bicycle, but since the pretty city of Greenhills where she lives has grown, Jill often uses the excellent public bus system for part of her journey. She has a folding bicycle which she carries with her when she uses the bus for the first part of her trip. When the bus reaches some pleasant part of the city, Jill gets off and rides her bicycle. She follows the bus route until she reaches her destination or she comes to a part of the city she does not like. In the latter event she will board the bus to finish her trip.

Through years of experience, Jill has rated each road on an integer scale of "niceness." Positive niceness values indicate roads Jill likes; negative values are used for roads she does not like. There are not zero values. Jill plans where to leave the bus and start bicycling, as well as where to stop bicycling and re-join the bus, so that the sum of niceness values of the roads she bicycles on is maximized. This means that she will sometimes cycle along a road she does not like, provided that it joins up two other parts of her journey involving roads she likes enough to compensate. It may be that no part of the route is suitable for cycling so that Jill takes the bus for its entire route. Conversely, it may be that the whole route is so nice Jill will not use the bus at all.

Since there are many different bus routes, each with several stops at which Jill could leave or enter the bus, she feels that a computer program could help her identify the best part to cycle for each bus route.

Input

The input file contains information on several bus routes. The first line of the file is a single integer b representing the number of route descriptions in the file. The identifier for each route (r) is the sequence number within the data file, $1 \leq r \leq b$. Each route description begins with the number of the route: an integer s, $2 \leq s \leq 20,000$ on a line by itself. The number of stops is followed by ($1 \leq i \leq s$) is an integer n_i representing Jill's assessment of the niceness of the

Max 1D Range Sum: UVa 507 - Jill Rides Again

- abridged problem statement
- given int array A with n non-zero ints
- $n \leq 20K$
- determine the maximum (1D) range sum of A
  - = find the maximum Range Sum Query (RSQ)
  - between two indices i and j in [0..n-1]
  - A[i] + A[i+1] + A[i+2] +...+ A[j]
- CS algo → $O(n^3)$
  - try all possible $O(n^2)$ pairs (i,j)
  - compute RSQ(i, j) in O(n)
  - pick max
  - n = 20K → TL solution
- recall
  - Segment Tree
  - Binary Indexed (Fenwick) Tree
- pre-process array A by computing
  - A[i] += A[i-1] $\forall i \in [1..n-1]$
  - so that A[i] contains sum of ints in A[0..i]
  - can now compute RSQ(i, j) in O(1):
    - RSQ(0, j) = A[j] and
    - RSQ(i, j) = A[j] - A[i-1] $\forall i > 0$
- → CS algo → $O(n^2)$
  - n = 20K → still TL
  - however useful in other cases

Jay Kadane's O(n) – Greedy | DP

```
int main() {
  int n = 9, A[] = { 4,-5, 4,-3, 4, 4,-4, 4,-5 };
  int sum = 0, ans = 0;
  for (int i = 0; i < n; ++i) {
    sum += A[i];
    ans = max(ans, sum);
    if (sum < 0) sum = 0;
  }
  printf("Max 1D Range Sum = %d\n", ans);
  return 0;
}
```

Joseph "Jay" Born Kadane is the Leonard J. Savage University Professor of Statistics, Emeritus in the Department of Statistics, Social and Decision Sciences at Carnegie Mellon University. Kadane is one of the early proponents of Bayesian statistics, particularly the subjective Bayesian philosophy.

index	0	1	2	3	4	5	6	7	8
array values	4	-5	4	-3	4	4	-4	4	-5

sum	~~0~~	~~4~~	~~-1~~	~~0~~	~~4~~	~~1~~	~~5~~	~~9~~	~~5~~	~~9~~	4
ans	~~0~~	~~4~~				~~5~~	9				
i		~~0~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~	~~6~~	~~7~~	~~8~~	9

Key idea:

- keep a running sum of the integers seen so far

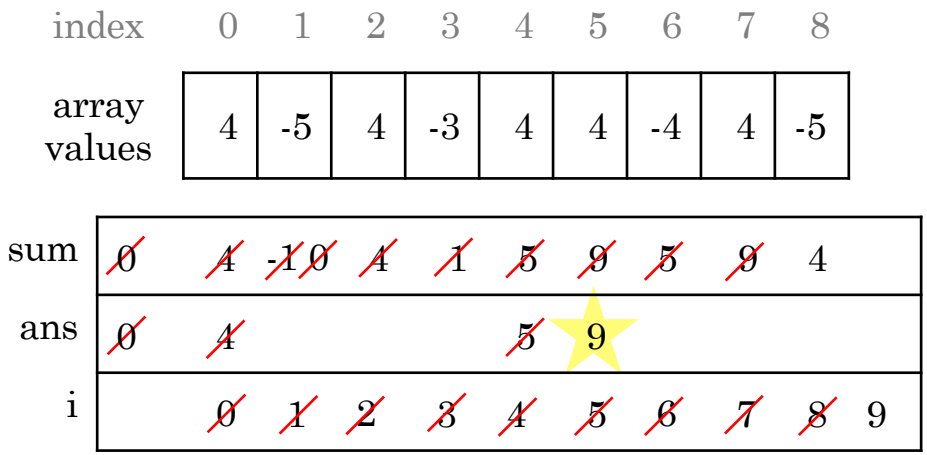
&

- greedily reset to 0 if it dips below 0

because re-starting from 0 is always better than continuing from a negative running sum

Jay Kadane's O(n) – Greedy | DP

```
int main() {
  int n = 9, A[] = { 4,-5, 4,-3, 4, 4,-4, 4,-5 };
  int sum = 0, ans = 0;
  for (int i = 0; i < n; ++i) {
    sum += A[i];
    ans = max(ans, sum);
    if (sum < 0) sum = 0;
  }
  printf("Max 1D Range Sum = %d\n", ans);
  return 0;
}
```



- can view it as a DP solution
- at each step, 2 choices:
  - either leverage previously accumulated max sum
  - or begin a new range
- dp(i) represents max sum of a range of ints that ends with element A[i]
- final answer is max over all values dp(i), i ∈ [0..n-1]
- if zero-length ranges are allowed
  - 0 must also be considered as a possible answer
- implementation efficient & utilizes space saving trick

Max 1D Range Sum: UVa 507 - Jill Rides Again

```
int main() {
    int testcase, cases = 0;
    int n, x;
    scanf("%d", &testcase);
    while(testcase--) {
        scanf("%d", &n);
        int tmp = 1, st = 0xffff, ed, sum = 0, ans = 0;
        for (int i = 2; i <= n; i++) {
            scanf("%d", &x);
            sum += x;
            if (sum < 0) sum = 0, tmp = i;
            if (sum >= ans) {
                if (sum > ans || (sum == ans && (i - tmp > ed - st))) {
                    st = tmp;
                    ed = i;
                }
                ans = sum;
            }
        }
        if (ans > 0)
            printf("The nicest part of route %d is between stops %d and %d\n", ++cases, st, ed);
        else
            printf("Route %d has no nice parts\n", ++cases);
    }
    return 0;
}
```

DP Classical Examples

1. Max 1D Range Sum
   a. UVa 507 - Jill Rides Again
2. Max 2D Range Sum
   a. UVa 108 - Maximum Sum
3. Longest Increasing Subsequence (LIS)
4. 0-1 Knapsack| Subset Sum problem
5. Coin Change (CC) - The General Version
6. Traveling Salesman Problem (TSP)

Max 2D Range Sum: UVa 108 - Maximum Sum

A problem that is simple to solve in one dimension is often much more difficult to solve in more than one dimension. Consider satisfying a boolean expression in conjunctive normal form in which each conjunct consists of exactly 3 disjuncts. This problem (3-SAT) is NP-complete. The problem 2-SAT is solved quite efficiently, however. In contrast, some problems belong to the same complexity class regardless of the dimensionality of the problem.

Given a 2-dimensional array of positive and negative integers, find the sub-rectangle with the largest sum. The sum of a rectangle is the sum of all the elements in that rectangle. In this problem the sub-rectangle with the largest sum is referred to as the *maximal sub-rectangle*.

A sub-rectangle is any contiguous sub-array of size 1×1 or greater located within the whole array. As an example, the maximal sub-rectangle of the array:

$$\begin{array}{rrrr} 0 & -2 & -7 & 0 \\ 9 & 2 & -6 & 2 \\ -4 & 1 & -4 & 1 \\ -1 & 8 & 0 & -2 \end{array}$$

is in the lower-left-hand corner:

$$\begin{array}{rr} 9 & 2 \\ -4 & 1 \\ -1 & 8 \end{array}$$

and has the sum of 15.

Input

The input consists of an $N \times N$ array of integers.

The input begins with a single positive integer N on a line by itself indicating the size of the square two dimensional array. This is followed by N^2 integers separated by white-space (newlines and spaces). These N^2 integers make up the array in row-major order (i.e., all numbers on the first row, left-to-right, then all numbers on the second row, left-to-right, etc.). N may be as large as 100. The numbers in the

the range $[-127, 127]$.

Max 2D Range Sum: UVa 108 - Maximum Sum

- Abridged problem statement:
  - n x n square matrix of integers A
  - $1 \le n \le 100$
  - each integer [-127..127]
  - find a sub-matrix of A with the maximum sum
- example:
  - 4 x 4 matrix (n = 4)

0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

  - 3 x 2 sub-matrix on lower-left with maximum sum of 15

- CS in $O(n^6)$ → TL(> 3s)

```
int main() {
  int n; scanf("%d", &n);
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      scanf("%d", &A[i][j]);
  int maxSubRect = -127*100*100;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      for (int k = i; k < n; ++k)
        for (int l = j; l < n; ++l) {
          int subRect = 0;
          for (int a = i; a <= k; ++a)
            for (int b = j; b <= l; ++b)
              subRect += A[a][b];
          maxSubRect = max(maxSubRect, subRect);
        }
  printf("%d\n", maxSubRect);
  return 0;
}
```

start coord

end coord

sum

Max 2D Range Sum: UVa 108 - Maximum Sum

- extend solution for Max 1D Range Sum
  - properly apply inclusion-exclusion principle
- deal with overlapping sub-matrices instead of overlapping sub-ranges
- turn n x n input matrix into cumulative sum matrix
  - A[i][j] = ∑ sub-matrix (0, 0) to (i, j)
  - done while reading input in $O(n^2)$
- sum sub-matrix (i, j) to (k, l) → O(1)

0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

0	-2	-9	-9
9	9	-4	-4
5	6	-11	-8
4	13	-4	-3

```
int n; scanf("%d", &n);
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j) {
    scanf("%d", &A[i][j]);
    if (i > 0) A[i][j] += A[i-1][j];
    if (j > 0) A[i][j] += A[i][j-1];
    if (i > 0 && j > 0) A[i][j] -= A[i-1][j-1];
  }
```

- ex sum of (1, 2) to (3, 3)
  - $\sum_{i=1}^{3}\sum_{j=2}^{3} v_{ij} = A[3][3] - A[0][3] - A[3][1] + A[0][1]$
  - $= A[3][3] - A[0][3] - A[3][1] + A[0][1]$
  - $= -3 - (-9) - 13 + (-2) = -9$
- problem solved in $O(n^4)$
  - n = 100 → fast enough

Max 2D Range Sum: UVa 108 - Maximum Sum

- extend solution for Max 1D Range Sum
  - properly apply inclusion-exclusion principle
- deal with overlapping sub-matrices instead of overlapping sub-ranges
- turn n x n input matrix into cumulative sum matrix
  - A[i][j] = ∑ sub-matrix (0, 0) to (i, j)
  - done while reading input in $O(n^2)$
- sum sub-matrix (i, j) to (k, l) → O(1)

0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

0	-2	-9	-9
9	9	-4	-4
5	6	-11	-8
4	13	-4	-3

```
int n; scanf("%d", &n);
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j) {
    scanf("%d", &A[i][j]);
    if (i > 0) A[i][j] += A[i-1][j];
    if (j > 0) A[i][j] += A[i][j-1];
    if (i > 0 && j > 0) A[i][j] -= A[i-1][j-1];
  }
int maxSubRect = -127*100*100;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      for (int k = i; k < n; ++k)
        for (int l = j; l < n; ++l) {
          int subRect = A[k][l];
          if (i > 0) subRect -= A[i-1][l];
          if (j > 0) subRect -= A[k][j-1];
          if (i > 0 && j > 0)
            subRect += A[i-1][j-1];
          maxSubRect = max(maxSubRect, subRect);}
```

start coord

end coord

sum O(1)

Dr Siba Haidar & Dr Antoun Yaacoub

Max 2D Range Sum: UVa 108 - Maximum Sum

- $O(n^3)$ 1D DP + greedy (Kadane's) solution, 0.000s

```
#define MAX_n 110
int A[MAX_n][MAX_n];
int main() {
  int n; scanf("%d", &n);
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) {
      scanf("%d", &A[i][j]);
      if (j > 0) A[i][j] += A[i][j-1];
    }
  int maxSubRect = -127*100*100;
  for (int l = 0; l < n; ++l)
    for (int r = l; r < n; ++r) {
      int subRect = 0;
      for (int row = 0; row < n; ++row) {
        if (l > 0)
          subRect += A[row][r] - A[row][l-1];
        else
          subRect += A[row][r];
        if (subRect < 0) subRect = 0;
        maxSubRect = max(maxSubRect, subRect);
      }
    }
  printf("%d\n", maxSubRect);
  return 0;
}
```

Kadane's algo on rows

restart if -ve

0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

0	-2	-9	-9
9	11	5	7
-4	-3	-7	-6
-1	7	7	5

(l,r) = (left,right)

row \ subRect	0,0	0,1	0,2	0,3	1,1	1,2	1,3	2,2	2,3	3,3
0	0	-2 0	-9 0	-9 0	-2 0	-9 0	-9 0	-7 0	-7 0	0
1	9	11	5	7	2	-4 0	-2 0	-6 0	-6 0	2
2	5	8	-2 0	1	3	-3 0	-2 0	-4 0	-4 0	3
3	4	15	7	6	9	8	6	0	-2 0	1

Conclusion about Max 1D & 2D Range Sum Problems

- not every range problem requires a Segment Tree or a Fenwick Tree
- static-input range-related problems are often solvable with DP techniques
- solution for a range problem natural to produce with bottom-up DP techniques as operand is already 1D | 2D array
  - can still write recursive top-down solution for a range problem
  - but not as natural

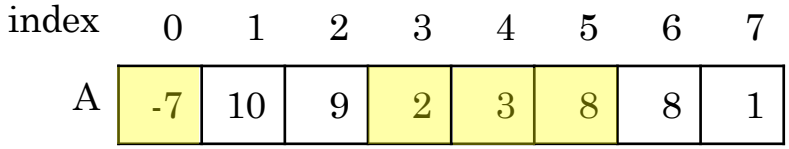
Dr Siba Haidar & Dr Antoun Yaacoub

DP Classical Examples

1. Max 1D Range Sum
   a. UVa 507 - Jill Rides Again
2. Max 2D Range Sum
   a. UVa 108 - Maximum Sum
3. Longest Increasing Subsequence (LIS)
4. 0-1 Knapsack| Subset Sum problem
5. Coin Change (CC) - The General Version
6. Traveling Salesman Problem (TSP)

Longest Increasing Subsequence (LIS)

- given a sequence
  - {A[0], A[1],..., A[n-1]},
- determine its LIS
  - Longest Increasing Subsequence
- subsequences not necessarily contiguous
- example:
  - n = 8, A, length-4 LIS



- CS
  - enumerates all possible subsequences
  - → too slow
  - $O(2^n)$ possible subsequences

Longest Increasing Subsequence (LIS)

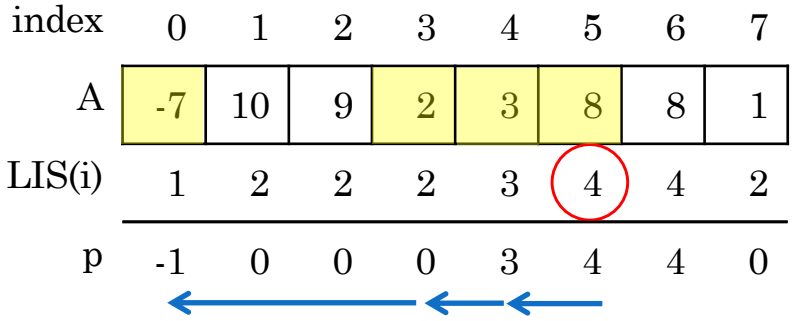
- DP
  - pb state → 1 param: i
- let LIS(i) be the LIS ending at index i
  - LIS(0) = 1
  - LIS(i) = 1 + LIS(j)
    - j < i & A[j] < A[i] & LIS(j) is largest
- overlapping sub-problems
  - to compute LIS(i) need to compute LIS(j) ∀j ∈ [0..i-1]
- but only n distinct states & each state needs O(n) loop
- → DP algo $O(n^2)$

index	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

```
const int MAX_N = 100010;
int n = 12, A[] = {-7, 10, 9, 2, 3, 8, 8, 1};

int memo[MAX_N];

int LIS(int i) {
  if (i == 0) return 1;
  int &ans = memo[i];
  if (ans != -1) return ans;
  ans = 0;
  for (int j = 0; j < i; ++j)
    if (A[j] < A[i])
      ans = max(ans, LIS(j)+1);
  return ans;
}
int main() {
  memset(memo, -1, sizeof memo);
  printf("LIS length is %d\n\n", LIS(n-1));
  return 0;
}
```

Longest Increasing Subsequence (LIS)

- display LIS solution(s)?
  - reconstruct by storing predecessor info & tracing
- example
  - LIS(5) → LIS(4) → LIS(3) → LIS(0)
  - optimal solution
    - index {0, 3, 4, 5} = {-7, 2, 3, 8}



LIS – O(nlogk)

- 2000 → accept O(n^2)
- 2020 → only accept O(n log k)
- can be solved using output-sensitive
  - O(n log k) greedy + D&C algo instead of O(n^2) where k length of LIS
- maintain a sorted array
  - amenable to binary search
- Let array L be an array such that
  - L(i) represents the smallest ending value of all length-i LISs found so far
  - complicated def but easy to see that
  - L(i-1) < L(i)
- →can binary search array L to determine the longest possible subsequence we can create by appending current element A[i]
- simply find the index of the last element in L that is less than A[i]
- answer is largest length of sorted array L at end of process

index	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

L								
	-7							
	-7	10						
	-7	9						
	-7	2						
	-7	2	3					
	-7	2	3	8				
	-7	2	3	8				
	-7	1	3	8				

LIS – O(nlogk)

index	0	1	2	3	4	5	6	7	8	9	10	11
A	-7	10	9	2	3	8	8	1	2	3	4	99

L	-7 0											
L_id	-7 0	10 1										
	-7 0	9 3										
	-7 0	2 3										
	-7 0	2 3	3 4									
	-7 0	2 3	3 4	8 5								
	-7 0	2 3	3 4	8 5								
	-7 0	1 7	3 4	8 5								
	-7 0	1 7	2 8	8 5								
	-7 0	1 7	2 8	3 9								
	-7 0	1 7	2 8	3 9	4 10							
	-7 0	1 7	2 8	3 9	4 10	99 11						

- → This is a greedy strategy:
- → By storing the LIS with smaller ending value, we maximize our ability to further extend the LIS with future values.

Dr Siba Haidar & Dr Antoun Yaacoub

LIS – O(nlogk)

lower_bound: returns an iterator pointing to the first element in the sorted range [first,last) which does not compare less than value.

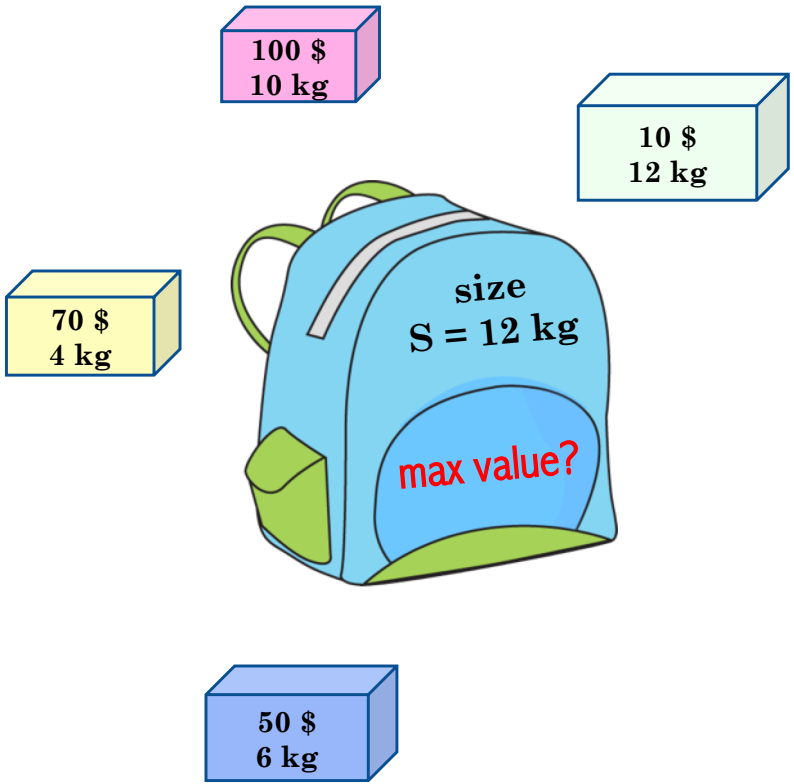
O(logk)

```
typedef vector<int> vi;
const int MAX_N = 100010;
int n = 12, A[] = {-7, 10, 9, 2, 3, 8, 8, 1, 2, 3, 4, 99};


vi p;
int main() {
    int k = 0, lis_end = 0;
    vi L(n, 0), L_id(n, 0);
    p.assign(n, -1);

    for (int i = 0; i < n; ++i) {
        int pos = lower_bound(L.begin(), L.begin()+k, A[i]) - L.begin();
        L[pos] = A[i];
        L_id[pos] = i;
        p[i] = pos ? L_id[pos-1] : -1;
        if (pos == k) {
            k = pos+1;
            lis_end = i;
        }
    }

    printf("Final LIS is of length %d. ", k);
    return 0;
}
```

DP Classical Examples

1. Max 1D Range Sum
   a. UVa 507 - Jill Rides Again
2. Max 2D Range Sum
   a. UVa 108 - Maximum Sum
3. Longest Increasing Subsequence (LIS)
4. 0-1 Knapsack| Subset Sum problem
5. Coin Change (CC) - The General Version
6. Traveling Salesman Problem (TSP)

0-1 Knapsack | Subset Sum problem



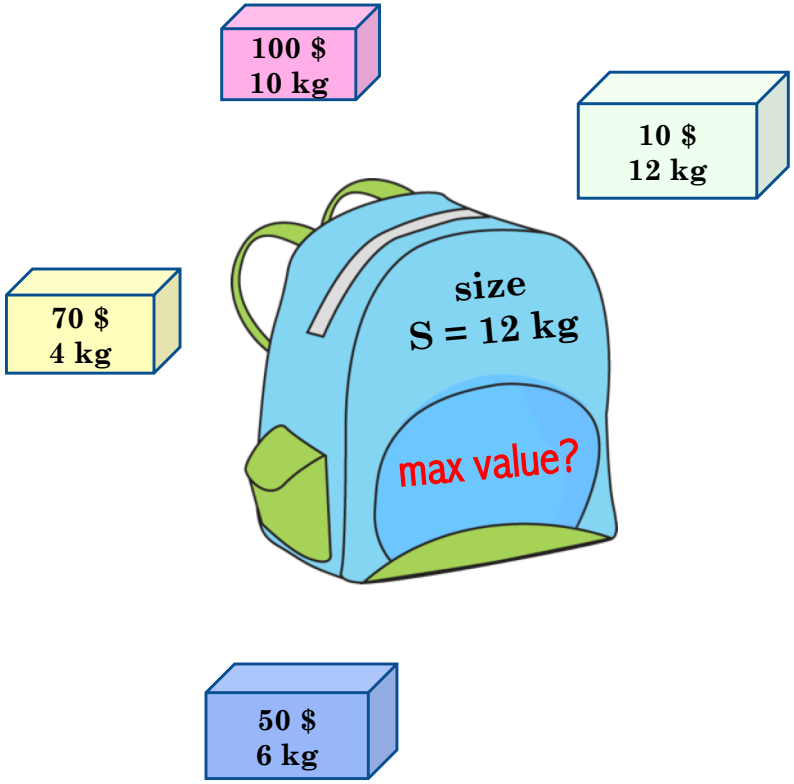
- given n items & max knapsack size S
- & value Vi & weight Wi, ∀i ∈ [0..n-1]
- ? max value we can carry
  - can either ignore (0) or take (1) any item

	0	1	2	3	
V	100	70	50	10	N=4
W	10	4	6	12	S=12

					total weight	total value
case 1	✓				10	100
case 2		✓	✓		10	120
case 3				✓	12	10

- other pb variants
  - given a set of ints & int S, is there a (non-empty) subset that has a sum equal to S?
  - fractional Knapsack problem with Greedy solution

0-1 Knapsack | Subset Sum problem



- solution: CS recurrences
- val(id, remW) =
  - remW = 0 → 0
  - id=n → val(n, remW) = 0
  - W[id] > remW →
    - val(id + 1, remW)
  - W[id] ≤ remW → max of
    - val(id + 1, remW)
    - V[id] + val(id + 1, remW - W[id])
- start calling val(0, S)
- overlapping sub-problems
- DP O(NS)
  - M = NS
  - k = 1
- here tdDP is faster than buDP
  - only required states executed
- if S large, NS >> 1M
  - → DP not feasible even with space saving trick!

0-1 Knapsack | Subset Sum
Example UVa 10130 - SuperSale

There is a SuperSale in a SuperHiperMarket. Every person can take only one object of each kind, i.e. one TV, one carrot, but for extra low price. We are going with a whole family to that SuperHiperMarket. Every person can take as many objects, as he/she can carry out from the SuperSale. We have given list of objects with prices and their weight. We also know, what is the maximum weight that every person can stand. What is the maximal value of objects we can buy at SuperSale?

Input

The input consists of T test cases. The number of them ($1 \leq T \leq 1000$) is given on the first line of the input file. Each test case begins with a line containing a single integer number N that indicates the number of objects ($1 \leq N \leq 1000$). Then follows N lines, each containing two integers: P and W. The first integer ($1 \leq P \leq 100$) corresponds to the price of object. The second integer ($1 \leq W \leq 30$) corresponds to the weight of object. Next line contains one integer ($1 \leq G \leq 100$) its the number of people in our group. Next G lines contains maximal weight ($1 \leq MW \leq 30$) that can stand this i-th person from our family ($1 \leq i \leq G$).

Output

For every test case your program has to determine one integer. Print out the maximal value of goods which we can buy with that family.

Sample Input

0-1 Knapsack | Subset Sum Example UVa 10130 - SuperSale

- Top-Down approach (faster in this problem)
  - only required states executed
  - example for 50 random test cases 20x less ops!

V	W		0	1	2	…	remW-wi	…	…	remW	…	MW
v0	w0	0	-1	-1	-1	…	-1	…	…	-1	…	-1
v1	w1	1	-1									
v2	w2	2	-1									
v3	w3	3	-1									
…	…	…	…	…	…	…		…	…	…	…	…
vi	wi	i	-1							=max(dp(i+1, remW),Vi+dp(i+1, remW-wi))		
vi+1	wi+1	i+1	-1									
…	…	…	…	…	…	…		…	…	…	…	…
vn-1	wn-1	n-1	-1									
-	-	n	-1									

0-1 Knapsack | Subset Sum Example UVa 10130 - SuperSale

- Bottom Up approach (slower in this problem)
  - all states must be calculated
  - answer in C[N][MW]

V	W		0	1	2	…	w-wi	…	…	w	…	MW
-	-	0	0	0	0	…	0	…	…	0	…	0
v1	w1	1	0									
v2	w2	2	0									
v3	w3	3	0									
…	…	…	…	…	…	…		…	…	…	…	…
vi-1	wi-1	i-1	0				C[i-1][w-wi]			C[i-1][w]		
vi	wi	i	0							= max (C[i-1][w], vi + C[i-1][w-wi])		
…	…	…	…	…	…	…		…	…	…	…	…
vn-1	wn-1	n-1	0									
vn	wn	n	0									

Dr Siba Haidar & Dr Antoun Yaacoub

0-1 Knapsack | Subset Sum Example

UVa 10130 - SuperSale

```
// 0-1 Knapsack DP (Top-Down, faster)
const int MAX_N = 1010;
const int MAX_W = 40;
int N, V[MAX_N], W[MAX_N], memo[MAX_N][MAX_W];
int dp(int id, int remW) {
  if ((id == N) || (remW == 0)) return 0;
  int &ans = memo[id][remW];
  if (ans != -1) return ans;
  if (W[id] > remW) return ans = dp(id+1,remW);
  return ans = max(dp(id+1, remW),
                   V[id]+dp(id+1, remW-W[id]));
}
int main() {
  int T; scanf("%d", &T);
  while (T--) {
    memset(memo, -1, sizeof memo);
    scanf("%d", &N);
    for (int i = 0; i < N; ++i)
      scanf("%d %d", &V[i], &W[i]);
    int ans = 0;
    int G; scanf("%d", &G);
    while (G--) {
      int MW; scanf("%d", &MW);
      ans += dp(0, MW); }
    printf("%d\n", ans);
  }
  return 0;}
```

```
// 0-1 Knapsack DP (Bottom-Up)
const int MAX_N = 1010;
const int MAX_W = 40;
int V[MAX_N], W[MAX_N], C[MAX_N][MAX_W];
int main() {
  int T; scanf("%d", &T);
  while (T--) {
    int N; scanf("%d", &N);
    for (int i = 1; i<= N; ++i)
      scanf("%d %d", &V[i], &W[i]);
    int ans = 0;
    int G; scanf("%d", &G);
    while (G--) {
      int MW; scanf("%d", &MW);
      for (int i = 0; i <= N;  ++i) C[i][0] = 0;
      for (int w = 0; w <= MW; ++w) C[0][w] = 0;
      for (int i = 1; i <= N; ++i)
        for (int w = 1; w <= MW; ++w) {
          if (W[i] > w) C[i][w] = C[i-1][w];
          else          C[i][w] = max(C[i-1][w],
V[i] + C[i-1][w-W[i]]);
        }
      ans += C[N][MW];
    }
    printf("%d\n", ans);
  }
  return 0;}
```

Dr Siba Haidar & Dr Antoun Yaacoub

DP Classical Examples

1. Max 1D Range Sum
   a. UVa 507 - Jill Rides Again
2. Max 2D Range Sum
   a. UVa 108 - Maximum Sum
3. Longest Increasing Subsequence (LIS)
4. 0-1 Knapsack| Subset Sum problem
5. Coin Change (CC) - The General Version
6. Traveling Salesman Problem (TSP)

Coin Change (CC) - The General Version

- problem:
  - given target amount V cents
  - list of denominations for n coins
  - coinValue[i], i ∈ [0..n-1]
  - ? min number of coins to use to represent V ?
  - assume unlimited supply
  - seen in greedy section
- concern: Greedy | DP?
  - it depends on the coin systems
  - canonical → greedy
  - arbitrary → DP

Example 1

	0	1	V=10
coinValue	1	5	N=2

	# coins
A	10		10
B	5	1	6
C		2	2

Example 2

	0	1	2	3	V=7
coinValue	1	3	4	5	N=4

					# coins
A	2			1	3
B		1	1		2

Coin Change (CC) - The General Version

- DP Top-Down memo
- state = (value)
  - rem amount to represent
- change(value) =
  - value = 0 → 0
  - value < 0 → ∞
  - 1 + min(change(value - coinValue[i])) ∀i ∈ [0..n-1]
- → O(VN)
- start by calling change (V)

```
int change(int value) {
  if (value == 0) return 0;
  if (value < 0) return INT_MAX-1;
  int &ans = memo[value];
  if (ans != -1) return ans;
  ans=INT_MAX;
  for(int i=0;i<N;i++)
    ans = min (ans,1+change(value-coinValue[i]));
  return ans;
}
```

Example 1

i 0 1 V=10

coinValue 1 5 N=2

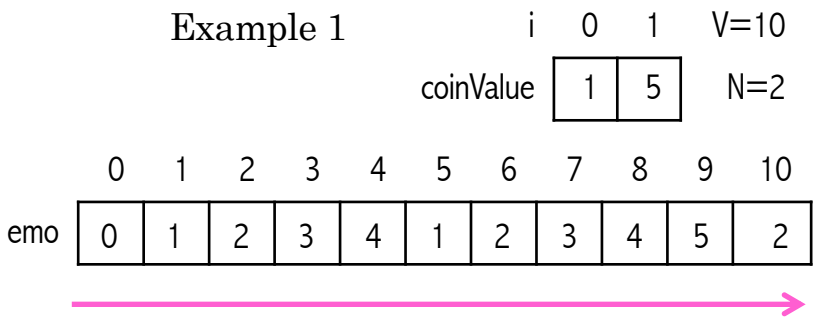
	0	1	2	3	4	5	6	7	8	9	10
memo	0	1	2	3	4	1	2	3	4	5	2
						?				?	?

state	change(state)
10	= 1+ min (change(10 -1), change(10-5)) = 1+ min (change(9), change(5))
9	= 1 + min(change(8), change(4))
8	= 1 + min(change(7), change(3))
7	= 1 + min(change(6), change(2))
6	= 1 + min(change(5), change(1))
5	= 1 + min(change(4), change(0))
4	= 1 + min(change(3), change(-1))
3	= 1 + min(change(2), change(-2))
2	= 1 + min(change(1), change(-3))
1	= 1 + min(change(0), change(-5))

Coin Change (CC) - The General Version

- Bottom-Up DP
  - O(VN)
- more logical than TD

```
int main() {
  while (scanf("%d", &V) != EOF){
    memset(memo, INT_MAX, sizeof memo);
    memo[0]=0;
    for (int i=1;i<=V;i++){
      memo[i] = INT_MAX-1;
      for(int j=0;j<N;j++){
        if (i - c[j] >= 0)
          memo[i] = min(memo[i],1+memo[i-c[j]]);
      }
    }
    printf("%d\n", memo[V]);
  }
  return 0;
}
```



CC # Ways: UVa 674 - Coin Change

Suppose there are 5 types of coins: 50-cent, 25-cent, 10-cent, 5-cent, and 1-cent. We want to make changes with these coins for a given amount of money.

For example, if we have 11 cents, then we can make changes with one 10-cent coin and one 1-cent coin, two 5-cent coins and one 1-cent coin, one 5-cent coin and six 1-cent coins, or eleven 1-cent coins. So there are four ways of making changes for 11 cents with the above coins. Note that we count that there is one way of making change for zero cent.

Write a program to find the total number of different ways of making changes for any amount of money in cents. Your program should be able to handle up to 7489 cents.

Input

The input file contains any number of lines, each one consisting of a number for the amount of money in cents.

Output

For each input line, output a line containing the number of different ways of making changes with the above 5 types of coins.

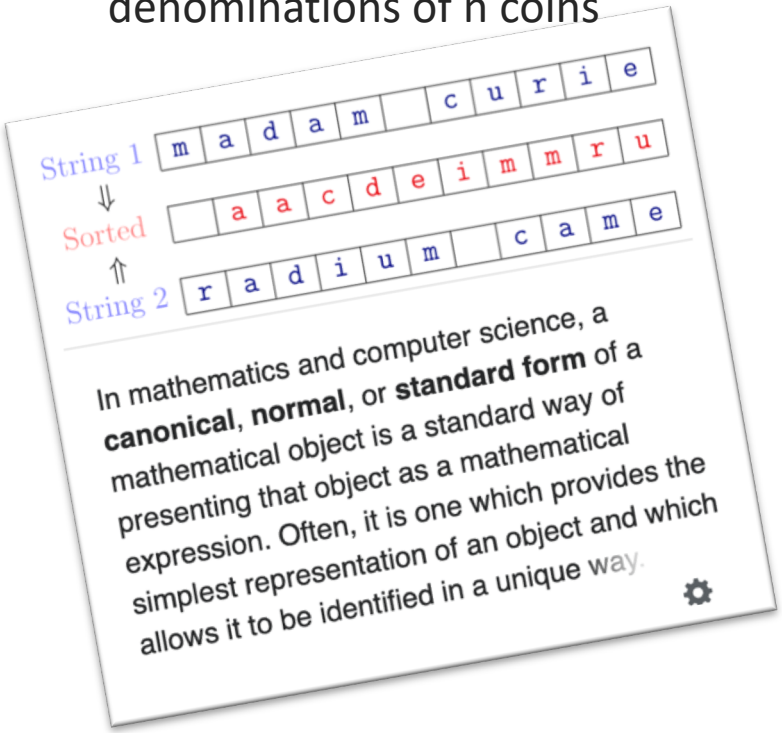
Sample Input

```
11
26
```

Sample Output

CC # Ways: UVa 674 - Coin Change

- variant:
  - count # canonical ways to get value V cents using list of denominations of n coins



- CS recurrence relation:
  - ways(type, value)
  - type: index of current coin
  - to avoid double-counting
- O(NV) = O(NV x 1)
- Top-Down

```
int N = 5, V, coinValue[5] = {1, 5, 10, 25, 50},
    memo[6][7500];
int ways(int type, int value) {
  if (value == 0) return 1;
  if ((value < 0) || (type == N)) return 0;
  int &ans = memo[type][value];
  if (ans != -1) return ans;
  return ans = ways(type+1, value) +
               ways(type, value-coinValue[type]);
}
int main() {
  memset(memo, -1, sizeof memo);
  while (scanf("%d", &V) != EOF)
    printf("%d\n", ways(0, V));
  return 0;}
```

only once cause constant coin system

CC # Ways: UVa 674 - Coin Change

- Bottom-Up
  - O(NV)
  - fill once, use many

```
void possibleWays() {
  for(int i = 1; i <= N; i++) {
    memo[i][0] = 1;
    for(int s = 1; s <= 7489; s++) {
      memo[i][s] = memo[i - 1][s];
      if(s-c[i-1]>=0) memo[i][s] += memo[i][s-c[i-1]];
    }
  }
}


int main() {
  memset(memo, 0, sizeof memo);
  possibleWays();
  while (scanf("%d", &V) != EOF){
    printf("%d\n", memo[N][V]);
  }
  return 0;
}
```

CC # Ways: UVa 674 - Coin Change

- Bottom-Up + Space Optimization
- every row uses only the previous → 1D memo

```
int N = 5, V,
    c[5] = {1, 5, 10, 25, 50},
    memo[7500];

void possibleWays() {
  memo[0] = 1;
  for(int i = 1; i <= N; i++) {
    for(int s = 1; s <= 7489; s++) {
      if(s-c[i-1]>=0) memo[s] += memo[s-c[i-1]];
    }
  }
}
int main() {
  memset(memo, 0, sizeof memo);
  possibleWays();
  while (scanf("%d", &V) != EOF){
    printf("%d\n", memo[V]);
  }
  return 0;
}
```

CC # Ways: Limited Supply

- only one piece 1 of each coin type
- if additional cond: "a coin cannot be used several times"

```
int N = 5, V,
    c[5] = {1, 5, 10, 25, 50},
    memo[7500];
void possibleWays() {
  memo[0] = 1;
  for(int i = 1; i <= N; i++) {
    for(int s = 7489; s >=1 ; s--) {
      if(s-c[i-1]>=0) memo[s] += memo[s-c[i-1]];
    }
  }
}
int main() {
  memset(memo, 0, sizeof memo);
  possibleWays();
  while (scanf("%d", &V) != EOF){
    printf("%d\n", memo[V]);
  }
  return 0;
}
```

if canonical coin system, # is always 0 or 1

loop in reverse order

DP Classical Examples

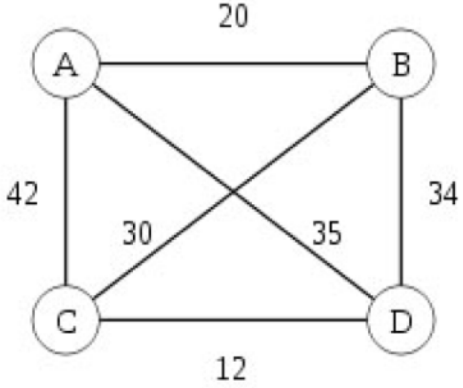
1. Max 1D Range Sum
   a. UVa 507 - Jill Rides Again
2. Max 2D Range Sum
   a. UVa 108 - Maximum Sum
3. Longest Increasing Subsequence (LIS)
4. 0-1 Knapsack| Subset Sum problem
5. Coin Change (CC) - The General Version
6. Traveling Salesman Problem (TSP)

YouTube

VISU ALGO
visualising data structures and algorithms

Traveling Salesman Problem (TSP)

- given n cities + pairwise distances: $dist$ of size $n \times n$
- ? → compute **min cost** of making a Hamiltonian tour
  - starts from any city s; goes through all the other $n-1$ cities exactly once; returns to the starting city s
- example $4! = 24\ tours$
  - ABCDA→ 97

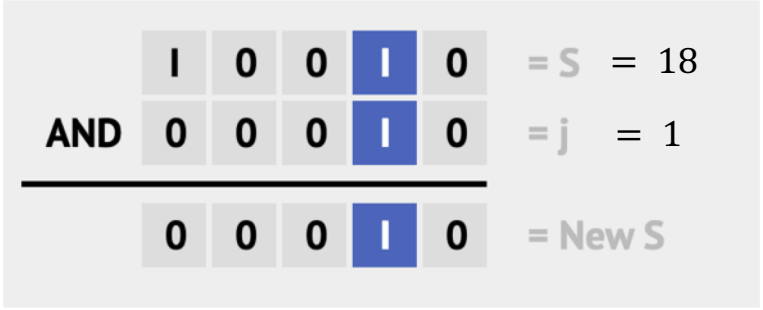


	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

- brute force solution
  - $O((n-1)!)$ fix 1st → symmetry
  - effective for $n \leq 12, 11! \approx 40M$
- overlap
  - is obvious → DP
  - if 4 or more (ex: A .. Z, n=26)
  - ABCD (n−4) other cities overlaps ACBD (n−4) other cities
- state
  - → 2 params:
  - last city/vertex visited pos
  - subset of visited cities

Traveling Salesman Problem (TSP)

- efficient set representation?
  - bitmask
  - n cities → binary int of length n
- example:
  - mask= $18_{10} = 10010_2$
  - cities {1, 4} are in the set

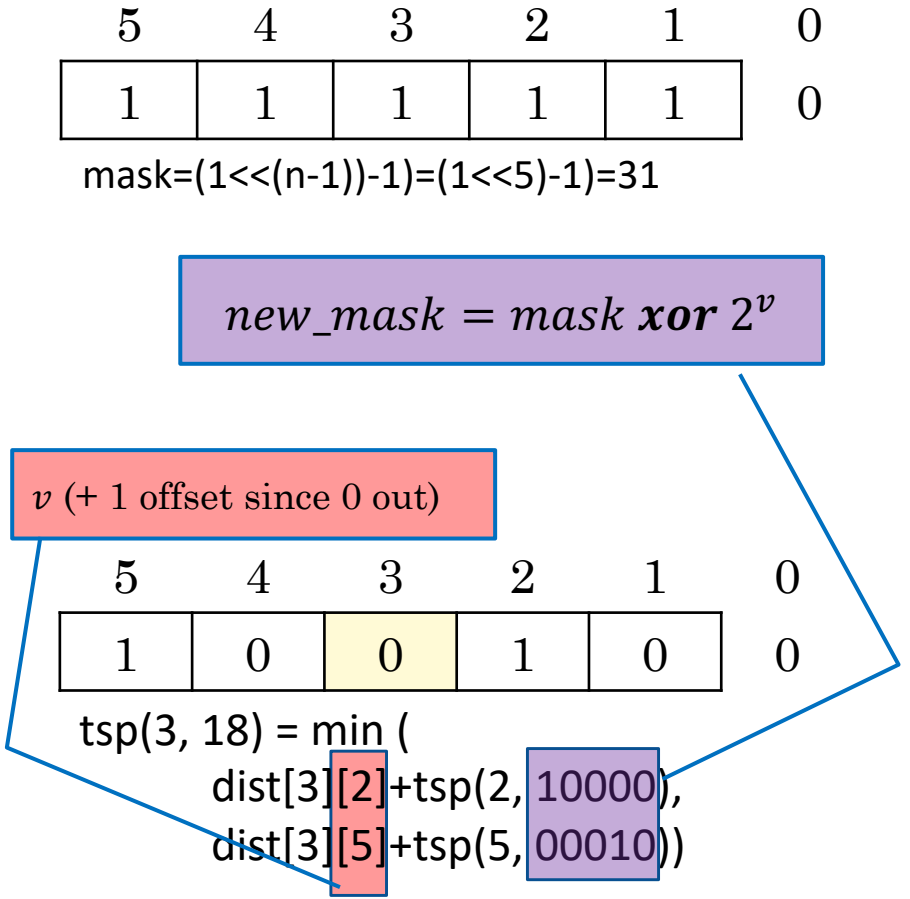


- check bit j → & (1 << j)
- set bit j → mask |= (1 << j)
- https://visualgo.net/en/bitmask

- bitmask → not visited cities
- CS recurrence relations:
  1. tsp(pos, 0) = dist[pos][0]
  2. tsp(pos, mask) =
     - $\min(dist[pos][v] + tsp(v, newmask))$
     - $\forall v$ available city not visited yet
- overall time complexity
  - $O(kM) = O(2^n \times n^2)$
  - $M = n \times 2^n$ distinct states
  - $k = n$ each state
- allows to solve up to n ≈ 16
  - $2^{16} \times 16^2 \approx 17M$
  - $2^{17} \times 17^2 \approx 132M$
- small improvement but exact solution
- for n++ → non-exact approaches

TSP - How to solve one state?

- state = (current_city, mask)
- example:
- consider 6 cities
  - start city 0 + 5 cities
  - omit first city: default start
  - begin by calling tsp (0, 11111)
  - all 5 cities are available = not visited yet
- then …
- suppose we reach a random state
  - (city=3, mask= $18_{10} = 10010_2$)
  - visited cities 0,1,4 and currently 3
  - still have to visit the cities 2 and 5
  - which order gives min tsp?
    - 2 then 5 or 5 then 2?



TSP - How to solve one state?

- obtain v quickly?

```
#define LSOne(S) ((S) & -(S))
int two_pow_v = LSOne(m);
int v =__builtin_ctz(two_pow_v)+1;
```

- built-in function:
  - int __builtin_ctz (unsigned int x)
  - returns the number of trailing 0-bits in x, starting at the least significant bit position
  - x = 0 → result undefined

- new_mask

```
new_mask = mask^two_pow_v;
```

- beging from right to left
  - find v=2 then find v=5

```
m -= two_pow_v;
```

m	two_pow_v	v
16 = 10000	16	5
12 = 01100	4	3
17 = 10001	1	1

$$new\_mask = mask \; \mathbf{xor} \; 2^v$$

v (+ 1 offset since 0 out)

5	4	3	2	1	0
1	0	0	1	0	0

tsp(3, 18) = min (
dist[3][2]+tsp(2, 10000),
dist[3][5]+tsp(5, 00010))

TSP: UVa 10496 - Collecting Beepers

Karel is a robot who lives in a rectangular coordinate system where each place is designated by a set of integer coordinates (x and y). Your job is to design a program that will help Karel pick up a number of beepers that are placed in her world. To do so you must direct Karel to the position where each beeper is located. Your job is to write a computer program that finds the length of the shortest path that will get Karel from her starting position, to each of the beepers, and return back again to the starting position.

Karel can only move along the x and y axis, never diagonally. Moving from one position (i, j) to an adjacent position $(i, j+1)$, $(i, j-1)$, $(i-1, j)$, or $(i+1, j)$ has a cost of one.

You can assume that Karel's world is never larger than 20×20 squares and that there will never be more than 10 beepers to pick up. Each coordinate will be given as a pair (x, y) where each value will be in the range 1 through the size of that particular direction of the coordinate system.

Input

First there will be a line containing the number of scenarios you are asked to help Karel in. For each scenario there will first be a line containing the size of the world. This will be given as two integers x-size and y-size). Next there will be one line containing two numbers giving the starting position of Karel. On the next line there will be one line containing two numbers giving the number of beepers. For each beeper there will be a line containing two numbers giving the coordinates of each beeper.

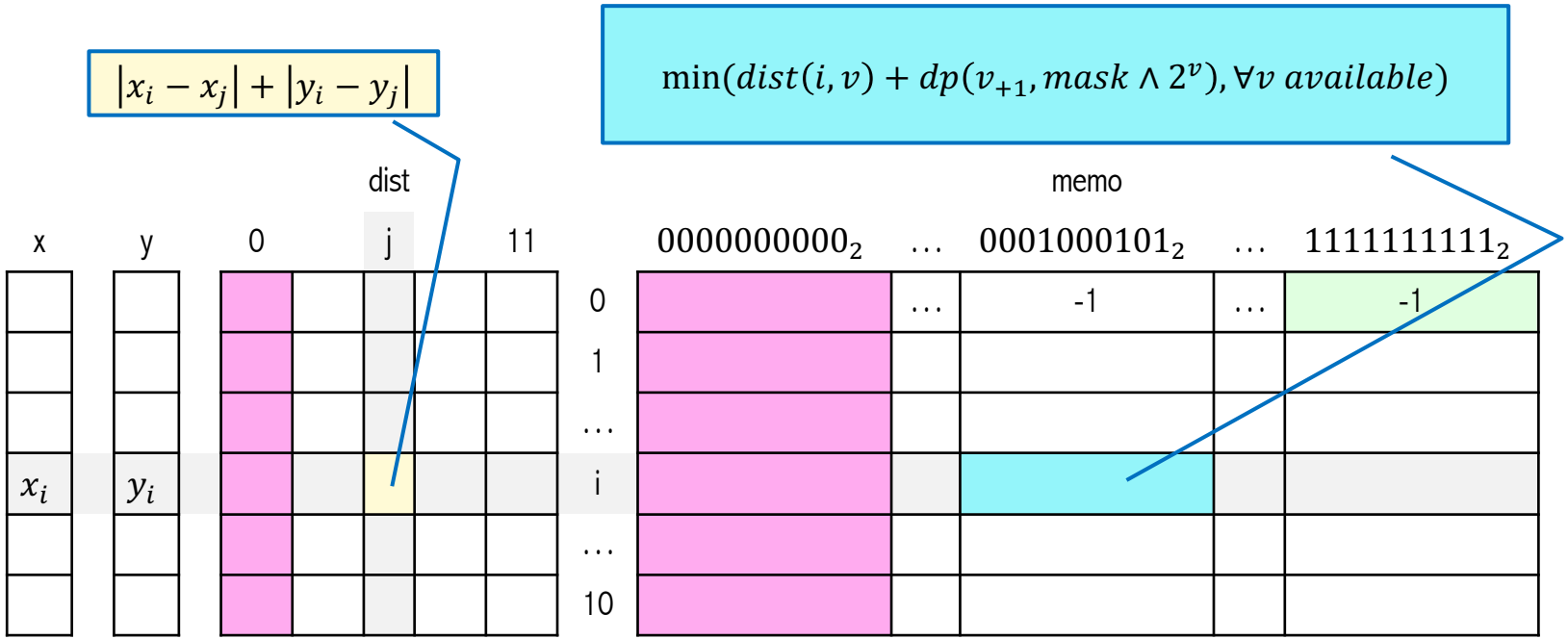
Output

The output will be one line per scenario, giving the minimum distance that Karel has to move to get from her starting position to each of the beepers and back again to the starting position.

Sample Input

TSP: UVa 10496 - Collecting Beepers

- start green → all available
- base case pink (mask = 0 none available = all visited)
- filling order is then random (upon need in recursion)



```cpp
#define LSOne(S) ((S) & -(S))
const int MAX_n = 11;
int dist[MAX_n][MAX_n], memo[MAX_n][1<<(MAX_n-1)];
int dp(int u, int mask) {
  if (mask == 0) return dist[u][0];
  int &ans = memo[u][mask];
  if (ans != -1) return ans;
  ans = 2000000000;
  int m = mask;
  while (m) {
    int two_pow_v = LSOne(m);
    int v = __builtin_ctz(two_pow_v)+1;
    ans = min(ans, dist[u][v] + dp(v, mask^two_pow_v));
    m -= two_pow_v;
  }
  return ans;
}
int main() {
  int TC; scanf("%d", &TC);
  while (TC--) {
    int xsize, ysize; scanf("%d %d", &xsize, &ysize); // not used
    int x[MAX_n], y[MAX_n];
    scanf("%d %d", &x[0], &y[0]);
    int n; scanf("%d", &n); ++n;
    for (int i = 1; i < n; ++i)
      scanf("%d %d", &x[i], &y[i]);
    for (int i = 0; i < n; ++i)
      for (int j = i; j < n; ++j)
        dist[i][j] = dist[j][i] = abs(x[i]-x[j]) + abs(y[i]-y[j]); // Manhattan dista
    memset(memo, -1, sizeof memo);
    printf(" The shortest path has length %d\n", dp(0, (1<<(n-1))-1));
  }
  return 0;
}
```

	1D RSQ	2D RSQ	LIS	Knapsack	CC	TSP
State	`(i)`	`(i,j)`	`(i)`	`(id,remW)`	`(v)`	`(pos,mask)`
Space	$O(n)$	$O(n^2)$	$O(n)$	$O(nS)$	$O(V)$	$O(n2^n)$
Transition	subarray	submatrix	all $j < i$	take/ignore	all n coins	all n cities
Time	$O(1)$	$O(1)$	$O(n^2)$	$O(nS)$	$O(nV)$	$O(2^n n^2)$

Table 3.4: Summary of Classical DP Problems in this Section

Non-Classical Examples

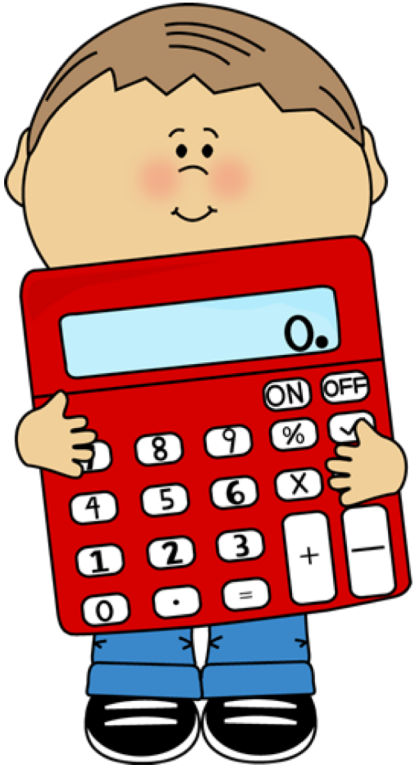
Background Photo:
Bridge

Non-Classical DP Problems

- not the pure form | variant of
  - 1D/2D Max Sum
  - LIS
  - 0-1 Knapsack/Subset Sum
  - Coin Change
  - TSP
- where the DP states and transitions can be “memorized”
- requires original formulation of DP states and transitions

State	distinct state
Space	number of distinct states
Transition	entails overlapping sub problems
Time	of one state
Overall Complexity	Space x Time
Answer	

Non-Classical Examples



1. UVa 10943 - How do you add?
2. UVa 10003 - Cutting Sticks

UVa 10943 - How do you add?

Larry is very bad at math — he usually uses a calculator, which worked well throughout college. Unforunately, he is now struck in a deserted island with his good buddy Ryan after a snowboarding accident.

They're now trying to spend some time figuring out some good problems, and Ryan will eat Larry if he cannot answer, so his fate is up to you!

It's a very simple problem — given a number N, how many ways can K numbers less than N add up to N?

For example, for $N = 20$ and $K = 2$, there are 21 ways:

0+20
1+19
2+18
3+17
4+16
5+15
...
18+2
19+1
20+0

Input

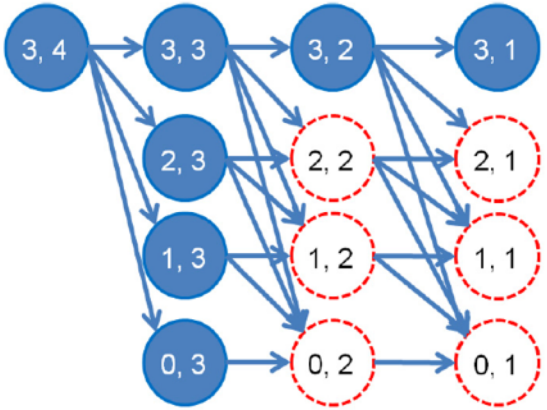
Each line will contain a pair of numbers N and K. N and K will both be an integer from 1 to 100, inclusive. The input will terminate on 2 0's.

Output

Since Larry is only interested in the last few digits of the answer, for each pair of numbers N and K, print a single number mod 1,000,000 on a single line.

UVa 10943 - How do you add?

- abridged problem description:
  - given int n
  - ? how many ways can k ints add up to n?
  - $0 \leq i_1 \ldots i_k \leq n$
  - constraints: $1 \leq n, k \leq 100$
- example:
  - $n = 20$ & $k = 2$ → 21 ways: 0+20, 1+19, 2+18, 3+17, . . . , 20+0.
- Math → Binomial coefficients
  - $ways = C_{k-1}^{n+k-1}$
- state params? (n, k)
- base case? $k = 1 \rightarrow ways = 1$
- general? $k > 1 \rightarrow$
  - $n = X + (n - X), X \in [0..n]$
  - $(n, k) \rightarrow (n - X, k - 1), X \in [0..n]$
- CS recurrence:
  1. $ways(n, 1) = 1$
  2. $ways(n, k) = \sum_{X=0}^{n} ways(n - X, k - 1)$
- overlapping sub-problems



UVa 10943 - How do you add?

Constraints	$1 \le n, k \le 100$
State	(n, k)
Space	$n \times k$
Transition	$(n - X, k - 1)$ $X \in [0..n]$
Time	$O(n)$
Overall Complexity	$O(n^2 k) \approx 1M$
Answer	$ways(n, k)$
Additional	result % 1M last 6 digits

- Top-Down

```
#include <cstdio>
#include <cstring>
using namespace std;
int N, K, memo[110][110];
int ways(int N, int K) {
  if (K == 1) return 1;
  int &ans=memo[N][K];
  if (ans!= -1) return ans;
  int total_ways = 0;
  for (int split = 0; split <= N; split++)
    total_ways = (total_ways
              + ways(N - split, K - 1)) % 1000000;
    ans = total_ways;
  return ans;
}
int main() {
  memset(memo, -1, sizeof memo);
  while (scanf("%d %d", &N, &K), (N || K))
    printf("%d\n", ways(N, K));
  return 0;
}
```


Dr Siba Haidar & Dr Antoun Yaacoub

UVa 10943 - How do you add?

- Bottom-Up

1	0	1	1	1	1	1	1	1	1	1
2	0	1	2	3	4	5	6	7	8	9
3	0	1	3	6	10	15	21	28	36	45
4	0	1	4	10	20	35	56	84	120	165
5	0	1	5	15	35	70	126	210	330	495
6	0	1	6	21	56	126	252	462	792	1287
7	0	1	7	28	84	210	462	924	1716	3003
8	0	1	8	36	120	330	792	1716	3432	6435
9	0	1	9	45	165	495	1287	3003	6435	12870
10	0	1	10	55	220	715	2002	5005	11440	24310

```
#include <cstdio>
#include <cstring>
using namespace std;
int main() {
  int i, j, split, dp[110][110], N, K;
  memset(dp, 0, sizeof dp);
  for (i = 0; i <= 100; i++)
    dp[i][1] = 1;
  // correct topological order
  for (j = 1; j < 100; j++)
    for (i = 0; i <= 100; i++)
      for (split = 0; split <= 100 - i; split++) {
        dp[i + split][j + 1] += dp[i][j];
        dp[i + split][j + 1] %= 1000000;
      }
  while (scanf("%d %d", &N, &K)&& (N || K))
    printf("%d\n", dp[N][K]);
  return 0;
}
```

- shared table → BU fill once ☺
- each column depends on the previous
  - proceed filling column then line

$$ways(n,k) = \sum_{X=0}^{n} ways(n-X, k-1)$$

Non-Classical Examples

1. UVa 10943 - How do you add?
2. UVa 10003 - Cutting Sticks

Dr Siba Haidar & Dr Antoun Yaacoub

UVa 10003 - Cutting Sticks

You have to cut a wood stick into pieces. The most affordable company, The Analog Cutting Machinery, Inc. (ACM), charges money according to the length of the stick being cut. Their procedure of work requires that they only make one cut at a time.

It is easy to notice that different selections in the order of cutting can led to different prices. For example, consider a stick of length 10 meters that has to be cut at 2, 4 and 7 meters from one end. There are several choices. One can be cutting first at 2, then at 4, then at 7. This leads to a price of $10 + 8 + 6 = 24$ because the first stick was of 10 meters, the resulting of 8 and the last one of 6. Another choice could be cutting at 4, then at 2, then at 7. This would lead to a price of $10 + 4 + 6 = 20$, which is a better price.

Your boss trusts your computer abilities to find out the minimum cost for cutting a given stick.

Input

The input will consist of several input cases. The first line of each test case will contain a positive number l that represents the length of the stick to be cut. You can assume $l < 1000$. The next line will contain the number n ($n < 50$) of cuts to be made.

The next line consists of n positive numbers c_i ($0 < c_i < l$) representing the places where the cuts have to be done, given in strictly increasing order.

An input case with $l = 0$ will represent the end of the input.

Output

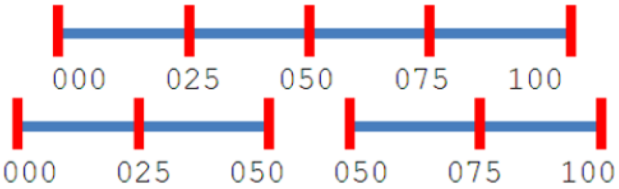
You have to print the cost of the optimal solution of the cutting problem, that is the minimum cost of cutting the given stick. Format the output as shown below.

Sample Input

```
100
```

UVa 10003 - Cutting Sticks

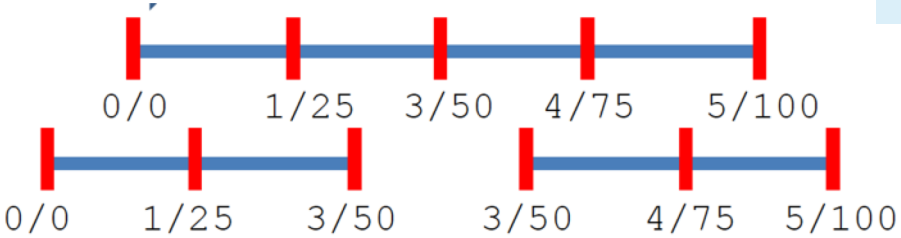
- abridged problem statement:
  - given a stick of length $1 \leq l \leq 1000$
  - and $1 \leq n \leq 50$ cuts to
  - $c_i \in [0..l], i = 1..n$
  - cost of cut = length of stick
  - ? find a cutting sequence to minimize overall cost
- example: $l = 100, n = 3, A = \{25,50,75\}$ (sorted)
- $optimal = 100$



- time complexity?
  - n choices for the cutting points
  - once we cut at a certain cutting point, we are left with n − 1 further choices of the second cutting point
  - repeats until we are left with zero cutting points
  - trying all possible cutting points this way leads to an O(n!)
  - →impossible for $1 \leq n \leq 50$
- overlapping sub-problems?
  - yes

UVa 10003 - Cutting Sticks

- state: index (l, r)
  - where $l, r \in [0..n+1]$ & $l < r$
  - Q: Why these two parameters?
- transition:
  - try all possible cutting points i between l and r
  - cut (l, r) into (l, i) and (i, r)
  - $+cost(A[r] - A[l])$



Constraint	$1 \le n \le 50$
State	(l, r)
Space	$O(n^2)$
Transition	$(A[r]-A[l])$ $+ min($ $cut(l,i)$ $+cut(i,r))$ $i \in [l..r]$
Time	$O(n)$
Overall Complexity	$O(n^3)$ feasible
Answer	$cut(0, n+1)$

UVa 10003 - Cutting Sticks

- Top-Down

```
int l, n, A[55], memo[55][55];
int cut(int left, int right) {
  if (left + 1 == right)       return 0;
    int &ans = memo[left][right];
  if (ans != -1) return ans;
  ans = 2000000000;
  for (int i = left + 1; i < right; i++)
    ans = min( ans, cut(left, i)
                  + cut(i, right)
                  + (A[right]-A[left]));
  return ans;
}
int main() {
  while (scanf("%d", &l)&& l) {
    A[0] = 0;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
      scanf("%d", &A[i]);
    A[n + 1] = l;
    memset(memo, -1, sizeof memo);
    printf("The minimum cutting is %d.\n",
            cut(0, n + 1));
  }
  return 0;
}
```

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Table 1.4: Rule of thumb time complexities for the 'Worst AC Algorithm' for various single test-case input sizes n, assuming that your CPU can compute 100M items in 3s.