GB21802 - Programming Challenges

Week 3 - Dynamic Programming

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2018-05-18,21

Last updated May 16, 2018

Results for the Previous Week

Non-grading Problems

Deadline: 2018/7/29 23:59:59 (74 days, 00:13 hours from now)
Problems Solved -- 0P:37, 1P:2, 3P:6,

click to show/hide

Week 0: Introduction and Problem Solving

Deadline: 2018/4/26 23:59:59 (expired)
Problems Solved -- 0P:3, 1P:1, 2P:3, 3P:14, 4P:5, 6P:2, 7P:1, 8P:16,

click to show/hide

Week 1: Data Structures

Deadline: 2018/5/10 23:59:59 (expired)
Problems Solved -- 0P:5, 1P:6, 2P:5, 3P:10, 4P:2, 5P:1, 6P:2, 7P:2, 8P:12,

click to show/hide

Week 2: Search Problems

Deadline: 2018/5/17 23:59:59 (1 day, 00:13 hours from now)
Problems Solved -- 0P:14, 1P:4, 2P:2, 3P:9, 4P:2, 5P:1, 6P:2, 7P:1, 8P:10,

click to show/hide

Quick Review of Last week

- Definition of a Search Problem
- Three types of algorithms for search problems:
  - Complete Search
  - Divide and Conquer
  - Greedy Search

This week we introduce the last, and arguably most important algorithm for search problems: Dynamic Programming

What is Dynamic Programming (DP)?

DP is technique for Search Problems which is based on the idea of building partial solutions.

Basic Idea of DP

Create a 2D table where:

- The rows are possible choices for the problem;
- The columns are sub-solutions for the problem;

First, fill the first column with each possible choice.

Second, fill column n+1 with the combination of best solution for n and best choice for n+1|n.

Characteristics of DP:

Used for *optimization* or *counting* problems

- “Count the number of solutions...”
- “Find the minimum cost...”
- “Find the maximum length...”

DP depends on *number of states* and *induction*

- The complexity of DP is based on the size of the table;
- The correctness of DP is based on the idea of induction;

Problem Example: Wedding Shopping – UVA 11450

Best way to understand DP is to do a lot of examples!

Buying problem:

- Buy one of each C classes of items.
- Maximize the cost, but do not exceed the budget M

- $M \leq 200$, $C \leq 20$;
- Each class has $0 < K_i \leq 20$ options;
- Total possible choices: 20^{20}!

Problem Example: Wedding Shopping – UVA 11450

Sample case 1: $C = 3$

Class	1	2	3	4
0	6	4	8	
1	5	10		
2	1	5	3	5

For budget $M = 20$, the answer is 19, which can be reached by buying items $(8 + 10 + 1)$, $(6 + 10 + 3)$ or $(4 + 10 + 5)$.

On the other hand, if the budget $M = 9$, the answer is "no solution", because the minimal possible budget is 10, reached by buying items $(4 + 5 + 1)$.

Problem Example: Wedding Shopping – UVA 11450

Sample case 1: $C = 3$

Class	1	2	3	4
0	6	4	8	
1	5	10		
2	1	5	3	5

This looks like a search problem! Can we do a greedy search?

One approach: For each class, choose most expensive item:

- $(8 + 10 + 1)$ – It works!
- But if $M = 12$... It does not work anymore.

Also no idea for divide and conquer?

Wedding Shopping (11450) – Complete search

Let's image a solution using Complete Search:

Recursive Approach

- Function shop(m,g) discovers the best item k in class C_g that can be bought with total money m
- For each k in C_g, the value for that choice is $V_k = \text{cost}_k + \text{shop}(m - \text{cost}_k, g + 1)$
- shop(m,g) returns the maximum $V_k =< m$;
- shop(m,$|C|$), when we pass the last item, returns 0;

This works! ... but Time Limit Exceeded.

Wedding Shopping (11450) - Complete search

Time Limit Exceeded:

20 categories of items, with 20 items each, a complete search will take: 20^{20} operations.

Problem: Too many overlapping subproblems

Sample case 2: $C = 4$

Class	1	2	3	4
0	6	4	8	12
1	4	6	6	2
2	1	5	1	5
3	2	4	6	2

How many times *shop(15,3)* is called?

Every time we call *shop(15,3)*, the solution is the same.

Wedding Shopping – the DP approach

Since the problem has an overlapping subproblem structure, we can think about using a DP approach.

The first step of using DP is constructing the state table.

How many states do we need?

Since we know the unique states are *(Money,Class)*, we need to make a table of *MxC*.

Since $0 \leq M \leq 200$ and $0 < C \leq 20$, our table will have $201 * 20 = 4020$ states.

Only 4020 states! This looks promising!

Wedding Shopping – the DP approach

Now that we have our state table, there are two approaches for building a DP solution:

- The top-down approach:
  Use the state table as a look-up table. Save the result of visited states, and do not re-calculate those.

- The bottom-up approach:
  Fill the base-states of the table, and iteratively fill the other states transitioning from the base.

Wedding Shopping – top-down DP

```
memset(table,-1,sizeof(table))   ** DP <3 memset **

shop(m,g):
   if (m < 0) return -INF
   if (g == C) return M - money
   if (table[m][g]) != -1 return table[m][g]   **NEW**
   return table[m][g] = max(shop(m-price[g][k],g+1)
                            for every k)
```

To implement top-down DP, simply add a table check to a complete recursive search.

Make sure that your states are independent from the parent. If they are not, you need to rethink your state table.

Wedding Shopping – bottom-up DP

Algorithm:

- Prepare a table with the problem states (same as top-down);
- Fill the table with base case values;
- Find the topological order in which the table is filled;
- Fill the non-basic cases;

The main problem for bottom-up DP is finding the base cases and the ordering between the cases. In some good cases, the ordering is just a list of nested loops!

Wedding Shopping – bottom-up DP

Example: M=10, G1=(2,4), G2=(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
G0											1
G1											
G2											
G3											

- Initial state: With 0 items, we can reach money 10;
- For each reachable money state in G_i, we can reach money state in G_{i+1} according to the costs of the items.
- If any money state in G_C is reachable, the problem is solvable.
- Solution is M - minimal reachable state;

Wedding Shopping – bottom-up DP

Example: M=10, G1=(2,4), G2=(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
G0											1
G1							1		1		
G2											
G3											

- Initial state: With 0 items, we can reach money 10;
- For each reachable money state in G_i, we can reach money state in G_{i+1} according to the costs of the items.
- If any money state in G_C is reachable, the problem is solvable.
- Solution is M - minimal reachable state;

Wedding Shopping – bottom-up DP

Example: M=10, G1=(2,4), G2=(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
G0											1
G1							1		1		
G2	1		1		1						
G3											

- Initial state: With 0 items, we can reach money 10;
- For each reachable money state in G_i, we can reach money state in G_{i+1} according to the costs of the items.
- If any money state in G_C is reachable, the problem is solvable.
- Solution is M - minimal reachable state;

Wedding Shopping – bottom-up DP

Example: M=10, G1=(2,4), G2=(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
G0											1
G1							1		1		
G2	1		1		1						
G3	1	1	1	1							

- Initial state: With 0 items, we can reach money 10;
- For each reachable money state in G_i, we can reach money state in G_{i+1} according to the costs of the items.
- If any money state in G_C is reachable, the problem is solvable.
- Solution is M - minimal reachable state;

Wedding Shopping – bottom-up DP

M=10, cost[1] = [2,4], cost[2] =(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
g = 0											1
g = 1							1		1		
g = 2	1		1		1						
g = 3	1	1	1	1							

```
memset(table,0,sizeof(table))

table[0][10] = 1

for g in (0:G-1)
  for i in (0:M):
    if table[g][i] == 1:
       for k in C[g+1]:
          table[g+1][i-cost[k]] = 1
```

DP: Top-down or Bottom-up?

Top-Down

Pros:

Easy to implement starting from a recursive search. Only compute states if necessary.

Cons:

Recursive calls are slower if there are many levels, state table memory usually can't be reduced.

Bottom-Up

Pros:

Faser if many sub-problems are visited. Can sometimes save memory space by keeping only s and s+1 in memory.

Cons:

Not very intuitive. If there are X states, all states values will have to be checked.

DP: What about the decision set?

In the previous example, we only cared about the final value of the solution. What if we want to know the exact items in an optimal solution?

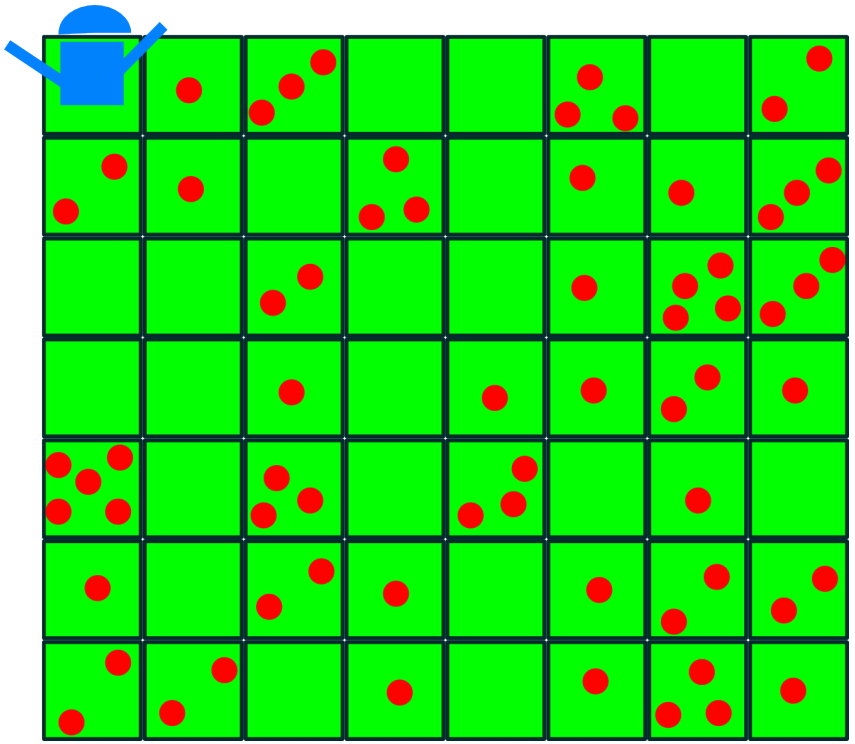
Together with the *state* table, keep a second *parent* table. Every time you update a value on the state table, write in the parent table which cell(s) was used to update that value.

Pay attention to whether the problem requires you to print the first solution, or the last, or a solution with some particular properties.

Let's see some code in the next DP example.

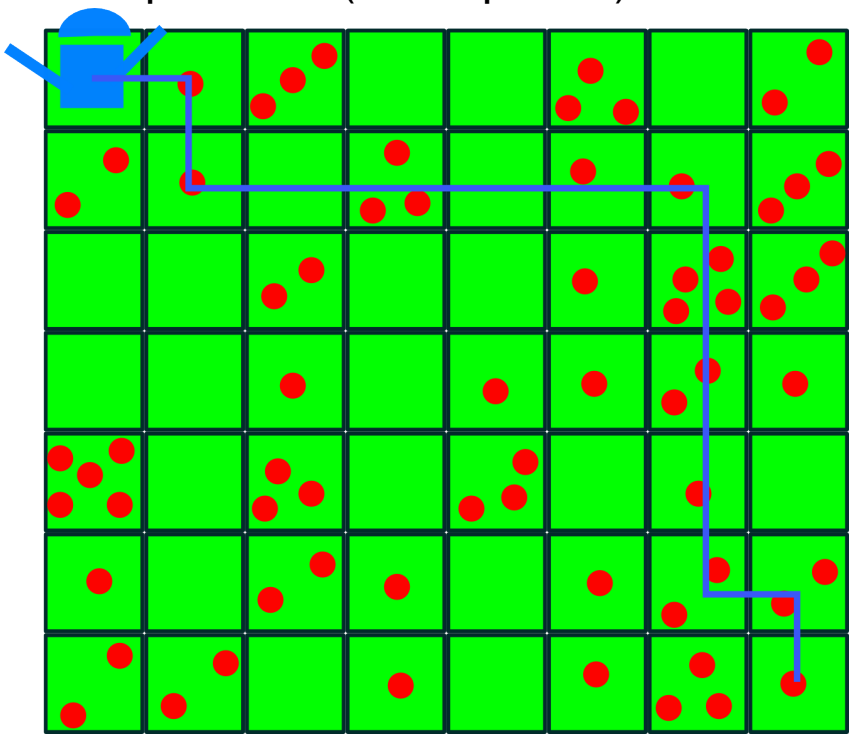
Example 2: Apple Field – not on UVA

Farmer Tanaka is growing apples in his field. He has a robot to collect the apples, but the robot can only walk east and south. If you know how many apples there are in each tree of his $m * n$ field, can you calculate the path that the robot should make to collect most apples?



Example 2: Apple Field – complete search

One possible (non-optimal) solution

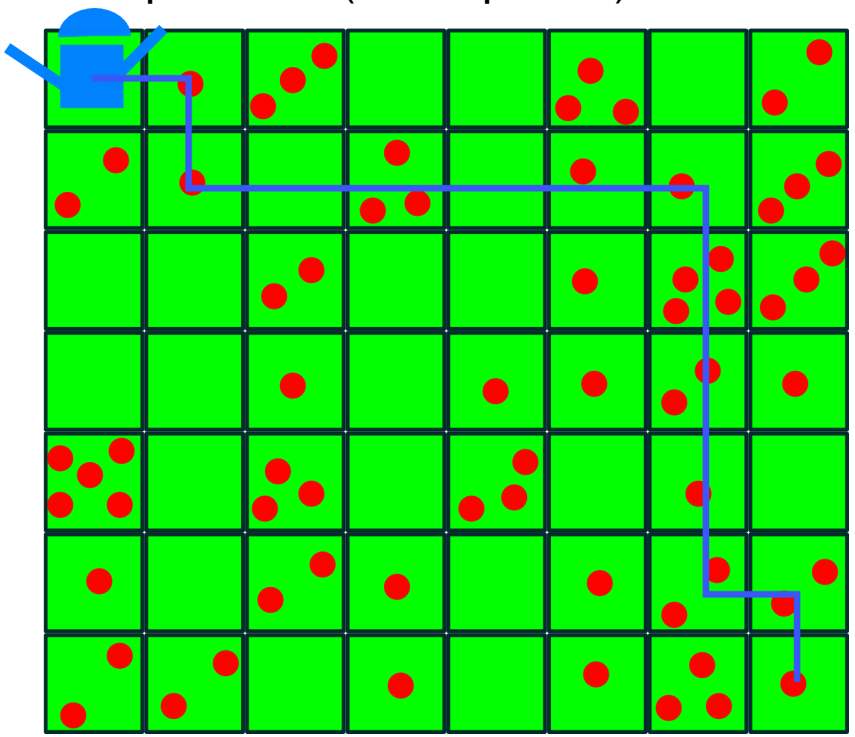


For every square, the robot can choose to go right (east) or down (south). How many possible paths exist?

And how many overlapping solutions are there?

Example 2: Apple Field – overlapping solutions

One possible (non-optimal) solution



If the robot reached position (x,y) with k apples, it does not matter how it did it. So the path from (x,y) to the goal is independent from the parent path.
This smells like DP! Let's solve it with a bottom up approach.

Example 2: Apple Field – Bottom up DP

- tables: We want to maximize the number of apples at the goal. Let's try a m*n position table. At each cell, we want to store the maximum number of apples achievable at that cell.

  We also want a second table, *path*, which will store the path used.

- initial condition: We can fill the top row (x,1) of the table with the sum of apples from 1 to x. And we can fill the Left column with the sum of apples from 1 to y.

  Alternatively, we can add an additional dummy "-1" column/line and fill it with zeros.

- transition: For each cell x,y, max(x,y) is either max(x-1,y) + apple(x,y) or max(x,y-1) + apple(x,y)

Example 2: Apple Field – Let's see some code

```
// Assume apple[0,] and apple[,0] are all zeroes.
int apple[m][n]// Input.
int sum[m][n]// DP table, set to 0 using memset
int parent[m][n][2]// Path table, set to 0 using memset

for i in (1:m):
  for j in (1:n):
    sum[m][n] = apple[m][n] + max(sum[m][n-1],sum[m-1][n])
    if (sum[m][n-1] > sum[m-1][n]):
       parent[m][n][0] = m, parent[m][n][1] = n-1
    else:
       parent[m][n][0] = m-1, parent[m][n][1] = n
```

Classical DPs: 1D Range Sum

Given an array A containing $< 20K$ non-zero integers, determine the maximum range sum of A.

A = 1, -3, 20, -2, -5, 10, 5, -4, 6, 47, -30, -3
S = -, –, 20, -2, -5, 10, 5, -4, 6, 47, —, –

What is a range sum?

A maximal range sum is two indexes $i, j, i < j$ in A that maximizes the sum $a_i + a_{i+1}, a_{i+2}, \ldots, a_j$.

Search Approaches for 1D Range Sum

Naive Complete Search O(n^3)

Naive approach: For every possible pair i,j, add the values between i and j and save the result.

```
maxindex = []
maxsum = 0
for i in (0:n):
   for j in (i:n):
      sum = 0
      for k in (i:j):
         sum += k
      if sum > maxsum:
         maxsum = sum
         maxindex = [i,j]
```

Search Approaches for 1D Range Sum

Better Complete Search – $O(n^2)$

If we initialize A with the right values, we can skip the sum in the loop of the previous slide.

```
A = [], maxsum = 0
for (i in 0:n):
   A[i] = A[i-1] + input
   // A[i] now has the sum 0..i

for (i in 0:n):
   for (j in i:n):
      sum = A[j] - A[i-1]
      if sum > maxsum:
         maxsum = sum
         maxindex = [i,j]
```

Search Approaches for 1D Range Sum

Using the $O(n^2)$ pre-computation:

Input = 1, -3, 20, -2, -5, 10, 5, -4, 6, 47, -30, -3
A = 1, -2, 18, 16, 11, 21, 26, 22, 28, 75, 45, 42

- Subset: 2 to 9 = A[9] - A[2-1] = 75 - (-2) = 77;
- Subset: 0 to 9 = A[9] - 0 = 75;
- Subset: 5 to 6 = A[6] - A[5-1] = 26 - 11 = 15;

Can we do better?

1D Sum – Kadane's Greedy Algorithm

Greedly Max Sum O(n)

```
A[] = { 4, -5, 4, -3, 4, 4, -4, 4, -5};
int sum = 0, ans = 0;
for (i in 0:n):
   sum += A[i], ans = max(ans, sum)
   if (sum < 0) sum = 0; // reset greedy
```

- Sum is the answer if we sum all the numbers
- Sum is reset to zero if we ever go negative (better to start again)
- At each step, we have two choices: add the sum from the previous step, or start from zero.

Classic DP – Maximum 2D sum

Problem Summary

Given an array of positive and negative numbrers, find the subarray with maximum sum.

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

How big would be a complete search for this problem?

Maximum 2D Sum – Complete Search

A complete search approach needs 6 loops (2 for each coordinate, 2 for the sum), for a total complexity of $O(n^6)$.

```
minvalue = -MIN_INT
for i in (0:n):
   for j in (0:n):
      for k in (i:n):
         for l in (j:n):
         sum = 0
         for a in (i:k):
            for b in (j:l):
               sum += A[a,b]
         if sum > minvalue:
            minvalue = sum
```

Max 2D sum using partial sums

The "partial sum" algorithm used for 1D arrays can also be adapted to 2D Matrices, using the "inclusion/exclusion" principle.

```
for i in (0:n):
   for j in (0:n):
       A[i][j] = input
       if (i > 0) A[i][j] += A[i-1][j]
       if (j > 0) A[i][j] += A[i][j-1]
       // Avoid double count
       if (i > 0 && j > 0) A[i][j] -= A[i-1][j-1]

for i,j in (0:n)(0:n):
   for k,l in (i:n)(j:n):
      sum = A[k][l]
      if (i > 0) sum -= A[i - 1][j];
      if (j > 0) sum -= A[i][j-1];
      if (i > 0 && j > 0) sum += A[i-1][j-1]
      maxsum = max(sum,maxsum)
```

Classic DP: Longest Increasing Subsequence

Problem Definition

Given a sequence $A = \{A[0], A[1], \ldots, A[n]\}$, determine the longest subsequence of increasing numbers.

Note that the numbers do not need to be contiguous.

Example:

A = { -7, 10, 9, 2, 3, 8, 8, 1}

Longest Increasing Subsequence: Complete Search

The complete search for the LIS problem requires you to test all possible subsets. This takes $O(2^n)$.

Very slow for any n big enough :-(

Remember that you can easily list all subsets using a loop on a bitmask:

```
for (i in 0:1<<n):
   subset = []
   ind = 0
   while (1 << ind) < i:
      if (1 << ind) & i:
         subset.append(set[ind])
      ind++

   if len(subset) > maxlen:
      maxlen = len;
```

Longest Increasing Subsequence – DP approach

1 – What is the state space?

For each element $A[i]$, we just need to know what is the size of the maximum subsequence up to that element.

The element at the end of that sequence is either $A[i]$, or an element before $A[i]$ with the same max sequence size.

A[i]	-7	10	9	2	3	8	8	1
LIS[i]	1	2	2	2	3	4	4	2

2 – What is the state transition?

For every $A[i+1]$ we find k in $0 : i$ that has the biggest LIS$[k]$ where $A[k] < A[i+1]$. LIS$[i+1] =$ LIS$[k] + 1$.

The complexity of this approach is $O(n^2)$

Longest Increasing Subsequence – DP approach

```
LIS[0:n] = 1
parent[0:n] = -1
for i in (1:n):
   for j in (0:i):
      if (LIS[j] >= LIS[i]) && (A[j] < A[i]):
         LIS[i] = LIS[j] + 1
         parent[i] = j
```

There is also a $O(n \log k)$ approach that uses greedy algorithm and binary search. I'll leave that one for you to study at home!

Classic DP: 0-1 Knapsack Problem

Problem Definition – also known as *subset sum*

Given a set A with n elements. Each element has a value V_i and size S_i. Calculate the subset with maximum sum V_{subset} and $S_{subset} \leq S$.

V_i	100	70	50	10
S_i	10	4	6	12

$S = 12$

Complete Search Recursive Solution – Similar to Wedding Problem

Consider the recursive function *value(id,size)*, where *id* is the item we want to test, and *size* is the size remaining in the backpack.

```
value(id,size):
   if size == 0, return 0 // bag is full
   if id == n, return 0 // checked all items
   if S[id] > size, return value(id+1,size) // too big
   return max(value(id+1,size),
              V[id] + value(id+1, size - S[id]))
```

0-1 Knapsack Problem: DP Approaches

We can easily modify the algorithm in the last slide to a Top-down DP.

Just be careful: the size of the state table is: nS.
If S is too large ($>> 1M$), this approach might be unfeasible.

Classical DP – The Coin Change Problem (CC)

Problem Summary

Given a target value V, and a list A of n coin sizes, what is the minimum number of coins necessary to represent V? Assume unlimited supply of coins.

Example: $V = 7, A = \{1, 3, 4, 5\}$

One answer is $5 + 1 + 1$. The optimal answer is $4 + 3$

Recurrence

The number of coins for value V is 1 + the number of coins for value $V - \textit{size of coin}$.

Coin Change – Complete Search Algorithm

Recursive Complete Search Again!

```
change(value):
   if value == 0: return 0 // 0 coins for 0
   if value < 0: return INF // impossible result
   min = INF
   for i in (0:n):
      t = 1 + change(value - A[i])
      if (t < min): min = t
   return t
```

Can you see the many overlapping subproblems?

- By now you should know how to do the top-down DP from here;
- Can you modify the recurrence to show how many different ways you can complete a value?
  (hint: complexity does not change);
- If you have many queries V, but with the same set of coins A, you don't need to clear the table.

Summary

Dynamic Programming is a smart way to do complete search, when a large part of the search space is overlapping.

- Top-down DP: recursive complete search with memoization;
- Bottom-up DP: state table with iterative completion recurrency;

Read more about DP

- http://people.csail.mit.edu/bdean/6.046/dp/
- http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg

Problem Discussion – At a Glance

- Wedding Shopping – Example Problem
- Jill Rides Again – Subset Sum (1D)
- Largest Submatrix – Subset Sum (2D)
- Is Bigger Smarter? – Longest Increasing Subsequence
- Murcia's Skyline – Longest Increasing Subsequence
- Trouble of 13 Dots – 0-1 Knapsack
- Exact Change – Coin Change
- Unidirectional TSP – Pathfinding

Problem Hints

- Wedding Shopping
- Jill Rides Again

Discussed during class – just apply the algorithm!

Problem Hints 2

Largest Submatrix

Find the largest rectangles of “1” inside a matrix of 1s and 0s.

Hints:

- You want to do a subset-sum to find the rectangle with biggest sum (biggest number of 1).
- But how do you avoid adding zeroes?

This algorithm is often used as a part of other problems, to calculate valid territory.

Problem Hints 3

Is bigger Smarter?

Out of a set of elephants, you have to find the largest subset that:

- A - Intelligence is decreasing
- B - Weight is increasing

At the same time.

Hints:

- You don't need to calculate both intelligence and weight at the same time.
- Organize your data well, and you only need to worry about one.

Problem Hints 4

Murcia Skyline

Find the longest increasing skyline and the longest decreasing skyline.

Hints

- Standard “Longest Increasing Subsequence”
- But the “longest” here is modified by the *building width*

Problem Hints 4

Trouble of 13 dots

Find the subset of items that:

- Mazimize flavor;
- Is inside the price budget

Hints

- This is a 1-0 knapsack problem:
- Find the set of items that fills in the knapsack maximizing flavor
- Beware that the knapsack change size if price > 2000!

Problem Hints 5

Exact Change

Find the smallest amount of overpay that you can do, with the smallest number of coins.

Hints

- Variation of the Coin Change problem discussed in Class;
- Calculate all possible changes above the desired value, and find the smallest;
- Order by smallest number of coins necessary;
- Bottom Up algorithm is probably best;

Problem Hints 6

Unidirectional TSP

Find the minimal path from left to right. Up and down are connected!

- Very similar to the “apple robot” problem;
- Note that two paths with the same weight, the smaller index is best!