Introduction to Competitive Programming

Instructor: William W.Y. Hsu

CONTENTS

- Dynamic programming
- Non classical DP problems

Dynamic Programming

The most challenging problem-solving technique…

A tastes of DP

- **UVa 11450 – Wedding shopping**
  - Given $1 \leq \boldsymbol{C} \leq 20$ classes of garments
    - e.g. shirt, belt, shoe
  - Given $1 \leq \boldsymbol{K} \leq 20$ different models for each class of garment
    - three shirts, two belts, four shoes, …, each with its own price
  - Task: Buy **just one** model of **each class** of garment
  - Our budget 1 ≤ **M** ≤ 200 is limited
    - We cannot spend more money than it.
    - But we want to spend the maximum possible.
  - What is our maximum possible spending?
  - Output "no solution" if this is impossible.

Demonstration

- Budget M=100
- C=3, K=3
  - Best solution is 75.

Model Garment	0	1	2	3
0	8	6	4	
1	5	10		
2	1	3	3	7
C=3	50	14	23	8

Demonstration

- Budget M=20
- C=2, K=3
  - Best solution is 19.
    - Alternative answers are possible!

Model Garment	0	1	2	3
0	4	6	8	
1	5	10		
2	1	3	5	5

- Budget M=5
  - No Solution!

Model Garment	0	1	2	3
0	6	4	8	
1	10	6		
2	7	3	1	7

Greedy solution?

- What if we buy the most expensive model for each garment which still fits our budget?
- Counter example:
  - M=12
  - Greedy algorithm produces: No solution! (uh oh~)
  - The correct answer is 12!
    - More than one answer is present.

Model Garment	0	1	2	3
0	6	4	8	
1 ?	5	10		
2	1	5	3	5

Complete search

- What is the potential **state** of the problem?
  - g (which garment?)
  - id (which model?)
  - $money$ (money left?)
- Answer:
  - $(money, g)$ or $(g, money)$
- Recurrence (recursive backtracking function):

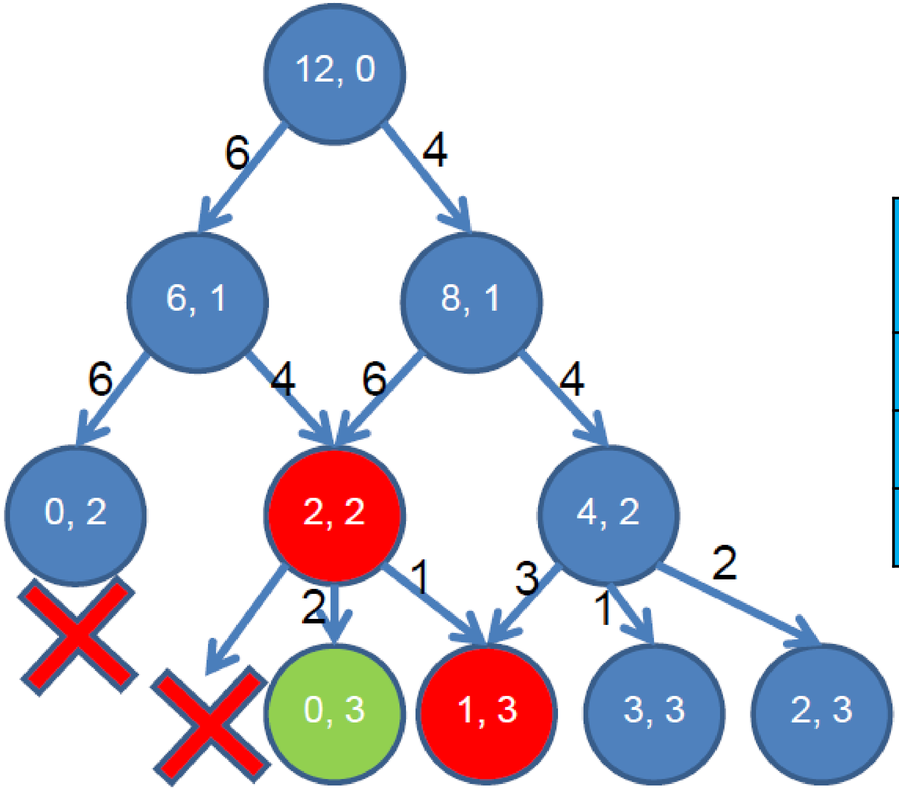
```
shop(money, g)
  if (money < 0) return -INF
  if (g == C) return M – money
  return max(shop(money – price[g][model], g + 1),
    ∀model∈[1..K]
```

Complete search

- Extreme analysis when $M = 200$, 20 models, and 20 garments.
  - Time complexity is 20^{20} (3 second TLE!)

Overlapping sub problems

› Observe the 20^{20} search space, you can find **many** **overlapping sub problems**.
  – Many routes lead to the same state.



Model Garment	0	1	2
0	6	4	
1	4	6	
2	1	2	3

Dynamic programming

- DP = Dynamic Programming
  - Programming here is not writing computer code, but a **“tabular method”**!
    - **Table** method
- **A programming paradigm that you must know!**
  - and hopefully, master…

Dynamic programming

- Use DP when the problem exhibits:
  - Optimal sub structure.
    - Optimal solution to the original problem contains optimal solution to sub problems.
  - This is **similar** as the requirement of **Greedy algorithm**
  - If you can formulate complete search recurrences, you can solve it.
  - Overlapping sub problems:
    - Number of **distinct sub problems** are actually “small”.
    - But they are **repeatedly computed.**
    - This is **different** from **Divide and Conquer.**

Dynamic programming

- There are two ways to implement DP:
  - Top-Down.
  - Bottom-Up.
- Top-Down (Demo):
  - Recursion as per normal + **memoization table**
    - It is just a simple change from backtracking (complete search) solution!

Turn recursion into memoization

Initialize memo table in main function (use ‘`memset`’)

```
return_value recursion(params/state) {
  if this state is already calculated,
    simply return the result from the memo table
  calculate the result using recursion(other_params/states)
  save the result of this state in the memo table
  return the result
}
```

Dynamic programming (Top-Down)

- For our example:

```
shop(money, g)
  if (money < 0) return -INF
  if (g == C) return M – money
  if (memo[money][g] != -1) return memo[money][g];
  return memo[money][g] = max(shop(money –
    price[g][model], g + 1),∀ model ∈ [1..K]
```

What if optimal solution(s) are needed

```
print_shop(money, g)
  if (money < 0 || g == C) return
  for each model ∈ [1..K]
    if shop(money – price[g][model], g + 1) ==
      memo[money][g]
    print "take model = " + model +
      " for garment g = " + g
    print_shop(money – price[g][model], g + 1)
    break
```

Dynamic programming (Bottom-Up)

- Another way: Bottom-Up:
  - Prepare a table that has size equals to the number of distinct states of the problem.
  - Start to fill in the table with base case values.
  - Get **the topological order** in which the table is filled
    - Some topological orders are natural and can be written with just (nested) loops!
  - Different way of thinking compared to Top-Down DP
- Notice that both DP variants use “table”!

Dynamic programming (Bottom-Up)

- Start with with table **can_reach** of size 20 (g) $*$ 201 (money)
  - The state $(money, g)$ is reversed to $(g, money)$ so that we can process bottom-up DP loops in row major fashion.
  - Initialize all entries to 0 (false).
  - Fill in the first row with money left (column) reachable after buying models from the first garment ($g = 0$)
  - Use the information of current row g to update the values at the next row $g + 1$.

Dynamic programming (Bottom-Up)

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	0	0	0	0	0	0	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

Model Garment	0	1	2	3
0	4	6	8	
1	5	10		
2	1	3	5	5

Dynamic programming (Bottom-Up)

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

Model Garment	0	1	2	3
0	4	6	8	
1	5	10		
2	1	3	5	5

Dynamic programming (Bottom-Up)

Money

		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
g	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

Model Garment	0	1	2	3
0	4	6	8	
1	5	10		
2	1	3	5	5

Top-down or Bottom-up?

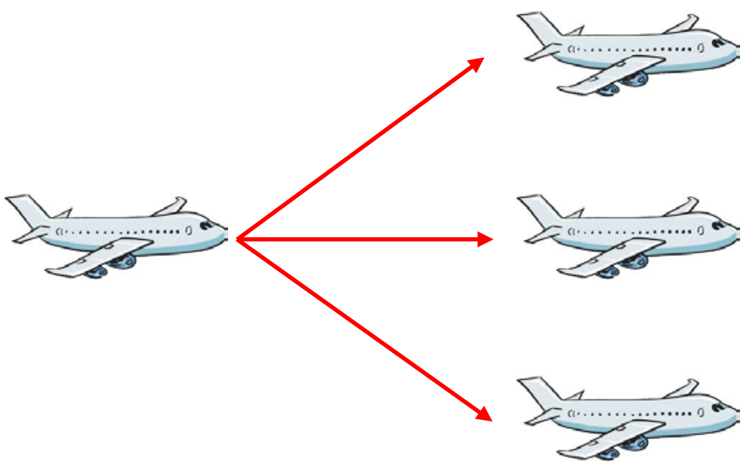
- Top-down
  - Pro:
    - Natural transformation from normal recursion.
    - Only compute sub problems when necessary.
  - Cons:
    - Slower if there are many sub problems due to recursive call overhead.
    - Use exactly $O(\text{states})$ tables size. (Could cause MLE)
- Bottom-up
  - Pro:
    - Faster if many sub problems are visited: no recursive calls!
    - Can save memory space (?)
  - Cons:
    - Maybe not intuitive for those inclined to recursion.
    - If there are X states, bottom up visits/fills the value of all these X states.

Solving UVa 10337 – Flight planner

- UVa 10337 – Flight planner
  - Unit: 1 mile altitude and 1 (x100) miles distance.
  - Given wind speed map.
  - Fuel cost: {Climb: +60, Hold: +30, Sink: +20} – wind speed `wsp[alt][dis]`.
  - Compute min fuel cost from (0,0) to (0,$x = 4$)!.

Solving UVa 10337 – Flight planner

› Wind table:
  - Tailwind: -1 unit of fuel.
  - Headwind: +1 unit of fuel.



1	1	1	1	1	9
1	1	1	1	1	8
1	1	1	1	1	7
1	1	1	1	1	6
1	1	1	1	1	5
1	1	1	1	1	4
1	1	1	1	1	3
1	1	1	1	1	2
1	9	9	1	1	1
1	-9	-9	1	1	0
0	1	2	3	4	(x100)

Of course, level 1

- First guess:
  - Do complete search/brute force/**backtracking**
  - Find *all possible* flight paths and pick the one that yield the minimum fuel cost!

Complete search

- Recurrence of the Complete Search
  - **fuel**(alt, dis) =
    ```
    min3(60 - wsp[alt][dis] + fuel(alt + 1, dis + 1),
    30 - wsp[alt][dis] + fuel(alt , dis + 1),
    20 - wsp[alt][dis] + fuel(alt - 1, dis + 1))
    ```
  - Stop when we reach final state (base case):
    - `alt=0` and `dis=X`, i.e. `fuel(0,X)=0`
  - Prune infeasible states (also base cases):
    - `alt<0` or `alt>9` or `dis>X`!, i.e. return `INF*`
- Answer of the problem is **`fuel(0,0).`**

Complete Search Solutions

SOLUTION 1

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
0	1	2	3	4	(x100)

29+39+39+29=**136**

SOLUTION 2

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
0	1	2	3	4	(x100)

29+39+69+19=**156**

Complete Search Solutions

SOLUTION 3

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
0	1	2	3	4	(x100)

29+69+11+29=**138**

SOLUTION 4

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
0	1	2	3	4	(x100)

59+11+39+29=**138**

Complete Search Solutions

SOLUTION 5

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
0	1	2	3	4	(x100)

29+69+21+19=**138**

SOLUTION 6

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
0	1	2	3	4	(x100)

59+21+11+29=**120(OPT)**

Complete Search Solutions

SOLUTION 7

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
0	1	2	3	4	(x100)

59+21+21+19=**120(OPT)**

SOLUTION 8

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
0	1	2	3	4	(x100)

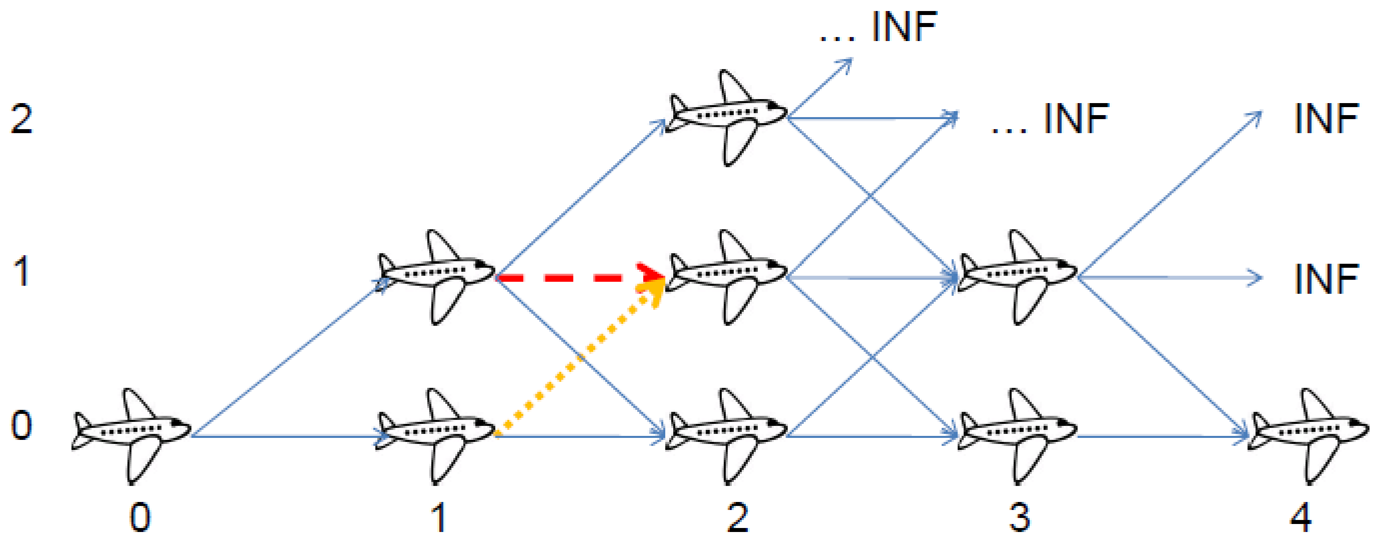
59+51+19+19=**148**

Complete search

- How large is the search space?
  - Max distance is 100,000 miles.
  - Each distance step is 100 miles.
  - That means we have **1,000** distance columns!
    - Note: this is an example of "coordinate compression"
  - Branching factor per step is 3… (climb, hold, sink).
  - That means complete search can end up performing 3^{1000} operation.

Overlapping sub problem issue

- In simple 3^{1000} Complete Search solution, you should have observe **many overlapping sub problems**!
  - Many ways to reach coordinate (alt, dis).



DP solution

- Recurrence of the Complete Search
  - **fuel**(alt, dis) =

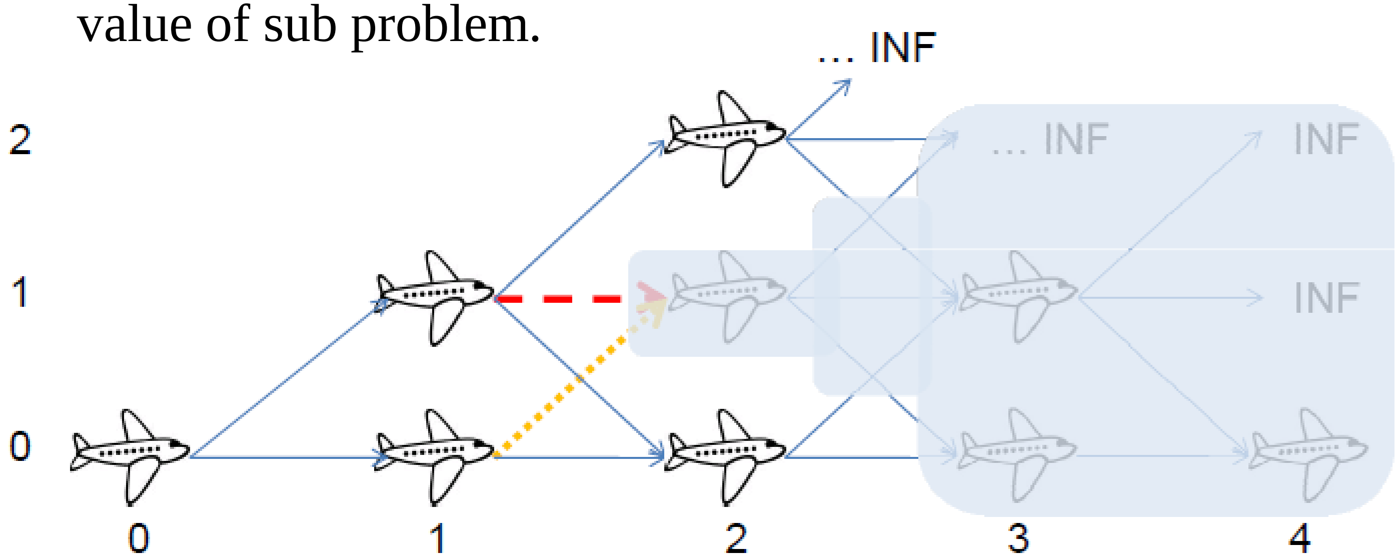
```
min3(60 - wsp[alt][dis] + fuel(alt + 1, dis + 1),
30 - wsp[alt][dis] + fuel(alt , dis + 1),
20 - wsp[alt][dis] + fuel(alt - 1, dis + 1))
```

- Sub-problem fuel(alt, dis) can be **overlapping**!
  - There are only 10 alt and 1,000 dis = **10,000** states
  - A lot of time saved if these are not re-computed!
  - Exponential 3^{1000} to polynomial 10×1000!

DP solution (Top-down)

- Create a 2-D table of size $10 \times (\frac{X}{100})$.
  - Save spaces.
  - Set "-1" for unexplored sub problems (`memset`)
  - Store the computation value of sub problem.

2	-1	-1	-1	∞	∞
1	-1	-1	40	19	∞
0	-1	-1	-1	29	0
	0	1	2	3	4



DP solution (Bottom-up)

- Memory saving tip:
  - Reduce the 2 dimensional array by keeping 2 recent columns.
  - Time complexity unchanged!

2	∞	∞			
1	∞	59			
0	0	29			
	0	1	2	3	4

2	∞	∞	110		
1	∞	59	80		
0	0	29	68		
	0	1	2	3	4

2	∞	∞	110	131	
1	∞	59	80	101	
0	0	29	68	91	
	0	1	2	3	4

2	∞	∞	110	131	-
1	∞	59	80	101	-
0	0	29	68	91	120
	0	1	2	3	4

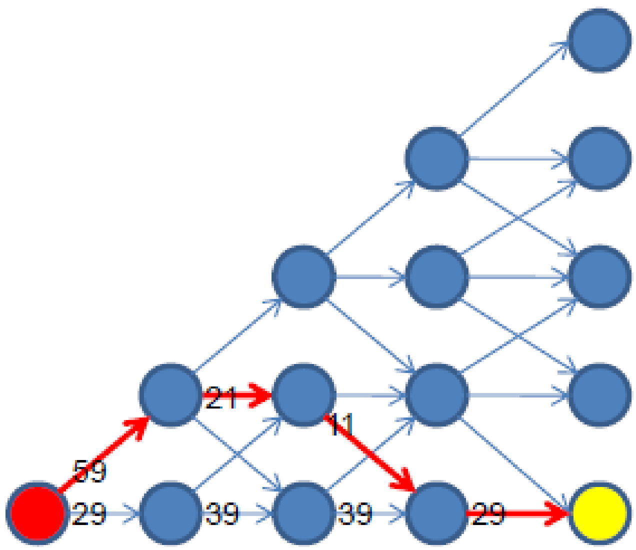
If optimal solution(s) are needed

- Although not often, sometimes this is asked!
- As we build the DP table, record which option is taken in each cell!
  - Usually, this information is stored in different table.
  - Do recursive scan(s) to output solution.
    - Sometimes, there are more than one solutions!

Remodel the problem

- Shortest path problem!
  - Model the problem as a **DAG.**
  - Vertex is each position in the unit map.
  - Edges connect vertices reachable from vertex `(alt,dis)`, i.e. `(alt+1,dis+1)`, `(alt,dis+1)`, `(alt-1,dis)`
    - Weighted according to flight action and wind speed!
    - Do not connect infeasible vertices.
      - `alt<0` or `alt>9` or `dis>X`

Visualization of the DAG



Source

What is the **shortest path** from source to destination?

Shortest path problem

- The problem: find the **shortest path** from vertex $(0, 0)$ to vertex $(0, X)$ on this DAG…
- $O(V + E)$ solution exists!
  - V is just $10 \times \left(\frac{X}{100}\right)$
  - E is just $3V$.
- Thus this solution is as good as the DP solution!

Non classical dynamic programming problems

Oh man…

Non classical DP problems

- Not the pure form (or simple 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.
- Throughout this lecture, we will talk mostly in *DP terms*
  - **State** (to be precise: "*distinct* state")
  - **Space Complexity** (i.e. the number of distinct states)
  - **Transition** (which entail overlapping sub problems)
  - **Time Complexity** (i.e. num of distinct states * time to fill one state)

The Cutting sticks problem

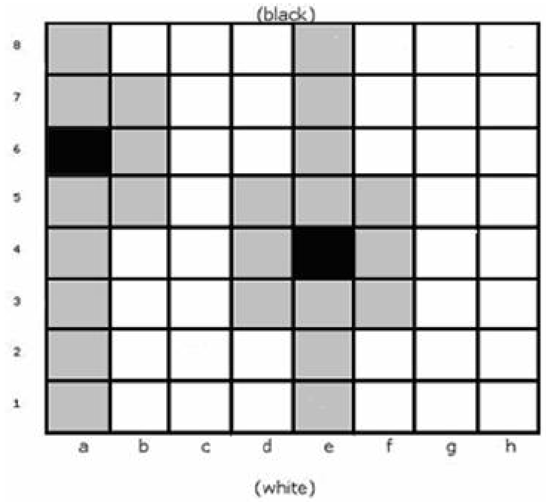
- UVa 10003 – Cutting sticks
- State: index $(\boldsymbol{l}, \boldsymbol{r})$ where $\boldsymbol{l}, \boldsymbol{r} \in [\mathbf{0}..\boldsymbol{n}+\mathbf{1}]$ and $\boldsymbol{l} < \boldsymbol{r}$
  - Q: Why these two parameters?
- Space Complexity: $\boldsymbol{O}(\boldsymbol{n}^{\mathbf{2}})$ distinct states
- Transition: Try all possible cutting points $\boldsymbol{i}$ between $\boldsymbol{l}$ and $\boldsymbol{r}$.
  - i.e. cut $(\boldsymbol{l}, \boldsymbol{r})$ into $(\boldsymbol{l}, \boldsymbol{i})$ and $(\boldsymbol{i}, \boldsymbol{r})$ with cost $(\boldsymbol{A}[\boldsymbol{r}]$ - $\boldsymbol{A}[\boldsymbol{l}])$
- Time Complexity: There are **O(n)** possible cutting points, thus overall $\boldsymbol{O}(\boldsymbol{n}^{\mathbf{2}} \times \boldsymbol{n}) = \boldsymbol{O}(\boldsymbol{n}^{\mathbf{3}})$

DP on DAG

- Dynamic Programming (DP) has a close relationship with (usually implicit) Directed Acyclic Graph (DAG).
  - The **states** are the **vertices** of the DAG.
  - Space complexity: Number of vertices of the DAG.
  - The **transitions** are the **edges** of the DAG.
    - Logical, since a recurrence is always **acyclic.**
  - Time complexity: Number of edges of the DAG.
  - Top-down DP: Process each vertex just once via **memorization.**
  - Bottom-up DP: Process the vertices in **topological order.**
    - Sometimes, the topological order can be written by just using simple (nested) loops.

The injured queen problem

- Like N-queens problem, but the queens are "injured" (can only attack the current column but acts as king otherwise)
- With some of K ($0 \leq K \leq N$) injured queens positions have been predetermined, count how many possible arrangements of the other (N- K) queens so that no two queens attack each other?



DP on math problems

- Some well-known mathematic problems involves DP.
  - Some combinatorics problem have recursive formulas which entail overlapping subproblems.
    - e.g. those involving Fibonacci number, $\boldsymbol{f(n) = f(n-1) + f(n-2)}$.
  - Some probability problems require us to search the entire search space to get the required answer.
    - If some of the sub problems are overlapping, use DP, otherwise, use complete search.
  - Mathematics problems involving **static** range sum/min/max!
    - Use dynamic tree DS for dynamic queries.

Dice throwing problem

- Throw N common 6-sided dice. ($1 \le N \le 24$)
- What is the probability that the sum of all thrown dices is at least x? ($0 \le x \le 150$)
- Basic probability = $\frac{\#\text{Events}}{\text{Sample space}}$
  - Sample space $= 6^{\mathrm{n}}$
  - How to compute #Events?

Dynamic programming issues

- Potential issues with DP problems:
  - They may be disguised as (or looks like) non DP
    - It looks like greedy can work but some cases fails…
      - problem looks like a shortest path with some constraints on graph, but the constraints fail *greedy* SSSP algorithm!
  - They may have subproblems but not overlapping
    - DP does not work if overlapping subproblems not exist
      - Anyway, this is still a good news as perhaps Divide and Conquer technique can be applied.

Dynamic programming issues

- Optimal substructures may not be obvious.
  - Find correct "states" that describe problem.
    - Perhaps extra parameters must be introduced?
  - Reduce a problem to (smaller) sub problems (with the same states) until we reach base cases
- There can be more than one possible formulation.
  - Pick the one that works!

DP problems in ICPC

- The number of problems in ICPC that must be solved using DP are growing!
  - At least one, likely two, maybe three per contest…
- These new problems are **not** the classical DP!
  - They require deep thinking…
  - Or those that look solvable using other (simpler) algorithms but actually must be solved using DP.
  - Do not think that you have "mastered" DP by only memorizing the classical DP solutions!

DP problems in ICPC

- In 1990ies, mastering DP can make you "king" of programming contests…
  - Today, it is a must-have knowledge…
  - So, get familiar with DP techniques!
- By mastering DP, your ICPC rank is probably:
  - from top ~[25-30] (solving 1-2 problems out of 10)
    - Only easy problems.
  - to top ~[15-20] (solving 3-4 problems out of 10)
    - Easy problems + brute force + DP problems.

Be a problem setter

One who sets a trap should know how to disarm one…

Be a problem setter

- Problem solver:
  - Read the problem.
  - Think of a good algorithm.
  - Create a solution.
  - Create tricky I/O test case.
  - WA/TLE: debug!
  - AC!

- Problem setter:
  - Write a good problem.
  - Write a good solution.
    - The correct/best one.
    - The incorrect/slower ones.
  - Set a good I/O test case.
  - Set problem setting (mem/time).
- A problem setter must think from a different angle!
  - By setting good problems, you will simultaneously be a better problem solver!

Problem setter tasks

- Write a good problem:
  - Options:
    - Pick an algorithm, then find problem/story or
    - Find a problem/story, then identify a good algorithm for it (hard).
  - Problem description must not be ambiguous.
    - Specify input constraints.
    - English!
    - Easy one: longer story.
    - Hard one: shorter story.
- Write good solutions:
  - Must be able to solve your own problem!
    - To set hard problem, one must increase his own programming skill!
  - Use the best possible algorithm with lowest time complexity (or memory complexity).
    - Use the inferior ones that barely works to set the WA/TLE/MLE parameters.

Problem setter tasks

- Set a good secret I/O:
  - Tricky test case to check WA verses AC.
    - Boundary cases!
  - Large test case to check TLE/MLE verses AC.
    - Use input generator to generate large test case.
    - Pass this large test case to our solution.
- Set problem settings:
  - Time limit:
    - Usually 2~3 times the timings of your own best solutions.
    - Java is slower than C++!
  - Memory Limit.
  - Problem Name:
    - Avoid revealing the algorithm in the problem name.

Be a contest organizer

- Contest organizer tasks:
  - Set problems of *various* topic.
    - Better set by >1 problem setter.
  - Must balance the difficulty of the problem set
    - Try to make it fun.
    - Each team solves some problems.
    - Each problem is solved by some teams.
    - No team solve all problems.
    - Every teams must work until the end of contest.