UCCPS Beginner Workshop

2022 Season
Episode I: *Introduction & Basics of Dynamic Programming*

What is competitive programming?

- Not just programming *competitively*
- Combination of **problem solving** and **elegant implementation**
- Learn algorithms & data structures
- Improve skills for coding interviews, too!

How do you do competitive programming?

- Usually have a **contest system** *(online judge)*
  - Today in the practical part we'll use **Kattis**
- The system/judge:
  - Shows you a **problem statement**, specifying the input & output
  - Allows you to **submit** a program
  - **Judges** your program and gives you a verdict
    - AC (accepted), WA (wrong answer), TLE (time limit), MLE, RE, …

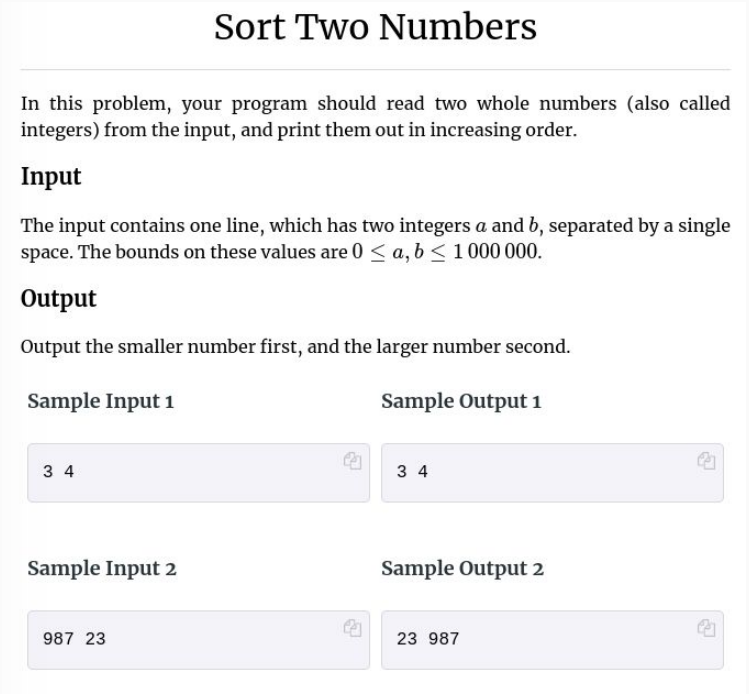
CP Problem morphology

Name (sometimes a letter)

Description (story, problem, ...)

Input/output specification
Limits (sometimes separate)

Samples
(to check your understanding)



Sort Two Numbers

In this problem, your program should read two whole numbers (also called integers) from the input, and print them out in increasing order.

Input

The input contains one line, which has two integers a and b, separated by a single space. The bounds on these values are $0 \leq a, b \leq 1\,000\,000$.

Output

Output the smaller number first, and the larger number second.

Sample Input 1	Sample Output 1
3 4	3 4

Sample Input 2	Sample Output 2
987 23	23 987

Judging solutions

- The main two metrics tested are:
  - Correctness
    - Some amount of programmed **test cases**
    - Often tests are hidden - only samples are known
  - Efficiency
    - Time
    - Memory

Comparing solutions

- Sometimes, a problem may have more than one solution
  - The simplest, slow one ("by definition") is often called **brute force**.
- In that case, often we are interested in the best **efficiency** - time to execute
- How can we sensibly compare **algorithms**?

Comparing algorithms - complexity analysis

- Most commonly computer scientists compare algorithms by **computational complexity**
- Ignores constant factors
- Usually considers the worst case (also: best case, average case)
- We make some simplifications
  - *Elementary operations* take some 1 unit of time (arithmetic, logic, etc.)
- We care about the *problem size* on input
  - For example, we may have a an array of size *k* or a number *N* (well, akshually...) on input
  - Ignore how many bytes e.g. each array element is - that's a constant

Quick introduction to Big O

- We'll use the **Big O** notation to simplify an algorithms' **time complexity**
- If we take *exactly* $2n^3 + 3n^2 + 17$ units of time, we just write $\boldsymbol{O(n^3)}$
  - Most significant term without constant factor
  - More formally: a function that bounds the time as $n \to \infty$ with some constant
- Rule of thumb - *10^7* elementary operations could be about a second.
  - Subject to what we take to be an *elementary operation*
  - This relates to the *hidden constant factor* of some algorithms
    - Better complexity != quicker on your data

Example Big O's

```
for i in range(n):
    total += i
```

We loop & add *n* times ⇒ ***O(n)***

```
for i in range(n):
    for j in range(k):
        total += i * j
```

n times we loop & add *k* times ⇒ ***O(nk)***

```
for i in range(n):
    for j in range(5):
        for x in range(j):
            total += i * j * x
```

n times we loop *5* times, then at most 5 times
⇒ *O(n * 5 * 5)* = ***O(n)***, as we drop the constants

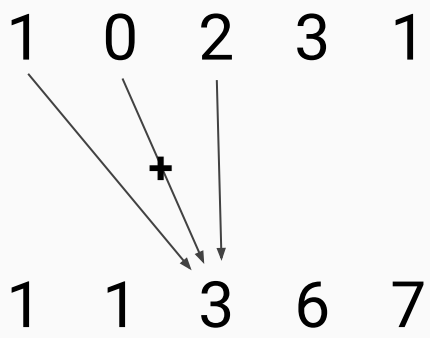
Dynamic programming

Let's look at prefix sums first

- We'll start with looking how one might compute *prefix sums*
- Prefix sums are a **really** useful concept for plenty of practical problems
  - Come up when
    - We are dealing with changes over a time period
    - More abstractly, we have range sum queries on an array

- Say you earn some amount of money on each day of your software engineer internship.
- How could you quickly tell how much you earnt until the *i*-th day?

Money earnt per day



Money earnt until day

Computing prefix sums

- We sum over all money earnt until the *i*-th day.

```
sums[i] = day[0] + day[1] + ... + day[i]  # O(n²) for all
```

- However, there's a **repeating substructure**
- The earnings until the *i*-th day start with the earnings until the *i - 1*-st

```
sums[i] = sums[i - 1] + day[i]  # O(n) for all
```

Side note - range sum queries

- With prefix sums, we can answer prefix sum queries.
- The money earnt between day *i* and *j* is:

```
day[i] + day[i+1] + … + day[j] = sum[j] - sum[i - 1]
```

- This makes prefix sums quite useful as an **auxiliary** structure.

Dynamic programming is about **repeating subproblems**.

EXAMPLE PROBLEM

You have N boxes, each with some amount of money.

You can take as many boxes as you want, but you can't take any two boxes that are next to each other.

What is the most money you can get?

>UC::CPS

The example problem from our stall @ Fresher's fair 2022!

The box problem

- Formally, find the **maximum sum non-adjacent subsequence**
  - The sequence is the list of cash amounts in each following box
- **Brute force** - consider all subsequences – *O(2^n)*
  - Check if they have no adjacent boxes – *O(n)*
  - Sum up the present values – *O(n)*
- That has *O(n 2^n)* complexity.
  - We can do **much** better.
  - And without any heuristics like tree search or **greedily** taking some boxes.

Solving the box problem

- The optimal solution is some subsequence of *a*
- Consider the following **two cases**:
  - 0. The **last** element of *a* **isn't** in the optimal solution
    - Throw it out and solve the problem without it.
  - 1. The **last** element of *a* **is** in the optimal solution
    - We can't use the second element in this case
    - Throw both out and solve the problem

```
a = [10, 1, 2, 9, 15, 9]

a0 = [10, 1, 2, 9, 15]

+ £9, a1 = [10, 1, 2, 9]
```

What have we just done?

- We've shown that solving the box problem for *n* boxes can be **reduced**
  - We need to know the solutions for *n - 1* and *n - 2* boxes
- We call the subproblems the **states**, and the reductions **transitions**
  - **States** - prefix (some first *k* boxes)
  - **Transitions** - for some state *k*, we either
    - take the last element of the prefix (go to *k - 2*)
    - or we don't (*k - 1*)

Writing down the solution

Say *opt[n]* is the optimal solution for the first *n* boxes.

Of the two possible transitions we pick the higher result.

```
opt[n] = max(
    opt[n - 1],                 # don't take last
    opt[n - 2] + box[n - 1]  # take last
)
```

Leveraging repeated substructure

- We only gave recursive equations to compute the result *opt[n]*
- However, dynamic programming **leverages repeated subproblems**
- We cannot solve a subproblem multiple times, as then we just consider all cases (all subsequences) like in brute-force

```
def opt(n):
    return max(opt(n - 1), opt(n - 2) + box[n - 1]) if n else 0
```

Memoization

- The way to use repeated subproblems is **memoization** of the results
- Basically, whenever we solve a subproblem we save the result
  - ⇒ each subproblem may only be solved once
- Often, dynamic programming is implemented as for-loops filling an array.
  - Sometimes, for tricky states and transitions, it's convenient to use recursion.

```
opt[0] = 0
for i in range(1, n+1):
    opt[i] = max(opt[i - 1], opt[i - 2] + box[i - 1])
```

Efficiency of dynamic programming

- We've solved the box problem in O(n) – linear – time, in comparison to the *O(n 2^n)* of the brute force approach.
  - For a million boxes this is the difference between a fraction of a second and finishing way past the lifetime of the universe.
- Sometimes we need such huge datasets
  - A naive Longest Common Subsequence algorithm could take months/years when working on a genetics dataset
  - Heuristic algorithms can do this way faster (though they man not be optimal or they only behave well on some data)

The dynamic programming approach

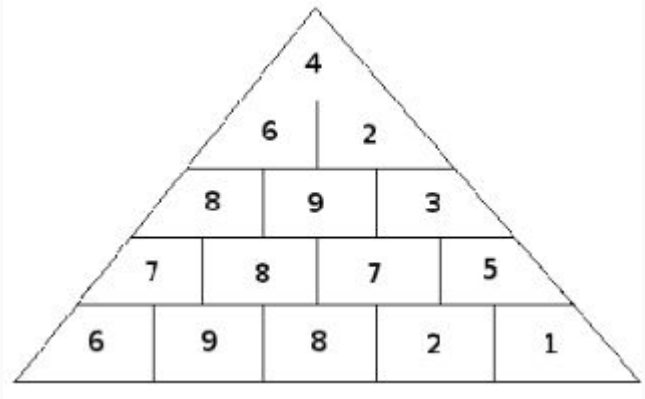
1. Pick your **states** and **transitions**.
2. Find the **base cases** (when do transitions fail? where do you start?)
3. Extract the result from the **final state** (representing the entire problem)
4. (Usually) order the states to loop through them and compute their results.

Pyramid problem

You're in a pyramid filled with **gold coins**. Starting at the **base** (in any room) you want to get to the very top, **collecting coins** in the rooms you pass.

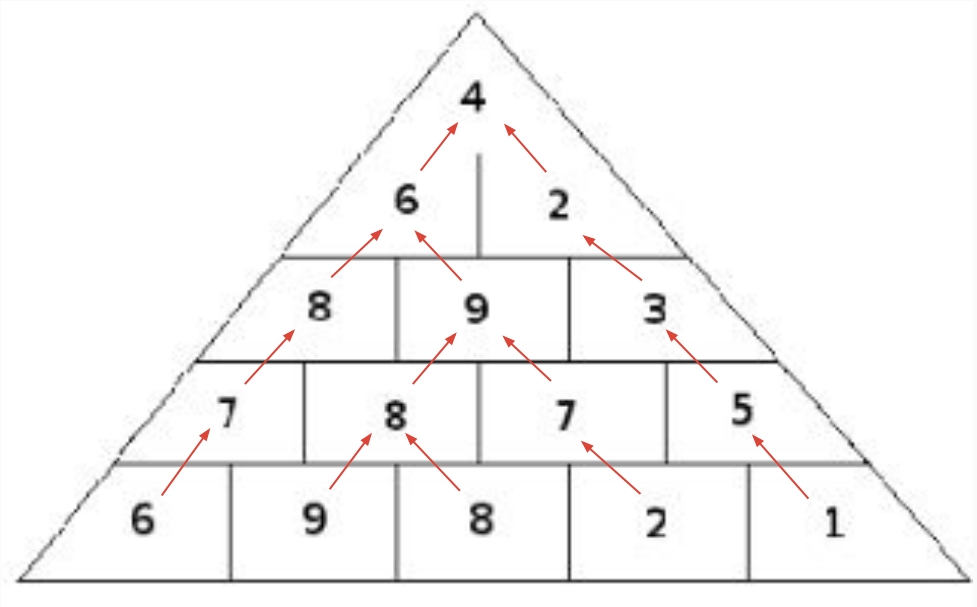
You can only proceed to the **two rooms above you** (not sideways or downstairs).

How many coins can you collect?



Brute forcing the pyramid

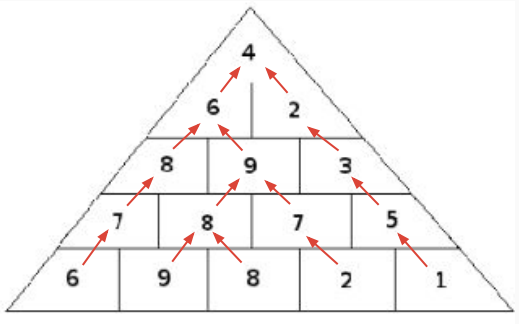
- Once again, there's a simple brute force approach
- For *n* floors of the pyramid, we have *O(2^n)* paths we process in *O(n)*.
- But… there's plenty of repeating subproblems!



Optimal paths to follow in the pyramid

Pyramid problem solution

- States
  - *(i, j)* for the optimal solution starting in row *i* and column *j*
- Transitions
  - From (i, j) we go up to the left or the right
  - We add the coins in the current position
- Base case
  - Sometimes no left/right room
  - The top has no rooms to proceed to
- Ordering
  - e.g. bottom to top, then left to right



Let's get coding!

- Check out the Discord for a workshop 'contest' link
- Solve through the problems (probably best in order)
- Whenever you get stuck - make sure to ask us for help! (raise your hand, come up, etc.). This is the best way to get the most out of the workshop!
- We'll get pizza at the end ;-)