Club Introduction & Dynamic Programming

Shang Zhou & Huize Mao

Meeting 1. October 7, 2024

Competitive Programming Club @ UC San Diego

What is competitive programming?

- Art of solving algorithmic problems in an efficient way
- Solutions should not only always be correct, but also fast
- Usually we are looking for deterministic algorithms finding the best solution
- The algorithm is tested on prepared test-cases, meaning that not only the solution has to be correct and fast, but it also has to be implemented without flaws

What is ICPC?

- ICPC – International Collegiate Programming Contest
- Organization preparing world-wide competitions in competitive programming on collegiate level

Format:

- 10-13 problems
- Teams of 3 people, but only with one computer
- Each correct solution is 1 point, total time of submissions is the tie-breaker
- Every incorrect solution – 20 minutes penalty
- Teams qualifying starting with regional contests, ending with the World Finals

Competitions

SoCal – happening every fall in Riverside
- UCSD sends 5-7 teams

NAC/World Finals
- Only the best UCSD team from SoCal participates
- Challenging but rewarding competitions

Other (less serious, more for practice/fun)
- CALICO
- USACO

UCSD Team selection
- October 19
- More information: https://icpc.ucsd.edu/team-selection/
- We will have one more mock practice this Sunday October 13

Platforms

Codeforces

- Great CP platform with varying difficulty of problems and contests
- A lot of blogs and tutorials – good place to learn

Leetcode

- Focused on interview prep, but still pretty good quality problems
- A lot of beginner materials

Vjudge

- We will use this one for our meetings – make sure to create an account!
- More resources: https://icpc.ucsd.edu/resources/

Officers

- Shang, Huize, Raymond, Manu, Julie, Ben, Qihao
- Adviced by CSE & HDSI professor Jingbo
- We are here to help you – please ask us any questions you have, both regarding logistics and competitive programming
- https://icpc.ucsd.edu/teams/

Meeting plan

- Mondays 4-6 p.m. at CSE Basement
- 4:00 – 5:45 p.m. Tutorials + Practice
- 5:45 – 6:00 p.m. Pizza

Dynamic Programming

Longest Increasing Subsequence

Given a sequence of numbers a = a[1], a[2], …, a[n], we say a′ is a subsequence of a if it can be obtained by deleting some (possible zero) elements from a.

- Example: a = [5, 1, 8, 1, 9, 2]
- [5, 8, 9] is a subsequence
- [1, 1] is a subsequence
- [5, 1, 8, 1, 9, 2] is a subsequence
- [] is a subsequence
- [8, 5] is **not** a subsequence
- [10] is **not** a subsequence

Longest Increasing Subsequence

An increasing sequence is such that the elements are in (strictly) increasing order

Problem: What is the length the Longest Increasing Subsequence (LIS) of a.

- [5, 8, 9] and [1, 8, 9] are the longest increasing subsequences of a = [5, 1, 8, 1, 9, 2]

Naive algorithm: There are 2^n subsequences, check if each is increasing. The complexity is $O(n2^n)$.

What about dynamic programming?

Longest Increasing Subsequence

1. Define subproblems
   Let dp[i] represent the length of the longest increasing subsequence ending at index i
   Our goal is to calculate max{dp[i]}
2. Formulate recurrence
   dp[i] = 1 + max{dp[j]} for all j < i where a[j] < a[i]
3. Solve base cases
   dp[0] = 0
4. Time Complexity: $O(n^2)$

Longest Increasing Subsequence: O(n log n)

1. Maintain a list tails where:
   tails[i] is the minimum value that ends a subsequence of length i
2. Update tails using the current element with Binary Search
   For each lament a[i], use Binary Search to find its position in tails.
   If element is greater than all in tails, append it
   Otherwise, replace the element in tails to maintain the smallest possible ending value
3. Final result: Length of tails in the length of the LIS

0-1 Knapsack

We have knapsack with maximum capacity W.

There are n available items. The i-th item has weight w[i] and gives us a value v[i].

Problem: What is the maximum value we can hold in our knapsack?

- Example: $n = 4$, $W = 10$, $w = [1, 3, 7, 8]$, $v = [20, 5, 50, 40]$
- [3, 4] doesn't work, weight is $15 > 10$.
- [2, 3] fits in the knapsack, but value 55 is not optimal.
- [1, 4] is the solution with value 60.

0-1 Knapsack

1. Approach
   Use a dp array to store the best achievable value for each capacity

2. dp[j] = max(dp[j], dp[j - weight[i]] + value[i])
   (iterating from high capacity to low)

3. Complexity: O(nW), where n is the number of items, and W is the capacity

0-1 Knapsack with Bitset

1. Use bitset to optimize memory and speed
2. Each bit in the bitset represents whether a certain weight is achievable
3. Use bitwise operations to update possible weights efficiently:
   Initialize: bits[0] = 1
   For each item: bits |= bits << weight[i]
4. The result: Bit at position j indicates if weight j is achievable

More DP Categories

DP on Trees
DP Using Bitfield
Digit DP
Monotonic Queue Optimization Trick
Slope Optimization Trick

Practice Problems

Today's practice problems link:

https://vjudge.net/contest/661444