Week 5: Lecture 9 - Lecture 10

#greedy #adhoc #string

Lecture 9

(1) Scheduling problem
Optimal strategy: takes the class that ends first
Class that ends earlier provides more spaces for the remaining choices

Adhoc problems don't require any knowledge of data structures and algorithms. They just need some observations.

(2) Min number of coins that sums up to n
Coins = {1, 5, 10}
N = 12
Min number of coins = 10 + 1 + 1 ==> 3 coins

This is NOT a greedy problem.
Counter example: coins = {1, 3, 4} and number = 6. Using greedy approach to select maximum value first, will use 3 coins: 4 + 1 + 1. Optimal solution is 3 + 3 using 2 coins.

(3) Even but not even
Problem - 1291A - Codeforces
observations:

- the ending digit must be odd. The number itself must be odd
- there should be at least 2 odd numbers

Algo:
Scan from begin to the end, select two odd numbers in order and print them.

Tweak: minimum number of removals to make a number EBNE

- Anything on the RHS of the last odd number must be removed
- on the LHS, if the number of odd elements is even (not including the last digit), remove one odd element

(4) Neighboring grid
Problem - 1375B - Codeforces

observations:

- Any cell value should not be greater than 5. (Neighboring max = 4)
- The following matrix is a good matrix, with each cell value equals to its maximum possible value

  [2 3 3 2]

  [3 4 4 3]

  [3 4 4 3]

  [2 3 3 2]

  Just need to *transform* the random matrix to the above matrix (extreme state)

(5) Zero array

Problem - 1201B - Codeforces

observations:

- sum of elements should be even
- problem equals to constructing a special sequence

  E.g. 10 10 2 2 2 2 2

  Treat 10 as {a, a, a, …… a} 10 * a

  The second 10 as {b, b, …. b} 10 * b

  …

  A sequence is constructed as

  ```
  A _ A _ A _A
    B.  B.  B.  B
  ```

  Where 2 neighboring elements are not repeated.

  If such a sequence is possible, the answer would be YES, otherwise NO.

  Todo: try out my own idea of this problem (matching max and min elements)

Lecture 10

#string #hashing

How it works:

Transform string to integer –> and compare equality of integers

Need to satisfy two cases:

(1) s1 == s2: hs(s1) == hs(s2) 100% probability

(2) s1 != s2: hs(s1) != hs(s2) high probability

Polynomial rolling hash

How it works

E.g. abcd where a ➡ 1; b ➡ 2; c ➡ 3; d ➡ 4 … use 26 letters

Then use base 31 to compute a numeric value

e.g. abcd = 1234

The numeric value is computed as 4 * 31^0 + 3 * 31 ^ 1 + 2 * 31 ^ 2 + 1 * 31 ^ 3

Computing hashes for prefix

hash('a') = some value

hash('ab') = hash('a') * 31 + 2

hash('abc') = hash('ab') * 31 + 3

Computing hashes for any substring

Method:

Compute hashes for all prefixes O(n)

Compute B^N

Time takes to do any query: O(1)

For substring s[I:j] inclusive, the hash(s[I:j] = hash(s[:j]) - hash[s[:I)] * B ^ (difference of length in string = j - I + 1)

The reason to take the power of difference of length is because the hash value of substring needs to be the same regardless of the position it appears.

E.g.

Code: cp24/hash.cpp at master · TheSithPadawan/cp24 · GitHub

Analysis

Probability of getting a hash collision = 1/Mod

Ways to reduce hash collision:

1. Use a larger MOD
2. Using 2 MODS, this reduces the probability to 1/Mod^2, the hash result becomes a tuple, only pass if two elements match

Applications

1. Finding substring of T to S.

```
step 0: pre-compute hashes of prefix of S
step 1: compute hash(T)
step 2: enumerate S for string size of T.size(), check if any matches hash(T)

Time complexity: O(m+n)
```

3. Given a string S, find the longest substring that occurs twice
   code:
5. Given a string S and q queries, at each query answer if S[I, j] is a palindrome

```

Compute prefix hashes of S and S' (reverse of S)
For each query check for the equality of the hashes of two substrings
```

6. Given string S, determine the longest prefix that is also the longest suffix

```

E.g. in string aabcdaab, the prefix aab is also the suffix aab

Based on the length of prefix, compute the start index from the end of string,
and check for the equality of the hashes of two substrings.
```

7. Longest palindromic substring
   Idea:
   (1) quadratic algorithm O(n^2). For each index, expand from middle twice depending on odd or even
   (2) manacher's algo O(nlg n)
   (3) rabin-karp
   property: center at index I, if from index I, a palindrome is of length L, then a cutting off the two sides of this substring is also a palindrome. This gives the monotonic characteristic for using binary search.
   For each index, find the max length (half string) that it can extend on left and right, do a binary search to find the optimal answer. O(log n) time
   Since there are O(n) such centers, the overall time complexity is O(n log n)