M1711 Word Puzzle

Theo

Description

- Given a N*N grid and a keyword, find out all positions so that the specified key can be found starting at those positions

- N <= 10

Solution

- Implementation!

- Just follow what you're asked to do
- There are at most N^2 positions to check
- Each position requires at most 8*N steps to check
- O(N^3)

Solution

- How to implement it easily?
- Write 8 directions separately

  Quite Long!!

  (And easy to have bugs)

```
bool check(int px, int py){
    int k = strlen(s);
    int cx = px, cy = py, ok = 1; // current checking pos on board
    for (int i = 0; i < k; i++){
        if (a[cx][cy] != s[i]){
            ok = 0;
        }
        cx--; // going up
    }
    if (ok) return true;

    cx = px; cy = py; ok = 1;
    for (int i = 0; i < k; i++){
        if (a[cx][cy] != s[i]){
            ok = 0;
        }
        cx++; // going down
    }
    if (ok) return true;

    cx = px; cy = py; ok = 1;
    for (int i = 0; i < k; i++){
        if (a[cx][cy] != s[i]){
            ok = 0;
        }
        cy--; // going left
    }
    if (ok) return true;
```

Solution

- Use a forloop!
- Define the directions using array of pairs of integers
- Use one forloop for 8 directions

```
int dx[9] = {-1, -1, -1, 0, 0, 1, 1, 1};
int dy[9] = {-1, 0, 1, -1, 1, -1, 0, 1};

bool check(int px, int py){
    for (int dir = 0; dir < 8; dir++){
        int cx = px, cy = py, ok = 1; // current checking pos on board
        for (int i = 0; i < k; i++){
            if (a[cx][cy] != s[i]){
                ok = 0;
            }
            cx += dx[dir];
            cy += dy[dir]; // add the corresponding transition vector
        }
        if (ok) return true;
    }
    return false;
}
```

Harder version

- Can you solve the problem if N <= 1000?

M1712
Subsequence Product

Author: Anson Ho

Statement

- Given a sequence (x_i) with length N
- Count the subsequences with product = M
- Answer mod 10^9+7

- $1 <= N <= 500$
- $1 <= M, x_i <= 10^{12}$

Subtask 1

- $1 <= N <= 20$

- Exhaustion
- $O(2^N)$

Subtask 2

- 1 <= M, x_i <= 5000

- Dynamic programming
- dp[i][j] = # of subsequences of (x_1, x_2, …, x_i) with product = j
- Dimension for i can be “reduced” by updating from larger j
- O(NM)

Full solution

- Only consider dp[i][j] when j is a factor of M
- $d(j)$ = # of factors of j
- max $d(j)$ = $d(963761198400)$ = 6720
  - Keyword: highly composite number
- Discretization (or map in C++) required
- $O(Nd(M)\log(d(M)))$
- Can be improved to $O(Nd(M))$

Thank you

M1713 Biscuit Clicker

Problem by: Charlie Li

Problem Statement

- When the production rate is P, Alice will get a biscuit every 1/P seconds
- The basic production rate is 1
- There are N upgrades where every upgrade can be bought at most once
- For the i^{th} upgrade, the cost is C_i and it multiply the production rate by P_i
- Find the fastest way to collect K biscuits in the bank

Subtasks

$1 <= N <= 100000$, $1 <= K$, $C_i <= 10^{12}$, $1 <= P_i <= 1000$

- Subtask 1 (5 points): $N = 1$, $P_i <= 8$
- Subtask 2 (10 points): $N <= 10$, $P_i <= 8$
- Subtask 3 (15 points): $N <= 15$, $P_i <= 8$
- Subtask 4 (10 points): All C_i are the same
- Subtask 5 (15 points): All C_i and P_i are non-negative powers of 2
- Subtask 6 (45 points): No additional constraint

Observation

- This is trivial but important:
- Given a sequence which is the order of buying upgrades, we will buy the next upgrade immediately when we have enough biscuits

Subtask 1

- Subtask 1 (5 points): N = 1, P_i <= 8
- Since N = 1, we only need to check whether we should by the upgrade
- Time needed if we do not buy = K
- Time needed if we buy the upgrade = $C_1 + \frac{K}{P_1}$
- We just need to compare these two time and then subtask 1 is solved

Subtask 2

- Subtask 2 (10 points): N <= 10, P_i <= 8
- For this subtask, we can check all permutations of buying upgrades
- For C++ users, next_permutation is a very nice function and help a lot
- Time complexity: O(N!)

Subtask 3

- Subtask 3 (15 points): N <= 15, P_i <= 8
- For this subtask, instead of checking all permutation one by one, we can use dp to speed up
- Time complexity: $O(N^2 2^N)$
- Note: this skill is called bitwise DP and will be taught in DP(III)

Subtask 4

- Subtask 4 (10 points): All C_i are the same
- For this subtask, we need to first sort all the upgrades by descending P_i
- What we need to do next is to check where to stop
- That is, we need to check $\frac{C_i}{\prod_{j=1}^{i-1} P_j} + \frac{K}{\prod_{j=1}^{i} P_j} < \frac{K}{\prod_{j=1}^{i-1} P_j}$
- Indeed, the productivity other than the upgrade we concerning is not important as they are the common denominator so we can simplify the formula to $C_i + \frac{K}{P_i} < K$

Full Solution

- The above observation is also true when considering the order of upgrades
- If we need to buy upgrade i and upgrade j, we will buy upgrade i first if

$$C_i + \frac{C_j}{P_i} < C_j + \frac{C_i}{P_j}$$

- And, we will buy upgrade i if $C_i + \frac{K}{P_i} < K$
- Therefore, we can solve this problem using greedy, sort the upgrades according to the first inequality and buy those upgrades satisfying the second upgrade
- Proofs are often the most difficult part of a greedy question, they are at the back (a bit rough)

Note

- If $C_i + \frac{K}{P_i} = K$, both buying or not buying upgrade i is acceptable
- For any optimal solution $\{a_1, a_2, \ldots, a_n\}$,
  - If $C_{a_i} + \frac{C_{a_{i+1}}}{P_{a_i}} = C_{a_{i+1}} + \frac{C_{a_i}}{P_{a_{i+1}}}$, swapping a_i with a_{i+1} is also a optimal solution
- Example:
  - Consider the input:

    3 36

    27 4

    9 7

    7 3
  - There are total 4 optimal solutions, can you find out all?

Code

```
…
bool cmp(int x, int y) {
        return c[x] + 1. * c[y] / p[x] < c[y] + 1. * c[x] / p[y];}
…
int main() {
        …
        sort(a, a + n, cmp);
        for (int i = 0; i < n; i++) if (c[a[i]] + 1. * t / p[a[i]] < t) ans[l++] = a[i];
        …
}
// precision is enough for a 0.001 gap
```

Ineligibility of Upgrades

- Statement: Buying upgrade i does not give a optimal solution iff $C_i + \frac{K}{P_i} > K$
- Proof: Suppose we have bought some upgrades (possibly zero) which cost us a total of T seconds and the overall productivity is P biscuits per second
- If we do not buy upgrade i, then the total time we need to get K biscuits in bank will be $T + \frac{K}{P}$
- If we do buy upgrade i, then the total time we need to get K biscuits in bank will be $T + \frac{C_i}{P} + \frac{K}{PP_i}$
- Buying upgrade i gives a worse solution iff $T + \frac{C_i}{P} + \frac{K}{PP_i} > T + \frac{K}{P}$
  - which is equivalent to $C_i + \frac{K}{P_i} > K$
- Therefore, the statement is true
- We can immediately conclude that those upgrade with $P_i = 1$ is useless

Possibility of Sorting Upgrades

- Statement: If all $P_i > 1$, than there exist at least one permutation $\{a_1, a_2, \ldots, a_n\}$ such that

$$C_{a_i} + \frac{C_{a_{i+1}}}{P_{a_i}} \leq C_{a_{i+1}} + \frac{C_{a_i}}{P_{a_{i+1}}} \; for \; i = 1 \ldots n-1$$

- Proof:

$$C_{a_i} + \frac{C_{a_{i+1}}}{P_{a_i}} \leq C_{a_{i+1}} + \frac{C_{a_i}}{P_{a_{i+1}}} \leftrightarrow C_{a_i} - \frac{C_{a_i}}{P_{a_{i+1}}} \leq C_{a_{i+1}} - \frac{C_{a_{i+1}}}{P_{a_i}}$$

$$\leftrightarrow \frac{C_{a_i}}{1 - \frac{1}{P_{a_i}}} \leq \frac{C_{a_{i+1}}}{1 - \frac{1}{P_{a_{i+1}}}} \leftrightarrow \frac{C_{a_i} P_{a_i}}{P_{a_i} - 1} \leq \frac{C_{a_{i+1}} P_{a_{i+1}}}{P_{a_{i+1}} - 1}$$

- Therefore we can obtain the permutation by sorting with $\frac{C_i P_i}{P_i - 1}$

Correctness of Answer

- Statement: the optimal way of buying n upgrades is using a permutation $\{a_1, a_2, \ldots, a_n\}$ where $C_{a_i} + \frac{C_{a_{i+1}}}{P_{a_i}} \leq C_{a_{i+1}} + \frac{C_{a_i}}{P_{a_{i+1}}} \; for \; i = 1 \,..\, n-1$
- Proof:
- Suppose $C_{a_i} + \frac{C_{a_{i+1}}}{P_{a_i}} > C_{a_{i+1}} + \frac{C_{a_i}}{P_{a_{i+1}}}$ for some i = 1 .. or n – 1
- We can swap a_i with a_{i+1} and we can obtain a better solution
  - since all others are unchanged but we buy upgrade a_i and upgrade a_{i+1} faster
- We cannot find a better solution iff $C_{a_i} + \frac{C_{a_{i+1}}}{P_{a_i}} \leq C_{a_{i+1}} + \frac{C_{a_i}}{P_{a_{i+1}}} \; for \; i = 1 \,..\, n-1$
- Therefore, it is a optimal solution.

M1714 Different Subarrays

Problem by: Charlie Li

Problem statement

- Given a sequence of length N, an integer M and K queries.
- For each query i, find the number of different subarrays within L_i and R_i (inclusively) such that it contains at least M distinct values.
- You can get 50% score if you can check whether such subarray exist for every query

Subtasks

- $1 <= M <= N <= 10^5$, $1 <= K <= 10^5$, $1 <= A_i <= 10^9$ and $1 <= L_i <= R_i <= N$
- Subtask 1 (10 points): $1 <= N, K, A_i <= 10$
- Subtask 2 (10 points): $1 <= N, K <= 100$, $1 <= A_i <= 10^6$
- Subtask 3 (20 points): $1 <= N <= 1000$
- Subtask 4 (10 points): All A_i are distinct
- Subtask 5 (50 points): No additional constraint

Subtask 1

- Subtask 1 (10 points): 1 <= N, K, A_i <= 10
- For every query, for every subarray, we check each A_i to see whether it appear in the subarray and count if it does
- Time complexity: $O(KN^2 \max A_i)$

Subtask 2

- Subtask 2 (10 points): 1 <= N, K <= 100, 1 <= A_i <= 10^6
- We can use a frequency array to store the occurrence of A_i and we can speed up
- Also after we have checked the subarray i to j, we can use the same frequency array to check the subarray i to j+1 and so on
- Time complexity: $O(KN^2)$

Subtask 3

- Subtask 3 (20 points): 1 <= N <= 1000
- Although A_i can be very large, but many of the possible value will not appear, so we can discretize them
  - This can be done using std::map<int, int> or sort all A_i and use binary search
- Then we can precompute for every subarray, whether it has at least M distinct values
- Next, we can use a 2D partial sum (aka, prefix sum) to store the result and can do O(1) query
- Time complexity: $O(N^2 \log(\max A_i) + K)$

Subtask 4

- Subtask 4 (10 points): All A_i are distinct
- We can use formula to calculate the answer for this subtask
- For every query L_i, R_i, the answer is

$$\max(0, \frac{(R_i - L_i - M + 3)(R_i - L_i - M + 2)}{2})$$

Observation

- Noted that if subarray i to j contains at least M distinct values, than subarray i to j+1 must contain at least M distinct values
- So for every i, we can find the minimum j (if such j exist) such that it contains at least M distinct values

Observation

- Code:

```
for i = 1 to N
 while j <= N and d < M
        if c[A[j]] = 0
              d++, j++
        c[A[j]]++
 f[i] = j
 if c[A[i]] = 1
        d--
 c[A[i]]--
```

Observation

- Then for each query L_i, R_i, we can find the answer by looking at the array f[]
- Code:

```
for j = Li to Ri
    if f[j] <= Ri
        ans += Ri – f[j]
```

- Noted that f[] is increasing, so we can use binary search to find some point x such that $f[j] <= R_i$ if $j <= x$ and $f[j] > R_i$ if $j > x$
- Then the answer will be come $(x – L_i + 1) * (R_i + 1) – (f[L_i] + f[L_i + 1] + … + f[x])$
- For the summation part, we can use partial sum (aka. prefix sum) to calculate quickly

Full solution

- The overall solution for this question is
- First, discretize the array A using std::map<int, int> or binary search or sth else
- Then precompute all f[i], where f[i] is the smallest index such that the subarray from i to f[i] contain at least M distinct values
  - If such f[i] does not exist, we set it to N + 1 so as to preserve the increasing property
- After that, calculate the prefix sum p[] of f[] (need long long)
- For each query L_i, R_i, use binary search to find x such that f[j] <= R_i if j <= x and f[j] > R_i if j > x
- Then the answer of the query is just $(x - L_i + 1) * (R_i + 1) - (p[x] - p[L_i - 1])$
- Time complexity: $O(N\log(\max A_i) + K\log N)$