卐 Short Revision Notes 卐

Sourabh Aggarwal (sourabh23)

Compiled on December 10, 2018

Contents

Think twice code once!

1 Maths

1.1 Game Theory

Games like chess or checkers are partizan type.

1.1.1 What is a Combinatorial Game?

1. There are 2 players.
2. There is a set of possible positions of Game
3. If both players have same options of moving from each position, the game is called impartial; otherwise partizan
4. The players move alternating.
5. The game ends when a postion is reached from which no moves are possible for the player whose turn it is to move. Under **normal play rule**, the last player to move wins. Under **misere play rule** the last player to move loses.
6. The game ends in a finite number of moves no matter how it is played.

P - Previous Player, **N** - Next Player

1. Label every terminal position as P - postion
2. Position which can move to a P position is N position
3. Position whole all moves are to N positoin is P position.

Note: Every Position is either a P or N.

Directed graph G = (X, F), where X is positions (vertices) and F is a function that gives for each $x \in X$ a subset of X, i.e. *followers of* x. If $F(x)$ is empty, x is called a terminal position.

$g(x) = min\{n \geq 0 : n \neq g(y) \text{ for } y \in F(x)\}$

Positions x for which $g(x)$ is 0 are P postions and all others are N positions.

4.1 The Sum of n Graph Games. Suppose we are given n progressively bounded graphs, $G_1 = (X_1, F_1), G_2 = (X_2, F_2), \ldots, G_n = (X_n, F_n)$. One can combine them into a new graph, $G = (X, F)$, called the **sum** of $G_1, G_2, \ldots, G_n$ and denoted by $G = G_1 + \cdots + G_n$ as follows. The set X of vertices is the Cartesian product, $X = X_1 \times \cdots \times X_n$. This is the set of all n-tuples $(x_1, \ldots, x_n)$ such that $x_i \in X_i$ for all i. For a vertex $x = (x_1, \ldots, x_n) \in X$, the set of followers of x is defined as

$$\begin{aligned} F(x) = F(x_1, \ldots, x_n) = {} & F_1(x_1) \times \{x_2\} \times \cdots \times \{x_n\} \\ & \cup \{x_1\} \times F_2(x_2) \times \cdots \times \{x_n\} \\ & \cup \cdots \\ & \cup \{x_1\} \times \{x_2\} \times \cdots \times F_n(x_n). \end{aligned}$$

Theorem 2. *If g_i is the Sprague-Grundy function of G_i, $i = 1, \ldots, n$, then $G = G_1 + \cdots + G_n$ has Sprague-Grundy function $g(x_1, \ldots, x_n) = g_1(x_1) \oplus \cdots \oplus g_n(x_n)$.*

Thus, if a position is a **N** position, we can cleverly see which position should we go to (which component game move to take) such that we reach **P** position.

2 Graphs

2.1 Tree

Undirected, acyclic, connected, $|V| - 1$ edges.

All edges are bridges, and internal vertices (degree > 1) are articulation points.

It is as well a bipartite graph.

SSSP: Simply take the sum of edge weights of that unique path. $O(|V|)$

APSP: Simply do SSSP from all vertices. $O(|V^2|)$

```
void preorder (v) {
  visit (v);
  preorder (left (v));
  preorder (right (v));
}
void inorder (v) {
  inorder (left (v));
  visit (v);
  inorder (right (v));
}
void postorder (v) {
  postorder (left (v));
  postorder (right (v));
  visit (v);
}
```

It is **impossible** to construct binary tree with just Preorder traversal.
It is **impossible** to construct binary tree with just Inorder traversal.
It is **impossible** to construct binary tree with just Postorder traversal.

2.1.1 Important Problems

- UVA 11695 Sol: Problem Desc: Find which edge to remove and add so as to minimise the number of hops to travel between flights. Problem Sol: Just link the center of diameters. Brute force which edge to remove.
- UVA 112 Sol, UVA 112 Prob: Just see how I processed the input.
- UVA 10029 Sol, UVA 10029 Prob: Edit steps, (lexicographic sequence of words)
- UVA 536 Sol, UVA 536 Prob: Construct binary tree with preorder and inorder
- UVA 10459 Sol, UVA 10029 Prob: Edit steps, (lexicographic sequence of words)

3 Some Basic

```
while (first || cin >> temp) {  // something }
```

4 Strings

To map keyboard etc, it is better to create 2 strings then loop through and map.
To transform complete string to lowercase:

```
transform (word.begin (), word.end (), word.begin (),
::tolower);
```

To concatenate two vectors:

```
vector1.insert (vector1.end (), vector2.begin (), vector2.end
());

string.substr (startposn, length); // Where startposn is 0
indexed.

int pos1 = line.find ("U=");
if (pos1 != -1) { // process }
line.replace (pos, len, newString); // pos = line.find (f), len
= f.size ()
```

We can iterate through all substrings of string $O(n^2)$ and see which all of them are palindromes in $O(n^3)$ or in $O(n^2)$ by using dp ($dp[startpos][endpos] = (s[startpos] == s[endpos] \&\& dp[startpos+1][endpos-1])$ or hash.

4.1 Minimum Edit Distance

```
void fillmem() {
   for (int j = 0; j <= a.size(); j++) mem[0][j] = j;
   for (int i = 0; i <= b.size(); i++) mem[i][0] = i;
   for (int i = 1; i <= b.size(); i++) {
       for (int j = 1; j <= a.size(); j++) {
           if (a[j - 1] == b[i - 1]) mem[i][j] = mem[i - 1][j -
           1];
           else mem[i][j] = min(mem[i - 1][j - 1], min(mem[i -
           1][j], mem[i][j - 1])) + 1;
       }
   }
    // mem[b.size ()][a.size ()] contains the answer
}
void print() {
   int i = b.size(), j = a.size();
   while (i || j) {
       if (i and j and a[j - 1] == b[i - 1]) { i--; j--;
       continue; }
       if (i and j and mem[i][j] == mem[i - 1][j - 1] + 1) {
           cout << "C" << b[i - 1]; if (j <= 9) cout << "0";
           cout << j;
           i--; j--;
           continue;
       }
       if (i and mem[i][j] == mem[i - 1][j] + 1) {
           cout << "I" << b[i - 1];
           if (j <= 9) cout << "0";
           cout << j + 1;
           i--;
           continue;
       }
       else if (j) {
           cout << "D" << a[j - 1];
           if (j <= 9) cout << "0";
           cout << j;
           j--;
       }
   }
   cout << "E\n";
}
```

4.2 Length of longest Palindrome possible by removing 0 or more characters

```
dp[startpos][endpos] = s[startpos] == s[endpos] ? 2 +
dp[startpos + 1][endpos - 1] : max (dp[startpos + 1][endpos],
dp[startpos][endpos - 1])
```

4.3 Longest Common Subsequence

```
memset (mem, 0, sizeof (mem));
for (int i = 1; i <= b.size (); i++) {
  for (int j = 1; j <= a.size (); j++) {
    if (b[i - 1] == a[j - 1]) mem[i][j] = mem[i - 1][j - 1] +
    1;
    else mem[i][j] = max (mem[i - 1][j], mem[i][j - 1])
  }
}
void printsol (int ui, int li) {
  ui--; li--;
  vector<string> ans;
  while (ui || li) {
    if (a[ui] == b[li]) {
      ans.push_back (a[ui]);
      ui--; li--;
      continue;
    }
    if (ui and mem[ui][li] == mem[ui - 1][li]) {
      ui--;
      continue;
    }
    if (li and mem[ui][li] == mem[ui][li - 1]) {
      li--;
      continue;
    }
  }
  reverse (ans.begin (), ans.end ());
  cout << ans << "\n";
}
```

4.4 Prefix Function and KMP

4.4.1 Prefix Function

The prefix function for this string is defined as an array π of length n, where $\pi[i]$ is the length of the longest proper prefix of the substring $s[0 \dots i]$ which is also a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. By definition, $\pi[0] = 0$. Example:
abcabchejfabcabca
00012300001234564
Note: $\pi[i+1] \leq \pi[i]+1$ as if $\pi[i+1] > \pi[i]+1$ then consider this suffix ending at position i + 1 & having length $\pi[i+1]$ - removing the last character we get a suffix ending in position i & having length $\pi[i+1]-1$ that is better than $\pi[i]$. Should be able to reason the following code.

```
vector<int> prefix_function(string &s) {  // O(n)
    int n = (int)s.length();
    vector<int> pi(n, 0);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

4.4.2 KMP

Given a text t and a string s, we want to find and display the positions of all occurrences of the string s in the text t.
For convenience we denote with n the length of the string s and with m the length of the text t.
We generate the string s+#+t, where # is a separator that doesn't appear in s and t. Let us calculate the prefix function for this string. Now think about the meaning of the values of the prefix function, except for the first n+1 entries (which belong to the string s and the separator). By definition the value π[i] shows the longest length of a substring ending in position i that coincides with the prefix. But in our case this is nothing more than the largest block that coincides with s and ends at position i. This length cannot be bigger than n due to the separator. But if equality π[i]=n is achieved, then it means that the string s appears completely in at this position, i.e. it ends at position i. Just do not forget that the positions are indexed in the string s+#+t.
Thus if at some position i we have π[i]=n, then at the position $i`(n+1)`n+1 = i`2n$ in the string t the string s appears.

As already mentioned in the description of the prefix function computation, if we know that the prefix values never exceed a certain value, then we do not need to store the entire string and the entire function, but only its beginning. In our case this means that we only need to store the string s+# and the values of the prefix function for it. We can read one character at a time of the string t and calculate the current value of the prefix function.

```
void kmp() {
    auto pref = prefix_function(p);
    int j = 0;
    int cnt = 0;
  // Note: pi[n] = 0, hence j = 0.
    for (int i = 0; i < t.size(); i++) {
        while (j > 0 and t[i] != p[j]) {
            j = pref[j - 1];
        }
        if (t[i] == p[j]) j++;
        if (j == p.size()) {  // j == n, that means we must
        dec. j.
    // And remember that if s[0...n - 1] == s[1...n - 1]s[n-1]
    that means s[0] = s[1], s[1] = s[2], s[n-2] = s[n-1]. That
    means all characters are same and hence we haven't lost
    anything as pref[n - 1] = n - 1.
            cnt++;  // occurence found
            j = pref[j - 1];
        }
    }
}
```

4.4.3 Counting number of occurrences of each prefix

```
vector<int> ans(n + 1);
for (int i = 0; i < n; i++)  // Longest prefix is favored and
will have correct count. But remember that longest prefix also
have smaller prefix in it. So here i is string index
    ans[pi[i]]++;
for (int i = n-1; i > 0; i--)  // here i is prefix length. Thus
we are doing backward propagation
    ans[pi[i-1]] += ans[i];
for (int i = 0; i <= n; i++)  // as only intermediate strings
were considered, we didn't consider original prefix.
    ans[i]++;
```

4.5 Notes

- In case of hashing a string, we follow polynomial rolling hash function, with p as a prime number roughly equal to the size of character domain and m as a huge prime number.

4.6 SAM

A suffix automaton for a given string s is a minimal DFA that accepts all the suffixes of the string s.

- A suffix automaton is an oriented acyclic graph.
- One of the states t_0 is the initial state
- All transitions originating from a state must have different labels
- One or multiple states are marked as terminal states. If we start from the initial state t_0 and move along transitions to a terminal state, then the labels of the passed transitions must spell one of the suffixes of the string s. Each of the suffixes of s must be spellable using a path from t_0 to a terminal state.

Consider any non-empty substring t of the string s. We will denote with endpos(t) the set of all positions in the string s, in which the occurrences of t end. For instance, we have endpos("bc")= {2,4} for the string "abcbc". We will call two substrings t1 and t2 endpos-equivalent, if their ending sets coincide i.e. endpos(t1) = endpos(t2). Thus all non-empty substrings of the string s can be decomposed into several equivalence classes according to their sets endpos.

It turns out, that in a suffix machine endpos-equivalent substrings correspond to the same state. In other words the number of states in a suffix automaton is equal to the number of equivalence classes among all substrings, plus the initial state.

Lemma 1: Two non-empty substrings u and w (with length(u) ≤ length(w)) are endpos-equivalent, if and only if the string u occurs in s only in the form of a suffix of w. (Proof is obvious)

Lemma 2: Consider two non-empty substrings u and w (with length(u) ≤ length(w)). Then their sets endpos either don't intersect at all, or endpos(w) is a subset of endpos(u). And it depends on if u is a suffix of w or not. (Proof is obvious)

Lemma 3: Consider an endpos-equivalence class. Sort all the substrings in this class by non-increasing length. Then in the resulting sequence each substring will be one shorter than the previous one, and at the same time will be a suffix of the previous one. In other words the substrings in the same equivalence class are actually each others suffixes, and take all possible lengths in a certain interval [x;y].

Consider some state v ≠ t_0 in the automaton. As we know, the state v corresponds to the class of strings with the same endpos values. And if we denote by w the longest of these strings, then all the other strings are suffixes of w. **suffix link** *link(v)* leads to the state that corresponds to the longest suffix of w that is another endpos-equivalent class.

Lemma 4: Suffix links form a tree with the root t_0.

Lemma 5: If we build a endpos tree from all the existing sets (according to the principle "the set-parent contains as subsets of all its children"), then it will coincide in structure with the tree of suffix references.

Note: For each state v one or multiple substrings match. We denote by longest(v) the longest such string, and through len(v) its length. We denote by shortest(v) the shortest such substring, and its length with minlen(v). Then all the strings corresponding to this state are different suffixes of the string longest(v) and have all possible lengths in the interval [minlength(v);len(v)]. For each state v ≠ t_0 a suffix link is defined as a link, that leads to a state that corresponds to the suffix of the string longest(v) of length minlen(v)`1. minlen(v) = len(link(v))+1

Number of states in suffix automaton of the string s of length n doesn't exceed $2n - 1$ (for n ≥ 2)

Number of transitions ≤ $3n - 4$.

4.7 Important Problems

Review: cf 631D

- UVA 10739 Sol, UVA 10739 Prob: String to palindrom, just see the minimum edit distance between this string and its reverse but need to divide by 2 later as both strings are it itself.
- Queries for the number of palindromic substrings within given range, **See this soln to see power of hashing.**:

  Note: Strings and arrays are considered 0-based in the following solution.

  Let $isPal[i][j]$ be 1 if $s[i...j]$ is palindrome, otherwise, set it 0. Let's define $dp[i][j]$ to be number of palindrome substrings of $s[i...j]$. Let's calculate $isPal[i][j]$ and $dp[i][j]$ in $O(|S|^2)$. First, initialize $isPal[i][i] = 1$ and $dp[i][i] = 1$. After that, loop over len which states length of substring and for each specific len, loop over $start$ which states starting position of substring. $isPal[start][start + len - 1]$ can be easily calculated by the following formula:

  ```
  isPal[start][start+len-1] = isPal[start+1][start+len-2] & (s[start] == s[start+len-1])
  ```

  After that, $dp[start][start + len - 1]$ can be calculated by the following formula which is derived from Inc-Exc Principle.

  ```
  dp[start][start+len-1] = dp[start][start+len-2] + dp[start+1][start+len-1] - dp[start+1]
  [start+len-2] + isPal[start][start+len-1]
  ```

  After preprocessing, we get queries l_i and r_i and output $dp[l_i - 1][r_i - 1]$. Overall complexity is $O(|S|^2)$.

- UVA 536 Sol, UVA 536 Prob: Construct binary tree with preorder and inorder
- UVA 10459 Sol, UVA 10029 Prob: Edit steps, (lexicographic sequence of words)