卐 Short Revision Notes 卐

Sourabh Aggarwal (sourabh23)

Compiled on December 12, 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 whose all moves are to N positoin is P position.

Note: Every Position is either a P or N. For games using misere play all is same except that step 1 is replaced by the condition that all terminal positions are **N** postions.

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. **Note:** $g(x)$ is 0 if x is a terminal position

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 (what move of a component game to take) such that we reach **P** position.

1.2 Mobius

Just read this and this.

Prob, Sol: $\sum_{g=1}^{i} h(g) * cnt[g]$ where $cnt[g]$ = no. of arrays with $gcd(a_1, a_2, a_3, ..., a_n) = g$ and where each $a_k \leq i$. Now $h(g)$ = Dirichlet identity function. Thus it is summation of mobius function. Ans thus we get $\sum_{d=1}^{i} \mu(d) * f(d)$ where $f(d)$ is the number of arrays with elements in range $[1, i]$ such that $gcd(a_1, \ldots, a_n)$ is divisible by j. Obviously $f(j) = (\lfloor i/j \rfloor)^n$.

1.3 Modulo

$(a + b) mod m = (a mod m + b mod m) mod m$

..... − .....

..... * .....

```
const int m1 = (int) 1e9 + 7;
template <typename T>
inline T add(T a, T b) {
    a += b;
    if (a >= m1) a -= m1;
    return a;
}
template <typename T>
inline T sub(T a, T b) {
    a -= b;
    if (a < 0) a += m1;
    return a;
}
```

```
template <typename T>
inline T mul(T a, T b) {
    return (T) (((long long) a * b) % m1);
}

template <typename T>
inline T power(T a, T b) {
    int res = 1;
    while (b > 0) {
        if (b & 1) {
            res = mul<T>(res, a);
        }
        a = mul<T>(a, a);
        b >>= 1;
    }
    return res;
}

template <typename T>
inline T inv(T a) {
    return power<T>(a, m1 - 2);
}
```

1.4 Prob and Comb

- $E[X] = \sum E(X|A_i)P(A_i)$
- k, p_a, p_b prob, Sol, if $n+m \geq k \rightarrow p_b(i+j) + p_a * p_b * (i+j+1) + p_a^2 * p_b * (i+j+2) \cdots = (i+j) + \frac{p_a}{p_b}$ Also

$$\begin{aligned} dp[0][0] &= p_a * dp[1][0] + p_b * dp[0][0] && (1)\\ &= p_a * dp[1][0]/(1-p_b) && (2)\\ &= dp[1][0] && (3) \end{aligned}$$

- **Dearrangement of n objects**
  $n! * \sum_{k=0}^{n}(-1)^k/k! = \ !n$
  $!n = (n-1) * [!(n-1)+!(n-2)]$ for n ≥ 2

1.5 Euler's Totient Function

Also known as phi-function $\phi(n)$, counts the number of integers between 1 and n inclusive, which are coprime to n.
If p is prime $\phi(p) = p-1$.
If p is a prime number and $k \geq 1$, then there are exactly p^k/p numbers between 1 and p^k that are divisible by p. Which gives us:$\phi(p^k) = p^k - p^{k-1}$.
If a and b are relatively prime, then: $\phi(ab) = \phi(a) \cdot \phi(b)$. This relation is not trivial to see. It follows from the Chinese remainder theorem.
In general, for not coprime a and b, the equation

$$\phi(ab) = \phi(a) \cdot \phi(b) \cdot \frac{d}{\phi(d)}$$

with $d = \gcd(a,b)$ holds.
$\phi(n) = \phi({p_1}^{a_1}) \cdot \phi({p_2}^{a_2}) \cdots \phi({p_k}^{a_k})$
$= ({p_1}^{a_1} - {p_1}^{a_1-1}) \cdot ({p_2}^{a_2} - {p_2}^{a_2-1}) \cdots ({p_k}^{a_k} - {p_k}^{a_k-1})$
$= p_1^{a_1} \cdot \left(1 - \frac{1}{p_1}\right) \cdot p_2^{a_2} \cdot \left(1 - \frac{1}{p_2}\right) \cdots p_k^{a_k} \cdot \left(1 - \frac{1}{p_k}\right)$
$= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right)$

Eulers Theorem:

$$a^{\phi(m)} \equiv 1 \pmod{m}$$

if a and m are relatively prime.

In the particular case when m is prime, Euler's theorem turns into Fermat's little theorem:

$$a^{m-1} \equiv 1 \pmod{m}$$

1.6 Catalan

$Cat(n) = \binom{2n}{n}/(n+1)$
$Cat(m) = (2m*(2m-1)/(m*(m+1))) * Cat(m-1)$
$Cat(n) =$

1. the number of ways a convex polygon with n+2 sides can be cut into n triangles
2. the number of ways to use n rectangles to tile a stairstep shape (1, 2, ..., n1, n).
3. No. of expressions containing n pairs of parentheses which are correctly matched.
4. the number of planar binary trees with n+1 leaves
5. No. of distinct binary trees with n vertices
6. No. of different ways in which n + 1 factors can be completely parenthesized. Like for {a, b, c, d}, one parenthing will be ((ab)c)d.
7. the number of monotonic paths of length 2n through an n-by-n grid that do not rise above the main diagonal
8. n pair of people on circle can do non cross hand shakes.

Note: Its better to use bigint for catalan computations. Also no. of binary trees with n labelled nodes $= cat[n] * fact[n]$

1.7 Floyd Cycle Finding

```
// mu = start of the cycle
// lam = its length
// O (mu + lam) time complexity
// O (1) space complexity
ii floydCycleFinding(int x0) {
    // 1st part: finding k * lam
    int tortoise = f(x0), hare = f (f (x0));
    // hare moves at twice speed
    while (tortoise != hare) {
        tortoise = f (tortoise); hare = f(f(hare));
    }
    // thus tor = x_i; hare = x_2i
    // i.e. x_2i = x_{i + k * lam}
    // i.e. k * lam = i.
    // Now if hare is set to beginning
    // i.e. hare = x_0, tor = x_i
    // thus if both now move same no. of steps and in between
    they become equal, i.e.
    // x_l = x_{i + l}
    // i.e. x_l = x_{l + k * lam}
    // Thus l must be the minimum index and therefore l = mu
    int mu = 0;
    hare = x0;
    while (tortoise != hare) {
        tortoise = f (tortoise); hare = f(hare); mu++
    }
    // finding lam
    int lam = 1; hare = f (tortoise);
    while (tortoise != hare) {
        hare = f (hare); lambda++;
    }
    return ii (mu, lambda);
}
```

1.8 Base Conversion

```
// decimal no. to some base
stack<int> S;
while (q) {
    s.push (q % b);
    q /= b;
}
while (!s.empty ()) {
    cout << process (s.top ()) << " ";
    s.pop ();
}
// base to decimal no.
ll baseToDec () {
    ll ret = 0;
    for (auto &c : num) {
        ret = (ret * base + (c - 48)); // can take mod if final
        answer is required in mod
    }
    return ret;
}
```

1.9 Extended Euclid

$ax + by = c$ this is called diophantine eqn and is solvable only when $d = gcd(a,b)$ divides c. so first solve $ax + by = d$ then multiply x, y with c/d. Also once we have found a particular soln to this eqn then their exist infinite solns of the form $(x0 + (b/d) * n, y0 - (a/d) * n)$ where n is any integer. Assume we hound the coefs (x1, y1) for (b, a mod b) $\rightarrow$
$b * x1 + (a \bmod b) y1 = g$
$\rightarrow b * x1 + (a - \lfloor (a/b) \rfloor * b) * y1 = g$
$\rightarrow a * y1 + b * (x1 - \lfloor (a/b) \rfloor * y1) = g$
$\rightarrow x = y1 \ \& \ y = x1 - \lfloor (a/b) \rfloor * y1$

```
void extendedEuclid(int a, int b) {
   if (b == 0) { x = 1; y = 0; d = a; return; } // base case
   extendedEuclid(b, a % b); // similar as the original gcd
   int x1 = y;
   int y1 = x - (a / b) * y;
   x = x1;
   y = y1;
}
```

1.10 Sieve

```
    ll _sieve_size; // ll is defined as: typedef long long ll;
bitset<10000010> bs; // 10^7 should be enough for most cases
vi primes; // compact list of primes in form of vector<int>
void sieve(ll upperbound) { // create list of primes in
[0..upperbound]
```

```
   _sieve_size = upperbound + 1; // add 1 to include upperbound
   bs.set(); // set all bits to 1
   bs[0] = bs[1] = 0; // except index 0 and 1
   for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
// cross out multiples of i starting from i * i!
          for (ll j = i * i; j <= _sieve_size; j += i) bs[j] =
          0;
          primes.push_back((int)i); // add this prime to the
          list of primes
      } } // call this method in main method
bool isPrime(ll N) { // a good enough deterministic prime
tester
    // O(#primes < sqrt(N))
    // O(sqrt(N)/ln(sqrt(N)))
   if (N <= _sieve_size) return bs[N]; // O(1) for small primes
   for (int i = 0; i < (int)primes.size(); i++)
       if (N % primes[i] == 0) return false;
   return true; // it takes longer time if N is a large prime!
} // note: only work for N <= (last prime in vi "primes")^2

vi primeFactors(ll N) { // remember: vi is vector<int>, ll is
long long
   vi factors;
   ll PF_idx = 0, PF = primes[PF_idx]; // primes has been
   populated by sieve
   while (PF * PF <= N) { // stop at sqrt(N); N can get smaller
       while (N % PF == 0) { N /= PF; factors.push_back(PF); }
       // remove PF
       PF = primes[++PF_idx]; // only consider primes!
   }
   if (N != 1) factors.push_back(N); // special case if N is a
   prime
   return factors; // if N does not fit in 32-bit integer and
   is a prime
} // then 'factors' will have to be changed to vector<ll>


memset(numDiffPF, 0, sizeof numDiffPF);
//Modified Sieve.
void pre() {
   for (int i = 2; i < MAX_N; i++)
       if (numDiffPF[i] == 0) // i is a prime number
           for (int j = i; j < MAX_N; j += i)
               numDiffPF[j]++; // increase the values of
               multiples of i
}
// Bottom up euler totient function
for (int i = 0; i <= limit; i++) eu[i] = i;
for (int i = 2; i <= limit; i++) {
    if (eu[i] == i) {
        for (int j = i; j <= limit; j += i) {
            eu[j] -= eu[j] / i;
        }
    }
}
```

1.11 Side Notes

1. People in cycle will commit suicide.
2. Every positive integer can be expressed uniquely as a sum of fibonacci numbers such that no two numbers are equal or consecutive fibonacci numbers.
3. Every even no. greater than or equal to 4 can be expressed as a sum of 2 prime nos.
4. No. of digits in a no. $n = \lfloor(\log_{10} n)\rfloor + 1$
5. No. of digits in $\binom{n}{k} = \lfloor(\sum_{i=n-k+1}^{n} \log_{10} i - \sum_{i=1}^{k} \log_{10} i))\rfloor + 1$
6. No. of digits of a no. in some base b= $floor(1 + \log_b no. + eps)$. Also make sure that input no. is not 0.
7. for $\binom{n}{r}$ always do $r = min(r, n-r)$. Also to compute it either we can use dp or for a specific pair, if it is guarenteed that the final solution lies within data types limit then we can compute it as.

```
ll ncr(ll n, ll r) {
    r = min (r, n - r);
    ll res = 1;
    for (int k = 1; k <= r; k++, n--) {
        res *= n;
        res /= k;
    }
}
```

8. $(t^a - 1)/(t^b - 1)$ is not an integer with less than 100 digits if t = 1 or $a < b$ or $a \bmod b \neq 0$ or $(a - b) * \log_{10} t > 99.0$
9.
```
for (int j = 0; j < bigint_var.a.size (); j++) {
    int temp = bigint_var.a[j];
    while (temp > 9) {
        sum += temp % 10;
        temp /= 10;
    }
    sum += temp;
}
```
10. $$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p = \begin{bmatrix} fib(p+1) & fib(p) \\ fib(p) & fib(p-1) \end{bmatrix}$$

    Thus higher fibs can be computed in $O(\log p)$

1.12 Important Problems

- UVA 11310 Prob: $dp[n] = dp[n-1] + 4 * dp[n-2] + 2 * dp[n-3]$
- UVA 11204 Prob: Tricky problem, it just asks *How many possible arrangements maximizing the assignment of the* ***first priority***. Thus only first priority instrument matters, so if 2 want A, 4 want B, 3 want C, then ans is $2 * 4 * 3$.
- UVA 10790 Prob: If there is an intersection that means we have a quadrilateral, hence answer is the number of quadrilaterals = $\binom{a}{2} * \binom{b}{2}$.
- last non zero digit of fact(n), Sol. If you know multiplicatioin limit, take mod with $(10^{\text{no. of digits}})$
- France 98, Sol.
- How many zeros and how many digits? Sol: then iterate through factors of base and get their powers in n!, take min. of all such powers divided by power of prime factor in that base. And for how many digits part, use that log formula.
- Given n, maximize (find x) $n - p * x$ where $p * x \leq n < (p+1) * x$ which somehow happens with $p = 1$
- **Prob:** Lenghts from 1 to n, max. no. of triangles?
  Sol:

```
void precal () {
    F[3] = P[3] = 0;
    ll var = 0;
    for (int i = 4; i <= 1000000; i++) {
        if (i % 2 == 0) {
            var++;
        }
        P[i] = P[i - 1] + var;
        F[i] = F[i - 1] + P[i];
    }
    // F[n] has ans
}
```

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 LCA

- Jammie and Tree, Sol: One stop soln to understand LCA.

How to find the LCA of u and v using the precomputed LCA table that assumes the root is vertex 1? Let's separate the situation into several cases. If both u and v are in the subtree of r, then query the LCA directly is fine. If exactly one of u and v is in the subtree of r, the LCA must be r. If none of u and v is in the subtree of r, we can first find the lowest nodes p and q such that p is an ancestor of both u and r, and q is an ancestor of both v and r. If p and q are different, we choose the deeper one. If they are the same, then we query the LCA directly. Combining the above cases, one may find the LCA is the lowest vertex among $lca(u, v), lca(u, r), lca(v, r)$.

After we have found the origin w of update (for query, it is given), how to identify the **subtree of a vertex** and carry out updates/queries on it? Again, separate the situation into several cases. If $w = r$, update/query the whole tree. If w is in the subtree of r, or w isn't an ancestor of r, update/query the subtree of w. Otherwise, update/query the whole tree, then undo update/exclude the results of the subtree of w', such that w' is a child of w and the subtree of w' contains r.

2.1.2 Important Problems

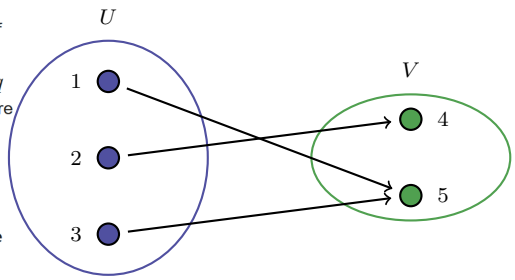
- UVA 11695 Sol: Problem Desc: Find which edge to remove and add so as to minimise the number of hops to travel between flights. Problem Sol: Just link the center of diameters. Brute force which edge to remove.
- UVA 112 Sol, UVA 112 Prob: Just see how I processed the input.
- UVA 10029 Sol, UVA 10029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

2.2 Terminology

- A **vertex cover** is a subset of vertices S, such that for each edge (u, v) in graph, either u or v (or both) are in S.
- An **independent set** is a subset of vertices S, such that no two vertices u, v in S are adjacent in graph.
- A subset of vertices is a vertex cover iff the complement of the set is an independent set. I.e. $MinVC + MaxIS = V$.
- A matching is a subset of edges such that each vertex is adjacent to at most one edge in the subset. Clearly Matching edges can be atmost $|V|/2$ as each edge joins two vertices and now no other matched edge can touch them.
- Once we have maximum matching. Clearly since these matching edges are aswell edges of the graph and minimum vertex cover should have vertices that are adjacent to these edges. But since matching edges have no vertex in common, size of minimum vertex cover is atleast the size of maximum matching.
- Maximum matchings can be found in polynomial time for any graph, while minimum vertex cover is NP complete. Thus, finding maximum independent sets is another NP-complete problem.
- The equivalence between matching and covering articulated in Kőnig's theorem allows minimum vertex covers and maximum independent sets to be computed in polynomial time for bipartite graphs, despite the NP-completeness of these problems for more general graph families.

2.3 Konigs Theorem

Size of Min VC in a bipartite graph is equal to the size of Max Matching in that graph.
Kőnig's theorem can be proven in a way that provides additional useful information beyond just its truth: the proof provides a way of constructing a minimum vertex cover from a maximum matching.
Proof: Let $G = (V, E)$ be a bipartite graph, and let the vertex set V V be partitioned into left set L and right set R. Suppose that M is a maximum matching for G.
let U be the set of unmatched vertices in L (possibly empty), and let Z be the set of vertices that are either in U or are connected to U by alternating paths. Let $K = (L \setminus Z) \cup (R \cap Z)$.
Every edge e in E either belongs to an alternating path (and has a right endpoint in K, or it has a left endpoint in K. For, if e is matched but not in an alternating path, then its left endpoint cannot be in an alternating path (for such a path would have had to have included e) and thus belongs to $L \setminus Z$. Alternatively, if e is unmatched but not in an alternating path, then its left endpoint cannot be in an alternating path, for such a path could be extended by adding e to it. Thus, K forms a vertex cover.
Additionally, every vertex in K is an endpoint of a matched edge. For, every vertex in $L \setminus Z$ is matched because Z is a superset of U. And every vertex in $R \cap Z$ must also be matched, for if there existed an alternating path to an unmatched vertex then changing the matching by removing the matched edges from this path and adding the unmatched edges in their place would increase the size of the matching. However, no matched edge can have both of its endpoints in K. Thus, K is a vertex cover of cardinality equal to M, and must be a minimum vertex cover.
Small diagram to understand proof well.



Matched edges are (2, 4) and (3, 5).
$U = \{1\}$, $Z = \{1, 5, 3\}$, $L = \{1, 2, 3\}$, $K = \{2, 5\}$

2.4 Bipartite Matching

2.4.1 Hopcroft Karp

1. **Free node or vertex:** Given a matching M, a node that is not a part of mathing is called a free node. Initially all vertices are free.
2. **Matching and not matching edges:** Given a matching M, edges that are part of matching are called matching edges and edges that are not part of M (or connect free nodes) are called non matching edges.
3. **Alternating Paths:** Given a matching M, an alternating path is a path in which edges belong alternatively to the matching and not matching.
4. **Augmenting path:** Given a matching M, an augmenting path is an alternating path that starts from and ends on free vertices.
5. The Hopcroft karp algorithm is based on below concept:
6. A matching M is not maximum if there exist an augmenting path. It is also true other way, i.e., a matching is maximum if no augmenting path exists.
7. Hopcroft Karp Algo $O(\sqrt{V} * E)$:
   (a) Initialize maximal matching M as empty.
   (b) While there exists an augmenting path P, remove matching edges of P from M and add not matching edges of P to M. (This increases size of M by 1 as P starts and ends with a free vertex).
   (c) Return M.

The following is the sol to problem UVA 11419 where we were just required to find minimum vertex cover.

```
#include <bits/stdc++.h>
#define FOR(i, a, b) for (int i = a; i <= b; i++)
#define REP(i, n) for (int i = 0; i < n; i++)
#define pb push_back
#define INF 500000000
#define maxN 1010
using namespace std;

int n, m, matchX[maxN], matchY[maxN];
int dist[maxN];
vector<int> adj[maxN];
bool Free[maxN];

bool bfs() {
    queue<int> Q;
    FOR (i, 1, n)
        if (!matchX[i]) {  // only free vertices are pushed
        in queue and have their distance set to 0. Thus
        already matched vertices in X will have their
        distance set to INF.
            dist[i] = 0;
            Q.push(i);
        }
        else dist[i] = INF;
    dist[0] = INF;  // 0 is nil
    // Thus we would always start from free vertices
    traverse then alternating path and if in end from Y
    there is no match i.e. its a free vertice, we found an
    augmenting path.
    // Side Notes: If we popped an already matched vertex
    from queue then it wont go to its matching edges
    neighbor as its matchY is popped vertex itself and
    hence it wont have distance set to INF.
    while (!Q.empty()) {
        int i = Q.front(); Q.pop();
        REP(k, adj[i].size()) {
            int j = adj[i][k];
            if (dist[matchY[j]] == INF) {
                dist[matchY[j]] = dist[i] + 1;
                Q.push(matchY[j]);
            }
        }
```

```
    }
    return dist[0] != INF;
}

bool dfs(int i) {
    if (!i) return true; // to handle nil.
    REP(k, adj[i].size()) {
        int j = adj[i][k];
        if (dist[matchY[j]] == dist[i] + 1 &&
        dfs(matchY[j])) {
            matchX[i] = j;
            matchY[j] = i;
            return true;
        }
    }
    dist[i] = INF;
    return false;
}

int hopcroft_karp() {
    int matching = 0;
    while (bfs())
        FOR (i, 1, n)
            if (!matchX[i] && dfs(i))
                matching++;
    return matching;
}

void dfs_konig(int i) {
    Free[i] = false;
    REP(k, adj[i].size()) {
        int j = adj[i][k];
        if (matchY[j] && matchY[j] != INF) {
            int x = matchY[j];
            matchY[j] = INF;  // as we have undirected
            edge, we dont want to traverse that same edge
            again, so its just a way of noting that.
            if (Free[x]) dfs_konig(x);
        }
    }
}

void solve() {
    printf("%d", hopcroft_karp());
    FOR (i, 1, n)
        if (!matchX[i])
            dfs_konig(i);  // finding Z.
    FOR (i, 1, n)
        if (matchX[i] && Free[i])  // i.e. in L but not in
        Z.
            printf(" r%d", i);
    FOR (j, 1, m)
        if (matchY[j] == INF)  // i.e. we traversed this
        edge i.e. its in R intersection Z.
            printf(" c%d", j);
    putchar('\n');
}

void initialize() {
    FOR (i, 1, n) {
        adj[i].clear();
        matchX[i] = 0;
        Free[i] = true;
    }
    memset(matchY, 0, (m + 1) * sizeof(int));
}

int ar[5];
char buff[20];
void read_line() {
    gets(buff);
    int len = strlen(buff), i = 0, m = 0;
    while (i < len)
        if (buff[i] != ' ') {
            ar[m] = 0;
            while (i < len && buff[i] != ' ')
                ar[m] = ar[m] * 10 + buff[i++] - 48;
            m++;
        }
        else i++;
}

main() {
    int k, u, v;
    while (scanf(" %d %d %d ", &n, &m, &k) != EOF) {
        if (!n && !m && !k) break;
        initialize();
        while (k--) {
            read_line();
            adj[ar[0]].pb(ar[1]);
        }
        solve();
    }
}
```

3 Some Basic

- `#pragma GCC optimize("Ofast")  // tells the compiler to optimize the code for speed to make it as fast as possible (and not look for space)`
  `#pragma GCC optimize ("unroll-loops")  // normally if we have a loop there is a "++i" instruction somewhere. We normally dont care because code inside the loop requires much more time but in this case there is only one instruction inside the loop so we want the compiler to optimize this.`
  `#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx,avx,tune=native")`
  `// tell the compiler that our cpu has simd instructions and allow him to vectorize our code`
- `while (first || cin >> temp) {  // something }`
- **Interval Covering:** Tell the minimum no. of intervals to cover the entire big interval.

```
void solve() {
    // Greedy Algorithm
    sort (data.begin (), data.end ());
    for (; i < data.size(); i = j) {
       if (data[i].first > rightmost) break;
       for (j = i + 1; j < data.size() and data[j].first <=
       rightmost; j++) {
           if (data[j].second > data[i].second) {
               i = j;
           }
       }
       ans.push_back(data[i]);
       rightmost = data[i].second;
       if (rightmost >= m) break;

    }
    if (rightmost < m) {
       cout << "0\n";
    }
}
```

- ```
  // time complexity O(log(min(a, b) / gcd(a, b)))
  int gcd (int a, int b) { return b == 0 ? a : gcd (b, a
  % b); }
  int lcm (int a, int b) { return a * (b / gcd (a, b)); }
  ```

- **Prob:** We have a stack of turtles and we have some final permutation of them, each turtule can crawl out of its position and move to top. Determine a minimal sequence of operations to obtain the final permutation.
  Sol:

```
for (int j = n - 1, next = n - 1; j >= 0; j--) {
    if (order[j].second != next)
    Toswap.push_back(order[j]);
    else next--;
}
sort (Toswap.begin(), Toswap.end());
```

- ```
  #define MAX_N 2 // Fibonacci matrix,
  increase/decrease this value as needed
  struct Matrix { int mat[MAX_N][MAX_N]; }; // we will return
  a 2D array
  Matrix matMul(Matrix a, Matrix b) { // O(n^3)
     Matrix ans; int i, j, k;
     for (i = 0; i < MAX_N; i++)
         for (j = 0; j < MAX_N; j++)
             for (ans.mat[i][j] = k = 0; k < MAX_N; k++) //
             if necessary, use
                 ans.mat[i][j] += a.mat[i][k] * b.mat[k][j];
                 // modulo arithmetic
     return ans;
  }
  ```

```
Matrix matPow(Matrix base, int p) { // O(n^3 log p)
   Matrix ans; int i, j;
   for (i = 0; i < MAX_N; i++) for (j = 0; j < MAX_N; j++)
           ans.mat[i][j] = (i == j); // prepare identity
           matrix
   while (p) { // iterative version of Divide & Conquer
   exponentiation
       if (p & 1) ans = matMul(ans, base); // if p is odd
       (last bit is on)
       base = matMul(base, base); // square the base
       p >>= 1; // divide p by 2
   }
   return ans;
}
```

```
// Months are 0 indexed
//The following Code solves problems: UVA 893

int numberDaysInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31,
30, 31, 30, 31};
int numberDaysInMonthLeap[] = {31, 29, 31, 30, 31, 30, 31,
31, 30, 31, 30, 31};

bool IsLeapYear(int year)
{
   return year % 4 == 0 && (year % 100 != 0 || year % 400
   == 0);
}

int MonthToDay(int month, int year)
{
   int daysBefore = 0;
   for (int i = 0; i < month; ++i)
       daysBefore += numberDaysInMonth[i];
   if (month > 1 && IsLeapYear(year))
       ++daysBefore;
   return daysBefore;
}

int YearToDay(int year)
{
   int base = year * 365;
   int numLeapYears = year / 4 - year / 100 + year / 400;
   return base + numLeapYears;
}

int GetYearFromNumDays(int& numDays)
{
   int year = 1;
   int sizeOfYear = 365;

   while (numDays > sizeOfYear)
   {
       numDays -= sizeOfYear;
       ++year;
       sizeOfYear = (IsLeapYear(year)) ? 366 : 365;
   }

   return year;
}

int GetMonthFromNumDays(int& numDays, int year)
{
   int month = 0;
   int * numDayUsed = (IsLeapYear(year)) ?
   numberDaysInMonthLeap : numberDaysInMonth;
   for (;numDays > numDayUsed[month]; ++month)
       numDays -= numDayUsed[month];
   return month + 1;
}

int main()
{
   int dayForward, day, month, year;
   while (cin >> dayForward >> day >> month >> year, year)
   {
       --month;
       day += MonthToDay(month, year);
       --year;
       day += YearToDay(year);
       day += dayForward;
       year = GetYearFromNumDays(day);
       month = GetMonthFromNumDays(day, year);
       cout << day << ' ' << month << ' ' << year << '\n';
   }
}
//--
string int2roman(int n) {
   string roman;
   string ones[] = {"", "I", "II", "III", "IV", "V", "VI",
   "VII", "VIII", "IX"};
   string tens[] = {"", "X", "XX", "XXX", "XL", "L", "LX",
   "LXX", "LXXX", "XC"};
   string hundreds[] = {"", "C", "CC", "CCC", "CD", "D",
   "DC", "DCC", "DCCC", "CM"};
   string thousands[] = {"", "M", "MM", "MMM"};

   int o = n % 10;
   n /= 10;
   int t = n % 10;
   n /= 10;
   int h = n % 10;
   n /= 10;
   int th = n % 10;

   roman += thousands[th] + hundreds[h] + tens[t] +
   ones[o];//Or
   //roman=thousands[th] + hundreds[h] + tens[t] + ones[o]
   but the written one is
   //faster.

   return roman;
}
```

- **Algorithm to convert from infix to postifx:**
  1. Scan the infix expression from left to right.
  2. If the scanned character is an operand, output it.
  3. Else,
     (a) If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty or the stack contains a '('), push it.
     (b) Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
  4. If the scanned character is an '(', push it to the stack.
  5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.
  6. Repeat steps 2-6 until infix expression is scanned.
  7. Print the output
  8. Pop and output from the stack until it is not empty.
- **Algorithm to convert from infix to prefix:**
  1. Properly reverse the infix exp.
  2. Gets its postfix as above
  3. Reverse postfix and output it.
- **Algorithm to convert from postfix to infix:**
  1. If the symbol is an operand, push it onto stack
  2. Else, if there are fewer than two values in stack, show error. Else, pop top 2 expressions from stack (say e1, e2), put the operator (op) between them and push to stack ((e1 op e2))
  3. After reading postfix expression, Stack should have only one item which is our answer
- **Merge Sort**

```
void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 =  r - m;

   /* create temp arrays */
   int L[n1], R[n2];

   /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
       L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
       R[j] = arr[m + 1+ j];

   /* Merge the temp arrays back into arr[l..r]*/
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = l; // Initial index of merged subarray
```

```
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else///i.e we need to swap
        {
            arr[k] = R[j];
            swaps+=n1-i;//Most important line. basically
            once we are doing arr[k]=R[j] that means we are
            ///putting R[j] before each of n1-i elements
            thus there are that many swaps.
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
  sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
```

- set is like min heap. Only unique elements are present.
- On a line you are given the x coordinates of various houses, tell the house of vito (h) such that $\sum |h_i - h|$ is minimised. **Obs1:** h could be any of h_i so $O(n^2)$ algo. will work. **Obs2:** Taking derivative we get $i - j = 0$ i.e. $i = j = n/2$, that means simply sort and output the middlemost house.
  Note: Some times the math become cumbersome, in such cases, use **ternary search**
- **Prob:** n people have to cross the bridge, one torch, atmost 2 can travel
  Sol: if $n = 3 \Rightarrow$ time $= x + y + z$, if $n \geq 4$ let A, B, a, b be the fastest, second fastest, slowest, second slowest resp. **Goal:** Get the slowest members to the other side. So choose the best among the two options.
  option 1: Fastest member does back and forth.
  option 2: The two fastest members go, allowing the two slowest two to go together.
- **Inversions:** From a permutation, parity of number of swaps needed to get to the identical permutation is same as parity of inversion count of this permutation.
  Parity of inversions can be calculated in $O(n)$ by finding the number of cycles.
  Exact value of number of inverions can be calculated in $(nlog(n))$ by using segment trees.
- **Prob:** You are given two positive integer numbers a and b. Permute (change order) of the digits of a to construct maximal number not exceeding b.
  Sol: Take the number as string. sort string a, then for each $i \in [1, n]$ swap it with j trying from $ntoi + 1$ such that it is $\leq b$ (normal string comparison can be used).
- **Prob:** From a digraph, remove atmost one edge so that it becomes DAG.
  Sol: Get any one cycle the iteratively try to remove each edge and see if it makes it DAG or not.
- **UFDS**

```
struct UFDS {
    vector<int> p, rank, setSizes;
    int numSets;
    UFDS(int N) {
        numSets = N;
        rank.assign(N, 0);
        p.assign(N, 0);
        for (int i = 0; i < N; i++)
            p[i] = i;
        setSizes.assign(N, 1);
    }
    int findSet(int i) {
        return (p[i] == i) ? i : p[i] = findSet(p[i]);
    }
    bool isSameSet(int i, int j) {
        return findSet(i) == findSet(j);
    }
    void unionSet(int i, int j) {
        if (!isSameSet(i, j)) {
            int x = findSet(i), y = findSet(j);
            if (rank[x] > rank[y]) {
                setSizes[findSet(x)] +=
                setSizes[findSet(y)];
                p[y] = x;
            } else {
                setSizes[findSet(y)] +=
                setSizes[findSet(x)];
                p[x] = y;
                if (rank[x] == rank[y])
                    rank[y]++;
            }
            numSets--;
        }
    }
    int setSize(int i) {
        return setSizes[findSet(i)];
    }
    int numDisjointSets() {
        return numSets;
    }
};
```

- UVA 10158 Prob, UVA 10158 Sol
- Imbalance of a tree, Sol, summation(max - min) is same as summation(max) - summation(min).
- Party Lemonade, Sol
- Logical Expression, Sol: pr denotes from what grammer it is derived. Also number of functions on n variables $= 2^{2^n}$.
- Jamie and binary sequence, Sol.

4 Data Structures

4.1 Segment Tree

- Jamie and to do list, Sol: Just basic application of Persistent segment tree. When updating some element, at most O(logn) nodes in the segment tree get changed: the nodes along the path from root to the updated leaf. For each timepoint, instead of creating a copy of the entire segment tree, copy only nodes on the path to be updated and update them. Therefore total storage is O(n + tlogn).

5 DP

5.1 Coin Change

```
/*No. of ways in which we can make change of that money
O(N*V)*/
// Recurrence: dp[value] = dp[value - type1] + ... + dp[value -
typen]
int N = 5, V, coinValue[5] = {1, 5, 10, 25, 50};
long long int memo[6][30000];
long long int ways(int type, int value) {
   if (value == 0)              return 1;
   if (value < 0 || type == N)  return 0;
   if (memo[type][value] != -1) return memo[type][value];
   return memo[type][value] = ways(type + 1, value) +
   ways(type, value - coinValue[type]);
}
```

```
/*Bottom up version of the above solution*/
long long int solve() {
   dp[0] = 1; //rest all are 0;
   for(i = 0; i < coinTypes; ++i){
       for(j = coins[i]; j <= value; ++j)
           dp[j] += dp[j - coins[i]];
   }
}
/*Of problem above, in case you want dp[i][j] where it means,
no. of ways to represent val j using coin
* types [0...i] */
void solve() {
   dp[0][0] = 1; //rest all are 0;
   for(int i = 0; i < coinType; i++){
       if(i) {
           for(int j = 0; j <= maxVal; j++) {
               dp[i][j] = dp[i - 1][j];
           }
       }
       for(int j = coinValue[i]; j <= maxVal; ++j)
           dp[i][j] += dp[i][j - coinValue[i]];
   }
}
// Minimum no. of coins/bills given to fullfill an amount >= x
when each coin/bill can be used any no. of times
// Recurrence: dp[value] = min_i{dp[value - type_i] + 1}
void solve() {
   vector<long long int> dp;
   dp.assign(30000, INT_MAX);
   dp[0] = 0;
   for(int i = 0; i < 5; i++) {
       for(int j = coinValue[i]; j <= V; j++) {
           if(dp[j - coinValue[i]] != INT_MAX) {
               dp[j] = min(dp[j], dp[j - coinValue[i]] + 1);
           }

       }
   }
   res = dp[V];
}
/*Minimum no. of coins/bills given to fullfill an amount >= x
when each coin/bill can be used only once*/
void solve() {
   int dp [10000 + 10];
   for ( int i = 0; i < 10010; i++ )
       dp [i] = INT_MAX;
   dp [0] = 0;
   for (int i = 0; i < coinNumber; i++) {
       for (int j = 10000 - coins[i]; j >= 0; j--) {
           if (dp[j] != INT_MAX && dp[j + coins[i]] > dp[j] +
           1)
               dp[j + coins[i]] = dp[j] + 1;
       }
   }
   for ( int i = x; i <= 10000; i++ ) {
       if ( dp [i] != INT_MAX ) {
           printf ("%d %d\n", i, dp [i]);
           break;
       }
   }
}
/*Minimum no. of coins/bills given to fullfill an amount >= x
when each coin/bill can be used
* a fixed no. of times*/
void solve() {
   vector<ll> buyer(505, LLONG_MAX);
   buyer[0] = 0;
   for (int i = 0; i < 6; i++) {
       for(int k = 0; k < cnt[i]; k++) {
           for (int j = 500 - coinValue[i]; j >= 0; j--) {
               if (buyer[j] != LLONG_MAX && buyer[j +
               coinValue[i]] > buyer[j] + 1)
                   buyer[j + coinValue[i]] = buyer[j] + 1;
           }
       }
   }
}
```

5.2 Balanced Bracket Sequence

A Balanced bracket sequence is a string consisting of only brackets, such that this sequence, when certain numbers and + is inserted gives a valid mathematical expression.

5.2.1 One type of bracket

Let depth be the current no. of open brackets, initially depth = 0. We iterate over all character of the string; if the current bracket character is an opening bracket then we increment depth, o/w we decrement it. I f at any time the variable depth gets negative, or at the end it is different from 0, then the string is not a balanced sequence otherwise it is.

5.2.2 MultiType

Maintain a stack, in chich we will store all opening brackets that we meet. If the current bracket character is an opening one, we put it onto the stack. If it is a closing one, then we check if the stack is non empty, and if the top element is of the same type as the current closing bracket, if both conditions are fulfilled, then we remove the opening bracker from the stack. If at any time one of the ocnditions is not fulfilled or at the end the stack is non empty, then the string is not balanced otherwise it is.

5.2.3 No. of balanced Sequences

The number of balanced bracket sequences with only one bracket type can be calculated using the Catalan numbers. The number of balanced bracket sequences of length $2n$ (n pairs of brackets) is:

$$\frac{1}{n+1}\binom{2n}{n}$$

If we allow k types of brackets, then each pair be of any of the k types (independently of the others), thus the number of balanced bracket sequences is:

$$\frac{1}{n+1}\binom{2n}{n}k^n$$

On the other hand these numbers can be computed using dynamic programming. Let $d[n]$ be the number of regular bracket sequences with n pairs of bracket. Note that in the first position there is always an opening bracket. And somewhere later is the corresponding closing bracket of the pair. It is clear that inside this pair there is a balanced bracket sequence, and similarly after this pair there is a balanced bracket sequence. So to compute $d[n]$, we will look at how many balanced sequences of i pairs of brackets are inside this first bracket pair, and how many balanced sequences with $n-1-i$ pairs are after this pair. Consequently the formula has the form:

$$d[n] = \sum_{i=0}^{n-1} d[i] \cdot d[n-1-i]$$

The initial value for this recurrence is $d[0] = 1$.

5.2.4 Lexicographically next balanced sequence

```
// Idea: "dep" indicates the imbalance in the string s[0...i
- 1]. Now after replacing s[i] with ')', dep dec. and we want
to add the lexicographically least string having 'dep - 1'
closing brackets reserved.
bool next_balanced_sequence(string & s) {
  int n = s.size();
  int depth = 0;
  for (int i = n - 1; i >= 0; i--) {
      if (s[i] == '(')
          depth--;
      else
          depth++;

      if (s[i] == '(' && depth > 0) {
          depth--;
          int open = (n - i - 1 - depth) / 2;
          int close = n - i - 1 - open;
          string next = s.substr(0, i) + ')' + string(open,
          '(') + string(close, ')');
          s.swap(next);
          return true;
      }
  }
  return false;
}
```

If it is required to find and output all balanced bracket sequences of a specific length n.

To generate them, we can start with the lexicographically smallest sequence $((\dots(())\dots))$, and then continue to find the next lexicographically sequences with the algorithm described above.

5.2.5 Sequence Index

Given a balanced bracket sequence with n pairs of brackets. We have to find its index in the lexicographically ordered list of all balanced sequences with n bracket pairs.

Let's define an auxiliary array $d[i][j]$, where i is the length of the bracket sequence (semi-balanced, each closing bracket has a corresponding opening bracket, but not every opening bracket has necessarily a corresponding closing one), and j is the current balance (difference between opening and

closing brackets). $d[i][j]$ is the number of such sequences that fit the parameters. We will calculate these numbers with only one bracket type.

For the start value $i = 0$ the answer is obvious: $d[0][0] = 1$, and $d[0][j] = 0$ for $j > 0$. Now let $i > 0$, and we look at the last character in the sequence. If the last character was an opening bracket (, then the state before was $(i-1, j-1)$, if it was a closing bracket), then the previous state was $(i-1, j+1)$. Thus we obtain the recursion formula:

$$d[i][j] = d[i-1][j-1] + d[i-1][j+1]$$

$d[i][j] = 0$ holds obviously for negative j. Thus we can compute this array in $O(n^2)$.

Now let us generate the index for a given sequence.

First let there be only one type of brackets. We will us the counter depth which tells us how nested we currently are, and iterate over the characters of the sequence. If the current character $s[i]$ is equal to (, then we increment depth. If the current character $s[i]$ is equal to), then we must add $d[2n-i-1][\text{depth}+1]$ to the answer, taking all possible endings starting with a (into account (which are lexicographically smaller sequences), and then decrement depth.

New let there be k different bracket types.

Thus, when we look at the current character $s[i]$ before recomputing depth, we have to go through all bracket types that are smaller than the current character, and try to put this bracket into the current position (obtaining a new balance ndepth = depth $\pm$ 1), and add the number of ways to finish the sequence (length $2n-i-1$, balance $ndepth$) to the answer:

$$d[2n-i-1][\text{ndepth}] \cdot k^{\frac{2n-i-1-ndepth}{2}}$$

This formula can be derived as follows: First we "forget" that there are multiple bracket types, and just take the answer $d[2n-i-1][\text{ndepth}]$. Now we consider how the answer will change is we have k types of brackets. We have $2n-i-1$ undefined positions, of which ndepth are already predetermined because of the opening brackets. But all the other brackets $((2n-i-i-\text{ndepth})/2$ pairs) can be of any type, therefore we multiply the number by such a power of k.

5.2.6 Finding the kth sequence

Let n be the number of bracket pairs in the sequence. We have to find the k-th balanced sequence in lexicographically sorted list of all balanced sequences for a given k.

As in the previous section we compute the auxiliary array $d[i][j]$, the number of semi-balanced bracket sequences of length i with balance j.

First, we start with only one bracket type.

We will iterate over the characters in the string we want to generate. As in the previous problem we store a counter depth, the current nesting depth. In each position we have to decide if we use an opening of a closing bracket. To put an opening bracket character, it $d[2n-i-1][\text{depth}+1] \geq k$. We increment the counter depth, and move on to the next character. Otherwise we decrement k by $d[2n-i-1][\text{depth}+1]$, put a closing bracket and move on.

```
string kth_balanced(int n, int k) {
  vector<vector<int>> d(2*n+1, vector<int>(n+1, 0));
  d[0][0] = 1;
  for (int i = 1; i <= 2*n; i++) {
      d[i][0] = d[i-1][1];
      for (int j = 1; j < n; j++)
          d[i][j] = d[i-1][j-1] + d[i-1][j+1];
      d[i][n] = d[i-1][n-1];
  }

  string ans;
  int depth = 0;
  for (int i = 0; i < 2*n; i++) {
      if (depth + 1 <= n && d[2*n-i-1][depth+1] >= k) {
          ans += '(';
          depth++;
      } else {
          ans += ')';
          if (depth + 1 <= n)
              k -= d[2*n-i-1][depth+1];
          depth--;
      }
  }
  return ans;
}
```

Now let there be k types of brackets. The solution will only differ slightly in that we have to multiply the value $d[2n-i-1][\text{ndepth}]$ by $k^{(2n-i-1-\text{ndepth})/2}$ and take into account that there can be different bracket types for the next character.

Here is an implementation using two types of brackets: round and square:

```
string kth_balanced2(int n, int k) {
  vector<vector<int>> d(2*n+1, vector<int>(n+1, 0));
  d[0][0] = 1;
  for (int i = 1; i <= 2*n; i++) {
      d[i][0] = d[i-1][1];
      for (int j = 1; j < n; j++)
          d[i][j] = d[i-1][j-1] + d[i-1][j+1];
      d[i][n] = d[i-1][n-1];
  }

  string ans;
  int depth = 0;
  stack<char> st;
  for (int i = 0; i < 2*n; i++) {
      // '('
      if (depth + 1 <= n) {
          int cnt = d[2*n-i-1][depth+1] << ((2*n-i-1-depth-1)
          / 2);
          if (cnt >= k) {
              ans += '(';
              st.push('(');
              depth++;
              continue;
          }
          k -= cnt;
      }

      // ')'
      if (depth && st.top() == '(') {
          int cnt = d[2*n-i-1][depth-1] << ((2*n-i-1-depth+1)
          / 2);
          if (cnt >= k) {
              ans += ')';
              st.pop();
              depth--;
              continue;
          }
          k -= cnt;
      }

      // '['
      if (depth + 1 <= n) {
          int cnt = d[2*n-i-1][depth+1] << ((2*n-i-1-depth-1)
          / 2);
          if (cnt >= k) {
              ans += '[';
              st.push('[');
              depth++;
              continue;
          }
          k -= cnt;
      }

      // ']'
      ans += ']';
      st.pop();
      depth--;
  }
  return ans;
}
```

6 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.

6.1 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";
}
```

6.2 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])
```

6.3 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";
}
```

6.4 Prefix Function and KMP

6.4.1 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;
}
```

6.4.2 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];
        }
    }
}
```

6.4.3 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]++;
```

6.5 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.
- If s is palindrome and if $s[0...n-2]$ is palindrome, that means all characters are same thus if all characters are not same then the longest non palindromic substring is $s[0...n-2]$ or $s[1...n-1]$

6.6 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:** $endpos(t_0) = \{-1, 0, \ldots, length(s)-1\}$

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 ≤ $3n-4$.

```
for (int j = 0; j < bigint_var.a.size (); j++) {
    int temp = bigint_var.a[j];
    while (temp > 9) {
        sum += temp % 10;
        temp /= 10;
    }
    sum += temp;
}
```

6.7 Important Problems

Review: cf 631D

- UVA 10739 Sol, UVA 10739 Prob: String to palindrome, 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 11107 Sol - simple, UVA 11107 Sol - complicated but more powerful: Problem is to find the longest substring shared by more than half of given strings.
- UVA 10459 Sol, UVA 10029 Prob: Edit steps, (lexicographic sequence of words)