CPC Algorithm Reference

Liberty University Competitive Programming

2024

C++ Library

```cpp
#include <bits/stdc++.h>

using namespace std;

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define For(i, a) for(int i = 0; i < (a); ++i)
#define all(x) begin(x), end(x)
#define is_in(x, s) ((s).find(x) != (s).end())
#define endl '\n'
#define pi acos(-1.0)
typedef long long ll;
template<class T> using V=vector<T>;
template<class T> using VV=vector<vector<T>>;
template<class K, class V> using umap=unordered_map<K, V>;
typedef pair<int, int> Point;
typedef vector<int> vi;
typedef unordered_map<int, int> uimap;
int main() {
  cin.tie(0)->sync_with_stdio(0);
  // if you want to use pow with large numbers
  // you can use powl
  // nth root of p is
  // (int) round(p, (1.0 / n))
  // (int) round(p, (1.0 / n))
}
```

Python Data Structures

```python
from collections import deque
from queue import Queue, PriorityQueue, LifoQueue
from heapq import heapify, heappush, heappop

list = []
dict = {}
set = set()
frozen_set = frozenset() # immutable set
tuple = tuple() # immutable list

fifoQueue = Queue()
lifoQueue = LifoQueue()
prioQueue = PriorityQueue()
doubleEnded = deque()

# min heaps (for max heap multiply each by -1)
heap = [1,2,3,4]
heapify(heap)
top = heappop(heap)
heappush(heap, 0)

ii = lambda:int(input())
mii = lambda:list(map(int, input().split()))
```

```cpp
#define set_diff(s1, s2, store) set_difference((s1).begin(), (s1).end(), (s2).begin(), (s2).end(), inserter(store, (store).begin()));
```

Dynamic Programming

Knapsack Problem

```python
def knapsack(items, weights, values, capacity):
    """
    Items - list of names (not necessary)
    Weights - the weights (counts against capacity) of items 0-n
    Values - the values (what is maximized) of items 0-n
    Capacity - the highest weight that can fit in the bag
    Returns max value in the bag and the names of the items used
    """
    n = len(items)
    table = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                table[i][w] = max(values[i - 1] + table[i - 1][w - weights[i - 1]],
table[i - 1][w])
            else:
                table[i][w] = table[i - 1][w]
    selected_items = [] # used for item names
    i, w = n, capacity
    while i > 0 and w > 0:
        if table[i][w] != table[i - 1][w]:
            selected_items.append(items[i - 1]) # used for item names
            w -= weights[i - 1]
        i -= 1
    return table[n][capacity], selected_items
```

Coin Problem

```python
def coinProblem(coins, target):
    # number of ways to hit target
    dp = [[0 for i in range(target+1)] for i in range(len(coins) + 1)]
    # used for backtracking, will be -1 for end or [coinInd, (nextX, nextY)]
    used = [[-1 for i in range(target+1)] for i in range(len(coins) + 1)]
    for i in range(1, len(dp)):
        dp[i][0] = 1 # only 1 way to get 0 money
    # basically like knapsack here
    for i in range(1, len(coins) + 1):
        for j in range(1, target+1):
            if coins[i - 1] > j:
                dp[i][j] = dp[i-1][j]
                used[i][j] = used[i-1][j]
            else:
                dp[i][j] = dp[i-1][j] + dp[i][j - coins[i - 1]]
                # used[i][j] = used[i - 1][j]
                if dp[i][j - coins[i - 1]] > 0:
                    used[i][j] = [i, (i, j - coins[i - 1])]
                elif dp[i-1][j] > 0:
                    used[i][j] = [-1, (i-1, j)]
    """ Backtracking to find coins used """
    # x = []
    # next = used[-1][i]
    # while next:
    #     if next == -1:
    #         break
    #     a, coord = next
    #     if a != -1:
    #         x.append(a)
    #     next = used[coord[0]][coord[1]]
    # print(*list(sorted(x)))
```

Binary Search (Find)

```python
def search(self, nums: List[int], target: int) -> int:
       i = 0
       j = len(nums) - 1
       k = j // 2


       while i <= j:
           if target == nums[k]:
               return k # found, return index
           elif target < nums[k]:
               j = k-1 # move upper pointer down
           else:
               i = k+1 # move lower pointer up
           k = (i+j)//2
       return -1 # not found, return -1
```

Binary Search (Find largest pile)

```python
def search(item: int, piles: list[int]) -> int:
    """
    Given a list of piles, find the index of the pile where the top is < item
    Note this keeps the piles in sorted order
    O(log n)
    """
    pos = len(piles)
    x = 0
    y = len(piles) - 1
    while x <= y:
        k = (x + y) // 2
        if piles[k][-1] >= item:
            x = k + 1
        else:
            y = k - 1
            pos = k
    # you would then put the item in the pile:
    # piles[pos].append(item)
    return pos
```

Edit Distance (With Operations)

```python
def editDist(first, second):
    """
    Find the edit distance between first and second
    Use this when you need to change the operations available
    O(n * m)
    """
    n = len(first)
    m = len(second)
    # Create a table to store results of subproblems
    dp = [[0 for x in range(n + 1)] for x in range(m + 1)]
    # Fill dp
    for i in range(m + 1):
        for j in range(n + 1):
            # first string is empty, insert all letters of second
            if i == 0:
                dp[i][j] = j  # Min. operations = j
            # second is empty, insert all letters of first
            elif j == 0:
                dp[i][j] = i  # Min. operations = i
            # last characters are the same
            elif second[i - 1] == first[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            # last characters are different
            else:
                dp[i][j] = 1 + min(dp[i][j - 1],  # Insert
                                   dp[i - 1][j],  # Remove
                                   dp[i - 1][j - 1])  # Replace
    return dp[m][n]
```

Edit Distance (Space Optimized)

```python
def editDist(first, second):
    """
    Find the edit distance between first and second
    Use this unless you need to change the operations available
    O(n * m)
    """
    n = len(first)
    m = len(second)
    prev = [j for j in range(m + 1)]
    curr = [0] * (m + 1)

    for i in range(1, n + 1):
        curr[0] = i
        for j in range(1, m + 1):
            if first[i - 1] == second[j - 1]:
                curr[j] = prev[j - 1]
            else:
                mn = min(1 + prev[j], 1 + curr[j - 1])
                curr[j] = min(mn, 1 + prev[j - 1])
        prev = curr.copy()

    return prev[m]
```

Longest Increasing Subsequence

```python
def lis(nums):
    import bisect
    n = len(nums)
    ans = [nums[0]]
    for i in range(1, n):
        if nums[i] > ans[-1]:
            ans.append(nums[i])
        else:
            x = bisect.bisect_left(ans, nums[i])
            ans[x] = nums[i]
    return len(ans)
```

Longest Common Subsequence

```cpp
int dp[1001][1001];
int lcs(const string &s, const string &t) {
  int m = s.size(), n = t.size();
  if (m == 0 || n == 0) return 0;
  For(i, m+1) dp[i][0] = 0;
  For(j, n+1) dp[0][j] = 0;
  For(i, m) {
      For(j, n) {
          if (s[i] == t[j]) dp[i + 1][j + 1] = dp[i][j] + 1;
          else dp[i + 1][j + 1] = max(dp[i + 1][j], dp[i][j + 1]);
      }
  }
  return dp[m][n];
}
```

Depth/Width of Tree

```python
def depth(root) -> int: # Make sure that root cannot be not null!
    if left[root] == 0 and right[root] == 0:
        return 1
    return 1 + max(depth(left[root]), depth(right[root]))

def width(root) -> int:
    from collections import deque
    q, width = deque([(root, 0)]), 0
    while q:
        # last minus first
        width = max(width, q[-1][1] - q[0][1])
        for _ in range(len(q)):
            node, k = q.popleft()
            if left[node]:
                q.append((left[node], k * 2 - 1))
            if right[node]:
                q.append((right[node], k * 2))
    return width + 1
```

Breadth First Search

```python
def bfs(graph, source, target):
    """
    :param graph: An adjacency list stored in a dict: graph[start] = [list of destination nodes]
    source=starting node,target=destination,return min nodes to visit to target
    """
    from collections import deque
    visited = {}
    queue = deque([source])
    visited[source] = 0
    while len(queue) > 0:  # Creating loop to visit each node
        m = queue.popleft()
        for neighbor in graph[m]:
            if neighbor not in visited:
                visited[neighbor] = 1 + visited[m]
                queue.append(neighbor)
                if neighbor == target:
                    return visited[m] + 1
    return -1
```

Depth First Search

```python
def dfs(graph, source, target):
    """
    Find a node in a graph
    :param graph: An adjacency list stored in a dict: graph[start] = [list of destination nodes]
    :param source: starting node
    :param target: destination node
    :return: whether it is in the graph
    """
    visited = {}
    stack = []
    stack.append(source)
    while stack:
        s = stack.pop()
        if s in visited:
            continue
        visited[s] = True
        for neighbor in graph[s]:
            stack.append(neighbor)
            if neighbor == target: # just return super early if found
                return True
```

Strongly Connected Component

```
class SCC:
    # Takes in an adjacency map
    # Map from a node to nodes it is connected to
    def __init__(self, graph):
        self.graph = graph

    def bfs(self, source):
        """Get all nodes in a graph in breadth order
        O(v+e)
        :param source: Starting node
        :return: List of visited nodes"""
        visited = {}
        stack = deque()
        visited[source] = True
        stack.append(source)
        while len(stack) > 0:  # Creating loop to visit each node
            m = stack.popleft()
            for neighbor in self.graph[m]:
                if neighbor not in visited:
                    visited[neighbor] = True
                    stack.append(neighbor)
        return visited

    def transpose(self):
        """ Transpose the saved graph
        O(ve)
        :return: The new graph"""
        g = {}
        for i in self.graph:
            for j in self.graph[i]:
                if j not in g:
                    g[j] = [i]
                else:
                    g[j].append(i)
            if i not in g:
                g[i] = []
        return g
```

```
   def isSSC(self):
        """
        Return whether a whole graph is Strongly Connected
        O(ve)
        :return: True or False
        """
        s = list(self.graph.keys())[0]
        v1 = self.bfs(s)
        old = self.graph
        self.graph = self.transpose()
        v2 = self.bfs(s)
        self.graph = old
        if set(v1) == set(v2):
            return True
        else:
            return False

    def getSSC_Size(self, start):
        """
        Return the size of the strongly connected component that starts at start
        O(ve)
        :param start: The starting node
        :return: The size of the SCC that the start node is in
        """
        s = start
        v1 = set(self.bfs(s))
        old = self.graph
        self.graph = self.transpose()
        v2 = set(self.bfs(s))
        self.graph = old
        return len(v1.intersection(v2))
```

Dijkstra’s Shortest Path

```
def dijkstra(graph, start, end):
"""O(n + mlogm)
    :param graph: Dict in form graph[node] = [(next, cost) ... ]Adjacency list of tuples
    You can remove the end param if you just want distances from start.
    Make sure that you also change the return value to just dist though
    :param start: The node to start from
    :return: distance to end"""
    from heapq import heappush, heappop
    dist = {node: float("inf") for node in graph.keys()}
    dist[start] = 0
    pq = [(0, start)]
    processed = set()
    while pq:
        cost, node_a = heappop(pq)
        #You can remove this if if you want all distances from start
        if node_a == end:
            return cost
        if node_a in processed: continue
        processed.add(node_a)

        for node_b, w in graph[node_a]:
            if dist[node_b] > dist[node_a] + w:
                dist[node_b] = dist[node_a] + w
                heappush(pq, (dist[node_b], node_b))
    return dist[end] if end in dist else float('inf')
```

Floyd Warshall (Shortest Path Any)

```
def floydWarshall(graph):
    """
    find the shortest path from any node to any node
    O(v^3)
    :param graph: matrix representation of a graph where
                  graph[a][b] = cost of path from a to b
    :return: the distances matrix
    """
    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))
    V = len(graph)
    for k in range(V):
        # pick all vertices as source one by one
        for i in range(V):
            # Pick all vertices as destination for the
            # above picked source
            for j in range(V):
                # If vertex k is on the shortest path from
                # i to j, then update the value of dist[i][j]
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
```

Bellman Ford (Find negative cycles in weighted graph)

```
def bellman_ford(graph, start):
    """ Will raise an error if there is a negative cycle
    :param graph: Dict in form graph[node] = [(next, cost), ...]
    :param start: node to start at
    :return distances"""
    dis = {vertex:float('inf') for vertex in graph}
    dis[start] = 0
    # Step 2: Relax edges |V| - 1 times
    for _ in range(len(graph) - 1):
        for vtx in graph:
            for v, w in graph[vtx]:
                if dis[vtx] != float('inf') and dis[vtx] + w < dis[v]:
                    dis[v] = dis[vtx] + w
    # Step 3: Check for negative weight cycles
    for vtx in graph:
        for v, w in graph[vtx]:
            if dis[vtx] != float('inf') and dis[vtx] + w < dis[v]:
                raise ValueError('Graph contains negative weight cycle')
    return dis
```

Ford Fulkersons Max Flow

```python
class FF:
   def __init__(self, nodes) -> None:
       self.rgraph = [[0] * nodes for i in range(nodes)]
       self.nodes = nodes
   def addEdge(self, source, target, capacity):
       self.rgraph[source][target] = capacity
   def bfs(self, source, target):
       from collections import deque
       visited = [False] * self.nodes
       q = deque()
       q.append([source, float("inf")])
       visited[source] = True
       self.parent[source] = -1
       # Standard BFS Loop
       while q:  # note, can just use parent instead of visited
           u, flow = q.popleft()
           for v in range(self.nodes):
               if visited[v] == False and self.rgraph[u][v] > 0:
                   q.append([v, min(flow, self.rgraph[u][v])])
                   self.parent[v] = u
                   visited[v] = True
                   if v == target:
                       return min(flow, self.rgraph[u][v])
       return 0
   def find_max_flow(self, source, target):
       """Get max flow from node s to t on O(v^2e)"""
       self.parent = [-1] * self.nodes
       max_flow = 0
       while True:
           path_flow = self.bfs(source, target)
           if path_flow == 0:
               break
           v = target
           while v != source:
               u = self.parent[v]
               self.rgraph[u][v] -= path_flow
               self.rgraph[v][u] += path_flow
               v = self.parent[v]
           max_flow += path_flow
       return max_flow
   def getEdgeFlow(self, source, target):
       return self.rgraph[target][source]
```

Topological Sorting

```cpp
int f[100] = {0}, ans[100] = {0};
bool g[100][100] = {0}, v[100] = {0};
int n = 0, m = 0;

void dfs(int k) {
   int i = 0;
   v[k] = true;
   for (int i = 1; i<= n; i++)
       if (g[k][i] && !v[i]) dfs(i);
   m++;
   ans[m] = k;
}

int main(void) {
   cin >> n >> m;
   for (int i = 1; i <= m; i++) {
       int x = 0, y = 0;
       cin >> x >> y;
       g[x][y] = true;
   }
   m = 0;
   for (int i = 1; i <= n; i++)
       if (!v[i]) dfs(i);
   for (int i =1; i<= n; i++) cout << ans[i] << " ";
}
```

Trie

```python
class Trie:
   def __init__(self, duplicates=False):
       # :param duplicates: True if include duplicates
       self.children = {}
       self.dups = duplicates

   def add(self, word):
       # O(wordlen): Add a Word to the Trie
       current_dict = self.children
       for letter in word:
           current_dict = current_dict.setdefault(ord(letter), {})
       if self.dups:
           if ord('_') in current_dict:
               current_dict[ord('_')] += 1
           else:
               current_dict[ord('_')] = 1
       else:
           current_dict[ord('_')] = 1

   def starts_with(self, prefix, rev=False):
       '''Returns a list of all words beginning with the given prefix
       O(n logn) where n is wordlen
       :param prefix: The prefix string to find matches
       :param rev: True if this is being used as a suffix tree '''
       words = list()
       current = self.children
       t = ""
       for char in prefix:
           if ord(char) not in current:
               return list()
           current = current[ord(char)]
           if rev:
               t = char + t
       if rev:
           self.__child_words_for(current, words, t, rev=True)
       else:
           self.__child_words_for(current, words, prefix)
       return words
```

```python
def __child_words_for(self, curr, words, stem, rev=False):
       """Helper function
       O(n logn) where n is wordlen
       :param curr: The current position in the Trie
       :param words: Found words
       :param stem: The prefix + letters in the current word
       :param rev: Whether it is running in suffix mode"""
       if ord('_') in curr:
           if curr[ord('_')] > 1:
               words.extend([stem]*curr[ord('_')])
           else:
               words.append(stem)
       for char in curr:
           if chr(char) != '_':
               if rev:
                   self.__child_words_for(curr[char],words,chr(char)+stem,
rev=True)
               else:
                   self.__child_words_for(curr[char], words, stem+chr(char))
```

Z-String (Linear time pattern match)

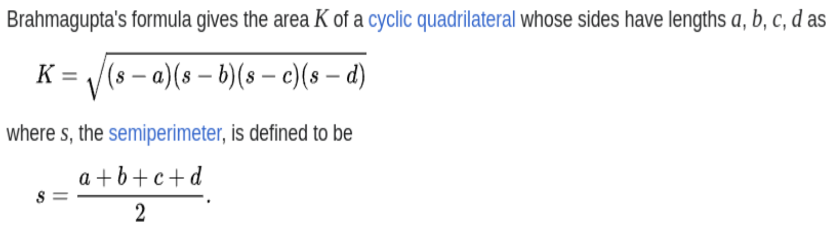
```python
def calcZ(string):
    """Find the z array of a string in O(n)"""
    z = [-1 for i in range(len(string))]
    n = len(string)
    l, r, k = 0, 0, 0
    for i in range(1, n):
        if i > r:
            l, r = i, i
            while r < n and string[r - l] == string[r]:
                r += 1
            z[i] = r - l
            r -= 1
        else:
            k = i - l
            if z[k] < r - i + 1:
                z[i] = z[k]
            else:
                l = i
                while r < n and string[r - l] == string[r]:
                    r += 1
                z[i] = r - l
                r -= 1
    return z
def search(text, pattern):
    """
    Find counts pattern in a string on O(n+m) time
    """
    # Create concatenated string "P$T"
    concat = pattern + "$" + text
    l = len(concat)
    z = calcZ(concat)
    count = 0
    for i in range(l):
        if z[i] == len(pattern):
            # found
            count += 1
    return count
```

Longest Palindromic Subsequence

```cpp
int lps(const string &s){
   int n = s.size();
   // Create two vectors: one for the current state (dp)
  // and one for the previous state (dpPrev)
   vi dp(n), dpPrev(n);
   // Loop through the string in reverse (starting from the end)
   for (int i = n - 1; i >= 0; --i){
       dp[i] = 1; // A single character is always a palindrome of length 1
       // Loop through the characters ahead of i
       rep(j, i + 1, n) {
           if (s[i] == s[j]){
               // Add 2 to the length of the palindrome between them
               dp[j] = dpPrev[j - 1] + 2;
           } else{
               // Take the maximum between excluding either i or j
               dp[j] = max(dpPrev[j], dp[j - 1]);
           }
       }
       // Update dpPrev to the current state of dp for the next iteration
       dpPrev = dp;
   }
   // Answer is the length of longest palindromic subsequence in entire string
   return dp[n - 1];
}
```

Math

Max Area of Quad (Side Len)



Brahmagupta's formula gives the area K of a cyclic quadrilateral whose sides have lengths a, b, c, d as

$$K = \sqrt{(s-a)(s-b)(s-c)(s-d)}$$

where s, the semiperimeter, is defined to be

$$s = \frac{a+b+c+d}{2}.$$

Linear Diophantine (ax+by = c)

```
/* Solves integer equations of the form ax + by = c for integers x and y.
 * Returns a bool if false, there are no solutions
 * otherwise there are infinite solutions of the form
 * x = x_0 + (b/d)n
 * y = y_0 - (a/d)n
 * Where d is gcd(a, b) and n is any integer
*/
bool ldioph(ll a, ll b, ll c, ll& x_0, ll& y_0) {
        ll local_x, local_y;
        ll d = eeuclid(a, b, local_x, local_y);
        // if c % d != 0 then there are no solutions
        if (c % d) return false;
        x_0 = local_x * (c/d);
        y_0 = local_y * (c/d);
        return true;
}
```

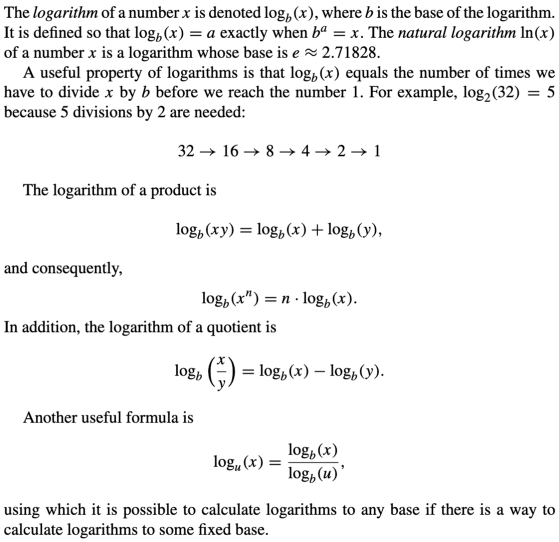
Euler's Totient/Phi

```
ll euler_phi(int x, vector<int>& primes){
    ll ret = x;
    for(int i =0; i<primes.size() && primes[i]*primes[i] <=x; ++i){
        if(x % primes[i] == 0){
            ret-= ret/primes[i];
            while(x % primes[i] == 0){
                x /= primes[i];
            }
        }
    }
    if(x>1)ret-=ret/x;
    return ret;
}

// OR
const int LIM = 5000000;
int phi[LIM];

void calculate_phi() {
    rep(i,0,LIM) phi[i] = i&1 ? i : i/2;
    for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
        for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

Logarithms:



The *logarithm* of a number x is denoted $\log_b(x)$, where b is the base of the logarithm. It is defined so that $\log_b(x) = a$ exactly when $b^a = x$. The *natural logarithm* $\ln(x)$ of a number x is a logarithm whose base is $e \approx 2.71828$.

A useful property of logarithms is that $\log_b(x)$ equals the number of times we have to divide x by b before we reach the number 1. For example, $\log_2(32) = 5$ because 5 divisions by 2 are needed:

$$32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

The logarithm of a product is

$$\log_b(xy) = \log_b(x) + \log_b(y),$$

and consequently,

$$\log_b(x^n) = n \cdot \log_b(x).$$

In addition, the logarithm of a quotient is

$$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y).$$

Another useful formula is

$$\log_u(x) = \frac{\log_b(x)}{\log_b(u)},$$

using which it is possible to calculate logarithms to any base if there is a way to calculate logarithms to some fixed base.

Number of Divisors

```
int number_of_divisors(int n, vi& primes) {
   ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
   while (PF * PF <= n) {
       ll power = 0;
       while (n % PF == 0) {
           n /= PF;
           power++;
       }
       ans *= power + 1;
       PF = primes[++PF_idx];
   }
   // last factor has pow = 1 -  we add 1 to it to get *= 2
   if (n != 1) ans *= 2;
   return ans;
}
```

Sieve of Eratosthenes

```
const int MAX_PR = 5'000'000;
bitset<MAX_PR> isprime;
vi eratosthenes_sieve(int lim) {
    isprime.set(); isprime[0] = isprime[1] = 0;
    for (int i = 4; i < lim; i += 2) isprime[i] = 0;
    for (int i = 3; i*i < lim; i += 2) if (isprime[i])
        for (int j = i*i; j < lim; j += i*2) isprime[j] = 0;
    vi pr;
    rep(i,2,lim) if (isprime[i]) pr.push_back(i);
    return pr;
}
vi primes = eratosthenes_sieve(1e6);
```

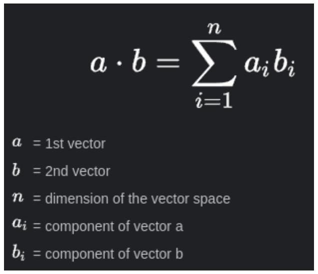
Extended Euclidean

```
// Returns GCD of a and b
// finds two integers x and y such that ax + bx = gcd(a, b)
ll eeuclid(ll a, ll b, ll &x, ll &y) {
        if (!b) return x = 1, y = 0, a;
        ll d = eeuclid(b, a % b, y, x);
        return y -= a/b * x, d;
}
```

C++ Mod Fix

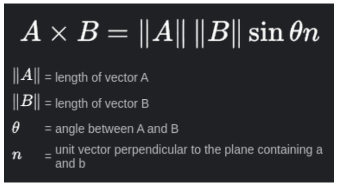
```
// Used to fix when modding negative numbers
ll mod(ll x, ll m) {
    return ((x % m) + m) % m;
}
```

Products (Dot & Cross)



$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

a = 1st vector
b = 2nd vector
n = dimension of the vector space
a_i = component of vector a
b_i = component of vector b



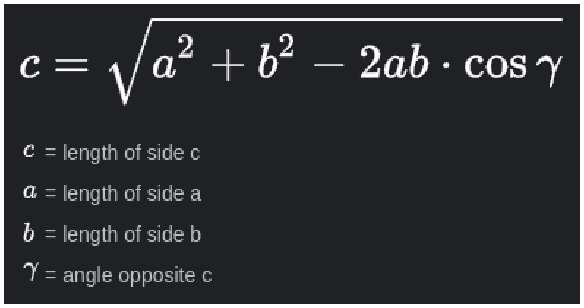
$$A \times B = \|A\| \, \|B\| \sin \theta n$$

$\|A\|$ = length of vector A
$\|B\|$ = length of vector B
θ = angle between A and B
n = unit vector perpendicular to the plane containing a and b

Sum of Divisors

```
ll sum_divisors(ll n) {
   ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
   while (PF * PF <= n) {
       ll power = 0;
       while (n % PF == 0) {
           n /= PF;
           power++;
       }
       ans *= ((ll) pow((double) PF, power + 1.0) - 1) / (PF - 1);
       PF = primes[++PF_idx];
   }
   if (n != 1)
      ans *= ((ll) pow((double) n, 2.0) - 1) / (n - 1);
   return ans;
}
```

Law of Cosines



$$c = \sqrt{a^2 + b^2 - 2ab \cdot \cos\gamma}$$

c = length of side c
a = length of side a
b = length of side b
γ = angle opposite c

GCD / LCM

```
int gcd(int a, int b) {
  int R;
  while ((a % b) > 0)  {
    R = a % b;
    a = b;
    b = R;
  }
  return b;
}
int lcm(int a, int b) { return a * (b / gcd(a, b)); }
```

Binary Exponentiation a^b in $O(log\, b)$

```
int bin_pow(int a, int n) {
   int res = 1;
   while (n) {
       if (n & 1) {
           res *= a;
           --n;
       } else {
           a *= a;
           n >>= 1;
       }
   }
   return res;
}
```

Leap Year / Binomial Coefficent

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

```
bool is_leap_year(int n) {
  if (n % 100 == 0)
      return n % 400 == 0;
  return n % 4 == 0;
}
#define MAXN 100  // largest n or m
long binonmial_coefficent(int n, int m) { // computer n choose m
  int i, j;
  long bc[MAXN][MAXN];
  For(i, n+1) bc[i][0] = 1;
  For(j, n+1) bc[j][j] = 1;
  rep(i, 1, n+1) {
      rep(j, 1, i) {
          bc[i][j] = bc[i - 1][j - 1] + bc[i - 1][j];
      }
  }
  return bc[n][m];
}
```

Fast Modular Exponentiation $a^b\, mod\, p$

```
ll pow_mod(ll base, ll exp, ll mod) {
   base %= mod;
   ll result = 1;
   while (exp > 0) {
       if (exp & 1) result = (result * base) % mod;
       base = (base * base) % mod;
       exp >>= 1;
   }
   return result;
}
```

Factorial mod $n!\, mod\, p$

```
int fact_mod(int n, int p) {
   ll res = 1;
   while (n > 1) {
       res = (res * pow_mod(p-1, n/p, p)) % p;
       for (int i = 2; i <= n%p; ++i)
           res = (res * i) % p;
       n /= p;
   }
   return int (res % p);
}
```

Find Prime and non-Prime Factors

```
const int MAX = 2'500'000;

ll factors[MAX] = {0};
ll prime_factors[MAX] = {0};
```

Prime and non prime continued….

```
void fill() {
    factors[1] = 1;
    for (int i = 2; i < MAX; i++) {
        factors[i] += 1;
        bool set_primes = prime_factors[i] == 0;
        for (int j = i; j < MAX; j += i) {
            factors[j]++;
            if (set_primes)
                prime_factors[j]++;
        }
    }
}
```

Sum Formulas

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c - 1}, c \neq 1$$

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(2n+1)(n+1)}{6}$$

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$

$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

Chinese Remainder Theorem

```
int inverse(int a, int b) {
  int x, y;
  int gcd = eeuclid(a, b, x, y);
  assert(gcd == 1); // inverse has to exist
  return (x % b + b) % b;
}

int findMinX(int num[], int rem[], int k) {
  int prod = 1;
  for (int i = 0; i < k; i++) {
    prod *= num[i];
  }
  int res = 0;
  for (int i = 0; i < k; i++) {
    int pp = prod / num[i];
    res += (rem[i] * inverse(pp, num[i]) * pp) % prod;
  }
  return res % prod;
}
```

Primality Check

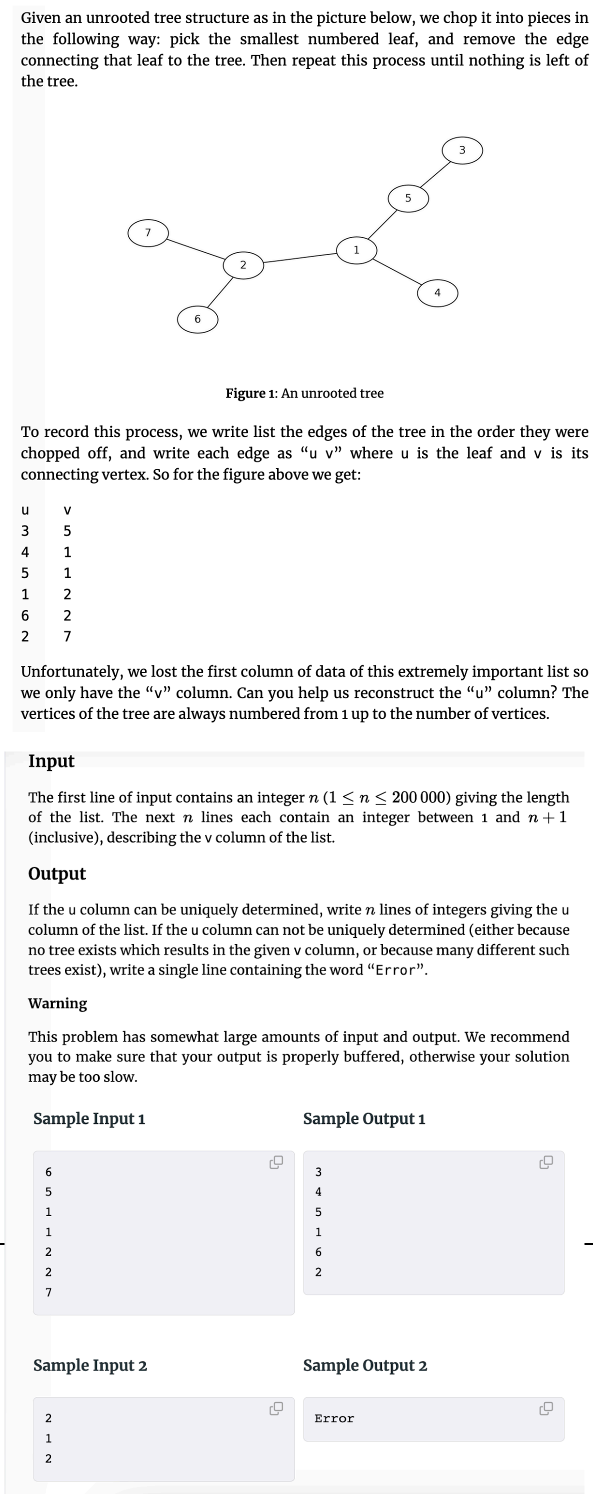
```
bool is_prime(int n) {
   if (n < 2) return false;
   if (n <= 3) return true;
   if (!(n%2) || !(n%3)) return false;
   for (int i = 5; i*i <= n; i += 6) {
       if (!(n%i) || !(n%(i+2))) return false;
   }
   return true;
}
```

Polygon Perimeter

```
import math
def polygon_perimeter(points):
    perimeter = 0.0
    num_points = len(points)
    for i in range(num_points):
        # Get the current point and the next point (wrapping around to the first point)
        x1, y1 = points[i]
        x2, y2 = points[(i + 1) % num_points]

        # Calculate the distance between the points
        distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
        perimeter += distance
    return perimeter
```

Prufer Code Tree Creation



Given an unrooted tree structure as in the picture below, we chop it into pieces in the following way: pick the smallest numbered leaf, and remove the edge connecting that leaf to the tree. Then repeat this process until nothing is left of the tree.

Figure 1: An unrooted tree

To record this process, we write list the edges of the tree in the order they were chopped off, and write each edge as “u v” where u is the leaf and v is its connecting vertex. So for the figure above we get:

u	v
3	5
4	1
5	1
1	2
6	2
2	7

Unfortunately, we lost the first column of data of this extremely important list so we only have the “v” column. Can you help us reconstruct the “u” column? The vertices of the tree are always numbered from 1 up to the number of vertices.

Input

The first line of input contains an integer n ($1 \leq n \leq 200\,000$) giving the length of the list. The next n lines each contain an integer between 1 and $n+1$ (inclusive), describing the v column of the list.

Output

If the u column can be uniquely determined, write n lines of integers giving the u column of the list. If the u column can not be uniquely determined (either because no tree exists which results in the given v column, or because many different such trees exist), write a single line containing the word “`Error`”.

Warning

This problem has somewhat large amounts of input and output. We recommend you to make sure that your output is properly buffered, otherwise your solution may be too slow.

Sample Input 1

```
6
5
1
1
2
2
7
```

Sample Output 1

```
3
4
5
1
6
2
```

Sample Input 2

```
2
1
2
```

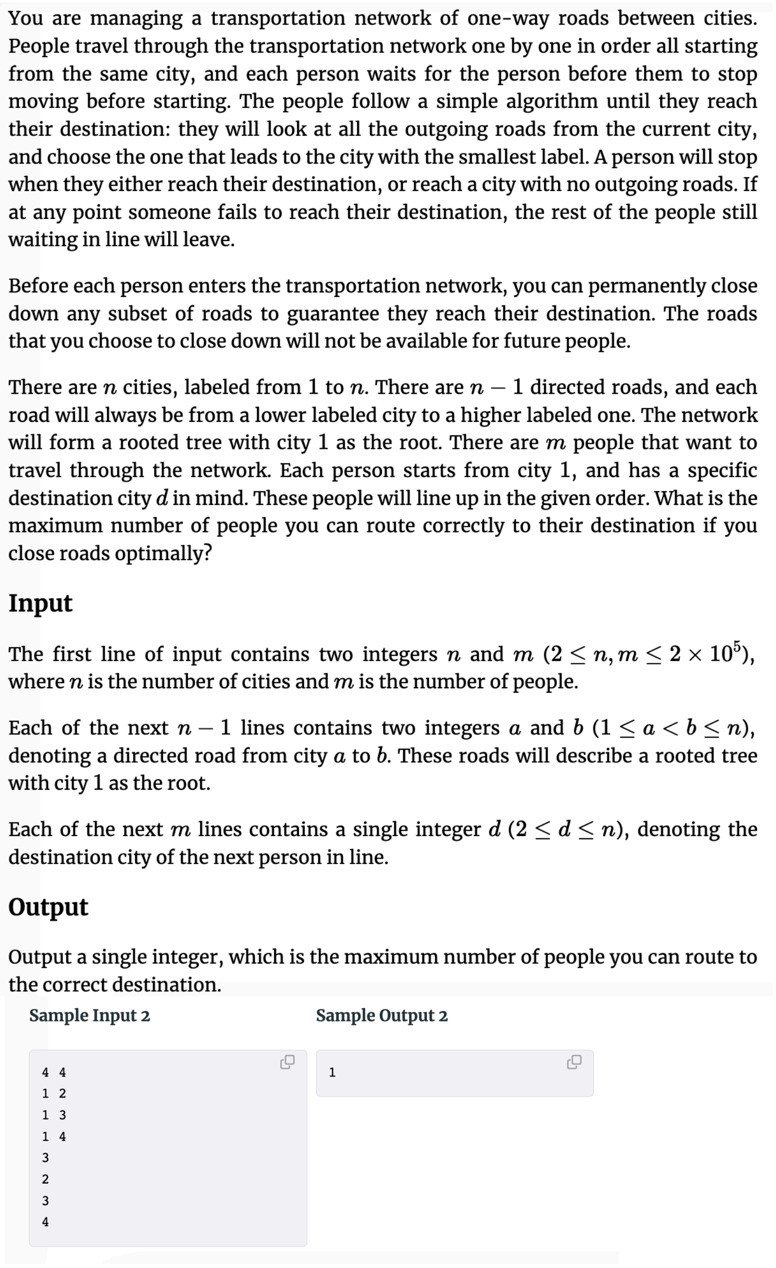
Sample Output 2

```
Error
```

```
int main() {
   cin.tie(0)->sync_with_stdio(0);
   priority_queue<int, vector<int>, greater<int>> values_to_use;  // min heap
   int n;
   cin >> n;
   uimap degree;
   vi input(n);

   For(i, n) {
       int a;
       cin >> a;
       input[i] = a;
       degree[a]++;
   }
   if (input[n - 1] != n + 1) {
       cout << "Error\n";
       return 0;
   }
   rep(i, 1, n + 2) {
       if (!is_in(i, degree)) values_to_use.push(i);
   }
   vi result(n);
   For(i, n) {
       result[i] = values_to_use.top();
       // decrease degree
       degree[input[i]]--;
       values_to_use.pop();
       // if the degree of the value is no longer in our input then we can
use
       // it
       if (degree[input[i]] == 0) {
           values_to_use.push(input[i]);
       }
   }
   for (auto i : result) {
       cout << i << '\n';
   }
}
```

Branch Manager - DFS



You are managing a transportation network of one-way roads between cities. People travel through the transportation network one by one in order all starting from the same city, and each person waits for the person before them to stop moving before starting. The people follow a simple algorithm until they reach their destination: they will look at all the outgoing roads from the current city, and choose the one that leads to the city with the smallest label. A person will stop when they either reach their destination, or reach a city with no outgoing roads. If at any point someone fails to reach their destination, the rest of the people still waiting in line will leave.

Before each person enters the transportation network, you can permanently close down any subset of roads to guarantee they reach their destination. The roads that you choose to close down will not be available for future people.

There are n cities, labeled from 1 to n. There are $n-1$ directed roads, and each road will always be from a lower labeled city to a higher labeled one. The network will form a rooted tree with city 1 as the root. There are m people that want to travel through the network. Each person starts from city 1, and has a specific destination city d in mind. These people will line up in the given order. What is the maximum number of people you can route correctly to their destination if you close roads optimally?

Input

The first line of input contains two integers n and m $(2 \leq n, m \leq 2 \times 10^5)$, where n is the number of cities and m is the number of people.

Each of the next $n-1$ lines contains two integers a and b $(1 \leq a < b \leq n)$, denoting a directed road from city a to b. These roads will describe a rooted tree with city 1 as the root.

Each of the next m lines contains a single integer d $(2 \leq d \leq n)$, denoting the destination city of the next person in line.

Output

Output a single integer, which is the maximum number of people you can route to the correct destination.

Sample Input 2

```
4 4
1 2
1 3
1 4
3
2
3
4
```

Sample Output 2

```
1
```

```cpp
void dfs(int node, int& curr_pos, VV<int>& paths, vi& d, V<bool>& visited) {
   visited[node] = true;
   // If we are at an end node with no children
   if (paths[node].size() == 0) {
       // We are able to say we can visit every node that we visited in our
       // current path
       while (curr_pos < d.size() && visited[d[curr_pos]]) {
           curr_pos++;
       }
   } else {
       // because its sorted we can do it this way
       for (int child : paths[node]) {
           dfs(child, curr_pos, paths, d, visited);
       }
   }
   visited[node] = false;
}
int main() {
   cin.tie(0)->sync_with_stdio(0);
   int n, m;
   cin >> n >> m;

   VV<int> paths(n, vi());
   rep(i, 0, n - 1) {
       int a, b;
       cin >> a >> b;
       paths[--a].push_back(--b);
   }
   rep(i, 0, n - 1) { sort(all(paths[i])); }
   vi d;
   rep(i, 0, m) {
       int c;
       cin >> c;
       d.push_back(--c);
   }
   int current_pos = 0;
   V<bool> visited(n);
   dfs(0, current_pos, paths, d, visited);
   cout << current_pos << endl;
}
```

Polygon Area

```python
def polygon_area(points):
  # Calculate the area with shoe-lace formula.
  # Works with both convex and concave polygons
  m = len(points)
  area = 0
  for i in range(m):
    x1, y1 = points[i]
    x2, y2 = points[(i+1) % m]
    area += x1*y2 - x2*y1
  return area / 2
```

Convex Hull Points

```python
# Convex hull takes in points and connects the outer points to make the largest polygon
def convex_hull(points): # O(n log n) complexity.
    # Remove duplicates to detect the case we have just one unique point.
MAKE SURE YOU DONT OVERUSE THIS SORT-POINTS NEEDS TO BE SORTED FOR CONVEX HULL,BUT IF POSSIBLE PRE-SORT THE POINTS!!
    points = sorted(set(points))
    if len(points) <= 1: #no points or a single point, possibly repeated multiple times
        return points
    # 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product. Returns a positive value, if OAB makes a counter-clockwise turn,negative for clockwise turn, and zero if the points are collinear.
    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])
    lower = [] # Build lower hull
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)
    upper = [] # Build upper hull
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)
    # Concatenation of the lower and upper hulls gives the convex hull.
    # Last point of each list is omitted because it is repeated at the beginning of the other list.
    return lower[:-1] + upper[:-1].
assert convex_hull([(i//10, i%10) for i in range(100)]) == [(0, 0), (9, 0), (9, 9), (0, 9)]
```

Range Queries

Sum of Range

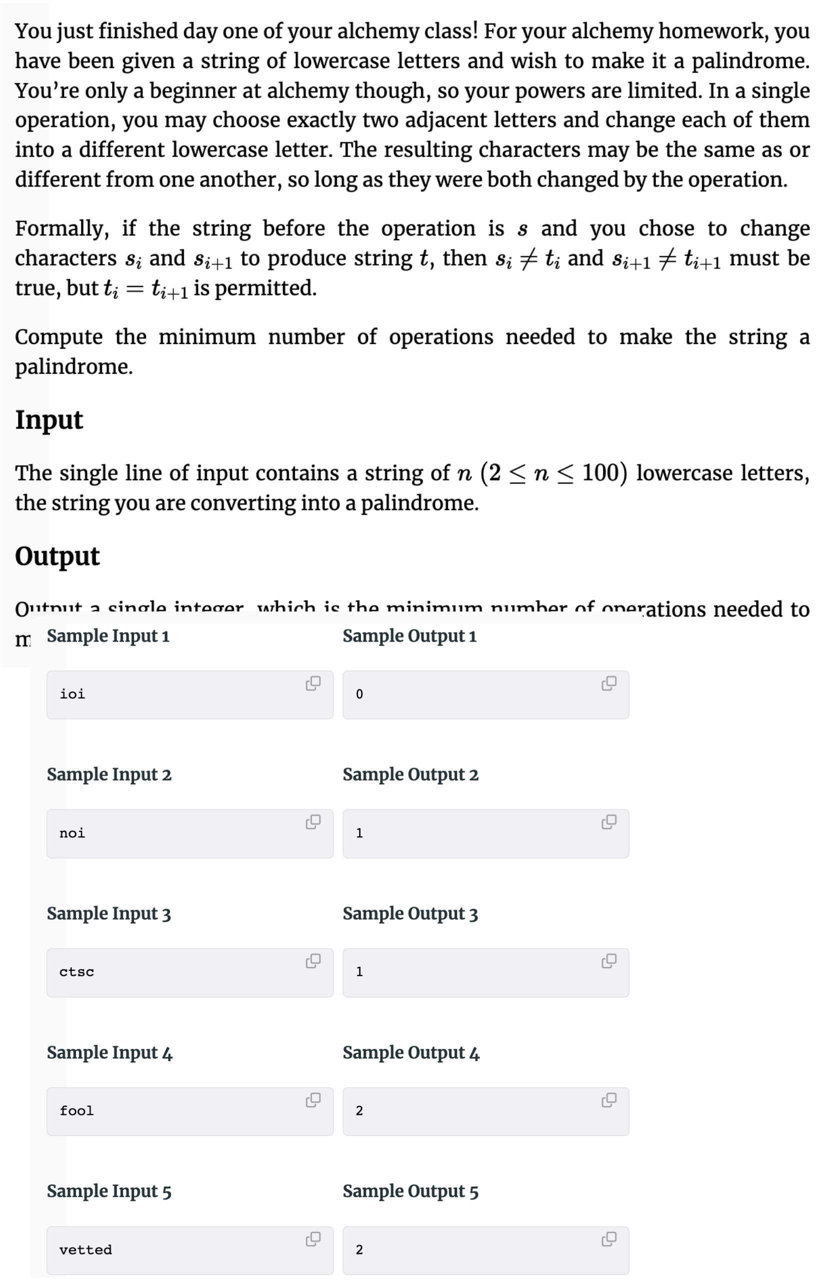
```python
class SumRangeQuery:
    def __init__(self, nums):
        """
        Create a sum range query obj -  O(n) Space(n)
        :param nums: The array of numbers to do a sum query on"""
        self.nums = nums
        self.pref = [nums[0]]
        for i in range(1, len(nums)):
            self.pref.append(self.pref[-1] + nums[i])
    def findSum(self, a, b):
        """
        Find the sum from a to b inclusive O(1)
        :param a: the starting index
        :param b: the ending index
        :return: the sum of the range"""
        second = self.pref[b]
        first = self.pref[a-1]
        if a == 0: first = 0
        return second - first
```

Min of Range

```python
class MinRangeQuery:# Create a min range query obj O(n logn) Space(n logn)
   def __init__(self, nums):# nums: The array of numbers to do a min query on
       self.nums = [i[1] for i in nums]
       self.n = len(nums)
       self.rows = math.ceil(math.log2(len(nums)))+1
       self.lookup = [[0] * self.rows for _ in range(self.n)]
       for r in range(self.n): self.lookup[r][0] = self.nums[r]
       for c in range(1, self.rows):
           _R = (1<<c)
           r = 0
           while r + _R <= self.n:
               self.lookup[r][c] = min(self.lookup[r][c-1], self.lookup[r+(1<<(c-1))][c-1])
               r += 1
   def findMin(self, a, b):#Find the sum from a to b inclusive O(1)-a:the starting index,b:the ending index
       # :return: the sum of the range
       length = b - a + 1
       k = int(math.log2(length))
       if self.nums[self.lookup[a][k]] <= self.nums[self.lookup[b - (1 << k) + 1][k]]:
           return self.nums[self.lookup[a][k]]
```

```
    else:
        return self.nums[self.lookup[b - (1 << k) + 1][k]]
```

Alchemy



You just finished day one of your alchemy class! For your alchemy homework, you have been given a string of lowercase letters and wish to make it a palindrome. You're only a beginner at alchemy though, so your powers are limited. In a single operation, you may choose exactly two adjacent letters and change each of them into a different lowercase letter. The resulting characters may be the same as or different from one another, so long as they were both changed by the operation.

Formally, if the string before the operation is s and you chose to change characters s_i and s_{i+1} to produce string t, then $s_i \neq t_i$ and $s_{i+1} \neq t_{i+1}$ must be true, but $t_i = t_{i+1}$ is permitted.

Compute the minimum number of operations needed to make the string a palindrome.

Input

The single line of input contains a string of n $(2 \leq n \leq 100)$ lowercase letters, the string you are converting into a palindrome.

Output

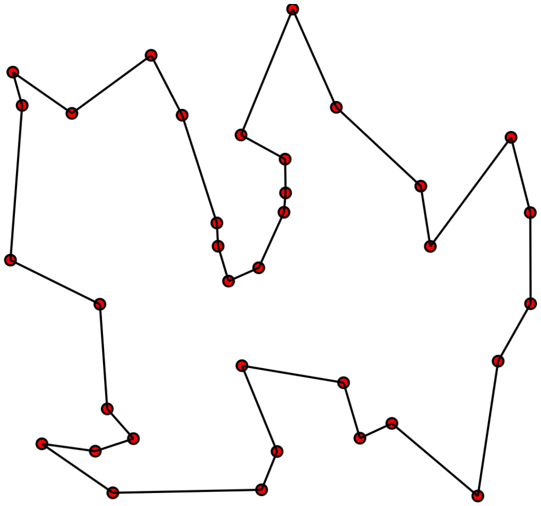
Output a single integer, which is the minimum number of operations needed to m

Sample Input 1	**Sample Output 1**
ioi	0
Sample Input 2	**Sample Output 2**
noi	1
Sample Input 3	**Sample Output 3**
ctsc	1
Sample Input 4	**Sample Output 4**
fool	2
Sample Input 5	**Sample Output 5**
vetted	2

```
#include <iostream>
#include <string>
using namespace std;

int main() {
 string S;
 while (cin >> S) {
   pair<int, int> dyn = {0, 1000000};
   for (int i = 0, j = S.size()-1; i < j; i++, j--) {
     if (S[i] == S[j]) {
       dyn = {min(dyn.first, dyn.second+1), dyn.second+1};
     } else {
       dyn = {dyn.second, min(dyn.first+1, dyn.second+1)};
     }
   }
   cout << min(dyn.first, dyn.second) << endl;
 }
}
```

Traveling Salesman Problem (TSP)



Sources:
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm#Example.5B4.5D
Algorithm from:
https://github.com/CarlEkerot/held-karp/blob/master/held-karp.py

Held-Karp Algorithm - Exact Solution in O(n^2 2^n) time

```python
# This algorithm does not run in reasonable time with 20+ nodes. 16 nodes or less is ideal.
def held_karp(dists): # n x n Distance matrix from node A to node B
 n = len(dists)
 C = {} # C[(subset_mask, k)] = (cost, previous_k)
 for k in range(1, n):
   C[(1 << k, k)] = (dists[0][k], 0) # init
 # Dynamic programming: choose best subsets based on smaller subsets
 for subset_size in range(2, n):
   for subset in itertools.combinations(range(1, n), subset_size):
     # Encode this subset as bitmask
     bits = 0
     for bit in subset:
       bits |= 1 << bit
     # Find the lowest cost to get to this subset
     for k in subset:
       prev = bits & ~(1 << k)
       res = []
       for m in subset:
         if m == 0 or m == k:
           continue
         res.append((C[(prev, m)][0] + dists[m][k], m))
       C[(bits, k)] = min(res)
 bits = (2**n - 1) - 1
 res = []
 for k in range(1, n):
   res.append((C[(bits, k)][0] + dists[k][0], k))
 opt, parent = min(res)
 # Backtrack to find shortest path
 path = []
 for i in range(n - 1):
   path.append(parent)
   new_bits = bits & ~(1 << parent)
   _, parent = C[(bits, parent)]
   bits = new_bits
 path.append(0)
 return opt, list(reversed(path))
```

Convex Hull in C++

```cpp
typedef pair<int, int> Point;

int cross(const Point& o, const Point& a, const Point& b) {
   return (a.first - o.first) * (b.second - o.second) - (a.second - o.second) * (b.first - o.first);
}

vector<Point> convex_hull(vector<Point>& points) {
   MAKE SURE YOU DONT OVERUSE THIS SORT - POINTS NEEDS TO BE SORTED FOR CONVEX HULL,
BUT IF POSSIBLE PRE-SORT THE POINTS!!
   sort(points.begin(), points.end());
   points.erase(unique(points.begin(), points.end()), points.end());

   if (points.size() <= 1) return points;

   vector<Point> lower, upper;
   for (const auto& p : points) {
       while (lower.size() >= 2 && cross(lower[lower.size() - 2], lower.back(), p) <= 0) {
           lower.pop_back();
       }
       lower.push_back(p);
   }

   for (auto it = points.rbegin(); it != points.rend(); ++it) {
       while (upper.size() >= 2 && cross(upper[upper.size() - 2], upper.back(), *it) <= 0) {
           upper.pop_back();
       }
       upper.push_back(*it);
   }

   lower.pop_back();
   upper.pop_back();
   lower.insert(lower.end(), upper.begin(), upper.end());
   return lower;
}
```

Combinations in C++

```cpp
vi my_array = {3, 4, 5, 7, 8, 9, 0, 34, 45, 2};

// 1 << n == 2^n
rep(i, 1, 1 << my_array.size()) {
   vi combo;
   For(j, my_array.size()) {
       if (i & (1 << j)) {
           int x = my_array[j];
           combo.push_back(x);
       }
   }

   For(i, combo.size()) cout << combo[i] << ' ';
   cout << '\n';
}
```

- Do we fully understand what the problem is asking?
    - How will the output need to be formatted?
    - Does it need to be ordered?
- Have we checked the bounds of the problem?
    - IE n=0 edge case
    - Have we considered a test case that tests the upper bounds?
- Does this problem contain any components of problems we have seen?
    - Parallel arrays or two pointer?
    - Set manipulation?
    - Heaps?
    - Finding all subsets?
- Can this problem be represented by a Graph?
    - Can the solution be found by traversing the graph once?
    - By organizing the graph?
    - By using a max or min flow?
    - By using the shortest path?
    - By creating a tree or union find?
    - By finding connected components?
- Can this problem be broken down into dependent subproblems?
    - Does it follow knapsack?
    - Is it similar to other dp problems like edit distance?
    - What are the states?
    - What are the choices?
- Are we accidentally adding an O(n) operation?
    - Item in list where list is size n
    - Are we copying an array in a loop?
    - Can we use maps or sets to remove duplicates?
- Can we combine parts of code to reduce time complexity?