Hakim Sabzevari University

CTRL+ALT+DEFEAT

Team Reference Document

Ali Ghanbari, Amirreza Zeraati, Rahmat Ansari

https://github.com/ctrl-alt-Defeat-icpc

2024

List of content

.... 1
1. STL 2
   1.1. bitscroll 2
   1.2. 128 bit 2
2. Segment Tree 2
   2.1. easy implementation 2
3. Math 3
   3.1. choose 3
4. Graph 4
   4.1. BFS 4
   4.2. bipartite 5
   4.3. cycle finding 5
   4.4. DFS 5
   4.5. floyd-warshall 6
   4.6. prim 6
   4.7. shortest cycle 6
   4.8. topologycal sort 7
5. Data Structures 10
   5.1. Array 10
   5.2. bitset 10
   5.3. dequeue 10
   5.4. link list 11
   5.5. Map 11
   5.6. priority queue 11
   5.7. queue 11
   5.8. set 12
   5.9. stack 12
   5.10. unordered map 12
   5.11. unordered set 12
   5.12. vector 13
   5.13. DS cheatsheet 13
6. Dynamic Programming 14
   6.1. counting paths matrix 14
   6.2. edit distance 15
   6.3. Egg Dropping 15
   6.4. fibonacci 15
   6.5. knapsack 01 16
   6.6. LCS 16
   6.7. LIS 16
   6.8. LPS 16
   6.9. MCM 17
   6.10. Minimum Coin Change 17
   6.11. optimal BST 17
   6.12. partition problem 18
   6.13. Regular Expression Matching 18
   6.14. ROD cutting 19
   6.15. subset sum 19
   6.16. two player game 19
   6.17. word break 20
   6.18. dp cheatsheet 20
10. other 22
   10.1. useful geo 22
   10.2. number of primes 22
   10.3. Factorials 23
   10.4. power of 3 23
   10.5. C(2n, n) 23
   10.6. Most Divisor 23

1. STL

1.1. bitscroll

```
__builtin_ctz(x); // first 1 from left (index)
__builtin_popcount(x); // count of 1 in numbers bit
__builtin_ctzll(x); // for long long
__builtin_popcountll(x); // ...
```

1.2. 128 bit

```
__int128 read() {
    __int128 x = 0, f = 1;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
        if (ch == '-') f = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    return x * f;
}
void print(__int128 x) {
    if (x < 0) {
        putchar('-');
        x = -x;
    }
    if (x > 9) print(x / 10);
    putchar(x % 10 + '0');
}
bool cmp(__int128 x, __int128 y) { return x > y; }

int main() {
    __int128 x = read();
    print(x);
    cout << endl;
    return 0;
}
```

2. Segment Tree

2.1. easy implementation

```
const int N = 1e5;  // limit for array size
```

```
int n;  // array size
int t[2 * N];

void build() {  // build the tree
  for (int i = n - 1; i > 0; --i) t[i] = t[i<<1] +
t[i<<1|1];
}

void modify(int p, int value) {  // set value at
position p
  for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] =
t[p] + t[p^1];
}

int query(int l, int r) {  // sum on interval [l, r)
  int res = 0;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) res += t[l++];
    if (r&1) res += t[--r];
  }
  return res;
}

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) scanf("%d", t + n +
i);
  build();
  modify(0, 2);
  printf("%d\n", query(3, 11));
  return 0;
}
```

2.2. with lazy propagation

```
const int N = 1e5 + 5;
int n;
int seg[2 * N], lazy[2 * N], a[N];
int segSize;

void build(int u = 1, int ul = 0, int ur = n) {
    if(ur - ul < 2){
        seg[u] = a[ul];
        return;
    }
    int mid = (ul + ur) / 2;
    build(u * 2, ul, mid);
    build(u * 2 + 1, mid, ur);
    seg[u] = seg[u * 2] + seg[u * 2 + 1];
}

void upd(int u, int ul, int ur, int x){
    lazy[u] += x;
    seg[u] += (ur - ul) * x;
}
void shift(int u, int ul, int ur){
    int mid = (ul + ur) / 2;
    upd(u * 2, ul, mid, lazy[u]);
    upd(u * 2 + 1, mid, ur, lazy[u]);
    lazy[u] = 0;
}
void increase(int l, int r, int x, int u = 1, int ul
= 0, int ur = n){
    if(l >= ur || ul >= r)return;
    if(l <= ul && ur <= r){
        upd(u, ul, ur, x);
        return;
    }
    shift(u, ul, ur);
    int mid = (ul + ur) / 2;
    increase(l, r, x, u * 2, ul, mid);
    increase(l, r, x, u * 2 + 1, mid, ur);
    seg[u] = seg[u * 2] + seg[u * 2 + 1];
}
int sum(int l, int r, int u = 1, int ul = 0, int ur
= n){
    if(l >= ur || ul >= r)return 0;
    if(l <= ul && ur <= r)return seg[u];
    shift(u, ul, ur);
    int mid = (ul + ur) / 2;
    return sum(l, r, u * 2, ul, mid) + sum(l, r, u *
2 + 1, mid, ur);
}

void showSegments() {
    for(int i = 0; i < segSize; i++)
        cout << seg[i] << ' ';
    cout << endl;
}

void Main() {
    cin >> n;
    segSize = 2;
    while(segSize / 2 <= n) segSize *= 2;
    for(int i = 0; i < n; i++)
        cin >> a[i];
    build();

}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0); cout.tie(0);
    Main();
    return 0;
}
```

3. Math

3.1. choose

```
#define ll long long
const int N = 2e3 + 5;
const ll M = 1e9 + 7;
```

```
ll fact[N], inv[N];
int r, n, q;

ll exp(ll b, ll p, ll m) {
    b %= m;
    ll result = 1;
    while(p) {
        if(p % 2)
            result = result * b % m;
        b = b * b % m;
        p /= 2;
    }
    return result;
}

void preProcess() {
    fact[0] = 1;
    for(int i = 1; i < N; i++)
        fact[i] = fact[i - 1] * i % M;
    inv[N - 1] = exp(fact[N - 1], M - 2, M);
    for(int i = N - 1; i > 0; i--)
        inv[i - 1] = inv[i] * i % M;
}

ll choose(int n, int r) {
    if(r > n) return 0;
    return fact[n] * inv[r] % M * inv[n - r] % M;
}

void Main() {
    cin >> q;
    while(q--) {
        cin >> n >> r;
        cout << choose(n, r) << '\n';
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0); cout.tie(0);
    preProcess();
    Main();
    return 0;
}
```

3.2. gcd

```
int gcd (int a, int b) {
    return b ? gcd (b, a % b) : a;
}

// fast version...
int gcd(int a, int b) {
    if (!a || !b)
        return a | b;
    unsigned shift = __builtin_ctz(a | b);
    a >>= __builtin_ctz(a);
    do {
        b >>= __builtin_ctz(b);
        if (a > b)
            swap(a, b);
        b -= a;
    } while (b);
    return a << shift;
}
```

3.3. compressing

```
// compressing
sort(temp_values, temp_values + n);
int numOfUnique = unique(temp_values, temp_values +
n) - temp_values;
for(int i = 0; i < n; i++)
    h[i] = lower_bound(temp_values, temp_values +
numOfUnique, h[i]) - temp_values;
```

3.4. lower bound and upper bound

```
int main() {
    vector<int> v = {11, 34, 56, 67, 89};

      // Finding lower bound of 56
    cout << *lower_bound(v.begin(), v.end(), 56)
      << endl;

      // Finding upper bound of 56
    cout << *upper_bound(v.begin(), v.end(), 56);
    return 0;
}
```

Output:

```
56
67
```

4. Graph

4.1. BFS

```
#define distance d
const int maxN = 1e5 + 10, oo = 1e9;
vector <int> adj[maxN];
int distance[maxN];
queue<int> q;

void BFS(int n, int r) {
    for (int i=1; i<=n; i++) distance[i] = oo;
    distance[r] = 0;
    q.push(r);
    while(q.size()) {
        int v = q.front();
        q.pop();

        for (auto u : adj[v])
            if(distance[u] > distance[v] + 1) {
                distance[u] = distance[v] + 1;
                q.push(u);
            }
    }
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;
    for (int i=0; i<m; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    BFS(n, 1);
    for (int i=1; i<=n; i++)
        cout << i << ':' << distance[i] << '\n';
}
```

4.2. bipartite

```
const int maxN = 1e5 + 10;
vector <int> adj[maxN];
bool mark[maxN];
int color[maxN];
bool bipartite = true;

void DFS(int v, int parent) {
    mark[v] = true;

    if(parent != -1) color[v] = 1 - color[parent];
    else color[v] = 1;

    for (auto u : adj[v]) {
        if(!mark[u])
            DFS(u, v);
        else if(color[u] == color[v])
            bipartite = false;
    }

}

int main() {
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;
    for (int i=0; i<m; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    for (int i=1; i<=n; i++) {
        if(mark[i]) continue ;

        DFS(i, -1); //root does not have parent.

    }
    if(bipartite) cout << "Graph Is Bipartite\n";
    else cout << "Graph Is Not Bipartite\n";
}
```

4.3. cycle finding

```
const int maxN = 1e5 + 10;
vector <int> adj[maxN];
bool mark[maxN];
bool cycle_found = false;

void DFS(int v, int parent) {
    mark[v] = true;

    for (auto u : adj[v]) {
        if(!mark[u]) DFS(u, v); //u's parent is v.
        else if(u != parent) cycle_found = true;
    }

}

int main() {
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;
    for (int i=0; i<m; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    for (int i=1; i<=n; i++) {
        if(mark[i]) continue ;
        DFS(i, -1); //root does not have parent.
    }
    if(cycle_found) cout << "Graph has Cycle\n";
    else cout << "Graph does not have Cycle\n";
}
```

4.4. DFS

```
const int maxN = 1e5 + 10;
vector <int> adj[maxN];
bool mark[maxN];
vector <int> component;

void DFS(int v) {
    mark[v] = true;
    component.push_back(v);
    for (auto u : adj[v])
        if(!mark[u]) DFS(u);
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;
    for (int i=0; i<m; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    for (int i=1; i<=n; i++) {
        if(mark[i]) continue ;
        component.clear();
        DFS(i);
        for (auto v : component)
            cout << v << ' ';
        cout << '\n';
    }
}
```

4.5. floyd-warshall

```
// Implementing floyd warshall algorithm
void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;
  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];
  // Adding vertices individually
  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] <
matrix[i][j])
          matrix[i][j] = matrix[i][k] +
matrix[k][j];
      }
    }
  }
//   printMatrix(matrix);
}
```

4.6. prim

```
// Function to find sum of weights of edges of the
Minimum Spanning Tree.
int spanningTree(int V, int E, vector<vector<int>>
&edges) {
    // Create an adjacency list representation of
the graph
    vector<vector<int>> adj[V];
    // Fill the adjacency list with edges and their
weights
    for (int i = 0; i < E; i++) {
        int u = edges[i][0];
        int v = edges[i][1];
        int wt = edges[i][2];
        adj[u].push_back({v, wt});
        adj[v].push_back({u, wt});
    }
    // Create a priority queue to store edges with
their weights
    priority_queue<pair<int,int>,
vector<pair<int,int>>, greater<pair<int,int>>> pq;
    // Create a visited array to keep track of
visited vertices
    vector<bool> visited(V, false);
    // Variable to store the result (sum of edge
weights)
    int res = 0;

    // Start with vertex 0
    pq.push({0, 0});

    // Perform Prim's algorithm to find the Minimum
Spanning Tree
    while(!pq.empty()){
        auto p = pq.top();
        pq.pop();

        int wt = p.first;  // Weight of the edge
        int u = p.second;  // Vertex connected to
the edge

        if(visited[u] == true){
            continue;  // Skip if the vertex is
already visited
        }
        res += wt;  // Add the edge weight to the
result
        visited[u] = true;  // Mark the vertex as
visited
        // Explore the adjacent vertices
        for(auto v : adj[u]){
            // v[0] represents the vertex and v[1]
represents the edge weight
            if(visited[v[0]] == false){
                pq.push({v[1], v[0]});  // Add the
adjacent edge to the priority queue
            }
        }
    }
    return res;  // Return the sum of edge weights
of the Minimum Spanning Tree
}

int main() {
    vector<vector<int>> graph = {{0, 1, 5},
                                 {1, 2, 3},
                                 {0, 2, 1}};
    cout << spanningTree(3, 3, graph) << endl;

    return 0;
}
```

4.7. shortest cycle

```
//this code works for simple graphs.
const int maxN = 1010, oo = 1e9;
vector <int> adj[maxN];
int deleted, distances[maxN];
queue<int> q;

void BFS(int n, int r) {
    for (int i=1; i<=n; i++) distances[i] = oo;
    distances[r] = 0;
    q.push(r);
    while(q.size()) {
        int v = q.front();
        q.pop();
        for (auto u : adj[v]) {
            if(v == r && u == deleted) continue;
//ignore deleted edge.
            if(distances[u] > distances[v] + 1) {
                distances[u] = distances[v] + 1;
                q.push(u);
            }
        }
    }
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;
    for (int i=0; i<m; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v); adj[v].push_back(u);
    }
    int length = oo;
    for (int i=1; i<=n; i++) {
        for (auto u : adj[i]) {
            deleted = u;
            BFS(n, i);
            length = min(length, distances[u] + 1);
        }
    }
    if(length == oo) cout << "Graph Does Not Have
Cycle\n";
    else cout << "Minimum Cycle Length is : " <<
length << '\n';
}
```

4.8. topologycal sort

```
int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
    ans.push_back(v);
}

void topological_sort() {
    visited.assign(n, false);
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    reverse(ans.begin(), ans.end());
}
```

4.9. lowest common Ancestor

```
struct LCA {
    vector<int> height, euler, first, segtree;
    vector<bool> visited;
    int n;

    LCA(vector<vector<int>> &adj, int root = 0) {
        n = adj.size();
        height.resize(n);
        first.resize(n);
        euler.reserve(n * 2);
        visited.assign(n, false);
        dfs(adj, root);
        int m = euler.size();
        segtree.resize(m * 4);
        build(1, 0, m - 1);
    }

    void dfs(vector<vector<int>> &adj, int node, int
h = 0) {
        visited[node] = true;
        height[node] = h;
        first[node] = euler.size();
        euler.push_back(node);
        for (auto to : adj[node]) {
            if (!visited[to]) {
                dfs(adj, to, h + 1);
                euler.push_back(node);
            }
        }
    }

    void build(int node, int b, int e) {
        if (b == e) {
            segtree[node] = euler[b];
        } else {
            int mid = (b + e) / 2;
            build(node << 1, b, mid);
            build(node << 1 | 1, mid + 1, e);
            int l = segtree[node << 1], r =
segtree[node << 1 | 1];
            segtree[node] = (height[l] < height[r])
? l : r;
        }
    }

    int query(int node, int b, int e, int L, int R)
{
        if (b > R || e < L)
```

```
            return -1;
        if (b >= L && e <= R)
            return segtree[node];
        int mid = (b + e) >> 1;

        int left = query(node << 1, b, mid, L, R);
        int right = query(node << 1 | 1, mid + 1, e,
L, R);
        if (left == -1) return right;
        if (right == -1) return left;
        return height[left] < height[right] ? left :
right;
    }

    int lca(int u, int v) {
        int left = first[u], right = first[v];
        if (left > right)
            swap(left, right);
        return query(1, 0, euler.size() - 1, left,
right);
    }
};
```

4.10. lowest common Ancestor (binary lifting)

```
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];

    for (int u : adj[v]) {
        if (u != p)
            dfs(u, v);
    }

    tout[v] = ++timer;
}

bool is_ancestor(int u, int v)
{
    return tin[u] <= tin[v] && tout[u] >= tout[v];
}

int lca(int u, int v)
{
    if (is_ancestor(u, v))
        return u;
    if (is_ancestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
    }
    return up[u][0];
}

void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root);
}
```

4.11. hungarian algorithm (assignment problem)

```
vector<int> u (n+1), v (m+1), p (m+1), way (m+1);
for (int i=1; i<=n; ++i) {
    p[0] = i;
    int j0 = 0;
    vector<int> minv (m+1, INF);
    vector<bool> used (m+1, false);
    do {
        used[j0] = true;
        int i0 = p[j0],  delta = INF,  j1;
        for (int j=1; j<=m; ++j)
            if (!used[j]) {
                int cur = A[i0][j]-u[i0]-v[j];
                if (cur < minv[j])
                    minv[j] = cur,  way[j] = j0;
                if (minv[j] < delta)
                    delta = minv[j],  j1 = j;
            }
        for (int j=0; j<=m; ++j)
            if (used[j])
                u[p[j]] += delta,  v[j] -= delta;
            else
                minv[j] -= delta;
        j0 = j1;
    } while (p[j0] != 0);
    do {
        int j1 = way[j0];
        p[j0] = p[j1];
        j0 = j1;
    } while (j0);
}
```

4.12. 2SAT

```
struct TwoSatSolver {
    int n_vars;
    int n_vertices;
    vector<vector<int>> adj, adj_t;
    vector<bool> used;
    vector<int> order, comp;
    vector<bool> assignment;

    TwoSatSolver(int _n_vars) : n_vars(_n_vars),
n_vertices(2 * n_vars), adj(n_vertices),
adj_t(n_vertices), used(n_vertices), order(),
comp(n_vertices, -1), assignment(n_vars) {
        order.reserve(n_vertices);
    }
    void dfs1(int v) {
        used[v] = true;
        for (int u : adj[v]) {
            if (!used[u])
                dfs1(u);
        }
        order.push_back(v);
    }

    void dfs2(int v, int cl) {
        comp[v] = cl;
        for (int u : adj_t[v]) {
            if (comp[u] == -1)
                dfs2(u, cl);
        }
    }

    bool solve_2SAT() {
        order.clear();
        used.assign(n_vertices, false);
        for (int i = 0; i < n_vertices; ++i) {
            if (!used[i])
                dfs1(i);
        }

        comp.assign(n_vertices, -1);
        for (int i = 0, j = 0; i < n_vertices; ++i)
{
            int v = order[n_vertices - i - 1];
            if (comp[v] == -1)
                dfs2(v, j++);
        }

        assignment.assign(n_vars, false);
        for (int i = 0; i < n_vertices; i += 2) {
            if (comp[i] == comp[i + 1])
                return false;
            assignment[i / 2] = comp[i] > comp[i +
1];
        }
        return true;
    }

    void add_disjunction(int a, bool na, int b, bool
nb) {
        // na and nb signify whether a and b are to
be negated
        a = 2 * a ^ na;
        b = 2 * b ^ nb;
        int neg_a = a ^ 1;
        int neg_b = b ^ 1;
        adj[neg_a].push_back(b);
        adj[neg_b].push_back(a);
        adj_t[b].push_back(neg_a);
        adj_t[a].push_back(neg_b);
    }

    static void example_usage() {
        TwoSatSolver solver(3); // a, b, c
        solver.add_disjunction(0, false, 1,
true); //     a  v  not b
        solver.add_disjunction(0, true, 1,
true);  // not a  v  not b
        solver.add_disjunction(1, false, 2, false);
//      b  v     c
        solver.add_disjunction(0, false, 0, false);
//      a  v     a
        assert(solver.solve_2SAT() == true);
        auto expected = vector<bool>(True, False,
True);
        assert(solver.assignment == expected);
    }
};
```

4.13. Heavy-light decomposition

```
vector<int> parent, depth, heavy, head, pos;
int cur_pos;

int dfs(int v, vector<vector<int>> const& adj) {
    int size = 1;
    int max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c, adj);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size, heavy[v] = c;
        }
    }
    return size;
}

void decompose(int v, int h, vector<vector<int>>
const& adj) {
    head[v] = h, pos[v] = cur_pos++;
    if (heavy[v] != -1)
        decompose(heavy[v], h, adj);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy[v])
```

```
            decompose(c, c, adj);
    }
}

void init(vector<vector<int>> const& adj) {
    int n = adj.size();
    parent = vector<int>(n);
    depth = vector<int>(n);
    heavy = vector<int>(n, -1);
    head = vector<int>(n);
    pos = vector<int>(n);
    cur_pos = 0;

    dfs(0, adj);
    decompose(0, 0, adj);
}

int query(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = parent[head[b]])
{
        if (depth[head[a]] > depth[head[b]])
            swap(a, b);
        int cur_heavy_path_max =
segment_tree_query(pos[head[b]], pos[b]);
        res = max(res, cur_heavy_path_max);
    }
    if (depth[a] > depth[b])
        swap(a, b);
    int last_heavy_path_max =
segment_tree_query(pos[a], pos[b]);
    res = max(res, last_heavy_path_max);
    return res;
}
```

5. Data Structures

5.1. Array

```
int main() {
    array<int, 5> arr = {1, 2, 3, 4, 5};

    // Accessing elements
    cout << "Element at index 2: " << arr[2] <<
endl;

    // Size of the array
    cout << "Size of array: " << arr.size() << endl;

    // Fill array with a value
    arr.fill(10);
    cout << "Array after fill: ";
    for (int num : arr)
        cout << num << " ";
    cout << endl;

    return 0;
}
```

5.2. bitset

```
int main() {
    bitset<8> b1; // All bits initialized to 0
    bitset<8> b2("11001010");

    // Set bit at index 3 to 1
    b1.set(3);
    cout << "b1 after set(3): " << b1 << endl;

    // Reset all bits of b2
    b2.reset();
    cout << "b2 after reset: " << b2 << endl;

    // Flip all bits of b1
    b1.flip();
    cout << "b1 after flip: " << b1 << endl;

    // Access the bit at index 2
    cout << "b1[2]: " << b1[2] << endl;

    // Test if bit at index 2 is set to 1
    if (b1.test(2))
    {
        cout << "Bit 2 is set to 1." << endl;
    }

    // Count the number of 1's in b1
    cout << "Number of 1's in b1: " << b1.count() <<
endl;

    return 0;
}
```

5.3. dequeue

```
int main() {
    deque<int> deq = {1, 2, 3, 4, 5};

    // Adding elements
    deq.push_front(0); // Add at front
    deq.push_back(6);  // Add at back

    // Removing elements
    deq.pop_front(); // Remove from front
    deq.pop_back();  // Remove from back

    // Accessing elements
    cout << "First element: " << deq.front() <<
endl;
    cout << "Last element: " << deq.back() << endl;

    // Iterating through deque
    cout << "Deque elements: ";
    for (int x : deq)
        cout << x << " ";
    cout << endl;

    // Size of deque
    cout << "Deque size: " << deq.size() << endl;

    return 0;
}
```

5.4. link list

```
int main() {
    list<int> lst = {1, 2, 3, 4, 5};

    // Adding elements
    lst.push_back(6);  // Add to the end
    lst.push_front(0); // Add to the front

    // Removing elements
    lst.pop_back();  // Remove from the end
    lst.pop_front(); // Remove from the front

    // Iterating through list
    cout << "List elements: ";
    for (int x : lst)
        cout << x << " ";
    cout << endl;

    // Size of list
    cout << "List size: " << lst.size() << endl;

    return 0;
}
```

5.5. Map

```
int main() {
    map<string, int> m;

    // Insert key-value pairs
    m["apple"] = 3;
    m["banana"] = 2;
    m["orange"] = 5;

    // Access value by key
    cout << "Value for apple: " << m["apple"] <<
endl;

    // Iterate through map
    cout << "Map elements: ";
    for (auto &pair : m)
    {
        cout << pair.first << ": " << pair.second <<
" ";
    }
    cout << endl;

    // Remove an element
    m.erase("banana");
    cout << "Map after erase: ";
    for (auto &pair : m)
    {
        cout << pair.first << ": " << pair.second <<
" ";
    }
    cout << endl;

    // Check if key exists
    if (m.find("orange") != m.end())
    {
        cout << "Orange is in the map." << endl;
    }

    return 0;
}
```

5.6. priority queue

```
int main() {
    priority_queue<int> pq;

    // Insert elements into the priority queue
    pq.push(10);
    pq.push(30);
    pq.push(20);

    cout << "Top element (max priority): " <<
pq.top() << endl;

    // Pop the top element
    pq.pop();
    cout << "Top element after pop: " << pq.top() <<
endl;

    // Size of the priority queue
    cout << "Size of priority queue: " << pq.size()
<< endl;

    return 0;
}
```

5.7. queue

```
int main() {
    queue<int> q;

    // Push elements onto the queue
    q.push(10);
    q.push(20);
    q.push(30);
    cout << "Front element: " << q.front() << endl;

    // Pop an element from the queue
    q.pop();
    cout << "Front element after pop: " << q.front()
<< endl;

    // Size of the queue
    cout << "Size of queue: " << q.size() << endl;

    // Check if the queue is empty
    if (q.empty()) {
        cout << "Queue is empty." << endl;
    }
    else {
        cout << "Queue is not empty." << endl;
    }

    return 0;
}
```

5.8. set

```
int main() {
    set<int> s;

    // Insert elements
    s.insert(10);
    s.insert(20);
    s.insert(15);

    // Print elements in sorted order
    cout << "Set elements: ";
    for (int num : s)
        cout << num << " ";
    cout << endl;

    // Check if an element exists
    if (s.find(15) != s.end())
    {
        cout << "15 is in the set." << endl;
    }

    // Remove an element
    s.erase(10);
    cout << "Set after erase: ";
    for (int num : s)
        cout << num << " ";
    cout << endl;

    // Size of the set
    cout << "Size of set: " << s.size() << endl;

    return 0;
}
```

5.9. stack

```
int main() {
    stack<int> s;

    // Push elements onto the stack
    s.push(10);
    s.push(20);
    s.push(30);
    cout << "Top element after push: " << s.top() <<
endl;

    // Pop an element from the stack
    s.pop();
    cout << "Top element after pop: " << s.top() <<
endl;

    // Check if stack is empty
    if (s.empty()) {
        cout << "Stack is empty." << endl;
    }
    else {
        cout << "Stack is not empty." << endl;
    }

    // Size of the stack
    cout << "Size of stack: " << s.size() << endl;

    return 0;
}
```

5.10. unordered map

```
int main() {
    unordered_map<int, string> um;

    // Inserting elements
    um[1] = "apple";
    um[2] = "banana";
    um[3] = "cherry";

    // Accessing elements
    cout << "Key 2 maps to: " << um[2] << endl;

    // Checking if key exists
    if (um.find(4) == um.end())
        cout << "Key 4 not found!" << endl;

    // Iterating through unordered map
    cout << "Unordered map elements: ";
    for (auto &pair : um)
    {
        cout << pair.first << " -> " << pair.second
<< " | ";
    }
    cout << endl;

    return 0;
}
```

5.11. unordered set

```
int main() {
    unordered_set<int> us;

    // Insert elements
    us.insert(10);
    us.insert(20);
    us.insert(15);

    // Print elements
    cout << "Unordered set elements: ";
    for (int num : us)
        cout << num << " ";
    cout << endl;

    // Check if an element exists
    if (us.find(15) != us.end())
    {
        cout << "15 is in the unordered set." <<
endl;
    }

    // Remove an element
    us.erase(10);
    cout << "Unordered set after erase: ";
    for (int num : us)
        cout << num << " ";
    cout << endl;

    return 0;
}
```

5.12. vector

```
int main() {
    // Create a vector of integers
    vector<int> v;

    // Add elements using push_back
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    cout << "Vector after push_back: ";
    for (int num : v)
        cout << num << " ";
    cout << endl;

    // Accessing elements using at() and indexing
    cout << "Element at index 1: " << v.at(1) <<
endl;
    cout << "Element at index 0: " << v[0] << endl;

    // Pop an element from the back
    v.pop_back();
    cout << "Vector after pop_back: ";
    for (int num : v)
        cout << num << " ";
    cout << endl;

    // Insert an element at a specific position
    v.insert(v.begin() + 1, 25); // Insert 25 at
index 1
    cout << "Vector after insert: ";
    for (int num : v)
        cout << num << " ";
    cout << endl;

    // Remove an element from the vector
    v.erase(v.begin() + 1); // Remove the element at
index 1
    cout << "Vector after erase: ";
    for (int num : v)
        cout << num << " ";
    cout << endl;

    // Resize the vector
    v.resize(5, 50); // Resize to size 5, fill new
elements with 50
    cout << "Vector after resize: ";
    for (int num : v)
        cout << num << " ";
    cout << endl;

    // Get the size of the vector
    cout << "Size of vector: " << v.size() << endl;

    // Clear the vector
    v.clear();
    cout << "Vector after clear: " << v.size() << "
(size is now zero)" << endl;

    return 0;
}
```

5.13. DS cheatsheet

## 1. Vector
- Description: Dynamic array that allows fast random access.
- Methods:
  - push_back(x): Add element x to the end.
  - pop_back(): Remove the last element.
  - at(i): Access the element at index i (bounds checked).
  - operator[]: Access the element at index i (no bounds check).
  - size(): Return the number of elements.
  - empty(): Check if the vector is empty.
  - resize(n, x): Resize the vector to size n and fill new elements with x.
  - clear(): Remove all elements.

## 2. Stack
- Description: Last-In-First-Out (LIFO) structure, used for backtracking problems.
- Methods:
  - push(x): Add element x to the top.
  - pop(): Remove the top element.
  - top(): Get the top element.
  - size(): Return the number of elements.
  - empty(): Check if the stack is empty.

## 3. Queue

- Description: First-In-First-Out (FIFO) structure, ideal for problems involving processing in order.
- Methods:
  - push(x): Add element x to the back.
  - pop(): Remove the front element.
  - front(): Get the front element.
  - back(): Get the back element.
  - size(): Return the number of elements.
  - empty(): Check if the queue is empty.

## 4. Priority Queue (Max-Heap by default)

- Description: A heap-based structure that always gives the maximum element.
- Methods:
  - push(x): Add element x to the queue.
  - pop(): Remove the largest element.
  - top(): Get the largest element.
  - size(): Return the number of elements.
  - empty(): Check if the queue is empty.

## 5. Set
- Description: Collection of unique elements in sorted order.
- Methods:
  - insert(x): Add element x.
  - erase(x): Remove element x.
  - find(x): Check if element x exists.
  - size(): Return the number of elements.
  - empty(): Check if the set is empty.
  - clear(): Remove all elements.

## 6. Map
- Description: Stores key-value pairs in sorted order based on keys.
- Methods:
  - insert({key, value}): Add key-value pair.
  - erase(key): Remove element by key.
  - find(key): Check if a key exists.
  - operator[]: Access the value associated with a key.
  - size(): Return the number of elements.
  - empty(): Check if the map is empty.
  - clear(): Remove all elements.

## 7. Unordered Set
- Description: Collection of unique elements with no specific order.
- Methods:
  - insert(x): Add element x.
  - erase(x): Remove element x.
  - find(x): Check if element x exists.
  - size(): Return the number of elements.
  - empty(): Check if the unordered set is empty.

## 8. Unordered Map
- Description: Stores key-value pairs with no specific order.
- Methods:
  - insert({key, value}): Add key-value pair.
  - erase(key): Remove element by key.
  - find(key): Check if a key exists.
  - operator[]: Access the value associated with a key.
  - size(): Return the number of elements.
  - empty(): Check if the unordered map is empty.

## 9. Bitset
- Description: A space-efficient container for a fixed-size sequence of bits (0 or 1).
- Methods:
  - set(i): Set bit at index i to 1.
  - reset(i): Set bit at index i to 0.
  - flip(i): Toggle the bit at index i.
  - test(i): Check if the bit at index i is 1.
  - count(): Count the number of bits set to 1.
  - size(): Return the number of bits.
  - operator[]: Access the bit at index i.
  - to_string(): Convert bitset to string.

## 10. Array
- Description: Fixed-size array used for fast access, but size cannot be changed after initialization.
- Methods:
  - fill(x): Fill all elements with the value x.
  - size(): Return the number of elements.
  - operator[]: Access element at index i.
  - at(i): Access element at index i with bounds checking.
  - front(): Get the first element.
  - back(): Get the last element.

## 11. Deque
- Description: Double-ended queue that allows fast insertion and removal at both ends.
- Methods:
  - push_front(x): Add element x to the front.
  - push_back(x): Add element x to the back.
  - pop_front(): Remove the front element.
  - pop_back(): Remove the back element.
  - front(): Get the front element.
  - back(): Get the back element.
  - size(): Return the number of elements.
  - empty(): Check if the deque is empty.

## 12. Linked List (Using STL List)

- Description: Doubly linked list that allows fast insertion and deletion at both ends.
- Methods:
  - push_back(x): Add element x to the back.
  - push_front(x): Add element x to the front.
  - pop_back(): Remove the last element.
  - pop_front(): Remove the first element.
  - size(): Return the number of elements.
  - empty(): Check if the list is empty.
  - front(): Get the first element.
  - back(): Get the last element.
  - clear(): Remove all elements.

6. Dynamic Programming

6.1. counting paths matrix

```
// Function to count the number of unique paths in a matrix
int countPaths(int n, int m) {
    vector<vector<int>> dp(n, vector<int>(m, 0));

    // Starting point: only one way to be at the start
    dp[0][0] = 1;

    // Fill the DP table for first row and first column
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (i > 0)
                dp[i][j] += dp[i - 1][j]; // From top
            if (j > 0)
```

```
                dp[i][j] += dp[i][j - 1]; // From
left
        }
    }

    return dp[n - 1][m - 1]; // Return the number of
paths to bottom-right corner
}

int main() {
    int n = 3, m = 3;                 // Example
matrix dimensions
    cout << countPaths(n, m) << endl; // Output the
result
    return 0;
}
```

6.2. edit distance

```
// Function to compute the Edit Distance
int editDistance(string str1, string str2, int m,
int n) {
    vector<vector<int>> dp(m + 1, vector<int>(n +
1));

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0)
                dp[i][j] = j;
            else if (j == 0)
                dp[i][j] = i;
            else if (str1[i - 1] == str2[j - 1])
                dp[i][j] = dp[i - 1][j - 1];
            else
                dp[i][j] = 1 + min({dp[i - 1][j -
1], dp[i][j - 1], dp[i - 1][j]});
        }
    }
    return dp[m][n];
}

int main() {
    string str1 = "sitting", str2 = "kitten";
    int m = str1.length(), n = str2.length();

    cout << "Edit Distance: " << editDistance(str1,
str2, m, n) << endl;
    return 0;
}
```

6.3. Egg Dropping

```
// Function to find the minimum number of attempts
needed
int eggDrop(int eggs, int floors)
{
    vector<vector<int>> dp(eggs + 1,
vector<int>(floors + 1, 0));

    for (int i = 1; i <= eggs; i++)
        dp[i][0] = 0;
    for (int j = 0; j <= floors; j++)
        dp[1][j] = j;

    for (int i = 2; i <= eggs; i++)
    {
        for (int j = 2; j <= floors; j++)
        {
            dp[i][j] = INT_MAX;
            for (int x = 1; x <= j; x++)
            {
                dp[i][j] = min(dp[i][j], 1 +
max(dp[i - 1][x - 1], dp[i][j - x]));
            }
        }
    }
    return dp[eggs][floors];
}

int main()
{
    int eggs = 2, floors = 10;
    cout << "Minimum attempts: " << eggDrop(eggs,
floors) << endl;
    return 0;
}
```

6.4. fibonacci

```
// Fibonacci sequence using dynamic programming
(Memoization)

// Function to compute Fibonacci number
int fib(int n, vector<int> &dp)
{
    // Base cases
    if (n <= 1)
        return n;

    // If the value is already computed, return it
    if (dp[n] != -1)
        return dp[n];

    // Store the computed value in dp array
    dp[n] = fib(n - 1, dp) + fib(n - 2, dp);
    return dp[n];
}

int main()
{
    int n = 10;
    vector<int> dp(n + 1, -1); // Initialize dp
array with -1
```

```
    cout << "Fibonacci of " << n << " is " << fib(n,
dp) << endl;
    return 0;
}
```

6.5. knapsack 01

```
// Function to solve the 0/1 Knapsack problem
int knapsack(int W, vector<int> &wt, vector<int>
&val, int n)
{
    vector<vector<int>> dp(n + 1, vector<int>(W + 1,
0)); // DP table

    for (int i = 1; i <= n; i++)
    {
        for (int w = 0; w <= W; w++)
        {
            if (wt[i - 1] <= w)
                dp[i][w] = max(dp[i - 1][w], val[i -
1] + dp[i - 1][w - wt[i - 1]]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }
    return dp[n][W];
}

int main()
{
    int W = 100;                        //
Capacity of the knapsack
    vector<int> val = {80, 24, 23, 22, 21}; //
Values of the items
    vector<int> wt = {80, 25, 25, 25, 25};  //
Weights of the items
    int n = val.size();                 //
Number of items

    cout << "Maximum value in knapsack: " <<
knapsack(W, wt, val, n) << endl;
    return 0;
}
```

6.6. LCS

```
// Function to compute the length of the Longest
Common Subsequence
int lcs(string X, string Y, int m, int n)
{
    vector<vector<int>> dp(m + 1, vector<int>(n + 1,
0)); // DP table

    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (X[i - 1] == Y[j - 1])
                dp[i][j] = 1 + dp[i - 1][j - 1];
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j
- 1]);
        }
    }
    return dp[m][n];
}

int main()
{
    string X = "AGGTAB", Y = "GXTXAYB";
    int m = X.length(), n = Y.length();

    cout << "Length of Longest Common Subsequence: "
<< lcs(X, Y, m, n) << endl;
    return 0;
}
```

6.7. LIS

```
// Function to find the length of the Longest
Increasing Subsequence
int lis(vector<int> &arr, int n)
{
    vector<int> dp(n, 1); // DP array, initialized
to 1

    for (int i = 1; i < n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (arr[i] > arr[j])
                dp[i] = max(dp[i], dp[j] + 1);
        }
    }
    return *max_element(dp.begin(), dp.end());
}

int main()
{
    vector<int> arr = {10, 22, 9, 33, 21, 50, 41,
60};
    int n = arr.size();

    cout << "Length of Longest Increasing
Subsequence: " << lis(arr, n) << endl;
    return 0;
}
```

6.8. LPS

```
// Function to compute the length of the Longest
Palindromic Subsequence
int lps(string s)
{
    int n = s.length();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++)
        dp[i][i] = 1; // Single character is a
palindrome

    for (int len = 2; len <= n; len++)
    {
        for (int i = 0; i < n - len + 1; i++)
        {
            int j = i + len - 1;
            if (s[i] == s[j])
                dp[i][j] = 2 + dp[i + 1][j - 1];
            else
                dp[i][j] = max(dp[i + 1][j], dp[i][j
- 1]);
        }
    }
    return dp[0][n - 1];
}

int main()
{
    string s = "bbabcbcab";
    cout << "Length of Longest Palindromic
Subsequence: " << lps(s) << endl;
    return 0;
}
```

6.9. MCM

```
// Function to compute the minimum number of scalar
multiplications
int matrixChainMultiplication(vector<int> &dims, int
n) {
    vector<vector<int>> dp(n, vector<int>(n, 0)); //
DP table
    for (int len = 2; len < n; len++)
    {
        for (int i = 1; i < n - len + 1; i++)
        {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            for (int k = i; k < j; k++)
            {
                int q = dp[i][k] + dp[k + 1][j] +
dims[i - 1] * dims[k] * dims[j];
                dp[i][j] = min(dp[i][j], q);
            }
        }
    }
    return dp[1][n - 1];
}

int main()
{
    vector<int> dims = {10, 20, 30, 40, 30};
    int n = dims.size();

    cout << "Minimum number of scalar
multiplications: " <<
matrixChainMultiplication(dims, n) << endl;
    return 0;
}
```

6.10. Minimum Coin Change

```
// Function to find the minimum number of coins
required to make a total
int minCoins(const vector<int> &coins, int total)
{
    int n = coins.size();
    vector<int> dp(total + 1, INT_MAX); //
Initialize DP array with infinity
    dp[0] = 0;                        // Base
case: 0 coins needed to make total 0

    // Fill DP array
    for (int i = 0; i < n; i++)
    {
        for (int j = coins[i]; j <= total; j++)
        {
            if (dp[j - coins[i]] != INT_MAX)
            {
                dp[j] = min(dp[j], dp[j - coins[i]]
+ 1); // Minimize coin count
            }
        }
    }

    return dp[total] == INT_MAX ? -1 : dp[total]; //
Return -1 if not possible
}

int main()
{
    vector<int> coins = {1, 2, 5};          //
Example denominations
    int total = 11;                         //
Target total
    cout << minCoins(coins, total) << endl; //
Output the result
    return 0;
}
```

6.11. optimal BST

```
// Function to calculate the minimum search cost for
an optimal BST
int optimalBST(const vector<int> &freq)
{
    int n = freq.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // Fill the DP table for subarrays of increasing
length
    for (int len = 1; len <= n; len++)
    { // len is the range length
        for (int i = 0; i <= n - len; i++)
        {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            int sum = 0;
            // Calculate sum of frequencies from i
to j
            for (int k = i; k <= j; k++)
            {
                sum += freq[k];
            }
            // Try each k as the root and calculate
the minimum cost
            for (int k = i; k <= j; k++)
            {
                int cost = (k == i ? 0 : dp[i][k -
1]) + (k == j ? 0 : dp[k + 1][j]);
                dp[i][j] = min(dp[i][j], cost +
sum);
            }
        }
    }
    return dp[0][n - 1]; // The minimum cost for the
entire range
}

int main()
{
    vector<int> freq = {34, 8, 50, 13}; // Example
frequencies of keys
    cout << optimalBST(freq) << endl;   // Output
the minimum cost
    return 0;
}
```

6.12. partition problem

```
// Function to determine if a given set can be
partitioned into two subsets
bool canPartition(vector<int> &nums)
{
    int sum = 0;
    for (int num : nums)
        sum += num;

    if (sum % 2 != 0)
        return false;

    int target = sum / 2;
    vector<bool> dp(target + 1, false);
    dp[0] = true;

    for (int num : nums)
    {
        for (int j = target; j >= num; j--)
        {
            dp[j] = dp[j] || dp[j - num];
        }
    }
    return dp[target];
}

int main()
{
    vector<int> nums = {1, 5, 11, 5};
    cout << "Can partition: " << (canPartition(nums)
? "Yes" : "No") << endl;
    return 0;
}
```

6.13. Regular Expression Matching

```
bool isMatch(const string &s, const string &p) {
    int m = s.size(), n = p.size();

    // DP table dp[i][j] will be true if s[0...i-1]
matches p[0...j-1]
    vector<vector<bool>> dp(m + 1, vector<bool>(n +
1, false));

    // Base case: empty string matches empty pattern
    dp[0][0] = true;

    // Handle patterns like "a*" or ".*" where "*"
can match 0 occurrence
    for (int j = 1; j <= n; j++)
    {
        if (p[j - 1] == '*')
        {
            dp[0][j] = dp[0][j - 2];
        }
    }

    // Fill the dp table
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (p[j - 1] == s[i - 1] || p[j - 1] ==
'.')
            {
                dp[i][j] = dp[i - 1][j - 1]; //
Character matches
            }
            else if (p[j - 1] == '*')
            {
                // '*' matches zero occurrence or
one/more occurrences of the preceding character
```

```
                dp[i][j] = dp[i][j - 2] || (dp[i -
1][j] && (s[i - 1] == p[j - 2] || p[j - 2] == '.'));
            }
        }
    }

    return dp[m][n]; // Final answer whether the
whole string matches the pattern
}

int main()
{
    string s = "aab";
    string p = "c*a*b";

    if (isMatch(s, p))
    {
        cout << "The string matches the pattern." <<
endl;
    }
    else
    {
        cout << "The string does not match the
pattern." << endl;
    }

    return 0;
}
```

6.14. ROD cutting

```
// Function to compute the maximum profit from
cutting a rod
int rodCutting(vector<int> &prices, int n)
{
    vector<int> dp(n + 1, 0); // DP array

    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= i; j++)
        {
            dp[i] = max(dp[i], prices[j - 1] + dp[i
- j]);
        }
    }
    return dp[n];
}

int main()
{
    vector<int> prices = {1, 5, 8, 9, 10, 17, 17,
20};
    int n = prices.size();

    cout << "Maximum profit from rod cutting: " <<
rodCutting(prices, n) << endl;
    return 0;
}
```

6.15. subset sum

```
// Function to determine if there's a subset with
sum equal to the target
bool subsetSum(vector<int> &nums, int sum)
{
    int n = nums.size();
    vector<vector<bool>> dp(n + 1, vector<bool>(sum
+ 1, false));

    for (int i = 0; i <= n; i++)
        dp[i][0] = true;

    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= sum; j++)
        {
            if (nums[i - 1] <= j)
                dp[i][j] = dp[i - 1][j] || dp[i -
1][j - nums[i - 1]];
            else
                dp[i][j] = dp[i - 1][j];
        }
    }
    return dp[n][sum];
}

int main()
{
    vector<int> nums = {3, 34, 4, 12, 5, 2};
    int sum = 9;

    cout << "Subset sum possible: " <<
(subsetSum(nums, sum) ? "Yes" : "No") << endl;
    return 0;
}
```

6.16. two player game

```
// Function to find the maximum sum player A can get
int maxCoins(const vector<int> &coins)
{
    int n = coins.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // Base case: when there's only one coin, player
A takes it
    for (int i = 0; i < n; i++)
    {
        dp[i][i] = coins[i];
    }

    // Fill DP table for subarrays of length 2 to n
    for (int len = 2; len <= n; len++)
    {
        for (int i = 0; i < n - len + 1; i++)
        {
            int j = i + len - 1;
            dp[i][j] = max(coins[i] + min(dp[i +
2][j], dp[i + 1][j - 1]),
                          coins[j] + min(dp[i +
1][j - 1], dp[i][j - 2]));
        }
    }

    return dp[0][n - 1]; // Maximum sum player A can
get
}

int main()
{
    vector<int> coins = {8, 15, 3, 7}; // Example
coins array
    cout << maxCoins(coins) << endl;   // Output the
result
    return 0;
}
```

6.17. word break

```
// Function to check if a word can be segmented into
words from a dictionary
bool wordBreak(string s, unordered_set<string>
&wordDict)
{
    int n = s.length();
    vector<bool> dp(n + 1, false);
    dp[0] = true;

    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (dp[j] && wordDict.find(s.substr(j, i
- j)) != wordDict.end())
            {
                dp[i] = true;
                break;
            }
        }
    }
    return dp[n];
}

int main()
{
    string s = "leetcode";
    unordered_set<string> wordDict = {"leet",
"code"};
    cout << "Can break: " << (wordBreak(s, wordDict)
? "Yes" : "No") << endl;
    return 0;
}
```

6.18. dp cheatsheet

## 1. Fibonacci Sequence (Memoization)

- Explanation: This algorithm calculates the nth Fibonacci number using memoization to store previously computed results, avoiding redundant calculations.

- When to Use: When you need to compute Fibonacci numbers for large values of `n` efficiently (i.e., for recursive problems that involve overlapping subproblems).

## 2. 0/1 Knapsack Problem

- Explanation: Given a set of items with weights and values, the goal is to determine the maximum value that can be obtained by putting items in a knapsack without exceeding the weight capacity.

- When to Use: In optimization problems where you need to maximize profit or value while respecting constraints (e.g., weight, space).

## 3. Minimum Coin Change

- Explanation: This algorithm calculates the minimum number of coins needed to make a given total using a set of coin denominations. It uses dynamic programming to store the results for all possible totals.

- When to Use: When you need to find the fewest coins needed to form a specific amount (e.g., for making change, budget optimization).

## 4. Longest Common Subsequence (LCS)

- Explanation: This algorithm finds the longest subsequence common to two sequences (strings, arrays, etc.). It uses a dynamic programming table to store intermediate results.

- When to Use: When comparing two sequences (e.g., DNA sequences, text comparison, diff tools) and need the longest subsequence they share.

## 5. Edit Distance (Levenshtein Distance)

- Explanation: This algorithm calculates the minimum number of operations (insertions, deletions, substitutions) needed to convert one string into another.

- When to Use: When you need to compare two strings and find the minimum edit operations required (e.g., spell checkers, natural language processing).

## 6. Longest Increasing Subsequence (LIS)

- Explanation: This algorithm finds the length of the longest increasing subsequence in a sequence of numbers. The subsequence need not be contiguous.

- When to Use: When you need to find the longest increasing subsequence in a sequence of numbers (e.g., stock price prediction, finding trends).

7. Matrix Chain Multiplication

- Explanation: This algorithm calculates the most efficient way to multiply a chain of matrices by minimizing the number of scalar multiplications.

- When to Use: When multiplying multiple matrices and you need to minimize the cost of multiplication (e.g., in computer graphics, optimization problems).

8. Rod Cutting Problem

- Explanation: This algorithm finds the maximum profit you can obtain by cutting a rod of length `n` into smaller pieces and selling them, based on the prices for each length.

- When to Use: When you need to solve problems related to cutting materials into pieces to maximize profit (e.g., resource allocation, profit optimization).

## 9. Subset Sum Problem

- Explanation: This algorithm checks if there is a subset of a given set of numbers that adds up to a target sum. It uses dynamic programming to keep track of achievable sums.

- When to Use: When you need to check whether a subset exists with a given sum (e.g., partitioning problems, subset analysis).

## 10. Egg Dropping Problem

- Explanation: This algorithm determines the minimum number of attempts required to find the highest floor from which an egg can be dropped without breaking, given `k` eggs and `n` floors.

- When to Use: In optimization problems where you need to minimize the number of trials in a worst-case scenario (e.g., testing, fault tolerance, hardware).

11. Partition Problem

- Explanation: This algorithm checks whether a given set can be partitioned into two subsets such that their sums are equal. It uses dynamic programming to check for subset sums.

- When to Use: When dividing a set of numbers into two equal subsets (e.g., load balancing, resource allocation).

## 12. Longest Palindromic Subsequence (LPS)

- Explanation: This algorithm finds the longest subsequence within a string that is a palindrome. It uses dynamic programming to build a table based on matching characters.

- When to Use: When you need to find the longest palindromic subsequence in a string (e.g., text processing, bioinformatics).

## 13. Word Break Problem

- Explanation: This algorithm checks if a string can be segmented into a space-separated sequence of words from a dictionary. It uses dynamic programming to store results for substrings.

- When to Use: When you need to determine whether a string can be split into valid words (e.g., for tokenizing sentences, text segmentation).

## 14. Regular Expression Matching

- Explanation: This algorithm checks if a string matches a pattern with . (any character) and * (zero or more of the previous character). It uses dynamic programming to track matching results for substrings.

- When to Use: When matching strings to patterns with wildcards like . and * (e.g., text matching, search engines, file pattern matching).

15. Optimal Binary Search Tree

- Explanation: This algorithm finds the minimum cost to construct a binary search tree based on the frequencies of elements. It uses dynamic programming to calculate the optimal cost for different subarrays.

- When to Use: When you need to minimize the cost of searching elements with different access frequencies (e.g., database indexing, search optimization).

## 16. 2 Player Game

- Explanation: This algorithm calculates the maximum sum player A can collect in a turn-based game where players alternate picking coins from either end of the array. It uses dynamic programming to determine the optimal strategy for player A.

- When to Use: When you need to calculate the optimal strategy in a turn-based game with alternating choices (e.g., maximizing outcomes in competitive games).

17. Counting Paths in Matrix

- Explanation: This algorithm calculates the number of unique paths from the top-left corner to the bottom-right corner of an n x m matrix. It uses dynamic programming to count paths by combining the results from adjacent cells.

- When to Use: When you need to find the number of distinct paths in a grid, where you can only move right or down (e.g., grid-based traversal problems).

7. More

7.1. longest polindrom substring (Mancher)

```
// O(n)

class Solution {
public:
    string longestPalindrome(std::string s) {
        if (s.length() <= 1) return s;

        // Preprocess the string with '#' characters
to handle even-length palindromes
        string modified_s = "#";
        for (char c : s) {
            modified_s += c;
            modified_s += '#';
        }

        int n = modified_s.size();
        vector<int> dp(n, 0); // dp array to store the radius of the palindrome centered at each character
        int center = 0, right = 0; // Initialize center and right boundary
        int max_len = 1; // Maximum length of palindrome found
        string max_str = s.substr(0, 1); // Initialize the max palindrome substring with the first character

        for (int i = 0; i < n; i++) {
            // If i is within the current right boundary, use previously calculated values to minimize comparisons
            if (i < right) {
                dp[i] = min(right - i, dp[2 * center - i]);
            }

            // Expand around center i
            while (i - dp[i] - 1 >= 0 && i + dp[i] + 1 < n && modified_s[i - dp[i] - 1] == modified_s[i + dp[i] + 1]) {
                dp[i]++;
            }

            // Update center and right boundary if we've expanded beyond the current right
            if (i + dp[i] > right) {
                center = i;
                right = i + dp[i];
            }

            // Update max_len and max_str if a longer palindrome is found
            if (dp[i] > max_len) {
                max_len = dp[i];
                max_str = modified_s.substr(i - dp[i], 2 * dp[i] + 1);
                max_str.erase(remove(max_str.begin(), max_str.end(), '#'), max_str.end());
            }
        }

        return max_str;
    }
};
```

7.2. median of 2 soted array

```
class Solution {
public:
    double findMedianSortedArrays(vector<int> &nums1, vector<int> &nums2) {
        int n = nums1.size();
        int m = nums2.size();
        if(n > m) return findMedianSortedArrays(nums2, nums1);
        int size = n + m;
        int left = (size + 1) / 2;
        int low = 0, high = n;
        int l1, l2, r1, r2, mid1, mid2;
        while(low <= high) {
            mid1 = (low + high) >> 1;
            mid2 = left - mid1;
            r1 = mid1 < n ? nums1[mid1] : 1e9;
            r2 = mid2 < m ? nums2[mid2] : 1e9;
            l1 = mid1 - 1 >= 0 ? nums1[mid1 - 1] : -1e9;
            l2 = mid2 - 1 >= 0 ? nums2[mid2 - 1] : -1e9;
            if(l1 <= r2 && l2 <= r1)
                if(size % 2) return max(l1, l2);
                else return (static_cast<double>(max(l1, l2) + min(r1, r2))) / 2.0;
            else if(l1 > r2) high = mid1 - 1;
            else low = mid1 + 1;
        }

        return 0;
    }
};
```

10. other

10.1. useful geo

Area of triangle with sides a, b, c: sqrt(S *(S-a)*(S-b)*(S-c)) where S = (a+b+c)/2

Area of equilateral triangle: s^2 * sqrt(3) / 4 where is side lenght

Pyramid and cones volume: 1/3 area(base) * height

if p1=(x1, x2), p2=(x2, y2), p3=(x3, y3) are points on circle, the center is

x = -((x2^2 - x1^2 + y2^2 - y1^2)*(y3 - y2) - (x2^2 - x3^2 + y2^2 - y3^2)*(y1 - y2)) / (2*(x1 - x2)*(y3 - y2) - 2*(x3 - x2)*(y1 - y2))

y = -((y2^2 - y1^2 + x2^2 - x1^2)*(x3 - x2) - (y2^2 - y3^3 + x2^2 - x3^2)*(x1 - x2)) / (2*(y1 - y2)*(x3 - x2) - 2*(y3 - y2)*(x1 - x2))

10.2. number of primes

30: 10
60: 17
100: 25
1000: 168
10000: 1229
100000: 9592
1000000: 78498

10000000: 664579

10.3. Factorials

1: 1
2: 2
3: 6
4: 24
5: 120
6: 720
7: 5040
8: 40320
9: 362880
10: 3628800
11: 39916800
12: 479001600
13: 6227020800
14: 87178291200
15: 1307674368000

10.4. power of 3

1: 3
2: 9
3: 27
4: 81
5: 243
6: 729
7: 2187
8: 6561
9: 19683
10: 59049
11: 177147
12: 531441
13: 1594323
14: 4782969
15: 14348907
16: 43046721
17: 129140163
18: 387420489
19: 1162261467
20: 3486784401

10.5. C(2n, n)

1: 2
2: 6
3: 20
4: 70
5: 252
6: 924
7: 3432
8: 12870
9: 48620
10: 184756
11: 705432
12: 2704156
13: 10400600
14: 40116600
15: 155117520

10.6. Most Divisor

<= 1e2: 60 with 12 divisors
<= 1e3: 840 with 32 divisors
<= 1e4: 7560 with 64 divisors
<= 1e5: 83160 with 128 divisors
<= 1e6: 720720 with 240 divisors
<= 1e7: 8648640 with 448 divisors
<= 1e8: 73513440 with 768 divisors
<= 1e9: 735134400 with 1344 divisors
<= 1e10: 6983776800 with 2304 divisors
<= 1e11: 97772875200 with 4032 divisors
<= 1e12: 963761198400 with 6720 divisors
<= 1e13: 9316358251200 with 10752 divisors
<= 1e14: 97821761637600 with 17280 divisors
<= 1e15: 866421317361600 with 26880 divisors
<= 1e16: 8086598962041600 with 41472 divisors
<= 1e17: 74801040398884800 with 64512 divisors
<= 1e18: 897612484786617600 with 103680 divisors

Useful formulas

$\genfrac{}{}{0pt}{}{n}{k} = \frac{n!}{k!(n\ \ k)!}$ — number of ways to choose k objects out of n

$\genfrac{}{}{0pt}{}{n+k\ \ 1}{k\ \ 1}$ — number of ways to choose k objects out of n with repetitions

$\genfrac{}{}{0pt}{}{n}{m}$ — Stirling numbers of the first kind; number of permutations of n elements with k cycles

$\genfrac{}{}{0pt}{}{n+1}{m} = n \genfrac{}{}{0pt}{}{n}{m} + \genfrac{}{}{0pt}{}{n}{m\ \ 1}$

$(x)_n = x(x \quad 1) \quad x \quad n+1 = \overset{n}{\underset{k=0}{}} (\quad 1)^{n \quad k} \genfrac{}{}{0pt}{}{n}{k} x^k$

$\genfrac{}{}{0pt}{}{n}{m}$ — Stirling numbers of the second kind; number of partitions of set $1 \quad n$ into k disjoint subsets.

$\genfrac{}{}{0pt}{}{n+1}{m} = k \genfrac{}{}{0pt}{}{n}{k} + \genfrac{}{}{0pt}{}{n}{k\ \ 1}$

$\overset{n}{\underset{k=0}{}} \genfrac{}{}{0pt}{}{n}{k} (x)_k = x^n$

$C_n = \frac{1}{n+1} \genfrac{}{}{0pt}{}{2n}{n}$ — Catalan numbers

$C(x) = \frac{1 \quad \overline{1 \quad 4x}}{2x}$

Binomial transform

If $a_n = \overset{n}{\underset{k=0}{}} \genfrac{}{}{0pt}{}{n}{k} b_k$, then $b_n = \overset{n}{\underset{k=0}{}} (\quad 1)^{n \quad k} \genfrac{}{}{0pt}{}{n}{k} a_k$

$a = (1 \ x \ x^2 \quad) \ b = (1 \ (x+1) \ (x+1)^2 \quad)$

$a_i = i^k \ b_i = \genfrac{}{}{0pt}{}{n}{i} \ i!$

Burnside's lemma

Let G be a group of *action* on set X (Ex.: cyclic shifts of array, rotations and symmetries of $n \quad n$ matrix, ...)

Call two objects x and y *equivalent* if there is an action f that transforms x to y: $f(x) = y$.

The number of equivalence classes then can be calculated as follows: $C = \frac{1}{G} \underset{f \ G}{} X^f$, where X^f is the set of *fixed points* of f: $X^f = \ x \ f(x) = x$

Generating functions

Ordinary generating function (o.g.f.) for sequence $a_0 \ a_1 \quad a_n \quad$ is $A(x) = \underset{n=0}{} a_i x^i$

Exponential generating function (e.g.f.) for sequence $a_0 \ a_1 \quad a_n \quad$ is $A(x) = \underset{n=0}{} a_i x^i$

$B(x) = A\ (x) \ b_{n \ \ 1} = n \quad a_n$

$c_n = \overset{n}{\underset{k=0}{}} a_k b_{n \ \ k}$ (o.g.f. convolution)

$c_n = \overset{n}{\underset{k=0}{}} \genfrac{}{}{0pt}{}{n}{k} a_k b_{n \ \ k}$ (e.g.f. convolution, compute with FFT using $a_n = \frac{a_n}{n!}$)

General linear recurrences

If $a_n = \overset{n}{\underset{k=1}{}} b_k a_{n \ \ k}$, then $A(x) = \frac{a_0}{1 \ \ B(x)}$. We also can compute all a_n with Divide-and-Conquer algorithm in $O(n \log^2 n)$.

Inverse polynomial modulo x^l

Given $A(x)$, find $B(x)$ such that $A(x)B(x) = 1 + x^l \quad Q(x)$ for some $Q(x)$

1. Start with $B_0(x) = \frac{1}{a_0}$
2. Double the length of $B(x)$: $B_{k+1}(x) = (\quad B_k(x)^2 A(x) + 2B_k(x)) \mod x^{2^{k+1}}$

Fast subset convolution

Given array a_i of size 2^k, calculate $b_i = \underset{j \& i = i}{} b_j$

```
for b = 0..k-1
  for i = 0..2^k-1
    if (i & (1 << b)) != 0:
      a[i + (1 << b)] += a[i]
```

Hadamard transform

Treat array a of size 2^k as k-dimentional array of size $2 \quad 2 \qquad 2$, calculate FFT of that array:

```
for b = 0..k-1
  for i = 0..2^k-1
    if (i & (1 << b)) != 0:
      u = a[i], v = a[i + (1 << b)]
      a[i] = u + v
      a[i + (1 << b)] = u - v
```