Competitive Programming Reference Sheet

Based on Antti Laaksonen's

Competitive Programmer's Handbook

Working with numbers

C++ integer types: `int` (32-bit), `long long` (64-bit). Use `__int128` for larger numbers (non-portable).

Floating-point: use `double`, `long double` if higher precision needed. Always compare doubles with tolerance ϵ.

Snippet: Modular arithmetic + floating comparison

```
// COMPILES: g++ -std=gnu++17 numbers.cpp -O2
#include <bits/stdc++.h>
using namespace std;

// modular multiplication (a*b mod m)
long long modmul(long long a, long long b, long long m) {
    return (a % m) * (b % m) % m;
}

// factorial mod m
long long factmod(int n, long long m) {
    long long res = 1;
    for (int i = 2; i <= n; i++) res = (res * i) % m;
    return res;
}

int main() {
    long long a = 123456789, b = 987654321, m = 1e9+7;
    cout << modmul(a, b, m) << "\n"; // 259106859

    double x = 0.3*3+0.1;
    if (fabs(x - 1.0) < 1e-9) cout << "Equal\n";
}
```

Pitfalls: - `int` overflow when squaring large values. - Comparing doubles with ==.

Shortening code

Contest code should be concise.

- Use `typedef` or `using` for type aliases.
- Use macros carefully (watch for precedence errors).

Snippet: Aliases and macros

```
// COMPILES: g++ -std=gnu++17 shortcode.cpp -O2
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef vector<int> vi;
typedef pair<int,int> pi;

#define F first
#define S second
#define PB push_back
#define MP make_pair

int main() {
    vi v;
    v.PB(10);
    v.PB(20);
    cout << v[0] << " " << v[1] << "\n"; // 10 20

    pi p = MP(3,4);
    cout << p.F + p.S << "\n"; // 7
}
```

Pitfall: #define macros can cause subtle bugs. Prefer inline functions when possible.

Mathematics

Essential formulas:

- Arithmetic progression: $\frac{n(a+b)}{2}$
- Geometric progression: $\frac{bk-a}{k-1}$
- Harmonic sum $\leq \log_2 n + 1$
- Factorial $n!$, Fibonacci, logarithms

Snippet: Basic math utilities

```
// COMPILES: g++ -std=gnu++17 mathutils.cpp -O2
#include <bits/stdc++.h>
using namespace std;

long long arith_sum(long long a, long long b, long long n)
     {
    return n * (a + b) / 2;
}

long long gcdll(long long a, long long b) {
    return b == 0 ? a : gcdll(b, a % b);
}

long long fib(int n) {
    if (n <= 1) return n;
    long long a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}

int main() {
    cout << arith_sum(3, 15, 4) << "\n"; // 36
    cout << gcdll(48, 18) << "\n"; // 6
    cout << fib(10) << "\n"; // 55
}
```

Time complexity

Time complexity measures how running time grows with input size n. We use asymptotic notation (big-O, big-Θ, big-Ω) to capture the growth rate.

Example: - Iterating once through an array of size n: $O(n)$ - Nested loops: $O(n^2)$

Snippet: Measuring complexity patterns

```
// COMPILES: g++ -std=gnu++17 complexity.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n = 1000;

    // O(n)
    long long sum1 = 0;
    for (int i = 0; i < n; i++) sum1++;

    // O(n^2)
    long long sum2 = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            sum2++;

    cout << sum1 << " " << sum2 << "\n"; // 1000 1000000
}
```

—

Big-O notation

Big-O describes the *upper bound* of runtime growth. Formally: $f(n) = O(g(n))$ if $\exists c, n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$.

Common complexities:

- $O(1)$: constant (array access)
- $O(\log n)$: binary search
- $O(n)$: simple loop
- $O(n \log n)$: mergesort, heapsort
- $O(n^2)$: naive all-pairs

Snippet: Binary search ($O(\log n)$)

```
// COMPILES: g++ -std=gnu++17 binary_search.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int binary_search_vec(vector<int>& v, int x) {
    int l = 0, r = (int)v.size()-1;
    while (l <= r) {
        int m = (l+r)/2;
        if (v[m] == x) return m;
        if (v[m] < x) l = m+1;
        else r = m-1;
    }
    return -1;
}

int main() {
    vector<int> v = {1,3,5,7,9};
    cout << binary_search_vec(v, 7) << "\n"; // 3
}
```

Pitfall: works only on sorted arrays.

—

Estimating time complexity

Rules of thumb: - 10^8 operations $\approx$ 1 second on modern CPU. - Always estimate: if $n = 10^5$, an $O(n^2)$ solution is infeasible.

Snippet: Estimating runtime growth

```
// COMPILES: g++ -std=gnu++17 estimate.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n = 1e5;
    long long ops = 1LL * n * n; // n^2
    cout << ops << "\n"; // 1e10 -> too large
}
```

Use this reasoning to reject naive solutions quickly.

—

Typical time complexities

Common scenarios:

- $n \leq 10^3$: $O(n^3)$ possible
- $n \leq 10^5$: $O(n \log n)$ possible
- $n \leq 10^6$: $O(n)$ possible
- $n \leq 10^9$: $O(\log n)$ or $O(1)$ needed

Snippet: Sorting ($O(n \log n)$)

```
// COMPILES: g++ -std=gnu++17 sortdemo.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> v = {5,2,8,1,3};
    sort(v.begin(), v.end()); // O(n log n)
    for (int x : v) cout << x << " "; // 1 2 3 5 8
}
```

Pitfall: Avoid $O(n^2)$ sorts (like bubble sort) for $n > 10^4$.

Logarithmic time

Logarithmic time arises when each step halves the problem size, such as binary search. Typical complexities: $O(\log n)$, $O(\log^2 n)$.

Snippet: Fast exponentiation (binary exponentiation)

```
// COMPILES: g++ -std=gnu++17 binexp.cpp -O2
#include <bits/stdc++.h>
using namespace std;

// computes a^b mod m in O(log b)
long long modpow(long long a, long long b, long long m) {
    long long res = 1 % m;
    while (b > 0) {
        if (b & 1) res = (res * a) % m;
        a = (a * a) % m;
        b >>= 1;
    }
```

```
    return res;
}

int main() {
    cout << modpow(2, 10, 1e9+7) << "\n"; // 1024
}
```

Pitfalls: - Watch out for overflow if not taking mod. - Negative exponents require modular inverse.

—

Recursion

Recursion solves a problem by reducing it to smaller instances of itself. It is natural for divide-and-conquer, tree traversals, backtracking.

Snippet: Depth-first search on a graph

```
// COMPILES: g++ -std=gnu++17 dfs.cpp -O2
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> adj;
vector<bool> visited;

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

int main() {
    int n = 4;
    adj.assign(n, {});
    adj[0] = {1,2};
    adj[1] = {2};
    adj[2] = {0,3};
    adj[3] = {};
    visited.assign(n, false);

    dfs(2);
    for (int i = 0; i < n; i++) cout << visited[i] << " ";
    // output: 1 1 1 1
}
```

Binary search applications

Binary search is not just for arrays, but any monotonic predicate:

$$Find smallest x such that condition(x) is true.$$

Snippet: Binary search on answer

```
// COMPILES: g++ -std=gnu++17 binsrch_ans.cpp -O2
#include <bits/stdc++.h>
using namespace std;

// Find smallest x with x^2 >= target
bool ok(long long x, long long target) {
    return x*x >= target;
}

long long binary_search_answer(long long target) {
    long long l = 0, r = 1e9, ans = -1;
    while (l <= r) {
        long long m = (l+r)/2;
        if (ok(m, target)) {
            ans = m;
            r = m-1;
        } else l = m+1;
    }
    return ans;
```

```
}

int main() {
    cout << binary_search_answer(30) << "\n"; // 6
}
```

Pitfall: Always ensure monotonicity of predicate.

—

Pitfalls: - Stack overflow for deep recursion (consider iterative approach or tail recursion). - Always mark visited to avoid infinite recursion.

—

Master theorem

The Master Theorem analyzes recurrence relations of divide-and-conquer. General recurrence: $T(n) = aT(n/b) + f(n)$

- If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
- If $f(n) = \Omega(n^{\log_b a+\epsilon})$ and regularity holds, then $T(n) = \Theta(f(n))$

Snippet: Merge sort recurrence example

```
// COMPILES: g++ -std=gnu++17 mergesort.cpp -O2
#include <bits/stdc++.h>
using namespace std;

void merge_sort(vector<int>& v) {
    if (v.size() <= 1) return;
    int mid = v.size()/2;
    vector<int> left(v.begin(), v.begin()+mid);
    vector<int> right(v.begin()+mid, v.end());
    merge_sort(left);
    merge_sort(right);
    merge(v.begin(), v.end(), left.begin(), left.end(),
          right.begin(), right.end());
}

int main() {
    vector<int> v = {5,2,8,1,3};
    merge_sort(v);
    for (int x : v) cout << x << " "; // 1 2 3 5 8
}
```

Here $T(n) = 2T(n/2) + O(n)$, so $T(n) = O(n \log n)$.

Two pointers

Two indices move across an array to maintain a condition. Typical for finding subarrays, merging, and problems like "count pairs with sum $<$ k".

Snippet: Counting pairs with sum $\leq k$

```
// COMPILES: g++ -std=gnu++17 twoptrs.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int count_pairs(vector<int>& v, int k) {
    sort(v.begin(), v.end());
    int l = 0, r = (int)v.size()-1, cnt = 0;
    while (l < r) {
```

```
        if (v[l] + v[r] <= k) {
            cnt += r - l;
            l++;
        } else r--;
    }
    return cnt;
}

int main() {
    vector<int> v = {1,2,3,4,5};
    cout << count_pairs(v, 5) << "\n"; // 6
}
```

Pitfall: Watch for double-counting pairs; move pointers carefully.

—

Sliding window

The sliding window technique maintains a subarray with given constraints. Useful for sums, distinct elements, max/min in window.

Snippet: Longest subarray with sum $\leq k$

```
// COMPILES: g++ -std=gnu++17 sliding.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int longest_subarray(vector<int>& v, int k) {
    int n = v.size(), l = 0, sum = 0, best = 0;
    for (int r = 0; r < n; r++) {
        sum += v[r];
        while (sum > k) {
            sum -= v[l++];
        }
        best = max(best, r - l + 1);
    }
    return best;
}

int main() {
    vector<int> v = {2,1,3,2,4};
    cout << longest_subarray(v, 5) << "\n"; // 2
}
```

Pitfall: Always shrink the window when constraints are broken.

Sorting basics

Sorting is fundamental. Standard C++ `sort` is $O(n \log n)$ introsort (quicksort + heapsort).

Snippet: Using sort with custom comparator

```
// COMPILES: g++ -std=gnu++17 sort_custom.cpp -O2
#include <bits/stdc++.h>
using namespace std;

struct Point { int x, y; };

int main() {
    vector<Point> pts = {{1,3},{2,2},{3,1}};
    sort(pts.begin(), pts.end(), [](auto a, auto b) {
        if (a.x != b.x) return a.x < b.x;
        return a.y < b.y;
    });
    for (auto p : pts) cout << "("<<p.x<<","<<p.y<<") ";
    // (1,3) (2,2) (3,1)
}
```

—

Meet-in-the-middle

When n is too large for $O(2^n)$ but small enough to split: compute two halves, then combine. Time $\approx O(2^{n/2})$.

Snippet: Subset sum with meet-in-the-middle

```
// COMPILES: g++ -std=gnu++17 mitm.cpp -O2
#include <bits/stdc++.h>
using namespace std;

// Check if subset of v sums to target
bool subset_sum(vector<int>& v, int target) {
    int n = v.size();
    int n1 = n/2;
    vector<int> A, B;

    // all subset sums of first half
    for (int mask = 0; mask < (1<<n1); mask++) {
        long long s = 0;
        for (int i = 0; i < n1; i++)
            if (mask & (1<<i)) s += v[i];
        A.push_back(s);
    }

    // all subset sums of second half
    for (int mask = 0; mask < (1<<(n-n1)); mask++) {
        long long s = 0;
        for (int i = 0; i < n-n1; i++)
            if (mask & (1<<i)) s += v[n1+i];
        B.push_back(s);
    }

    sort(B.begin(), B.end());
    for (long long x : A) {
        if (binary_search(B.begin(), B.end(), target - x))
            return true;
    }
    return false;
}

int main() {
    vector<int> v = {3, 34, 4, 12, 5, 2};
    cout << (subset_sum(v, 9) ? "YES" : "NO") << "\n"; //
     YES
}
```

Pitfalls: - Works only up to $n \approx 40$. - Requires careful split to balance both halves.

—

Quicksort

Divide-and-conquer: pick pivot, partition, recurse. Average: $O(n \log n)$; worst: $O(n^2)$.

Snippet: Quicksort implementation

```
// COMPILES: g++ -std=gnu++17 quicksort.cpp -O2
#include <bits/stdc++.h>
using namespace std;

void quicksort(vector<int>& v, int l, int r) {
    if (l >= r) return;
    int pivot = v[(l+r)/2];
    int i = l, j = r;
    while (i <= j) {
        while (v[i] < pivot) i++;
        while (v[j] > pivot) j--;
        if (i <= j) swap(v[i++], v[j--]);
    }
```

```
    quicksort(v, l, j);
    quicksort(v, i, r);
}

int main() {
    vector<int> v = {5,3,8,4,2};
    quicksort(v, 0, v.size()-1);
    for (int x: v) cout << x << " "; // 2 3 4 5 8
}
```

Pitfall: Poor pivot selection degrades to $O(n^2)$.

—

Mergesort

Divide array in half, recursively sort, then merge. Always $O(n \log n)$ time, $O(n)$ extra memory.

Snippet: Mergesort

```
// COMPILES: g++ -std=gnu++17 mergesort2.cpp -O2
#include <bits/stdc++.h>
using namespace std;

void merge_sort(vector<int>& v) {
    if (v.size() <= 1) return;
    int mid = v.size()/2;
    vector<int> left(v.begin(), v.begin()+mid);
    vector<int> right(v.begin()+mid, v.end());
    merge_sort(left);
    merge_sort(right);
    merge(v.begin(), v.end(), left.begin(), left.end(),
          right.begin(), right.end());
}

int main() {
    vector<int> v = {9,7,5,3,1};
    merge_sort(v);
    for (int x: v) cout << x << " "; // 1 3 5 7 9
}
```

Pitfall: Requires extra memory; not in-place.

—

Counting sort

Counting sort works when values lie in small integer range. Time: $O(n + k)$, where k is max value. Stable if implemented carefully.

Snippet: Counting sort

```
// COMPILES: g++ -std=gnu++17 countsort.cpp -O2
#include <bits/stdc++.h>
using namespace std;

void counting_sort(vector<int>& v, int maxval) {
    vector<int> cnt(maxval+1);
    for (int x: v) cnt[x]++;
    v.clear();
    for (int i = 0; i <= maxval; i++)
        while (cnt[i]--) v.push_back(i);
}

int main() {
    vector<int> v = {4,2,2,8,3,3,1};
    counting_sort(v, 8);
    for (int x: v) cout << x << " "; // 1 2 2 3 3 4 8
}
```

Pitfall: Only works with small range integers.

—

Radix sort

Radix sort processes digits from least to most significant, using stable sort (like counting sort). Time: $O(d(n+k))$, $d =$ digits.

Snippet: Radix sort (base 10)

```
// COMPILES: g++ -std=gnu++17 radix.cpp -O2
#include <bits/stdc++.h>
using namespace std;

void radix_sort(vector<int>& v) {
    int maxv = *max_element(v.begin(), v.end());
    for (int exp = 1; maxv/exp > 0; exp *= 10) {
        vector<int> output(v.size());
        int cnt[10] = {0};
        for (int x: v) cnt[(x/exp)%10]++;
        for (int i=1;i<10;i++) cnt[i]+=cnt[i-1];
        for (int i=v.size()-1;i>=0;i--) {
            int d = (v[i]/exp)%10;
            output[--cnt[d]] = v[i];
        }
        v = output;
    }
}

int main() {
    vector<int> v = {170,45,75,90,802,24,2,66};
    radix_sort(v);
    for (int x: v) cout << x << " ";
    // 2 24 45 66 75 90 170 802
}
```

—

Bucket sort

Bucket sort distributes numbers into buckets and sorts each, then concatenates. Best when data is uniformly distributed.

Snippet: Bucket sort on [0,1)

```
// COMPILES: g++ -std=gnu++17 bucket.cpp -O2
#include <bits/stdc++.h>
using namespace std;

void bucket_sort(vector<double>& v) {
    int n = v.size();
    vector<vector<double>> B(n);
    for (double x: v) {
        int idx = n * x;
        B[idx].push_back(x);
    }
    for (int i=0;i<n;i++) sort(B[i].begin(), B[i].end());
    v.clear();
    for (int i=0;i<n;i++)
        for (double x: B[i]) v.push_back(x);
}

int main() {
    vector<double> v =
     {0.78,0.17,0.39,0.26,0.72,0.94,0.21,0.12,0.23,0.68};
    bucket_sort(v);
    for (double x: v) cout << x << " ";
}
```

Pitfall: Only efficient when distribution is fairly uniform.

Arrays and vectors

An array has fixed size, but `std::vector` is dynamic. Random access in $O(1)$; insertion at end amortized $O(1)$.

Snippet: Vector usage

```
// COMPILES: g++ -std=gnu++17 vectordemo.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> v = {1,2,3};
    v.push_back(4); // add at end
    v[0] = 10;      // direct access
    for (int x: v) cout << x << " ";
    // 10 2 3 4
}
```

—

Sets

`std::set` is an ordered balanced BST (logarithmic operations). `std::unordered_set` is a hash set ($O(1)$ average).

Snippet: Set operations

```
// COMPILES: g++ -std=gnu++17 setdemo.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    set<int> s;
    s.insert(5);
    s.insert(1);
    s.insert(5); // ignored
    cout << s.count(5) << "\n"; // 1
    s.erase(1);
    for (int x: s) cout << x << " "; // 5
}
```

—

Maps

`std::map` is an ordered associative container (balanced BST). `std::unordered_map` is hash-based.

Snippet: Frequency map

```
// COMPILES: g++ -std=gnu++17 mapdemo.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> a = {1,2,2,3,3,3};
    map<int,int> freq;
    for (int x: a) freq[x]++;
    for (auto [key,val] : freq)
        cout << key << "->" << val << "\n";
    // 1->1  2->2  3->3
}
```

—

Stacks

LIFO structure. Push and pop are $O(1)$.

Snippet: Stack demo

```
// COMPILES: g++ -std=gnu++17 stackdemo.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    stack<int> st;
    st.push(1); st.push(2); st.push(3);
    while (!st.empty()) {
        cout << st.top() << " ";
        st.pop();
    }
    // 3 2 1
}
```

Applications: expression evaluation, DFS iterative.

—

Queues

FIFO structure. Push back, pop front in $O(1)$.

Snippet: Queue demo

```
// COMPILES: g++ -std=gnu++17 queuedemo.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    queue<int> q;
    q.push(10); q.push(20); q.push(30);
    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    // 10 20 30
}
```

—

Deques

Double-ended queue: push/pop both ends in $O(1)$.

Snippet: Deque demo

```
// COMPILES: g++ -std=gnu++17 dequedemo.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    deque<int> d;
    d.push_back(1);
    d.push_front(2);
    d.push_back(3);
    for (int x: d) cout << x << " "; // 2 1 3
}
```

—

Priority queues

Heap-based structure for efficiently extracting max (or min). `std::priority_queue` is max-heap by default.

Snippet: Priority queue demo

```
// COMPILES: g++ -std=gnu++17 pqdemo.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    priority_queue<int> pq;
    pq.push(5); pq.push(1); pq.push(10);
    while (!pq.empty()) {
```

Bitset

`std::bitset<N>` stores fixed-size bits efficiently. Bitwise operations are $O(N/wordsize)$.

Snippet: Bitset usage

```
// COMPILES: g++ -std=gnu++17 bitsetdemo.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    bitset<8> b(string("10110011"));
    cout << b.count() << "\n";   // number of 1s -> 5
    cout << b[0] << " " << b[7]; // LSB=1, MSB=1
}
```

—

Fenwick tree (Binary Indexed Tree)

Supports prefix sums in $O(\log n)$, with $O(\log n)$ updates. Space $O(n)$.

Snippet: Fenwick tree

```
// COMPILES: g++ -std=gnu++17 fenwick.cpp -O2
#include <bits/stdc++.h>
using namespace std;

struct Fenwick {
    int n;
    vector<long long> bit;
    Fenwick(int n): n(n), bit(n+1,0) {}

    void add(int idx, long long val) {
        for (; idx <= n; idx += idx & -idx)
            bit[idx] += val;
    }
    long long sum(int idx) {
        long long r=0;
        for (; idx > 0; idx -= idx & -idx)
            r += bit[idx];
        return r;
    }
    long long range_sum(int l,int r){
        return sum(r)-sum(l-1);
    }
};

int main() {
    Fenwick ft(5);
    ft.add(1,2); ft.add(2,3); ft.add(3,5);
    cout << ft.range_sum(2,3) << "\n"; // 8
}
```

Pitfall: Indexing is usually 1-based.

—

```
        cout << pq.top() << " ";
        pq.pop();
    }
    // 10 5 1
}
```

Pitfall: For min-heap, use `priority_queue<int, vector<int>, greater<int»`.

Segment tree

Stores information about intervals, supports range queries and updates. Time: $O(\log n)$ per query/update. Space: $O(n)$.

Snippet: Segment tree (sum)

```
// COMPILES: g++ -std=gnu++17 segtree.cpp -O2
#include <bits/stdc++.h>
using namespace std;

struct SegTree {
    int n;
    vector<long long> tree;
    SegTree(int n): n(n), tree(4*n) {}

    void build(vector<int>& a, int v, int tl, int tr) {
        if (tl==tr) tree[v]=a[tl];
        else {
            int tm=(tl+tr)/2;
            build(a,v*2,tl,tm);
            build(a,v*2+1,tm+1,tr);
            tree[v]=tree[v*2]+tree[v*2+1];
        }
    }
    long long sum(int v,int tl,int tr,int l,int r){
        if (l>r) return 0;
        if (l==tl && r==tr) return tree[v];
        int tm=(tl+tr)/2;
        return sum(v*2,tl,tm,l,min(r,tm))
             + sum(v*2+1,tm+1,tr,max(l,tm+1),r);
    }
    void update(int v,int tl,int tr,int pos,int val){
        if (tl==tr) tree[v]=val;
        else {
            int tm=(tl+tr)/2;
            if (pos<=tm) update(v*2,tl,tm,pos,val);
            else update(v*2+1,tm+1,tr,pos,val);
            tree[v]=tree[v*2]+tree[v*2+1];
        }
    }
};

int main() {
    vector<int> a={1,2,3,4,5};
    SegTree st(a.size());
    st.build(a,1,0,a.size()-1);
    cout<<st.sum(1,0,a.size()-1,1,3)<<"\n"; // 2+3+4=9
}
```

—

Sparse table

Precomputes answers for intervals with overlap. Good for idempotent operations (min, gcd). Query: $O(1)$; build: $O(n \log n)$.

Snippet: RMQ with sparse table

```
// COMPILES: g++ -std=gnu++17 sparsetable.cpp -O2
#include <bits/stdc++.h>
using namespace std;
```

```cpp
struct SparseTable {
    int n, K;
    vector<vector<int>> st;
    vector<int> lg;
    SparseTable(vector<int>& a) {
        n=a.size(); K=__lg(n)+1;
        st.assign(K, vector<int>(n));
        st[0]=a;
        for (int k=1;k<K;k++)
            for (int i=0;i+(1<<k)<=n;i++)
                st[k][i]=min(st[k-1][i],
                             st[k-1][i+(1<<(k-1))]);
        lg.assign(n+1,0);
        for (int i=2;i<=n;i++) lg[i]=lg[i/2]+1;
    }
    int query(int l,int r){
        int j=lg[r-l+1];
        return min(st[j][l],st[j][r-(1<<j)+1]);
    }
};

int main(){
    vector<int> a={1,3,-2,8,5};
    SparseTable sp(a);
    cout<<sp.query(1,3)<<"\n"; // -2
}
```

—

Disjoint set union (Union-Find)

Maintains partition of elements into disjoint sets. With union by rank + path compression: almost $O(1)$ per op.

Snippet: DSU

```cpp
// COMPILES: g++ -std=gnu++17 dsu.cpp -O2
#include <bits/stdc++.h>
using namespace std;

struct DSU {
    vector<int> p, sz;
    DSU(int n): p(n), sz(n,1) {
        iota(p.begin(),p.end(),0);
    }
    int find(int x){
        if (p[x]==x) return x;
        return p[x]=find(p[x]);
    }
    bool unite(int a,int b){
        a=find(a); b=find(b);
        if (a==b) return false;
        if (sz[a]<sz[b]) swap(a,b);
        p[b]=a; sz[a]+=sz[b];
        return true;
    }
};

int main(){
    DSU d(5);
    d.unite(0,1);
    d.unite(3,4);
    cout<<(d.find(1)==d.find(0))<<"\n"; // 1
}
```

Applications: Kruskal's MST, connected components.

Breadth-first search (BFS)

BFS explores a graph layer by layer. Time: $O(n+m)$ where n=vertices, m=edges.

Snippet: BFS shortest path in unweighted graph

```cpp
// COMPILES: g++ -std=gnu++17 bfs.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n=6;
    vector<vector<int>> adj(n);
    adj[0]={1,2}; adj[1]={0,3,4};
    adj[2]={0,4}; adj[3]={1,5};
    adj[4]={1,2,5}; adj[5]={3,4};

    vector<int> dist(n,-1);
    queue<int> q;
    dist[0]=0; q.push(0);

    while(!q.empty()){
        int u=q.front(); q.pop();
        for(int v:adj[u])
            if(dist[v]==-1){
                dist[v]=dist[u]+1;
                q.push(v);
            }
    }
    for(int d:dist) cout<<d<<" ";
    // 0 1 1 2 2 3
}
```

—

Depth-first search (DFS)

DFS explores deeply before backtracking. Time: $O(n+m)$.

Snippet: Connected components with DFS

```cpp
// COMPILES: g++ -std=gnu++17 components.cpp -O2
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> adj;
vector<bool> vis;

void dfs(int u, vector<int>& comp){
    vis[u]=true; comp.push_back(u);
    for(int v:adj[u]) if(!vis[v]) dfs(v,comp);
}

int main(){
    int n=5;
    adj={{1},{0,2},{1},{4},{3}};
    vis.assign(n,false);
    vector<vector<int>> comps;
    for(int i=0;i<n;i++) if(!vis[i]){
        vector<int> c; dfs(i,c); comps.push_back(c);
    }
    for(auto& c:comps){
        for(int x:c) cout<<x<<" ";
        cout<<"\n"; // comp1: 0 1 2 , comp2: 3 4
    }
}
```

—

Topological sort

Applies to DAGs: orders nodes so all edges go forward. Time: $O(n+m)$.

Snippet: Kahn's algorithm

```
// COMPILES: g++ -std=gnu++17 toposort.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n=6;
    vector<vector<int>> adj(n);
    adj[5]={2,0}; adj[4]={0,1};
    adj[2]={3}; adj[3]={1};
    vector<int> indeg(n,0);
    for(int u=0;u<n;u++) for(int v:adj[u]) indeg[v]++;

    queue<int> q;
    for(int i=0;i<n;i++) if(!indeg[i]) q.push(i);

    vector<int> order;
    while(!q.empty()){
        int u=q.front(); q.pop();
        order.push_back(u);
        for(int v:adj[u]){
            if(--indeg[v]==0) q.push(v);
        }
    }
    for(int x:order) cout<<x<<" ";
    // One valid order: 4 5 2 3 1 0
}
```

—

Dijkstra's algorithm

Finds shortest paths in weighted graphs with nonnegative edges. Time: $O((n+m)\log n)$ with priority queue.

Snippet: Dijkstra

```
// COMPILES: g++ -std=gnu++17 dijkstra.cpp -O2
#include <bits/stdc++.h>
using namespace std;
using P=pair<int,int>;

int main(){
    int n=5;
    vector<vector<P>> adj(n);
    adj[0]={{1,2},{2,4}};
    adj[1]={{2,1},{3,7}};
    adj[2]={{4,3}};
    adj[3]={{4,1}};
    vector<int> dist(n,1e9);
    dist[0]=0;
    priority_queue<P,vector<P>,greater<P>> pq;
    pq.push({0,0});
    while(!pq.empty()){
        auto [d,u]=pq.top(); pq.pop();
        if(d!=dist[u]) continue;
        for(auto [v,w]:adj[u]){
            if(dist[v]>d+w){
                dist[v]=d+w;
                pq.push({dist[v],v});
            }
        }
    }
    for(int i=0;i<n;i++) cout<<dist[i]<<" ";
    // 0 2 3 9 6
}
```

Pitfall: Fails if edges have negative weights.

—

Bellman–Ford

Handles negative weights; detects negative cycles. Time: $O(nm)$.

Snippet: Bellman–Ford

```
// COMPILES: g++ -std=gnu++17 bellmanford.cpp -O2
#include <bits/stdc++.h>
using namespace std;
struct Edge{int u,v,w;};

int main(){
    int n=5;
    vector<Edge> edges={{0,1,6},{0,2,7},{1,2,8},{1,3,5},
                        {1,4,-4},{2,3,-3},{2,4,9},
                        {3,1,-2},{4,3,7}};
    vector<int> dist(n,1e9);
    dist[0]=0;
    for(int i=1;i<n;i++){
        for(auto e:edges){
            if(dist[e.u]<1e9)
                dist[e.v]=min(dist[e.v],dist[e.u]+e.w);
        }
    }
    // detect negative cycle
    bool negcycle=false;
    for(auto e:edges){
        if(dist[e.u]<1e9 && dist[e.u]+e.w<dist[e.v])
            negcycle=true;
    }
    cout<<(negcycle?"NEG CYCLE":"OK")<<"\n";
}
```

—

Floyd–Warshall

All-pairs shortest paths in $O(n^3)$. Works with negative edges (but no negative cycles).

Snippet: Floyd–Warshall

```
// COMPILES: g++ -std=gnu++17 floyd.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main(){
    const int INF=1e9;
    int n=4;
    vector<vector<int>> d(n, vector<int>(n,INF));
    for(int i=0;i<n;i++) d[i][i]=0;
    d[0][1]=5; d[0][3]=10;
    d[1][2]=3; d[2][3]=1;

    for(int k=0;k<n;k++)
      for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
          if(d[i][k]<INF && d[k][j]<INF)
            d[i][j]=min(d[i][j],d[i][k]+d[k][j]);

    for(auto& row:d){
        for(int x:row) cout<<x<<" ";
        cout<<"\n";
    }
}
```

—

Minimum spanning tree — Prim's algorithm

Prim's algorithm grows a tree from any start node. Time: $O((n+m)\log n)$.

Snippet: Prim

```
// COMPILES: g++ -std=gnu++17 prim.cpp -O2
#include <bits/stdc++.h>
using namespace std;
using P=pair<int,int>;

int main(){
    int n=4;
    vector<vector<P>> adj(n);
    adj[0]={{1,1},{2,4}};
    adj[1]={{0,1},{2,2},{3,6}};
    adj[2]={{0,4},{1,2},{3,3}};
    adj[3]={{1,6},{2,3}};

    vector<int> dist(n,1e9);
    vector<bool> inMST(n,false);
    dist[0]=0;
    priority_queue<P,vector<P>,greater<P>> pq;
    pq.push({0,0});
    int cost=0;

    while(!pq.empty()){
        auto [d,u]=pq.top(); pq.pop();
        if(inMST[u]) continue;
        inMST[u]=true;
        cost+=d;
        for(auto [v,w]:adj[u])
            if(!inMST[v] && w<dist[v]){
                dist[v]=w; pq.push({w,v});
            }
    }
    cout<<"MST cost="<<cost<<"\n"; // 6
}
```

—

Minimum spanning tree — Kruskal's algorithm

Sort edges and unite sets with DSU. Time: $O(m \log m)$.

Snippet: Kruskal

```
// COMPILES: g++ -std=gnu++17 kruskal.cpp -O2
#include <bits/stdc++.h>
using namespace std;

struct DSU{
    vector<int> p,sz;
    DSU(int n):p(n),sz(n,1){iota(p.begin(),p.end(),0);}
    int find(int x){return p[x]==x?x:p[x]=find(p[x]);}
    bool unite(int a,int b){
        a=find(a); b=find(b);
        if(a==b) return false;
        if(sz[a]<sz[b]) swap(a,b);
        p[b]=a; sz[a]+=sz[b]; return true;
    }
};

struct Edge{int u,v,w;};

int main(){
    int n=4;
    vector<Edge> edges
     ={{0,1,1},{0,2,4},{1,2,2},{2,3,3},{1,3,6}};
    sort(edges.begin(),edges.end(),[](auto&a,auto&b){
     return a.w<b.w;});
    DSU d(n);
    int cost=0;
    for(auto e:edges){
        if(d.unite(e.u,e.v)) cost+=e.w;
    }
    cout<<"MST cost="<<cost<<"\n"; // 6
}
```

—

Bipartite check

Graph is bipartite if we can 2-color without conflicts. Time: $O(n + m)$.

Snippet: Bipartite check with BFS

```
// COMPILES: g++ -std=gnu++17 bipartite.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n=4;
    vector<vector<int>> adj={{1,3},{0,2},{1,3},{0,2}};
    vector<int> color(n,-1);
    bool ok=true;
    for(int i=0;i<n;i++) if(color[i]==-1){
        queue<int> q; q.push(i); color[i]=0;
        while(!q.empty()){
            int u=q.front(); q.pop();
            for(int v:adj[u]){
                if(color[v]==-1){
                    color[v]=color[u]^1;
                    q.push(v);
                } else if(color[v]==color[u]) ok=false;
            }
        }
    }
    cout<<(ok?"BIPARTITE":"NOT")<<"\n"; // BIPARTITE
}
```

—

Maximum bipartite matching (augmenting paths)

Kuhn's algorithm: DFS-based matching in $O(nm)$. Efficient for $n \le 500$ or so.

Snippet: Maximum matching (Kuhn)

```
// COMPILES: g++ -std=gnu++17 kuhn.cpp -O2
#include <bits/stdc++.h>
using namespace std;

int n1=3,n2=3;
vector<vector<int>> g={{0,1},{1,2},{0,2}};
vector<int> mt;
vector<char> used;

bool try_kun(int v){
    if(used[v]) return false;
    used[v]=true;
    for(int to:g[v]){
        if(mt[to]==-1 || try_kun(mt[to])){
            mt[to]=v; return true;
        }
    }
    return false;
}

int main(){
    mt.assign(n2,-1);
    int match=0;
    for(int v=0;v<n1;v++){
        used.assign(n1,false);
        if(try_kun(v)) match++;
    }
    cout<<"Matching size="<<match<<"\n"; // 3
}
```

Pitfall: Use Hopcroft–Karp for $O(\sqrt{n}m)$ if n is larger.

—

Ford–Fulkerson method

Max flow = repeatedly augment flow along s–t paths. Generic form; worst-case exponential, but if implemented with BFS (Edmonds–Karp) or DFS carefully, is polynomial.

—

Edmonds–Karp

Ford–Fulkerson with BFS to find shortest augmenting paths. Time: $O(nm^2)$.

Snippet: Edmonds–Karp

```
// COMPILES: g++ -std=gnu++17 edmondskarp.cpp -O2
#include <bits/stdc++.h>
using namespace std;

struct Edge { int v, cap, rev; };

struct DinicLike { // simplified Edmonds-Karp
    int n;
    vector<vector<Edge>> g;
    DinicLike(int n): n(n), g(n) {}
    void add_edge(int u,int v,int c){
        g[u].push_back({v,c,(int)g[v].size()});
        g[v].push_back({u,0,(int)g[u].size()-1});
    }
    int bfs(int s,int t, vector<int>&p, vector<int>&pe){
        fill(p.begin(),p.end(),-1);
        queue<int> q; q.push(s); p[s]=s;
        while(!q.empty()){
            int u=q.front();q.pop();
            for(int i=0;i<(int)g[u].size();i++){
                Edge &e=g[u][i];
                if(p[e.v]==-1 && e.cap>0){
                    p[e.v]=u; pe[e.v]=i;
                    if(e.v==t) return 1;
                    q.push(e.v);
                }
            }
        }
        return 0;
    }
    int maxflow(int s,int t){
        int flow=0;
        vector<int> p(n),pe(n);
        while(bfs(s,t,p,pe)){
            int aug=1e9;
            for(int v=t;v!=s;v=p[v]){
                auto &e=g[p[v]][pe[v]];
                aug=min(aug,e.cap);
            }
            for(int v=t;v!=s;v=p[v]){
                auto &e=g[p[v]][pe[v]];
                e.cap-=aug;
                g[v][e.rev].cap+=aug;
            }
            flow+=aug;
        }
        return flow;
    }
};

int main(){
    DinicLike mf(4);
    mf.add_edge(0,1,3);
    mf.add_edge(0,2,2);
    mf.add_edge(1,2,1);
    mf.add_edge(1,3,2);
    mf.add_edge(2,3,4);
    cout<<mf.maxflow(0,3)<<"\n"; // 5
}
```

—

Dinic's algorithm

Layered graph with blocking flows. Time: $O(n^2m)$ worst-case, $O(\sqrt{n}m)$ on unit networks.

Snippet: Dinic

```
// COMPILES: g++ -std=gnu++17 dinic.cpp -O2
#include <bits/stdc++.h>
using namespace std;

struct Edge{int v,cap,rev;};

struct Dinic {
    int n;
    vector<vector<Edge>> g;
    vector<int> level, it;
    Dinic(int n): n(n), g(n), level(n), it(n) {}
    void add_edge(int u,int v,int c){
        g[u].push_back({v,c,(int)g[v].size()});
        g[v].push_back({u,0,(int)g[u].size()-1});
    }
    bool bfs(int s,int t){
        fill(level.begin(),level.end(),-1);
        queue<int> q; q.push(s); level[s]=0;
        while(!q.empty()){
            int u=q.front(); q.pop();
            for(auto &e:g[u])
                if(e.cap>0 && level[e.v]==-1){
                    level[e.v]=level[u]+1;
                    q.push(e.v);
                }
        }
        return level[t]!=-1;
    }
    int dfs(int u,int t,int f){
        if(u==t) return f;
        for(int &i=it[u];i<(int)g[u].size();i++){
            Edge &e=g[u][i];
            if(e.cap>0 && level[e.v]==level[u]+1){
                int ret=dfs(e.v,t,min(f,e.cap));
                if(ret){
                    e.cap-=ret;
                    g[e.v][e.rev].cap+=ret;
                    return ret;
                }
            }
        }
        return 0;
    }
    int maxflow(int s,int t){
        int flow=0,f;
        while(bfs(s,t)){
            fill(it.begin(),it.end(),0);
            while((f=dfs(s,t,1e9))) flow+=f;
        }
        return flow;
    }
};

int main(){
    Dinic mf(4);
    mf.add_edge(0,1,3);
    mf.add_edge(0,2,2);
    mf.add_edge(1,2,1);
    mf.add_edge(1,3,2);
    mf.add_edge(2,3,4);
    cout<<mf.maxflow(0,3)<<"\n"; // 5
}
```

—

Flow applications

1. Bipartite matching: Construct source–left, right–sink, capacity 1 edges.

2. Edge disjoint paths: Assign capacity 1 to edges,

Tree traversal basics

DFS/BFS on trees is same as graphs but with $n-1$ edges. Subtree sizes and parent arrays can be computed in $O(n)$.

Snippet: DFS collecting parent + subtree size

```
// COMPILES: g++ -std=gnu++17 treedfs.cpp -O2
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> adj;
vector<int> parent, sz;

void dfs(int u,int p){
    parent[u]=p; sz[u]=1;
    for(int v:adj[u]) if(v!=p){
        dfs(v,u);
        sz[u]+=sz[v];
    }
}

int main(){
    int n=5;
    adj={{1,2},{0,3,4},{0},{1},{1}};
    parent.assign(n,-1); sz.assign(n,0);
    dfs(0,-1);
    for(int i=0;i<n;i++)
        cout<<"node "<<i<<" parent="<<parent[i]
            <<" size="<<sz[i]<<"\n";
}
```

—

Tree diameter

Diameter = longest path in tree. Two-pass DFS/BFS finds it in $O(n)$.

Snippet: Diameter via two BFS

```
// COMPILES: g++ -std=gnu++17 diameter.cpp -O2
#include <bits/stdc++.h>
using namespace std;

pair<int,int> bfs_far(int s, vector<vector<int>>&adj){
    int n=adj.size();
    vector<int> d(n,-1); d[s]=0;
    queue<int> q; q.push(s);
    while(!q.empty()){
        int u=q.front(); q.pop();
        for(int v:adj[u]) if(d[v]==-1){
            d[v]=d[u]+1; q.push(v);
        }
    }
    int u=max_element(d.begin(),d.end())-d.begin();
    return {u,d[u]};
}

int main(){
    vector<vector<int>> adj={{1},{0,2,3},{1},{1,4},{3}};
    auto a=bfs_far(0,adj);
    auto b=bfs_far(a.first,adj);
    cout<<"diameter length="<<b.second<<"\n"; // 3
}
```

—

maxflow gives count.

3. Minimum cut: The cut corresponding to max flow gives smallest set of edges disconnecting s, t.

—

Lowest common ancestor (LCA) — Binary lifting

Preprocess parent table with $O(n \log n)$, queries in $O(\log n)$.

Snippet: LCA with binary lifting

```
// COMPILES: g++ -std=gnu++17 lca.cpp -O2
#include <bits/stdc++.h>
using namespace std;

const int LOG=20;
vector<vector<int>> adj;
vector<int> depth;
vector<vector<int>> up;

void dfs(int u,int p){
    up[u][0]=p;
    for(int i=1;i<LOG;i++)
        up[u][i]=(up[u][i-1]==-1?-1:up[up[u][i-1]][i-1]);
    for(int v:adj[u]) if(v!=p){
        depth[v]=depth[u]+1;
        dfs(v,u);
    }
}

int lca(int a,int b){
    if(depth[a]<depth[b]) swap(a,b);
    int k=depth[a]-depth[b];
    for(int i=LOG-1;i>=0;i--)
        if(k&(1<<i)) a=up[a][i];
    if(a==b) return a;
    for(int i=LOG-1;i>=0;i--)
        if(up[a][i]!=up[b][i]){
            a=up[a][i]; b=up[b][i];
        }
    return up[a][0];
}

int main(){
    int n=5;
    adj={{1,2},{0,3,4},{0},{1},{1}};
    depth.assign(n,0);
    up.assign(n,vector<int>(LOG,-1));
    dfs(0,-1);
    cout<<lca(3,4)<<"\n"; // 1
}
```

—

Centroid decomposition

Divide-and-conquer on tree by repeatedly removing centroid. Useful for distance queries, optimization on trees.

Snippet: Centroid decomposition skeleton

```
// COMPILES: g++ -std=gnu++17 centroid.cpp -O2
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> adj;
vector<int> sz,par;
vector<bool> dead;

int dfs_sz(int u,int p){
```

```
    sz[u]=1;
    for(int v:adj[u]) if(v!=p && !dead[v])
        sz[u]+=dfs_sz(v,u);
    return sz[u];
}

int find_centroid(int u,int p,int n){
    for(int v:adj[u]) if(v!=p && !dead[v]){
        if(sz[v]>n/2) return find_centroid(v,u,n);
    }
    return u;
}

void decompose(int u,int p){
    int n=dfs_sz(u,-1);
    int c=find_centroid(u,-1,n);
    par[c]=p; dead[c]=true;
    for(int v:adj[c]) if(!dead[v])
        decompose(v,c);
}

int main(){
    int n=5;
    adj={{1,2},{0,3,4},{0},{1},{1}};
    sz.assign(n,0); par.assign(n,-1); dead.assign(n,false)
     ;
    decompose(0,-1);
    for(int i=0;i<n;i++) cout<<"centroid parent "<<i<<"="
     <<par[i]<<"\n";
}
```

—

Heavy-Light Decomposition (HLD)

Decompose tree into heavy paths + light edges. Allows path queries in $O(\log^2 n)$ with segment tree or BIT.

Snippet: HLD skeleton with segment tree

```
// COMPILES: g++ -std=gnu++17 hld.cpp -O2
#include <bits/stdc++.h>
using namespace std;

const int N=100005,LOG=17;
vector<int> adj[N];
int parent[N],depth[N],heavy[N],head[N],pos[N],sz[N];
int curPos=0;

int dfs(int u){
    sz[u]=1; int mx=0;
    for(int v:adj[u]) if(v!=parent[u]){
        parent[v]=u; depth[v]=depth[u]+1;
        int s=dfs(v);
        sz[u]+=s;
        if(s>mx){mx=s; heavy[u]=v;}
    }
    return sz[u];
}

void decompose(int u,int h){
    head[u]=h; pos[u]=curPos++;
    if(heavy[u]!=-1) decompose(heavy[u],h);
    for(int v:adj[u]) if(v!=parent[u] && v!=heavy[u])
        decompose(v,v);
}

int main(){
    int n=5;
    adj[0]={1,2}; adj[1]={0,3,4}; adj[2]={0};
    memset(heavy,-1,sizeof heavy);
    dfs(0); decompose(0,0);
    for(int i=0;i<n;i++) cout<<i<<" pos="<<pos[i]<<" head=
     "<<head[i]<<"\n";
}
```

—

Euler tour technique

DFS entry/exit times flatten tree into array. Subtree queries → range queries.

Snippet: Euler tour

```
// COMPILES: g++ -std=gnu++17 euler.cpp -O2
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> adj;
vector<int> tin,tout,order;
int timer=0;

void dfs(int u,int p){
    tin[u]=timer++;
    order.push_back(u);
    for(int v:adj[u]) if(v!=p) dfs(v,u);
    tout[u]=timer-1;
}

int main(){
    int n=5;
    adj={{1,2},{0,3,4},{0},{1},{1}};
    tin.resize(n); tout.resize(n);
    dfs(0,-1);
    for(int i=0;i<n;i++)
        cout<<"node "<<i<<" range=["<<tin[i]<<","<<tout[i
     ]<<"]\n";
}
```

—

Game trees and Grundy numbers

Impartial combinatorial games: - Position → Grundy number (mex of child states). - Winning iff xor of Grundy numbers $\neq 0$.

Snippet: Grundy example (Nim-like piles)

```
// COMPILES: g++ -std=gnu++17 grundy.cpp -O2
#include <bits/stdc++.h>
using namespace std;

// mex of a set
int mex(const set<int>& s){
    int g=0;
    while(s.count(g)) g++;
    return g;
}

int main(){
    vector<int> piles={3,4,5};
    int x=0;
    for(int p:piles) x^=p; // nim heap = value itself
    cout<<(x?"FIRST":"SECOND")<<"\n"; // FIRST
}
```

—

Greatest common divisor (Euclid)

Snippet: GCD + LCM

```
// COMPILES: g++ -std=gnu++17 gcd.cpp -O2
#include <bits/stdc++.h>
using namespace std;
```

```
long long gcd(long long a,long long b){return b?gcd(b,a%b)
    :a;}
long long lcm(long long a,long long b){return a/gcd(a,b)*b
    ;}
int main(){cout<<gcd(48,18)<<" "<<lcm(6,15)<<"\n";} // 6
    30
```

—

Modular arithmetic

Fast modular exponentiation, modular inverse.

Snippet: Modpow + inverse

```
// COMPILES: g++ -std=gnu++17 mod.cpp -O2
#include <bits/stdc++.h>
using namespace std;

long long modpow(long long a,long long b,long long m){
    long long r=1%m;
    while(b){ if(b&1) r=r*a%m; a=a*a%m; b>>=1; }
    return r;
}

// inverse only if m is prime
long long modinv(long long a,long long m){
    return modpow(a,m-2,m);
}

int main(){
    cout<<modpow(2,10,1e9+7)<<" "<<modinv(3,1e9+7)<<"\n";
}
```

—

Primes and sieve of Eratosthenes

Time $O(n \log\log n)$.

Snippet: Sieve

```
// COMPILES: g++ -std=gnu++17 sieve.cpp -O2
#include <bits/stdc++.h>
using namespace std;

vector<int> primes;
vector<bool> isprime;

void sieve(int n){
    isprime.assign(n+1,true);
    isprime[0]=isprime[1]=false;
    for(int i=2;i*i<=n;i++) if(isprime[i])
        for(int j=i*i;j<=n;j+=i) isprime[j]=false;
    for(int i=2;i<=n;i++) if(isprime[i]) primes.push_back(
     i);
}

int main(){
    sieve(30);
    for(int p:primes) cout<<p<<" "; // 2 3 5 7 11 13 17 19
      23 29
}
```

—

Extended Euclidean algorithm

Finds $gcd(a, b)$ and coefficients x, y such that $ax+by = g$.

Snippet: Extended GCD

```
// COMPILES: g++ -std=gnu++17 exgcd.cpp -O2
#include <bits/stdc++.h>
using namespace std;
long long exgcd(long long a,long long b,long long &x,long
     long &y){
    if(b==0){x=1;y=0;return a;}
    long long x1,y1;
    long long g=exgcd(b,a%b,x1,y1);
    x=y1;
    y=x1-(a/b)*y1;
    return g;
}
int main(){long long x,y; cout<<exgcd(30,18,x,y)<<" "<<x<<
     " "<<y<<"\n";}
```

—

Chinese Remainder Theorem (CRT)

Solve system of congruences $x \equiv a_i \pmod{m_i}$.

Snippet: CRT for coprime moduli

```
// COMPILES: g++ -std=gnu++17 crt.cpp -O2
#include <bits/stdc++.h>
using namespace std;

long long exgcd(long long a,long long b,long long &x,long
     long &y){
    if(b==0){x=1;y=0;return a;}
    long long x1,y1; long long g=exgcd(b,a%b,x1,y1);
    x=y1; y=x1-(a/b)*y1; return g;
}

long long crt(vector<long long> r, vector<long long> m){
    long long R=0,M=1;
    for(size_t i=0;i<r.size();i++){
        long long x,y; long long g=exgcd(M,m[i],x,y);
        if((r[i]-R)%g) return -1;
        long long t=(r[i]-R)/g*x%(m[i]/g);
        R+=M*t; M*=m[i]/g;
        R=(R%M+M)%M;
    }
    return R;
}

int main(){
    cout<<crt({2,3,2},{3,5,7})<<"\n"; // 23
}
```

—

Binomial coefficients

Precompute factorial + invfact.

Snippet: nCr mod prime

```
// COMPILES: g++ -std=gnu++17 ncr.cpp -O2
#include <bits/stdc++.h>
using namespace std;
const int MOD=1e9+7,N=1e6;
long long fact[N+1],invfact[N+1];
long long modpow(long long a,long long b){
    long long r=1;while(b){if(b&1)r=r*a%MOD;a=a*a%MOD;b
     >>=1;}return r;}
void init(){
    fact[0]=1;
    for(int i=1;i<=N;i++) fact[i]=fact[i-1]*i%MOD;
    invfact[N]=modpow(fact[N],MOD-2);
    for(int i=N;i>0;i--) invfact[i-1]=invfact[i]*i%MOD;
}
long long nCr(int n,int r){
    if(r<0||r>n) return 0;
```

```
    return fact[n]*invfact[r]%MOD*invfact[n-r]%MOD;
}
int main(){init(); cout<<nCr(5,2)<<"\n";} // 10
```

—

Matrix exponentiation

Use for linear recurrences. Complexity: $O(k^3 \log n)$.

Snippet: Fibonacci via matrix exp

```
// COMPILES: g++ -std=gnu++17 matpow.cpp -O2
#include <bits/stdc++.h>
using namespace std;
using M=array<array<long long,2>,2>;
const long long MOD=1e9+7;
M mul(M a,M b){
    M c={0};
    for(int i=0;i<2;i++) for(int j=0;j<2;j++){
        c[i][j]=0;
        for(int k=0;k<2;k++)
            c[i][j]=(c[i][j]+a[i][k]*b[k][j])%MOD;
    }
    return c;
}
M mpow(M a,long long e){
    M r={1,0,0,1};
    while(e){if(e&1)r=mul(r,a);a=mul(a,a);e>>=1;}
    return r;
}
int main(){
    M f={{1,1},{1,0}};
    long long n=10;
    cout<<mpow(f,n)[0][1]<<"\n"; // 55
}
```

—

Geometry basics

Use long long for integer coordinates; double/long double for real. Cross product sign = orientation test.

Snippet: Point struct + orientation

```
// COMPILES: g++ -std=gnu++17 point.cpp -O2
#include <bits/stdc++.h>
using namespace std;

struct Point{
    long long x,y;
    Point operator+(Point o){return {x+o.x,y+o.y};}
    Point operator-(Point o){return {x-o.x,y-o.y};}
    long long cross(Point o){return x*o.y - y*o.x;}
};

int orientation(Point a,Point b,Point c){
    long long v=(b-a).cross(c-a);
    if(v==0) return 0; // collinear
    return v>0?1:-1;   // 1=ccw, -1=cw
}

int main(){
    Point a{0,0},b{4,0},c{2,2};
    cout<<orientation(a,b,c)<<"\n"; // 1
}
```

—

Convex hull (Graham/Monotone chain)

Sort by x, then build upper/lower hull. Complexity: $O(n \log n)$.

Snippet: Convex hull (monotone chain)

```
// COMPILES: g++ -std=gnu++17 hull.cpp -O2
#include <bits/stdc++.h>
using namespace std;
struct P{long long x,y;};
long long cross(P a,P b,P c){
    return (b.x-a.x)*(c.y-a.y)-(b.y-a.y)*(c.x-a.x);
}
vector<P> hull(vector<P>&pts){
    sort(pts.begin(),pts.end(),[](P a,P b){return a.x<b.x
     ||(a.x==b.x&&a.y<b.y);});
    vector<P> h;
    for(auto p:pts){
        while(h.size()>=2 && cross(h[h.size()-2],h.back(),
     p)<=0) h.pop_back();
        h.push_back(p);
    }
    size_t k=h.size();
    for(int i=pts.size()-2;i>=0;i--){
        auto p=pts[i];
        while(h.size()>k && cross(h[h.size()-2],h.back(),p
    )<=0) h.pop_back();
        h.push_back(p);
    }
    h.pop_back();
    return h;
}
int main(){
    vector<P> pts={{0,0},{1,1},{2,0},{1,2}};
    auto h=hull(pts);
    cout<<h.size()<<"\n"; // 3
}
```

—

Rotating calipers

Compute diameter (max distance) of convex polygon in $O(n)$.

Snippet: Polygon diameter

```
// COMPILES: g++ -std=gnu++17 calipers.cpp -O2
#include <bits/stdc++.h>
using namespace std;
struct P{long long x,y;};
long long dist2(P a,P b){return (a.x-b.x)*(a.x-b.x)+(a.y-b
    .y)*(a.y-b.y);}
int main(){
    vector<P> poly={{0,0},{2,0},{2,2},{0,2}};
    int n=poly.size();
    long long best=0; int j=1;
    for(int i=0;i<n;i++){
        while(dist2(poly[i],poly[(j+1)%n])>dist2(poly[i],
     poly[j]))
            j=(j+1)%n;
        best=max(best,dist2(poly[i],poly[j]));
    }
    cout<<sqrt(best)<<"\n"; // 2.828...
}
```

—

String hashing

Polynomial rolling hash with modulus. Useful for substring comparisons in $O(1)$.

Snippet: Prefix hash

```
// COMPILES: g++ -std=gnu++17 hash.cpp -O2
#include <bits/stdc++.h>
using namespace std;
const long long MOD=1e9+7,BASE=911382323;
vector<long long> h,pw;
void build(const string&s){
    int n=s.size();
    h.assign(n+1,0); pw.assign(n+1,1);
    for(int i=0;i<n;i++){
        h[i+1]=(h[i]*BASE+s[i])%MOD;
        pw[i+1]=pw[i]*BASE%MOD;
    }
}
long long gethash(int l,int r){
    return (h[r]-h[l]*pw[r-l]%MOD+MOD)%MOD;
}
int main(){
    string s="abcab";
    build(s);
    cout<<(gethash(0,3)==gethash(2,5))<<"\n"; // 1
}
```

—

Prefix function (KMP)

Computes longest border for each prefix. Complexity: $O(n)$.

Snippet: KMP prefix function

```
// COMPILES: g++ -std=gnu++17 kmp.cpp -O2
#include <bits/stdc++.h>
using namespace std;
vector<int> prefix(const string&s){
    int n=s.size(); vector<int> pi(n);
    for(int i=1;i<n;i++){
        int j=pi[i-1];
        while(j>0 && s[i]!=s[j]) j=pi[j-1];
        if(s[i]==s[j]) j++;
        pi[i]=j;
    }
    return pi;
}
int main(){
    string s="ababc";
    auto pi=prefix(s);
    for(int x:pi) cout<<x<<" "; // 0 0 1 2 0
}
```

—

Z-function

$Z[i]$ = longest prefix of s starting at i. Complexity: $O(n)$.

Snippet: Z-function

```
// COMPILES: g++ -std=gnu++17 z.cpp -O2
#include <bits/stdc++.h>
using namespace std;
vector<int> zfunc(const string&s){
    int n=s.size(); vector<int> z(n);
    for(int i=1,l=0,r=0;i<n;i++){
        if(i<=r) z[i]=min(r-i+1,z[i-l]);
        while(i+z[i]<n && s[z[i]]==s[i+z[i]]) z[i]++;
        if(i+z[i]-1>r){l=i;r=i+z[i]-1;}
    }
    return z;
}
int main(){
    auto z=zfunc("aabcaabxaaaz");
    for(int x:z) cout<<x<<" ";
```

```
}
```

—

Suffix array (doubling)

Build suffix array in $O(n \log n)$.

Snippet: Suffix array

```
// COMPILES: g++ -std=gnu++17 sa.cpp -O2
#include <bits/stdc++.h>
using namespace std;
vector<int> sa_build(const string&s){
    int n=s.size();
    vector<int> sa(n),r(n),tmp(n);
    for(int i=0;i<n;i++){sa[i]=i;r[i]=s[i];}
    for(int k=1;k<n;k<<=1){
        auto cmp=[&](int a,int b){
            if(r[a]!=r[b]) return r[a]<r[b];
            int ra=a+k<n?r[a+k]:-1;
            int rb=b+k<n?r[b+k]:-1;
            return ra<rb;
        };
        sort(sa.begin(),sa.end(),cmp);
        tmp[sa[0]]=0;
        for(int i=1;i<n;i++) tmp[sa[i]]=tmp[sa[i-1]]+(cmp(
     sa[i-1],sa[i])?1:0);
        r=tmp;
    }
    return sa;
}
int main(){
    auto sa=sa_build("banana");
    for(int x:sa) cout<<x<<" "; // 5 3 1 0 4 2
}
```

—

Suffix automaton

DFA of all substrings, linear construction $O(n)$.

Snippet: Suffix automaton

```
// COMPILES: g++ -std=gnu++17 sam.cpp -O2
#include <bits/stdc++.h>
using namespace std;

struct SAState{
    int link,len;
    map<char,int> next;
};

struct SuffixAutomaton{
    vector<SAState> st; int last;
    SuffixAutomaton(string s=""){st.reserve(2*s.size());
     st.push_back({-1,0,{}}); last=0;
        for(char c:s) extend(c);}
    void extend(char c){
        int cur=st.size();
        st.push_back({0,st[last].len+1,{}});
        int p=last;
        while(p!=-1 && !st[p].next.count(c)){
            st[p].next[c]=cur; p=st[p].link;
        }
        if(p==-1) st[cur].link=0;
        else{
            int q=st[p].next[c];
            if(st[p].len+1==st[q].len) st[cur].link=q;
            else{
                int clone=st.size();
                st.push_back(st[q]);
                st[clone].len=st[p].len+1;
                while(p!=-1 && st[p].next[c]==q){
```

```
                    st[p].next[c]=clone; p=st[p].link;
                }
                st[q].link=st[cur].link=clone;
            }
        }
        last=cur;
    }
```

```
};

int main(){
    SuffixAutomaton sa("ababa");
    cout<<sa.st.size()<<"\n"; // 9
}
```