

# Competitive Programming Reference

ngmh

Last Updated: 07/01/2026

## Contents

<b>1 Data Structures</b>	<b>2</b>
1.1 Prefix Sums . . . . .	2
1.1.1 1D . . . . .	2
1.1.2 2D . . . . .	2
1.2 Fenwick Trees . . . . .	2
1.2.1 Point Update Range Query . . . . .	2
1.2.2 Range Update Point Query . . . . .	3
1.2.3 Range Update Range Query . . . . .	3
<b>2 Graph Theory</b>	<b>3</b>
2.1 Depth First Search . . . . .	3
2.2 Breadth First Search . . . . .	4
2.3 Union Find Disjoint Subset . . . . .	4
<b>3 Dynamic Programming</b>	<b>4</b>
3.1 Maxsum . . . . .	4
3.1.1 1D . . . . .	4
3.2 Longest Increasing Subsequence . . . . .	5
3.2.1 $N^2$ DP . . . . .	5
3.3 Coin Combinations . . . . .	5
3.4 Coin Change . . . . .	5
3.5 Knapsack . . . . .	5
3.5.1 0-1 . . . . .	5
3.6 Digit DP . . . . .	6
<b>4 Math</b>	<b>7</b>
4.1 Fast Exponentiation . . . . .	7
4.2 Prime Factorisation . . . . .	7
4.3 Sieve of Eratosthenes . . . . .	7
4.4 Greatest Common Divisor . . . . .	8
4.5 Lowest Common Multiple . . . . .	8
4.6 Modular Inverse . . . . .	8
4.7 $\binom{n}{k}$ . . . . .	8
4.8 Fibonacci . . . . .	9
<b>5 Algorithms</b>	<b>9</b>
5.1 Binary Search . . . . .	9
5.2 Binary Search using Lifting . . . . .	9
5.3 Sliding Set . . . . .	10
<b>6 Miscellaneous</b>	<b>10</b>
6.1 Fast I/O . . . . .	10
6.2 Superfast I/O . . . . .	10

# 1 Data Structures

Data Structure	Precomputation / Update	Query	Memory	Notes
Prefix Sum	$O(N) / X$	$O(1)$	$O(N)$	Associative Functions (+, XOR)
Sparse Table	$O(N \log N) / X$	$O(1)$	$O(N \log N)$	Non-Associative Functions (max, gcd)
Fenwick Tree	$X / O(\log N)$	$O(\log N)$	$O(N)$	Prefix Sum with Updates
Segment Tree	$X / O(\log N)$	$O(\log N)$	$O(4N)$	Allows more Information

Table 1: Quick Summary of Data Structures

## 1.1 Prefix Sums

### 1.1.1 1D

$O(N)$  precomputation,  $O(1)$  query.

---

```
//Query - 1-Indexed
int query(int s, int e){
    return ps[e]-ps[s-1];
}

//Precomputation
ps[0] = 0;
for(int i = 1; i <= n; i++) ps[i] = ps[i-1]+a[i];
```

---

### 1.1.2 2D

$O(R \cdot C)$  precomputation,  $O(1)$  query.

---

```
//Query - 1-Indexed
int query(int x1, int y1, int x2, int y2){
    return ps[x2][y2]-ps[x1-1][y2]-ps[x2][y1-1]+ps[x1-1][y1-1];
}

//Precomputation
for (int i = 0; i <= r; i++) ps[i][0] = 0;
for (int j = 0; j <= c; j++) ps[0][j] = 0;
for (int i = 1; i <= r; i++) {
    for (int j = 1; j <= c; j++) {
        ps[i][j] = ps[i-1][j]+ps[i][j-1]-ps[i-1][j-1]+a[i][j];
    }
}
```

---

## 1.2 Fenwick Trees

### 1.2.1 Point Update Range Query

$O(\log N)$  update and query.

---

```
inline int ls(int x){ return (x)&(-x); }

int fw[MAXN]; // 1-Indexed

void pu(int i, int v) {
    for(; i <= n; i += ls(i)) fw[i] += v;
}

int pq(int i) {
    int t = 0;
```

---

```

        for(; i; i -= ls(i)) t += fw[i];
    return t;
}

int rq(int s, int e) {
    return pq(e) - pq(s - 1);
}

```

---

### 1.2.2 Range Update Point Query

$O(\log N)$  update and query.

---

```

// Requires PURQ Code (PU, PQ)

void ru(int s, int e, int v) {
    pu(s, v);
    pu(e+1, -v);
}

```

---

### 1.2.3 Range Update Range Query

$O(\log N)$  update and query.

---

```

// Requires PURQ Code (PU, PQ)
// Functions need to be modified to take in array parameter
// e.g. int pu(*tree, int i, int v)

void ru(int s, int e, int v) {
    pu(fw1, s, v);
    pu(fw1, e+1, -v);
    pu(fw2, s, -v*(s-1));
    pu(fw2, e+1, v*e);
}

int ps(int i) {
    return pq(fw1, i)*i + pq(fw2, i);
}

int rq(int s, int e) {
    return ps(e) - ps(s - 1);
}

```

---

## 2 Graph Theory

### 2.1 Depth First Search

Runs in  $O(V)$ .

---

```

// For adjacency lists
void dfs(int x, int p) {
    for (int y : adj[x]) {
        if (y != p) {
            dist[y] = dist[x]+1;
            dfs(y, x);
        }
    }
}

```

---

## 2.2 Breadth First Search

Runs in  $O(V)$ .

---

```
// For adjacency lists
visited[s] = 1;
dist[s] = 0;
q.push(s);
while (!q.empty()) {
    int f = q.front(); q.pop();
    for (int i : adjlist[f]) {
        if (!visited[i]) {
            q.push(i);
            visited[i] = 1;
            dist[i] = dist[f] + 1;
        }
    }
}
```

---

## 2.3 Union Find Disjoint Subset

With both path compression and union by rank, runs in  $O(\alpha(n))$  (basically constant time).

---

```
int p[MAXN];
int sz[MAXN];

int root(int x) {
    if (p[x] == -1) return x;
    return p[x] = root(p[x]);
}

void connect(int x, int y) {
    x = root(x); y = root(y);
    if (x == y) return;
    if (sz[x] < sz[y]) swap(x, y);
    p[y] = x;
    sz[x] += sz[y];
}

fill(p, p+MAXN, -1);
fill(sz, sz+MAXN, 1);
```

---

## 3 Dynamic Programming

### 3.1 Maxsum

#### 3.1.1 1D

Kadane's Algorithm. Runs in  $O(N)$ .

---

```
int ans = nums[0], cur = nums[0];
for (int i = 1; i < nums.size(); i++) {
    if (cur < 0) cur = 0;
    cur += nums[i];
    ans = max(ans, cur);
}
```

---

## 3.2 Longest Increasing Subsequence

### 3.2.1 $N^2$ DP

---

```
int ans = 0, dp[n];
memset(dp, 0, sizeof(dp));
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        if (a[j] < a[i]) {
            dp[i] = max(dp[i], dp[j]);
        }
    }
    dp[i]++;
    ans = max(ans, dp[i]);
}
```

---

## 3.3 Coin Combinations

Runs in  $O(N \cdot V)$ .

---

```
int ways[v+1];
memset(ways, 0, sizeof(ways));
ways[0] = 1;
for (int i = 0; i < n; i++) {
    int c = coins[i];
    for (int sum = c; sum <= v; sum++) {
        ways[sum] = (ways[sum] + ways[sum - c]) % MOD;
    }
}
cout << ways[v];
```

---

## 3.4 Coin Change

Runs in  $O(N \cdot V)$ .

---

```
const int INF = 1e9;
vector<int> dp(v + 1, INF);
dp[0] = 0;
for (int i = 1; i <= v; i++) {
    for (int j = 0; j < n; j++) {
        if (i >= c[j] && dp[i - c[j]] != INF) {
            dp[i] = min(dp[i], dp[i - c[j]] + 1);
        }
    }
}
cout << dp[v];
```

---

## 3.5 Knapsack

### 3.5.1 0-1

Runs in  $O(N \cdot S)$ .

---

```
for (int i = 0; i < n; i++) {
    for (int j = s; j >= w[i]; j--) {
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}
cout << dp[s];
```

---

### 3.6 Digit DP

Runs in  $O(D)$ .

---

```

// Example - Numbers
// Compute the number of palindrome free numbers in a given range

vector<int> num;
ll dp[20][11][11][2][2]; // idx, last1, last2, tight, hasStarted

ll derp(int pos, int last1, int last2, bool tight, bool hasStarted) {
    if(pos == num.size()) return 1; // successfully populated whole number

    if(dp[pos][last1][last2][tight][hasStarted] != -1) {
        // state already visited
        return dp[pos][last1][last2][tight][hasStarted];
    }

    ll res = 0;
    int limit = tight ? num[pos] : 9; // do we need to keep to the range
    for(int d = 0; d <= limit; d++) { // try all next digits
        bool newHasStarted = hasStarted || (d != 0);
        bool newTight = tight && (d == limit);
        // skip palindromes only if the number has started
        if(newHasStarted) {
            if(d == last1) continue; // palindrome length 2
            if(d == last2) continue; // palindrome length 3
        }
        int newLast1 = newHasStarted ? d : 10;
        int newLast2 = hasStarted ? last1 : 10;
        res += derp(pos+1, newLast1, newLast2, newTight, newHasStarted);
    }
    return dp[pos][last1][last2][tight][hasStarted] = res;
}

// convert number to digits
void dcmp(ll x){
    num.clear();
    if(x == 0) num.push_back(0);
    while(x > 0) {
        num.push_back(x % 10);
        x /= 10;
    }
    reverse(num.begin(), num.end());
}

// Total Valid with value <= x
// Use PIE to get number within [a, b]
ll solve(ll x){
    dcmp(x);
    memset(dp, -1, sizeof(dp));
    return derp(0, 10, 10, true, false);
}

// To compute the kth string satisfying
// Either binary search or build character by character:
int count = 0;
vector<int> ans;

```

---

```

for (int i = 0; i < n; i++) {
    int x = 0;
    for (int j = 1; j < 10; j++) { // adjust to size of alphabet
        if (count + dp(i, j) > k) {
            break;
        }
        x = j;
    }
    count += dp(i, x); // position i is bounded by x
    ans.push_back(x);
}

```

---

## 4 Math

### 4.1 Fast Exponentiation

Runs in  $O(\log b)$ .

---

```

int powmod(int a, int b, int m) {
    int res = 1;
    while (b > 0) {
        if (b & 1) res = (res * a) % m;
        a = (a * a) % m;
        b >= 1;
    }
    return res % m;
}

```

---

### 4.2 Prime Factorisation

Runs in  $O(\sqrt{x})$ .

---

```

map<int, int> cnt;
while (x % 2 == 0) {
    cnt[2]++;
    x /= 2;
}
for (int i = 3; i * i <= x; i++) {
    while (x % i == 0) {
        cnt[i]++;
        x /= i;
    }
}
if (x > 1) {
    cnt[x]++;
}

```

---

### 4.3 Sieve of Eratosthenes

Runs in  $O(n \log \log n)$  with high constant.

---

```

bitset<MAXN> prime;
prime.set();
prime[0] = prime[1] = 0;
for (int i = 2; i < MAXN; i++) {
    if (prime[i]) {
        for (int j = i*i; j < MAXN; j += i) {

```

---

```

        prime[j] = 0;
    }
}

```

---

## 4.4 Greatest Common Divisor

Runs in  $O(\log \min(a, b))$ .

---

```

int gcd(int a, int b) {
    if (a > b) swap(a, b);
    while (a != 0) {
        b %= a;
        swap(a, b);
    }
    return b;
}

```

---

## 4.5 Lowest Common Multiple

---

```

int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}

```

---

## 4.6 Modular Inverse

---

```

ll modinv(ll a){
    return powmod(a, MOD-2, MOD);
}

```

---

## 4.7 $\binom{n}{k}$

Precomputation takes  $O(MAXN)$  time, queries answered in  $O(1)$ .

---

```

ll fac[MAXN+1], modinv[MAXN+1];

ll nck(ll n, ll k) {
    if (n < k) return 0;
    ll res = fac[n];
    res = (res * modinv[k]) % MOD;
    res = (res * modinv[n-k]) % MOD;
    return res;
}

fac[0] = 1;
for(int i = 1; i <= MAXN; i++) {
    fac[i] = fac[i-1] * i % MOD;
}
modinv[MAXN] = powmod(fac[MAXN], MOD-2, MOD);
for(int i = MAXN; i > 0; i--) {
    modinv[i-1] = modinv[i] * i % MOD;
}

```

---

## 4.8 Fibonacci

Runs in  $O(\log N)$  time.

---

```
struct Mat {
    ll a, b, c, d; // 2x2 Matrix: [a b; c d]
};

Mat mul(Mat x, Mat y) {
    return {
        x.a*y.a + x.b*y.c,
        x.a*y.b + x.b*y.d,
        x.c*y.a + x.d*y.c,
        x.c*y.b + x.d*y.d
    };
}

Mat mpow(Mat base, long long exp) {
    Mat res = {1, 0, 0, 1}; // Identity Matrix
    while (exp) {
        if (exp & 1) res = mul(res, base);
        base = mul(base, base);
        exp >>= 1;
    }
    return res;
}

ll fib(long long n) {
    if (n == 0) return 0;
    Mat m = {1, 1, 1, 0}; // Fibonacci Seed Matrix
    return mpow(m, n-1).a;
}
```

---

## 5 Algorithms

### 5.1 Binary Search

Find the cuberoot of  $n$ . Runs in  $O(\log N)$ .

---

```
long long n; cin >> n;
long long mini = 0, maxi = 1e6, medi;
while (mini < maxi) {
    medi = mini+(maxi-mini)/2;
    if (medi * medi * medi >= n) maxi = medi;
    else mini = medi+1;
}
cout << mini << "\n";
```

---

### 5.2 Binary Search using Lifting

Find the cuberoot of  $n$ . Runs in  $O(\log N)$ .

---

```
long long n; cin >> n;
long long cur = 0, gap = 1e6, next;
while (gap > 0) {
    while (next = cur + gap, next * next * next < n) {
        cur = next;
    }
}
```

```
    gap >= 1;
}
cout << cur+1 << "\n";
```

---

### 5.3 Sliding Set

Speeds up DP from  $O(N^2)$  to  $O(N \log N)$ .

---

```
// Example - Candymountain
// Jump across with minimax candies
// Populate with initial window
for(int i = 0; i < k; i++) {
    dp[i] = candies[i];
    s.insert(candies[i]);
}
// Sliding Set
for(int i = k; i < n; i++){
    dp[i] = max(candies[i], *s.begin());
    s.erase(s.find(dp[i-k]));
    s.insert(dp[i]);
}
```

---

## 6 Miscellaneous

### 6.1 Fast I/O

Cannot use with `scanf`, `printf`.

---

```
ios_base::sync_with_stdio(false);
cin.tie(0);
```

---

### 6.2 Superfast I/O

Only for non-negative integer input.

---

```
inline ll ri () {
    ll x = 0;
    char ch = getchar_unlocked();
    while (ch < '0' || ch > '9') ch = getchar_unlocked();
    while (ch >= '0' && ch <= '9') {
        x = (x << 3) + (x << 1) + ch - '0';
        ch = getchar_unlocked();
    }
    return x;
}
```

---