# Competitive Programmer's Handbook Note's

- Mathematics
- Time Complexity
- Sorting
- Data Structures
  - Dynamic Arrays
  - Set Structures
  - Map Structures
  - Iterators and Ranges
  - Other Structures
- Complete Search
- Greedy Algorithms
- Dynamic programming
- Amortized analysis

## Mathematics

Important knowledge to know/review when solving programming problems.

#### Sum Formulas

$\sum_{x=1}^{n} x^k = 1^k + 2^k + 3^k + ... + n^k$

#### Arithmetic progressions

#### Geometric progressions

#### Harmonic sum

#### Set Theory

**Intersection:** Items in both A and B.

$A = \{1, 2, 5\}$
$B = \{2, 4\}$
$A \cap B = \{2\}$

**Union:** Items in A and B or both.

$A = \{3, 7\}$
$B = \{2, 3, 8\}$
$A \cup B = \{2, 3, 4, 8\}$

**Complement:** Elements that are not in A.

$U = \{1, ..., 10\}$
$A = \{1, 2, 3, 7\}$
$\overline{A} = \{4, 5, 6, 8, 9, 10\}$

**Difference:** Elements that are in A but not in B.

$A = \{2, 3, 7, 8\}$
$B = \{2, 5, 8\}$
$A \backslash D = A \cap \overline{B} = \{2, 7\}$

**Subset:** If each element of A belongs to S, we say that A is a subset of S. Denoted $A \subset S$.

### Logic

### Functions

### Logarithms

## Time Complexity

## Sorting

## Data Structures

### Dynamic Arrays

A **dynamic array** is an array whose size can be changed during the execution of the program (Similar to Python lists).

### Methods

```cpp
// initialize an empty vector
vector<int> v;

// append element to the back of the array using
// push_back(el) method.
// python: append(el)
v.push_back(3); // [3]
v.push_back(2); // [3,2]
v.push_back(5); // [3,2,5]

// indexing
cout << v[0] << '\n'; // 3
cout << v[1] << '\n'; // 2
cout << v[2] << '\n'; // 5

// iterating through a vector
for(int i = 0; i < v.size(); i++){
  cout << v[i] << '\n';
}

// shorter syntax
for(int x : v){
  cout << x << '\n';
}

// back and pop_back
// python: [-1] and pop()
vector<int> v;
v.push_back(5);
v.push_back(2);
cout << v.back() << '\n'; // 2
v.pop_back();
cout << v.back() << '\n'; // 5

// initialize a vector with elements
vector<int> k = {1, 2, 3, 4};
```

```cpp
// size 10, initial value 0
vector<int> k(10);

// size 10, initial value 5
vector<int> v(10, 5);
```

**Strings** The **string** structure is also a dynamic array that can be used almost like a vector. Strings can use the `+` operator `substr`, and `find(t)`.

```cpp
string a = "linear";
string b = a + a;
cout << b << '\n'; // linearlinear
// unlike python strings c++ strings are mutable.
b[1] = 'z';
cout << b << '\n'; // lznearlinear

// find returns the first index of the
// given char
cout << b.find('z') << '\n'; // 1

// substr(staring intdex, length of substring)
// Unlike python you don't specify the ending
// index rather the length of the substring.
string c = b.substr(2, 4);
cout << c << '\n'; // near
```

### Set Structures

A **set** is a data structure that maintains a collection of elements. Basic operations include insertion, search, and removal.

**Implementations** The structure `set` is based on a balanced binary tree and its operations work in `O(log n)` time.

The structure `unordered_set` uses hashing, and its operations work in `O(1)` time on average.

| Differences | set | unordered_set |
|---|---|---|
| Ordering | increasing order (by default) | no ordering |
| Implementation | Self balancing BST (Like Red-Black Tree) | Hash Table |
| Search time | log(n) | O(1) -> Average O(n) -> Worst Case |
| Insertion time | log(n) + Rebalance | Same as search |
| Deletion time | log(n) + Rebalance | Same as search |

Reference

### Methods

```
// initialize empty set
set<int> s;
set<int> t = {1, 2, 3};

// append element
// python: append(el)
s.insert(1);
s.insert(2);
s.insert(3);
cout << s.count(3) << '\n'; // 1
cout << s.count(4) << '\n'; // 0

// remove element
// python: remove(el)
s.erase(3);
cout << s.count(3) << '\n'; // 0
```

Unlike vectors elements of sets cannot be accessed using [] notation. The

```
// initialize set with elements
set<int> s = {1, 2, 3, 4};

// size() method returns the length of the set
// python: len(s)
cout << s.size() << '\n'; // 4

// prints each element in the set
for(auto x : s){
  cout << x << '\n';
}
```

All elements of a set are *distinct*, meaning they only occur once. Therefore, the method `count` always returns either 0 or 1. The method `insert` never adds a duplicate element to the set.

```
set<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << '\n'; // 1
```

**Multiset** C++ also contains the structure `multiset` and `unordered_multiset` that work similarly to sets, but contain multiple instances of an element.

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << '\n'; // 3

s.erase(5);
cout << s.count(5) << '\n'; // 0

s.erase(s.find(5));
cout << s.count(5) << '\n'; // 2
```

### Map Structures

A **map** is generalized array that consists of key-value-pairs. In an array the keys are the index's 0,1,…,n-1 where n is the size of the array, the keys in a map can be any data type.

**Implementations** The `map`is based on a balanced binary tree and accessing elements take `O(log n)`.

The `unordered_map` uses hashing and accessing elements takes `O(1)` time on average.

```
map<string, int> m;
m["Ripple"] = 12;
m["Ellie"] = 6;
m["Forty"] = 2;
cout << m["Ellie"] << '\n'; // 6
```

If the value of the key is requested but the map does not contain it, the key is automatically added to the map with default value of 0.

```
map<string, int> m;
cout << m["Kanou"] << '\n'; // 0
```

**Methods** `count` checks if a key exists in a map:

```
if(m.count("Kanou")){
  // key exists
}
```

To print all the keys and values:

```
for(auto x : m){
  cout << x.first << " " << x.second << '\n';
}
```

### Iterators and ranges

An **iterator** is a variable that points to an element in a data structure.

```
{ 3, 4, 6, 8, 12, 13, 14, 17 }
  ^                           ^
  |                           |
  s.begin()                   s.end()
```

**Working with ranges** Sorting vectors

```
sort(v.begin(), v.end());
reverse(v.begin(), v.end());
random_shuffle(v.begin(), v.end());
```

Sorting arrays

```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

**Set iterators** The following code create an iterator `it` that points to the smallest element in a set:

```
set<int>::iterator it = s.begin();

// Shorter way to write
auto it = s.begin();
```

iterator pointers can be accessed using the * symbol.

```
auto it = s.begin();
cout << *it << '\n';
```

Iterators can be moved using the operators ++ (forward) and -- (backwards). Thus, iterators can be used to print all the elements in increasing order:

```
for(auto it = s.begin(); it != s.end(); it++){
  cout << *it << '\n';
}
```

`find(x)` returns an iterator that points to an element whose value is `x`. If the set does nto contain x, the iterator will be equal to `.end()`

```
auto it = s.find(x);
if(it == s.end()){
  // x is not found
}
```

`lower_bound(x)` returns an iterator to the smallest element in the set whose value is *at least* $x$.

`upper_bound(x)` returns an iterator to the smallest element in the set whose value is *larger than* $x$.

Ex. Finds the element nearest to $x$:

```
auto it = s.lower_bound(x);
if (it == s.begin()) {
  cout << *it << '\n';
} else if (it == s.end()) {
  it--;
  cout << *it << '\n';
} else {
  int a = *it; it--;
  int b = *it;
  if (x-b < a-x) cout << b << '\n';
  else cout << a << '\n';
}
```

### Other Structures

**Bitset** A **bitset** is an array whose each value is either 0 or 1.

Ex. Creates a bitset that contains 10 elements:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
cout << s[4] << '\n'; // 1
cout << s[5] << '\n'; // 0
```

Bitsets only uses 1 bit of of memory per element while in a normal array 32 bits are allocated for each element. Additionally, bit operators can be used to efficiently manipulate bitsets.

```
bitset<10> s(string("0010011010")); // from right to left
cout << s[4] << '\n'; // 1
cout << s[5] << '\n'; // 0

// Return the number of ones in the bitset
```

```
cout << s.count() << '\n'; // 4

// Using bit operations
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a&b) << "\n"; // 0010010000
cout << (a|b) << "\n"; // 1011111110
cout << (a^b) << "\n"; // 1001101110
```

**Deque** A **deque** is a dynamic array whose size can be efficiently changed at both ends of the array.

Deque provides `push_back`, `pop_back`, `push_front`, and `pop_front`.

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

**Stack** A **stack** is a data structure that provides two `O(1)` time operations for adding an element to the top and removing an element from the top. It is only possible to access the top element of a stack.

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top() << '\n'; // 5
s.pop();
cout << s.top() << '\n'; // 2
```

**Queue** A **queue** also provides two `O(1)` time operations for adding an element to the end and removing the first element.

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front() << '\n'; // 3
q.pop();
cout << q.front() << '\n'; // 3
```

**Priority Queue** A **priority queue** maintains set of elements. Insertion and removal take `O(log n)` time, and retrieval takes `O(1)` time.

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
```

```
cout << q.top() << "\n"; // 6
q.pop();
```

## Complete Search

**Complete search** implements a brute force search that generates all possible solutions to the problem and then selects the best solution or count the number of solutions, depending on the problem.

If complete search is too slow, other techniques, such as greedy algorithms or dynamic programming can be used.

### Generating subsets

Problem covered in leetcode 78 we want to generate all the subsets of a given set. For example, the subsets of `{1, 2, 3}` are `{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}` and `{1, 2, 3}`

**Method 1** Use recursion. The follow function `search` generates the subsets of the set `{0, 1, ..., n - 1}`.

```
void search(int k){
  if (k == n){
    // process subset
  } else {
    search(k+1);
    subset.push_back(k);
    search(k+1);
    subset.pop_back();
  }
}
```

The following tree illustrates the function calls when $n = 3$. We can always choose either the left branch ($k$ is not included in the subset) or the right branch ($k$ is included in the subset).

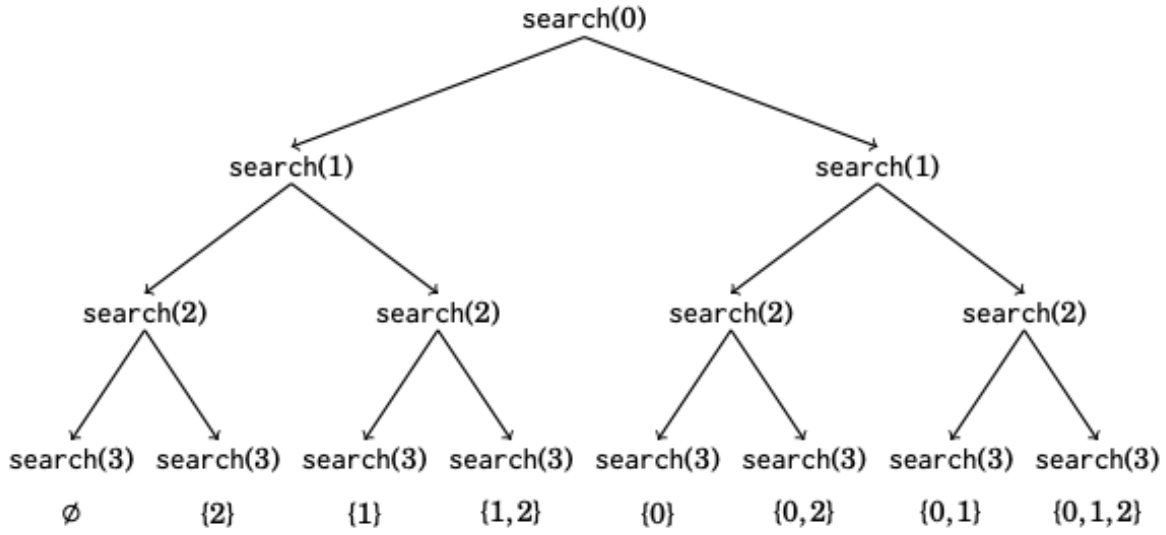



Figure 1: Search function tree

**Method 2** Another way to generate subsets is based on bit representation of integers. For example, the bit representation 25 is 11001, which corresponds to the subset {0, 3, 4}.

```
for (int b = 0; b < (1<<n); b++){
  // process subset
}

// Using bit sequence
for (int b = 0; b < (i<<n); b++){
  vector<int> subset;
  for (int i = 0; i < n; i++){
    if (b&(1<<i)) subset.push_back(i);
  }
}
```

### Generating permutation

Permutations of {0, 1, 2} are (0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), and (2, 1, 0). Two approaches: Recursion or go through the permutations iteratively.

### Method 1

```
void search() {
  if (permutation.size() == n){
    // process permutation
  } else {
    for (int i = 0; i < n; i ++){
      if (chosen[i]) continue;
      chosen[i] = true;
      permutations.push_back(i);
      search();
      chosen[i] = false;
      permutation.pop_back();
    }
  }
}
```

### Method 2

```
vector<int> permutation;
for (int i = 0; i < n; i++){
  permutation.push_back(i);
}
do {
  // process permutation
} while (next_permutation(permutation.begin(), permutation.end()));
```

### Backtracking

A **backtracking** algorithm begins with an empty solution and extends the solution step by step.

Implementation:

```
void search(int y){
  if (y == n){
    count++;
    return;
```

```
  }
  for (int x = 0; x < n; x++){
    if (column[x] || diag1[x + y] || diag2[x-y+n-1]) continue;
    column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
    search(y+1)
    column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
  }
}
```

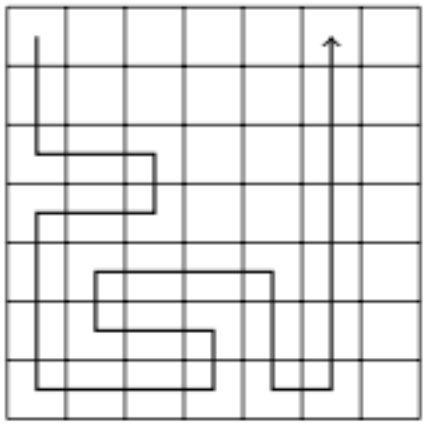
### Pruning the search

The idea is to add "intelligence" to the algorithm so that it will notice as soon as possible if a partial solution cannot be extended to a complete solution.

**Optimization 1** We know that the paths are symmetric about the diagonal of the grid. So, we can decide that we always first move one step down or right which reduces are search spaces. Then we multiply the solution by two to make up for the symmetric solutions.

**Optimization 2** If the lower-right square is reached before all other nodes.

**Optimization 3** If the path touches a wall and can turn either left or right, the grid splits into two parts that contain unvisited squares.



Example:

**Optimization 4** The idea of Optimization 3 can be genetalized: if the path cannot continue forward but can turn either left or right, the grid splits int two parts that both contain unvisited squares.

This is a usual phenomenon in backtracking, because the search tree is usually large and even simple observations can effectively prune the search.

### Meet in the middle

**Meet in the middle** is a technique where the search space is divided into two parts of about equal size. A separate search is performed for both parts and the results are combined.

**Example:** Go through all subsets of the elements and check if the sum of the subsets is x. $O(2^n)$, because there are $2^n$ subsets. Using meet in the middle technique, we can achieve $O(n^{n/2})$.

Divided both the lists into A and B such that both lists contain about half of the numbers. The first search generates all subsets of A and stores their sums to a list S_A$. Correspondingly, the second search creates

a list $S_B$ from B. After this, it suffices to check if it is possible to choose one element from $S_A$ and another element from $S_B$ such that their sum is x.

## Greedy algorithms

A **greedy algorithm** constructs a solution to the problem by always making a choice that looks the best at the moment. A greedy algorithm never takes back its choices, but directly constructs the final solution.

### Coin problem

{1, 2, 5, 10, 20, 50, 100, 200}

If n = 520, we need at least four coins. The optimal solution is to select coins 200 + 200 + 100 + 20 which equals 520.

### Greedy algorithm

Always select the largest possible coin, until the required sum of money has been constructed.

### Scheduling

Many scheduling problems can be used using a greedy algorithm.

**Algorithm 1** Always selects the the shortest events possible. - Not optimal

**Algorithm 2** Always select the next possible event that begins as early as possible. - Not optimal

**Algorithm 3** Always select the next possible even that ends as early as possible. - Optimal

### Tasks and deadlines

Consider a problem where we are given $n$ tasks with durations and deadlines and our task is to choose an order to perform the tasks.

The optimal solution is just the durations sorted in increasing order.

### Minimizing sums

Consider the problem where we are given $n$ numbers from $a_1, a_2, ..., a_n$ and our task is to find a value $x$ that minimizes the sum.

$|a_1 - x|^c + |a_2 - x|^c + ... + |a_n - x|^c$.

**Case c = 1** For example, if the numbers [1, 2, 9, 2, 6], the best solution is to select the median number (the middle number).

**Case c = 2** $(a_1 - x)^2 + (a_2 - x)^2 + ... + (a_n - x)^2$

In the general case, teh best choice for $x$ is the average of the numbers.

### Data compression

A **binary code** assigns for each character of a string a **codeword** that consists of bits.

By using the same length for each codeword we can create **constant-length** compressed strings.

**Huffman coding** **Huffman coding** is a greedy algorithm that constructs an optimal code for compressing a given string. The algorithm builds a binary tree based on the frequencies of each character.

## Dynamic programming

**Dynamic programming** is a technique that combines the correctness of complete search and the efficiency of the greedy algorithm. It can be applied if the problem can be divided into overlapping sub-problems that can be solved independently.

Two uses for dynamic programming: - Finding the optimal solution - Counting the number of solutions

### Coin problem

The dynamic programming algorithm is based on a recursive function that goes through all possibilities how to form the sum, like a brute force algorithm. However, the dynamic programming algorithm is efficient because it uses *memoization* and calculates the answer to each subproblem only once.

### Recursive formulation

```
int solve(int x) {
  if (x < 0) return INF;
  if (x == 0) return 0;
  int best = INF;
  for (auto c : coins) {
    best = min(best, solve(x-c)+1);
  }
  return best;
}
```

**Using memoization** **Memoization** stores the previous values calculated in an array after they've been calculated. You could also implement memoization using a hashmap or python dictionary.

```
int solve(int x) {
  if (x < 0) return INF;
  if (x == 0) return 0;
  if (ready[x]) return value[x];
  int best = INF;
  for (auto c : coins) {
    best = min(best, solve(x-c)+1);
  }
  value[x] = best;
  ready[x] = true;
  return best;
}
```

Using an iterative approach:

```
value[0] = 0;
  for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
      if (x-c >= 0) {
        value[x] = min(value[x], value[x-c]+1);
    }
  }
}
```

**Constructing a solution** Sometimes we need to produce the coins in the optimal solution.

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
  value[x] = INF;
  for (auto c : coins) {
    if (x-c >= 0 && value[x-c]+1 < value[x]) {
      value[x] = value[x-c]+1;
      first[x] = c;
    }
  }
}

while (n > 0) {
  cout << first[n] << "\n";
  n -= first[n];
}
```

**Counting the number of solutions** Calculate the total number of ways to produce a sum $x$ using the coins.

```
count[0] = 1;
for (int x = 1; x <= n; x++) {
  for (auto c : coins) {
    if (x-c >= 0) {
      count[x] += count[x-c];
    }
  }
}
```

### Longest increasing subsequence

Find the maximum-length sequence of array elements that goes from left to right, and each element in the sequence is larger than the previous element.

```
# Given
[6, 2, 5, 1, 7, 4, 8, 3]

# Answer
[2, 5, 7, 8]
```

```
for (int k = 0; k < n; k++) {
  length[k] = 1;
  for (int i = 0; i < k; i++) {
    if (array[i] < array[k]) {
      length[k] = max(length[k],length[i]+1);
    }
  }
}
```

### Paths in a grid

```
// O(n^2)
for (int y = 1; y <= n; y++) {
  for (int x = 1; x <= n; x++) {
    sum[y][x] = max(sum[y][x-1],sum[y-1][x])+value[y][x];
  }
}
```

### Knapsack problems

The term **knapsack** refers to problems where a set of objects is given, and subsets with some properties have to be found.

Given a list of weights $[w_1, w_2, ..., w_n]$, determine all sums that can be constructed using the weights. For example, if the weights are [1,3,3,5], the following sums are possible: Every weight between 0-12, expect 2 and 10.

Let $W$ denote the total sum of the weights. The following $O(nW)$ time dynamic programming solutions corresponds to the recursive function:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
  for (int x = 0; x <= W; x++) {
    if (x-w[k] >= 0) possible[x][k] |= possible[x-w[k]][k-1];
    possible[x][k] |= possible[x][k-1];
  }
}
```

Better implementation using a trick to update the array from right to left for each new weight: `{c, echo=TRUE, eval=TRUE} possible[0] = true for (int k = 1; k <= n; k++) {  for (int x = W; x >= 0; x--) {    if (possible[x]) possible[x+w[k]] = true;  } }`

### Edit distance

The **edit distance** or **Levenshtein distance** is the minimum number of editing operations needed to transform a string into another string. Following edit operations: - Insert (ABC -> ABCA) - Remove (ABC -> AC) - Modify (ABC -> ADC)

For example, the edit distance between LOVE and MOVIE is 2.

## Amortized analysis

**Amortized analysis** can be used to analyze algorithms that contain operations whose time complexity varies. The idea is to estimate the total time used to all such operations during the execution of the algorithm, instead of focusing on individual operations.

### Two pointers method

### Nearest smaller elements

### Sliding window minimum