C_Program/algorithm_list.txt

C_Programディレクトリ - アルゴリズム詳細一覧
==

Chapter 1: 入門

ALDS1_1_A: 挿入ソート（Insertion Sort）
  - N個の整数を含む配列を昇順に整列
  - 各ステップでの配列の状態を出力
  - 時間計算量: O(N²)

ALDS1_1_B: 最大公約数（Greatest Common Divisor）
  - ユークリッドの互除法を用いて2つの自然数の最大公約数を求める
  - gcd(x, y) = gcd(y, x mod y) の再帰的な性質を利用
  - 時間計算量: O(log min(x,y))

ALDS1_1_C: 素数判定（Prime Numbers）
  - n個の整数を読み込み、素数の個数を出力
  - 素数: 1とその数自身のみを約数に持つ自然数
  - √nまでの数で割り切れるかチェック

ALDS1_1_D: 最大利益（Maximum Profit）
  - FX取引での最大利益を求める
  - 時刻tにおける価格列から「買い→売り」で得られる最大差分を計算
  - 動的に最小値を更新しながら最大差分を求める

Chapter 2: ソート

ALDS1_2_A: バブルソート（Bubble Sort）
  - 隣接要素を比較して交換を繰り返す
  - 交換回数をカウント
  - 時間計算量: O(N²)、安定ソート

ALDS1_2_B: 選択ソート（Selection Sort）
  - 各ステップで最小値を選択して先頭に移動
  - 交換回数をカウント
  - 時間計算量: O(N²)、不安定ソート

ALDS1_2_C: 安定なソート（Stable Sort）
  - トランプカードを整列し、ソートの安定性を検証
  - バブルソートと選択ソートを両方実装して比較
  - 安定性: 同じキーを持つ要素の相対順序が保たれるか

ALDS1_2_D: シェルソート（Shell Sort）
  - 挿入ソートを改良した高速なソートアルゴリズム
  - 一定間隔(gap)ごとに要素を取り出して挿入ソートを適用
  - 使用したgap列と比較回数を出力

Chapter 3: データ構造

ALDS1_3_A: スタック（Stack）
  - 逆ポーランド記法の数式を計算
  - LIFO(後入れ先出し)構造
  - push/pop操作で四則演算を実行

ALDS1_3_B: キュー（Queue）
  - ラウンドロビンスケジューリングをシミュレート
  - FIFO(先入れ先出し)構造

  - クオンタム時間ごとにプロセスを処理

ALDS1_3_C: 双方向連結リスト（Doubly Linked List）
  - insert, delete, deleteFirst, deleteLast操作を実装
  - 前後のノードへのポインタを持つ
  - 要素の追加・削除が効率的

ALDS1_3_D: 断面図の水たまり面積（Areas on Cross-Section Diagram）
  - 地形の断面図から水たまりの面積を計算
  - スタックを使って左右の壁を追跡
  - 各水たまりの面積と総面積を出力

Chapter 4: 探索

ALDS1_4_A: 線形探索（Linear Search）
  - 配列Sの要素がすべて配列Tに含まれるか判定
  - 順番に要素を比較
  - 時間計算量: O(N×Q)

ALDS1_4_B: 二分探索（Binary Search）
  - ソート済み配列に対して効率的に探索
  - 探索範囲を半分ずつ絞り込む
  - 時間計算量: O(Q log N)

ALDS1_4_C: 辞書（Dictionary）
  - ハッシュテーブルを用いた辞書の実装
  - insert: 文字列を追加
  - find: 文字列が存在するか判定
  - 平均時間計算量: O(1)

ALDS1_4_D: 割り当て問題（Allocation）
  - N個の荷物をK台のトラックに積む
  - 二分探索を使って最小の最大積載量を求める
  - 判定問題に帰着させて解く

Chapter 5: 再帰・分割統治法

ALDS1_5_A: 総当たり（Exhaustive Search）
  - 部分和問題: 配列の要素の組み合わせで指定の値が作れるか
  - ビット全探索で2^N通りの組み合わせをチェック
  - 時間計算量: O(2^N × Q)

ALDS1_5_B: マージソート（Merge Sort）
  - 分割統治法による効率的なソート
  - 配列を分割→再帰的にソート→マージ
  - 時間計算量: O(N log N)、安定ソート

ALDS1_5_C: コッホ曲線（Koch Curve）
  - フラクタル図形を再帰的に生成
  - 線分を3等分し、中央に正三角形の突起を作る
  - 深さNの再帰呼び出しで頂点座標を出力

ALDS1_5_D: 反転数（The Number of Inversions）
  - 数列中のi<jかつA[i]>A[j]となるペアの個数
  - マージソートの過程でカウント
  - バブルソートの交換回数と等しい

Chapter 6: ソート応用

ALDS1_6_A: 計数ソート（Counting Sort）

  - 各値の出現回数をカウントして整列
  - 0以上k以下の整数に対して線形時間でソート
  - 時間計算量: O(N+k)、安定ソート

ALDS1_6_B: パーティション (Partition)
  - クイックソートの基本操作
  - 配列を基準値より小さい部分と大きい部分に分割
  - 基準値の最終位置を返す

ALDS1_6_C: クイックソート (Quick Sort)
  - パーティションを再帰的に適用
  - トランプカードをソートして安定性を検証
  - 平均時間計算量: O(N log N)、不安定ソート

ALDS1_6_D: 最小コストソート (Minimum Cost Sort)
  - 要素の交換にコスト(重みの合計)がかかる場合の最小コスト
  - 各サイクルごとに最適な交換方法を選択
  - サイクル内の最小値または全体の最小値を使う

Chapter 7: 木構造

ALDS1_7_A: 根付き木 (Rooted Trees)
  - 各節点の親、深さ、種類(根/内部節点/葉)、子のリストを出力
  - 木構造の基本的な性質を理解
  - DFSまたはBFSで深さを計算

ALDS1_7_B: 二分木 (Binary Trees)
  - 各節点の親、兄弟、子の数、深さ、高さ、種類を出力
  - 左の子と右の子を持つ
  - 再帰的に高さを計算

ALDS1_7_C: 木の巡回 (Tree Walk)
  - 先行順巡回(Preorder): 根→左→右
  - 中間順巡回(Inorder): 左→根→右
  - 後行順巡回(Postorder): 左→右→根
  - 3種類の巡回結果を出力

ALDS1_7_D: 木の再構築 (Reconstruction of a Tree)
  - Preorder(先行順)とInorder(中間順)から木を復元
  - Postorder(後行順)の巡回結果を出力
  - 分割統治法で再帰的に構築

Chapter 8: 二分探索木

ALDS1_8_A: 二分探索木 I (Binary Search Tree I)
  - insert命令でキーを挿入
  - print命令でinorder巡回とpreorder巡回の結果を出力
  - 二分探索木条件: 左部分木 ≤ 根 < 右部分木

ALDS1_8_B: 二分探索木 II (Binary Search Tree II)
  - find命令を追加: キーが存在するか判定
  - 探索操作を実装
  - 平均時間計算量: O(log N)

ALDS1_8_C: 二分探索木 III (Binary Search Tree III)
  - delete命令を追加: キーを持つ節点を削除
  - 3つのケース: 葉/片側の子/両側の子
  - 後続節点(successor)を使って削除

ALDS1_8_D: Treap (Treap)

  - 二分探索木とヒープを組み合わせた平衡木
  - keyとpriorityの2つの値を持つ
  - 回転操作で平衡を保つ

Chapter 9: ヒープ

ALDS1_9_A: 完全二分木 (Complete Binary Tree)
  - 配列表現された完全二分木
  - 親・左の子・右の子のインデックス関係
  - parent(i)=i/2, left(i)=2i, right(i)=2i+1

ALDS1_9_B: 最大ヒープ構築 (Maximum Heap)
  - max-heap条件: 親 ≥ 子
  - buildMaxHeapで配列をヒープ化
  - maxHeapify操作で部分木をヒープ化

ALDS1_9_C: 優先度付きキュー (Priority Queue)
  - insert: 要素を挿入
  - extract: 最大キーを持つ要素を取り出し
  - max-heapを使って実装

ALDS1_9_D: ヒープソート (Heap Sort)
  - max-heapを使った効率的なソート
  - buildMaxHeap→繰り返しextractMax
  - 時間計算量: O(N log N)、不安定ソート

Chapter 10: 動的計画法

ALDS1_10_A: フィボナッチ数列 (Fibonacci Number)
  - fib(n) = fib(n-1) + fib(n-2)
  - メモ化で計算量を削減
  - 時間計算量: O(N)

ALDS1_10_B: 連鎖行列積 (Matrix Chain Multiplication)
  - N個の行列の積の計算順序を最適化
  - スカラー乗算回数を最小化
  - 動的計画法で最適な分割点を求める

ALDS1_10_C: 最長共通部分列 (Longest Common Subsequence)
  - 2つの列の最長共通部分列の長さを求める
  - dp[i][j]: X[0..i]とY[0..j]のLCS長
  - 時間計算量: O(M×N)

ALDS1_10_D: 最適二分探索木 (Optimal Binary Search Tree)
  - 探索コストの期待値が最小となる二分探索木
  - 各キーの探索確率を考慮
  - 動的計画法で最適な構造を求める

Chapter 11: グラフ I

ALDS1_11_A: グラフ表現 (Graph)
  - 隣接リスト形式から隣接行列形式へ変換
  - N×Nの行列で辺の有無を表現
  - 空間計算量: O(V²)

ALDS1_11_B: 深さ優先探索 (Depth First Search)
  - できるだけ深く探索する戦略
  - 発見時刻(d)と完了時刻(f)を記録
  - スタック(再帰)を使って実装

ALDS1_11_C: 幅優先探索（Breadth First Search）
- できるだけ浅く、層ごとに探索
- 始点から各頂点への最短距離を求める
- キューを使って実装

ALDS1_11_D: 連結成分分解（Connected Components）
- SNSの友達関係で到達可能性を判定
- Union-Find木を使って素集合を管理
- 経路圧縮で効率化

Chapter 12: グラフ II（最短経路・最小全域木）

ALDS1_12_A: 最小全域木（Minimum Spanning Tree）
- プリム法で全頂点を含む最小コストの木を構築
- 訪問済み頂点から最小コストの辺を選択
- 時間計算量: O(V²)

ALDS1_12_B: 単一始点最短経路 I（Single Source Shortest Path I）
- ダイクストラ法で始点から各頂点への最短経路を求める
- 距離が最小の未訪問頂点を順番に確定
- 時間計算量: O(V²)

ALDS1_12_C: 単一始点最短経路 II（Single Source Shortest Path II）
- ダイクストラ法の高速化版
- min-heapを使って最小距離の頂点を効率的に取得
- 時間計算量: O((V+E) log V)

Chapter 13: 発見的探索

ALDS1_13_A: 8クイーン問題（8 Queens Problem）
- 8×8チェス盤に8個のクイーンを配置
- どのクイーンも他を襲撃できない配置を求める
- バックトラックで全探索、衝突検出配列で高速化

ALDS1_13_B: 8パズル（8 Puzzle）
- 3×3の盤面で空白を動かしてゴール状態を目指す
- 双方向BFSで最短手数を求める
- ハッシュテーブルで状態管理

ALDS1_13_C: 15パズル（15 Puzzle）
- 4×4の盤面で空白を動かしてゴール状態を目指す
- A*探索でヒューリスティック関数を使用
- マンハッタン距離を評価関数に利用

補足: 入出力・実装メモ（ファイル別）

ALDS1_1_A: 入力=配列長と数列 / 出力=各ステップの配列 / 主要構造=配列
【コード例】
```
while(j>=0 && A[j]>v){
  A[j+1] = A[j];
  j--;
}
A[j+1] = v;
```

ALDS1_1_B: 入力=2整数 / 出力=gcd / 主要構造=再帰 or ループ
【コード例】
```
int gcd(int x, int y){
  if(y == 0) return x;
  return gcd(y, x % y);
}
```


ALDS1_1_C: 入力=nとn個の整数 / 出力=素数の個数 / 主要構造=ループ
【コード例】
```
int is_prime(int n){
  for(int i=2; i*i<=n; i++){
    if(n%i==0) return 0;
  }
  return 1;
}
```

ALDS1_1_D: 入力=価格列 / 出力=最大利益 / 主要構造=最小値の更新
【コード例】
```
int maxProfit = 0, minPrice = price[0];
for(int i=1; i<n; i++){
  maxProfit = max(maxProfit, price[i] - minPrice);
  minPrice = min(minPrice, price[i]);
}
```

ALDS1_2_A: 入力=数列 / 出力=整列結果と交換回数 / 主要構造=配列
【コード例】
```
for(int i=n-1; i>=0; i--){
  if(A[i] < A[i-1]){
    swap(&A[i], &A[i-1]);
    flag = 1;
  }
}
```

ALDS1_2_B: 入力=数列 / 出力=整列結果と交換回数 / 主要構造=配列
【コード例】
```
for(int i=0; i<n; i++){
  int minj = i;
  for(int j=i; j<n; j++){
    if(A[j] < A[minj]) minj = j;
  }
  swap(&A[i], &A[minj]);
}
```

ALDS1_2_C: 入力=カード列 / 出力=各ソート結果と安定性 / 主要構造=構造体配列
【コード例】
```
typedef struct { int value; char suit; int id; } Card;
// ソート後に元のidで安定性を検証
```

ALDS1_2_D: 入力=数列 / 出力=gap列と比較回数と結果 / 主要構造=配列
【コード例】
```
for(int gap=n/2; gap>0; gap/=2){
  insertionSort(A, n, gap);
}
```

ALDS1_3_A: 入力=逆ポーランド式 / 出力=計算結果 / 主要構造=スタック
【コード例】
```
if(op == '+') push(pop() + pop());
else if(op == '-'){ int a=pop(), b=pop(); push(b-a); }
else if(op == '*') push(pop() * pop());
```

ALDS1_3_B: 入力=プロセス列と量子 / 出力=完了時刻 / 主要構造=キュー
【コード例】
```
if(p.time <= q) done = 1;
else{
  currentTime += q;
  p.time -= q;
```

```
    enqueue(&p);
  }
```

ALDS1_3_C: 入力=操作列 / 出力=リスト内容 / 主要構造=双方向連結リスト
【コード例】
```
Node *node = (Node*)malloc(sizeof(Node));
node->next = head->next;
node->prev = head;
head->next->prev = node;
head->next = node;
```

ALDS1_3_D: 入力=断面図の文字列 / 出力=総面積と各水たまり / 主要構造=スタック
【コード例】
```
if(s[i]=='\\') push(stack, i);
else if(s[i]=='/'){
  if(!empty(stack)){
    int j = pop(stack);
    area += i - j - 1;
  }
}
```

ALDS1_4_A: 入力=配列SとT / 出力=一致数 / 主要構造=線形探索
【コード例】
```
int count = 0;
for(int i=0; i<q; i++){
  for(int j=0; j<n; j++){
    if(T[i] == S[j]){ count++; break; }
  }
}
```

ALDS1_4_B: 入力=ソート済み配列SとT / 出力=一致数 / 主要構造=二分探索
【コード例】
```
int l=0, r=n;
while(l < r){
  int m = (l+r)/2;
  if(S[m] < key) l = m+1;
  else r = m;
}
```

ALDS1_4_C: 入力=insert/find命令 / 出力=yes/no / 主要構造=ハッシュ表
【コード例】
```
#define HASH_SIZE 100001
Node *table[HASH_SIZE] = {NULL};
int h = key % HASH_SIZE;
// ハッシング + チェーン法
```

ALDS1_4_D: 入力=荷物重量と台数 / 出力=最小積載量 / 主要構造=二分探索判定
【コード例】
```
int ok=1, count=1, sum=0;
for(int i=0; i<n; i++){
  if(sum + w[i] <= mid) sum += w[i];
  else{ count++; sum = w[i]; }
}
```

ALDS1_5_A: 入力=数列と問い合わせ / 出力=yes/no / 主要構造=再帰全探索
【コード例】
```
void solve(int i, int sum){
  if(i == n){ if(sum == q) found=1; return; }
  solve(i+1, sum);
  solve(i+1, sum + A[i]);
```



```
  }
```

ALDS1_5_B: 入力=数列 / 出力=整列結果と比較回数 / 主要構造=分割統治
【コード例】
```
merge(A, left, mid, right){
  // 2つのソート済み部分配列をマージ
  L[] = A[left..mid], R[] = A[mid+1..right]
  // 小さい順に戻す
}
```

ALDS1_5_C: 入力=深さN / 出力=頂点座標 / 主要構造=再帰
【コード例】
```
void koch(double x1, double y1, double x2, double y2, int n){
  if(n == 0){
    printf("%.10f %.10f\n", x2, y2);
    return;
  }
  // 線分を3等分して再帰
```

ALDS1_5_D: 入力=数列 / 出力=反転数 / 主要構造=マージソート
【コード例】
```
// マージ時に左配列の要素が右配列より大きい場合
if(L[i] > R[j]) inversions += (mid-i+1);
```

ALDS1_6_A: 入力=数列 / 出力=整列結果 / 主要構造=頻度配列
【コード例】
```
int C[MAX_K] = {0};
for(int i=0; i<n; i++) C[A[i]]++;
for(int j=1; j<=max; j++) C[j] += C[j-1];
// 後ろから配列に戻す
```

ALDS1_6_B: 入力=数列 / 出力=partition後の列 / 主要構造=配列
【コード例】
```
int x = A[r];
int i = p-1;
for(int j=p; j<r; j++){
  if(A[j] <= x) swap(A[++i], A[j]);
}
swap(A[++i], A[r]);
return i;
```

ALDS1_6_C: 入力=カード列 / 出力=整列結果と安定性 / 主要構造=クイックソート
【コード例】
```
quicksort(A, p, r){
  if(p < r){
    int q = partition(A, p, r);
    quicksort(A, p, q-1);
    quicksort(A, q+1, r);
  }
}
```

ALDS1_6_D: 入力=重み列 / 出力=最小コスト / 主要構造=サイクル分解
【コード例】
```
// 各サイクルについて最小値か全体最小を使う
int cost1 = minInCycle + sumOfCycle;
int cost2 = minGlobal + sumOfCycle + minInCycle;
minCost += min(cost1, cost2);
```

ALDS1_7_A: 入力=親子関係 / 出力=節点情報 / 主要構造=隣接リスト
【コード例】

```
  void dfs(int u, int d){
    depth[u] = d;
    for(int i=0; i<children[u].size(); i++){
      dfs(children[u][i], d+1);
    }
  }
```

ALDS1_7_B: 入力=二分木情報 / 出力=各節点情報 / 主要構造=配列で子管理

【コード例】

```
  struct Node{
    int left, right;
    int parent;
  } node[MAX];
```

ALDS1_7_C: 入力=二分木情報 / 出力=3巡回結果 / 主要構造=再帰

【コード例】

```
  preorder(u){ print(u); preorder(left[u]); preorder(right[u]); }
  inorder(u){ inorder(left[u]); print(u); inorder(right[u]); }
  postorder(u){ postorder(left[u]); postorder(right[u]); print(u); }
```

ALDS1_7_D: 入力=preorder/inorder / 出力=postorder / 主要構造=再帰分割

【コード例】

```
  Node* build(int pl, int pr, int il, int ir){
    int root = preorder[pl];
    int m = find(root, inorder[il..ir]);
    node->left = build(pl+1, pl+1+m-il, il, m-1);
    node->right = build(pl+1+m-il, pr, m+1, ir);
    return node;
  }
```

ALDS1_8_A: 入力=命令列 / 出力=巡回結果 / 主要構造=BST

【コード例】

```
  Node* insert(Node* node, int key){
    if(node == NULL) return createNode(key);
    if(key < node->key) node->left = insert(node->left, key);
    else node->right = insert(node->right, key);
    return node;
  }
```

ALDS1_8_B: 入力=命令列 / 出力=yes/noと巡回 / 主要構造=BST

【コード例】

```
  Node* find(Node* node, int key){
    if(node == NULL) return NULL;
    if(key == node->key) return node;
    if(key < node->key) return find(node->left, key);
    return find(node->right, key);
  }
```

ALDS1_8_C: 入力=命令列 / 出力=巡回結果 / 主要構造=BST+削除

【コード例】

```
  Node* deleteNode(Node* node, int key){
    if(node == NULL) return NULL;
    if(key < node->key) node->left = deleteNode(node->left, key);
    else if(key > node->key) node->right = deleteNode(node->right, key);
    else{
      // 削除処理: 子なし/子1つ/子2つの3ケース
    }
    return node;
  }
```


ALDS1_8_D: 入力=命令列 / 出力=巡回結果 / 主要構造=Treap

【コード例】

```
  // 優先度でヒープ条件、キーでBST条件を保つ
  if(leftPriority > rightPriority) rotateRight(node);
  else rotateLeft(node);
```

ALDS1_9_A: 入力=配列 / 出力=親子情報 / 主要構造=配列木

【コード例】

```
  int parent(i) { return i/2; }
  int left(i) { return 2*i; }
  int right(i) { return 2*i+1; }
```

ALDS1_9_B: 入力=配列 / 出力=最大ヒープ / 主要構造=heapify

【コード例】

```
  void maxHeapify(int i){
    int l = left(i), r = right(i), largest = i;
    if(l < n && H[l] > H[largest]) largest = l;
    if(r < n && H[r] > H[largest]) largest = r;
    if(largest != i){
      swap(&H[i], &H[largest]);
      maxHeapify(largest);
    }
  }
```

ALDS1_9_C: 入力=命令列 / 出力=extract結果 / 主要構造=最大ヒープ

【コード例】

```
  int extractMax(){
    int max = H[0];
    H[0] = H[--n];
    maxHeapify(0);
    return max;
  }
```

ALDS1_9_D: 入力=配列 / 出力=整列結果 / 主要構造=ヒープソート

【コード例】

```
  buildMaxHeap();
  for(int i=n-1; i>0; i--){
    swap(&H[0], &H[i]);
    n--;
    maxHeapify(0);
  }
```

ALDS1_10_A: 入力=n / 出力=fib(n) / 主要構造=DP配列

【コード例】

```
  dp[0]=0; dp[1]=1;
  for(int i=2; i<=n; i++){
    dp[i] = dp[i-1] + dp[i-2];
  }
```

ALDS1_10_B: 入力=行列次元 / 出力=最小コスト / 主要構造=DP表

【コード例】

```
  for(int l=2; l<=n; l++){
    for(int i=1; i<=n-l+1; i++){
      int j = i+l-1;
      for(int k=i; k<j; k++){
        cost = m[i][k]*m[k+1][j] + dp[i][k] + dp[k+1][j];
        dp[i][j] = min(dp[i][j], cost);
      }
    }
  }
```

ALDS1_10_C: 入力=複数の文字列ペア / 出力=LCS長 / 主要構造=DP表
【コード例】
```
for(int i=1; i<=m; i++){
  for(int j=1; j<=n; j++){
    if(X[i]==Y[j]) dp[i][j] = dp[i-1][j-1]+1;
    else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
  }
}
```

ALDS1_10_D: 入力=キーと確率 / 出力=最小期待コスト / 主要構造=DP表
【コード例】
```
for(int l=2; l<=n; l++){
  for(int i=1; i<=n-l+1; i++){
    int j=i+l-1;
    for(int k=i; k<=j; k++){
      int cost = dp[i][k] + dp[k+1][j] + sum_p[i..j];
      dp[i][j] = min(dp[i][j], cost);
    }
  }
}
```

ALDS1_11_A: 入力=隣接リスト / 出力=隣接行列 / 主要構造=行列
【コード例】
```
for each edge (u,v):
  matrix[u][v] = 1;  // 有向グラフ
```

ALDS1_11_B: 入力=隣接リスト / 出力=発見/完了時刻 / 主要構造=DFS
【コード例】
```
void dfs(int u){
  d[u] = ++timer;
  for each neighbor v:
    if(visited[v] == 0){
      visited[v] = 1;
      dfs(v);
    }
  f[u] = ++timer;
}
```

ALDS1_11_C: 入力=隣接リスト / 出力=最短距離 / 主要構造=BFS
【コード例】
```
queue<int> q;
dist[s] = 0;
q.push(s);
while(!q.empty()){
  int u = q.front(); q.pop();
  for each neighbor v:
    if(dist[v] == INF){
      dist[v] = dist[u] + 1;
      q.push(v);
    }
}
```

ALDS1_11_D: 入力=辺と問合せ / 出力=yes/no / 主要構造=Union-Find
【コード例】
```
int find(int x){
  if(parent[x] != x) parent[x] = find(parent[x]);
  return parent[x];
}
void unite(int x, int y){
```



```
  int px = find(x), py = find(y);
  if(px != py) parent[px] = py;
}
```

ALDS1_12_A: 入力=隣接行列 / 出力=最小全域木コスト / 主要構造=Prim
【コード例】
```
minCost[1] = 0;
for(int i=0; i<n; i++){
  int u = -1;
  for(int v=1; v<=n; v++){
    if(!visited[v] && (u==-1 || minCost[v]<minCost[u])) u=v;
  }
  visited[u] = 1;
  totalCost += minCost[u];
  for(int v=1; v<=n; v++){
    if(!visited[v] && w[u][v]<minCost[v]) minCost[v]=w[u][v];
  }
}
```

ALDS1_12_B: 入力=隣接リスト / 出力=各頂点の距離 / 主要構造=Dijkstra
【コード例】
```
dist[0] = 0;
for(int i=0; i<n; i++){
  int u = -1;
  for(int v=0; v<n; v++){
    if(!visited[v] && dist[v]!=-1 && (u==-1 || dist[v]<dist[u])) u=v;
  }
  visited[u] = 1;
  for each edge (u,v,w):
    if(dist[u]+w < dist[v]) dist[v] = dist[u]+w;
}
```

ALDS1_12_C: 入力=隣接リスト / 出力=各頂点の距離 / 主要構造=Dijkstra+heap
【コード例】
```
priority_queue<pair<int,int>> pq;
dist[0] = 0;
pq.push({0, 0});
while(!pq.empty()){
  int d = pq.top().first, u = pq.top().second;
  pq.pop();
  if(visited[u]) continue;
  visited[u] = 1;
  for each edge (u,v,w):
    if(dist[u]+w < dist[v]){
      dist[v] = dist[u]+w;
      pq.push({dist[v], v});
    }
}
```

ALDS1_13_A: 入力=既配置クイーン / 出力=盤面 / 主要構造=バックトラック
【コード例】
```
void check(int row){
  if(row == 8){ found(); return; }
  for(int col=0; col<8; col++){
    if(!conflicted(row, col)){
      place_queen(row, col);
      check(row+1);
      remove_queen(row, col);
    }
  }
```

```
  }

ALDS1_13_B: 入力=8パズル盤面 / 出力=最短手数 / 主要構造=双方向BFS
  【コード例】
  // 始点からと終点から同時にBFS
  // キューの状態をハッシュで管理
  while(forward_queue, backward_queue が空でない){
    if(状態が一致) break;
  }

ALDS1_13_C: 入力=15パズル盤面 / 出力=最短手数 / 主要構造=A*探索
  【コード例】
  priority_queue<pair<int,State>> pq; // (f値, 状態)
  f(state) = g(state) + h(state);  // g:現在までのコスト h:ヒューリ
  h(state) = マンハッタン距離



================================================================
合計: 49ファイル

基本的なアルゴリズムとデータ構造を網羅的に実装した教育用コレクション
================================================================
【実装上の落とし穴・注意点】
================================================================

【ソート関連】
- インデックスの起点: 0-originと1-originの混在に注意
  例: シェルソートのgap計算は実装依存

- 安定性: バブルソート(安定)と選択ソート(不安定)の違い
  → 同じ値の要素の相対順序が重要な場合は安定ソートが必須

- 計算量の実際: O(N²)でN=10^5の場合、10^10演算≒10秒
  → テスト時間制限内に入るかを事前に確認

- よくあるバグ:
  * ループ条件の不等号: for(i=0; i<n; i++) vs for(i=0; i<=n; i++)
  * swap処理忘れ: temp変数を使わずに直接代入

【探索関連】
- 二分探索前提: 配列は必ずソート済みである必要がある
  → ソートされていない配列での使用は致命的

- ハッシュテーブル: 衝突処理が重要
  → チェーン法 vs 開始アドレス法

- 計算量の見積もり:
  * 線形探索 O(N): N=10^7で実用的な範囲
  * 二分探索 O(log N): N=10^9でも実用的

- よくあるバグ:
  * 二分探索でのオーバーフロー: (left+right)/2 → (left+right)>>1
  * 終了条件の確認: leftとrightの関係が最後に正しいか

【グラフ関連】
- 隣接行列 vs 隣接リスト:
  * 隣接行列: N=300程度が上限（メモリが300×300=90000）
  * 隣接リスト: 要素が少ないグラフに最適

- DFS vs BFS:
```



```
  * DFS: 深さ優先（スタック、再帰）→ スタックオーバーフローに注意
  * BFS: 幅優先（キュー）→ メモリ効率が良い

- ダイクストラ法の前提: 負の辺がない
  → 負の辺がある場合はベルマンフォード法を使用

- よくあるバグ:
  * 訪問フラグの初期化忘れ
  * 無限ループ: visited配列をチェックせずに繰り返す

【DP関連】
- 配列サイズの確保: DP テーブルのインデックス範囲を明確に
  例: N=1000なら dp[1001][1001] (1-origin対応)

- 初期値設定: 最小値/最大値の区別
  * 最大値求め: dp[i][j] = -INF で初期化
  * 最小値求め: dp[i][j] = INF で初期化

- 遷移式の方向: 小さい方から大きい方へ計算
  → 逆方向で計算するとまだ計算されていない値を参照してしまう

- よくあるバグ:
  * INFの設定: INT_MAX/2 を使用（INT_MAX+INF でオーバーフロー回避）
  * 遷移漏れ: すべてのケースをカバーしているか確認

【その他】
- 配列外参照: ループ条件 i < n は明確に
  → off-by-one エラーは最も多いバグ

- 型の選択:
  * int: -2×10^9 ～ 2×10^9
  * long long: -9×10^18 ～ 9×10^18
  * 計算途中でオーバーフロー可能性を常に考慮

- スタックサイズ: 再帰が深い場合は増やす必要あり
  → DFS で N=10^6 はスタックオーバーフロー

================================================================
【各アルゴリズムの選択基準】
================================================================

【問題文から判断するキーワード】

「最も小さい/大きい」
  → ソートして1番目を選ぶ（貪欲法）
  → DP で最適解を求める
  → ヒープで効率化

「並び替える/整列する」
  → ソートアルゴリズムが必須
    * 速度重視: クイックソート or マージソート (N≦10^6)
    * 安定性重視: バブルソート or マージソート
    * 特殊な値: カウントソート (0～K の範囲)

「含まれるか/存在するか」
  → 線形探索: 小さなN、未ソート配列
  → 二分探索: ソート済み配列、N が大きい
  → ハッシュテーブル: 平均 O(1) で超高速

「経路/接続」
```

→ グラフアルゴリズムの出番
  * 距離が最短: ダイクストラ or BFS
  * 訪問順序: DFS
  * 連結性判定: Union-Find

「組み合わせ/部分集合」
→ ビット全探索: N≤20 (2^20 = 100万)
→ バックトラック: N≤15 程度
→ DP: N が大きい場合の最適化

「最大公約数/最小公倍数」
→ ユークリッドの互除法（高速）
→ 素因数分解は遅い（不使用推奨）

【入力サイズ別選択ガイド】

N ≤ 10^2 (100):
- どのアルゴリズムでも OK
- 実装の簡単さを優先（バブルソートなど）

N ≤ 10^3 (1000):
- O(N²) アルゴリズムが許容範囲
- DP の状態数が NxM でも OK

N ≤ 10^4:
- O(N log N) または O(N) が必須
- ソート基盤のアルゴリズムが安全

N ≤ 10^5:
- O(N log N) が標準
- クイックソート、マージソート必須
- グラフなら O(N + M)

N ≤ 10^6:
- O(N log N) がギリギリ
- より高速化が必要な場合がある
- メモリ効率も重要

N > 10^6:
- O(N) アルゴリズムのみ
- 定数倍の最適化が必須

【推奨アルゴリズム組み合わせ】

ソート + 探索:
→ ソート → 二分探索 (O(N log N + Q log N))

ソート + 二ポインタ:
→ ソート → 2本のポインタで線形走査 (O(N log N + N))

グラフ最短路:
→ N 小: Floyd-Warshall O(N³)
→ N 中: ダイクストラ O((N+M) log N)
→ N 大: BFS（非加重）O(N+M)

DP + グリーディ:
→ DP で最適構造を把握
→ グリーディで高速化

==========

【テンプレートコード】
==========

【挿入ソート】
```
for(int i = 1; i < n; i++){
  int v = a[i];
  int j = i - 1;
  while(j >= 0 && a[j] > v){
    a[j+1] = a[j];
    j--;
  }
  a[j+1] = v;
}
```

【バイナリサーチ】
```
int binarySearch(int x[], int n, int target){
  int left = 0, right = n - 1;
  while(left <= right){
    int mid = (left + right) / 2;
    if(x[mid] == target) return mid;
    if(x[mid] < target) left = mid + 1;
    else right = mid - 1;
  }
  return -1;  // 見つからない
}
```

【DFS（グラフ）】
```
void dfs(int u){
  visited[u] = 1;
  for each neighbor v of u:
    if(!visited[v]){
      dfs(v);
    }
}
```

【BFS（グラフ）】
```
void bfs(int start){
  queue<int> q;
  dist[start] = 0;
  q.push(start);
  while(!q.empty()){
    int u = q.front();
    q.pop();
    for each neighbor v of u:
      if(dist[v] == -1){
        dist[v] = dist[u] + 1;
        q.push(v);
      }
  }
}
```

【Union-Find】

```
int find(int x){
  if(parent[x] != x){
    parent[x] = find(parent[x]);  // 経路圧縮
  }
  return parent[x];
}

void unite(int x, int y){
  int px = find(x);
  int py = find(y);
  if(px != py){
    parent[px] = py;
  }
}
```

【DP（部分和問題）】
```
for(int i = 0; i < n; i++){
  for(int j = k; j >= a[i]; j--){
    dp[j] = dp[j] || dp[j - a[i]];
  }
}
```

【DP（LCS）】
```
for(int i = 1; i <= m; i++){
  for(int j = 1; j <= n; j++){
    if(s[i-1] == t[j-1]){
      dp[i][j] = dp[i-1][j-1] + 1;
    } else {
      dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }
  }
}
```

【スタック】
```
#define MAX 100
int stack[MAX];
int top = 0;

void push(int x){
  stack[top++] = x;
}

int pop(){
  return stack[--top];
}

int isEmpty(){
  return top == 0;
}
```

【キュー】
```
```



```
#define MAX 100
int queue[MAX];
int head = 0, tail = 0;

void enqueue(int x){
  queue[tail++] = x;
}

int dequeue(){
  return queue[head++];
}

int isEmpty(){
  return head == tail;
}
```

==========
【利用シーン分類】
==========

【ソート系】

アルゴリズム	安定性	推奨シーン
挿入ソート	安定	N<100, 教育
バブルソート	安定	安定性重視
選択ソート	不安定	最小移動数
シェルソート	不安定	N<10000
マージソート	安定	安定・確実
クイックソート	不安定	N大・高速
カウントソート	安定	値が0〜Kに限定

【探索系】

アルゴリズム	時間計算量	推奨シーン
線形探索	O(N)	小N, 未ソート
二分探索	O(log N)	ソート済み
ハッシュテーブル	O(1) 平均	高速検索要求

【グラフ系】

アルゴリズム	計算量	推奨シーン
DFS	O(V+E)	連結性, 経路探索
BFS	O(V+E)	最短路（無加重）
ダイクストラ	O((V+E)log V)	最短路（加重）
ベルマンフォード	O(VE)	負の辺対応
全点対最短路	O(V³)	小N全点間最短路
Prim	O(V²)	最小全域木
Kruskal	O(E log E)	最小全域木
Union-Find	O(α(N))	連結成分, グループ

【DP系】

アルゴリズム	状態数	推奨シーン

部分和問題	O(NX)	容量制限
ナップサック問題	O(NW)	容量W制限
LCS	O(NM)	文字列比較
編集距離	O(NM)	文字列距離
行列連鎖乗算	O(N³)	乗算順序最適化

【グリーディ系】

アルゴリズム	計算量	推奨シーン
活動選択問題	O(N log N)	時間割スケジュ
コイン両替問題	O(NK) or O(N)	最小枚数
区間スケジュル	O(N log N)	最大非重複選択

【全探索系】

アルゴリズム	計算量	推奨シーン
ビット全探索	O(2^N)	N≤20
バックトラック	O(N!)相当	N≤12, 剪定有
深さ優先探索(グラフ)	O(V+E)	パス列挙

==========
【問題分類と対応アルゴリズム】
==========

「数値計算」
- 最大公約数: ユークリッドの互除法
- 素数判定: 試し割り（√n まで）
- 素因数分解: 試し割り

「ソート・順序付け」
- 単純ソート: 挿入/バブル/選択
- 高速ソート: クイック/マージ/ヒープ
- 特殊ソート: カウント/基数

「探索・検索」
- 未ソート: 線形探索
- ソート済み: 二分探索
- 辞書: ハッシュテーブル

「データ構造」
- LIFO: スタック
- FIFO: キュー
- 階層: ツリー
- 任意接続: グラフ

「グラフ問題」
- 全頂点訪問: DFS/BFS
- 最短経路: ダイクストラ/BFS/Floyd-Warshall
- 最小全域木: Prim/Kruskal
- 連結成分: Union-Find/DFS

「最適化問題」
- 最大/最小値: DP/グリーディ
- 背パック問題: DP

- 文字列: LCS/編集距離
- 行列: 行列連鎖乗算

「列挙・組み合わせ」
- 全部分集合: ビット全探索（N≤20）
- 全順列: バックトラック（N≤10）
- グラフパス: DFS

==========