Cấu trúc dữ liệu

khái niệm con trỏ trong tài liệu này chỉ dùng để chỉ giá trị được trỏ đến trong mảng, không phải định nghĩa chính xác của con trỏ

pair

khai báo

```
pair<int,char> p;
```

truy vấn

```
p.first; gọi giá trị đầu tiên của pair

p.seccond; gọi giá trị thứ 2 của pair
```

struct

khai báo

```
struct(
    int a;
    char b,c;
    string e,f,g;
    pair<int,int> amogus;
) tên;
```

truy vấn

```
tên.a;
tên.b;
tên.g;
tên.amogus.first;
```

mảng

khai báo

```
int a[n];
double a[n][m]; mảng 2 chiều
pair<int,int> a[n][m][l]; mảng 3 chiều
```

Các hàm thông dụng

sort (sắp xếp) O(NlogN)

sort(a+1,a+1+n);

hoặc **sort(a,a+n);** nếu mảng bắt đầu từ i=0

binary search (Tìm kiếm nhị phân) O(logN)

Chỉ có thể sử dụng trên mảng đã sắp xếp

binary_search(a+1,a+1+n,val); Tìm giá trị val

lower_bound(a+1,a+1+n, val); Tìm giá trị đầu tiên >=val

upper_bound(a+1,a+1+n, val); Tìm giá trị đầu tiên >val

2 hàm trên sẽ trả về n+1 nếu không có giá trị nào trong mảng thỏa mãn

---

vector

khai báo

```
vector<int> v; //vector 1 chiều
vector<vector<int>> v; //vector 2 chiều
vector<vector<vector<int>> v; //vector 3 chiều
vector<int> v(n); //tạo vector với n phần tử có sẵn(có thể đặt giá trị vào các vị
trí)
vector<int> v(n,0); //tạo vector với n phần tử có sẵn với giá trị 0.
```

truy vấn

```
v[i] //truy vấn tại vị trí i+1(index i)
v.at(i) //truy vấn tại vị trí i-1 nhưng có thêm cảnh báo nếu
v.push_back(i); //chèn giá trị i vào cuối vector
v.pop_back(i); //loại bỏ phần tử cuối của vector
v.size(); //gọi kích thước của
v.empty(); //kiểm tra xem vector có rỗng không ->trả về true(có) hoặc false(không)


truy vấn các phần tử trong vector:
```

```
vòng for thông thường:
for(int i=0;i<v.size();i++){
    cout<<v.at(i)<<' ';
}

vòng for với pointer(con trỏ):
for(auto &i: v){
    cout<<i<<' ';
}
```

các hàm thông dụng

tương tự với mảng được triển khai bằng cú pháp:

hàm(v.begin(),v.end(),val);

ví dụ:

sort(v.begin(),v.end());

lower_bound(v.begin(),v.end(),i);

stack(LIFO)

vào trước ra sau

khai báo

```
stack<long long> stk;
stack<pair<long long, double>> stk;
```

truy vấn

```
stk.top(); //lấy giá trị trên cùng của stack
stk.push(); //chèn giá trị vào vị trí trên cùng của stack
stk.pop(); //loại bỏ giá trị trên cùng của set
stk.size(); //lấy độ dài của
stk.empty(); //kiểm tra stack có rỗng không tương tự như vector
```

Queue (FIFO)

vào trước ra trước

khai báo

```
queue<int> q;
queue<pair<int,string>> q;
queue<vector<pair<double,string>>> q;
```

truy vấn

```
queue không thể truy vấn bằng index(vị trí)
q.front(); //Lấy giá trị đầu tiên trong queue
q.back(); //lấy giá trị cuối cùng trong q
q.pop(); //loại bỏ giá trị đầ tiên trong q
q.size(); //kích thước q
q.empty(); //kiểm tra q có rỗng
```

deque (queue 2 đầu)

khai báo

```
deque<int> dq;
...
```

truy vấn

```
dq[i]; //gọi giá trị tại vị trí i+1(index i)
dp.at(i) //gọi giá trị tại vị trí i+1
dp.front(); //gọi giá trị đầu của dequeue
dp.back(); //gọi giá trị cuối của dq
dq.push_front(i); //chèn i vào đầu dq
dq.push_back(i); //chèn i vào cuối dq
dq.pop_front(); //loại bỏ phần tử đầu tiên của dq
dq.pop_back(); //loại bỏ phần tử cuối cùng của
dq.size();
dq.empty();
```

set

mảng lưu trữ:

- giá trị độc nhất
- tự động sắp

khai báo

```
set<string> st;
set<int, greater<int>> st; //với set sắp sếp từ lớn -> bé
```

truy vấn

set không thể truy vấn bằng vị trí(index)

```
st.insert(i); //chèn phần tử i vào set
st.erase(i); //xóa phần tử i khỏi set
st.clear(); //xóa toàn bộ phần tử của set
st.size();
st.empty();

duyệt set:
for(auto &i:set){
    cout<<i<<' ';
}
```

map

Map lưu trữ các giá trị bằng "khóa" thay vì vị trí. Map sẽ tự động sắp sếp các khóa theo thứ tự tăng

khai báo

```
map<int,int> mp;
map<int,int,greater<int>> mp; //map sắp sếp theo chiều lớn ->
unordered_map<int,int> uo_mp; //map không sắp sếp(độ phức tạp khi truy xuất ít
hơn)
```

truy vấn

```
mp[i]; //gọi giá trị của khóa
mp.at(i); //gọi giá trị của khóa i(không nên dùng)
mp.insert({i,j}); //chèn khóa i có giá trị j vào map
mp.erase(i); //xóa khóa i khỏi map
mp.clear(); //xóa toàn bộ
mp.count(i); //kiểm tra khóa i có tồn tại trong map không
mp.size();
mp.empty();

duyệt map:
for(auto &i:mp){
    cout<<i.first<<' '<<i.second<<'\n';
}
```

Đồ thị/Cây

khai báo

```
vector<vector<int>> adj; //khởi tạo mảng 2 chiều lưu trữ các nút con của nút tại
vị trí i
vector<vector<pair<int,int>>> adj; //đồ thị với trọng số cạnh
vector<vector<int>> adj;
vector<int> weight; //độ thị với trọng số điểm
```

truy vấn

nhập các cạnh {u,v}

```
//đồ thị vô hướng
adj[u].push_back(v);
adj[v].push_back(u);
//đồ thị có hướng
adj[u].push_back(v);
//đồ thị có trọng số đường đi w
adj[u].push_back({v,w}) //hoặc {w,v} nếu muốn sắp sếp theo trọng số
//đồ thị có trọng số điểm w
for(int i=0;i<n;i++){
    cin>>weight[i];
}
adj[u].push_back(v);
```

Thuật toán(giải thuật)

hàm lambda

đùng để thay đổi các hàm có sẵn ví dụ:

```
//dùng để sort vector pair theo phần tử thứ 2
vector<pair<int,int>> a;
sort(a.begin(),a.end())[](pair<int,int> &fst, pair<int,int> &sec){
    return fst.second<sec.second;
}
```

đệ quy

Gọi hàm trong hàm


```
int recursion(int n){
    retrurn recursion(n);
}
```

ví dụ thực tế: FIBONACHI

```
int fibo(int n){
    if(n==1) return 1;
    if(n==0) return 0;
    return fibo(n-1)+fibo(n-2);
}
```

Tham lam

cố gắng tìm cách tối ưu bằng cách duyệt qua tất cả các khả năng để tìm cách tối ưu nhất.

```
Đặt N là số hoạt động và
{I} là hoạt động thứ I  (1 <= I <= N)

Với mỗi {I}, xét S[I] và F[I] lần lượt là thời gian bắt đầu và kết thúc của hoạt
động đó.
Sắp xếp lại các hoạt động theo thứ tự tăng dần của thời gian kết thúc.
- Có nghĩa là, với I < J ta phải có F [I] <= F [J]
```

```
//  A là tập hợp các hoạt động được chọn
A = {1}
//  J là hoạt động cuối cùng được chọn
J = 1
For I = 2  to N
// ta có thể chọn I nếu nó là hoạt động cuối cùng
// việc chọn lựa đã hoàn thành
If S [I] >= F [J]
    // lựa chọn hoạt động 'I'
    A = A + {I}

    // hoạt động 'I' giờ trở thành hoạt động cuối cùng được lựa chọn
    J = I
Endif
Endfor

Return A
```

Tham lam thường dễ nghĩ ra, dễ cài đặt và chạy nhanh, nhưng không phải lúc nào cũng đúng. Khi bạn sử dụng duyệt hoặc quy hoạch động, nó giống như bạn đang di chuyển trên mặt đất an toàn. Còn đối với tham lam, thì giống như bạn đang đi trên một bãi mìn.

Không tồn tại một công thức chung nào cho việc áp dụng Tham lam, tuy nhiên, ta có thể nhìn ra thuật tham bằng việc phân tích các tính chất của bài toán, kinh nghiệm cũng như trực giác.

Một vài lưu ý nhỏ

- Nhưng bài tập mà có vẻ cực kỳ phức tạp (như TCSocks) có thể xem như là dấu hiệu để tiếp cận bằng phương pháp Tham lam.
- Nhưng bài toán mà dữ liệu đầu vào rất lớn (mà kể cả thuật toán có độ phức tạp vẫn không kịp) thường được giải bằng tham lam hơn là quay lui hoặc quy hoạch động.
- Mặc dù nó có vẻ rùng rợn, nhưng bạn nên nhìn thuật toán tham lam dưới đôi mắt của một thám tử chứ không phải là dưới cặp kính của một nhà toán học.

Quy hoạch động(dynamic progframing)

về cơ bản thì là dùng kết quả của các bước trước để tính kết quả của bước hiện tại

ví dụ:

FIBONACI:

```
int main()
{
    long long n, f[100010];
    cin >> n;
    f[0] = 0;
```

```
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];
    cout << f[n];
    return 0;
}
```

Bước cầu thang

đếm cách để bước lên cầu thang thứ n khi bước lên 1-> k bậc

```
int step(int n, int k){
    vector<int> dp(n+1,0);
    dp[0]=dp[1]=1;
    for(int i=2;i<=n;i++){
        for(int j=1;j<=k;j++){
            dp[i]+=dp[i-j];
            if(j==i) break;
        }
    }
    return dp[n];
}
```

bài toán knapsack

Đi mua hàng, có n món hàng với cân năng a1,a2,a3,...,an và giá trị v1,v2,v3,...vn. Bạn có thể mang theo m kg. Tính giá trị lớn nhất mà bạn mang theo được.

Unbounded Knapsack

Áp dụng với bài toán có thể lấy một đồ nhiều lần

```
// C++ program to implement
// unbounded knapsack problem using space optimised
#include <bits/stdc++.h>
using namespace std;

int knapSack(int capacity, vector<int> &val, vector<int> &wt) {

    // 1D matrix for tabulation.
    vector<int> dp(capacity + 1, 0);

    // Calculate maximum profit for each
    // item index and knapsack weight.
    for (int i = val.size() - 1; i >= 0; i--) {
        for (int j = 1; j <= capacity; j++) {

            int take = 0;
```

```
            if (j - wt[i] >= 0) {
                take = val[i] + dp[j - wt[i]];
            }
            int noTake = dp[j];

            dp[j] = max(take, noTake);
        }
    }

    return dp[capacity];
}

int main() {

    vector<int> val = {1, 1}, wt = {2, 1};
    int capacity = 3;
    cout << knapSack(capacity, val, wt);
}
```

0/1 knapsack

áp dụng với bài toán chỉ có thể lấy đồ 1 lần

```
#include <bits/stdc++.h>
using namespace std;

// Function to find the maximum profit
int knapsack(int W, vector<int> &val, vector<int> &wt) {

    // Initializing dp vector
    vector<int> dp(W + 1, 0);

    // Taking first i elements
    for (int i = 1; i <= wt.size(); i++) {

        // Starting from back, so that we also have data of
        // previous computation of i-1 items
        for (int j = W; j >= wt[i - 1]; j--) {
            dp[j] = max(dp[j], dp[j - wt[i - 1]] + val[i - 1]);
        }
    }
    return dp[W];
}

int main() {
    vector<int> val = {1, 2, 3};
    vector<int> wt = {4, 5, 1};
    int W = 4;

    cout << knapsack(W, val, wt) << endl;
    return 0;
}
```

Xử lí đồ thị/cây:

BFS(duyệt theo chiều rộng)

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// BFS for a single connected component
void bfsConnected(vector<vector<int>>& adj, int src, vector<bool>& visited,
vector<int>& res) {
    queue<int> q;
    visited[src] = true;
    q.push(src);

    while (!q.empty()) {
        int curr = q.front();
        q.pop();
        res.push_back(curr);

        // visit all the unvisited
        // neighbours of current node
        for (int x : adj[curr]) {
            if (!visited[x]) {
                visited[x] = true;
                q.push(x);
            }
        }
    }
}

// BFS for all components (handles disconnected graphs)
vector<int> bfs(vector<vector<int>>& adj) {
    int V = adj.size();
    vector<bool> visited(V, false);
    vector<int> res;

    for (int i = 0; i < V; i++) {
        if (!visited[i])
            bfsConnected(adj, i, visited, res);
    }
    return res;
}
void addEdge(vector<vector<int>>& adj, int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main() {
```

```
    int V = 6;
    vector<vector<int>> adj(V);

    // creating adjacency list
    addEdge(adj, 1, 2);
    addEdge(adj, 0, 3);
    addEdge(adj, 2, 0);
    addEdge(adj, 5, 4);

    vector<int> res = bfs(adj);

    for (int i : res)
        cout << i << " ";
}
```

DFS(duyệt theo chiều sâu)

```
#include <iostream>
#include <vector>

using namespace std;

void dfsRec(vector<vector<int>> &adj,
vector<bool> &visited, int s, vector<int> &res) {

    visited[s] = true;

    res.push_back(s);

    // Recursively visit all adjacent
    // vertices that are not visited yet
    for (int i : adj[s])
        if (visited[i] == false)
            dfsRec(adj, visited, i, res);
}

vector<int> dfs(vector<vector<int>> &adj) {
    vector<bool> visited(adj.size(), false);
    vector<int> res;
    // Loop through all vertices
    // to handle disconnected graph
    for (int i = 0; i < adj.size(); i++) {
        if (visited[i] == false) {
            dfsRec(adj, visited, i, res);
        }
    }

    return res;
}

void addEdge(vector<vector<int>>& adj, int u, int v) {
```

```
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main() {
    int V = 6;
    vector<vector<int>> adj(V);

    // creating adjacency list
    addEdge(adj, 1, 2);
    addEdge(adj, 0, 3);
    addEdge(adj, 2, 0);
    addEdge(adj, 5, 4);

    vector<int> res = dfs(adj);

    for (auto it : res)
        cout << it << " ";

    return 0;
}
```

Dijkstra

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

vector<int> dijkstra(vector<vector<pair<int,int>>>& adj, int src) {

    int V = adj.size();

    // Min-heap (priority queue) storing pairs of (distance, node)
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;

    vector<int> dist(V, INT_MAX);

    // Distance from source to itself is 0
    dist[src] = 0;
    pq.emplace(0, src);

    // Process the queue until all reachable vertices are finalized
    while (!pq.empty()) {
        auto top = pq.top();
        pq.pop();

        int d = top.first;
        int u = top.second;
```

```
        // If this distance not the latest shortest one, skip it
        if (d > dist[u])
            continue;

        // Explore all neighbors of the current vertex
        for (auto &p : adj[u]) {
            int v = p.first;
            int w = p.second;

            // If we found a shorter path to v through u, update it
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.emplace(dist[v], v);
            }
        }
    }

    // Return the final shortest distances from the source
    return dist;
}


int main() {
    int src = 0;

    vector<vector<pair<int,int>>> adj(5);
    adj[0] = {{1,4}, {2,8}};
    adj[1] = {{0,4}, {4,6}, {2,3}};
    adj[2] = {{0,8}, {3,2}, {1,3}};
    adj[3] = {{2,2}, {4,10}};
    adj[4] = {{1,6}, {3,10}};

    vector<int> result = dijkstra(adj, src);

    for (int d : result)
        cout << d << " ";
    cout << "
";

    return 0;
}
```