Thuật toán ứng dụng

BÀI TẬP LẬP TRÌNH

Dành cho Trợ giảng thực hành

SOICT — HUST

02/02/2020

Lưu ý không show toàn bộ code cho sinh viên, chỉ cần chữa chi tiết phần thuật toán xử lý chính của bài

Outline

01. ADD (Thuận)

- Cho hai số a và b, hãy viết chương trình bằng C/C++ tính số $c = a + b$
- Lưu ý giới hạn: $a, b < 10^{19}$ dẫn đến c có thể vượt quá khai báo `long long`

Thuật toán (Mô tả thuật toán giải bài và những lưu ý cần thiết)

- Chỉ cần khai báo a, b, c kiểu `unsigned long long`, trường hợp tràn số chỉ xảy ra khi a, b có 19 chữ số và c có 20 chữ số

1. Tách $a = a1 \times 10 + a0$
2. Tách $b = b1 \times 10 + b0$
3. Tách $a0 + b0 = c1 \times 10 + c0$
4. In ra liên tiếp $a1 + b1 + c1$ và $c0$

Code (chỉ cần đoạn code chính thể hiện thuật toán)

```cpp
int main() {
    unsigned long long a,b,c;
    cin >> a>>b;

    unsigned long long a0 = a % 10;
    unsigned long long a1 = (a-a0) / 10;
    unsigned long long b0 = b % 10;
    unsigned long long b1 = (b-b0) /10;
    unsigned long long c0 = (a0+b0) % 10;
    unsigned long long c1 = (a0+b0-c0) / 10;
    c1 = a1 + b1 + c1;
    if (c1>0) cout << c1;
    cout << c0;
    return 0;
}
```

01. SUBSEQMAX (Thuận)

- Cho dãy số $s = \langle a_1, \ldots, a_n \rangle$
- một dãy con từ i đến j là $s(i,j) = \langle a_i, \ldots, a_j \rangle$, $1 \leq i \leq j \leq n$
- với trọng số $w(s(i,j)) = \sum_{k=i}^{j} a_k$
- Yêu cầu: tìm dãy con có trọng số lớn nhất
- `http://www.spoj.com/problems/MAXSUMSU/`

Ví dụ

- dãy số: -2, 11, -4, 13, -5, 2
- Dãy con có trọng số cực đại là 11, -4, 13 có trọng số 20

Có bao nhiêu dãy con?

- Số lượng cặp (i,j) với $1 \leq i \leq j \leq n$
- $\binom{n}{2} + n$
- Thuật toán trực tiếp!

Thuật toán trực tiếp — $\mathcal{O}(n^3)$

- Duyệt qua tất cả $\binom{n}{2} + n = \frac{n^2+n}{2}$ dãy con

```
public long algo1(int[] a){
  int n = a.length;
  long max = a[0];
  for(int i = 0; i < n; i++){
    for(int j = i; j < n; j++){
      int s = 0;
      for(int k = i; k <= j; k++)
        s = s + a[k];
      max = max < s ? s : max;
    }
  }
  return max;
}
```

Thuật toán tốt hơn — $\mathcal{O}(n^2)$

- Quan sát: $\sum_{k=i}^{j} a[k] = a[j] + \sum_{k=i}^{j-1} a[k]$

```
public long algo2(int[] a){
  int n = a.length;
  long max = a[0];
  for(int i = 0; i < n; i++){
    int s = 0;
    for(int j = i; j < n; j++){
      s = s + a[j];
      max = max < s ? s : max;
    }
  }
  return max;
}
```

Thuật toán Chia để trị

- Chia dãy thành 2 dãy con tại điểm giữa $s = s_1 :: s_2$
- Dãy con có trọng số cực đại có thể
  - nằm trong s_1 hoặc
  - nằm trong s_2 hoặc
  - bắt đầu tại một vị trí trong s_1 và kết thúc trong s_2
- Code Java:

```
private long maxSeq(int i, int j){
  if(i == j) return a[i];
  int m = (i+j)/2;
  long ml = maxSeq(i,m);
  long mr = maxSeq(m+1,j);
  long maxL = maxLeft(i,m);
  long maxR = maxRight(m+1,j);
  long maxLR = maxL + maxR;
  long max = ml > mr ? ml : mr;
  max = max > maxLR ? max : maxLR;
  return max;
}
public long algo3(int[] a){
  int n = a.length;
  return maxSeq(0,n-1);
}
```

Chia để trị — $\mathcal{O}(n \log n)$

```java
private long maxLeft(int i, int j){
  long maxL = a[j];
  int s = 0;
  for(int k = j; k >= i; k--){
    s += a[k];
    maxL = maxL > s ? maxL : s;
  }
  return maxL;
}
private long maxRight(int i, int j){
  long maxR = a[i];
  int s = 0;
  for(int k = i; k <= j; k++){
    s += a[k];
    maxR = maxR > s ? maxR : s;
  }
  return maxR;
}
```

Thuật toán Quy hoạch động

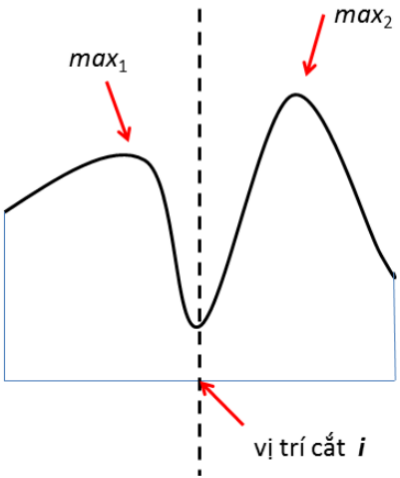
- Thiết kế hàm tối ưu:
  - Đặt s_i là trọng số của dãy con có trọng số cực đại của dãy $a_1, \ldots, a_i$ mà kết thúc tại a_i
- Công thức Quy hoạch động:
  - $s_1 = a_1$
  - $s_i = \max\{s_{i-1} + a_i, a_i\}, \forall i = 2, \ldots, n$
  - Đáp án là $\max\{s_1, \ldots, s_n\}$
- Độ phức tạp thuật toán là n (thuật toán tốt nhất!)

Quy hoạch động — $\mathcal{O}(n)$

```java
public long algo4(int[] a){
  int n = a.length;
  long max = a[0];
  int[] s = new int[n];
  s[0] = a[0];
  max = s[0];
  for(int i = 1; i < n; i++){
    if(s[i-1] > 0) s[i] = s[i-1] + a[i];
    else s[i] = a[i];
    max = max > s[i] ? max : s[i];
  }
  return max;
}
```

02. SIGNAL (TungTT)

- Cho một dãy tín hiệu độ dài n có độ lớn lần lượt là $a_1, a_2, ..., a_n$ và một giá trị phân tách b.
- Một tín hiệu được gọi là phân tách được khi tồn tại một vị trí i $(1 < i < n)$ sao cho $max\{a_1, .., a_{i-1}\} - a_i \geq b$ và $max\{a_{i+1}, .., a_n\} - a_i \geq b$
- Tìm vị trí i phân tách được sao cho $max\{a_1, .., a_{i-1}\} - a_i + max\{a_{i+1}, .., a_n\} - a_i$ đạt giá trị lớn nhất.
- In ra giá trị lớn nhất đó. Nếu không tồn tại vị trí phân tách được thì in ra giá trị -1.



max_2
max_1
vị trí cắt ***i***

Thuật toán

- Chuẩn bị mảng $maxPrefix[i] = max\{a_1, .., a_i\}$.
- Chuẩn bị mảng $maxSuffix[i] = max\{a_i, .., a_n\}$
- Duyệt qua hết tất cả các vị trí i $(1 < i < n)$. Với mỗi vị trí kiểm tra xem liệu đó có phải là vị trí phân tách được hay không bằng cách kiểm tra $maxPrefix[i-1] - a[i] >= b$ và $maxSuffix[i+1] - a[i] >= b$.
- Lấy max của giá trị $maxPrefix[i-1] - a_i + maxSuffix[i+1] - a_i$ tại các vị trí i thoả mãn.
- Độ phức tạp thuật toán O(n).

Cài đặt

```
int main() {
    const int MAX = 1e9 + 5;
    int n, b;

    cin >> n >> b;
    vector < int > a(n + 2, 0);
    vector < int > max_prefix(n + 2, 0);
    vector < int > max_suffix(n + 2, 0);
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    // khoi tao gia tri max o bien
    max_prefix[0] = -MAX; max_suffix[n + 1] = -MAX;

    // tinh max_prefix
    for (int i = 1; i <= n; i++) {
        max_prefix[i] = max(max_prefix[i - 1], a[i]);
    }
```

Cài đặt

```
    // tinh max_suffix
    for (int i = n; i >= 1; i--) {
        max_suffix[i] = max(max_suffix[i + 1], a[i]);
    }

    // tinh ket qua
    int ans = -1;
    for (int i = 2; i < n; i++) {
        // Kiem tra vi tri i
        if (max_prefix[i - 1] - a[i] >= b &&
            max_suffix[i + 1] - a[i] >= b) {
            // lay ket qua
            ans = max(ans, max_prefix[i - 1] - a[i] +
                                max_suffix[i + 1] - a[i])
        }
    }
    cout << ans << endl;
```

02. LOCATE

HùngĐM

- Cho T test, mỗi test gồm 2 bản đồ kích thước LxC, thể hiện cùng một địa điểm tại 2 thời điểm khác nhau.
- Mỗi bản đồ biểu diễn bởi các số 0, 1. 1 ứng với vị trí có vật thể bay (có thể là chim hoặc chiến đấu cơ), 0 là vị trí không có vật thể nào.
- Biết rằng tất cả các chiến đấu cơ trên bản đồ di chuyển theo cùng một quy luật.
- Tính số chiến đấu cơ tối đa có thể xuất hiện trong cả hai bản đồ.
- Biết rằng: $1 \leq L, C \leq 1000$. Tổng số các số 1 không quá 10000.

Thuật toán

- Tổng số các số 1 không quá 10000 nên có thể lưu toạ độ các đỉnh 1 của mỗi trạng thái vào 2 mảng.
- So sánh từng cặp đỉnh của mỗi mảng để đếm số khoảng cách có thể.

Code

```
const int N = 1010;

int n, m;
vector<pair<int, int>> a, b;
int cnt[N * 2][N * 2];

int main() {
    //freopen("test.in", "r", stdin);
    ios_base::sync_with_stdio(0); cin.tie(0);
    int tc;
    cin >> tc;
    while (tc--) {
        memset(cnt, 0, sizeof cnt);
        cin >> n >> m;
        a.clear(); b.clear();
        ...
```

Code

```
...
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            int u;
            cin >> u;
            if (u == 1) a.push_back({i, j});
        }
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            int u;
            cin >> u;
            if (u == 1) b.push_back({i, j});
        }
    }
...
```

Code

```
        ...
        for (auto u : a) {
            for (auto v : b) {
                pair<int, int> w =
                    {u.first - v.first + N,
                     u.second - v.second + N};
                cnt[w.first][w.second]++;
            }
        }
        int res = 0;
        for (int i = 0; i < N * 2; i++) {
            res = max(res,
                  *max_element(cnt[i], cnt[i] + N * 2))
        }
        cout << res << '\n';
    }
    return 0;
}
```

02. POSTMAN (HieuNT)

- Một nhân viên giao hàng cần nhận các kiện hàng tại trụ sở công ty ở vị trí $x = 0$, và chuyển phát hàng đến n khách hàng, được đánh số từ 1 đến n.
- Người khách thứ i ở vị trí x_i và cần nhận m_i kiện hàng.
- Nhân viên giao hàng chỉ có thể mang theo tối đa k kiện hàng mỗi lần.
- Nhân viên giao hàng xuất phát từ trụ sở, nhận một số kiện hàng và di chuyển theo đại lộ để chuyển phát cho một số khách hàng. Khi giao hết các kiện hàng mang theo, nhân viên lại quay trở về trụ sở và lặp lại công việc nói trên cho đến khi chuyển phát hết tất cả các kiện hàng.
- Sau khi giao xong, nhân viên cần quay lại công ty để nộp hóa đơn của ngày hôm đó.
- Giả thiết là: tốc độ di chuyển là 1 đơn vị khoảng cách trên một đơn vị thời gian. Thời gian nhận hàng ở trụ sở công ty và thời gian bàn giao hàng cho khách được coi là bằng 0.
- Giả sử thời điểm nhân viên giao hàng bắt đầu công việc là 0.
- Tìm cách hoàn thành công việc tại thời điểm sớm nhất.

Thuật toán

- Nhận xét: Vì công ty nằm ở vị trí $x = 0$ và thời gian nhận hàng ở công ty bằng 0 nên ta có thể chia khách hàng thành 2 tập: $x < 0$ và $x > 0$. Kết quả bằng tổng thời gian chuyển trong 2 tập
- Thuật toán
  1. Phân chia khách hàng thành 2 tập: $x < 0$ và $x > 0$.
  2. Với mỗi tập khách hàng, ta sắp xếp các khách hàng theo khoảng cách từ vị trí của họ đến trụ sở công ty.
  3. Nhân viên giao hàng sẽ phát từ khách hàng xa nhất trong tập, nếu còn dư số kiện hàng sẽ phát tiếp cho khách hàng liền kề đó.
- Độ phức tạp: $O(n)$

Code

```
long long calSegment(pair<int, int> p[], int np)
{
  long long res = 0;
  int cur = 0;
  for(int i = 1; i <= np; i++) {
    if(p[i].second > 0) {
      if(cur >= p[i].second){
        // Du so kien de phat
        cur -= p[i].second;
      } else {
        // Khong du so kien de phat
        p[i].second -= cur;
        int times = (p[i].second - 1)/ k + 1;
        res += 2ll * abs(p[i].first) * times;
        cur = times * k - p[i].second;
      }
    }
  }
  return res;
}
```

Code

```
const int N = 1002;
typedef pair<int, int> pii;
int n, nn, np, k, x, m;
long long ans = 0;
pii negCus[N], posCus[N];

int main()
{
  cin >> n >> k;
  nn = np = 0;
  for(int i = 1; i <= n; i++) {
    cin >> x >> m;
    // Chia thanh 2 tap khach hang
    if(x < 0) negCus[++nn] = make_pair(x, m);
    else posCus[++np] = make_pair(x, m);
}
  ...
```

Code

```cpp
    ...
    // Sap xep khach hang trong tap theo khoang cach
    sort(negCus + 1, negCus + nn + 1);
    sort(posCus + 1, posCus + np + 1, greater<pii>());

    // Tinh khoang thoi gian nho nhat voi moi tap
    long long negSeg = calSegment(negCus, nn, k);
    long long posSeg = calSegment(posCus, np, k);
    ans = negSeg + posSeg;
    cout << ans;
    return 0;
}
```

04. WATERJUG-BFS (LongTV)

- Có hai bình đựng nước, một bình có dung tích a lít, bình còn lại dung tích b lít. Tìm cách để có thể đong được chính xác c lít nước.
- Đầu vào: Dòng đầu tiên cho một số nguyên $T \leq 1000$ là số lượng test, với mỗi test sẽ gồm 3 số nguyên dương $a, b, c \leq 10^3$

Thuật toán

Nhận xét

- Kí hiệu (X, Y) tương ứng với bình 1 có X lít nước, bình 2 có Y lít nước, với mỗi trạng thái như vậy ta có thể thực hiện các cách đổ sau:
    1. Làm trống 1 bình, $(X, Y) \to (0, Y)$, đổ hết bình 1.
    2. Đổ đầy một bình, $(0, 0) \to (X, 0)$, đổ đầy bình 1.
    3. Đổ nước từ một bình sang một bình còn lại cho đến khi một trong hai bình đầy hoặc hết nước, $(X, Y) \to (X + d, Y - d)$

Thuật toán

1. Bắt đầu với trạng thái (0, 0), chúng ta chạy thuật toán duyệt theo chiều rộng (BFS) với mỗi đỉnh duyệt là một trạng thái đã có, và các đỉnh khác sẽ được sinh ra bằng 3 cách đổ nước bên trên từ những đỉnh trước đó.
2. Thuật toán sẽ dừng khi đã tìm được trạng thái có chứa c lít trong đó hoặc đã duyệt hết các trạng thái có thể sinh ra.

Code

```
#include <bits/stdc++.h>
#define pii pair<int, int>
#define mp make_pair
using namespace std;

// level is number of steps to (X,Y) State
map<pii, int> level;
// queue to maintain states
queue<pii> q;

// Changing state of jugs from (u1, u2) to (a,b)
void Pour(int a, int b, pii u){
    // if this state isn't visited
    if (level[{ a, b }] == 0)
    {
        // Save
        q.push({a, b });
        level[{a, b}] = level[{u.first, u.second}] +
    }
}
```

Code

```
void BFS(int a, int b, int target)
{
    // Map is used to store the states, every
    bool isSolvable = false;
    level.clear();
    q = queue<pii>();
    // queue to maintain states
    // Initialing with initial state
    q.push({ 0, 0 });
    level[{0,0}] = 1;

    while (!q.empty()) {
        pii u = q.front(); // current state
        q.pop(); // pop off used state
        // if we reach solution state
        if(u.first==target||u.second==target){
            isSolvable = true;
            cout<<level[{u.first, u.second}]-1;
            break;
        }
```

Code

```
        Pour(u.first, b, u); // fill Jug2
        Pour(a, u.second, u);// fill Jug1
        Pour(u.first, 0, u);// Empty Jug2
        Pour(0, u.second, u);// Empty Jug1

        for (int ap=0; ap<=max(a, b); ap++) {
            // pour amount ap from Jug2 to Jug1
            int c = u.first + ap;
            int d = u.second - ap;
            // check if this state is possible
            if ((c == a && d >= 0)
                    || (d == 0 && c <= a))
                Pour(c, d, u);
            // Pour amount ap from Jug 1 to Jug2
            c = u.first - ap;
            d = u.second + ap;
            // check if this state is possible
            if ((c == 0 && d <= b)
                    || (d == b && c >= 0))
                Pour(c, d, u);
```

Code

```
            }
      }

      // No, solution exists if ans=0
      if (!isSolvable)
            cout << -1 << endl;
}
int main()
{
      int T;
      cin >> T;
      while (T > 0)
      {
            int Jug1, Jug2, target;
            cin >> Jug1 >> Jug2 >> target;
            BFS(Jug1, Jug2, target);
            T --;
      }
      return 0;
}
```

02. HIST (DucLA)

- Có N cột với độ cao l_1, l_2, ...
- Tìm diện tích hình chữ nhật lớn nhất nằm gọn trong N cột này

Thuật toán

- Nhận xét: Một hình chữ nhật với hai biên là các cột thứ i và j có diện tích là $(j - i + 1) * min(l_i, ..., l_j)$
- Dễ dàng nhận thấy thuật toán $O(N^3)$: thử hết các cặp i và j và tính min 1 đoạn trong $O(N)$
- Nhận xét $min(l_i, ..., l_j) = min(min(l_i, ..., l_{j-1}), l_j)$
- Vậy min đoạn i đến j có thể cập nhật trong $O(1)$ khi i giữ nguyên và j tăng lên 1, ta có thuật toán $O(N^2)$

Thuật toán

- Góc nhìn khác: thay vì đi thử các bộ i và j, ta thử các chiều cao của hình chữ nhật, tức là nhân tử $min(l_i, ..., l_j)$
- Cụ thể: với mỗi cột i, ta xét xem nếu nó là cột có độ cao thấp nhất của một hình chữ nhật nào đó, thì chiều rộng của hình chữ nhật đó tối đa là bao nhiêu
- Ta đi tính các giá trị $left_i$ và $right_i$. Trong đó $left_i$ là vị trí của cột gần nhất bên trái i mà có độ cao nhỏ hơn cột i, tương tự đối với $right_i$ nhưng là ở bên phải
- Kết quả bài toán là max của các giá trị $(right_i - left_i - 1) * l_i$ với mọi i

Thuật toán

- Vấn đề còn lại là làm sao tính nhanh được các giá trị $left_i$ và $right_i$
- => Sử dụng stack
- Ví dụ với *left*, ta duyệt i tăng dần, luôn duy trì một stack chứa vị trí trước i và có độ cao nhỏ hơn cột i, trong đó các vị trí tăng dần và top của stack lưu vị trí lớn nhất. Như vậy $left_i$ chính là giá trị trên đỉnh stack khi ta duyệt đến i.
- Cập nhật stack: khi nào mà $l_i <= l_{stack_{top}}$ thì pop phần tử đỉnh stack. Sau đó push i vào stack
- Độ phức tạp $O(N)$, vì một phần tử chỉ vào và ra stack tối đa 1 lần.

Code

```
template<class RandomIt>
vector<int> calc_extend(RandomIt first, RandomIt last) {
    vector<int> result;
    stack<RandomIt> s;
    for (RandomIt it = first; it != last; ++it) {
        while (!s.empty() && *s.top() >= *it) s.pop();
        result.push_back(it - (s.empty() ? (first - 1) : s.top()));
        s.push(it);
    }
    return result;
}

int main() {
    int n;
    while ((cin >> n) && n) {
        vector<int> height(n);
        for (int i = 0; i < n; ++i) cin >> height[i];
        vector<int> L = calc_extend(height.begin(), height.end());
        vector<int> R = calc_extend(height.rbegin(), height.rend());
        long long result = 0;
        for (int i = 0; i < n; ++i) {
            result = max(result, 1LL * (L[i] + R[n - i - 1] - 1) * height[i]);
        }
        cout << result << endl;
    }
}
```

02. REROAD (QuangLM)

- Cho N đoạn đường, đoạn thứ i có loại nhựa đường là t_i.
- Định nghĩa một phần đường là một dãy liên tục các đoạn đường được phủ cùng loại nhựa phủ t_k và bên trái và bên phải phần đường đó là các đoạn đường (nếu tồn tại) được phủ loại nhựa khác.
- Độ gập ghềnh của đường bằng tổng số lượng phần đường.
- Mỗi thông báo bao gồm 2 số là số thứ tự đoạn đường được sửa và mã loại nhựa được phủ mới.
- Sau mỗi thông báo, cần tính độ gập ghềnh của mặt đường hiện tại.

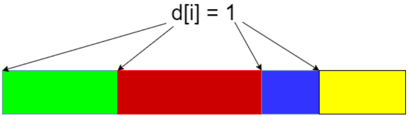
Ví dụ

Đoạn đường ban đầu với độ gập ghềnh là 4

Đoạn đường sau khi update với độ gập ghềnh là 6

Thuật toán

- Gọi d[i] là mảng nhận giá trị 1 nếu $a[i] \neq a[i-1]$ và giá trị 0 trong trường hợp ngược lại
- Nhận thấy mỗi phần đường có một và chỉ một phần tử bắt đầu, số lượng phần đường (hay độ gập ghềnh) chính là số lượng phần tử bắt đầu.
- Nói cách khác thì độ gập ghềnh $= \sum_{i=1}^{n} d[i]$
- Nhận thấy với mỗi lần đổi 1 phần tử i trong mảng a thì ta chỉ thay đổi giá trị của nhiều nhất là 2 phần tử trong mảng d đó là $d[i]$ và $d[i+1]$



Code

```
int bat_dau(int u) {
    if (u == 1) return 1;
    return t[u] != t[u - 1];
}
int main() {
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> t[i];
    int kq = 0;
    for (int i = 1; i <= N; i++) kq += bat_dau(i);
    cin >> Q;
    for (int i = 1; i <= Q; i++) {
        cin >> p >> c;
        kq -= bat_dau(p);
        if (p < N) kq -= bat_dau(p + 1);
        t[p] = c;
        kq += bat_dau(p);
        if (p < N) kq += bat_dau(p + 1);
        cout << kq << endl;
    }
}
```

03. TSP (Thai9cdb)

- Cho một đồ thị đầy đủ có trọng số.
- Tìm một cách di chuyển qua mỗi đỉnh đúng một lần và quay về đỉnh xuất phát sao cho tổng trọng số các cạnh đi qua là nhỏ nhất (chu trình hamilton nhỏ nhất).

Thuật toán 1

- Mỗi cách di chuyển ứng với một hoán vị của *n* đỉnh.
- Xét hết các hoán vị và tìm nghiệm tốt nhất.
- Để xét hết các hoán vị, có thể dùng đệ quy - quay lui hoặc thuật toán sinh kế tiếp.

Code 1

//i là số đỉnh đã đi qua, sum là tổng trọng số đường đi. Mảng x[.] toàn cục lưu danh sách các đỉnh đi qua theo thứ tự. Mảng c[.][.] toàn cục lưu ma trận trọng số

```
void duyet(int i,int sum){
    if (i==n+1){
        if (sum+c[x[n]][1] < best)
            best=sum+c[x[n]][1];
        return;
    }
    for (int j=2;j<=n;++j)
        if (!mark[j]){
            if (sum+c[x[i-1]][j]<best){
                mark[j]=1;
                x[i]=j;
                duyet(i+1,sum+c[x[i-1]][j]);
                mark[j]=0;
            }
        }
}
```

Thuật toán 2

- Để ý cây đệ quy (tạo ra từ quá trình duyệt) có rất nhiều cây con giống nhau. Ta có thể chỉ duyệt một lần và lưu trữ lại kết quả.
- Trạng thái duyệt là danh sách các đỉnh đã đi qua và đỉnh hiện tại đang đứng (hay danh sách các số đã xuất hiện và số cuối cùng trong phần hoán vị đã xây dựng). Hai cây con giống nhau nếu trạng thái tại nút gốc của chúng giống nhau.
- Ta có thể lưu trữ nghiệm tối ưu cho từng trạng thái để không phải duyệt lại nhiều lần. Sử dụng kỹ thuật bitmask để lưu trữ thuận tiện hơn.

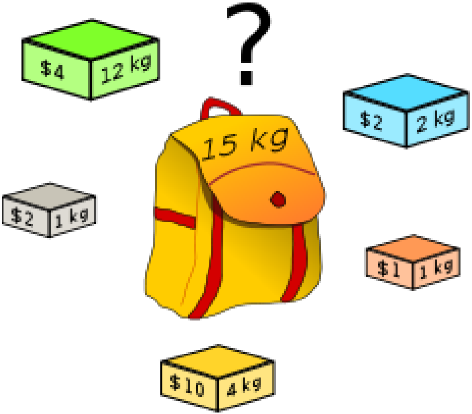
Code 2

//p là đỉnh đang đứng, X là tập các đỉnh đã đi qua

```
int duyet(int X, int p){
    if (__builtin_popcount(X) == n) return c[p][0];
    if (save[X][p] != -1) return save[X][p];
    int ans = 2e9;
    for (int s = 0; s < n; ++s)
        if ((X >> s & 1) == 0){
            ans = min(ans, c[p][s]
            + duyet(1 << s | X, s));
        }
    save[X][p] = ans;
    return ans;
}
```

03. KNAPSAC (Thai9cdb)

- Cho một cái túi có thể chứa tối đa khối lượng M và n đồ vật. Mỗi đồ vật có khối lượng và giá trị của nó.
- Tìm cách lấy một số đồ vật sao cho tổng khối lượng không vượt quá M và tổng giá trị lớn nhất có thể.


?
15 kg
$4
12 kg
$2
2 kg
$2
1 kg
$1
1 kg
$10
4 kg

Thuật toán 1

- Mỗi cách chọn lấy các đồ vật tương ứng với một dãy nhị phân độ dài n. Bit thứ i là 0/1 tương ứng là không lấy/có lấy đồ vật thứ i.
- Xét hết các xâu nhị phân độ dài n và tìm nghiệm tốt nhất.
- Để xét hết các xâu nhị phân độ dài n, có thể dùng đệ quy - quay lui hoặc chuyển đổi giữa thập phân với nhị phân.
- Độ phức tạp $O(2^n \times n)$

Code 1a

```
void duyet(int i,int sum,int val){
    if (i>n){
        if (val>best) best=val;
        return;
    }
    duyet(i+1,sum,val);//bit 0
    if (sum+m[i]<=M)
        duyet(i+1,sum+m[i],val+v[i]);//bit 1
}
```

Code 1b

```
main(){
    cin >> n >> M;
    for (int i = 0; i < n; ++i)
    cin >> m[i] >> v[i];
    int ans = 0;
    for (int mask = 1 << n; mask--; ){//mask = 0..2^n
        int sumM = 0, sumV = 0;
        for (int i = 0; i < n; ++i)
            if (mask >> i & 1){//bit thu i = 1
                sumM += m[i];
                sumV += v[i];
            }
        if (sumM <= M) ans = max(ans, sumV);
    }
    cout << ans;
}
```

Thuật toán 2

- Chia tập đồ vật làm hai phần A và B. Mỗi cách chọn lấy các đồ vật tương ứng với một cách lấy bên A kết hợp với một cách lấy bên B.
- Ý tưởng chính ở đây là lưu trữ hết các cách lấy bên B và sắp xếp trước theo một thứ tự. Sau đó với mỗi cách lấy bên A, ta có thể tìm kiếm nghiệm tối ưu bên B một cách nhanh chóng.
- Giả sử cách lấy tối ưu là lấy m_A bên A và m_B bên B, ta sẽ xét tuần tự từng m_A một và tìm kiếm nhị phân m_B.
- Độ phức tạp $\mathrm{O}(2^A \times \log(2^B) + 2^B) = \mathrm{O}(2^{n/2} \times n)$ nếu chọn $|\mathrm{A}| = |\mathrm{B}| = \mathrm{n} / 2$

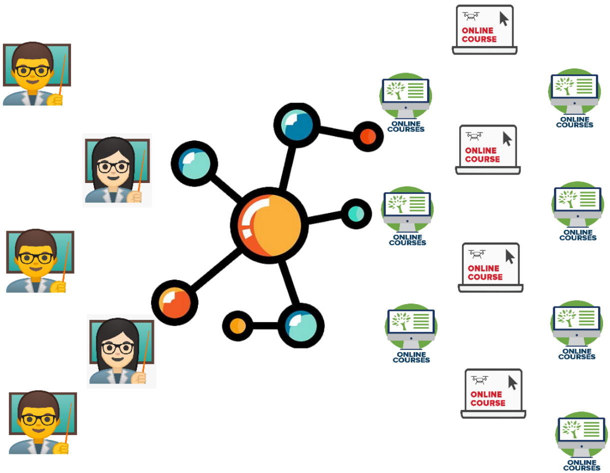
Code 2

//S1, S2 là tập các cách lấy bên A, bên B. maxV[j] là max(S2[0..j].v)
//S1 và S2 được tính bằng đệ quy - quay lui

```
    int ans = -1e9;
    for(int i=0;i<S1.size();++i){
        int L = 0, H = S2.size()-1, j = -1;
        while (L<=H){
            int m = (L+H)/2;
            if (S1[i].m + S2[m].m <= M){
                j = m;
                L = m+1;
            } else H = m-1;
        }
        if (j!=-1){
            ans = max(ans, S1[i].v + maxV[j]);
        }
    }
    cout << ans;
```

03. BCA (ngocbh)

- Có n khóa học và m giáo viên, mỗi giáo viên có danh sách các khóa có thể dạy.
- Có danh sách các khóa học không thể để cùng một giáo viên dạy do trùng giờ.
- Load của một giáo viên là số khóa phải dạy của giáo viên đó.
- **Yêu cầu:** Tìm cách xếp lịch cho giáo viên sao cho Load tối đa của các giáo viên là tối thiểu.



ONLINE COURSE
ONLINE COURSES
ONLINE COURSE
ONLINE COURSES
ONLINE COURSES
ONLINE COURSES
ONLINE COURSE
ONLINE COURSES
ONLINE COURSES
ONLINE COURSE
ONLINE COURSES

Thuật toán I

- Sử dụng thuật toán vét cạn, duyệt toàn bộ khóa học, xếp giáo viên dạy khóa học đó.
- Sử dụng thuật toán nhánh cận:
  - Chọn khóa học chưa có người dạy có số giáo viên dạy ít nhất để phân công trước.
  - Nếu phân công cho giáo viên A môn X, mọi môn học trùng lịch với môn X không thể được dạy bởi giáo viên A sau này.
  - Nếu maxLoad hiện tại lớn hơn minLoad tối ưu thu được trước đó thì không duyệt nữa.

Code

```
void arrange(int i) {
    if (i > n) {
        // Check result...
    } else {
        int courseID = -1;
        int totalTeacher = 999;
        // Find non-assigned course the least
        // number of teachers to assign first.
        // ...

        if (courseID == -1) return;

        int maxLoad = 0;
        // Find current maxLoad...
        if (maxLoad >= minLoad) return;
```

Code

```
        sjList[courseID].selected = true;
        for (int j=1; j<=m; j++) {
            if (sjList[courseID].teacher[j]) {
                bool tmp[31];
                for (int k=1; k<=n; k++) {
                    tmp[k] = sjList[k].teacher[j];
                    if (conflict[courseID][k]) {
                        sjList[k].teacher[j]=false;
                    }
                }
                schedule[j][courseID] = true;
                arrange(i+1);
                schedule[j][courseID] = false;
                for (int k=1; k<=n; k++)
                    sjList[k].teacher[j]=tmp[k];
            }
        }
        sjList[courseID].selected = false;
    }
}
```

03. BACP (PQD)

- The BACP is to design a balanced academic curriculum by assigning periods to courses in a way that the academic load of each period is balanced .
- There are N courses $1, 2, \ldots, N$ that must be assigned to M periods $1, 2, \ldots, M$. Each course i has credit c_i and has some courses as prerequisites. The load of a period is defined to be the sum of credits of courses assigned to that period.
- The prerequisites information is represented by a matrix $A_{N \times N}$ in which $A_{i,j} = 1$ indicates that course i must be assigned to a period before the period to which the course j is assigned. Given constants a, b. Compute the solution satisfying constraints
  - Total number of courses assigned to each period is greater or equal to a and smaller or equal to b
  - The maximum load for all periods is minimal

03. BACP (PQD)

- Input
  - Line 1 contains N and M $(2 \leq N \leq 16, 2 \leq M \leq 5)$
  - Line 2 contains $c_1, c_2, \ldots, c_N$
  - Line $i + 2(i = 1, \ldots, N)$ contains the i^{th} line of the matrix A
- Output: Unique line contains that maximum load for all periods of the solution found

Cài đặt

```
#include <bits/stdc++.h>
#define MAX_N 30
#define MAX_M 10

// input
int N, M;
int c[MAX_N];// c[i] is the credits of course i
int A[MAX_N][MAX_N];
// solution representation
int x[MAX_N];// x[c] is the period to which the course
        // assigned
int f_best;
int load[MAX_M];// load [p] is the load of period p
```

Cài đặt

```
int check(int v, int k){
    for(int i = 1; i <= k-1; i++){
        if(A[i][k] == 1){
                if(x[i] >= v) return 0;
        } else if(A[k][i] == 1){
            if(v >= x[i]) return 0;
        }
    }
    return 1;
}
void solution(){
    int max = load[1];
    for(int i = 2; i <= M; i++)
        max = max < load[i] ? load[i] : max;
    if(max < f_best){
        f_best = max;
    }
}
```

Cài đặt

```
void TRY(int k){
    for(int v = 1; v <= M; v++){
        if(check(v,k)){
                x[k] = v;
                load[v] += c[k];
                if(k == N) solution();
                else TRY(k+1);
                load[v] -= c[k];
        }
    }
}
void input(){
    scanf("%d%d",&N,&M);
    for(int i = 1; i <= N; i++)
        scanf("%d",&c[i]);
    for(int i = 1; i <= N; i++)
        for(int j = 1; j <= N; j++)
            scanf("%d",&A[i][j]);
}
```

Cài đặt

```
void solve(){
    for(int i = 1; i <= M; i++) load[i] = 0;
    f_best = 1000000;
    TRY(1);
    printf("%d\n",f_best);
}
int main(){
    input();
    solve();
}
```

03. CONTAINER (name)

03. CBUS (VUONGDX)

- Có n hành khách $1, 2, \ldots, n$, hành khách i cần di chuyển từ địa điểm i đến địa điểm $i + n$
- Xe khách xuất phát ở địa điểm 0 và có thể chứa tối đa k hành khách
- Cho ma trận c với $c(i, j)$ là khoảng cách di chuyển từ địa điểm i đến địa điểm j
- Tính khoảng cách ngắn nhất để xe khách phục vụ hết n hành khách và quay trở về địa điểm 0
- **Lưu ý:** Ngoại trừ địa điểm 0, các địa điểm khác chỉ được thăm tối đa 1 lần

Thuật toán I

- Sử dụng thuật toán vét cạn, thực hiện bằng thủ tục đệ quy để thử mọi thứ tự thăm
- Sử dụng *bitmask* s để lưu trạng thái các địa điểm đã được thăm, biến l lưu lại số lượng hành khách ở trên xe khách và biến v lưu đỉnh cuối cùng thuộc đường đi đang xét
- Tại mỗi lần gọi đệ quy, ta xét các địa điểm chưa được thăm i
  - Nếu $1 \leq i \leq n$
    - Nếu $l = k$, không thể thăm địa điểm i
    - Nếu $l < k$, cập nhật $l = l + 1$, thêm i vào đường đi hiện tại và gọi đệ quy
  - Nếu $n + 1 \leq i \leq 2n$
    - Nếu địa điểm $i - n$ đã được thăm, cập nhật $l = l - 1$, thêm i vào đường đi hiện tại và gọi đệ quy. Ngược lại, không thể thăm địa điểm i

Thuật toán II

- Sử dụng kỹ thuật nhánh cận để giảm số lần gọi đệ quy. Xe khách luôn cần đi qua $2 \times N + 1$ cạnh. Gọi f là độ dài đường đi hiện tại, r là số cạnh mà xe khách còn phải đi qua, c_{min} là độ dài cạnh nhỏ nhất, ta có công thức cận:

$$bound = f + r \times c_{min} \tag{1}$$

Code

```
void _try(int v) {
if (r == 1) {
    res = min(res, f + c[v][0]);
    return;
}

if (l < k) {
    for (int i = 1; i <= n; i++) {
        if (((s >> i) & 1) == 0) {
            f += c[v][i];
            r--;
            l += 1;
            s += (1 << i);
            bound = f + r * c_min;
            if (bound < res) {
                _try(i);
            }
            s -= (1 << i);
            l -= 1;
            r++;
```

Code

```
    for (int i = n + 1; i <= 2 * n; i++) {
        if (((s >> i) & 1) == 0) {
            if (((s >> (i - n)) & 1) == 1) {
                f += c[v][i];
                r--;
                l -= 1;
                s += (1 << i);
                bound = f + r * c_min;
                if (bound < res) {
                    _try(i);
                }
                s -= (1 << i);
                l += 1;
                r++;
                f -= c[v][i];
            }
        }
    }
}
```

03. CVRP (PQD)

- A fleet of K identical trucks having capacity Q need to be scheduled to deliver pepsi packages from a central depot 0 to clients $1, 2, \ldots, n$. Each client i requests $d[i]$ packages.
- Solution: For each truck, a route from depot, visiting clients and returning to the depot for deliverying requested pepsi packages such that:
  - Each client is visited exactly by one route
  - Total number of packages requested by clients of each truck cannot exceed its capacity
- Goal
  - Compute number R of solutions

03. CVRP (PQD)

- Input
  - Line 1: n, K, Q $(2 \leq n \leq 10, 1 \leq K \leq 5, 1 \leq Q \leq 20)$
  - Line 2: $d[1], ..., d[n]$ $(1 \leq d[i] \leq 10)$
- Output: $R \bmod 10^9 + 7$

Cài đặt

```
#include <bits/stdc++.h>
#define MAX_N 30
#define MAX_M 10

// input
int N, M;
int c[MAX_N];// c[i] is the credits of course i
int A[MAX_N][MAX_N];
// solution representation
int x[MAX_N];// x[c] is the period to which the course
        // assigned
int f_best;
int load[MAX_M];// load [p] is the load of period p
```

Cài đặt

```c
#include <stdio.h>
#define MAX 50
int n,K,Q;
int d[MAX];
int c[MAX][MAX];

int x[MAX];// x[i] is the next point of i (i = 1,...,
           // in {0,1,...,n}
int y[MAX];// y[k] is the start point of route k
int load[MAX];
int visited[MAX];// visited[i] = 1 means that client
int segments;// number of segments accumulated
int nbRoutes;
```

Cài đặt

```
void input(){
    scanf("%d%d%d",&n,&K,&Q);
    for(int i = 1; i <= n; i++){
        scanf("%d",&d[i]);
    }
    d[0] = 0;
    for(int i = 0; i <= n; i++){
        for(int j = 0; j <= n; j++){
            scanf("%d",&c[i][j]);
        }
    }
}
```

Cài đặt

```
void solution(){
   for(int k = 1; k <= K; k++){
        int s = y[k];
        printf("route[%d]:  0 ",k);
        for(int v = s; v != 0; v = x[v]){
            printf("%d ",v);
        }
        printf("0\n");
    }
    printf("---------------------------\n");
}
```

Cài đặt

```
int checkX(int v,int k){
    if(v > 0 && visited[v]) return 0;
    if(load[k] + d[v] > Q) return 0;
    return 1;
}
int checkY(int v, int k){
    if(v == 0) return 1;
    if(load[k] + d[v] > Q) return 0;
    return !visited[v];
}
```

Cài đặt

```
void TRY_X(int s, int k){
 if(s == 0){
  if(k < K) TRY_X(y[k+1],k+1);
  return;
 }
 for(int v = 0; v <= n; v++){
  if(checkX(v,k)){
   x[s] = v; visited[v] = 1; load[k] += d[v]; segments
   if(v > 0) TRY_X(v,k);
   else{
    if(k == K){
     if(segments == n+nbRoutes) solution();
     }else  TRY_X(y[k+1],k+1);
    }
    segments--;  load[k] -= d[v];  visited[v] = 0;
   }
  }
}
```

Cài đặt

```
void TRY_Y(int k){
 for(int v = (y[k-1]==0 ? 0 : y[k-1] + 1); v <= n; v+
  if(checkY(v,k)){
   y[k] = v; visited[v] = 1; load[k] += d[v];
   if(v > 0) segments += 1;
   if(k < K) TRY_Y(k+1);
   else{
    nbRoutes = segments;
    TRY_X(y[1],1);
   }
   load[k] -= d[v]; visited[v] = 0;
   if(v > 0) segments -= 1;
  }
 }
}
```

Cài đặt

```
void solve(){
    for(int v = 1; v <= n; v++) visited[v] = 0;
    y[0] = 0;
    TRY_Y(1);
}
int main(){
    input();
    solve();
}
```

03. CVRP OPT (DucLA)

- A fleet of K identical trucks having capacity Q need to be scheduled to deliver pepsi packages from a central depot 0 to clients $1, 2, \ldots, n$. Each client i requests $d[i]$ packages.
- Problem: For each truck, a route from depot, visiting clients and returning to the depot such that:
  - Each client is visited exactly by one route
  - Total number of packages requested by clients of each truck cannot exceed its capacity
- Goal
  - Find a solution having minimal total travel distance

Thuật toán

- $K = 1, Q = INFINITY$: Chính là bài toán TSP
- $K = 1$: Là bài TSP nhưng có thêm chặn Q, thậm chí dễ hơn
- $K > 1$: Cần duyệt các cách phân tập $1..N$ thành K tập con khác rỗng, mỗi tập con chính là 1 bài TSP độc lập nhau.
- Mỗi phần tử có thể thuộc 1 trong K tập, ta duyệt hết K^N cách phân tập

Cài đặt

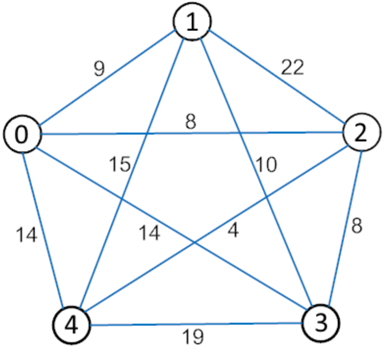
```
const int N = 13;
int n, k, q, d[N], c[N][N];
int f[1 << N], s[5], res = 1e9;

void go(int u, int m, int q, int g) {
    f[m] = std::min(f[m], g + c[u][0]);
    for (int i = 1; i <= n; ++i)
        if (q >= d[i] && !(m >> i - 1 & 1))
            go(i, m | 1 << i - 1, q - d[i], g + c[u][i]);
}

void opt(int u) {
    if (u == n) {
        int t = 0;
        for (int i = 0; i < k; ++i) t += f[s[i]];
        res = std::min(res, t);
    } else {
        for (int i = 0; i < k; ++i)
            if (f[s[i] | 1 << u] < res)
                s[i] ^= 1 << u, opt(u + 1), s[i] ^= 1 << u;
    }
}

int main() {
    scanf("%d %d %d", &n, &k, &q);
    for (int i = 1; i <= n; ++i) scanf("%d", d + i);
    for (int i = 0; i <= n; ++i)
        for (int j = 0; j <= n; ++j) scanf("%d", &c[i][j]);
    memset(f, 0x3F, sizeof f);
    go(0, 0, q, 0);
    opt(0);
    printf("%d", res);
}
```

03.TAXI(TungTT)

- Nêu ra lần đầu tiên năm 1930 về tối ưu hóa
- Dưới dạng bài toán "The Saleman Problem"

Phát biểu bài toán gốc

- Một tài xế Taxi xuất phát từ điểm 0, và nếu khoảng cách giữa hai điểm bất kỳ được biết thì đâu là đường đi ngắn nhất mà người Taxi có thể thực hiện sao cho đi hết tất cả các điểm mỗi điểm một lần để quay về lại điểm A.
- Đầu vào: khoảng cách giữa các điểm, tài xế Taxi xuất phát từ điểm 0, và có n điểm từ $1, 2, 3, ..n$ cần đi qua.
- Đầu ra: đường đi ngắn nhất $0 \rightarrow i \rightarrow j \ldots \rightarrow 0$
- Dưới dạng đồ thị: bài toán người lái taxi được mô hình hóa như một đồ thị vô hướng có trọng số, trong đó mỗi điểm đến là một đỉnh của đồ thị, đường đi từ một điểm đến điểm khác là khoảng cách hay chính là độ dài cạnh.



- Tổng quãng đường đi từ $0 \to 2 \to 4 \to 1 \to 3 \to 0 = 8 + 4 + 15 + 10 + 4 = 51$
- Tổng quãng đường đi từ $0 \to 1 \to 4 \to 2 \to 3 \to 0 = 9 + 15 + 4 + 8 + 4 = 40$

Ứng dụng

- Lập kế hoạch như bài toán Taxi là tối ưu trong quãng đường phục vụ, bài toán Người bán hàng,...
- Thiết kế vi mạch $\rightarrow$ Tối ưu về đường nối, điểm hàn
- Trong lĩnh vực phân tích gen sinh học
- Trong lĩnh vực du lịch

TAXI

- Có n hành khách được đánh số từ 1 tới n.
- Có $2n+1$ địa điểm được đánh số từ 0 tới $2n$
- Hành khách thứ i muốn đi từ địa điểm thứ i đến địa điểm thứ $i+n$
- Taxi xuất phát ở địa điểm thứ 0 và phải phục vụ n hành khách và quay lại địa điểm thứ 0 sao cho không điểm nào được đi lại 2 lần và tại một thời điểm chỉ có 1 hành khách được phục vụ.
- Cho khoảng cách giữa các địa điểm. Tính quãng đường nhỏ nhất mà tài xế Taxi phải đi.

Thuật toán

- Nhận xét với mỗi cách chọn đường đi ta sẽ ánh xạ về được một hoán vị từ 1 đến n.
- Ta duyệt hết $n!$ hoán vị.
- Với mỗi hoán vị tính toán khoảng cách phải di chuyển.
- Độ phức tạp thuật toán $O(n! * n)$, có thể cải tiến xuống $O(n!)$ hoặc thậm chí $O(2^n * n^2)$.

Công thức

- Gọi $c[i][j]$ là khoảng cách di chuyển từ địa điểm i đến địa điểm j
- Gọi một hoán vị có dạng $a_1, a_2, ..., a_n$
- Công thức :
$\sum_{i=1}^{n} c[a_i][a_i + n] + \sum_{i=1}^{n-1} c[a_i + n][a_{i+1}] + c[0][a_1] + c[a_n + n][0]$

Code

```
int calc(int a[]) {
    // tra ve chi phi di chuyen cua hoan vi a
    // tinh theo cong thuc da neu
}

int main() {
    Nhap n
    Nhap ma tran c[i][j]
    Khoi tao hoan vi a[] = {1, 2, ..., n}
    answer = INF
    while (1) {
        cost = calc(a)
        answer = min(answer, cost)
        if (a == {n, n - 1, ..., 1}) {
            break;
        }
        next_permutation(a)
    }
    In ra answer
}
```

04. PIE (VUONGDX)

- Có N cái bánh và $F + 1$ người.
- Mỗi cái bánh có hình tròn, bán kính r và chiều cao là 1.
- Mỗi người chỉ được nhận một miếng bánh từ một chiếc bánh.
- Cần chia bánh sao cho mọi người có lượng bánh bằng nhau (có thể bỏ qua vụn bánh).
- Tìm lượng bánh lớn nhất mỗi người nhận được.

Thuật toán

- Gọi $p[i]$ là số người ăn chiếc bánh thứ i. Lượng bánh mỗi người nhận được là $min_i\{\frac{V[i]}{p[i]}\}$ với $V[i]$ là thể tích của chiếc bánh thứ i.
- **Cách 1 - Tìm kiếm theo mảng p:** Tìm kiếm vét cạn mọi giá trị của p.
- **Cách 2 - Tìm kiếm theo lượng bánh mỗi người nhận được:** Thử từng kết quả, với mỗi kết quả, kiểm tra xem có thể chia bánh cho tối đa bao nhiêu người.
- **Tối ưu cách 2:** Sử dụng thuật toán tìm kiếm nhị phân để tìm kiếm kết quả.

Code

```
    sort(r, r + N);

        double lo = 0, hi = 4e8, mi;

        for(int it = 0; it < 100; it++){
            mi = (lo + hi) / 2;

            int cont = 0;

            for(int i = N - 1;
            i >= 0 && cont <= F; --i)
                cont += (int)
                floor(M_PI * r[i] / mi);

            if(cont > F) lo = mi;
            else hi = mi;
        }
```

04. AGGRCOW (quanglm)

- Có N chuồng bò và C con bò.
- Chuồng bò thứ i có tọa độ là x_i.
- Cần xếp các con bò vào các chuồng sao cho khoảng cách nhỏ nhất giữa 2 con bò bất kỳ là lớn nhất.

Thuật toán

- **Thuật toán 1:** Duyệt vét cạn từng con bò vào từng chuồng rồi tính khoảng cách ngắn nhất, $O(N^C \times C)$.
- **Thuật toán 2:** Duyệt giá trị kết quả bài toán d. Mỗi d, xếp các con bò vào chuồng 1 cách tham lam sao cho con bò sau cách con bò trước ít nhất d đơn vị. Nếu xếp đủ C con bò thì d là một giá trị hợp lệ. Tìm d lớn nhất. $O(max(x_i) \times N)$.
- **Thuật toán 3:** Tìm kiếm nhị phân với giá trị d. $O(\log max(x_i) \times N)$.

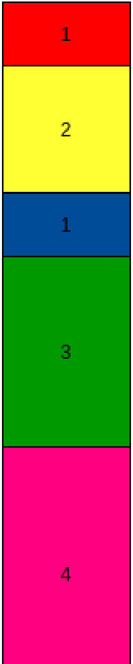
Code

```
    sort(x + 1, x + n + 1);
    int low = -1, high = (int)1e9 + 10;
    while (high - low > 1) {
        int mid = (low + high) / 2;
        int num = 0;
        int last = (int)-1e9;
        for (int i = 1; i <= n; i++) {
            if (x[i] >= last + mid) {
                num++;
                last = x[i];
            }
        }
        if (num >= C) low = mid;
        else high = mid;
    }
    cout << low << endl;
```

04. BOOKS1 (TungTT)

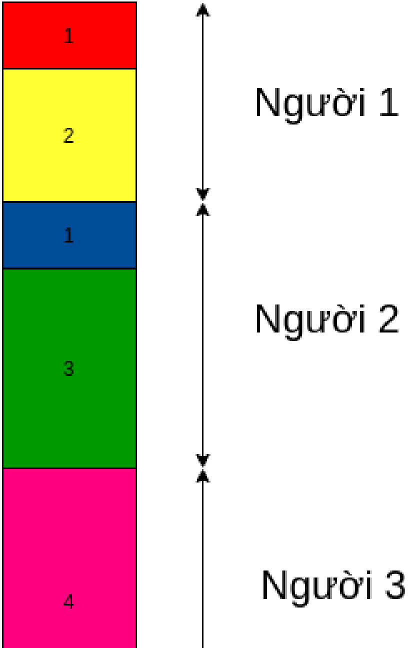
- Có m quyển sách, quyển sách thứ i dày p_i trang.
- Phải chia số sách trên cho đúng k người, mỗi người sẽ nhận được một đoạn sách liên tiếp nhau.
- In ra cách chia để số trang sách lớn nhất được nhận bởi một người là nhỏ nhất.
- Nếu có nhiều kết quả lớn nhất thì ưu tiên số sách nhận bởi người 1 là ít nhất, sau đó đến người 2, ...

Ví dụ



- Đầu vào có 5 quyển sách và phải chia số sách trên cho 3 người
- Mỗi quyển sách có độ dày như hình bên

Ví dụ



- Kết quả của bài toán là 4
- Có 2 cách chia để đạt được kết quả trên :
  1 2/1 3/4 hoặc
  1 2 1/3/4
- Cách chia như hình bên là kết quả của bài toán

Thuật toán 1

- Duyệt kết quả của bài toán từ nhỏ đến lớn, cố định số trang sách lớn nhất được chia bởi 1 người.
- Với mỗi kết quả ta đi kiểm tra có chia được cho đúng k người hay không bằng thuật toán tham lam.
- In ra kết quả ngay khi tìm được kết quả thỏa mãn
- Độ phức tạp thuật toán $O(MAX * n)$

Code 1

```
bool check(long long max_val) {
    vector < int > pos;
    long long sum = 0;
    for (int i = n; i >= 1; i--) {
        if (sum + a[i] <= max_val) {
            sum += a[i];
        } else {
            sum = a[i];
            if (a[i] > max_val) { return false; }
            pos.push_back(i);
        }
    }
    if (pos.size() >= k) { return false; }
    In kq
    return true;
}
```

Thuật toán 2

- Gọi *maxVal* là số trang lớn nhất được chia bởi 1 người.
- Nhận thấy nếu với giá trị $maxVal = x$ có thể chia dãy thành $\leq k$ đoạn thì với $maxVal = x + 1$ cũng có thể chia dãy thành $\leq k + 1$ đoạn với cách chia như cũ.
- Ta chặt nhị phân giá trị *maxVal*.
- Độ phức tạp thuật toán $O(\log MAX * n)$

Code 2

```
bool check(long long max_val) {
    // Giong voi ham o Code 1
}
int main() {
    int q; cin >> q;
    while (q--) {
        cin >> n >> k;
        for (int i = 1; i <= n; i++) { cin >> a[i]; }
        long long l = 0, r = MAX;
        while (r - l > 1) {
            long long mid = (l + r) >> 1;
            if (check(mid)) {
                r = mid;
            } else {
                l = mid;
            }
        }
        ** In kq tuong ung voi gia tri r **
    }
}
```

04. EKO (ngocbh)

- Cho n cái cây có chiều cao khác nhau $a_1, a_2, ...a_n$
- Có thể thực hiện một phát cắt độ cao h với tất cả các cây.
- Số lượng gỗ thu được là phần chóp của các cây cao hơn h.
- Tìm h nhỏ nhất có thể để số lượng gỗ thu được lớn hơn m.
- VD:
  - có 4 cây $20, 15, 10, 17$.
  - chọn $h = 15 \rightarrow$ số lượng gỗ thu được ở mỗi cây là $5, 0, 0, 2$. tổng là 7.
  - vậy ta thu được 7 mét gỗ.

Thuật toán

- **Thuật toán 1:** tìm tất cả các giá trị $h \in \{0, max(a[i])\}$. Với mỗi h, tính số lượng gỗ thu đượ. ĐPT: $O(max(a[i]) * n)$.
- **Thuật toán 2:** chặt nhị phân giá trị h.

Code

```
    long long count_wood(int height) {
        long long ret = 0;
        for (int i = 1; i <= n; i++)
            if ( a[i] > height )
                ret += a[i] - height;
        return ret;
    }
    int l = 0, r = max(r,a[i]);

    while (l < r-1) {
        int mid = (l+r)/2;
        if (count_wood(mid) >= m ) l = mid;
        else r = mid;
    }
    cout << l;
```

04. FIBWORDS (vuongdx)

- Dãy Fibonacci Words của xâu nhị phân được định nghĩa như sau:

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2), & \text{if } n \geq 2 \end{cases}$$

- Cho n và một xâu nhị phân p. Đếm số lần p xuất hiện trong $F(n)$ (các lần xuất hiện này có thể chồng lên nhau).
- Giới hạn: $0 \leq n \leq 100$, p có không quá 100000 ký tự, kết quả không vượt quá 2^{63}.

Thuật toán I

- **Thuật toán 1 - Vét cạn:** So sánh xâu p với mọi xâu $f(n)[i..(i + len(p))]$.
- **Thuật toán 2 - Chia để trị:** Xâu $f(n)$ gồm 2 xâu con là $f(n-1)$ và $f(n-2)$.
  - Đếm số lần p xuất hiện trong $f(n-1)$, $f(n-2)$.
  - Đếm số lần p xuất hiện ở đoạn giữa của xâu $f(n)$ (đoạn đầu của p là đoạn cuối của $f(n-1)$, đoạn cuối của p là đoạn đầu của $f(n-2)$).

Thuật toán II

- Đếm số lần p xuất hiện trong $f(i)$ với i nhỏ: Sử dụng **thuật toán 1**.
- Đếm số lần p xuất hiện ở đoạn giữa xâu $f(n)$:
  - Giả sử 2 xâu $f(i-1)$ và $f(i)$ có độ dài lớn hơn độ dài xâu p, $f(i-1)$ có dạng $x..a$, $f(i)$ có dạng $y..b$, trong đó x, y, a, b có độ dài bằng độ dài của p (x và a hay y và b có thể chồng lên nhau).
  - **Nhận xét 1:** $x = y$.
  - **Nhận xét 2:** Nếu $n \equiv i(mod 2)$ thì đoạn giữa của $f(n)$ là $..ax..$, ngược lại, đoạn giữa của $f(n)$ là $..bx..$.

Thuật toán III

- Cài đặt:
  - **void preprocessing()**: Tính trước các xâu fibonacci word, 2 xâu cuối cùng có độ dài không nhỏ hơn 10^5.
  - **long long count(string s, string p)**: Đếm số lần p xuất hiện trong s theo thuật toán 1.
  - **long long count(int n, string p)**: Đếm số lần p xuất hiện trong $f(n)$ theo thuật toán 2.
  - **long long solve(int n, string p)**:
    - Xử lý trường hợp $f(n)$ có độ dài nhỏ hơn độ dài của p.
    - Khởi tạo mảng c - $c[i]$ là số lần xuất hiện của p trong $f(i)$.
    - Sử dụng hàm count(s, p) để đếm số lần xuất hiện của p trong $f(i)$ và $f(i-1)$ với $f(i-1)$ là fibonacci word đầu tiên có độ dài không nhỏ hơn độ dài của p rồi lưu vào mảng c.
    - Sử dụng hàm count(s, p) để đếm số lần xuất hiện của p trong ax và bx, lưu vào mảng mc.
    - Sử dụng hàm count(n, p) để đếm số lần xuất hiện của p trong $f(n)$.

Code

```
long long solve(int n, string p) {
    int lp = p.size();
    if (n < n_prepare && l[n] < lp) {return 0;}
    for (int j = 0; j <= n; j++) {c[j] = -1;}
    int i = 1;
    while (l[i - 1] < lp) {i++;}
    c[i - 1] = count(f[i - 1], p);
    c[i] = count(f[i], p);
    string x = f[i].substr(0, lp - 1);
    string a =
    f[i - 1].substr(f[i - 1].size() - (lp - 1));
    string b =
    f[i].substr(f[i].size() - (lp - 1));
    mc[i % 2] = count(a + x, p);
    mc[(i + 1) % 2] = count(b + x, p);
    return count(n, p);
}
```

Code

```
long long count(int n, string p) {
    if (c[n] < 0) {
        c[n] = count(n - 1, p)
        + count(n - 2, p)
        + mc[n % 2];
    }
    return c[n];
}
```

04. CLOPAIR ()

05. GOLD MINING (vuongdx)

- Có n nhà kho nằm trên một đoạn thẳng.
- Nhà kho i có toạ độ là i và chứa lượng vàng là a_i.
- Chọn một số nhà kho sao cho:
  - Tổng lượng vàng lớn nhất.
  - 2 nhà kho liên tiếp có khoảng cách nằm trong đoạn $[L_1, L_2]$.

Thuật toán 1a

Tìm kiếm vét cạn:

- Nhà kho thứ i có thể được chọn hoặc không $\rightarrow$ có 2^n cách chọn.
- Với mỗi cách chọn, kiểm tra xem 2 nhà kho liên tiếp $i, j (i < j)$ có thoả mãn $L_1 <= j - i <= L_2$ không, nếu thoả mãn thì tính tổng số vàng và cập nhật kết quả tốt nhất.
- Có thể sử dụng stack để lưu danh sách các nhà kho được chọn.
- Độ phức tạp: $O(2^n \times n)$.

Code 1a

```
void _try(int x) {
    if (x == n) {
        updateResult();
    }
    _try(x + 1);
    s.push(x);
    _try(x + 1);
    s.pop();
}

void main() {
    try(0);
}
```

Thuật toán 1b

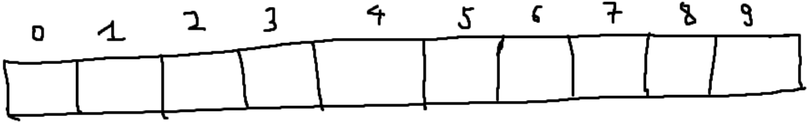
Nhận xét:

- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.

Thuật toán 1b

Nhận xét:

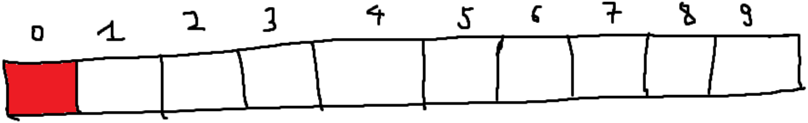
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

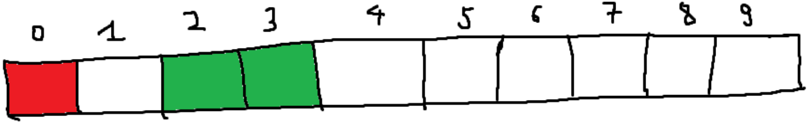
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

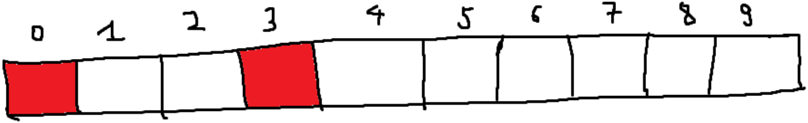
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

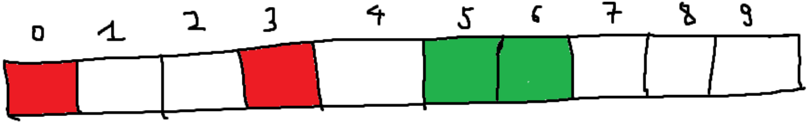
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

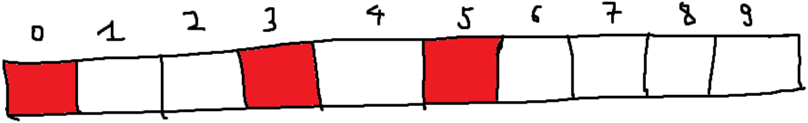
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2]$ → hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

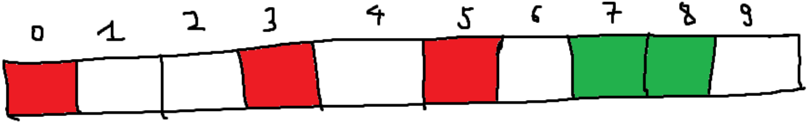
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

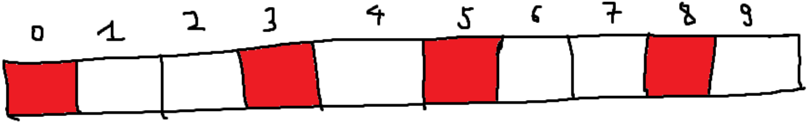
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

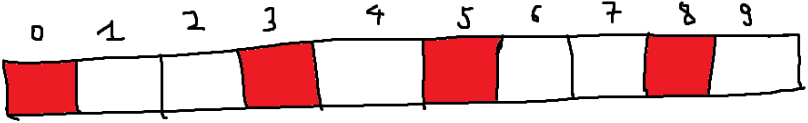
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



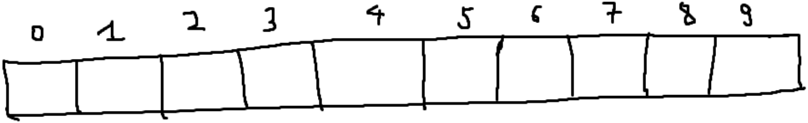
Thuật toán 1b

Nhận xét:

- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



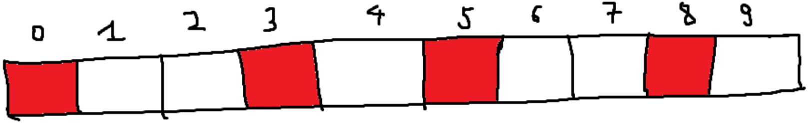
- Có thể xét các nhà kho theo thứ tự ngược lại.



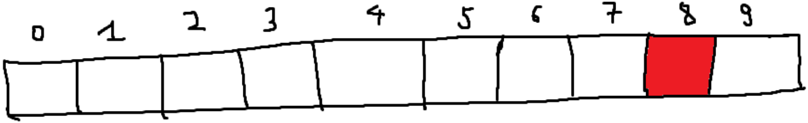
Thuật toán 1b

Nhận xét:

- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2]$ $\rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



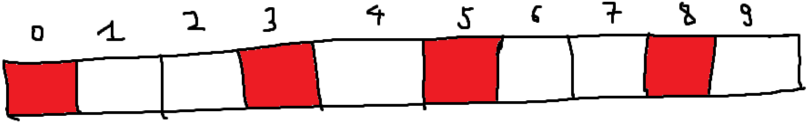
- Có thể xét các nhà kho theo thứ tự ngược lại.



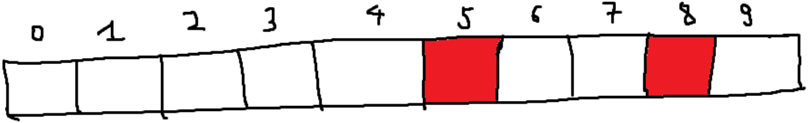
Thuật toán 1b

Nhận xét:

- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



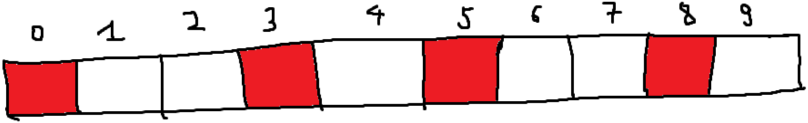
- Có thể xét các nhà kho theo thứ tự ngược lại.



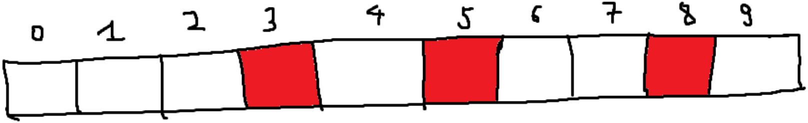
Thuật toán 1b

Nhận xét:

- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



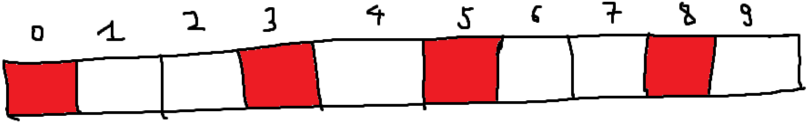
- Có thể xét các nhà kho theo thứ tự ngược lại.



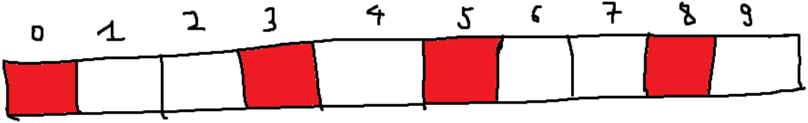
Thuật toán 1b

Nhận xét:

- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



- Có thể xét các nhà kho theo thứ tự ngược lại.

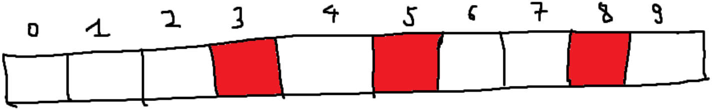


Code 1b

```
void _try(int x) {
    if (x < 0) {
        updateResult();
    }
    s.push(x);
    for (int i = x - l2; x <= i - l1; i++) {
        _try(i);
    }
    s.pop();
}

void main() {
    for (int i = n - l1 + 1; i < n; i++) {
        _try(i);
    }
}
```

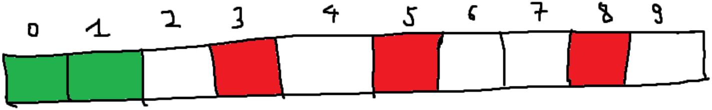
Thuật toán 1c

- Sau khi chọn nhà kho x, rõ ràng ta chỉ cần quan tâm đến tổng lượng vàng lớn nhất khi chọn các nhà kho phía trước nhà kho x.



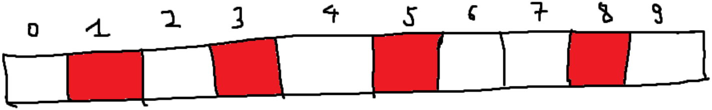
Thuật toán 1c

- Sau khi chọn nhà kho x, rõ ràng ta chỉ cần quan tâm đến tổng lượng vàng lớn nhất khi chọn các nhà kho phía trước nhà kho x.



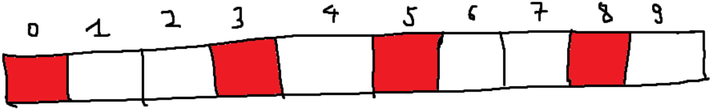
Thuật toán 1c

- Sau khi chọn nhà kho x, rõ ràng ta chỉ cần quan tâm đến tổng lượng vàng lớn nhất khi chọn các nhà kho phía trước nhà kho x.

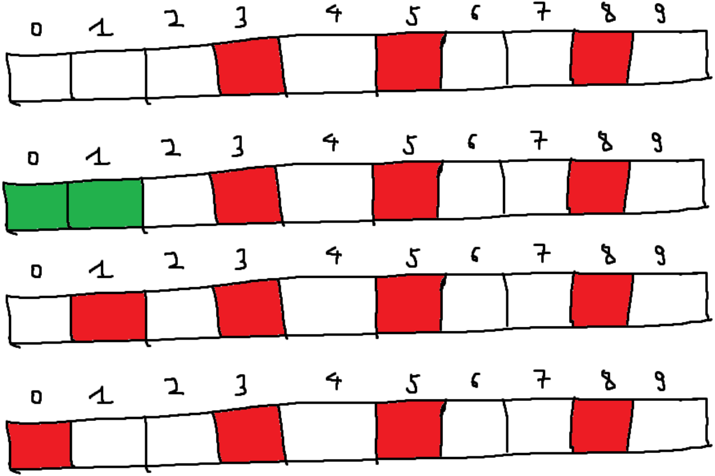


Thuật toán 1c

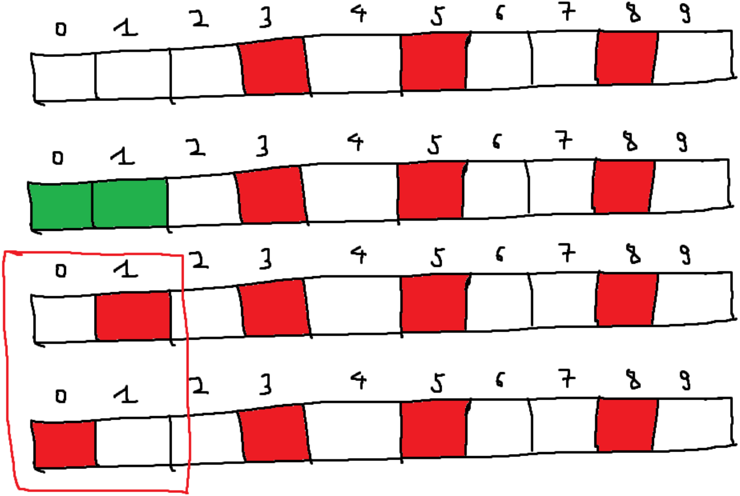
- Sau khi chọn nhà kho x, rõ ràng ta chỉ cần quan tâm đến tổng lượng vàng lớn nhất khi chọn các nhà kho phía trước nhà kho x.



Thuật toán 1c



Thuật toán 1c



Thuật toán 1c

- Sửa đổi hàm _try(x): Trả về tổng lượng vàng lớn nhất khi chọn một số nhà kho trong số các nhà kho từ 0 đến x.

Code 1c

```
int _try(int x) {
    if (x < 0) {
        return 0;
    }
    int tmp = 0;
    for (int i = x - l2; x <= i - l1; i++) {
        tmp = max(tmp, _try(i));
    }
    return tmp + a[x];
}

void main() {
    int res = 0;
    for (int i = n - l1 + 1; i < n; i++) {
        res = max(res, _try(i));
    }
}
```

Thuật toán 2a

- Thuật toán 1c chưa tối ưu: Hàm _try được gọi nhiều lần với cùng tham số x nào đó.
- Khắc phục:
  - Lưu lại $F(x)$ là tổng lượng vàng lớn nhất khi chọn một số nhà kho trong các nhà kho từ 0 đến x.
  - Mỗi khi _try(x) được gọi, nếu $F(x)$ chưa được tính thì tính giá trị cho $F(x)$, sau đó luôn trả về $F(x)$.
- Đây chính là thuật toán quy hoạch động, sử dụng hàm đệ quy (có nhớ).

Code 2a

```
int _try(int x) {
    if (x < 0) {
        return 0;
    }
    if (F[x]) < 0) {
        int tmp = 0;
        for (int i = x - l2; x <= i - l1; i++) {
            tmp = max(tmp, _try(i));
        }
        F[x] = tmp + a[x];
    }
    return F[x];
}

void main() {
    int res = 0;
    for (int i = n - l1 + 1; i < n; i++) {
        res = max(res, _try(i));
    }
}
```

Thuật toán 2b

Ta có thể dễ dàng cài đặt thuật toán 2a bằng phương pháp lặp:

- Gọi $F[i]$ là tổng số vàng nếu nhà kho i là nhà kho cuối cùng được chọn.
- Khởi tạo: $F[i] = a[i], \forall i < L_1$.
- Công thức truy hồi:

$$F[i] = max_{j \in [i-L_2, i-L_1]}(a[i] + F[j]), \forall i \in [L_1, n) \qquad (2)$$

- Kết quả: $max_i F[i]$.
- Độ phức tạp: $O(N \times (L_2 - L_1)) = O(N^2)$.

Code 2b

```
int main() {
    ...
    for (int i = 0; i < n; i++) {
        F[i] = a[i];
    }
    for (int i = l1; i < n; i++) {
        for (int j = i - l2; j <= i - l1; j++) {
            F[i] = max(F[i], F[j] + a[i]);
        }
    }
    ...
}
```

Cải tiến thuật toán

- Nhận thấy việc tìm giá trị lớn nhất của $F[j], \forall j \in [i - L_2, i - L_1]$ khá tốn kém ($O(n)$), liệu ta có thể giảm chi phí của bước này?
- Để cải tiến thuật toán, ta cần kết hợp các cấu trúc dữ liệu nâng cao để tối ưu việc truy vấn.

Cải tiến hàm đệ quy

- Sử dụng các cấu trúc dữ liệu hỗ trợ truy vấn khoảng tốt như Segment Tree, Interval Tree (IT), Binary Index Tree (BIT).
- Các cấu trúc trên đều cho phép cập nhật một giá trị và truy vấn (tổng, min, max) trên khoảng trong thời gian $O(logn)$.
- Với bài tập này, ta cần duy trì song song 2 cấu trúc (1 để truy vấn lượng vàng lớn nhất, 1 để truy vấn các giá trị $F[x]$ chưa được tính).
- Các cấu trúc dữ liệu trên đều không được cài đặt sẵn trong thư viện và không "quá dễ hiểu".

Sử dụng hàng đợi ưu tiên

Hàng đợi ưu tiên:

- Hàng đợi ưu tiên (priority queue) là một hàng đợi có phần tử ở đầu là phần tử có độ ưu tiên cao nhất.
- Thường cài đặt bằng Heap nên có độ phức tạp cho mỗi thao tác push, pop là $O(logn)$.

Cải tiến:

- Mỗi phần tử trong hàng đợi là một cặp giá trị $(j, F[j])$.
- Ưu tiên phần tử có $F[j]$ lớn.
- Khi xét đến nhà kho i, thêm cặp giá trị $(i - L_1, F[i - L_1])$ vào hàng đợi.
- Loại bỏ phần tử j ở đầu hàng đợi trong khi $i - j > L_2$, gán $F[i] = a[i] + F[j]$.
- Độ phức tạp: $O(n + n \times log(n)) = O(n \times log(n))$

Code 3a

```
class comp {
bool reverse;
public:
    comp(const bool& revparam=false) {
        reverse=revparam;
    }

    bool operator() (const pil& lhs,
    const pil&rhs) const {
        if (reverse) {
            return (lhs.second>rhs.second);
        }
        else {
            return (lhs.second<rhs.second);
        }
    }
};
```

Code 3a

```
int main() {
    ...
    for (int i = l1; i < n; i++) {
        int j = i - l1;
        q.push(make_pair(j, f[j]));
        while (q.top().first < i - l2) {
            q.pop();
        }
        F[i] = a[i] + q.top().second;
    }
    ...
}
```

Sử dụng hàng đợi 2 đầu

Hàng đợi 2 đầu:

- Hàng đợi 2 đầu (dequeue) là cấu trúc dữ liệu kết hợp giữa hàng đợi và ngăn xếp $\rightarrow$ phần tử có thể được lấy ra ở đầu hoặc cuối dequeue.

Ta định nghĩa các thao tác push và pop cho dequeue dùng trong bài:

- push(x): Xoá mọi phần tử i mà $F[i] \leq F[x]$ trong hàng đợi, thêm x vào cuối hàng đợi.
- pop(): Lấy ra phần tử ở đầu hàng đợi và xoá nó khỏi hàng đợi.

Sử dụng hàng đợi 2 đầu

Áp dụng vào bài toán:

- Tính $F[i]$ theo thứ tự.
  - Gọi push(i - L1).
  - Gọi $u = pop()$ cho đến khi $u >= i - L2$.
  - $F[i] = F[u] + a[i]$.

Sử dụng hàng đợi 2 đầu

Khi tính $F[i]$:

- Hàng đợi sắp thêm theo thứ tự giảm dần của giá trị $F[]$, do $i - L1$ được thêm vào cuối hàng đợi (khi đã loại hết các giá trị nhỏ hơn nó).
- Các nhà kho trong hàng đợi cũng được sắp xếp theo thứ tự được thêm vào hàng đợi.
- Nhà kho $i - L1$ là nhà kho cuối cùng được thêm vào hàng đợi, nên không có nhà kho nào quá gần i.
- Mọi nhà kho cách quá xa i đều bị loại khỏi hàng đợi (thao tác pop()).
- **Kết luận**: Những nhà kho còn lại trong hàng đợi đều thoả mãn ràng buộc, và nhà kho đầu tiên của hàng đợi là lựa chọn tối ưu.

Sử dụng hàng đợi 2 đầu

Độ phức tạp:

- Khi tính $F[i]$, push(i - L1) và vòng lặp các thao tác pop() đều có chi phí tối đa là $O(n)$.
- Tổng chi phí cũng chỉ là $O(n)$:
  - Mỗi nhà kho được thêm vào hàng đợi tối đa 1 lần và được lấy ra khỏi hàng đợi tối đa 1 lần.
  - n nhà kho chỉ được đưa vào và lấy ra tổng cộng $2n = O(n)$ lần.
- **Độ phức tạp**: $O(n)$.

Code 3b

```
int main() {
    ...
    for (int i = l1; i < n; i++) {
        int j = i - l1;
        dq.push(j);
        while (dq.top() < i - l2) {
            dq.pop();
        }
        F[i] = a[i] + F[dq.top()];
    }
    ...
}
```

Bàn luận

Bàn luận

Tại sao dequeue lại hiệu quả hơn priority queue trong trường hợp này?

Bàn luận

Tại sao dequeue lại hiệu quả hơn priority queue trong trường hợp này?

- Do dequeue luôn xoá các nhà kho chắc chắn không thể dùng đến, nên các thao tác truy vấn sau đó sẽ hiệu quả hơn.

Bàn luận

Tại sao dequeue lại hiệu quả hơn priority queue trong trường hợp này?

- Do dequeue luôn xoá các nhà kho chắc chắn không thể dùng đến, nên các thao tác truy vấn sau đó sẽ hiệu quả hơn.

Tại sao không dễ tối ưu cài đặt sử dụng hàm đệ quy?

Bàn luận

Tại sao dequeue lại hiệu quả hơn priority queue trong trường hợp này?

- Do dequeue luôn xoá các nhà kho chắc chắn không thể dùng đến, nên các thao tác truy vấn sau đó sẽ hiệu quả hơn.

Tại sao không dễ tối ưu cài đặt sử dụng hàm đệ quy?

- Do hàm đệ quy gọi _try(x) không theo thứ tự của x, nên không thể áp dụng các cấu trúc như priority queue và dequeue.

Bàn luận

Tại sao dequeue lại hiệu quả hơn priority queue trong trường hợp này?

- Do dequeue luôn xoá các nhà kho chắc chắn không thể dùng đến, nên các thao tác truy vấn sau đó sẽ hiệu quả hơn.

Tại sao không dễ tối ưu cài đặt sử dụng hàm đệ quy?

- Do hàm đệ quy gọi _try(x) không theo thứ tự của x, nên không thể áp dụng các cấu trúc như priority queue và dequeue.

Truy vết

- Hầu hết các bài toán không chỉ yêu cầu đưa ra giá trị tối ưu mà còn yêu cầu đưa ra lời giải.
- Để đưa ra lời giải, ta cần một mảng đánh dấu để có thể truy vết ngược lại. Ví dụ được thể hiện ở code bên dưới:

Code 3c

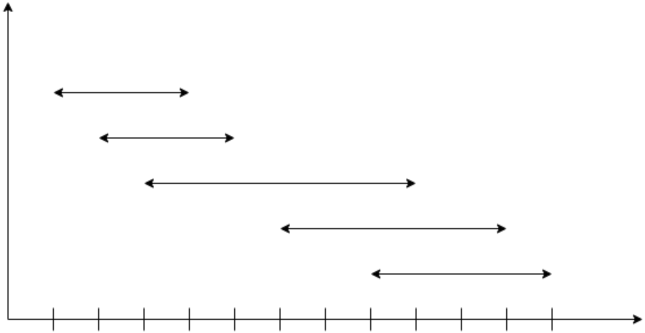
```
int main() {
    ...
    for (int i = l1; i < n; i++) {
        int j = i - l1;
        dq.push(j);
        while (dq.top() < i - l2) {
            dq.pop();
        }
        F[i] = a[i] + F[dq.top()];
        trace[i] = dq.top();
    }
    int i = argmax(F);
    while (i >= 0) {
        select.add(i);
        i = trace[i];
    }
    ...
}
```

05. MACHINE (TungTT)

- Cho n đoạn, đoạn thứ i bắt đầu từ s_i đến t_i.
- Số tiền nhận được khi chọn đoạn thứ i là $t_i - s_i$.
- 2 đoạn i, j được gọi là tách biệt nếu $t_i < s_j$ hoặc $t_j < s_i$.
- Cần chọn 2 đoạn tách biệt sao cho số tiền nhận được là lớn nhất.
- In ra số tiền nhận được

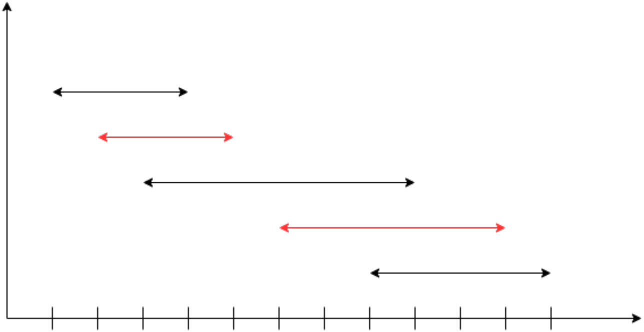
Ví dụ

- Có 5 đoạn thẳng $[8, 12]; [6, 11]; [3, 9]; [2, 5]; [1, 4]$



Ví dụ

- Cách chọn tối ưu : chọn 2 đoạn $[6, 11]$ và $[1, 4]$.
- Số tiền nhận được : 8



Thuật toán 1

- Duyệt toàn bộ $\frac{n(n-1)}{2}$ cách chọn, mỗi cách chọn kiểm tra điều kiện và lấy kết quả tối ưu.
- Độ phức tạp $O(n^2)$

Thuật toán 2

- Sử dụng quy hoạch động.
- Gọi $maxAmount[x]$ là giá trị đoạn lớn nhất có điểm cuối $\leq x$
- Giả sử đoạn i là một đoạn được chọn và có điểm cuối t_i lớn hơn đoạn còn lại thì giá trị lớn nhất mà ta có thể nhận được là $t_i - s_i + maxAmount[s_i - 1]$
- Lấy giá trị max $t_i - s_i + maxAmount[s_i - 1]$ của tất cả các vị trí i
- Độ phức tạp : $O(n)$

Code

```
int main() {
    const int N = 2e6 + 5;
    for (int i = 1; i <= n; i++) {
        maxs[t[i]] = max(maxs[t[i]], t[i] - s[i]);
    }

    for (int i = 1; i < N; i++) {
        maxs[i] = max(maxs[i - 1], maxs[i]);
    }

    int ans = -1;
    for (int i = 1; i <= n; i++) {
        if (maxs[s[i] - 1] > 0) {
            ans = max(ans,
                      maxs[s[i] - 1] + t[i] - s[i]);
        }
    }
    cout << ans << endl;
}
```

05. MARBLE (quanglm)

- Có một tấm đá có kích thước $W \times H$.
- Cần cắt tấm đá thành các miếng có kích thước nằm trong $W_1 \times H_1, W_2 \times H_2, \ldots, W_n \times H_n$.
- Tấm đá có vân nên không thể xoay, có nghĩa là miếng đá $A \times B$ khác miếng đá $B \times A$.
- Các lát cắt phải thẳng và được cắt tại các điểm nguyên theo cột hoặc theo hàng, và phải cắt hết hàng hoặc hết cột.
- Các miếng đá không có kích thước như trên sẽ bị bỏ đi.
- Tìm cách cắt sao cho diện tích bỏ đi là ít nhất.

Thuật toán

- **Thuật toán 1:** Duyệt vét cạn tất cả các cách cắt.
- **Thuật toán 2:** Quy hoạch động: Gọi $dp_{i,j}$ là phần diện tích bỏ đi ít nhất khi miếng đá có kích thước là $i \times j$.
  - Ta sẽ tính $dp_{i,j}$ dựa trên các giá trị của $dp_{i',j'}$ với $i' \leq i$ và $j' \leq j$ đã được tính từ trước.
  - $dp_{i,j} = 0$ nếu $\exists k(1 \leq k \leq n) : (i,j) = (W_k, H_k)$.
  - Nếu cắt theo chiều ngang, ta có:

$$dp_{i,j} = \min_{i_0=1}^{i-1}(dp_{i_0,j} + dp_{i-i_0,j})$$

  - Nếu cắt theo chiều dọc, ta có:

$$dp_{i,j} = \min_{j_0=1}^{j-1}(dp_{i,j_0} + dp_{i,j-j_0})$$

  - Kết quả là $dp_{W,H}$. ĐPT thuật toán $O(WH(N + W + H))$.

Code

```
for (int i = 1; i <= W; i++) {
    for (int j = 1; j <= H; j++) {
        dp[i][j] = i * j;
        for (int k = 1; k <= n; k++) {
            if (i == w[k] && j == h[k]) {
                dp[i][j] = 0;
                break;
            }
        }
        for (int k = 1; k < i; k++) {
            dp[i][j] = min(dp[i][j],
                        dp[k][j] + dp[i - k][j]);
        }
        for (int k = 1; k < j; k++) {
            dp[i][j] = min(dp[i][j],
                        dp[i][k] + dp[i][j - k]);
        }
    }
}
```

05. TOWER (ngocbh)

- n loại hình hộp chữ nhật có kích thước (x_i, y_i, z_i) có số lượng tùy ý và có thể xoay hoặc lật trong không gian 3 chiều.
  - i.e. $(x_i, y_i, z_i) \rightarrow (y_i, z_i, x_i)$
- Một tòa tháp $t^{(1)}, t^{(2)}, \ldots$ được xây mỗi tầng là một hình hộp sao cho mặt sàn tầng dưới lớn hơn chặt mặt sàn tầng trên:
  - $t_x^{(i)} > t_x^{(i+1)}, t_y^{(i)} > t_y^{(i+1)}$
- Mục tiêu: Xây tòa tháp cao nhất có thể.

$$\sum_{i=1} t_z^{(i)} \rightarrow max$$

Nhận xét

- $t_x^{(i)} > t_x^{(i+1)}, t_y^{(i)} > t_y^{(i+1)} \rightarrow$ bỏ khả năng xoay và lật của hình hộp thì mỗi hình hộp chỉ được sử dụng một lần.
- có tối đa 6 cách xoay cho mỗi hình hộp
- $\rightarrow$ sinh ra $6 * n$ hình hộp, mỗi hình hộp dùng một lần.
- sắp xếp lại các hình hộp theo độ lớn giảm dần của $x_i \rightarrow y_i \rightarrow z_i$.
- $\rightarrow$ với mỗi hình hộp, đảm bảo hình hộp đứng sau không lớn hơn hình hộp đứng trước.
- $\rightarrow$ chia bài toán thành $6 * n$ bài toán nhỏ, bài toán i ứng với xây tòa tháp độ cao lớn nhất sử dụng các hình hộp từ $1 \ldots i \rightarrow$ quy hoạch động.

Thuật toán

- $dp[i]$ chiều cao tòa tháp cao nhất sử dụng hình hộp i làm chóp.
- $$dp[i] = \max_{j \in [0..i-1], x[i] < x[j], y[i] < y[j]} (dp[j] + z[i])$$
- kết quả $\max_{i \in [1, 6*n]}(dp[i])$

Code

```
bool cmp(const Rec a, const Rec b) {
    if ( a.x != b.x ) return a.x > b.x;
    if ( a.y != b.y ) return a.y > b.y;
    return a.z > b.z;
}

int m = 0;
for (int i = 0; i < n; i++) {
    int x[3];
    cin >> x[0] >> x[1] >> x[2];
    sort(x,x+3);
    do {
        a[++m].x = x[0], a[m].y = x[1], a[m].z = x[2];
    } while ( next_permutation(x,x+3) );
}

sort(a+1, a+m+1, cmp);

a[0].x = a[0].y = INF, a[0].z = 0;

for (int i = 1; i <= m; i++) {
    for (int j = 0; j < i; j++)
        if ( a[j].y > a[i].y && a[j].x > a[i].x ) {
            dp[i] = max(dp[i], dp[j] + a[i].z);
        }
    ans = max(ans, dp[i]);
}
```

05. WAREHOUSE (ngocbh)

- N nhà kho được đặt tại các vị trí từ $1...N$. Mỗi nhà kho có:
  - a_i là số lượng hàng.
  - t_i là thời gian lấy hàng.
- một tuyến đường lấy hàng đi qua các trạm $x_1 < x_2 < ... < x_k$ $(1 \leq x_j \leq N, j = 1...k)$ sao cho:
  - $x_{i+1} - x_i \leq D \ \forall i \in [1, k]$.
  - $\sum_{i=1}^{k} t[x_i] \leq T$

Thuật toán

- gọi $dp[i][k]$ là số lượng hàng tối đa thu được khi xét các nhà kho từ 1...i, lấy hàng ở kho i và thời gian lấy hàng không quá k.
- $$dp[i][k] = \begin{cases} 0 & \text{if } k < t[i] \\ \max\limits_{j \in [i-D, i-1]} (dp[j][k - t[i]] + a[i]) & \text{if } k \geq t[i] \end{cases}$$
- kết quả $ans = \max_{i \in [1,n], k \in [1,T]} (dp[i][k])$

Code

```
for (int i = 1; i <= n; i++) {
    for (int k = t[i]; k <= T; k++) {
        for (int j = i-1; j >= max(0,i-D); j--)
            dp[i][k] = max(dp[i][k],
                dp[j][k-t[i]] + a[i]);
        ans = max(ans, dp[i][k]);
    }
}
```

05. RETAIL OUTLETS (DucLA)

- Đếm số cách phân bổ M cửa hàng cho N chi nhánh
- Hai cách được coi là khác nhau nếu có một chi nhánh có số cửa hàng được phân bổ khác nhau trong 2 cách
- Điều kiện: số cửa hàng được phân bổ cho chi nhánh i phải là số nguyên dương chia hết cho $a[i]$

Thuật toán

- Gọi $F(i,j)$ là số cách phân bổ j cửa hàng cho i chi nhánh đầu tiên
- $F(0,0) = 1$
- $$F(i,j) = \sum_{k>0, k \vdots a[i]} F(i-1, j-k)$$
- Kết quả là $F(N,M)$
- Số trạng thái: $O(N * M)$
- Chi phí chuyển trạng thái: $O(M)$
- Độ phức tạp: $O(N * M^2)$

Code

```
f[0][0] = 1;
for (int i = 1; i <= n; ++i)
    for (int j = a[i]; j <= m; ++j)
        for (int k = j - a[i]; k >= 0; k -= a[i])
            (f[i][j] += f[i - 1][k]) %= 1000000007;
```

05. DRONE PICKUP (vuongdx

- N địa điểm được đặt tại các vị trí $1 \ldots N$. Mỗi địa điểm có:
  - c_i là số lượng hàng.
  - a_i năng lượng.
- Một drone cần bay từ điểm 1 đến điểm N:
  - Không được dừng ở quá $K + 1$ điểm (kể cả điểm xuất phát và đích).
  - Nếu dừng ở điểm i thì điểm dừng kế tiếp xa nhất là $i + a_i$.
- **Yêu cầu:** Tìm lộ trình bay để drone lấy được nhiều hàng nhất.

Nhận xét

- Bài toán tương tự bài WAREHOUSE:
  - Thời gian lấy hàng ở mỗi địa điểm đều bằng 1.
  - Tổng thời gian lấy hàng là $K + 1$.
- Khoảng cách di chuyển xa nhất của drone không cố định như bài WAREHOUSE:
  - $max(a_i) = 50$ nên có thể coi $D = 50$ và kiểm tra thêm điều kiện $j + a[j] \geq i$.

Thuật toán 1

- Gọi $dp[i][k]$ là số lượng hàng tối đa thu được khi xét các địa điểm 1...i, lấy hàng ở địa điểm i và số địa điểm đã lấy hàng không vượt quá k.
- $$dp[i][k] = \begin{cases} -\infty & \text{if } k \leq 0 \\ \max(dp[j][k-1] + c[i], & \\ j \in [i - max(a_i), i - 1), & \\ j + a[j] \geq i & \text{if } k > 0 \end{cases}$$
- kết quả $ans = max(dp[n][k], k \in [1, K+1])$

Code 1

```
int D = max(a[]);
for (int i = 1; i <= n; i++) {
    for (int k = 1; k <= K + 1; k++) {
        for (int j = i-1; j >= max(0,i-D); j--)
            if (j + a[j] >= i) {
                dp[i][k] = max(dp[i][k],
                dp[j][k-1] + c[i]);
            }
    }
    ans = max(dp[n][]);
}
```

Cải tiến

- Để không cần phải xét cả 50 địa điểm kề trước i, ta quy dẫn bài toán đã cho thành bài toán sau:
  - Cần tìm 1 lộ trình đi từ N về 1.
  - Có thể di chuyển trực tiếp sang địa điểm i từ mọi địa điểm $j \leq i + a_i$.
  - $\rightarrow$ Có thể giải bằng thuật toán của bài WAREHOUSE.

Thuật toán 2

- Gọi $dp[i][k]$ là số lượng hàng tối đa thu được khi xét các địa điểm $i...N$, lấy hàng ở địa điểm i và số địa điểm đã lấy hàng không vượt quá k.
- $$dp[i][k] = \begin{cases} -\infty & \text{if } k \leq 0 \\ \max(dp[j][k-1] + c[i], j \in [i+1, i+a[i]]) & \text{if } k > 0 \end{cases}$$
- kết quả $ans = max(dp[1][k], k \in [1, K+1])$

Code 2

```
int D = max(a[]);
for (int i = n; i >= 1; i--) {
    for (int k = 1; k <= K + 1; k++) {
        for (int j = 1; j <= a[i]; j++) {
            dp[i][k] = max(dp[i][k],
            dp[j][k - 1] + c[i]);
        }
    }
    ans = max(dp[1][]);
}
```

05. NURSE (KienPT)

- Cần sắp xếp lịch làm việc cho một y tá trong N ngày
- Lịch làm việc bao gồm các giai đoạn làm việc được xen giữa bởi các ngày nghỉ
- Các giai đoạn làm việc là các ngày làm việc liên tiếp thỏa mãn hai điều kiện sau
  - Thời gian nghỉ giữa hai giai đoạn không quá một ngày
  - Số ngày làm việc của mỗi giai đoạn lớn hơn hoặc bằng K_1 và bé hơn hoặc bằng K_2
- Tìm số phương án xếp lịch thỏa mãn.

Thuật toán 1

- Mỗi cách xếp lịch tương ứng với một dãy nhị phân độ dài n. Bit thứ i là 0/1 tương ứng là ngày đó y tá được nghỉ hoặc phải đi làm
- Xét hết các xâu nhị phân độ dài n và tìm số lượng xâu thỏa mãn điều kiện

Thuật toán 2

- Gọi $F[x][i]$ là số cách xếp lịch thỏa mãn cho đến ngày thứ i và x là trạng thái nghỉ hoặc làm việc của ngày đó.
- Trường hợp cơ sở:
  - $F[0][0] = F[1][0] = 1$: Trường hợp không có ngày làm việc nào, ta luôn có một cách xếp lịch.
  - $F[0][1] = 1$.
  - $\forall i = 1, \ldots k_1 - 1 : F[1][i] = 0, F[0][i+1] = F[1][i]$.
  - $F[1][k_1] = 1$.
- Công thức truy hồi, $\forall i >= k_1$:
  - Với i là ngày nghỉ, ta có: $F[0][i] = F[1][i-1]$
  - Với i là ngày làm việc, ta có: $F[1, i] = \sum_{k=max(0,i-K_2)}^{i-K_1} F[0][k]$
- Kết quả của bài toán là: $F[0][n] + F[1][n]$

ICBUS(TungTT)

- Cho n thị trấn được đánh số từ 1 tới n.
- Có k con đường hai chiều nối giữa các thị trấn.
- Ở thị trấn thứ i sẽ có một tuyến bus với giá vé là c_i và đi được quãng đường tối đa là d_i.
- Tìm chi phí tối thiểu để đi từ thị trấn 1 tới thị trấn n.

Thuật toán

- **Bước 1 :** Tính khoảng cách di chuyển ngắn nhất của tất cả các cặp đỉnh u, v bằng thuật toán BFS. Lưu vào mảng $dist[u][v]$
- **Bước 2 :** Tạo một đồ thị mới một chiều trong đó đỉnh u được nối tới đỉnh v khi $dist[u][v] <= d[u]$ và cạnh này có trọng số là $c[u]$
- **Bước 3 :** Tìm đường đi ngắn nhất từ 1 tới n trên đồ thị mới được tạo ra bằng thuật toán Dijkstra.
- Độ phức tạp thuật toán $O(n^2)$

Code

```
void calculate_dist() {
    ** Calculate dist[u][v] using BFS algorithm **
}
void find_shortest_path() {
    for (int i = 0; i <= n; i++) {
        ans[i] = MAX;
        visit[i] = 0;
    }
    ans[1] = 0;
    int step = n;
    while (step--) {
        int min_vertex = 0;
        for (int i = 1; i <= n; i++) {
            if (visit[i] == 0 && ans[min_vertex] > ans[i]) {
                min_vertex = i;
            }
        }
        visit[min_vertex] = 1;
        for (int i = 1; i <= n; i++) {
            if (dist[min_vertex][i] <= d[min_vertex]) {
                ans[i] = min(ans[i], ans[min_vertex] + c[min_vertex]);
            }
        }
    }
    cout << ans[n] << endl;
}
```

06. ADDEDGE (name)

06. BUGLIFE (DucLA)

- Cho một đồ thị vô hướng
- Kiểm tra xem nó có phải là đồ thị hai phía hay không

Thuật toán

- Cần tô màu mỗi đỉnh thành màu đỏ hoặc đen
- Một đỉnh màu đỏ chỉ được kề với các đỉnh màu đen và ngược lại, một đỉnh màu đỉnh chỉ được kề với các đỉnh màu đỏ
- Xét một thành phần liên thông của đồ thị, nhận xét rằng nếu ta tô màu một đỉnh trong đó thì màu của tất cả các đỉnh còn lại đều được xác định duy nhất nếu tồn tại cách tô màu thỏa mãn
- Thuật toán DFS, với mỗi thành phần liên thông, gán nhãn bất kỳ cho một đỉnh, thử tô màu xác định cho các đỉnh còn lại.
- Nếu sau quá trình tô màu trên mà tồn tại 2 đỉnh kề cùng màu, đồ thị không phải là 2 phía.

Code

```
vector<int> a[N];
int color[N];

void dfs(int u) {
    for (int v : a[u]) {
        if (color[v] == -1) {
            color[v] = !color[u];
            dfs(v);
        }
    }
}
```

Code

```
    for (int i = 1; i <= n; ++i) color[i] = -1;
    for (int i = 1; i <= n; ++i) {
        if (color[i] == -1) {
            color[i] = 0;
            dfs(i);
        }
    }
    bool bipartite = true;
    for (int u = 1; u <= n; ++u) {
        for (int v : a[u]) {
            bipartite &= color[u] != color[v];
        }
    }
```

06. ELEVTRBL (name)

07. CHANGE (quanglm)

- Cho các đồng tiền có mệnh giá lần lượt là \$1, \$5, \$10, \$50, \$100, \$500.
- Cần tìm cách sử dụng ít đồng tiền nhất để tạo ra tổng tiền N $(1 \leq N \leq 999)$.

Thuật toán

- **Thuật toán 1:** Duyệt vét cạn tất cả các cách chia tiền, tìm cách có số lượng đồng tiền nhỏ nhất.
- **Thuật toán 2:** Tham lam: Xét lần lượt các mệnh giá từ lớn đến nhỏ, lấy tối đa số đồng tiền có thể để tổng tiền không vượt quá N. Cứ làm như vậy cho đến khi lấy đủ số tiền.

Tính đúng đắn

- Luôn tạo ra được tổng N: do khi xét mỗi mệnh giá, ta lấy tối đa có thể để tổng không vượt quá N, vậy ta luôn có tổng tiền $S \leq N$. Mà ta lại có mệnh giá \$1, nên sẽ tồn tại cách chọn để $S = N$.
- Cách chọn này là tối ưu: để ý rằng số đồng tiền \$1 được chọn < 5, do ngược lại ta có thể đổi 5 đồng \$1 lấy 1 đồng \$5. Tương tự số đồng \$5 < 2, ... $(*)$
- Giả sử cách chọn của chúng ta lấy a đồng \$500, 1 cách chọn tối ưu lấy $b < a$, $(b + a0 = a)$ đồng \$500. Ta có

$$N = a * 500 + a' = b * 500 + b' = (a - a0) * 500 + b'$$

  với a', b' là số tiền tạo ra từ các tờ tiền nhỏ hơn. Nên: $a' + 500 \leq b'$, mà từ các đồng bé hơn \$500 không thể tạo ra tổng ≤ 500 được do $(*)$ nên không tồn tại b'. Vậy lấy a đồng là tối ưu.
- Tương tự với các mệnh giá nhỏ hơn.

Code

```
int a[6] = {1, 5, 10, 50, 100, 500};
int res = 0;
for (int i = 5; i >= 0; i--) {
    res += n / a[i];
    n %= a[i];
}
cout << res << endl;
```

07. ATM (name)

07. PTREES (DucLA)

- Có N cái cây, cây thứ i cần t_i ngày để mọc
- Mỗi ngày trồng được một cây
- Hỏi ngày sớm nhất mà tất cả các cây đều mọc xong?

Thuật toán

- Cây càng mọc chậm thì càng phải trồng sớm
- Vì vậy ta sắp xếp các cây theo thứ tự mọc từ chậm đến nhanh, và trồng các cây theo thứ tự đó
- Cây thứ i sau khi sắp xếp sẽ được trồng ở ngày thứ i.

Code

```
int n; cin >> n;
vector<int> a(n);
for (int i = 0; i < n; ++i) cin >> a[i];
sort(rbegin(a), rend(a));
for (int i = 0; i < n; ++i) a[i] += i;
cout << *max_element(begin(a), end(a)) + 2 << endl;
```