Thuật toán ứng dụng
Bài thực hành số 4: Quy hoạch động

TS. Đinh Viết Sang, TA. Đặng Xuân Vương

Trường Đại học Bách khoa Hà Nội
Viện Công nghệ thông tin và Truyền thông

Ngày 30 tháng 11 năm 2020

Mục lục

Mục lục

MỞ ĐẦU

Quy hoạch động là gì?

- Quy hoạch động là một mô hình giải bài.
- Ý tưởng là giải lần lượt các bài toán con, kích thước tăng dần để thu được lời giải của bài toán ban đầu.
- Có thể cài đặt bằng hàm đệ quy hoặc vòng lặp.

MỞ ĐẦU

Quy hoạch động là gì?

- Quy hoạch động là một mô hình giải bài.
- Ý tưởng là giải lần lượt các bài toán con, kích thước tăng dần để thu được lời giải của bài toán ban đầu.
- Có thể cài đặt bằng hàm đệ quy hoặc vòng lặp.

Quy hoạch động và **Chia để trị** có gì khác nhau?

MỞ ĐẦU

Nhắc lại về **Chia để trị**:

- Chia để trị là một mô hình giải bài.
- Ý tưởng là chia bài toán lớn thành các bài toán con độc lập, giải từng bài toán con rồi kết hợp lời giải để thu được lời giải của bài toán ban đầu.
- Thường được cài đặt bằng hàm đệ quy.

Mô hình	**Quy hoạch động**	**Chia để trị**
Bài toán con	"Gối nhau", kích thước tăng dần	Độc lập, thường có kích thước bằng nhau
Cài đặt	Hàm đệ quy hoặc Vòng lặp	Hàm đệ quy

Bảng 1: Sự khác nhau giữa mô hình Quy hoạch động và mô hình Chia để trị

Mục lục

05. GOLD MINING

- 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

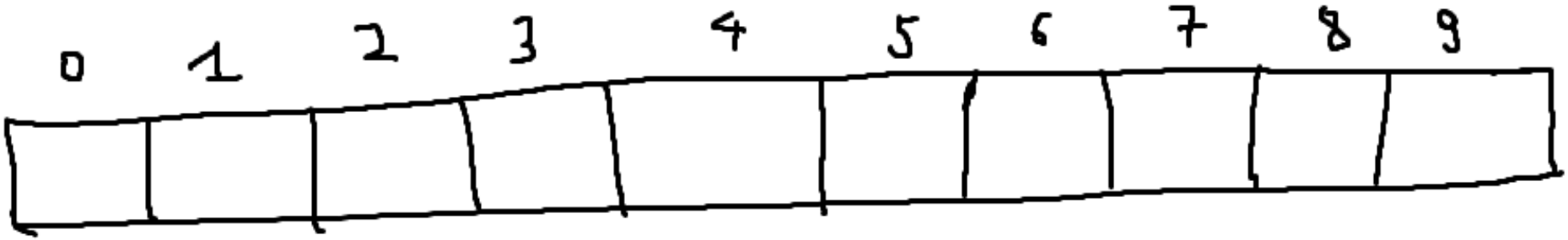
Nhận xét:

- 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:

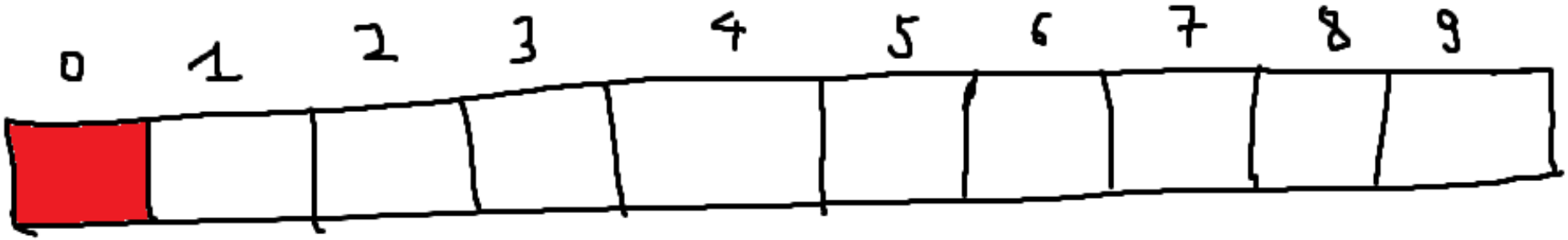
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

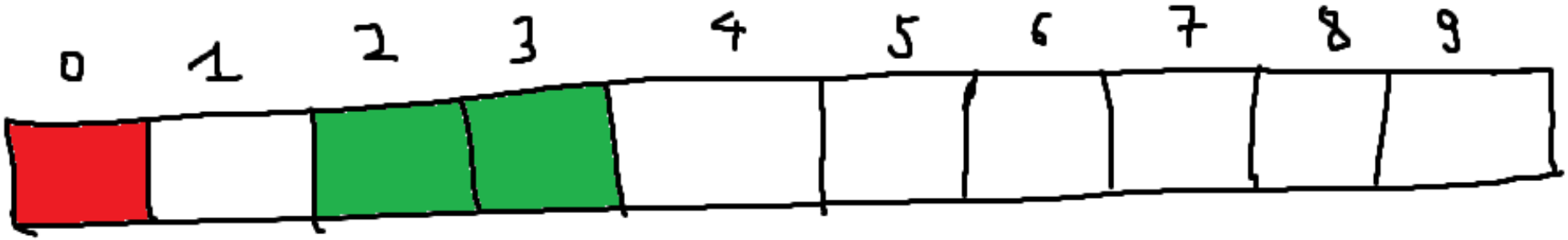
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

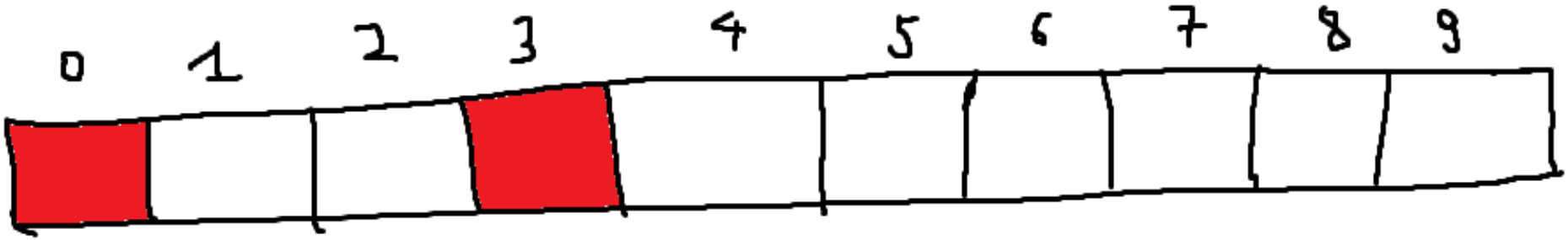
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

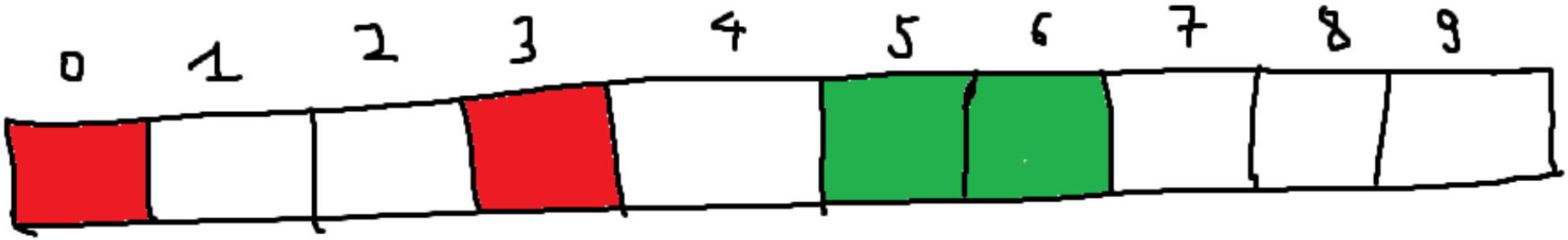
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

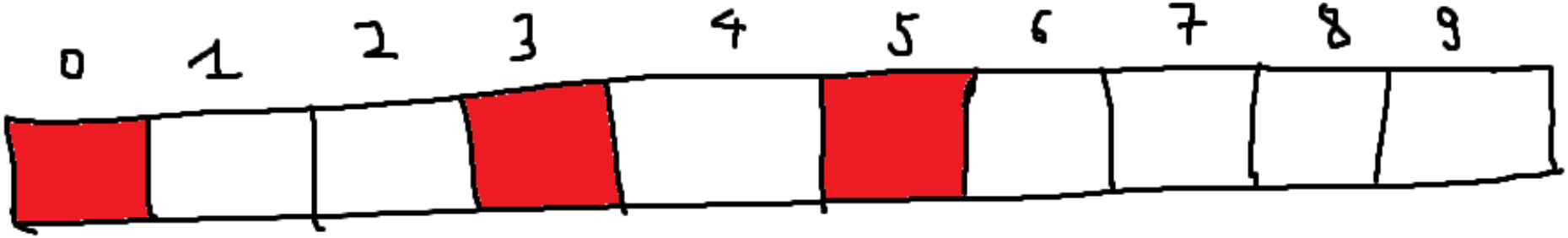
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

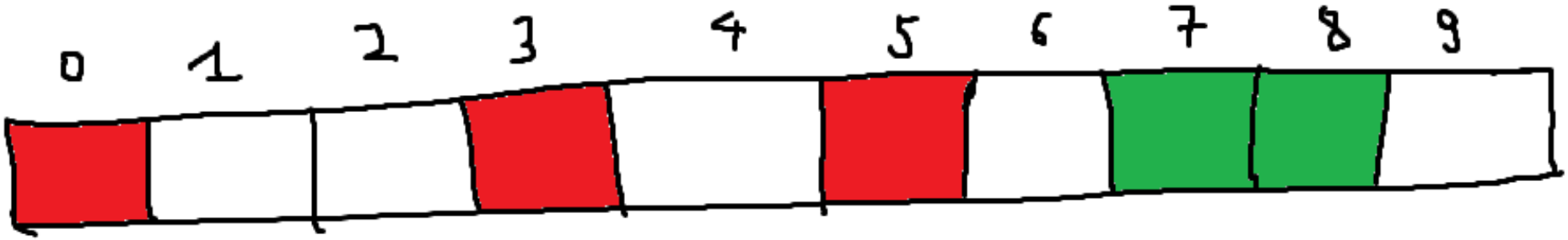
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

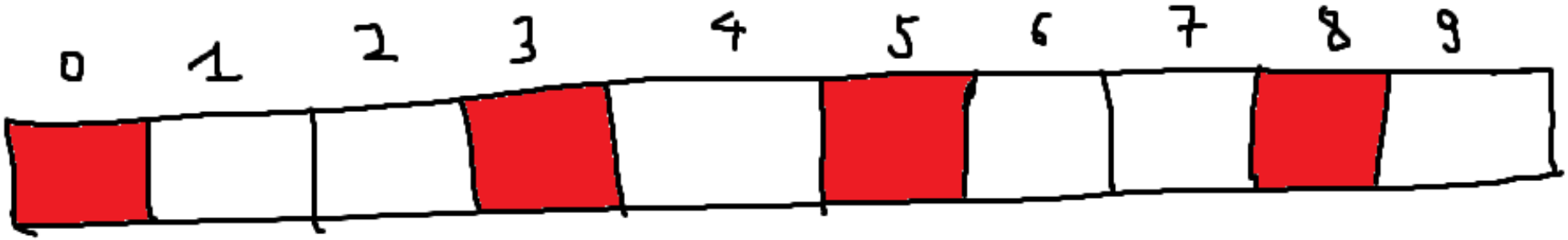
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

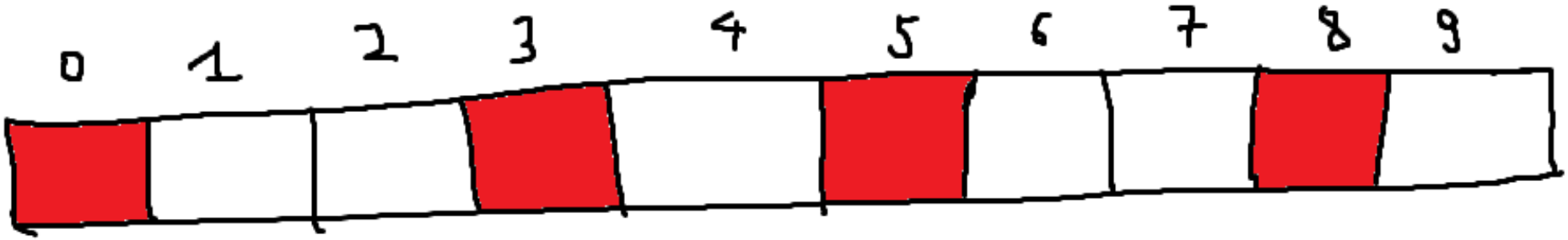
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



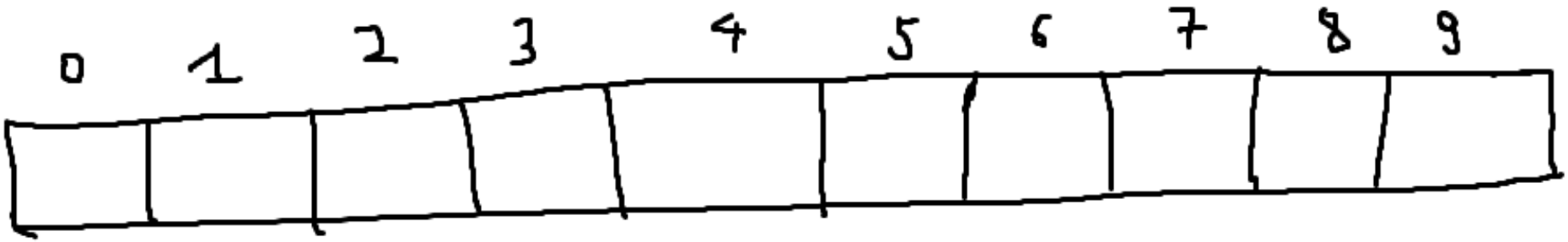
Thuật toán 1b

Nhận xét:

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



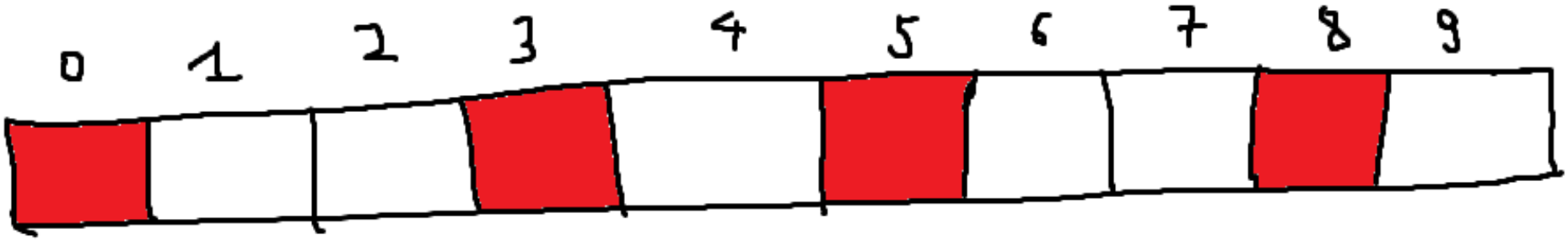
- Có thể xét các nhà kho theo thứ tự ngược lại.



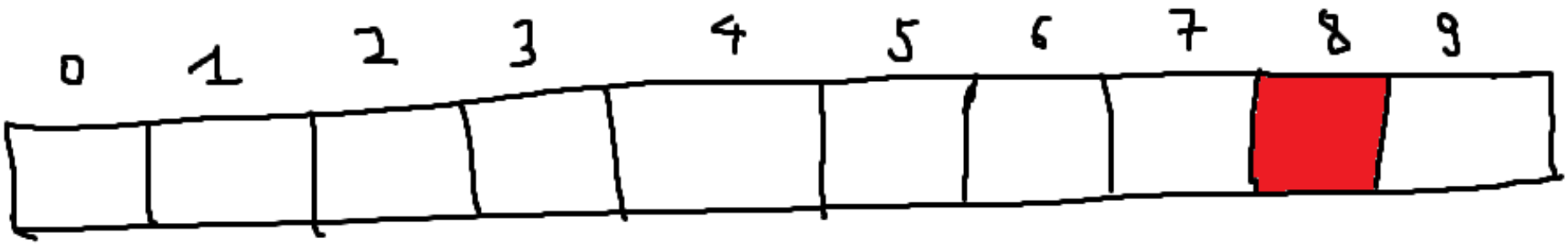
Thuật toán 1b

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: $n = 10, L_1 = 2, L_2 = 3$:



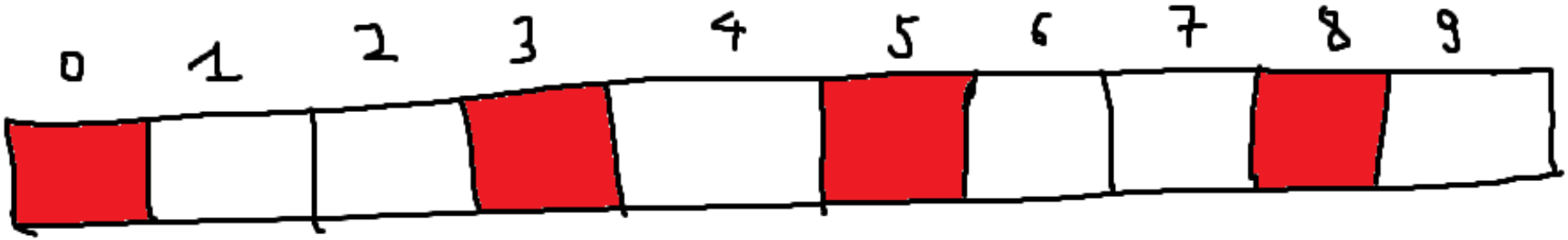
- Có thể xét các nhà kho theo thứ tự ngược lại.



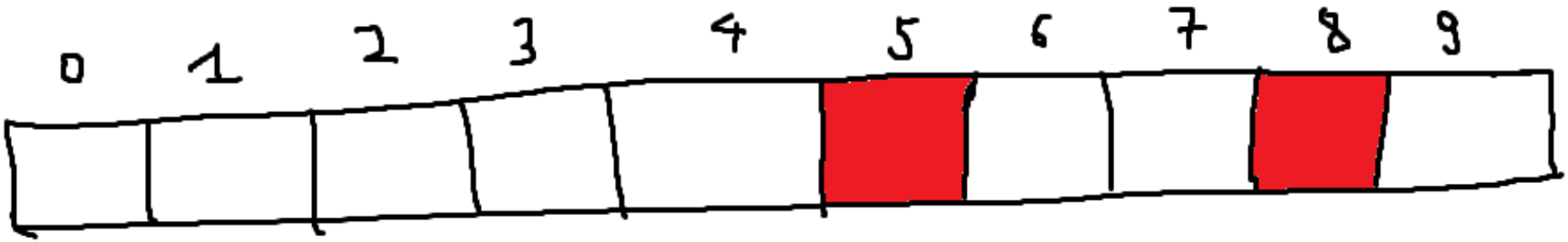
Thuật toán 1b

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: $n = 10, L_1 = 2, L_2 = 3$:



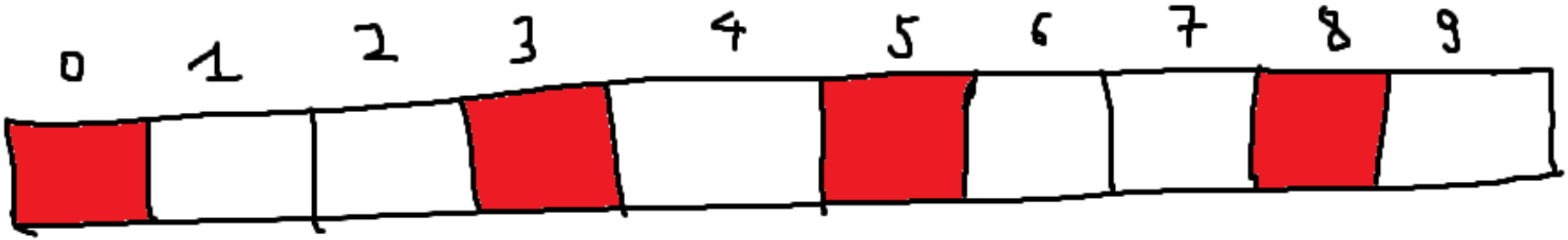
- Có thể xét các nhà kho theo thứ tự ngược lại.



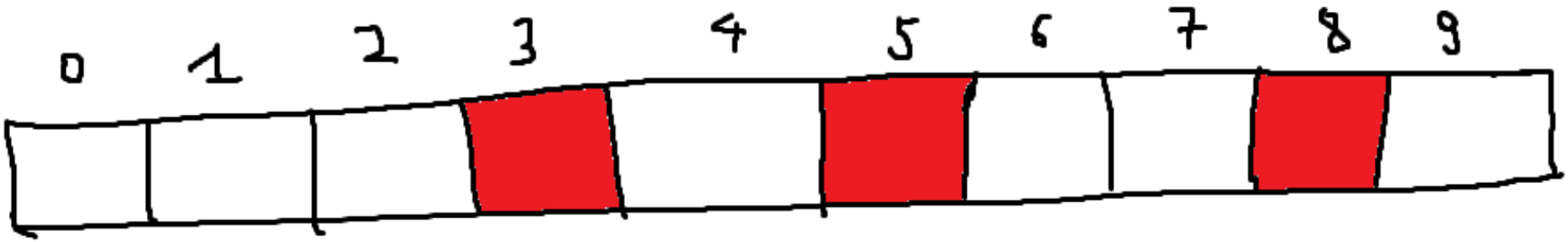
Thuật toán 1b

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: $n = 10, L_1 = 2, L_2 = 3$:



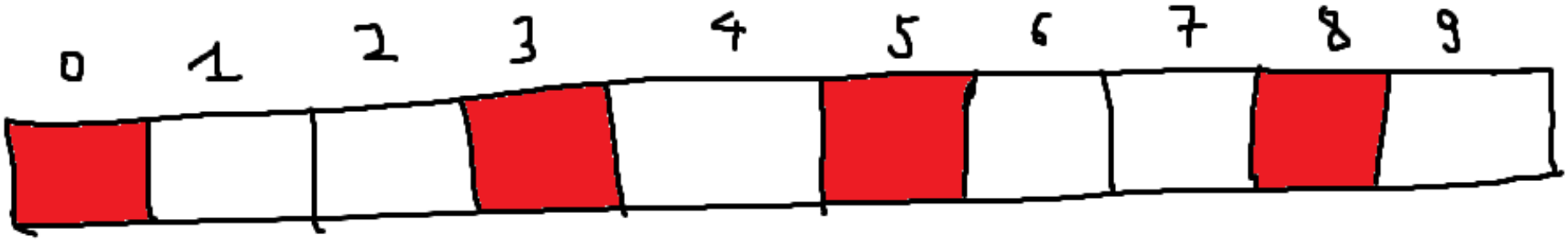
- Có thể xét các nhà kho theo thứ tự ngược lại.



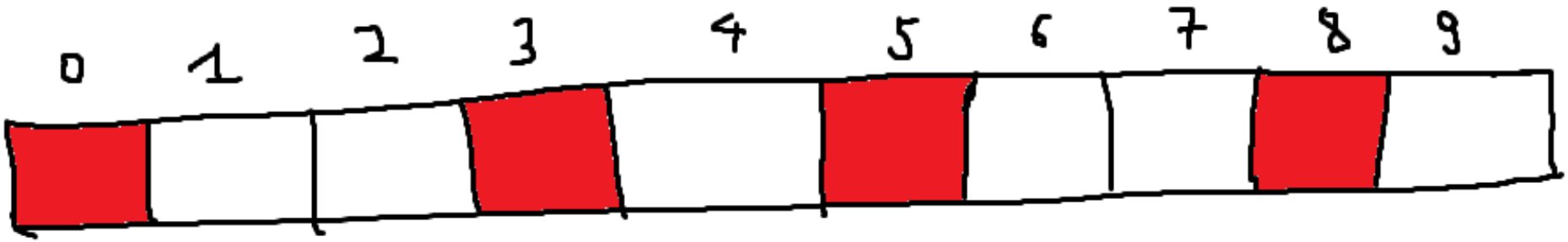
Thuật toán 1b

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.



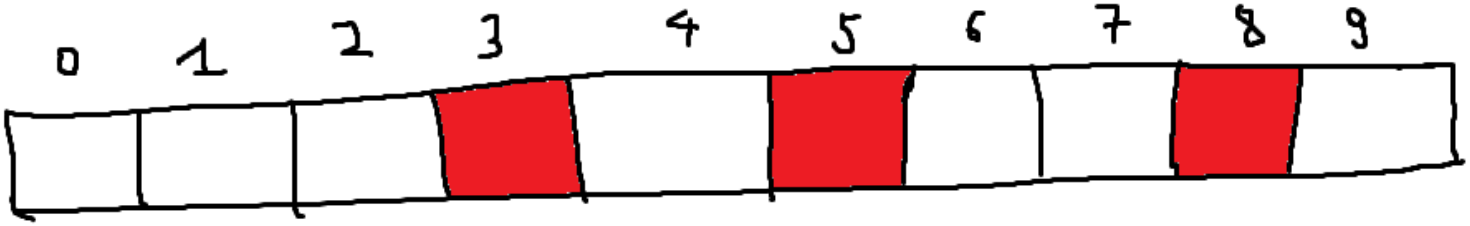
Code 1b

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

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

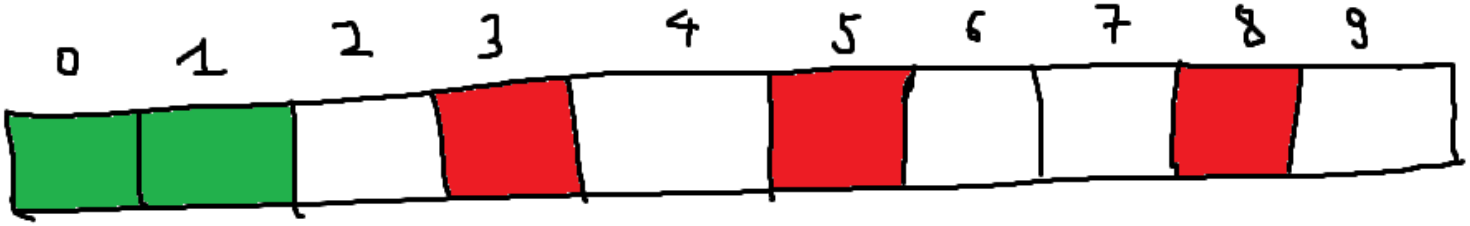
Thuật toán 1c

- 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.



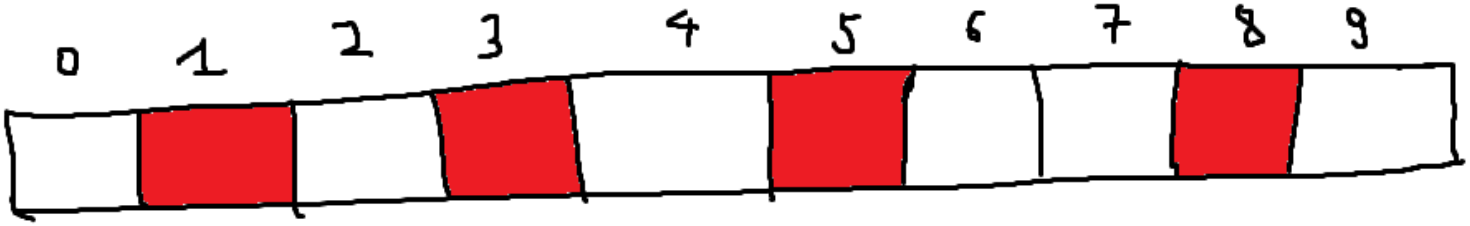
Thuật toán 1c

- 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.



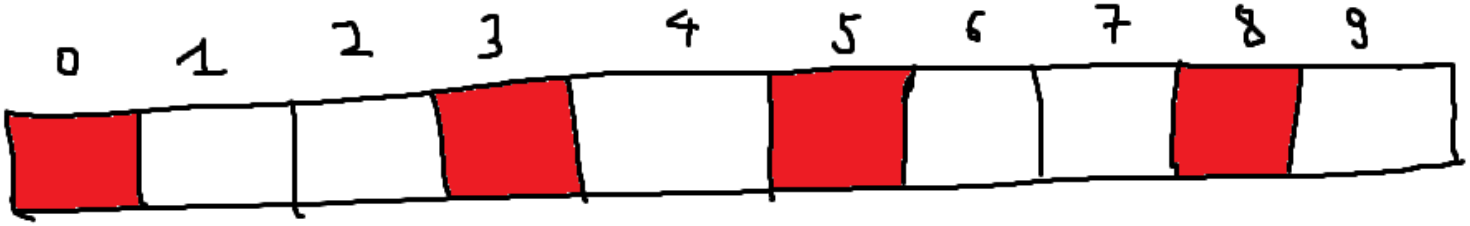
Thuật toán 1c

- 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.

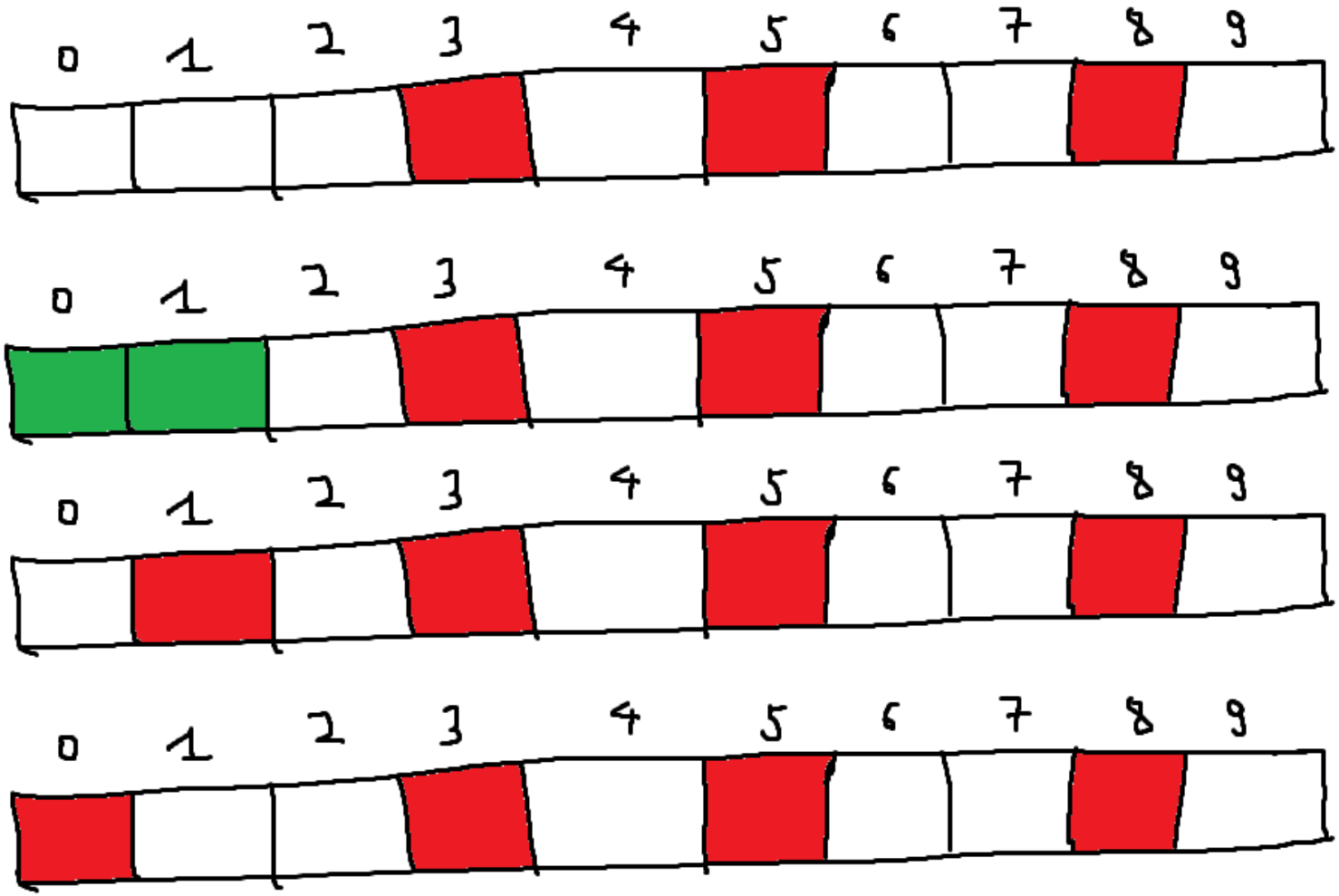


Thuật toán 1c

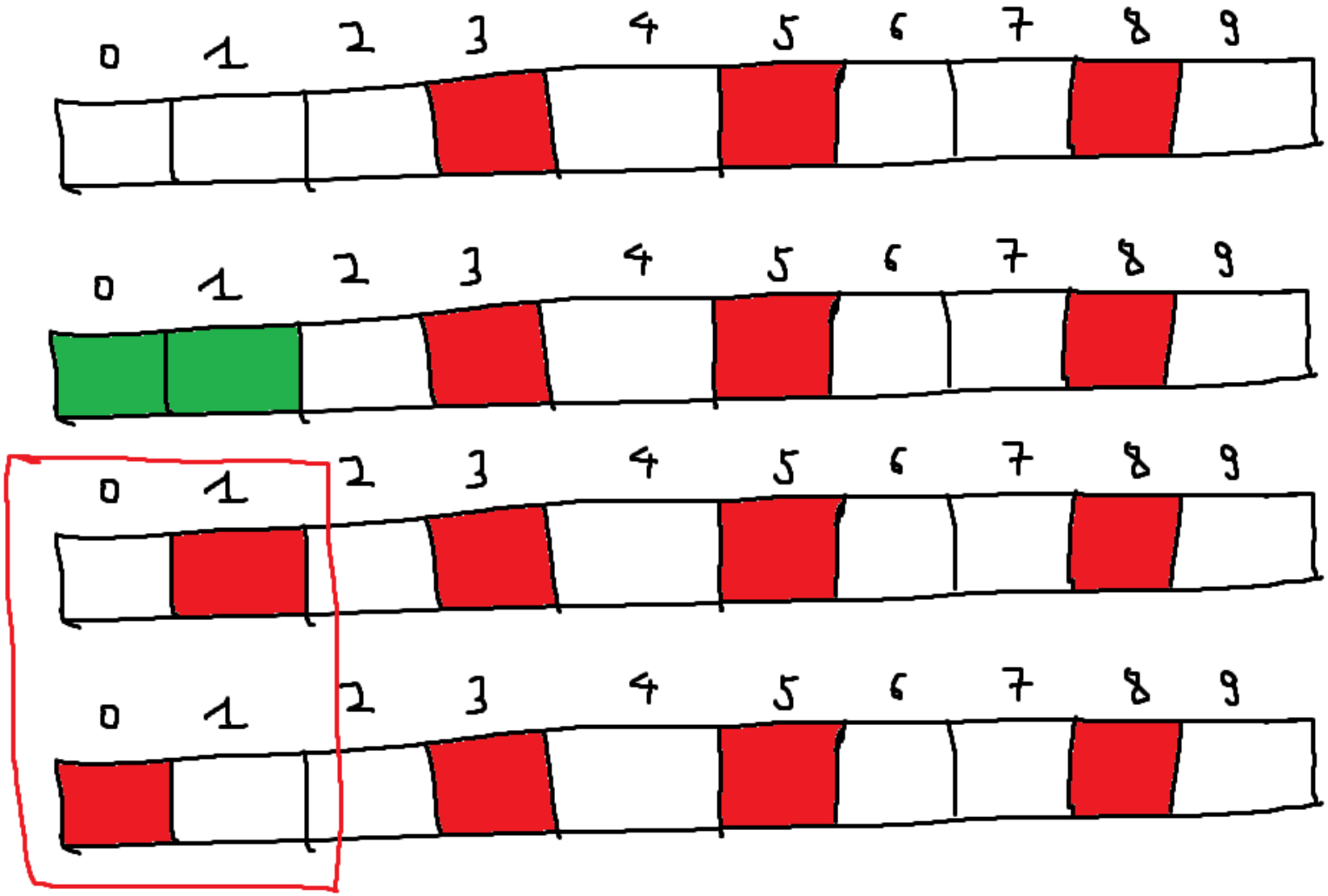
- Sau khi chọn nhà kho x, rõ ràng ta chỉ cần quan tâm đến tổng lượng vàng lớn nhất khi chọn các nhà kho phía trước nhà kho x.



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; i <= x - l1; i++) {
        tmp = max(tmp, _try(i));
    }
    return tmp + a[x];
}

void main() {
    int res = 0;
    for (int i = n - l1; 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; i <= x - 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; 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:

$$F[i] = max_{j \in [i-L_2, i-L_1]}(a[i] + F[j]), \forall i \in [L_1, n) \quad (1)$$

- 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 j mà $F[j] \leq F[x]$ 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).
  - Trong khi $top() < i - L2$ gọi $pop()$.
  - $F[i] = F[top()] + a[i]$.

Sử dụng hàng đợi 2 đầu

Khi tính $F[i]$:

- Hàng đợi sắp xếp 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];
    }
    ...
}
```

Mục lục

05. WAREHOUSE

- N nhà kho được đặt tại các vị trí $1...N$. 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:
  - $x_{i+1} - x_i \leq D \ \forall i \in [1, k]$.
  - $\sum_{i=1}^{k} t[x_i] \leq T$
- **Yêu cầu:** Tìm lộ trình để xe tải lấy được nhiều hàng nhất.

Nhận xét

- Bài toán tương tự như bài GOLD MINING.
- $L_1 = 1, L_2 = D \leq 10 \rightarrow$ không cần sử dụng priority queue hoặc dequeue để tối ưu truy vấn.
- Có thêm yếu tố thời gian.

Thuật toán

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

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]);
    }
}
```

Mục lục

05. DRONE PICKUP

- N địa điểm được đặt tại các vị trí $1...N$. 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 = i + 1; j <= i + a[i]; j++) {
            dp[i][k] = max(dp[i][k],
            dp[j][k - 1] + c[i]);
        }
    }
    ans = max(dp[1][]);
}
```

Thuật toán ứng dụng
Bài thực hành số 4: Quy hoạch động

TS. Đinh Viết Sang, TA. Đặng Xuân Vương

Trường Đại học Bách khoa Hà Nội
Viện Công nghệ thông tin và Truyền thông

Ngày 30 tháng 11 năm 2020