Training 5: Dynamic Programming

Ngoc Bui, Xuan-Vuong Dang, Kien-Trung Phan,Anh-Duc Le

Hanoi University of Science and Technology

Hanoi, 2020

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

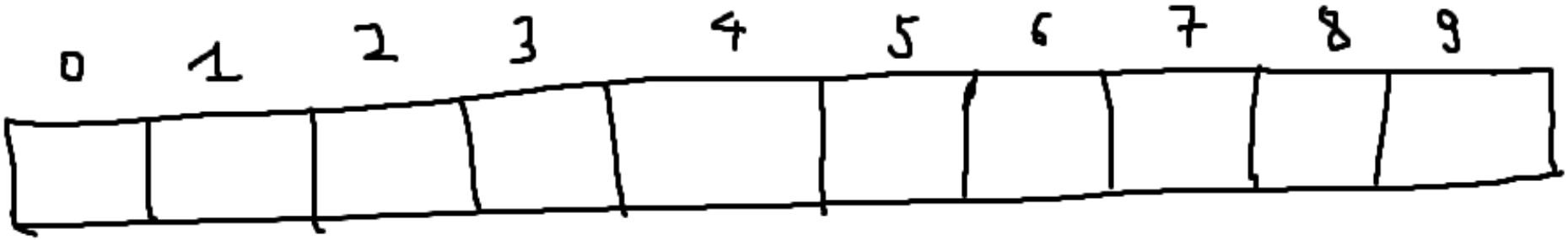
Nhận xét:

- 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:

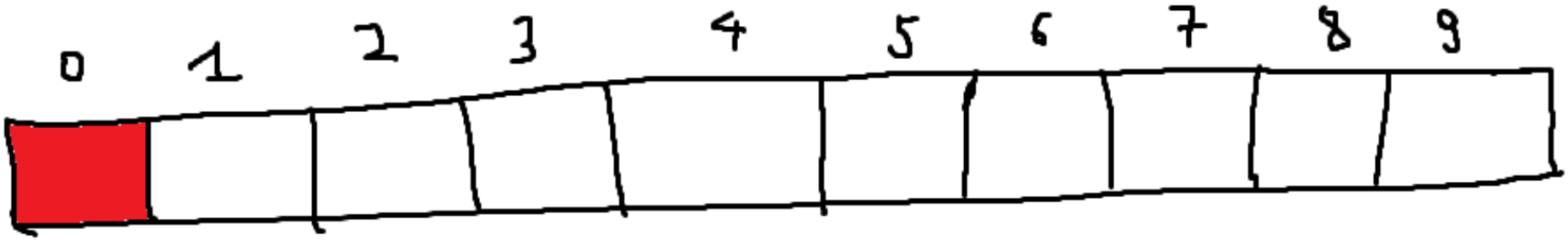
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

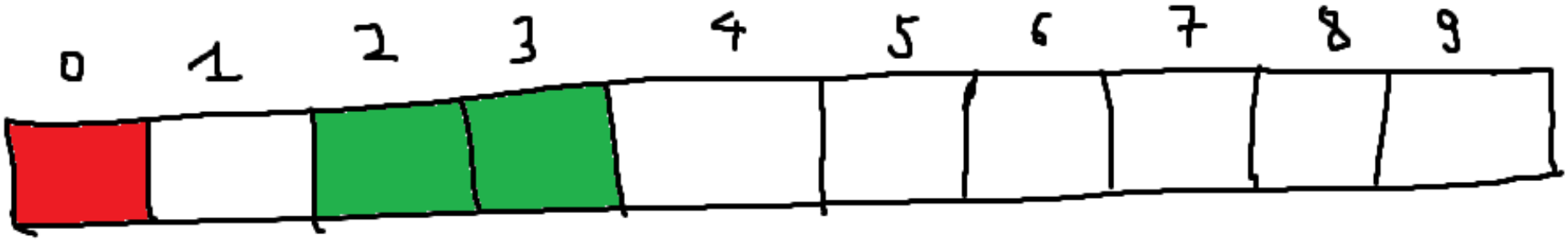
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

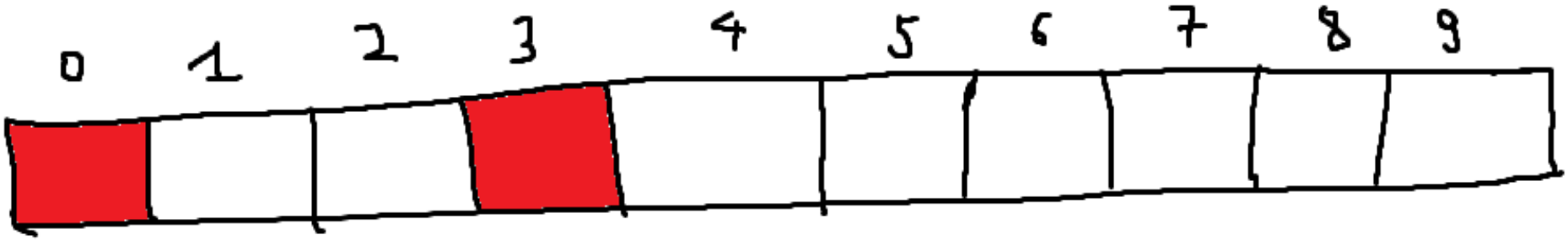
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

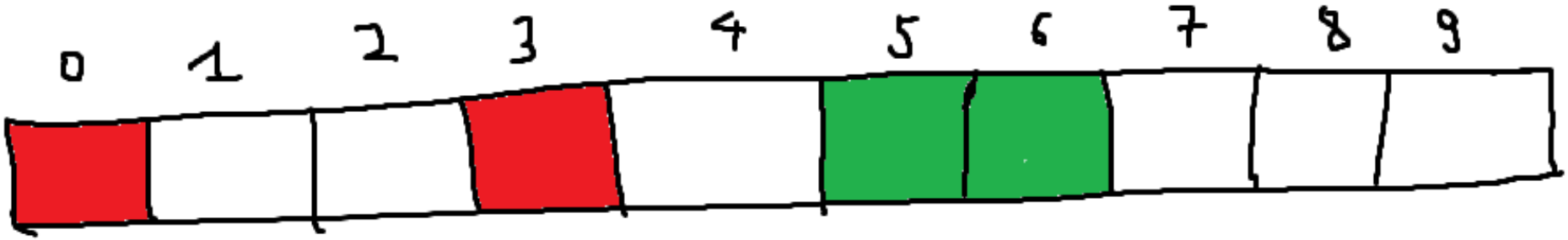
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

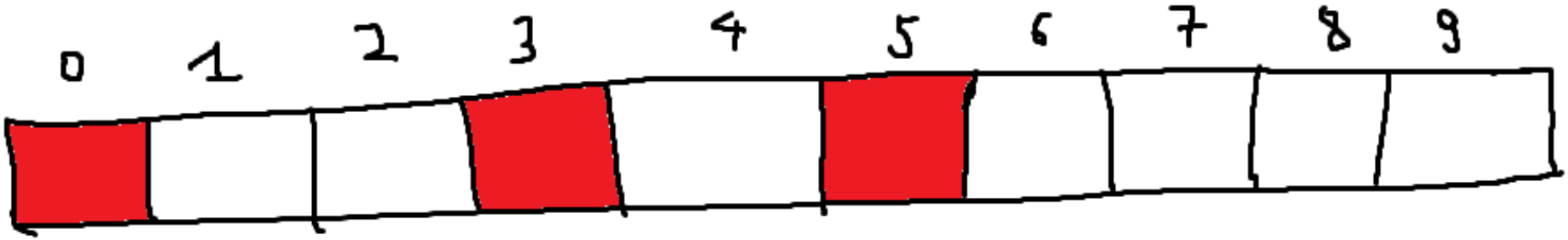
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

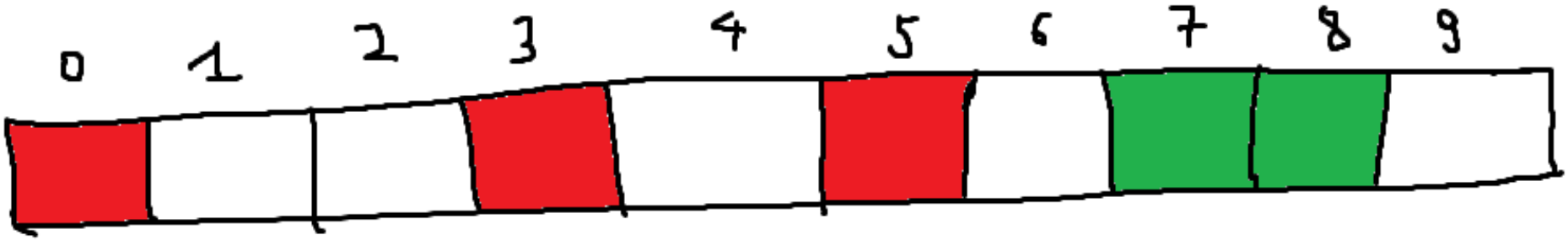
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

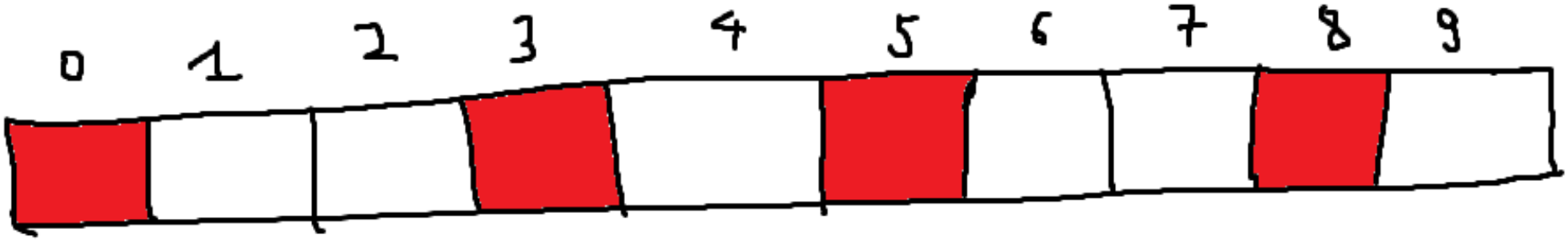
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



Thuật toán 1b

Nhận xét:

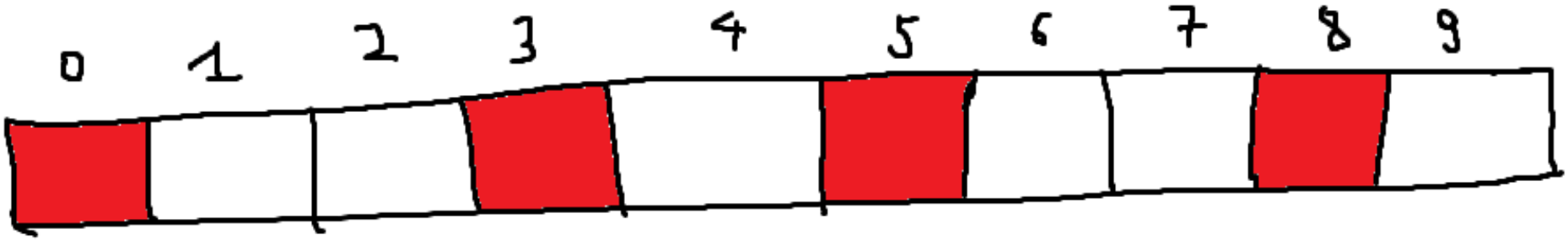
- Nếu nhà kho thứ i được chọn, ta chỉ có thể chọn các nhà kho $[i + L_1, i + L_2] \rightarrow$ hàm đệ quy không cần gọi qua các giá trị $[i + 1, i + L_1 - 1]$.
- Ví dụ: $n = 10, L_1 = 2, L_2 = 3$:



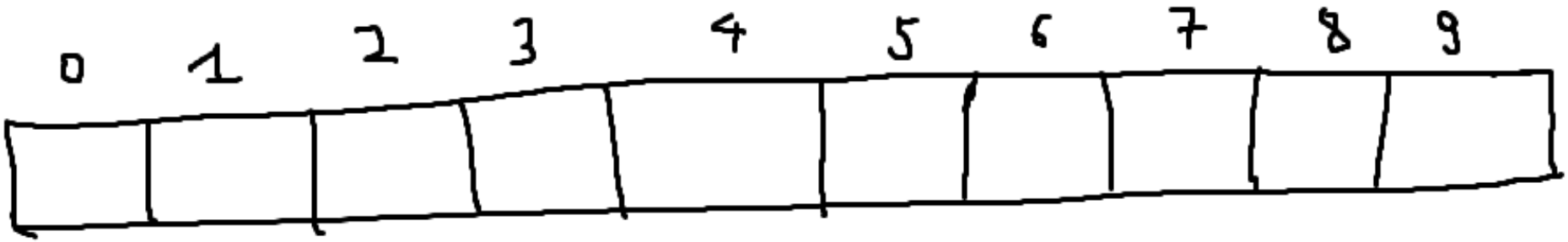
Thuật toán 1b

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



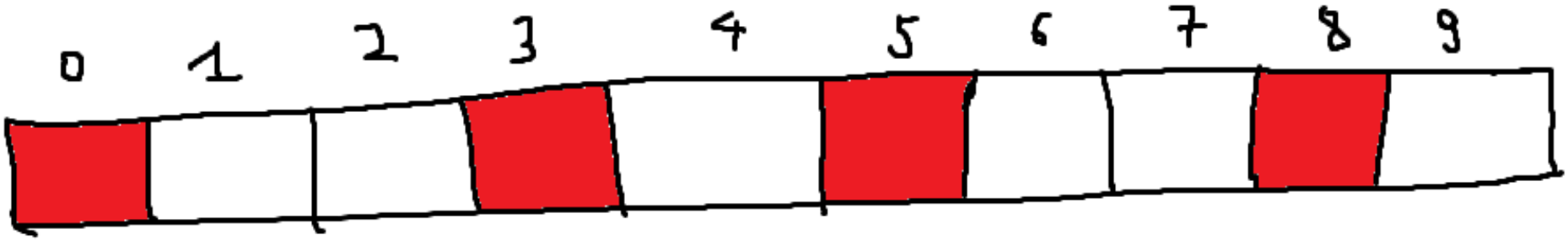
- Có thể xét các nhà kho theo thứ tự ngược lại.



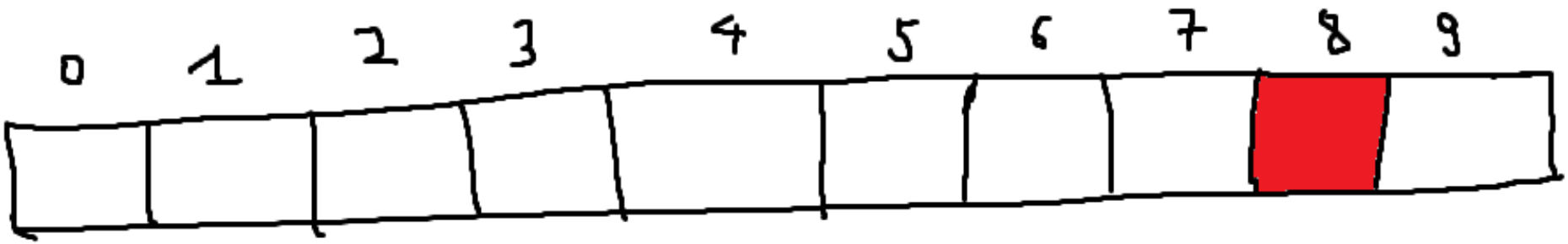
Thuật toán 1b

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



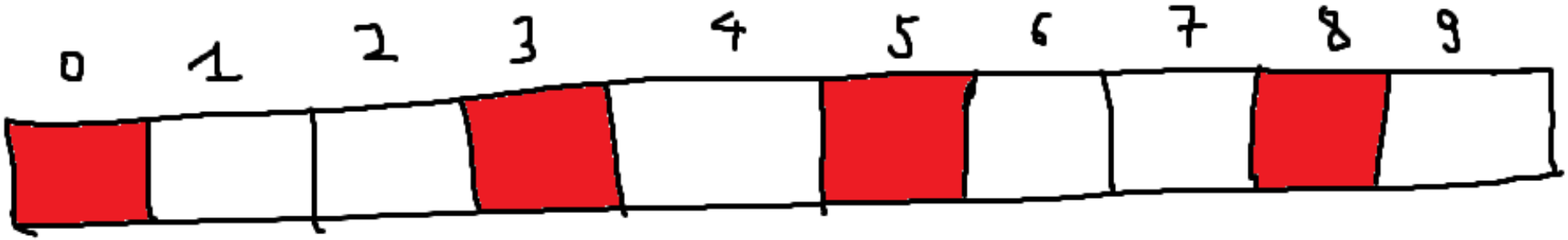
- Có thể xét các nhà kho theo thứ tự ngược lại.



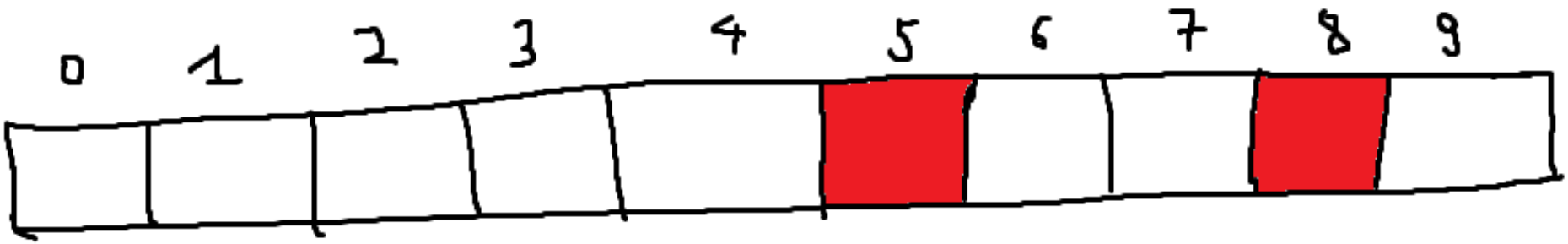
Thuật toán 1b

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



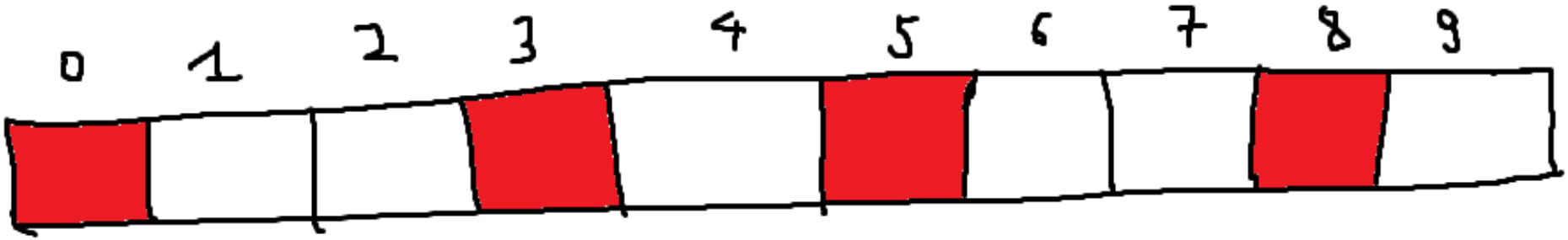
- Có thể xét các nhà kho theo thứ tự ngược lại.



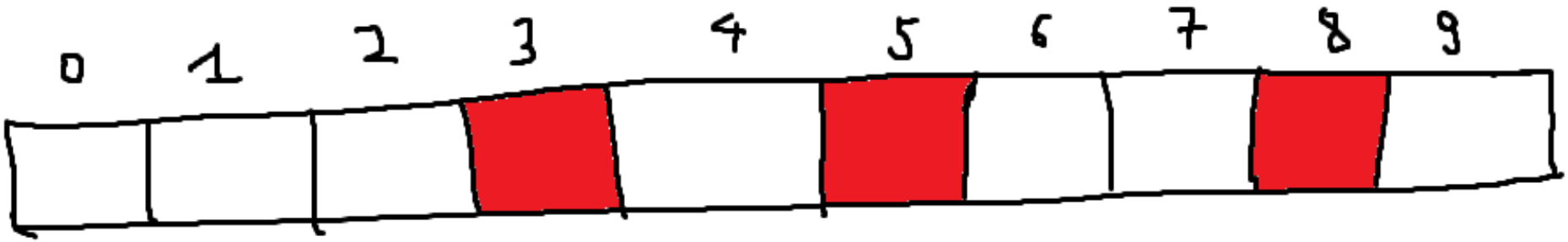
Thuật toán 1b

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



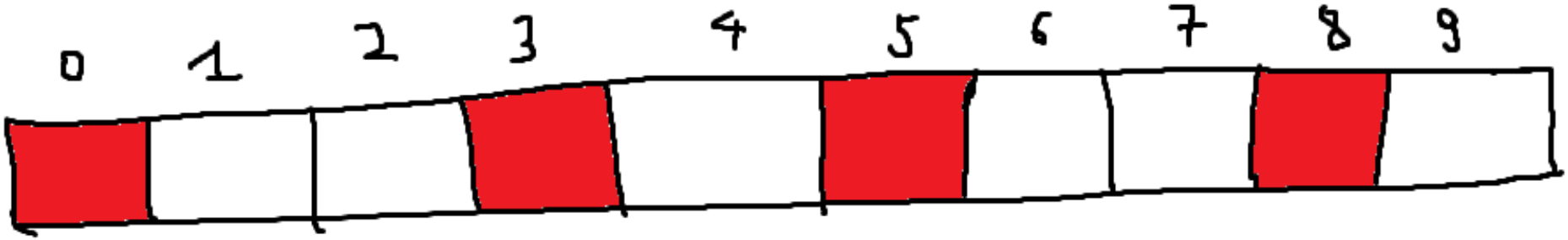
- Có thể xét các nhà kho theo thứ tự ngược lại.



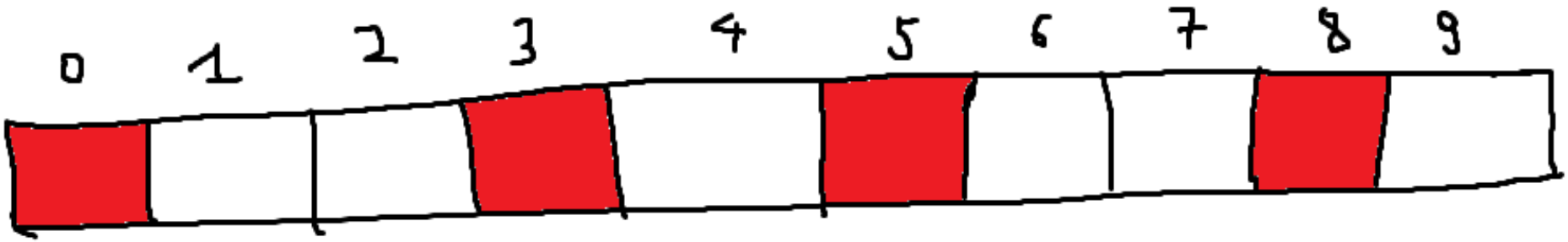
Thuật toán 1b

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.



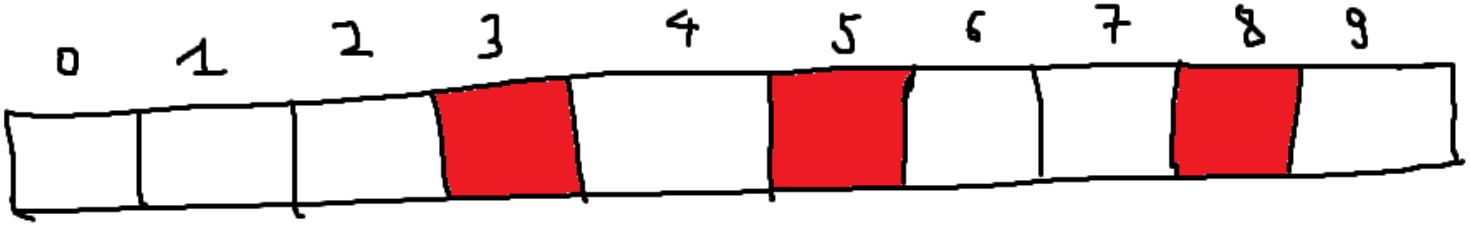
Code 1b

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

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

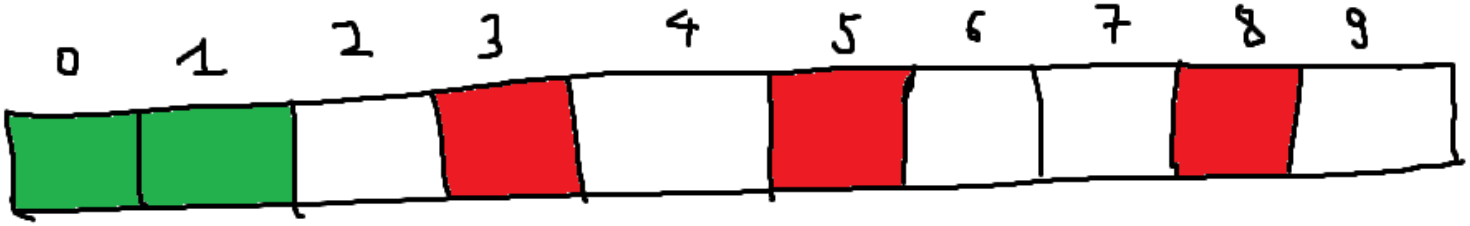
Thuật toán 1c

- 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.



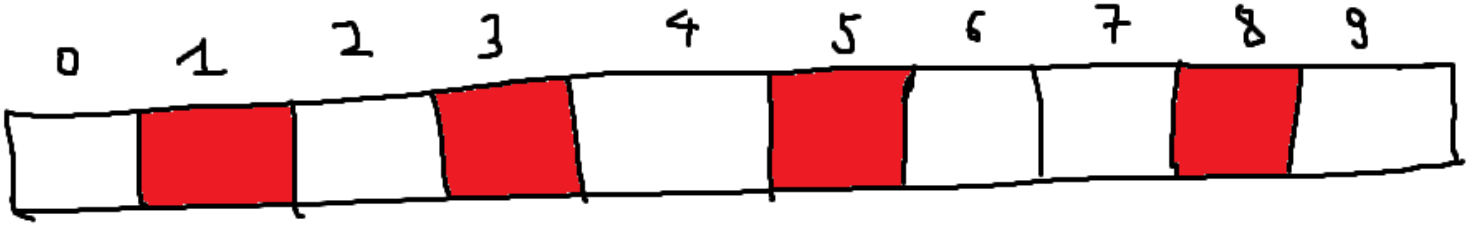
Thuật toán 1c

- 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.



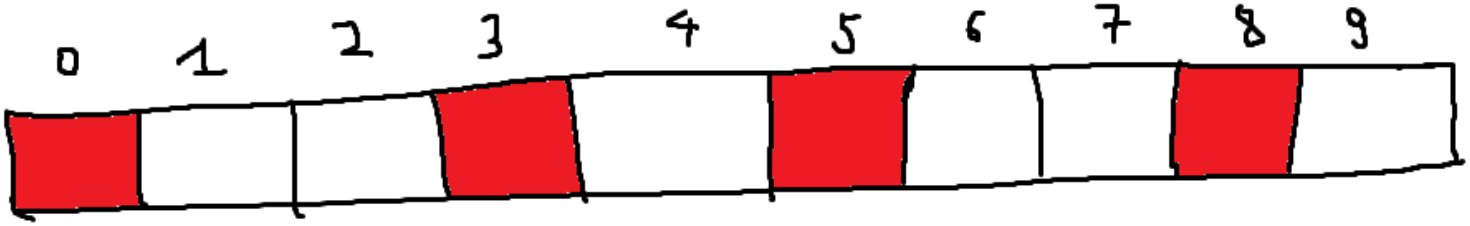
Thuật toán 1c

- 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.

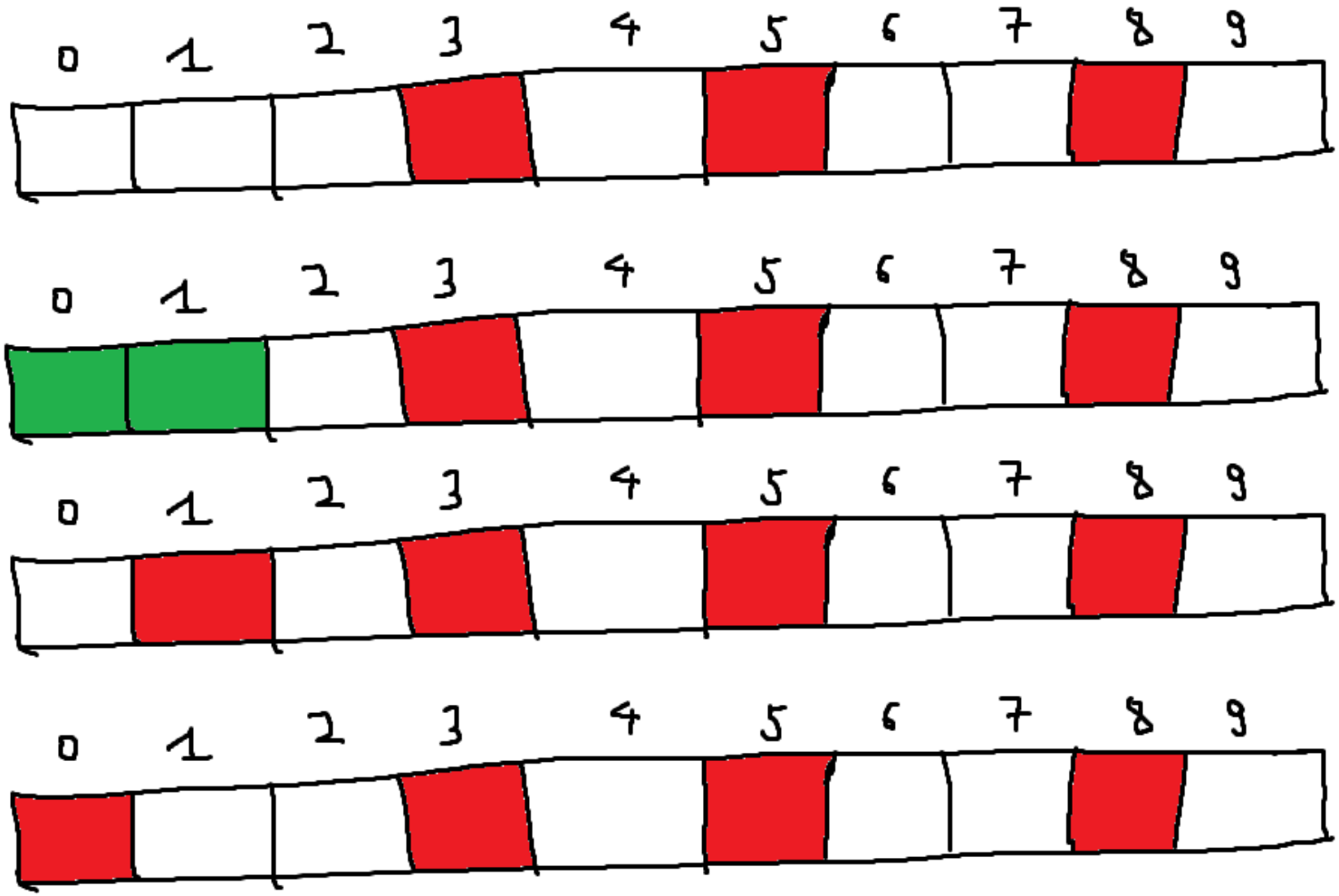


Thuật toán 1c

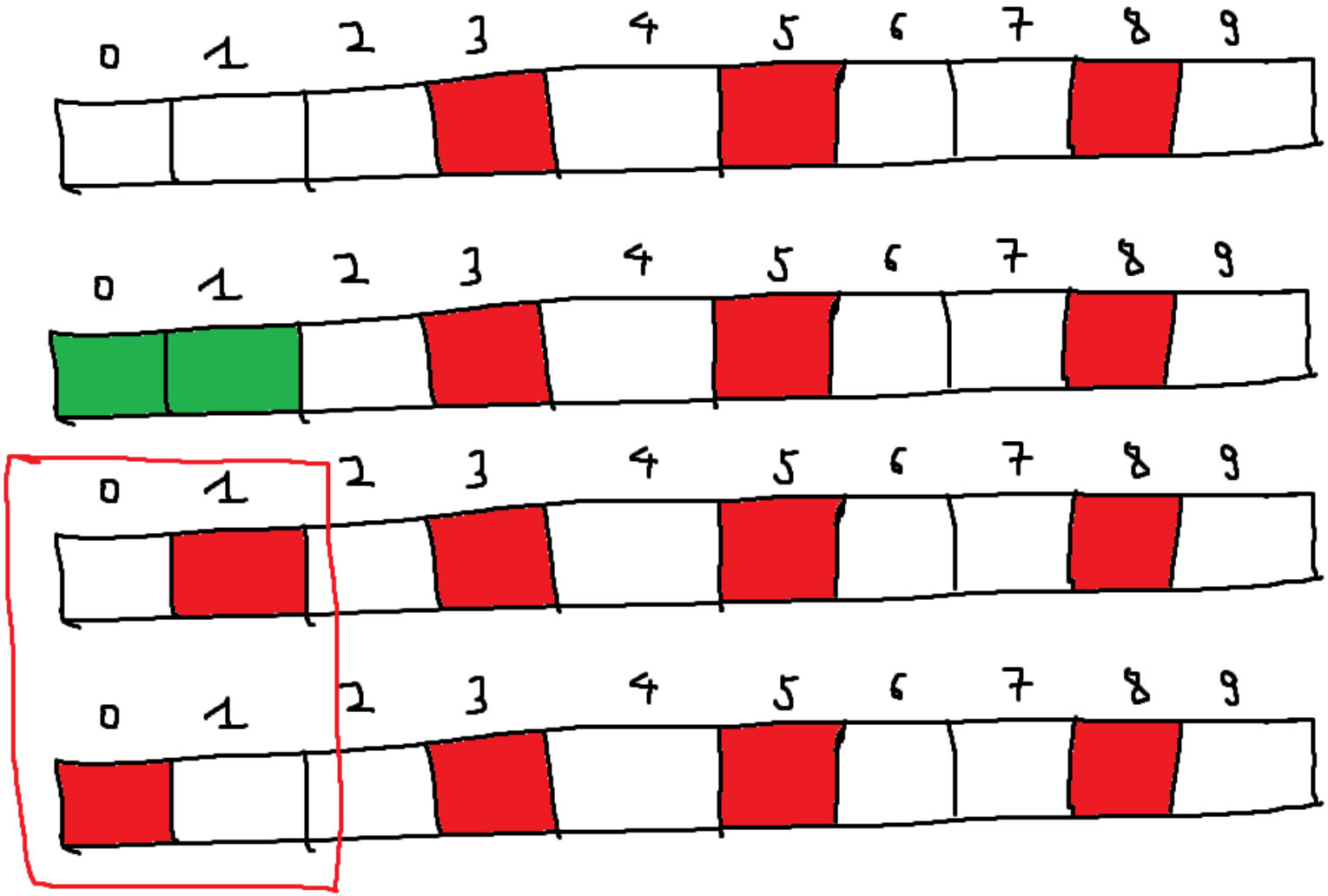
- Sau khi chọn nhà kho x, rõ ràng ta chỉ cần quan tâm đến tổng lượng vàng lớn nhất khi chọn các nhà kho phía trước nhà kho x.



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:

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

- 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 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. 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.
  - $x = 0$: i là ngày nghỉ.
  - $x = 1$: i là ngày cuối cùng đi làm của chu kỳ làm việc tính đến i.
- 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: $F[0][n] + F[1][n]$

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
- $F(0, 0) = 1$
- $$F(i, j) = \sum_{k>0, a[i]*k \leq j} F(i-1, j - a[i] * k)$$
- 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. 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 $1...i \rightarrow$ 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]);
}
```