

---

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**

**KHOA KHOA HỌC MÁY TÍNH**

**ĐỒ ÁN MÔN HỌC**

**NGUYÊN LÝ VÀ PHƯƠNG PHÁP LẬP TRÌNH**

**CHỦ ĐỀ: TỐI ƯU HÓA CHƯƠNG TRÌNH CHO NGÔN NGỮ C++**



**UIT**

Giảng viên hướng dẫn:

Trịnh Quốc Sơn

Người thực hiện:

Phạm Văn Hùng – 21522124

Tp. HCM, ngày 4 tháng 5 năm 2023

## Mục lục

<b>I.</b>	<b>Giới thiệu tối ưu hóa chương trình.....</b>	<b>3</b>
<b>II.</b>	<b>Kỹ thuật tối ưu hóa cho ngôn ngữ C++.....</b>	<b>3</b>
1.	Sử dụng Inline function (Hàm nội tuyến) .....	3
2.	Sử dụng Union và Struct.....	5
3.	Tối ưu hóa I/O.....	6
<b>III.</b>	<b>Kỹ thuật tối ưu hóa cho nhiều ngôn ngữ khác nhau, áp dụng cho ngôn ngữ C++.....</b>	<b>6</b>
1.	Kỹ thuật tối ưu hóa vòng lặp.....	6
2.	Kỹ thuật tối ưu hóa lệnh rẽ nhánh.....	9
3.	Thay đổi cấu trúc dữ liệu .....	11
4.	Giảm số lần gọi một hàm, thủ tục .....	13
5.	Nguyên lý phân cấp bộ nhớ VANWIK .....	15
6.	Nguyên lý KISS, DRY, SOLID.....	17
<b>IV.</b>	<b>Một số giải thuật giúp tối ưu hóa chương trình .....</b>	<b>22</b>
1.	Giải thuật tham lam (Greedy algorithm).....	22
2.	Thuật toán chia để trị (Divide and Conquer algorithm).....	23
3.	Thuật toán quy hoạch động (Dynamic programming).....	25
<b>V.</b>	<b>Chương trình demo đánh giá một số kỹ thuật tối ưu hóa chương trình, chạy trên C++ .....</b>	<b>26</b>
1.	Kỹ thuật tối ưu hóa vòng lặp.....	26
2.	Kỹ thuật sử dụng bảng truy cập, bảng băm.....	27
3.	Thuật toán Quy hoạch động.....	29
<b>VI.</b>	<b>Kết luận.....</b>	<b>31</b>
	Tài liệu tham khảo (đến ngày 4/5/2023) : .....	32

# **Tối ưu hóa chương trình cho ngôn ngữ C++**

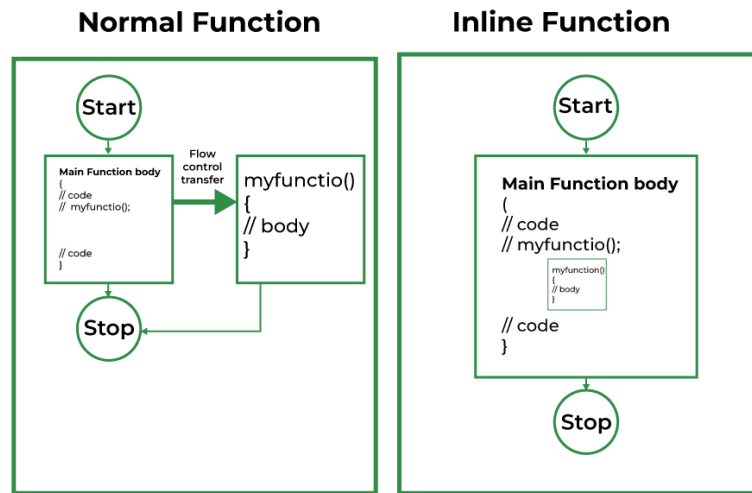
## **I. Giới thiệu tối ưu hóa chương trình**

- Tối ưu hóa chương trình là quá trình cải thiện hiệu suất của một chương trình máy tính bằng cách tối ưu hóa thời gian thực thi hoặc tối ưu hóa sử dụng bộ nhớ.
- Việc tối ưu hóa chương trình nhằm giảm giúp chương trình chạy nhanh hơn, hiệu quả hơn, tối ưu các đoạn mã chương trình ngắn gọn, dễ kiểm soát.
- Có 4 vấn đề cân quan tâm khi tối ưu hóa chương trình:
  - Tính tin cậy: Chương trình phải chạy đúng, mô tả đúng thuật toán.
  - Tính uyển chuyển: Chương trình phải dễ sửa đổi.
  - Tính trong sáng: Chương trình phải dễ đọc, dễ hiểu.
  - Tính hữu hiệu: Chương trình phải chạy nhanh và ít tốn bộ nhớ, tiết kiệm cả không gian và thời gian.

## **II. Kỹ thuật tối ưu hóa cho ngôn ngữ C++**

### **1. Sử dụng Inline function (Hàm nội tuyến)**

- Inline function giảm chi phí khi gọi hàm.

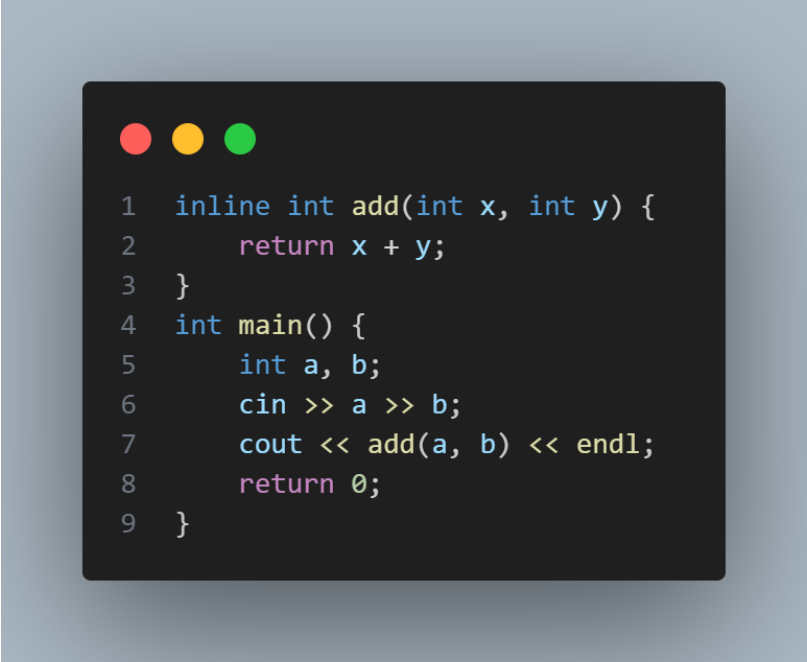


<https://www.geeksforgeeks.org/inline-functions-cpp/>

- Khi dùng inline function, chương trình chèn toàn bộ code của hàm nội tuyến thay thế lời gọi hàm. Như vậy sẽ giúp giảm bớt thời gian truy xuất bộ nhớ so với gọi hàm thông thường.
- Tuy nhiên, hàm nội tuyến không được coi là một lệnh, nên trình biên dịch có thể bỏ qua trong trường hợp:
  - Trong hàm chứa vòng lặp.
  - Trong hàm chứa biến tĩnh.
  - Hàm là đệ quy.
  - Hàm khác void nhưng không có giá trị trả về trong thân hàm.
  - Trong hàm chứa goto hoặc switch.
- Hàm nội tuyến sẽ giúp chương trình nhanh và tiết kiệm chi phí hơn (nếu nó nhỏ).
- Nhận xét: Sử dụng Inline function là kỹ thuật tối ưu hóa tính hữu hiệu của chương trình. Tuy nhiên nên hạn chế sử dụng hàm nội tuyến, vì nó

để làm tăng kích thước của file thực thi do sự trùng lặp các đoạn mã giống nhau.

- Ví dụ:

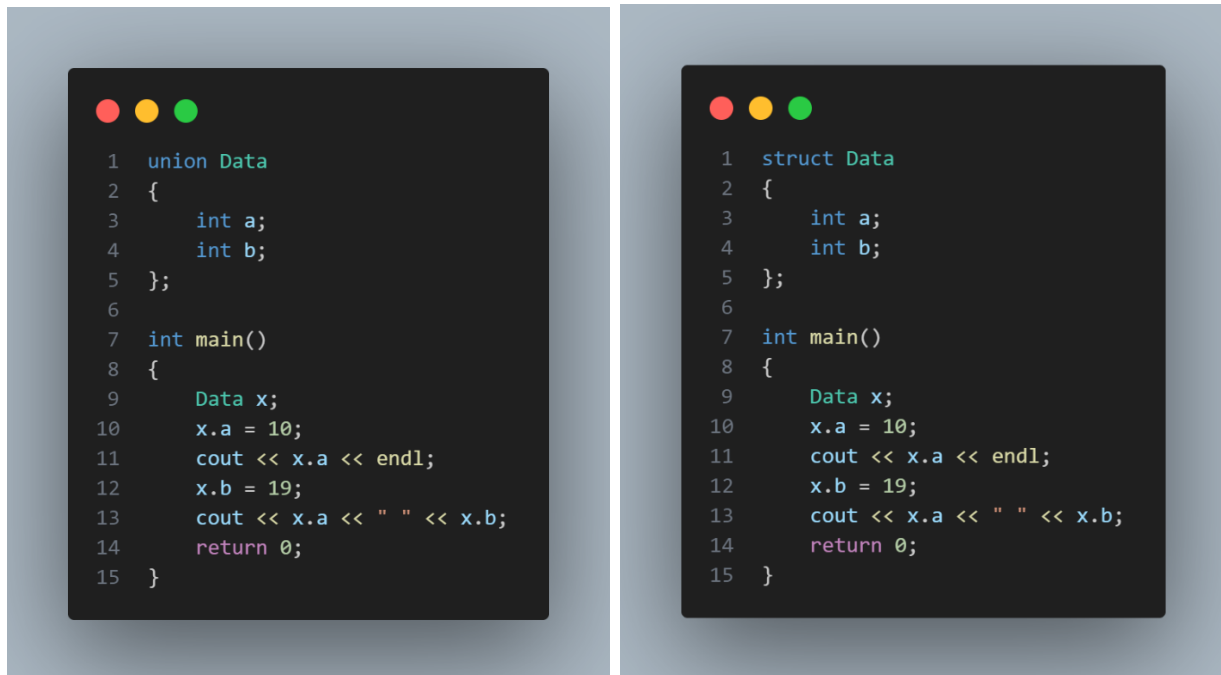


```
1 inline int add(int x, int y) {  
2     return x + y;  
3 }  
4 int main() {  
5     int a, b;  
6     cin >> a >> b;  
7     cout << add(a, b) << endl;  
8     return 0;  
9 }
```

## 2. Sử dụng Union và Struct

- Union tiết kiệm bộ nhớ hơn do các thành viên trong union chia sẻ chung một vùng nhớ, tuy nhiên do dùng chung một vùng nhớ nên dữ liệu bị đè lên nhau.
- Các thành viên của struct được lưu trữ trong vùng nhớ riêng nên kích thước vùng nhớ của struct bằng tổng kích thước các thành viên.
- Nhận xét: Union là tối ưu về tính hữu hiệu (không gian bộ nhớ) của Struct, tuy nhiên nên sử dụng cẩn trọng do dữ liệu có thể bị đè lên nhau.

- Ví dụ: Sự khác nhau giữa union và struct



Kết quả sau khi chạy chương trình:

```

10
19 19

```

```

10
10 19

```

### 3. Tối ưu hóa I/O

- Có thể sử dụng **std::ios\_base::sync\_with\_stdio(false);** ở đầu chương trình để cin/cout không mất thời gian đồng bộ với stdin/stdout.
- Sử dụng **cin.tie(NULL);** để ngăn sự đồng bộ giữa cin và cout. Khi nhập xong hết thì mới xuất (và ngược lại).

## III. Kỹ thuật tối ưu hóa cho nhiều ngôn ngữ khác nhau, áp dụng cho ngôn ngữ C++

### 1. Kỹ thuật tối ưu hóa vòng lặp

- Khi thực thi một vòng lặp, chương trình sẽ thực hiện các thao tác:
  - Khởi tạo giá trị ban đầu cho biến điều khiển vòng lặp

- Kiểm tra điều kiện
- Thực hiện các lệnh trong vòng lặp
- Tăng giá trị biến điều khiển
- Quay lại bước kiểm tra điều kiện
- Để tối ưu hóa vòng lặp, ta nên thực hiện:
  - Tách các lệnh không phụ thuộc chỉ số lặp ra ngoài vòng lặp (Giảm số thao tác phải thực hiện trong mỗi vòng lặp).
  - Giảm số vòng lặp (Giảm số lần phải kiểm tra điều kiện và tăng giá trị biến điều khiển)
  - Vòng lặp có số lần lặp ít thì đưa ra ngoài (Giảm số lần khởi tạo vòng lặp).
  - Thực hiện hợp nhất các vòng lặp nếu có thể (Giảm số vòng lặp).
- Nhận xét: Kỹ thuật tối ưu hóa vòng lặp là kỹ thuật tối ưu về tính hữu hiệu của (thời gian thực thi) của chương trình. Tuy nhiên tính trong sáng và tính uyển chuyển có thể bị ảnh hưởng (do đòi hỏi phải làm nhiều hơn trong một vòng lặp để giảm số lần lặp).
- Ví dụ:

Đoạn chương trình gốc

```

1  for (int i = 0; i < 100; i++)
2      {
3          for (int j = 0; j < 50; j++)
4              {
5                  sum1 += 2 * x + a[i] + b[j];
6              }
7          for (int k = 0; k < 50; k++)
8              {
9                  sum2 += 3 * x + a[i] + k;
10             }
11     }
12

```

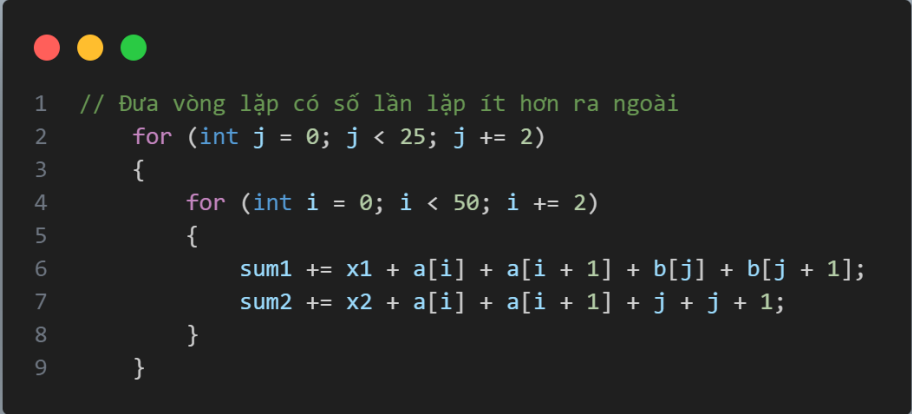
Thực hiện tối ưu hóa

```

1  // Tách các lệnh không phụ thuộc vòng lặp ra bên ngoài
2  int x1 = 2 * x;
3  int x2 = 3 * x;
4  for (int i = 0; i < 50; i += 2) // Giảm số vòng lặp
5      {
6          for (int j = 0; j < 25; j += 2) // Hợp nhất vòng lặp
7              {
8                  sum1 += x1 + a[i] + a[i + 1] + b[j] + b[j + 1];
9                  sum2 += x2 + a[i] + a[i + 1] + j + j + 1;
10             }
11     }

```





```

1  // Đưa vòng lặp có số lần lặp ít hơn ra ngoài
2  for (int j = 0; j < 25; j += 2)
3  {
4      for (int i = 0; i < 50; i += 2)
5      {
6          sum1 += x1 + a[i] + a[i + 1] + b[j] + b[j + 1];
7          sum2 += x2 + a[i] + a[i + 1] + j + j + 1;
8      }
9  }

```

## 2. Kỹ thuật tối ưu hóa lệnh rẽ nhánh

- Phải sắp xếp các điều kiện theo xác suất sai giảm dần:


○  $A_1 \text{ and } A_2 \dots \text{ and } A_n \rightarrow \text{Xác suất sai } A_i \text{ giảm dần}$

Khi  $A_i$  sai thì không cần kiểm tra các điều kiện từ  $A_{i+1}$

○  $A_1 \text{ or } A_2 \dots \text{ or } A_n \rightarrow \text{Xác suất đúng } A_i \text{ giảm dần}$

Khi  $A_i$  đúng thì không cần kiểm tra các điều kiện từ  $A_{i+1}$

- Nhận xét: Kỹ thuật tối ưu hóa lệnh rẽ nhánh là kỹ thuật tối ưu tính hữu hiệu (thời gian thực thi) của chương trình, do giảm số lần phải kiểm tra điều kiện.
- Ví dụ:




```

1  for (int x = 0; x < 100; x++) {
2      for (int y = 0; y < 100; y++) {
3          if (x < 10 and y > 10)
4              cout << x << " " << y << endl;
5      }
6  }

```

- Cân nhắc sử dụng giữa switch - case và if – else.
  - Trong trường hợp cần kiểm tra một biểu thức có nhiều giá trị, nên sử dụng switch - case.
  - Trong trường hợp cần kiểm tra điều kiện đơn giản hoặc kiểm tra nhiều điều kiện thì nên sử dụng if -else.
  - Nhận xét: Việc sử dụng switch – case và if – else không ảnh hưởng đến tính hữu hiệu của chương trình, tuy nhiên nó giúp chương trình tối ưu về tính trong sáng và uyển chuyển.
  - Ví dụ:

Sử dụng if – else:

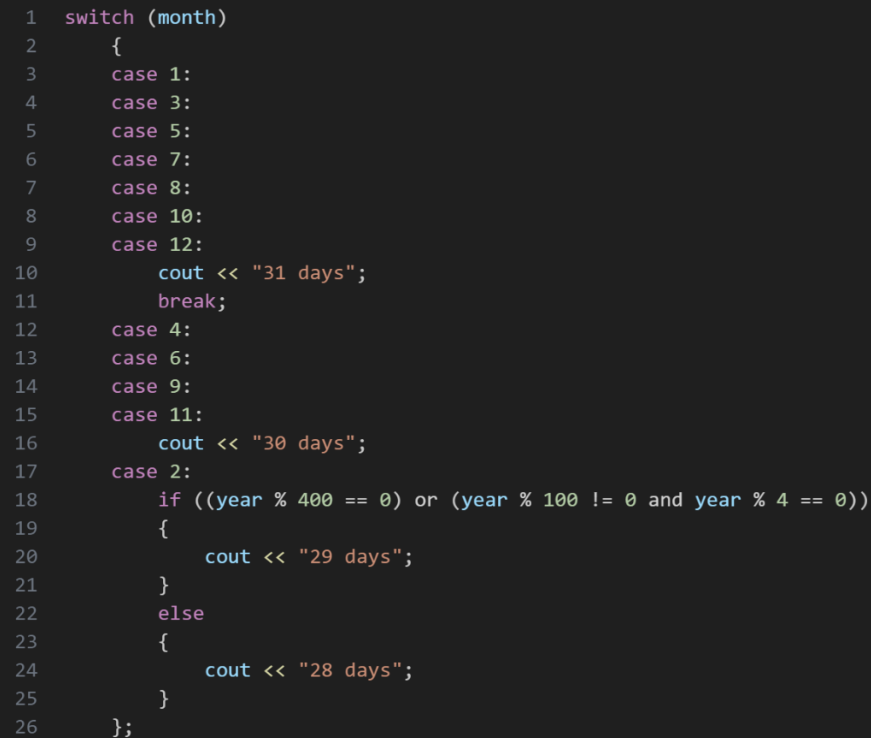


```

1  if (a > 0 and b > 0 and c > 0 and a + b > c and b + c > a and a + c > b) {
2      cout << "a, b, c la 3 canh cua tam giac";
3  }
4  else cout << "a, b, c khong la 3 canh cua tam giac";

```

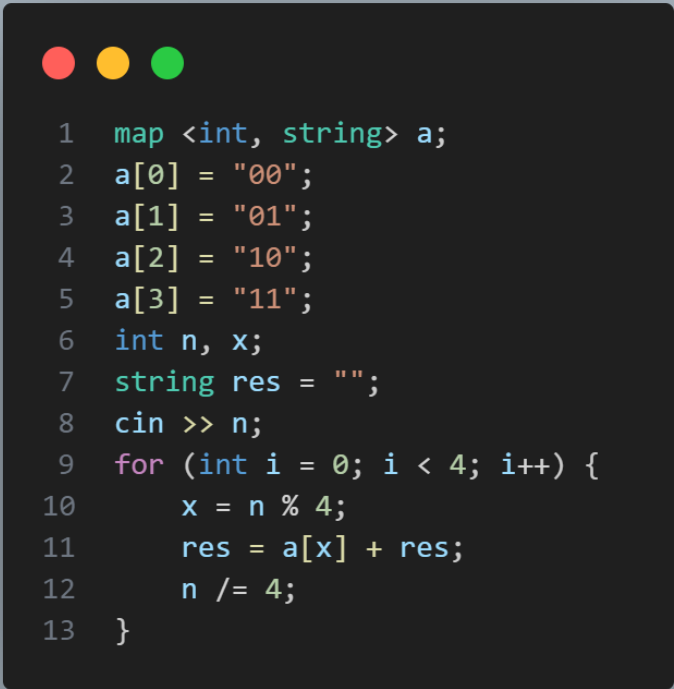
Sử dụng switch – case:



```
1  switch (month)
2  {
3      case 1:
4      case 3:
5      case 5:
6      case 7:
7      case 8:
8      case 10:
9      case 12:
10         cout << "31 days";
11         break;
12     case 4:
13     case 6:
14     case 9:
15     case 11:
16         cout << "30 days";
17     case 2:
18         if ((year % 400 == 0) or (year % 100 != 0 and year % 4 == 0))
19         {
20             cout << "29 days";
21         }
22         else
23         {
24             cout << "28 days";
25         }
26     };
```

### 3. Thay đổi cấu trúc dữ liệu

- Dùng bảng truy cập cho các giá trị phổ biến (table look-up).
  - Lập bảng truy cập cho các giá trị phổ biến.
  - Cách này làm giảm thời gian chạy chương trình đáng kể, tuy nhiên lại làm tăng sử dụng bộ nhớ.
  - Ví dụ:  
Chuyển n thành dãy nhị phân 8 bits.



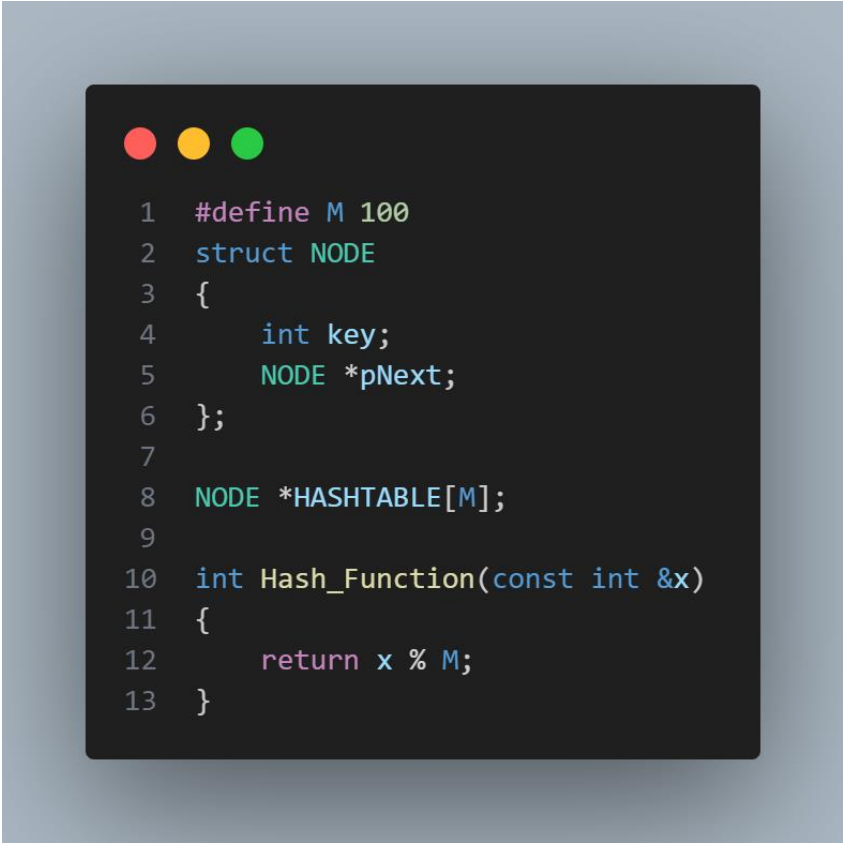
```

1  map <int, string> a;
2  a[0] = "00";
3  a[1] = "01";
4  a[2] = "10";
5  a[3] = "11";
6  int n, x;
7  string res = "";
8  cin >> n;
9  for (int i = 0; i < 4; i++) {
10     x = n % 4;
11     res = a[x] + res;
12     n /= 4;
13 }

```

- Sử dụng bảng băm (Hash table)
  - Băm giúp tổ chức lại cấu trúc dữ liệu của một mảng, danh sách liên kết, ... giúp tăng tốc độ truy xuất đến các phần tử nhanh hơn.
  - Băm bao gồm:
    - Khoá (Key): có thể là một số hoặc một chuỗi số nguyên giúp xác định vị trí lưu trữ một phần tử.
    - Hàm băm (Hash function): ánh xạ các giá trị khóa đến vị trí khóa.
    - Bảng băm (Hash table): là mảng lưu trữ các record, bao gồm các khóa và các giá trị.
  - Phép băm có hiệu quả hay không phụ thuộc vào hàm băm. Khi hàm băm không xảy ra đụng độ và một key ứng chỉ với 1 value, lúc đó bảng băm là 1 table look-up.

- Ví dụ:



```
1  #define M 100
2  struct NODE
3  {
4      int key;
5      NODE *pNext;
6  };
7
8  NODE *HASHTABLE[M];
9
10 int Hash_Function(const int &x)
11 {
12     return x % M;
13 }
```

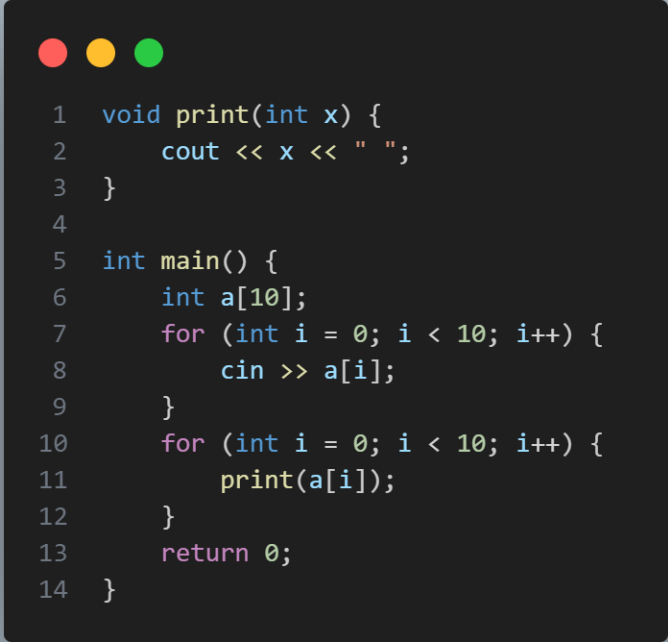
Giải thích: Hàm băm chia các phần tử trong 1 mảng thành các phần khác nhau dựa vào số dư khi chia cho 100.

- Nhận xét: Lập bảng truy cập hay bảng băm là kỹ thuật tối ưu tính hữu hiệu (thời gian thực thi) của chương trình nhưng đồng thời cũng ảnh hưởng đến tính hữu hiệu (không gian bộ nhớ) của chương trình.

#### 4. Giảm số lần gọi một hàm, thủ tục

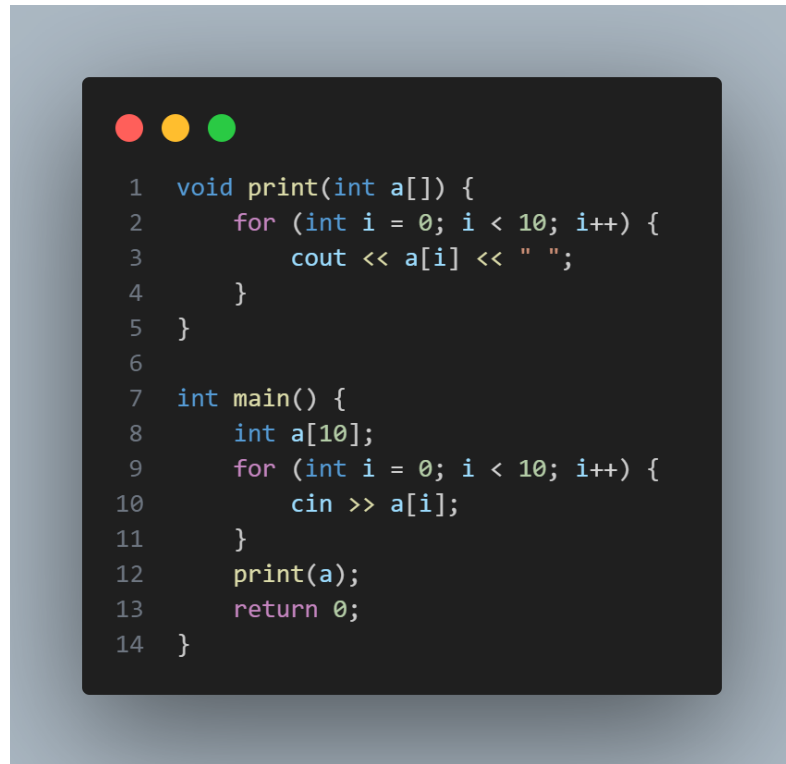
- Khi gọi một hàm / thủ tục, chương trình phải thực hiện các thao tác:
  - Lưu trạng thái hiện tại của chương trình (ví dụ: giá trị của các biến, con trỏ lệnh, địa chỉ trả về) để có thể quay lại sau khi thực thi xong hàm.
  - Nếu hàm có tham số, chương trình truyền giá trị hoặc địa chỉ của các biến vào hàm thông qua các tham số.

- Cấp phát bộ nhớ (nếu hàm có biến cục bộ hoặc đối tượng cần cấp phát bộ nhớ).
- Giải phóng bộ nhớ (nếu cần).
- Phục hồi trạng thái trước.
- Các thao tác đó làm tăng thời gian truy xuất bộ nhớ khi thực thi chương trình.
- Cố gắng làm nhiều nhất trong một hàm, thủ tục. Khi cần sử dụng vòng lặp hãy làm luôn trong hàm / thủ tục.
- Nhận xét: Giảm số lần gọi hàm / thủ tục là kỹ thuật tối ưu tính hữu hiệu của chương trình. Tuy nhiên, sử dụng hàm lại tối ưu tính trong sáng và uyển chuyển của chương trình (làm chương trình ngắn gọn, dễ sửa hơn).
- Ví dụ:



```
1 void print(int x) {
2     cout << x << " ";
3 }
4
5 int main() {
6     int a[10];
7     for (int i = 0; i < 10; i++) {
8         cin >> a[i];
9     }
10    for (int i = 0; i < 10; i++) {
11        print(a[i]);
12    }
13    return 0;
14 }
```

Đoạn chương trình gốc

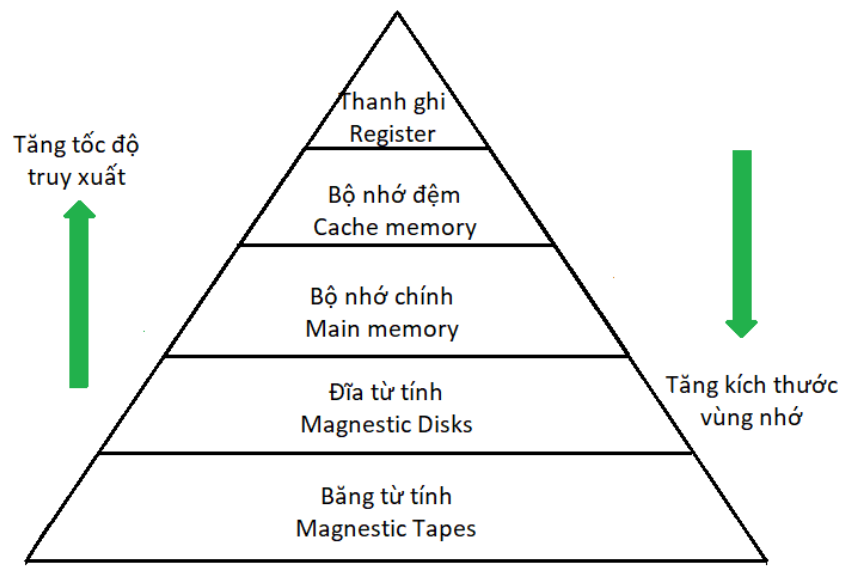


```
1 void print(int a[]) {
2     for (int i = 0; i < 10; i++) {
3         cout << a[i] << " ";
4     }
5 }
6
7 int main() {
8     int a[10];
9     for (int i = 0; i < 10; i++) {
10         cin >> a[i];
11     }
12     print(a);
13     return 0;
14 }
```

Đoạn chương trình tối ưu

## 5. Nguyên lý phân cấp bộ nhớ VANWIK

- Nội dung: Dữ liệu nhỏ và cần truy cập nhiều thì nên lưu vào vùng trên của phân cấp bộ nhớ, dữ liệu lớn và ít cần truy cập thì lưu vào vùng dưới của phân cấp bộ nhớ.
- Hệ thống phân cấp bộ nhớ:



- Thời gian truy cập vào bộ nhớ có dung lượng càng nhỏ thì càng nhanh, càng lớn thì càng chậm.
- Ví dụ: Ta có thể dùng register để lưu trữ biến có kích cỡ tối đa bằng 1 thanh ghi (thường là 1 từ) và không có toán tử một ngôi '&' vì không có địa chỉ bộ nhớ.

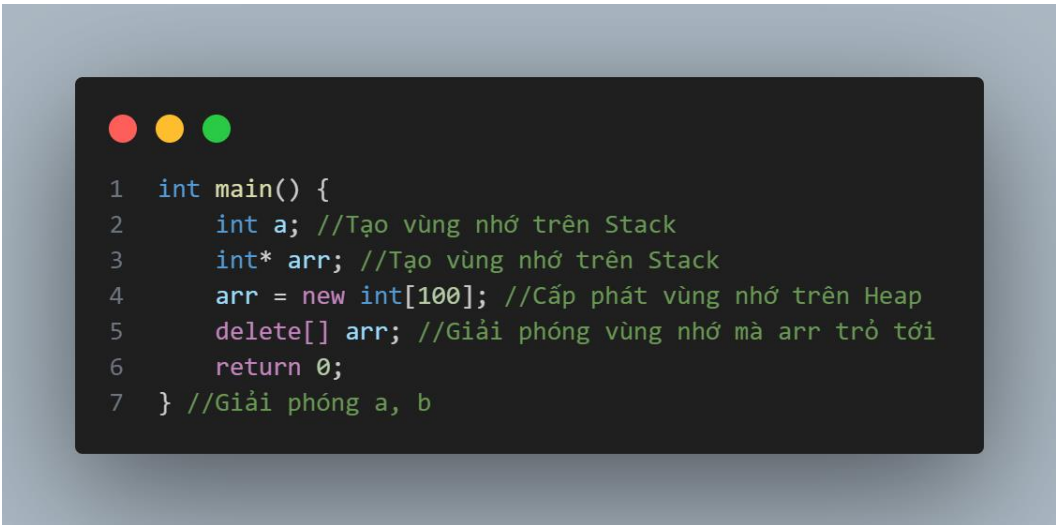
```
register int a = 10;
```

Thường thì ta dùng để lưu các biến đếm. Tuy nhiên từ C++11 thì 'register' đã không còn nữa và không nên sử dụng.

- Tuy nhiên, ta vẫn có thể áp dụng nguyên lý Vanwik trong lập trình C++ bằng cách sử dụng hợp lý Stack và Heap:
  - Stack:
    - Sử dụng để lưu trữ các biến cục bộ
    - Khi kết thúc một hàm, vùng nhớ Stack sẽ được tự động giải phóng



- Kích thước nhỏ hơn
- Tốc độ truy xuất nhanh hơn và được ưu tiên hơn
- Heap:
  - Sử dụng để lưu trữ các biến toàn cục hoặc các đối tượng con trỏ được cấp phát
  - Vùng nhớ Heap phải được giải phóng thông qua hàm bởi lập trình viên.
  - Kích thước lớn hơn
  - Tốc độ truy xuất chậm hơn
- Nhận xét: Nguyên lý phân cấp bộ nhớ Vanwik là kỹ thuật tối ưu tính hữu hiệu của chương trình.
- Ví dụ:



```
1  int main() {
2      int a; //Tạo vùng nhớ trên Stack
3      int* arr; //Tạo vùng nhớ trên Stack
4      arr = new int[100]; //Cấp phát vùng nhớ trên Heap
5      delete[] arr; //Giải phóng vùng nhớ mà arr trỏ tới
6      return 0;
7  } //Giải phóng a, b
```

## 6. Nguyên lý KISS, DRY, SOLID

- Keep it short and simple (KISS)
  - Chia nhỏ code của bạn thành những phần nhỏ và dễ nhìn hơn.
  - Điều này giúp code trông dễ nhìn và dễ sửa hơn.
  - Nhận xét: Nguyên lý KISS là kỹ thuật tối ưu tính trong sáng và uyển chuyển của chương trình.

- Don't Repeat Yourself (DRY)
  - Khi một đoạn mã cần dùng nhiều lần, hãy đưa nó vào hàm hoặc thủ tục và gọi nó khi cần dùng.
  - Nhận xét: Nguyên lý DRY là kỹ thuật tối ưu tính trong sáng của chương trình, tuy nhiên lại làm ảnh hưởng đến tính hữu hiệu của chương trình (Trái ngược với kỹ thuật giảm số lần gọi một hàm / thủ tục).
- SOLID
  - Nguyên lý SOLID giúp tăng tính linh hoạt, bảo trì và mở rộng của mã nguồn, tránh xảy ra bug trong quá trình lập trình.
  - SOLID là viết tắt của 5 chữ cái đầu trong 5 nguyên tắc:
    - Single responsibility principle (SRP)
    - Open/Closed principle (OCP)
    - Liskov substitution principle (LSP)
    - Interface segregation principle (ISP)
    - Dependency inversion principle (DIP)
  - Single responsibility principle (Nguyên tắc trách nhiệm đơn lẻ): Một class chỉ nên giữ một trách nhiệm duy nhất.
  - Open/close principle (Nguyên tắc đóng mở): Khi muốn mở rộng một class, không được sửa đổi bên trong class đó mà phải tạo một class mới kế thừa hoặc sở hữu class đó.
  - Liskov Substitution Principle (Nguyên tắc phân vùng Liskov): Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình.

- Interface Segregation Principle (Nguyên tắc phân tích giao diện): Nếu Interface quá lớn thì nên tách thành các Interface nhỏ hơn, với nhiều mục đích cụ thể.
- Dependency inversion principle (Nguyên tắc đảo ngược phụ thuộc): Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào Interface.
- Nhận xét: 5 nguyên lý SOLID là kỹ thuật tối ưu hóa tính uyển chuyển của chương trình. Tuy nhiên nó lại gây ảnh hưởng đến tính trong sáng của chương trình.
- Ví dụ: Đoạn chương trình gốc

```
1  class Human {
2      void Eat() {
3          cout << "Eating action" << endl;
4      }
5      void Sleep() {
6          cout << "Sleeping action" << endl;
7      }
8      void Play() {
9          cout << "Playing action" << endl;
10     }
11     void Learn() {
12         cout << "Learning action" << endl;
13     }
14     void Work() {
15         cout << "Working action" << endl;
16     }
17 };
```

## Đoạn chương trình sau khi áp dụng SOLID

```
1 // Nguyên tắc đảo ngược phụ thuộc
2 class Action {
3 public:
4     virtual void Act() = 0;
5 };
6
7 //Nguyên tắc trách nhiệm đơn lẻ
8 //Nguyên tắc phân tách giao diện Interface
9 class Eater : public Action {
10 public:
11     void Act() override
12     {
13         cout << "Eating action" << endl;
14     }
15 };
16
17 class Sleeper : public Action {
18 public:
19     void Act() override
20     {
21         cout << "Sleeping action" << endl;
22     }
23 };
24
25 class Player : public Action {
26 public:
27     void Act() override
28     {
29         cout << "Playing action" << endl;
30     }
31 };
```

```
1 class Learner : public Action
2 {
3 public:
4     void Act() override
5     {
6         cout << "Learning action" << endl;
7     }
8 };
9
10 class Worker : public Action {
11 public:
12     void Act() override
13     {
14         cout << "Working action" << endl;
15     }
16 };
17
18 // Nguyên tắc đóng mở
19 class Thinker : public Action {
20 public:
21     void Act() override
22     {
23         cout << "Thinking action" << endl;
24     }
25 };
26
27 class Human : public Action {
28 public:
29     // Nguyên tắc phân vùng Likov
30     void Act() override
31     {
32         Eater().Act();
33         Sleeper().Act();
34         Player().Act();
35         Learner().Act();
36         Worker().Act();
37         Thinker().Act();
38     }
39 };
```

#### **IV. Một số giải thuật giúp tối ưu hóa chương trình**

Một bài toán có phương pháp để xử lý và luôn có phương pháp tối ưu hơn những phương pháp còn lại. Vậy nên việc chọn đúng giải thuật để giải một bài toán có ảnh hưởng rất nhiều đến độ tối ưu trong việc giải bài toán đó.

Một số giải thuật tôi cảm thấy có thể sẽ giúp tối ưu trong việc xử lý một bài toán là:

##### **1. Giải thuật tham lam (Greedy algorithm)**

- Tham lam xây dựng lời giải cho một bài toán có nhiều bước giải, luôn lựa chọn hành động tốt nhất cho từng bước.
- Nhận xét: Giải thuật tham lam là phương pháp tối ưu tính hữu hiệu (thời gian thực thi) của chương trình. Tuy nhiên giải thuật tham lam có thể ảnh hưởng đến tính chính xác do khó có thể chứng minh tính đúng đắn của hàm dự đoán.
- Ví dụ: Bài toán Knapsack

```

1  struct Item
2  {
3      int price;
4      int weight;
5      double value;
6  };
7
8  int Knapsack(Item a[], int n, int w)
9  {
10     int result = 0;
11     for (int i = 0; i < n; i++)
12     {
13         a[i].value = (double)a[i].price / (double)a[i].weight;
14     }
15     for (int i = 0; i < n - 1; i++)
16     {
17         for (int j = i + 1; j < n; j++)
18         {
19             if (a[i].value < a[j].value)
20             {
21                 swap(a[i], a[j]);
22             }
23         }
24     }
25     for (int i = 0; i < n; i++)
26     {
27         if (a[i].weight < w)
28         {
29             w -= a[i].weight;
30             result += a[i].price;
31         }
32         else
33         {
34             break;
35         }
36     }
37     return result;

```

## 2. Thuật toán chia để trị (Divide and Conquer algorithm)

- Chia để trị giải một bài toán có nhiều bước giải bằng cách xử lý từng bước và kết hợp lại.
- Kỹ thuật này chia làm 3 phần:
  - Chia (Divide): Chia bài toán thành các bài toán con.
  - Trị (Conquer): Giải các bài toán con bằng đệ quy cho đến khi được giải quyết.

- Kết hợp (Combine): Kết hợp các bài toán con lại để tìm ra lời giải cho bài toán ban đầu.
- Nhận xét: Chia để trị không làm ảnh hưởng đến tính tin cậy của chương trình. Tuy nhiên, lời giải cung cấp bằng phương pháp chia để trị có thể tối ưu về tính hữu hiệu (thời gian thực thi) hoặc không trong từng trường hợp. Ví dụ với bài toán tìm kiếm một phần tử trong một mảng, nếu phần tử nằm ở giữa thì tìm kiếm nhị phân (áp dụng chia để trị) sẽ nhanh hơn, nếu phần tử nằm ở đầu mảng thì tìm kiếm tuyến tính sẽ nhanh hơn.
- Ví dụ: Quick sort



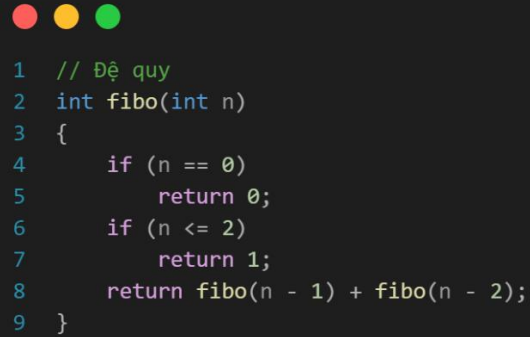
```

1 void QuickSort(int a[], int left, int right)
2 {
3     int i, j, pivot;
4     pivot = a[(left + right) / 2];
5     i = left;
6     j = right;
7     while (i <= j)
8     {
9         while (a[i] < pivot)
10             i++;
11         while (a[j] > pivot)
12             j--;
13         if (i <= j)
14         {
15             swap(a[i], a[j]);
16             i++;
17             j--;
18         }
19     }
20     if (left < j)
21         QuickSort(a, left, j);
22     if (i < right)
23         QuickSort(a, i, right);
24 }
25
26 void Sort(int a[], int n) {
27     QuickSort(a, 0, n - 1);
28 }

```

### 3. Thuật toán quy hoạch động (Dynamic programming)

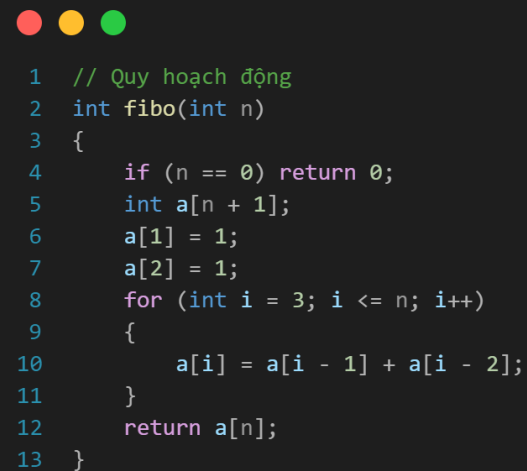
- Dùng để tối ưu bài toán đệ quy đơn giản bằng cách lưu trữ một phần hoặc toàn bộ kết quả ở mỗi bước với mục đích sử dụng lại.
- Nhận xét: Quy hoạch động giúp bài toán tối ưu về tính hữu hiệu (thời gian thực thi) tuy nhiên cũng ảnh hưởng đến tính hữu hiệu (không gian bộ nhớ) của chương trình.
- Ví dụ: Tính số fibonacci thứ n



```

1 // Đệ quy
2 int fibo(int n)
3 {
4     if (n == 0)
5         return 0;
6     if (n <= 2)
7         return 1;
8     return fibo(n - 1) + fibo(n - 2);
9 }

```



```

1 // Quy hoạch động
2 int fibo(int n)
3 {
4     if (n == 0) return 0;
5     int a[n + 1];
6     a[1] = 1;
7     a[2] = 1;
8     for (int i = 3; i <= n; i++)
9     {
10         a[i] = a[i - 1] + a[i - 2];
11     }
12     return a[n];
13 }

```

## V. Chương trình demo đánh giá một số kĩ thuật tối ưu hóa chương trình, chạy trên C++

### 1. Kĩ thuật tối ưu hóa vòng lặp

```

1  #include <iostream>
2  #include <chrono>
3  #include <cmath>
4  using namespace std;
5
6  int main() {
7      long long n = 30000;
8      long long sum1 = 0, sum2 = 0;
9      int x = 8;
10
11     auto start1 = chrono::system_clock::now();
12     for (int i = 0; i < n; i++) {
13         for (int j = 0; j < n; j++) {
14             sum1 += 2 * x + i + j;
15         }
16         for (int k = 0; k < n; k++) {
17             sum2 += 3 * x + i - k;
18         }
19     }
20     auto end1 = chrono::system_clock::now();
21     chrono::duration<double> elapsed_seconds1 = end1 - start1;
22     cout << "Thời gian thực hiện: " << elapsed_seconds1.count() << " giây" << std::endl;
23
24     auto start2 = chrono::system_clock::now();
25     int a = 2 * x;
26     int b = 3 * x;
27     for (int i = 0; i < n/2; i+=2) {
28         for (int j = 0; j < n/2; j+=2) {
29             sum1 += a + i + i + 1 + j + j + 1;
30             sum2 += b + i + i + 1 - j - j - 1;
31         }
32     }
33     auto end2 = chrono::system_clock::now();
34     chrono::duration<double> elapsed_seconds2 = end2 - start2;
35     cout << "Thời gian thực hiện: " << elapsed_seconds2.count() << " giây" << std::endl;
36 }

```

Thời gian thực hiện: 2.85562 giây

Thời gian thực hiện: 0.119825 giây

Đánh giá: Với giá trị  $n = 30000$ , thời gian chương trình giảm đáng kể (gần 24 lần) khi sử dụng kỹ thuật tối ưu hóa vòng lặp.

## 2. Kỹ thuật sử dụng bảng truy cập, bảng băm

```

1  #include <iostream>
2  #include <chrono>
3  #include <cmath>
4  #include <map>
5  using namespace std;
6
7  int main() {
8      int n = 1000000;
9      int m = n;
10
11     string res = "";
12     int x;
13     auto start1 = chrono::system_clock::now();
14     string a[16];
15     a[0] = "0000";
16     a[1] = "0001";
17     a[2] = "0010";
18     a[3] = "0011";
19     a[4] = "0100";
20     a[5] = "0101";
21     a[6] = "0110";
22     a[7] = "0111";
23     a[8] = "1000";
24     a[9] = "1001";
25     a[10] = "1010";
26     a[11] = "1011";
27     a[12] = "1100";
28     a[13] = "1101";
29     a[14] = "1110";
30     a[15] = "1111";
31     for (int i = 0; i < 80000; i++) {
32         x = m % 16;
33         res = a[x] + res;
34         m /= 16;
35     }
36     auto end1 = chrono::system_clock::now();
37     chrono::duration<double> elapsed_seconds1 = end1 - start1;
38     cout << endl << "Thoi gian thuc hien: " << elapsed_seconds1.count() << " giay" << endl;
39     res = "";
40     m = n;
41     auto start2 = chrono::system_clock::now();
42     string b[2];
43     b[0] = "0";
44     b[1] = "1";
45     for (int i = 0; i < 320000; i++) {
46         x = m % 2;
47         res = b[x] + res;
48         m /= 2;
49     }
50     auto end2 = chrono::system_clock::now();
51     chrono::duration<double> elapsed_seconds2 = end2 - start2;
52     cout << endl << "Thoi gian thuc hien: " << elapsed_seconds2.count() << " giay" << endl;
53     cout << endl << "Chenh lech khong gian bo nho: " << sizeof(a) - sizeof(b) << " bytes\n";
54     return 0;
55 }

```

```
Thời gian thực hiện: 0.22338 giây  
Thời gian thực hiện: 0.907366 giây  
Chênh lệch không gian bộ nhớ: 448 bytes
```

Đánh giá: Tốn thêm 448 bytes không gian dữ liệu để chương trình in ra dãy bits 320000 chữ số nhanh hơn 0.7 giây (gấp hơn 4 lần).

### 3. Thuật toán Quy hoạch động

```

1  #include <iostream>
2  #include <chrono>
3  #include <cmath>
4
5  using namespace std;
6
7  // Quy hoạch động
8  long long dynamic(int n, int &x)
9  {
10     if (n == 0) return 0;
11     long long a[n + 1];
12     a[1] = 1;
13     a[2] = 1;
14     for (int i = 3; i <= n; i++)
15     {
16         a[i] = a[i - 1] + a[i - 2];
17     }
18     x = sizeof(a);
19     return a[n];
20 }
21
22 // Đệ quy
23 long long recuse(int n)
24 {
25     if (n == 0)
26         return 0;
27     if (n <= 2)
28         return 1;
29     return recuse(n - 1) + recuse(n - 2);
30 }
31
32 int main() {
33     int n;
34     cin >> n;
35     auto start1 = chrono::system_clock::now();
36     recuse(n);
37     auto end1 = chrono::system_clock::now();
38     chrono::duration<double> elapsed_seconds1 = end1 - start1;
39     cout << endl << "Thời gian thực hiện: " << elapsed_seconds1.count() << " giây" << endl;
40
41     auto start2 = chrono::system_clock::now();
42     int x;
43     dynamic(n,x);
44     auto end2 = chrono::system_clock::now();
45     chrono::duration<double> elapsed_seconds2 = end2 - start2;
46     cout << endl << "Thời gian thực hiện: " << elapsed_seconds2.count() << " giây" << endl;
47     cout << endl << "Không gian bộ nhớ: " << x << " bytes" << endl;
48     return 0;
49 }

```

```
45
Thoi gian thuc hien: 3.54291 giay
Thoi gian thuc hien: 4e-07 giay
Khong gian bo nho: 368 bytes
```

Đánh giá: Để in ra số fibonacci thứ 45, thuật toán đệ quy cần 3.5 giây để thực hiện trong khi đó quy hoạch động không cần 1 mili giây để thực thi và tốn thêm 368 bytes.

## VI. Kết luận

Tối ưu hoá chương trình giúp chương trình chạy nhanh hơn, tốn ít bộ nhớ hơn, dễ đọc và dễ sửa chữa hơn. Tuy nhiên tối ưu chương trình đòi hỏi người lập trình có kỹ năng nếu không sẽ dễ gây phản tác dụng (do các phương pháp tối ưu thường là tối ưu về mặt này và bù trừ vào mặt khác) hoặc làm chương trình chạy không chính xác.

Tài liệu tham khảo (đến ngày 4/5/2023) :

- (1) *Giải thuật và lập trình – Lê Minh Hoàng, Phần 2 Chương 1*
- (2) <https://www.geeksforgeeks.org/inline-functions-cpp/>
- (3) <https://vi.jf-parede.pt/memory-hierarchy-computer-architecture>
- (4) <https://codelearn.io/sharing/heap-va-stack-khac-biet-den-nhu-the-nao>
- (5) <https://codelearn.io/sharing/ap-dung-nguyen-tac-solid-nhu-the-nao>
- (6) <https://www.geeksforgeeks.org/fundamentals-of-algorithms/>