MÔN CƠ SỞ LẬP TRÌNH

BÀI TẬP THỰC HÀNH 5:

Tạ Gia Huy – 23IT6 – BIT230195

Bài 1: Sắp xếp mảng – Viết chương trình sắp xếp mảng sử dụng Bubble sort và Selection sort. So sánh hiệu xuất 2 thuật toán bằng cách đo thời gian thực thi

Bài làm

- Bubble sort là gì ?

Bubble sort là một thuật toán sắp xếp đơn giản trong khoa học máy tính. Nó hoạt động bằng cách so sánh lần lượt các cặp phần tử liền kề trong danh sách và hoán đổi chúng nếu chúng không được sắp xếp đúng thứ tự. Quá trình này được lặp cho đến khi không còn cặp phần tử nào cần được hoán đổi nữa.

Cách hoạt động của Bubble sort như sau:

1. Bắt đầu từ đầu danh sách, so sánh phần tử đầu tiên với phần tử thứ hai. Nếu phần tử đầu tiên lớn hơn phần tử thứ hai, hoán đổi chúng.

2. Tiếp tục di chuyển xuống qua danh sách, so sánh phần tử thứ hai với phần tử thứ ba. Nếu phần tử thứ hai lớn hơn phần tử thứ ba, hoán đổi chúng

3. Tiếp tục quá trình này cho đến khi bạn so sánh phần tử thứ n-1 với phần tử thứ n (n là số phần tử trong danh sách).

4. Khi bạn hoàn thành vòng lặp đầu tiên, phần tử lớn nhất sẽ đã được đặt ở vị trí cuối cùng của danh sách.

5. Lặp lại các bước trên cho đến khi không còn cặp phần tử nào cần được hoán đổi trong quá trình duyệt qua toàn bộ danh sách.

6. Khi không còn hoán đổi nào xảy ra trong một vòng lặp, danh sách được coi là đã được sắp xếp và thuật toán dừng lại.

Bubble sort có độ phức tạp thời gian là O(n^2), trong đó n là số lượng phần tử trong danh sách. Điều này đồng nghĩa với việc thời gian thực hiện Bubble sort tăng theo bình phương của số lượng phần tử. Do đó, thuật toán này không hiệu quả với các danh sách lớn. Tuy nhiên, vì tính đơn giản và dễ hiểu, Bubble sort vẫn được sử dụng trong các ví dụ minh họa và giảng dạy cơ bản về sắp xếp.

Code mẫu :

```
void bubble_sort(int arr[], int size)
{
    int temp = 0;
    for (int j = 0; j < size - 1; j++)
    {
        for (int i = 0; i < size - j - 1; i++)
        {
            if (arr[i] > arr[i + 1])
            {
                temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
            }
        }
    }
}
```

- Selection sort là gì ?

Selection sort là một thuật toán sắp xếp đơn giản trong khoa học máy tính. Nó hoạt động bằng cách tìm kiếm phần tử nhỏ nhất (hoặc lớn nhất) trong danh sách và đặt nó vào vị trí đầu tiên. Sau đó, tiếp tục tìm kiếm phần tử nhỏ nhất (hoặc lớn nhất) trong phần còn lại của danh sách và đặt nó vào vị trí tiếp theo. Quá trình này được lặp lại cho đến khi danh sách được sắp xếp.

Cách hoạt động của Selection sort như sau:

1. Bắt đầu từ vị trí đầu tiên của danh sách, tìm kiếm phần tử nhỏ nhất (hoặc lớn nhất) trong toàn bộ danh sách.

2. Hoán đổi phần tử nhỏ nhất (hoặc lớn nhất) với phần tử ở vị trí đầu tiên của danh sách.

3. Di chuyển đến vị trí tiếp theo trong danh sách và tìm kiếm phần tử nhỏ nhất (hoặc lớn nhất) trong phần còn lại của danh sách.

4. Tiếp tục quá trình này cho đến khi bạn đã duyệt qua toàn bộ danh sách.

5. Khi bạn hoàn thành vòng lặp cuối cùng, danh sách sẽ đã được sắp xếp theo thứ tự tăng dần (hoặc giảm dần).

Selection sort có độ phức tạp thời gian là O(n^2), trong đó n là số lượng phần tử trong danh sách. Điều này có nghĩa là thời gian thực hiện Selection sort tăng theo bình phương của số lượng phần tử.

Do đó, thuật toán này không hiệu quả với các danh sách lớn. Tuy nhiên, nhược điểm này được bù lại bởi tính đơn giản và dễ hiểu của thuật toán.

Lưu ý rằng Selection sort có thể được thực hiện theo hai phiên bản: Selection sort tăng dần (sắp xếp theo thứ tự tăng dần) và Selection sort giảm dần (sắp xếp theo thứ tự giảm dần). Cả hai phiên bản đều hoạt động theo cùng một cách thức, chỉ khác nhau ở việc tìm kiếm phần tử nhỏ nhất hoặc lớn nhất để đặt vào vị trí phù hợp.

Code mẫu :

```
void selectionSort(int arr[], int size)
{
    int i, j, c, minIndex;

    for (i = 0; i < size - 1; i++)
    {
        minIndex = i;
        for (j = i + 1; j < size; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }
        c = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = c;
    }
}
```

Tiếp theo: để tính thời gian thực thi 2 lệnh trên tôi đã sử dụng đoạn code sau :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
```

```
                arr[j + 1] = temp;
            }
        }
    }
}

void selectionSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

int main() {
    int arr[] = {      };
    int n = sizeof(arr) / sizeof(arr[0]);

    clock_t start, end;
    double cpu_time_used;

    // Đo thời gian thực thi cho Bubble sort
    start = clock();
    bubbleSort(arr, n);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Thoi gian thuc thi cua Bubble sort: %f\n", cpu_time_used);

    // Đo thời gian thực thi cho Selection sort
    start = clock();
    selectionSort(arr, n);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Thoi gian thuc thi cua Selection sort: %f\n", cpu_time_used);

    return 0;
}
```

```
    int arr[] = {       };
```

ở đây tôi cho mảng arr khoảng chục ngàn số để tăng độ phức tạp của bài toán

Và kết quả cho ra là :

```
Thoi gian thuc thi cua Bubble sort: 0.007000
Thoi gian thuc thi cua Selection sort: 0.006000
```

Vậy thời gian thực hiện của Selection sort nhanh hơn Bubble sort

Bài 2: Tìm kiếm trong mảng – viết chương trình tìm kiếm tuyến tính và nhị phân cho giá trị x trong mảng, so sánh hiệu suất của 2 phương pháp

- Tìm kiếm tuyến tính là gì ?

Tìm kiếm tuyến tính (linear search) trong mảng là quá trình tìm kiếm một phần tử cụ thể trong mảng bằng cách kiểm tra từng phần tử trong mảng theo thứ tự từ đầu đến cuối cho đến khi tìm thấy phần tử cần tìm hoặc kiểm tra hết tất cả các phần tử trong mảng mà không tìm thấy.

Thuật toán tìm kiếm tuyến tính hoạt động bằng cách so sánh phần tử cần tìm với từng phần tử trong mảng. Nếu phần tử được tìm thấy, thuật toán trả về vị trí của phần tử trong mảng. Nếu không tìm thấy, thuật toán trả về một giá trị đặc biệt (thông thường là -1 hoặc một giá trị không hợp lệ) để chỉ ra rằng phần tử không tồn tại trong mảng.

Code mẫu:

```
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++){
        if (arr[i] == target) {
            return i;
        }
    }

    return -1;
}
```

- Tìm kiếm nhị phân là gì?

Tìm kiếm nhị phân (binary search) là một thuật toán tìm kiếm được áp dụng trên các mảng đã được sắp xếp theo thứ tự tăng dần (hoặc giảm dần). Thuật toán này hoạt động bằng cách chia mảng thành hai nửa, sau đó so sánh phần tử cần tìm với phần tử ở giữa mảng. Nếu phần tử cần tìm bằng phần tử ở giữa, thuật toán trả về vị trí tìm thấy. Nếu phần tử cần tìm nhỏ hơn phần tử ở giữa, thuật toán tiếp tục tìm kiếm trong nửa đầu tiên của mảng. Nếu phần tử cần tìm lớn hơn phần tử ở giữa, thuật toán tiếp tục tìm kiếm trong nửa thứ hai của mảng. Quá trình này lặp lại cho đến khi tìm thấy phần tử cần tìm hoặc xác định rằng phần tử không tồn tại trong mảng.

Thuật toán tìm kiếm nhị phân hoạt động hiệu quả trên các mảng đã được sắp xếp, vì nó loại bỏ một nửa các phần tử trong mỗi lần so sánh. Điều này dẫn đến hiệu suất tìm kiếm tốt hơn so với tìm kiếm tuyến tính, đặc biệt là trên các mảng lớn.

Code mẫu :

```
int binarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // Trả về vị trí của phần tử trong mảng
        }

        if (arr[mid] < target) {
            left = mid + 1; // Tiếp tục tìm kiếm trong nửa thứ hai của mảng
        } else {
            right = mid - 1; // Tiếp tục tìm kiếm trong nửa đầu tiên của mảng
        }
    }

    return -1; // Trả về -1 nếu không tìm thấy phần tử
}
```

Nhưng do cách tìm kiếm này chỉ hiệu quả đối với một mảng được sắp xếp từ trước, vì thế khi ta cần tìm dữ liệu trên một mảng bất kỳ chưa được sắp xếp, chúng ta lại phải sử dụng lại hàm sắp xếp như Bubble sort hoặc Selection sort như ở bài 1. Dẫn đến bài toán chở nên phức tạp hơn và chậm hơn so với phương pháp tìm kiếm tuyến tính

Dưới đây chúng ta sẽ so sánh trường hợp cần phải tìm dữ liệu trong một mảng chưa được sắp xếp từ trước

Sau đây là Code để tìm được thời gian thực hiện của 2 thuật toán:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int binarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // Trả về vị trí của phần tử trong mảng
        }

        if (arr[mid] < target) {
            left = mid + 1; // Tiếp tục tìm kiếm trong nửa thứ hai của mảng
        } else {
            right = mid - 1; // Tiếp tục tìm kiếm trong nửa đầu tiên của mảng
        }
    }

    return -1; // Trả về -1 nếu không tìm thấy phần tử
}

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++){
        if (arr[i] == target) {
            return i;
        }
    }

    return -1;
}

void selectionSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
```

```
                min_idx = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}


int main() {
    int arr[] = {};
    int target = ;
    int size = sizeof(arr) / sizeof(arr[0]);

    clock_t start, end;
    double cpu_time_used;

    // Đo thời gian thực thi cho Bubble sort
    start = clock();
    int result1 = linearSearch(arr, size, target);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Thoi gian thuc thi cua linearSearch sort: %f\n", cpu_time_used);

    // Đo thời gian thực thi cho Selection sort
    start = clock();
    selectionSort(arr,size);
    int result2 = binarySearch(arr, size, target);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Thoi gian thuc thi cua binarySearch sort: %f\n", cpu_time_used);
    printf("%d , % d",result1, result2);
    return 0;
}
```

Tôi đã thu được kết quả sau đây

```
Thoi gian thuc thi cua linearSearch sort: 0.000000
Thoi gian thuc thi cua binarySearch sort: 0.005000
```

Với arr là một dãy chứa hàng nghìn chữ số

Ta thấy rằng binarySearch cùng với selectionSort kết hợp tạo nên thuật toán có sự phức tạp cao hơn

Bài 3: Viết chương trình nhập điểm của sinh viên trong một lớp học vào một mảng, tính và in ra điểm trung bình của lớp

```c
#include <stdio.h>
float GPA(int arr[], int size) {
    int GPA = 0;
    for (int i = 0; i < size; i++) {
        GPA += arr[i];
    }
    return GPA / (float)size;
}

int main() {
    int size;
    printf("lop hoc cua ban co bao nhieu SV ? :\n");
    scanf("%d", &size);

    int arr[size];
    for (int i = 0; i < size; i++) {
        printf("Nhap diem nguoi thu %d : ", i+1);
        scanf("%d", &arr[i]);
        printf("\n");
    }

    float gpa = GPA(arr, size);
    printf("diem trung binh cua ca lop la : %f", gpa);


    return 0;
}
```