**Đại học Quốc gia TP Hồ Chí Minh**

**Đại học Khoa học tự nhiên TP Hồ Chí Minh**

**Khoa Công nghệ thông tin**

**Bộ môn Công nghệ phần mềm**



**Môn học:** CTDL & GT

**Năm học:** 2018 - 2019

**Học kì:** Học Kì I

**GVHD:**

**BÁO CÁO ĐỒ ÁN**

# Thông tin sinh viên thực hiện

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **STT** | **MSSV** | **Họ và tên** | **Email** | **Điện thoại** | **Kí tên** |
| 1 | 1712884 | Nguyễn Thanh Tùng | Nguyenthanhtung884318@gmail.com | 0868042318 | Tung |

# Phần báo cáo

1. **Trình bày các thuật toán**
2. **Selection Sort:**

* **Ý tưởng:** Ý tưởng của thuật toán chọn trực tiếp mô phỏng một trong những cách sắp xếp tự nhiên nhất trong thực tế: chọn phần tử nhỏ nhất trong N phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu dãy hiện hành; sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn N-1 phần tử của dãy ban đầu, bắt đầu từ vị trí thứ 2; lặp lại quá trình trên cho dãy hiện hành,… đến khi dãy hiện hành chỉ còn 1 phần tử. Dãy ban đầu có N phần tử, vậy tóm tắt ý tưởng thuật toán là thực hiện N-1 lượt việc đưa phần tử nhỏ nhất trong dãy hiện hành về vị trí đúng ở đầu dãy.
* **Thuật toán:**
* /\*Thuật toán sắp xếp chọn trực tiêp\*/
* void SelectionSort(int A[], int N)
* {
* int i, j, index;
* for (i = 0; i < N - 1; i++)
* {
* index = i;
* for (j = i + 1; j < N; j++)
* {
* if (A[j] < A[index])
* {
* index = j; //Ghi nhận vị trí phần tử hiện tại nhỏ nhất
* }
* }
* Swap(A[i], A[index]);
* }

}

* **Đánh giá thuật toán:**

-Về số phép so sánh, ở lượt xét thứ i luôn có (n-i) phép so sánh để tìm min và không phụ thuộc tình trạng ban đầu của dãy nên số phép so sánh ước lượng là: (n-1) + (n-2) + ... + 1 = n(n - 1)/2.

|  |  |  |
| --- | --- | --- |
| **Trường hợp** | **Số lần So Sánh** | **Số lần gán** |
| **Tốt nhất** |  |  |
| **Xấu nhất** |  |  |

-Độ phức tạp là: **O(n²).**

### **Ưu điểm**

-Thuật toán đơn giản, dễ hiện thực.

-Có số lần hoán đổi các vị trí ít.

### **Nhược điểm**

-Chỉ được áp dụng trong các trường hợp có số lượng phần tử cần so sánh ít.

-Không nhận biết được mảng đã được sắp xếp.

### **Insertion Sort:**

* **Ý tưởng:**

-Ý tưởng chính của giải thuật sắp xếp bằng phương pháp chèn trực tiếp là tìm cách chèn phần tử **** vào vị trí thích hợp của đoạn đã được sắp để có dãy mới ****, ****,…**** trở nên có thứ tự. Vị trí này chính là vị trí giữa 2 phần tử **** và **** thỏa  (1ki)

* **Thuật toán:**
* /\*Thuật toán sắp xếp chèn trực tiêp\*/
* void InsertionSort(int A[], int N)
* {
* int pos, i, x;
* for (i = 1; i < N; i++)
* {
* x = A[i];
* pos = i - 1;
* while ((pos >= 0) && (A[pos] > x))
* {
* A[pos + 1] = A[pos];
* pos--;
* }
* A[pos + 1] = x;
* }

}

* **Đánh giá thuật toán:**

|  |  |  |
| --- | --- | --- |
| **Trường hợp** | **Số lần So Sánh** | **Số lần gán** |
| **Tốt nhất** |  |  |
| **Xấu nhất** |  |  |

-Độ phức tạp: **O()**.

* **Ưu điểm:** Chạy nhanh khi mảng nhỏ hay được sắp xếp một phần. Nếu danh sách đã gần đúng thứ tự, Insertion Sort sẽ chạy rất nhanh. Ví dụ bạn cần sắp xếp Highscore trong game.
* **Nhược điểm:** Hiệu suất thấp, không đủ nhanh với dữ liệu lớn.

### **Binary Insertion Sort:**

* **Ý tưởng:** Giống như Insertion Sort nhưng việc chèn không còn tuyến tính nữa mà là nhị phân (thời gian sắp xếp sẽ nhanh hơn).
* **Thuật toán:**
* /\*Thuật toán sắp xếp chèn nhị phân\*/
* void BinaryInsertionSort(int A[], int N)
* {
* int i, j, left, right, mid;
* int x;
* for (i = 1; i < N; i++)
* {
* left = 1; right = i - 1;
* x = A[i];
* while (left <= right)
* {
* mid = (int)(left + right) / 2;
* if (x < A[mid])
* {
* right = mid - 1;
* }
* else
* {
* left = mid + 1;
* }
* }
* for (j = i; j > left; j--)
* {
* A[j] = A[j - 1];
* }
* A[left] = x;
* }

}

* **Đánh giá thuật toán:**

|  |  |  |
| --- | --- | --- |
| **Trường hợp** | **Số lần So Sánh** | **Số lần gán** |
| **Tốt nhất** |  |  |
| **Xấu nhất** |  |  |

1. **Interchange Sort:**

* **Ý tưởng:** Ý tưởng chính của giải thuật là xuất phát từ đầu dãy, tìm tất cả nghịch thế chứa phần từ này, triệt tiêu chúng bằng cách đổi chổ phần tử này với phần tử tương ứng trong cặp nghịch thế. Lặp lại xử lý trên với các phần tử tiếp theo trong dãy.
* **Thuật toán:**
* /\*Thuật toán sắp xếp đổi chổ trực tiếp\*/
* void InterchageSort(int A[], int N)
* {
* int i, j;
* for (i = 0; i < N - 1; i++)
* {
* for (j = i + 1; j < N; j++)
* {
* if (A[j] < A[i])
* {
* Swap(A[j], A[i]);
* }
* }
* }

}

* **Đánh giá thuật toán:**

|  |  |  |
| --- | --- | --- |
| **Trường hợp** | **Số lần So Sánh** | **Số lần gán** |
| **Tốt nhất** |  | 0 |
| **Xấu nhất** |  |  |

-Độ phức tạp là: **O(n²)**

1. **Buble Sort – Thuật toán sắp xếp nổi bọt**

* **Ý tưởng:**

-Giả sử dãy ban đầu cần sắp xếp có N phần tử a1, a2, a3, …, aN. Ta sẽ xét lần lượt 2 phần tử liền kề nhau, bắt đầu từ cuối dãy số. Nếu 2 phần tử này là nghịch thế của nhau (a[i] > a[i – 1]) thì ta tiến hành hoán vị đổi chỗ hai phần tử này. Cứ như thế, ta sẽ đem được phần tử nhỏ nhất lên đầu dãy và loại phần tử này ra khỏi dãy hiện hành. Tiếp tục thực hiện lại các bước trên đối với dãy mới để đưa dần các phần tử nhỏ nhất về đầu dãy. Chúng ta sẽ thực hiện lại các thao tác trên cho đến khi dãy hiện hành chỉ còn 1 phần tử duy nhất.

* **Thuật toán:**
* /\*Thuật toán sắp xếp nổi bọt\*/
* void BubbleSort(int A[], int N)
* {
* int i, j;
* for (i = 0; i < N - 1; i++)
* {
* for (j = N - 1; j > i; j--)
* {
* if (A[j] < A[j - 1])
* {
* int temp = A[j];
* A[j] = A[j - 1];
* A[j - 1] = temp;
* }
* }
* }

}

* **Đánh giá thuật toán:**

|  |  |  |
| --- | --- | --- |
| **Trường hợp** | **Số lần So Sánh** | **Số lần gán** |
| **Tốt nhất** |  | 0 |
| **Xấu nhất** |  |  |

-Độ phức tạp: O();

* **Ưu điểm:** Code đơn giản, dễ hiểu. Không tốn thêm bộ nhớ
* **Nhược điểm:** Không đủ nhanh với dữ liệu lớn.

1. **Shaker Sort – Sắp xếp rung lắc:**

* **Ý tưởng:**

-Shaker Sort là một cải tiến của Bubble Sort. Sau khi đưa phần tử nhỏ nhất về đầu dãy, thuật toán sẽ giúp chúng ta đưa phần tử lớn nhất về cuối dãy. Do đưa các phần tử về đúng vị trí ở cả hai đầu nên Shaker Sort sẽ giúp cải thiện thời gian sắp xếp dãy số.

* **Thuật toán:**
* /\*Thuật toán sắp xếp rung lắc\*/
* void ShakerSort(int A[], int N)
* {
* int left = 1, right = N - 1, i, j, k = 0;
* do
* {
* for (i = right; i >= left; i--)
* {
* if (A[i - 1] > A[i])
* {
* int temp1 = A[i - 1];
* A[i - 1] = A[i];
* A[i] = temp1;
* k = i;
* }
* }
* left = k + 1;
* for (j = left; j <= right; j++)
* {
* if (A[j - 1] > A[j])
* {
* int temp2 = A[j - 1];
* A[j - 1] = A[j];
* A[j] = temp2;
* k = j;
* }
* }
* right = k - 1;
* } while (left <= right);
* }
* **Đánh giá thuật toán:**

- Độ phức tạp: Gần giống Bubble Sort

* Độ phức tạp cho trường hợp tốt nhất là O(n).
* Độ phức tạp cho trường hợp xấu nhất O(n2).
* Độ phức tạp trong trường hợp trung bình là O(n2).
* Thuật toán nhận diện được mảng đã sắp xếp.

- Chi phí chuyển dời luôn cao hơn (Chi phí hoán vị thường mất 3 phép gán) so sánh nên cải tiến có thể xem như không đáng kể. Thuật toán vẫn được xếp ở **O()**

1. **Shell Sort – Sắp xếp với độ dài bước giảm dần:**

* **Ý tưởng:** Giải thuật ShellSort là một phương pháp cải tiến của phương pháp chèn trực tiếp. Ý tưởng của phương pháp sắp xếp là phần chia dãy ban đầu thành những dãy con gồm các phần tử ở cách nhau **h** vị trí.
* **Thuật toán:**
* /\*Thuật toán sắp xếp với độ dài bước giảm dần\*/
* void ShellSort(int A[], int N)
* {
* int i, j, gap, temp;
* for (gap = N / 2; gap > 0; gap = gap / 2)
* {
* for (i = 0; i < N; i = i + gap)
* {
* temp = A[i];
* for (j = i; j > 0 && A[j - gap] > temp; j = j - gap)
* {
* A[j] = A[j - gap];
* }
* A[j] = temp;
* }
* }

}

* **Đánh giá thuật toán:**
* Độ phức tạp: 
* Thuật toán phức tạp. Tùy thuộc vào cách chọn khoảng cách h trong từng bước sắp xếp và số bước sắp xếp k.

1. **Heap Sort – Sắp xếp vun đống:**

* **Ý tưởng:**

-Giải thuật Heapsort còn được gọi là giải thuật vun đống, nó có thể được xem như bản cải tiến của Selection sort khi chia các phần tử thành 2 mảng con, 1 mảng các phần tử đã được sắp xếp và mảng còn lại các phần tử chưa được sắp xếp. Trong mảng chưa được sắp xếp, các phần tử lớn nhất sẽ được tách ra và đưa vào mảng đã được sắp xếp. Điều cải tiến ở Heapsort so với Selection sort ở việc sử dụng cấu trúc dữ liệu heap thay vì tìm kiếm tuyến tính (linear-time search) như Selection sort để tìm ra phần tử lớn nhất.

-Heapsort là thuật toán in-place, nghĩa là không cần thêm bất cứ cấu trúc dữ liệu phụ trợ trong quá trình chạy thuật toán. Tuy nhiên, giải thuật này không có độ ổn định (stability).

* **Thuật toán:**
* /\*Thuật toán sắp xếp cây (vung đống)\*/
* void Shift(int A[], int l, int r)
* {
* int i, j, x;
* i = l; j = 2 \* i + 1;
* x = A[i];
* while (j <= r)
* {
* if (j < r)
* {
* if (A[j] < A[j + 1]) //Tìm phần tử lớn nhất A[j] và A[j+1]
* {
* j++; //Lưu chỉ số của phần tử nhỏ nhất trong hai phần tử
* }
* }
* if (A[j] <= x)
* {
* return;
* }
* else
* {
* A[i] = A[j];
* A[j] = x;
* i = j; j = 2 \* i + 1;
* x = A[i];
* }
* }
* }
* void CreateHeap(int A[], int N)
* {
* int l;
* l = (N / 2) - 1;
* while (l >= 0)
* {
* Shift(A, l, N - 1);
* l = l - 1;
* }
* }
* void HeapSort(int A[], int N)
* {
* int r;
* CreateHeap(A, N);
* r = N - 1;
* while (r > 0)
* {
* int temp = A[0]; //A[0] là nút gốc
* A[0] = A[r];
* A[r] = temp;
* r--;
* if (r > 0)
* {
* Shift(A, 0, r);
* }
* }

}

* **Đánh giá thuật toán:**
* Độ phức tạp: Trường hợp xấu nhất là **O()**
* Ưu điểm: Chạy nhanh.
* Nhược điểm: Không ổn định.

### [**Merge sort - Thuật toán sắp xếp trộn**](http://blog.huynhbuuson.com/2013/12/thuat-toan-sap-xep-merge-sort.html)

* **Ý Tưởng:**

-Ý tưởng chúng ta sẽ chia mảng lớn thành những mảng con nhỏ hơn bằng cách chia đôi mảng lớn và chúng ta tiếp tục chia đôi các mảng con cho tới khi mảng con nhỏ nhất chỉ còn 1 phần tử. Sau đó chúng ta sẽ tiếng hành so sánh 2 mảng con có cùng mảng cơ sở (khi chúng ta chia đôi mảng lớn thành 2 mảng con thì mảng lớn đó chúng ta gọi là mảng cơ sở của 2 mảng con đó) khi so sánh chúng sẽ vừa sắp xếp vừa ghép 2 mảng con đó lại thành mảng cơ sở, chúng ta tiếp tục so sánh và ghép các mảng con lại đến khi còn lại mảng duy nhất thì đó là mảng đã được sắp xếp.

* **Thuật toán:**
* /\*Phương pháp trộn trực tiếp\*/
* int Min(int a, int b)
* {
* if (a < b)
* {
* return a;
* }
* return b;
* }
* void Merge(int A[], int temp1[], int n1, int temp2[], int n2, int k)
* {
* int i1, i2, i;
* int k1, k2;
* int j1, j2;
* i = i1 = i2 = j1 = j2 = 0;
* while (n1 > 0 && n2 > 0)
* {
* //Xác định độ dài từng dãy con 2 dãy phụ
* k1 = Min(k, n1);
* k2 = Min(k, n2);
* //Xét và trộn dãy con vào dãy
* if (temp1[i1 + j1] < temp2[i2 + j2])
* {
* A[i++] = temp1[i1 + j1];
* j1++;
* //Trộn dãy con còn lại vào dãy
* if (j1 == k1)
* {
* for (; j2 < k2; j2++)
* {
* A[i++] = temp2[i2 + j2];
* }
* i1 += k1; i2 += k2;
* j1 = j2 = 0;
* n1 -= k1; n2 -= k2;
* }
* }
* else
* {
* A[i++] = temp2[i2 + j2];
* j2++;
* //Trộn dãy con còn lại vào dãy
* if (j2 == k2)
* {
* for (; j1 < k1; j1++)
* {
* A[i++] = temp1[i1 + j1];
* }
* i1 += k1; i2 += k2;
* j1 = j2 = 0;
* n1 -= k1; n2 -= k2;
* }
* }
* }
* }
* void MergeSort(int A[], int N)
* {
* int n1, n2;
* int i, k, ik;
* int \*temp1 = new int[N];
* int \*temp2 = new int[N];
* k = 1;
* do
* {
* i = n1 = n2 = 0;
* //Chia mảng ra 2 mảng phụ
* while (i < N)
* {
* ik = 0;
* while (ik < k&&i < N)
* {
* temp1[n1++] = A[i++];
* ik++;
* }
* ik = 0;
* while (ik < k&&i < N)
* {
* temp2[n2++] = A[i++];
* ik++;
* }
* }
* Merge(A, temp1, n1, temp2, n2, k);
* //Tăng độ lớn tối đa dãy con
* k \*= 2;
* } while (k < N);
* delete[] temp1;
* delete[] temp2;

}

* **Đánh giá thuật toán:**

-Độ phức tạp: **O()**.

* **Ưu điểm:** Hiệu suất của merge sort rất cao. Ổn định. Sắp sếp nhanh hơn so với các thuật toán cơ bản (Insertion Sort, Selection Sort, Interchage Sort), nhanh hơn Quick Sort trong một số trường hợp.
* **Nhược điểm:** Code thuật toán này khá phức tạp, khó cài đặt, không nhận dạng được mảng đã được sắp.

### [**Quick sort - Thuật toán sắp xếp nhanh**](http://blog.huynhbuuson.com/2013/12/thuat-toan-sap-xep-quick-sort-sap-xep.html)

* **Ý tưởng:**

**-**Giải thuật QuickSort dựa trên việc phân hoạch dãy ban đầu thành 2 phần:

**+** Dãy con 1: Gồm các phần tử có giá trị không lớn hơn **x**

**+** Dãy cpn 2: Gồm các phần tử có giá trị không nhỏ hơn **x**

Với **x** là giá trị của một phần tử tùy ý trong dãy dữ liệu ban đầu. Sau khi thực hiện phân hoạch, dãy ban đầu được phân thành 3 phần:

* ak < x, với k=1,…i
* ak = x, với k=i…j
* ak > x, với k=j…N

Trong đó dãy con thứ 2 đã có thứ tự, nếu dãy con 1 và 3 có 1 phần tử thì chúng cũng đã có thứ tự, khi đó dãy ban đầu đã được sắp. Ngược lại, nếu các dãy con 1 và 3 có nhiều hơn 1 phần tử thì dãy ban đầu chỉ có thứ tự khi các dãy con 1 và 3 được sắp. Để sắp xếp dãy con 1 và 3, ta lần lượt tiến hành việc phân hoạch tương ứng dãy con thep cùng phương pháp phân hoạch dãy ban đầu vừa trình bày…

* **Thuật toán:**
* /\*Thuật toán sắp xếp dựa trên phân hoạch\*/
* void QuickSort(int A[], int N, int l, int r)
* {
* int i, j;
* int x;
* x = A[(l + r) / 2]; //Chọn phần tử giữa làm giá trị mốc
* i = l; j = r;
* do
* {
* while (A[i] < x)
* {
* i++;
* }
* while (A[j] > x)
* {
* j--;
* }
* if (i <= j)
* {
* int temp = A[i];
* A[i] = A[j];
* A[j] = temp;
* i++;
* j--;
* }
* } while (i < j);
* if (l < j)
* {
* QuickSort(A, N, l, j);
* }
* if (i < r)
* {
* QuickSort(A, N, i, r);
* }

}

* **Đánh giá thuật toán:**

-Độ phức tạp: Tùy thuộc vào Trường hợp

|  |  |
| --- | --- |
| **Trường hợp** | **Độ phức tạp** |
| **Tốt nhất** |  |
| **Trung bình** |  |
| **Xấu nhất** |  |

* **Ưu điểm:** Tuỳ cách chọn pivot mà tốc độ của thuật toán nhanh hay chậm. Chạy nhanh (nhanh nhất trong các thuật toán sắp xếp dựa trên việc só sánh các phần tử).
* **Nhược điểm:** Code khá phức tạp.
* Tùy thuộc vào cách chia thành 2 phần, nếu chia không tốt, độ phức tạp trong trường hợp xấu nhất có thể là O(N2)O(N2). Nếu ta chọn pivot ngẫu nhiên, thuật toán chạy với độ phức tạp trung bình là O(N∗logN)O(N∗logN) (trong trường hợp xấu nhất vẫn là O(N2)O(N2), nhưng ta sẽ không bao giờ gặp phải trường hợp đó).
* Không ổn định.

1. **Counting Sort:**

* **Ý tưởng:**

-Giả sử tôi cho mảng cần sắp xếp các giá trị của mảng đều là số nguyên và thuộc khoảng [0..M].

-Ý tưởng của thuật toán là đếm trong khoảng [0..M] có bao nhiêu giá trị 0 (giả sử có C(0) giá trị), bao nhiêu giá trị 1 (giả sử có C(1) giá trị),… có bao nhiêu giá trị M (giả sử có C(M) giá trị). Sau đó tôi sắp xếp lại mảng bằng cách đặt C(0) phần tử 0 ở đầu, tiếp theo đặt C(1) phần tử 1 tiếp theo,… và đặt C(M) phần tử M cuối cùng.

-Độ phức tạp của thuật toán này là O(max(N,M)). Giá trị của M càng nhỏ thì thuật toán sắp sếp càng nhanh.

* **Thuật toán:**
* /\*Thuật toán sắp xếp bằng phương pháp đếm phân phối\*/
* void CountingSort(int A[], int N)
* {
* int \*count = new int[1000000];
* for (int i = 0; i < N; i++)
* {
* count[i] = 0;
* }
* if (count == NULL)
* {
* return;
* }
* for (int i = 0; i < N; i++)
* {
* ++count[A[i]];
* }
* int outPutIndex = 0;
* for (int j = 0; j < 1000000; j++)
* {
* while (count[j]-- > 0)
* {
* A[outPutIndex++] = j;
* }
* }
* delete[] count;

}

* **Đánh giá thuật toán:**

-Độ phức tạp: **O().**

* **Nhược điểm:**

– Chỉ sắp xếp được cho số nguyên.

– Phải biết miền giá trị của số nguyên.

– Miền giá trị lớn quá 10^7 sẽ không thể tạo mảng dd để lưu trữ.

* **Ưu điểm:**

– Cài đặt đơn giản.

– Độ phức tạp thuật toán phụ thuộc vào miền giá trị.

1. **Radix Sort – Sắp xếp cơ số:**

* **Ý tưởng:**

-Giả sử mỗi phần tử ai trong dãy a0, a1, …, an-1 là một số nguyên có tối đa m chữ số. Phân loại các phần tử này lần lượt theo các chữ số hàng đơn vị, hàng chục, hàng trăm, …, tương tự việc phân loại thư theo tỉnh thành, quận huyện, phường xã, …

* **Thuật toán:**
* /\*Thuật toán sắp xếp theo cơ số\*/
* void RadixSort(int A[], int N)
* {
* int i;
* int m = A[0], Exp = 1;
* int \*B = new int[N];
* if (N == NULL)
* {
* printf("Khong du bo nho.!!\n");
* return;
* }
* for (i = 0; i < N; i++)
* {
* if (A[i] > m)
* {
* m = A[i];
* }
* }
* while (m / Exp > 0)
* {
* int Bucket[10] = { 0 };
* for (i = 0; i < N; i++)
* {
* Bucket[A[i] / Exp % 10]++;
* }
* for (i = 1; i < 10; i++)
* {
* Bucket[i] += Bucket[i - 1];
* }
* for (i = N - 1; i >= 0; i--)
* {
* B[--Bucket[A[i] / Exp % 10]] = A[i];
* }
* for (i = 0; i < N; i++)
* {
* A[i] = B[i];
* }
* Exp \*= 10;
* }

}

* **Đánh giá thuật toán:**

-Độ phức tạp: **O(n)**

* **Ưu điểm:** Có thể chạy nhanh hơn các thuật toán sắp xếp sử dụng so sánh. Ví dụ nếu ta sắp xếp các số nguyên (int) 32 bit, và chia nhóm theo 1 bit, thì độ phức tạp là O(n). Trong trường hợp tổng quát, độ phức tạp là 
* **Nhược điểm:** Không thể sắp xếp số thực.

1. **Flash Sort:**

* **Ý tưởng:**

-Flash sort là một thuật toán sắp xếp tại chỗ (in-situ, không dùng mảng phụ) không đệ qui, gồm có 3 bước: (1) Phân lớp dữ liệu, tức là dựa trên giả thiết dữ liệu tuân theo 1 phân bố nào đó, chẳng hạn phân bố đều, để tìm 1 công thức ước tính vị trí (lớp) của phần tử sau khi sắp xếp. (2) Hoán vị toàn cục, tức là dời chuyển các phần tử trong mảng về lớp của mình. (3) Sắp xếp cục bộ, tức là để sắp xếp lại các phần tử trong phạm vi của từng lớp.

* **Thuật toán:**
* int min(int a, int b)
* {
* return a > b ? b : a;
* }
* int max(int a, int b)
* {
* return a > b ? a : b;
* }
* /\*Thuật toán sắp xếp tại chỗ (in-situ, không dùng mảng phụ)\*/
* void FlashSort(int A[], int N)
* {
* int ng = (int)(0.45 \* N);
* int \*sg = new int[ng + 1];
* int \*a = new int[N];
* // 1. Tìm giá trị max và min
* int minValue = INT\_MAX;
* int maxValue = INT\_MIN;
* for (int i = 0; i < N; i++)
* {
* minValue = min(minValue, A[i]);
* maxValue = max(maxValue, A[i]);
* }
* double c = ((double)ng - 1) / (maxValue - minValue);
* // 2. Tính toán kích thước cho mỗi nhóm flashsort
* for (int i = 0; i < ng; i++)
* {
* sg[i] = 0;
* }
* for (int i = 0; i < N; i++)
* {
* int gi = (int)(c \* (A[i] - minValue));
* ++sg[gi];
* }
* // 3. Trong mỗi nhóm, tính toán chỉ số mảng cuối
* for (int i = 0; i < ng; ++i)
* {
* sg[i + 1] += sg[i];
* }
* // 4. Hoán vị
* for (int i = 0; i < N; ++i)
* {
* int gi = (int)(c \* (A[i] - minValue));
* int bi = sg[gi] - 1;
* a[bi] = A[i];
* --sg[gi];
* }
* // 5. Sắp xếp danh sách phần tử trong mỗi nhóm, sử dụng thuật toán Insertion sort
* for (int gi = 0; gi < ng; ++gi)
* {
* int x = sg[gi];
* int y = sg[gi + 1];
* // Insertion sort
* for (int i = x; i < y; ++i) {
* int t = a[i];
* int j = i - 1;
* for (; j >= 0 && a[j] >= t; --j) {
* a[j + 1] = a[j];
* }
* a[j + 1] = t;
* }
* }
* // 6. Cập nhật mảng A
* for (int i = 0; i < N; ++i) {
* A[i] = a[i];
* }
* delete[] a;
* delete[] sg;

}

* **Đánh giá thuật toán:**

-Độ phức tạp: **O()**

1. **Trình bày về kết quả thí nghiệm và cho nhận xét**

* **Kết quả thí nghiệm:**

1. **Bộ dữ liệu Sorted:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Size** | **SelectionSort** | **InsertionSort** | **Binary InsertionSort** | **InterchangeSort** | **BubbleSort** | **ShakerSort** | **ShellSort** | **HeapSort** | **MergeSort** | **QuickSort** | **CountingSort** | **RadixSort** | **FlashSort** |
| **1000** | 0.002 | 0 | 0 | 0.002 | 0.002 | 0 | 0 | 0 | 0 | 0 | 0.005 | 0 | 0 |
| **5000** | 0.033 | 0 | 0 | 0.033 | 0.037 | 0 | 0 | 0.001 | 0.001 | 0 | 0.007 | 0.001 | 0.001 |
| **10000** | 0.133 | 0 | 0 | 0.213 | 0.15 | 0 | 0 | 0.001 | 0.002 | 0.001 | 0.005 | 0.003 | 0 |
| **50000** | 3.32 | 0 | 0.002 | 3.342 | 4.013 | 0 | 0 | 0.009 | 0.016 | 0.003 | 0.006 | 0.007 | 0.003 |
| **100000** | 12.653 | 0 | 0.006 | 12.687 | 14.718 | 0.001 | 0.001 | 0.018 | 0.048 | 0.007 | 0.009 | 0.012 | 0.011 |
| **250000** | 81.713 | 0.001 | 0.02 | 84.706 | 96.043 | 0.002 | 0.002 | 0.06 | 0.107 | 0.025 | 0.009 | 0.036 | 0.02 |

1. **Bộ dữ liệu Reversed:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Size** | **SelectionSort** | **InsertionSort** | **Binary InsertionSort** | **InterchangeSort** | **BubbleSort** | **ShakerSort** | **ShellSort** | **HeapSort** | **MergeSort** | **QuickSort** | **CountingSort** | **RadixSort** | **FlashSort** |
| **1000** | 0.001 | 0.001 | 0.001 | 0.012 | 0.002 | 0.002 | 0.001 | 0 | 0 | 0 | 0.005 | 0 | 0 |
| **5000** | 0.036 | 0.035 | 0.023 | 0.48 | 0.05 | 0.065 | 0.03 | 0 | 0.001 | 0.001 | 0.006 | 0 | 0 |
| **10000** | 0.154 | 0.14 | 0.095 | 1.314 | 0.202 | 0.232 | 0.089 | 0.001 | 0.003 | 0 | 0.013 | 0.001 | 0.001 |
| **50000** | 6.163 | 3.644 | 2.486 | 23.752 | 5.084 | 5.766 | 2.829 | 0.009 | 0.016 | 0.003 | 0.006 | 0.006 | 0.004 |
| **100000** | 21.267 | 14.344 | 9.766 | 66.293 | 20.886 | 23.295 | 11.345 | 0.019 | 0.033 | 0.007 | 0.007 | 0.012 | 0.008 |
| **250000** | 102.144 | 93.868 | 65.297 | 233.602 | 133.716 | 151.308 | 63.395 | 0.049 | 0.094 | 0.019 | 0.007 | 0.032 | 0.021 |

1. **Bộ dữ liệu Random:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Size** | **SelectionSort** | **InsertionSort** | **Binary InsertionSort** | **InterchangeSort** | **BubbleSort** | **ShakerSort** | **ShellSort** | **HeapSort** | **MergeSort** | **QuickSort** | **CountingSort** | **RadixSort** | **FlashSort** |
| **1000** | 0.001 | 0.001 | 0.001 | 0.008 | 0.002 | 0.002 | 0.001 | 0.001 | 0 | 0 | 0.005 | 0 | 0.006 |
| **5000** | 0.036 | 0.018 | 0.012 | 0.194 | 0.06 | 0.042 | 0.013 | 0.001 | 0.003 | 0 | 0.005 | 0.001 | 0 |
| **10000** | 0.137 | 0.071 | 0.048 | 0.809 | 0.271 | 0.195 | 0.055 | 0.002 | 0.005 | 0.001 | 0.006 | 0.003 | 0 |
| **50000** | 3.284 | 1.812 | 1.361 | 16.625 | 8.06 | 5.567 | 1.584 | 0.012 | 0.029 | 0.008 | 0.006 | 0.006 | 0.003 |
| **100000** | 13.177 | 7.176 | 4.864 | 51.443 | 32.115 | 22.272 | 6.151 | 0.025 | 0.068 | 0.016 | 0.007 | 0.016 | 0.008 |
| **250000** | 86.688 | 48.818 | 31.362 | 208.956 | 203.177 | 140.672 | 36.898 | 0.072 | 0.169 | 0.042 | 0.008 | 0.031 | 0.021 |

1. **Bộ dữ liệu Nearly Sorted:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Size** | **SelectionSort** | **InsertionSort** | **Binary InsertionSort** | **InterchangeSort** | **BubbleSort** | **ShakerSort** | **ShellSort** | **HeapSort** | **MergeSort** | **QuickSort** | **CountingSort** | **RadixSort** | **FlashSort** |
| **1000** | 0.001 | 0 | 0 | 0.002 | 0.002 | 0 | 0 | 0 | 0 | 0 | 0.007 | 0.001 | 0 |
| **5000** | 0.04 | 0 | 0.001 | 0.033 | 0.039 | 0 | 0 | 0.001 | 0.001 | 0 | 0.005 | 0 | 0 |
| **10000** | 0.138 | 0 | 0 | 0.136 | 0.154 | 0 | 0.001 | 0.001 | 0.006 | 0 | 0.005 | 0.001 | 0.001 |
| **50000** | 3.396 | 0.001 | 0.003 | 3.598 | 4.178 | 0 | 0 | 0.009 | 0.018 | 0.003 | 0.008 | 0.007 | 0.004 |
| **100000** | 14.545 | 0.001 | 0.01 | 13.794 | 14.817 | 0.001 | 0.002 | 0.019 | 0.037 | 0.007 | 0.006 | 0.012 | 0.007 |
| **250000** | 78.157 | 0.002 | 0.016 | 79.503 | 92.643 | 0.002 | 0.002 | 0.053 | 0.12 | 0.021 | 0.008 | 0.037 | 0.022 |

* **Nhận xét:**
* Phần lớn các thuật toán sắp xếp đơn giản dựa trên sự so sánh các phần tử có trong mảng, danh sách,….điều đó lại tốn chí phí, độ phức tạp lớn, thời gian thực thi lâu, số phép so sánh , số phép gán lớn, nhưng lại cài đặt rất dễ dàng, dễ hiểu tiêu biểu là các thuật toán: Interchange Sort, InsertionSort, Bubble Sort,…Các thuật toán này đều có chi phí thực hiện là O()
* Một số thuật toán sắp xếp cải tiến hơn, chẳng hạn như: BinaryInsertion Sort (cải tiến của Insertion Sort), Shaker Sort(cải tiến của Bubble Sort),...mặc dù có chi phí thấp hơn, thời gian thực thi có thể nhanh hơn một chút, nhưng vẫn nằm trong nhóm thuật toán có độ phức tạp là **O().** Một số thuật toán khác như: thuật toán Shell Sort (cải tiến của Insertion Sort), Heap Sort (cải tiến của Selection Sort) thì có độ phức tạp nhỏ hơn hẳn.
* Ngoài ra, phương pháp chia để trị cũng có thể áp dụng cho thuật toán sắp xếp chẳng hạn như: Merge Sort và Quick Sort. Mặc dù cài đặt chúng có vẻ phức tạp hơn nhiều so với các thuật toán trên nhưng chi phí lại rất thấp, thời gian thực thi rất nhanh. Cả 2 thuật toán đều có độ phức tạp **O()**

Thuật toán Merge Sort thì phải dùng thêm bộ nhớ đệm để phân hoạch còn thuật toán Quick Sort giống như tên gọi của nó là nhanh nhất trong các thuật toán sắp xếp về so sánh giá trị 2 phần tử, tuy nhiên chi phí trong trường hợp xấu nhất là: **O().**

* Riêng thuật toán Radix Sort thì được phát triển dựa trên sự mô phỏng qui trình phát thư, thuật toán này có độ phức tạp tuyến tính.
* Do đó, nhìn vào bảng số liệu và đồ thị, ta thấy rằng kích thước số phần tử càng nhỏ thì thời gian chênh lệch giữa các thuật toán rất nhỏ, không đáng kể, nhưng kích thước số lượng phần tử càng lớn thì thời gian thực thi giữa các thuật toán chênh lệch rõ rệt, rất lớn.
* Ngoài ra, còn một yếu tố nữa để đánh giá thời gian thực thi giữa các thuật toán là bộ dữ liệu ban đầu sắp xếp.
* Một vấn để quan trọng nữa của các thuật toán sắp xếp là cấu trúc dữ liệu lưu trữ các phần tử, do phần những thuật toán dễ cài đặt, dễ hiểu chẳng hạn như: Interchange Sort, Insertion Sort,… thì ta dùng mảng một chiều để lưu trữ, còn đối với những thuật toán phức tạp hơn, khó cài đặt, khó hiểu hơn chảng hạn như: Merge Sort, Quick Sort, Radix Sort thì ta nên dùng danh sách liên kết để lưu trữ các phần tử và sắp xếp thì như thế thuật toán sẽ trở nên hiệu quả hơn.
* *“Không có 1 bất kỳ thuật toán sắp xếp nào cụ thể cả, nó còn phụ thuộc vào rất nhiều yếu tố”.* Do đó, có rất nhiều thuật toán sắp xếp khác nhau ra đời. Ví dụ dưới đây, cho thấy những yếu tố nào sẽ ảnh hưởng đến việc lựa chọn thuật toán phù hợp:
* Quick Sort sẽ là tốt nhất nếu ...

1. Không lo lắng về các case đầu vào kể cả trường hợp xấu nhất (trật tự nói chung là bộ dữ liệu Random (ngẫu nhiên))
2. Không quan tâm đến dung lượng bộ nhớ, bộ nhớ là hoàn toàn lý tưởng và phù hợp ở đây

* Nếu dữ liệu đã được sắp xếp sẵn, thì nên chọn Insertion Sort hoặc Shell Sort sẽ tốt hơn.
* Nếu chúng ta thực sự phải loại bỏ case xấu nhất, có thể sử dụng Heap (hoặc ít nhất là Quick3) với độ phức tạp NlogN
* Tim Sort sẽ có độ phức tạp thấp hơn Quick Sort ở cả Best Case lẫn Worse Case, Tim Sort là sự kết hợp của Merge Sort và Insertion Sort. Python sử dụng thuật toán sắp xếp này là mặc định của họ
* Trong trường hợp, dữ liệu rất ít phần tử (10-20 phần tử), lựa chọn Selection Sort sẽ nhanh hơn Quick Sort.
* *Do có nhiều tiêu chí ưu – nhược điểm, cũng như mỗi thuật toán phụ thuộc vào nhiều yếu tố như: dữ liệu đầu vào như thế nào, có sắp xếp sẵn hay không, dung lượng bộ nhớ, tốc độ xử lý CPU,… nên ta không thể so sánh các* *thuật toán 1 cách chính xác.* ***Nhưng nhìn chung thuật toán Quick Sort tối ưu nhất trong tất cả các thuật toán trong phần lớn các trường hợp.!!***
* *Dựa trên nhiều bài toán cũng như dự án mà cần 1 thuật toán sắp xếp phù hợp nhất.*
* *Với bộ dữ liêu nhỏ, ta có thể dùng thuật toán nào cũng được, tuy nhiêvới bộ dữ liệu rất lớn ta phải dựa trên nhiều tiêu chí để có thể chọn thuật toán sắp xếp thích hợp nhất.*