KHOA CÔNG NGHỆ THÔNG TIN

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

CÁC THUẬT TOÁN SẮP XẾP

Lê Thanh Trường Vinh - 1712910

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

1

THUẬT TOÁN

1.1 BUBBLE SORT

1.1.1 Ý Tưởng Thuật Toán:

Sắp xếp nổi bọt (bubble sort) là một thuật toán với các thao tác cơ bản là so sánh hai phần tử kề nhau, nếu chưa đúng thứ tự thì đổi chỗ.

1.1.2 Các Bước Thực Hiện:

```
void bubble(int*a, int n)
{
    for (int i = 0; i < n-1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (a[j]<a[i])
            {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

1.1.3 Tính Hiệu Quả Thuật Toán:

Với mỗi i = 1,2,….,n-1 sẽ có i lần phép so sánh. Do đó nhiều lần so sánh nhất là

$$(n-1)+(n-2)+\ldots+2+1=\frac{(n-1)n}{2}$$

Vậy độ phức tạp thuật toán là $O(n^2)$

1.2 SELECTION SORT

1.2.1 Ý Tưởng Thuật Toán:

Sắp xếp chọn là một thuật toán sắp xếp đơn giản, dựa trên việc so sánh đổi chỗ.
Chọn phần tử nhỏ nhất trong n phần tử ban đầu, sau đó đưa phần tử này về đầu của dãy hiện tại, sau đó coi như phần tử đó không còn xuất hiện và mảng hiện tại chỉ còn n-1 phần tử.
Tiếp tục thực hiện cho đến khi chỉ còn một phần tử (đã sắp xếp xong).

1.2.2 Các Bước Thực Hiện:

```
void selection(int*a, int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
        {
            if (a[minIndex]>a[j])
            {
                minIndex = j;
            }
            if (minIndex != i)
            {
                int temp = a[minIndex];
                a[minIndex] = a[i];
                a[i] = temp;
            }
        }
    }
}
```

1.2.3 Tính Hiệu Quả Thuật Toán:

Tương tự như Bubble Sort, Selection Sort có độ phức tạp thời gian là $O(n^2)$

1.3 INSERTION SORT

1.3.1 Ý Tưởng Thuật Toán:

Sắp xếp chèn là một thuật toán dựa vào việc so sánh đổi chỗ. Tương tự việc sắp xếp một bộ bài Tây theo đúng thứ tự. Bắt đầu từ vị trí thứ 2 của mảng và bắt đầu đặt vào đúng vị trí thích hợp.

1.3.2 Các Bước Thực Hiện:

```
void insertion(int*a, int n)
{
    int key;
    for (int i = 1; i < n; i++)
    {
        key = a[i];
        int j = i - 1;
        while (j >= 0 && a[j]>key)
        {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
}
```

1.3.3 Tính Hiệu Quả Thuật Toán:

Độ phức tạp thuật toán là $O(n^2)$

1.4 HEAP SORT

1.4.1 Ý Tưởng Thuật Toán:

Sắp xếp vun đống dựa trên cấu trúc dữ liệu được gọi là đống nhị phân.
Đống nhị phân: mỗi mảng được xem là một cây nhị phân gần đầy đủ, với gốc ở phần tử thứ nhất, con là phần tử thứ 2*i + 1 và 2*i + 2.

1.4.2 Các Bước Thực Hiện:

Bước 1: Xây Dựng Đống.
Bước 2: Lan Truyền.
Bước 3: Sắp Xếp.

Xây dựng đống với gốc là phần tử thứ i và phần tử 2*i+1 và 2*i+2 là lá. Nếu chưa đúng theo nguyên tắc cây là gốc luôn lớn hơn là thì bắt đầu lan truyền bằng việc thay đổi vị trí của gốc và lá sao cho đúng nguyên tắc. Sau khi đã Xây Dựng xong Đống và đã Lan Truyền xong thì lấy phần tử ở Gốc đầu tiên (phần tử đầu tiên hay phần tử lớn nhất) đổi chỗ với phần tử cuối cùng ở trong mảng. Tiếp tục Lan Truyền cho đến khi cây đã sắp xếp xong.

```
void heapSort(int*a, int n)
{
    for (int i = n / 2; i >= 0; i--)
        heapify(a, n, i);
    for (int i = n - 1; i >= 0; i--)
    {
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;
        heapify(a, i,0);
    }
}
void heapify(int*a, int n, int i)
{
    int root = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && a[left]>a[root])
    {
        root = left;
    }
    if (right < n&&a[right] > a[root])
    {
        root = right;
    }
    if (root != i)
    {
        int temp = a[root];
        a[root] = a[i];
        a[i] = temp;
        heapify(a, n, root);
    }
}
```

1.4.3 Tính Hiệu Quả Thuật Toán:

Độ phức tạp thời gian khi Xây Dựng Đống là O(n)
Độ phức tạp thời gian khi Lan Truyền là O(nlogn)
Độ phức tạp thời gian của cả quá trình là O(nlogn)

1.5 QUICK SORT

1.5.1 Ý Tưởng Thuật Toán:

Quick Sort là một thuật toán sắp xếp dựa trên chiến thuật Divide and Conquer của Đệ Quy. Quick Sort chia danh sách thành 2 danh sách có có kích thước tương đối bằng nhau bởi một phần tử gọi là mốc (pivot). Những phần tử lớn hơn mốc được đưa ra một danh sách và ngược lại. Cứ tiếp tục như vậy ở các danh sách con cho đến khi kích thước của danh sách con là 1 bằng phương pháp đổi chỗ.

1.5.2 Các Bước Thực Hiện:

Đoạn chương trình dưới đây thực hiện theo hình thức khử đệ quy và phần tử mốc là phần tử ở giữa danh sách.

```
void quick(int*a, int left, int right)
{
    int mid = a[(left + right) / 2];
    int i = left;
    int j = right;
    do
    {
        while (a[i]<mid)
        {
            i++;
        }
        while (a[j]>mid)
        {
            j--;
        }
        if (i <= j)
        {
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++;
            j--;
        }
    } while (i <= j);
    if (left < j) quick(a, left, j);
    if (i < right) quick(a, i, right);
}
```

1.5.3 Tính Hiệu Quả Thuật Toán:

Độ phức tạp thời gian của thuật toán:
- Trường hợp tốt nhất là O(nlogn)
- Trường hợp xấu nhất là $O(n^2)$

1.6 MERGE SORT

1.6.1 Ý Tưởng Thuật Toán:

Tương tự như Quick Sort dùng Đệ Quy có chiến lược Divide and Conquer.
Tuy nhiên Merge Sort sẽ chia thành 2 danh sách bằng nhau sau đó nối 2 danh sách đó lại theo thứ tự. Tiếp tục thực hiện cho đến khi danh sách chỉ còn 1 phần tử.

1.6.2 Các Bước Thực Hiện:

Bước 1: Cắt danh sách tại điểm giữa.
Bước 2: Nối 2 danh sách theo thứ tự.

```
void merge(int*a ,int*L, int*R, int left, int right)
{
    int  mid = (left + right) / 2;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    for (int i = 0; i < n1; i++)
    {
        L[i] = a[left + i];
    }
    for (int j = 0; j < n2; j++)
    {
        R[j] = a[mid + 1 + j];
    }
    int i, j, k;
    i = j = 0;
    k = left;
    while (i < n1&&j < n2)
    {
        if (L[i] < R[j])
        {
            a[k] = L[i];
            i++;
        }
        else
        {
            a[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        a[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        a[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(int*a, int*L, int*R, int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2;
        mergeSort(a, L, R, left, mid);
        mergeSort(a, L, R, mid + 1, right);
        merge(a, L, R, left, right);
    }
}
```

1.6.3 Tính Hiệu Quả Thuật Toán:

Độ phức tạp thời gian của Merge Sort là O(nlogn)

1.7 RADIX SORT

1.7.1 Ý Tưởng Thuật Toán:

Radix Sort là một thuật toán sắp xếp không cần so sánh. Thuật toán này dựa trên ý tưởng nếu một dãy số đã được sắp xếp thì từng chữ số của nó cũng được sắp xếp dựa trên giá trị của chữ số đó. Thuật toán này yêu cầu dãy cần được sắp xếp có thể so sánh thứ tự các vị trí vì thế Radix Sort không giới hạn ở tập số nguyên.

1.7.2 Các Bước Thực Hiện:

Áp dụng kỹ thuật của Counting Sort.

Tạo mảng gồm 10 phần tử là các số tự nhiên từ 0 đến 9. (bucket)

Lấy phần tử hàng đơn vị của mỗi số trong danh sách bỏ vào bucket tương ứng. Sau khi hết danh sách thì lấy ra theo thứ tự. Tiếp tục như vậy ở hàng chục, trăm,…. Cho đến số lớn nhất.

```
int getMax(int*a, int n)
{
    int max = a[0];
    for (int i = 1; i < n; i++)
    {
        if (a[i]>max)
        {
            max = a[i];
        }
    }
    return max;
}
void radix(int*a, int n)
{
    int max = getMax(a, n);
    for (int exp = 1; max / exp>0; exp *= 10)
    {
        countSort(a, n, exp);
    }
}
void countSort(int*a, int n, int exp)
{
    int* output = (int*)malloc(n*sizeof(int));
    int count[10] = { 0 };
    for (int i = 0; i < n; i++)
        count[(a[i] / exp) % 10]++;
    for (int i = 1; i < 10; i++)
    {
        count[i] += count[i - 1];
    }
    for (int i = n - 1; i >= 0; i--)
    {
        output[count[(a[i] / exp) % 10] - 1] = a[i];
        count[(a[i] / exp) % 10]--;
    }
    for (int i = 0; i < n; i++)
    {
        a[i] = output[i];
    }
}
```

1.7.3 Tính Hiệu Quả Thuật Toán:

Độ phức tạp thời gian là O(n*k)

1.8 FLASH SORT

1.8.1 Ý Tưởng Thuật Toán:

Chia danh sách thành các vùng nhỏ hơn và sắp xếp trên các vùng đó.

1.8.2 Các Bước Thực Hiện:

Bước 1: Chia Danh sách thành các phân vùng (khuyến nghị m = 0.43*n).
Bước 2: Dịch chuyển các phần tử về đúng các phân vùng với công thức tìm phân vùng k = (m-1)*(a[i]-a[min])/(a[max] – a[min]).
Bước 3: Dùng các thuật toán sắp xếp có hiệu quả cao đối với dữ liệu nhỏ để sắp xếp các vùng (khuyến nghị insertion sort).

```
void flash(int*a, int n)
{
    int max, min;
    max = min = a[0];
    for (int i = 1; i < n; i++)
    {
        if (a[i]>max)
            max = a[i];
        if (a[i] < min)
            min = a[i];
    }
    int m = 0.43*n;
    int *count = new int[m + 10];
    for (int i = 0; i < m; i++)
        count[i] = 0;
    int k;
    for (int i = 0; i < n; i++)
    {
        k = ((m - 1)*(a[i] - min)) / (max - min);
        count[k]++;
    }
    for (int i = 1; i < m; i++)
        count[i] += count[i - 1];
    int i = 0;
    int x, temp;
while (i < n)
{
    k = (m - 1)*(a[i] - min) / (max - min);
    x = a[i];
    while (i < count[k])
    {
        temp = a[count[k] -  1];
        a[count[k] - 1] = x;
        count[k]--;
        x = temp;
        k = (m - 1)*(x - min) / (max - min);
    }
    i++;
}
for (int i = 0; i < m - 1; i++)
{
    int key, k;
    for (int j = count[i]; j < count[i + 1]; j++)
    {
        key = a[j];
        k = j - 1;
        while (k >= count[i] && a[k]>key)
        {
            a[k+1] = a[k];
            k--;
        }
        a[k + 1] = key;
    }
}
```

1.8.3 Tính Hiệu Quả Thuật Toán:

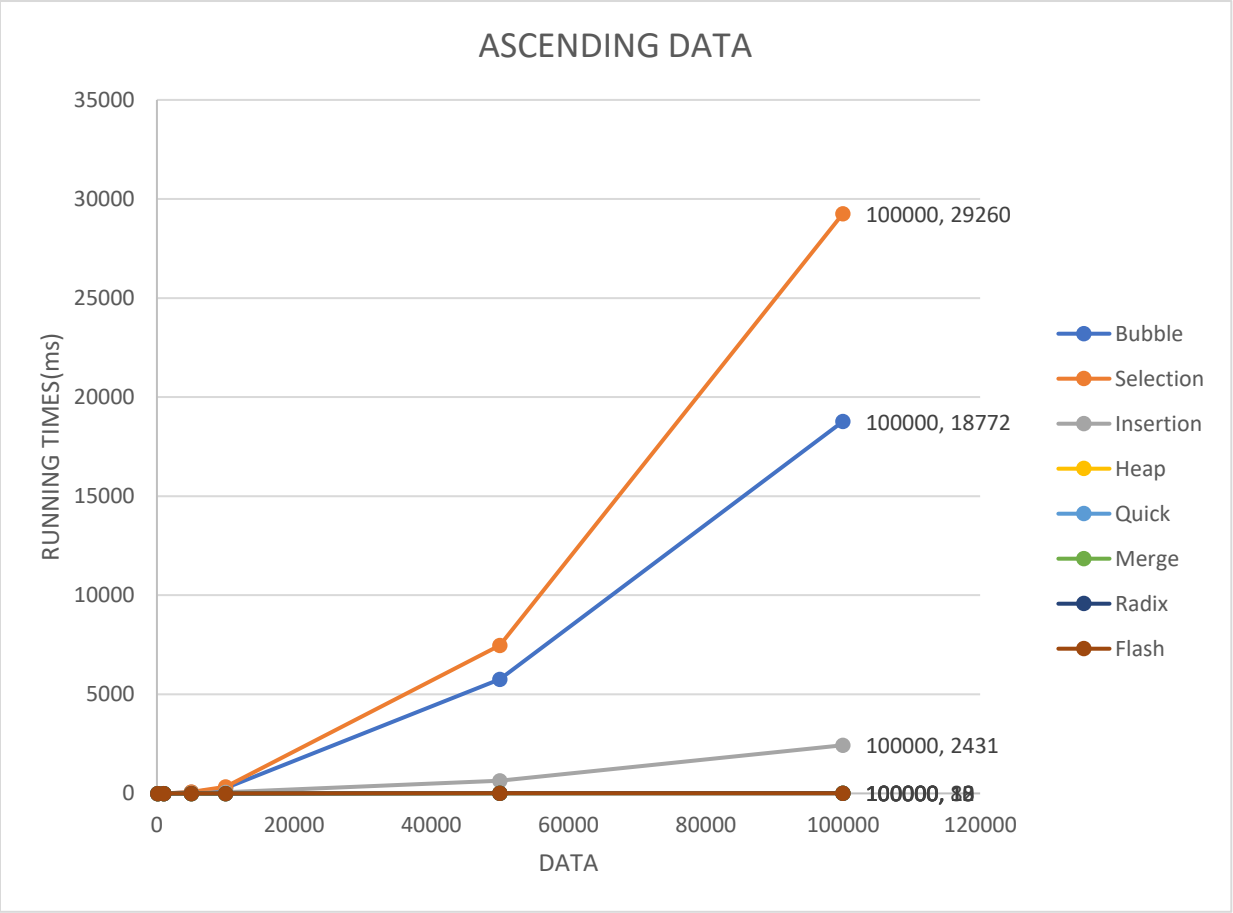
Trường Hợp Tốt Nhất Xảy ra là O(n)

2

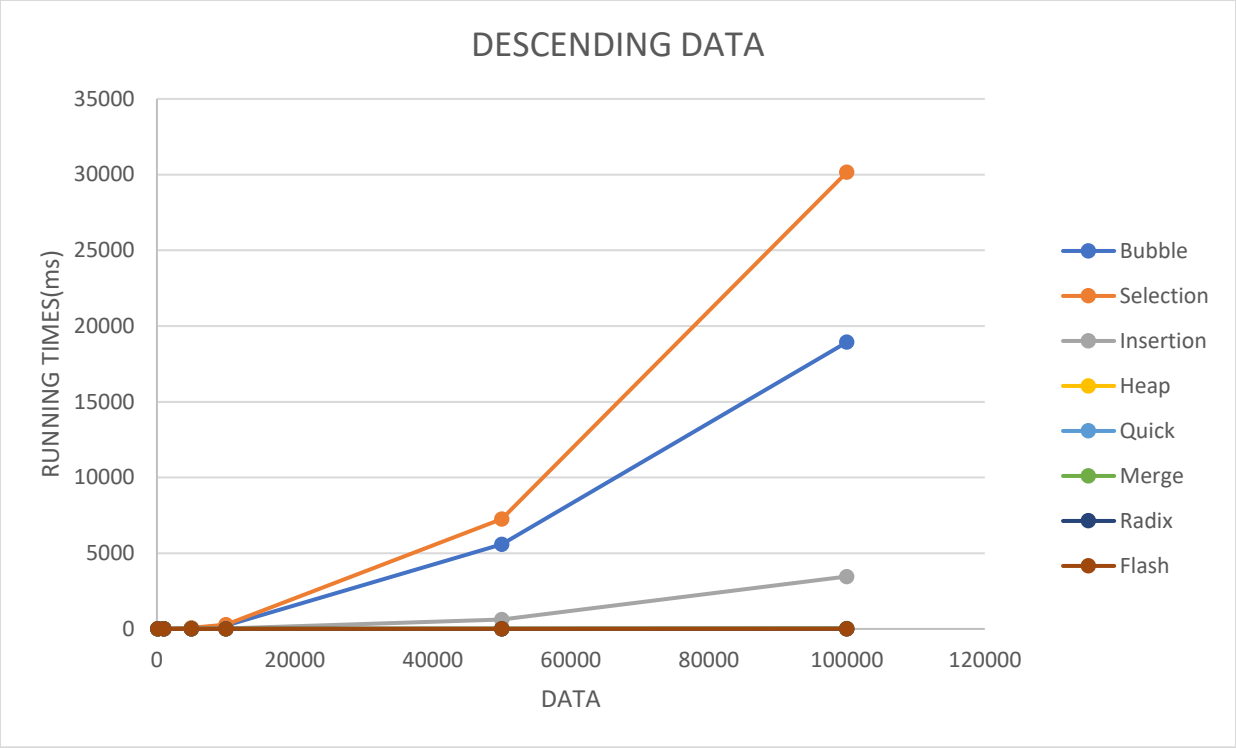
KẾT QUẢ THỰC NGHIỆM

2.1 Trường Hợp Dữ Liệu Ngẫu Nhiên:

2.2 Trường Hợp Sắp Xếp Tăng Dần:



2.3 Trường Hợp Sắp Xếp Giảm Dần:



3

NHẬN XÉT

3.1 Dữ Liệu Ngẫu Nhiên:

Các thuật toán sắp xếp O(nlogn) hay O(n*k) có tốc độ nhanh.
Các thuật toán cổ điển như Bubble, Selection, Insertion có tốc độ chậm.

3.2 Dữ Liệu Tăng Dần:

Như trên tuy nhiên Insertion có phần cải tiến do đã lưu được dãy đã tăng.

3.3 Dữ Liệu Giảm Dần:

Các thuật toán đặc biệt là các thuật toán $O(n^2)$ có tốc độ chậm hơn đáng thấy.