BÁO CÁO THỰC NGHIỆM - THUẬT TOÁN SẮP XẾP

Người thực hiện: Võ Đặng Phương Thùy
MSSV: 22521459
Lớp: ATTN2022
Link Github: https://github.com/S0me0ne24/First-project

I. Giới thiệu thuật toán

1.Quick Sort

1.1. Mô tả

- Là một thuật toán hiệu quả dựa trên ý tưởng chia để trị với cách hoạt động như sau:
    + Chọn một phần tử làm chốt và phân vùng mảng đã cho xung quanh chốt đã chọn sao cho mảng được chia thành hai phần bằng cách so sánh từng phần tử của mảng với phần tử chốt. Một mảng bao gồm các phần tử nhỏ hơn hoặc bằng phần tử chốt và một mảng gồm các phần tử lớn hơn phần tử chốt.
    + Quá trình phân chia này diễn ra cho đến khi độ dài của các mảng con đều bằng 1.

1.2. Độ phức tạp

- Độ phức tạp thời gian trung bình và trong trường hợp tốt nhất là O(nlogn)
- Độ phức tạp thời gian trong trường hợp xấu nhất là O(n^2) xảy ra khi chốt được chọn kém.

1.3. Đánh giá

- Ưu điểm:
    + Hiệu quả trên các tập dữ liệu lớn.
    + Có chi phí hoạt động thấp vì chỉ yêu cầu một lượng nhỏ bộ nhớ để hoạt động.
- Nhược điểm:
    + Không phải là một lựa chọn tốt cho các tập dữ liệu nhỏ.
    + Có thể nhạy cảm với sự lựa chọn của chốt. Thuật toán sẽ chạy chậm khi chọn phải cực trị.

1.4. Cài đặt

```
1.  void quickSort(float a[], int l, int r){//sắp xếp dãy từ vị trí l đến r của dãy a không giảm
2.      if (l>=r) return;
3.      int pv=l; //chọn mốc là vị trí đầu tiên của dãy cần sắp xếp
4.      int i=l;
5.      for (int j=l+1; j<=r; j++){
6.          if (a[j]<a[pv]){
7.              i++;
8.              swap(a[i],a[j]);
9.          }
10.     }
11.     swap(a[i],a[pv]);
12.     quickSort(a,l,i-1); quickSort(a,i+1,r);
13. }
14.
```

2.Merge Sort

2.1. Mô tả

- Là một thuật toán sắp xếp hoạt động bằng cách chia một mảng thành các mảng con nhỏ hơn, sắp xếp từng mảng con, sau đó hợp nhất các mảng con đã sắp xếp lại với nhau để tạo thành mảng được sắp xếp cuối cùng.
- Điều này có nghĩa là nếu mảng trở nên trống hoặc chỉ còn lại một phần tử, phép chia sẽ dừng lại, tức là trường hợp cơ sở để dừng đệ quy.

2.2. Độ phức tạp

- Độ phức tạp về thời gian là O(nlogn) trong cả 3 trường hợp (tệ nhất, trung bình và tốt nhất) vì sắp xếp hợp nhất luôn chia mảng thành hai nửa và mất thời gian tuyến tính để hợp nhất hai nửa.
- Độ phức tạp bộ nhớ O(n)

2.3. Đánh giá

- Ưu điểm:
  + Tương đối hiệu quả để sắp xếp các tập dữ liệu lớn.
  + Là một sắp xếp ổn định, có nghĩa là thứ tự của các phần tử có giá trị bằng nhau được giữ nguyên trong quá trình sắp xếp.
- Nhược điểm:
  + Chậm hơn so với các thuật toán sắp xếp khác cho các tập dữ liệu nhỏ hơn.

2.4. Cài đặt

```
void mergeSort(float a[], float b[], int l, int r){
    if (l>=r) return;
    int mid=(l+r)/2;
    mergeSort(a,b,l,mid); mergeSort(a,b,mid+1,r);
    int il=l,el=mid,ir=mid+1,er=r;
    int i=l-1;
    while(il<=el || ir<=er){
        if (il>el){
            for (int j=ir; j<=er; j++){
                b[++i]=a[j];
            }
            break;
        }
        if (ir>er){
            for (int j=il; j<=el; j++){
                b[++i]=a[j];
            }
            break;
        }
        if (a[il]<=a[ir]){
            b[++i]=a[il]; il++;
        }
        else{
            b[++i]=a[ir]; ir++;
        }
    }
    for (int j=l; j<=r; j++){
        a[j]=b[j];
    }
}
```

3. Heap sort

3.1. Mô tả

- Là kỹ thuật sắp xếp dựa trên so sánh dựa trên cấu trúc dữ liệu Binary Heap. Tương tự như selection sort. Giúp sắp xếp các phần tử trong danh sách sao cho phần tử lớn nhất được xếp vào cuối danh sách. Và quá trình này được lặp lại cho các phần tử còn lại trong danh sách.

3.2. Độ phức tạp

- Độ phức tạp thời gian: O(nlogn)
- Độ phức tạp bộ nhớ: O(1)

3.3. Đánh giá

- Ưu điểm:
    + Không cần thêm dung lượng bộ nhớ để hoạt động
    + Chạy nhanh
- Nhược điểm:
    + Không ổn định

3.4. Cài đặt

```
void heapify(float a[], int n, int root){
    int imax=root;
    int l=root*2+1; int r=root*2+2;
    if (l<n && a[imax]<a[l]) imax=l;
    if (r<n && a[imax]<a[r]) imax=r;
    if (imax!=root){
        swap(a[imax],a[root]);
        heapify(a,n,imax);
    }
}
void heapSort(float a[], int n){
    for (int i=n/2-1; i>=0; i--){
        heapify(a,n,i);
    }
    for (int i=n-1; i>=0; i--){
        swap(a[0],a[i]);
        heapify(a,i,0);
    }
}
```

4. Std::sort (hàm có sẵn trong thư viện của c++)

4.1. Mô tả

- Hàm sort() của c++ sử dụng thuật toán Intro Sort, là một thuật toán được kết hợp từ nhiều thuật toán sắp xếp khác để tăng hiệu suất, giảm thời gian chạy như Quick Sort, Heap Sort và Insertion Sort.

4.2. Độ phức tạp

- Độ phức tạp thời gian trung bình là O(nlogn)
- Độ phức tạp bộ nhớ là O(1)

4.3. Đánh giá

- Ưu điểm
  - + Tốc độ chạy nhanh
  - + Dễ cài đặt vì là hàm có sẵn trong thư viện
- Nhược điểm
  - + Không ổn định

4.4. Cài đặt

```
sort(arr,arr+n); //nếu như sắp xếp dãy không giảm
sort(arr,arr+n,greater<float>()); //nếu như sắp xếp dãy không tăng
```

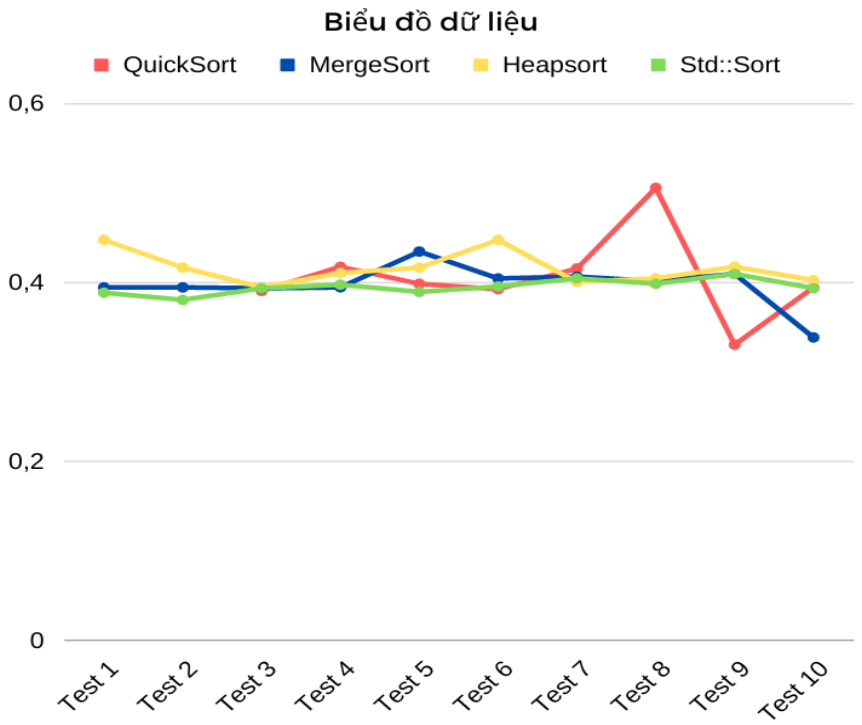
II. Kết quả thực nghiệm

- Bộ dữ liệu được tạo gồm 10 test từ 01 đến 10, mỗi test gồm 1 dãy khoảng 1 triệu số thực (ngẫu nhiên); dãy ở test thứ nhất đã có thứ tự tăng dần, dãy ở test thứ hai có thứ tự giảm dần, 8 dãy còn lại trật tự ngẫu nhiên.

Bảng dữ liệu thời gian thực thi

	QuickSort	MergeSort	Heapsort	Std::Sort
test01	inf	0.395	0.448	0.389
test02	inf	0.395	0.417	0.381
test03	0.391	0.394	0.395	0.394
test04	0.418	0.395	0.411	0.398
test05	0.399	0.435	0.417	0.390
test06	0.393	0.405	0.448	0.396
test07	0.416	0.407	0.401	0.405
test08	0.506	0.402	0.405	0.399
test09	0.331	0.410	0.418	0.410
test10	0.395	0.339	0.403	0.394

Đơn vị: giây (s)



III. Báo cáo thực nghiệm

- Thuật toán Quick Sort khi chạy trong trường hợp ở test 1 và 2 thì xảy ra lỗi chạy quá thời gian. Nguyên nhân do chốt được chọn luôn ở vị trí đầu, và vị trí này

luôn là cực trị, dẫn đến không tìm được vị trí có thể đổi chỗ ở hai bên chốt. Có thể giải quyết bằng cách chọn chốt là vị trí trung vị hay dùng Random QuickSort thay cho QuickSort.

- Mặc dù độ phức tạp về thời gian trong trường hợp xấu nhất của QuickSort là O(n^2) cao hơn nhiều thuật toán sắp xếp khác như MergeSort và HeapSort, nhưng trên thực tế, QuickSort nhanh hơn vì vòng lặp bên trong của nó có thể được triển khai hiệu quả trên hầu hết các kiến trúc và trong hầu hết các dữ liệu thế giới thực. QuickSort có thể được triển khai theo nhiều cách khác nhau bằng cách thay đổi lựa chọn trục, do đó trường hợp xấu nhất hiếm khi xảy ra đối với một loại dữ liệu nhất định. Tuy nhiên, MergeSort thường được coi là tốt hơn khi dữ liệu lớn và được lưu trữ trong bộ nhớ ngoài.
- MergeSort chạy ổn định và không có chênh lệch thời gian nhiều sau mỗi test case được sinh ngẫu nhiên. Tuy nhiên, nếu ta giảm số lượng input xuống thì mergesort lại thực thi rất lâu.
- Thuật toán HeapSort có thời gian chạy lâu hơn so với các thuật toán khác do bị tốn chi phí hằng số lớn. Khi mảng sắp xếp tăng dần thì việc xây dựng cấu trúc dữ liệu Heap trở nên tốn thời gian hơn do các phần tử trong Heap liên tục bị đổi gốc. Ngược lại khi dãy sắp xếp giảm dần thì thuật toán chạy hiệu quả và nhanh hơn.