**BÁO CÁO**

VỀ KẾT QUẢ THÍ NGHIỆM GIỮA CÁC THUẬT TOÁN SẮP XẾP

# thuật toán

Bao gồm 10 thuật toán sau đây:

* SelectionSort
* RadixSort
* InsertionSort
* CoutingSort
* ShakerSort
* MergeSort
* HeapSort
* QuickSort
* InterchangeSort
* BubbleSort

Lưu ý: chỉ xem xét yêu cầu sắp xếp mảng theo hướng tăng dần. Ý tưởng vẫn đúng với trường hợp sắp xếp giảm dần

## Ý tưởng

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ý tưởng | Time Complexity | Space Complexity |
| Bubble Sort | * Xét lần lượt các cặp 2 phần tử liên tiếp. Nếu phần tử đứng sau nhỏ hơn phần tử đứng trước, ta đổi chỗ 2 phần tử. Nói cách khác, phần tử nhỏ nhất sẽ **nổi** lên trên. * Lặp lại đến khi không còn 2 phần tử nào thỏa mãn. Có thể chứng minh được số lần lặp không quá N - 1, do một phần tử chỉ có thể **nổi** lên trên không quá N−1 lần. | O(n2) | 0 |
| Interchange Sort | thuật toán tương tự như Bubble sort. Interchange sort đưa phần tử về đúng vị trí của nó bằng cách so sánh tất cả các cặp phần tử trong mảng với nhau | O(n2) | 0 |
| Shaker Sort | Thuật toán cải thiện 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. | Tốtnhất:  O(n) | 0 |
| Xấu nhất: O(n2) |
| Insertion Sort | Ý tưởng chính của thuật toán là sắp xếp lần lượt từng đoạn gồm 1, 2, ..., N phần tử.  Giả sử ta đã sắp xếp xong i phần tử của mảng. Để sắp xếp i + 1 phần tử đầu tiên, ta tìm vị trí phù hợp của phần tử thứ i + 1 và "chèn" nó vào đó. | O(n2) | 0 |
| Selection Sort | Chọn phần tử nhỏ nhất đưa về vị trí đầu tiên của dãy hiện tại và không cần quan tâm đến nó nữa, khi đó dãy chỉ còn lại n-1 phần tử của dãy ban đầu, lúc đó dãy ta xét sẽ bắt đầu từ phần tử thứ 2 của mảng, chúng ta lập lại cho đến khi dãy hiện tại chỉ còn 1 phần tử. | O(n2) | 0 |
| Quick Sort | Thuật toán **chia để trị <Divide and Conquer>**.   * Chia dãy thành 2 phần, một phần "lớn" và một phần "nhỏ". * Chọn một khóa **pivot** * Những phần tử lớn hơn **pivot** chia vào phần lớn * Những phần tử nhỏ hơn hoặc bằng **pivot** chia vào phần nhỏ. * Gọi đệ quy để sắp xếp 2 phần. | Tốtnhất:  O(nLogn) | 0 |
| Xấu nhất: O(n2) |
| Couting Sort | Thuật toán sắp xếp dựa trên việc sử dùng **các giá trị khoá <key values>** trên một khoảng xác định. Xây dựng một mảng để đếm số lượng các phần tử có các **giá trị khoá <key value>** khác nhau. Sau đó tính toán để đưa mỗi phần tử về vị trí đúng trong mảng. | O(n +k),  n: số lượng phần tử trong mảng  k: khoảng giá trị của mảng | O(n + k) |
| Radix Sort | Khác với tất cả các thuật toán nêu trên, RadixSort không sử dụng việc so sánh 2 phần tử.   * Đầu tiên, thuật toán sẽ chia các phần tử thành các nhóm, dựa trên chữ số cuối cùng (hoặc dựa theo bit cuối cùng, hoặc vài bit cuối cùng). * Sau đó ta đưa các nhóm lại với nhau, và được danh sách sắp xếp theo chữ số cuối của các phần tử. Quá trình này lặp đi lặp lại với chữ số át cuối cho tới khi tất cả vị trí chữ số đã sắp xếp. | O(n) | O(n) |
| Merge Sort | Thuật toán **chia để trị <Divide and Conquer>**.  Merge Sort hoạt động kiểu đệ quy:   * Đầu tiên chia dữ liệu thành 2 phần, và sắp xếp từng phần. * Sau đó gộp 2 phần lại với nhau. Để gộp 2 phần, ta làm như sau: * Tạo một dãy A mới để chứa các phần tử đã sắp xếp. * So sánh 2 phần tử đầu tiên của 2 phần. Phần tử nhỏ hơn ta cho vào A và xóa khỏi phần tương ứng. * Tiếp tục như vậy đến khi ta cho hết các phần tử vào dãy A. | O(nLogn) | O(n) |
| Heap Sort | Thuật toán so sánh sử dụng cấu trúc dữ liệu **<Binary Heap>** thoả điều kiện giá trị của “parent” phải luôn lớn hơn giá trị của “child". Thuật toán tìm phần tử lớn nhất và đặt vào vị trí cuối mảng và lặp lại tương tự với các phần tử còn lại. | O(nLogn) | 0 |

## Các bước thực hiên thuật toán

|  |  |
| --- | --- |
| Thuật toán  (có ảnh) | Các bước thực hiện  (Sắp xếp tăng dần, kích thước mảng là n, kí hiệu mảng arr[]) |
| [Bubble Sort](Hình%20ảnh/Bubble%20Sort.PNG) | Đặt i = 1, j = 0.  Trong khi (i <= n-2):  Bước 1: trong khi (j < n – i), so sánh arr[j] và arr[j+1], nếu arr[j] > arr[j + 1] thì swap(arr[j],arr[j+1]), tăng j lên 1.  Bước 2: Đặt lại j = 0. |
| [Interchange Sort](Hình%20ảnh/Interchage%20Sort.PNG) | Lần lượt so sánh các cặp phần tử trong mảng, hoán đổi vị trí nếu cặp phần tử có vị trí không đúng. |
| [Shaker Sort](Hình%20ảnh/Shaker%20Sort.PNG) | Gọi i và j lần lượt là vị trí phần tử đầu và cuối  Bước 1: Duyệt qua mảng từ vị trí i đến vị trí j   * 1. So sánh các cặp giá trị liên tiếp, hoán đổi nếu giá trị phần tử bên trái lớn hơn phần tử bên phải.   2. Giảm j đi 1. Lúc này phần tử lớn nhất đã được đẩy về j.   Bước 2: bắt đầu duyệt mảng từ j về i   * 1. So sánh các cặp giá trị liên tiếp, hoán đổi nếu giá trị phần tử bên phải nhỏ hơn phân tử bên trái   2. Tăng i lên 1. Lúc này phân tử nhỏ nhất được đưa về i   Bước 3: Lặp lại bước 1 và 2 cho đến khi i = j |
| [Insertion Sort](Hình%20ảnh/Insertion%20Sort.PNG) | Lần lượt xét các phần tử có vị trí i trong mảng  Bước 1: gọi j = i-1 là vị trí phần tử phía bên trái phần tử đang xét, x là giá trị của phần tử đang xét.  Bước 2:   * 1. Trong khi (j >= 0), so sánh giá trị phần tử j với x, nếu lớn hơn thì gán giá trị x cho phần tử j. Giảm j xuống 1.   2. Gán giá trị x cho phần tử j+1. |
| [Selection Sort](Hình%20ảnh/Selection%20Sort.PNG) | Bước 1: Tìm giá trị nhỏ nhất trong mảng  Bước 2: Đưa giá trị đó về đầu mảng  Bước 3: lặp lại các bước trên với mảng chứa n – 1 phần tử còn lại |
| Quick Sort | Bước 1: chọn mốc là vị trí giữa (pivot)  Bước 2: gọi [find\_pivot\_index()](Hình%20ảnh/QuickSort_find_pivot_index.PNG) để sắp xếp và trả ra vị trí đúng của pivot, thoả mãn sao cho các phần tử bên trái pivot nhỏ hơn pivot và bên phải thì lớn hơn  Bước 3: gọi lại [QuickSort()](Hình%20ảnh/QuickSort_QuickSortTlk.PNG) với mảng bên trái pivot và bên phải pivot. |
| [Couting Sort](Hình%20ảnh/Couting%20Sort.PNG) | Gọi phần tử lớn nhất trong mảng là max,  Phần tử nhỏ nhất trong mảng là min.  Khi đó ta có khoảng giá trị range = [min, max] với k = max – min + 1.  bước 1: tạo mảng đếm với kích thước bằng k, giá trị ban đầu là 0. Mảng này có nhiệm vụ đếm số lần xuất hiện của 1 giá trị thuộc range. Kí hiệu mảng đếm là count[].  bước 2: duyệt qua các phần tử trong mảng cần sắp xếp. Với mỗi phần tử ta tăng số lần xuất hiện của giá trị tương ứng lên 1 (count[arr[i]]++).  bước 3: điều chỉnh mảng đếm bắng cách lấy giá trị phần tử sau bẳng tổng chính nó cộng với phần tử kế bên trái.  bước 4:, tạo 1 mảng output tạm thời có kích thước bằng với mảng cần sắp xếp. Kí hiệu là output[]. Duyệt qua các phần tử trong mảng cần sắp xếp:   1. Nếu count[arr[i]] > 0, output[count[arr[i]] -1] = arr[i]. 2. Giảm count[arr[i]] đi 1.   bước 5: copy output vào arr. |
| [Radix Sort](Hình%20ảnh/Radix%20Sort.PNG) | Gọi phần tử lớn nhất trong mảng là max  Bước 1: tạo các mảng đại diện cho các chữ số từ -9 đến 9  Bước 2: gọi i = 1 là chữ số đầu tiên. Nếu max / (10 \* i) <= 0:   1. Đưa các phần tử có chữ số thứ i vào mảng tương ứng 2. Sắp xếp lại mảng bằng cách lấy các phần tử trong các mảng tương ứng với các chữ số từ -9 đến 9 đưa vào mảng 3. Lặp lại bước 2 với i = i + 1 |
| [Merge Sort](Hình%20ảnh/MergeSort_MergeTlk.PNG) | Hàm [partition()](Hình%20ảnh/Merging_partition.PNG) liên tục phân mảng thành hai nữa cho đến khi mảng con có kích thước bằng 1, sau đó tiến hành hợp 2 mảng đã sắp xếp lại bằng merging().  Hàm [merging()](Hình%20ảnh/Merging_merging.PNG) coi 2 mảng cần hợp đã sắp xếp tăng dần, lần lượt duyệt qua các phần tử của hai mảng và đưa vào mảng tạm thời, sau đó copy vào mảng cần sắp xếp.  Bước 1: chọn điểm chính giữa mid. Gọi hàm partition(0,mid) và partition(mid+1,n-1).  Bước 2: gọi hàm merging(0,mid,n-1) để tiến hành merge 2 nữa đã được sắp xếp. |
| [HeapSort](Hình%20ảnh/HeapSort_HeapTlk.PNG) | Hàm [pull()](Hình%20ảnh/HeapSort_pull.PNG) có nhiệm vụ xây dựng cấu trúc Binary Heap cho mảng, hàm tìm GTLN giữa “parent” và các "child" của nó, sau đó hoán đổi vị trí của “parent” và "child"”chứa GTLN.  Bước 1: gọi hàm pull() tính từ “parent” cuối cùng trong mảng  Bước 2: hoán đổi phần tử đầu tiên với phần tử cuối cùng.  Bước 3: Lặp lại các bước trên với các phần tử còn lại. |

## Đánh giá thuật toán

* Các thuật toán so sánh Bubble sort, Insertion sort, Interchange sort, Selection sort, Shaker sort không thích hợp với các mảng dữ liệu có kích thước quá lớn.
* Độ phức tạp thuật toán của Couting sort phụ thuộc vào khoảng giá trị của mảng, chi phí thời gian càng lớn nếu khoảng giá trị quá lớn so với kích thước mảng
* Merge sort, Radix sort, Couting có nhược điểm là cần phải chi thêm bộ nhớ.
* Tốc độ Quick sort phụ thuộc vào việc chọn phần tử mốc.

|  |  |  |
| --- | --- | --- |
| Thuật toán  (có ảnh) | Ưu điểm | Nhược điểm |
| Bubble Sort | * Code đơn giản, dễ hiểu. * Không tốn thêm bộ nhớ | * Không đủ nhanh với dữ liệu lớn |
| Interchange Sort | * Code đơn giản, dễ hiểu. * Không tốn thêm bộ nhớ | * Không đủ nhanh với dữ liệu lớn |
| Shaker Sort | * Thuật toán nhận diện được mảng đã sắp xếp * Không tốn thêm bộ nhớ | * Không đủ nhanh với dữ liệu lớn |
| Insertion Sort | * Chạy rất nhanh nếu danh sách có thứ tự gần đúng | * Không đủ nhanh với dữ liệu lớn |
| Selection Sort | * thuật toán đơn giản, dễ hiện thực * Có số lần hoán đổi vị trí ít | * Chỉ được áp dụng trong các trường hợp có số lần so sánh phần tử ít. |
| Quick Sort | Chạy nhanh (nhanh nhất trong các thuật toán sắp xếp dựa trên việc so sánh phần tử). Do đó Quick Sort được sử dụng trong nhiều thư viện của các ngôn ngữ như Java, C++. | * 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). Nếu ta chọn pivot ngẫu nhiên, thuật toán chạy với độ phức tạp trung bình là O(nlogn). * Không ổn định. |
| Couting Sort | Chạy nhanh. | Độ phức tạp thuật toán có thể lên tới O(N2) nếu khoảng giá trị k quá lớn so với kích thước mảng. |
| )Radix Sort | Có thể chạy nhanh hơn các thuật toán sử dụng so sánh. | Không thể sắp xếp số thực |
| Merge Sort | Chạy nhanh, ổn định | Cần dùng thêm bộ nhớ để lưu mảng |
| Heap Sort | Chạy nhanh | Không ổn định |

# thí nghiệm

## [Kết quả thí nghiệm](Result.xlsx)

## Đánh giá thí nghiệm

### Mảng Sorted

* Tốc độ: Shaker, Insertion > Merge > Quick > Radix > Shaker > Interchange > Bubble > Selection > Couting.

### Mảng Reversed

* Quick > Radix > Heap > Merge > Bubble > Insertion > Interchange > Shaker > Selection > Couting

### Mảng Random

* Radix > Quick > Heap > Merge > Insertion > Bubble > Interchange > Shaker > Selection > Couting

### Mảng Nearly Sorted

* Merge > Quick > Radix > Heapsort > Insertion > Couting > Shaker > Interchange > Bubble > Selection

## Nhận xét

* Shaker, Interchange, Bubble sắp xếp kém hiệu quả với mảng có dữ liệu ngẫu nhiên.
* Các thuật toán chạy tốt nhất: Radix, Couting, Merge, Heap, Quick.
* Insertion Sort xử lí kém với mảng được sắp xếp ngược.
* Radix Sort, CoutingSort và Selection Sort không phụ thuộc vào tình trạng dữ liệu, thời gian xử lí tương đối giống nhau giữa các tình trạng dữ liệu khác nhau.

Người báo cáo

Nguyễn Mạnh Tuấn

MSSV 1712875

Lớp 17CTT7B