**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

**---oOo---**



**BÁO CÁO THỰC HÀNH**

**CÁC THUẬT TOÁN SẮP XẾP**

Họ và tên sinh viên: Huỳnh Tấn Thọ

Mã số sinh viên: 19120383

Lớp: 19CTT2B

Giảng viên: Lê Đình Ngọc

Thành phố Hồ Chí Minh, tháng 12 năm 2020

**MỤC LỤC**

MỤC LỤC 2

PHẦN 1: CÀI ĐẶT THUẬT TOÁN SẮP XẾP TĂNG DẦN 3

1. Thuật toán sắp xếp chọn – Selection Sort 3

2. Thuật toán sắp xếp nổi bọt – Bubble Sort 4

3. Thuật toán sắp xếp chèn – Insertion Sort 6

4. Thuật toán sắp xếp trộn – Merge Sort 8

5. Thuật toán sắp xếp nhanh – Quick Sort 11

6. Thuật toán sắp xếp vun đống – Heap Sort 15

7. Thuật toán sắp xếp chèn nhị phân – Binary-Insertion Sort 18

PHẦN 2: KẾT QUẢ THỰC NGHIỆM VÀ NHẬN XÉT 21

1. Mảng có thứ tự ngẫu nhiên 21

2. Mảng được sắp xếp tăng sẵn 24

3. Mảng có thứ tự ngược 26

4. Mảng gần như được sắp xếp tăng 29

5. Nhận xét chung 31

TÀI LIỆU THAM KHẢO 32

**PHẦN 1: CÀI ĐẶT CÁC THUẬT TOÁN SẮP XẾP TĂNG DẦN**

**1. Thuật toán sắp xếp chọn – Selection Sort**

**1.1. Ý tưởng:**

* Trong mảng một chiều, ta duyệt mảng từ đầu đến cuối và chia mảng ta cần sắp xếp thành hai mảng con nhỏ hơn, bao gồm phần đã sắp xếp (nằm phía trước) và phần chưa được sắp xếp (nằm phía sau).
* Ta tìm phần tử nhỏ nhất của phần chưa được sắp xếp trong mảng. Sau đó di chuyển nó lên vị trí đầu tiên của phần đang xét.
* Sau mỗi lần lặp, phần tử nhỏ nhất của phần mảng chưa được sắp xếp sẽ được chọn và di chuyển lên phần mảng đã sắp xếp. Lặp lại đến khi kết thúc, ta sẽ được một mảng sắp xếp tăng dần.

**1.2. Thuật toán từng bước:**

* Ta có mảng một chiều A với N phần tử.
* Gọi i, j là chỉ số của các phần tử trong mảng. Gọi min là chỉ số của phần tử nhỏ nhất.
* Ta cài đặt thuật toán dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| For i from 0 to N – 2    min = i    For j from i + 1 to N – 1  If A[j] < A[min], min = j  end for  swap(A[i], A[min])    end for | Duyệt N – 1 phần tử đầu tiên của mảng, do ta muốn chia mảng thành hai phần: đã sắp xếp và chưa sắp xếp, nên ta chỉ cho i chạy từ 0 đến N – 2.  Khai báo biến min và khởi tạo min là chỉ số của phần tử đang xét.  Duyệt phần còn lại của mảng để tìm phần tử bé nhất trong đó. Do j chạy từ i + 1 nên ở vòng for phía trên, i chỉ chạy từ 0 đến N – 1.  Sau khi tìm được phần tử bé nhất trong phần chưa sắp xếp, ta hoán đổi vị trí của nó với phần tử đang xét, chính là A[i]. |

**1.3. Đánh giá thuật toán:**

* Độ phức tạp về thời gian:
  + Trường hợp tốt nhất: **O()**
  + Trường hợp trung bình: **O()**
  + Trường hợp xấu nhất: **O()**
* Độ phức tạp về không gian: do trong quá trình thực hiện, thuật toán không yêu cầu thêm bất kỳ bộ nhớ nào (trừ các biến tạm, biến chỉ số phần tử cần thiết phải có) nên độ phức tạp về không gian là **O(1)**.
* Độ ổn định của thuật toán: **không ổn định**. Trong một số trường hợp, các phần tử có cùng giá trị sau khi được sắp xếp sẽ hoán đổi vị trí với nhau.

**2. Thuật toán sắp xếp nổi bọt – Bubble Sort:**

**2.1. Ý tưởng:**

* Ta sử dụng hai vòng lặp để hoán đổi vị trí của hai phần tử kế cận nhau nếu chúng đang ở sai thứ tự mà ta đang cần sắp xếp.
* Lặp lại cho đến khi hết mảng. Khi kết thúc, ta được một mảng sắp xếp tăng dần.

**2.2 Thuật toán:**

* Ta có mảng một chiều A với N phần tử.
* Gọi i, j là chỉ số của các phần tử trong mảng.
* Ta cài đặt thuật toán dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| for i from 0 to N – 2  for j from 0 to N – i – 2  if A[j] > A[j + 1]  HoanVi(A[j], A[j + 1])  End for  End for | Duyệt mảng.  So sánh giá trị của A[j] và A[j + 1].  Nếu A[j] > A[j + 1], tức là hai phần tử này đang ở sai thứ tự, ta thực hiện hoán đổi chúng. |

**2.3. Đánh giá thuật toán:**

* Độ phức tạp về thời gian:
  + Trường hợp tốt nhất: **O(*n*)**. Trường hợp này xảy ra khi mảng đã được sắp xếp tăng sẵn, lúc này thuật toán không cần phải thực hiện bất cứ phép hoán vị nào.
  + Trường hợp trung bình: **O()**
  + Trường hợp xấu nhất: **O()**. Trường hợp này xảy ra khi mảng được sắp xếp giảm dần. Do đó, thuật toán phải thực hiện n2 phép hoán vị. Mặc dù độ phức tạp trong trường hợp này vẫn là O(), thực nghiệm cho thấy thời gian chạy của thuật toán trong trường hợp này lâu hơn đáng kể.
* Độ phức tạp về không gian: do trong quá trình thực hiện, thuật toán không yêu cầu thêm bất kỳ bộ nhớ nào (trừ các biến tạm, biến chỉ số phần tử cần thiết phải có) nên độ phức tạp về không gian là **O(1)**.
* Độ ổn định của thuật toán: **ổn định**. Các phần tử có cùng giá trị sau khi được sắp xếp sẽ không hoán đổi vị trí với nhau.

**3. Thuật toán sắp xếp chèn – Insertion Sort**

**3.1. Ý tưởng:**

* + - Ta duyệt từng phần tử và chia mảng đang xét thành hai mảng con: phần mảng từ đầu đến vị trí trước phần tử đang xét (phần mảng này đã được sắp xếp sẵn) và phần còn lại của mảng.
    - Tại phần tử đang xét, ta tìm cách chèn nó vào đúng vị trí trong mảng con đầu tiên (mảng này đã được sắp xếp sẵn) sao cho mảng con đã xếp đó vẫn đảm bảo tính tăng dần.
    - Cụ thể hơn, trước hết ta có một mảng con chỉ gồm phần tử đầu tiên, đương nhiên đó là mảng tăng dần. Ta duyệt lần lượt từ phần tử thứ 2 trở đi. Tại i, ta chèn phần tử A[i] đó vào một vị trí thích hợp nào đó trong đoạn từ A[0] đến A[i - 1] sao cho dãy số từ A[0] đến A[i – 1] vẫn đảm bảo tính tăng dần.
    - Ta lặp lại cho đến khi hết mảng. Khi kết thúc, ta được một mảng sắp xếp tăng dần.

**3.2. Thuật toán:**

* Ta có mảng một chiều A với N phần tử.
* Gọi i, j là chỉ số của các phần tử trong mảng.
* Gọi k là một biến tạm.
* Ta cài đặt thuật toán dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| For i from 1 to N – 1    k = A[i]    j = i – 1      while j >= 0 and a[j] > k      a[j + 1] = a[j]      j = j - 1  end while  a[j + 1] = k  end for | Duyệt từng phần tử trong mảng A. Ta duyệt mảng từ A[1] thay vì A[0] là bởi trước tiên ta muốn tạo một mảng con đã được sắp xếp tăng sẵn và ta mặc định ngầm mảng con chỉ có 1 phần tử là mảng tăng, do đó ta lấy phần tử A[0] làm mảng con tăng sẵn.  Lưu giá trị của phần tử đang xét vào biến tạm k.  Khởi tạo biến chạy j = i – 1, dùng để tìm vị trí thích hợp để chèn A[i] (lúc này cũng là k).  Dùng vòng lặp while để kiểm tra. Điều kiện j ≥ 0 để đảm bảo vị trí chèn vẫn nằm trong mảng, điều kiện a[j] > k để đảm bảo vị trí chèn là thích hợp sao cho mảng con vẫn tăng dần sau khi chèn.  Nếu vị trí chèn không thích hợp, ta dời phần tử đang xét về bên phải của mảng để nhường chỗ.  Giảm j một đơn vị để lần lượt tìm vị trí chèn phù hợp.  Sau khi vòng while kết thúc, cũng là lúc ta đã tìm được vị trí chèn phù hợp. Sở dĩ ta gán a[j + 1] = k là vì điều kiện kết thúc vòng lặp while.  Thứ nhất, vòng lặp kết thúc có thể do j < 0, cũng có nghĩa là tất cả phần tử của mảng con đang xét đều lớn hơn biến k, lúc này nếu ta chèn a[j] = k sẽ gây lỗi.  Thứ hai, khi gặp phần tử a[j] < k cũng sẽ kết thúc vòng lặp while, nếu ta gán a[j] = k sẽ gây mất dữ liệu và bị sai, ta phải gán a[j + 1] = k, và điều này đúng vì trước đó ta đã dời phần tử a[j + 1] sang bên phải, nên sẽ không gây mất dữ liệu. |

**3.3. Đánh giá thuật toán:**

* Độ phức tạp về thời gian:
  + - Trường hợp tốt nhất: **O(*n*)**, khi mảng đã được sắp xếp tăng sẵn, lúc này thuật toán không cần phải thực hiện vòng lặp while ở bên trong mà chỉ đơn giản thực hiện công việc duyệt các phần tử có trong mảng.
    - Trường hợp trung bình: **O()**
    - Trường hợp xấu nhất: **O()**
* Độ phức tạp về không gian: do trong quá trình thực hiện, thuật toán không yêu cầu thêm bất kỳ bộ nhớ nào (trừ các biến tạm, biến chỉ số phần tử cần thiết phải có) nên độ phức tạp về không gian là **O(1)**.
* Độ ổn định của thuật toán: **ổn định**. Các phần tử có cùng giá trị sau khi được sắp xếp sẽ không hoán đổi vị trí với nhau.

**4. Thuật toán sắp xếp trộn – Merge Sort:**

**4.1. Ý tưởng:**

* + - Sử dụng đệ quy theo phương pháp chia để trị, ta gọi đệ quy và chia mảng cần sắp xếp ra thành hai nửa, và lưu giá trị của chúng vào hai mảng con.
    - Sau đó ta tiếp tục gọi đệ quy cho từng mảng con để tiếp tục chia nó thành hai nửa, cứ tiếp tục như vậy cho đến khi hai mảng con này chỉ chứa một phần tử, hoặc cho đến khi ta được hai mảng con này đã sắp xếp tăng dần (mảng chỉ chứa một phần tử cũng được tính là mảng tăng dần).
    - Lúc này, ta tìm cách trộn hai mảng con này thành một mảng lớn hơn sao cho mảng được tạo ra vẫn đảm bảo tính tăng dần.
    - Sau khi kết thúc, ta được một mảng đã sắp xếp theo thứ tự tăng dần.

**4.2. Thuật toán:**

* Ta có mảng một chiều A với N phần tử.
* Dựa theo ý tưởng trên, ta cần ba hàm để thực hiện sắp xếp trộn.
  + - Hàm newArray, dùng để tạo và lưu giá trị vào mảng con mới.
    - Hàm merge, dùng để trộn hai mảng con này thành một mảng lớn hơn sao cho mảng được tạo ra vẫn đảm bảo tính tăng dần.
    - Hàm mergeSort, dùng để gọi đệ quy và tiếp tục chia mảng thành hai nửa. Ngoài ra, hàm này còn gọi hàm newArray và hàm merge.
* Trước tiên, ta cài đặt hàm mergeSort dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| mergeSort(A, left, right):  if right <= left, return  mid = (left+right)/2  A1 = newArray(A, left, mid)  A2 = newArray(A, mid+1, right)  mergeSort(A1, 0, mid - left)  mergeSort(A2, 0, right – mid - 1)  merge(A, A1, mid - left + 1, A2, right - mid);  end | Hàm mergeSort nhận 3 tham số là mảng A cần sắp xếp, left là chỉ số của phần tử đầu và right là chỉ số của phần tử cuối.  Nếu chỉ số cuối ≤ chỉ số đầu, thoát khỏi hàm.  Ngược lại, khai báo biến mid là chỉ số của phần tử nằm chính giữa left và right, có giá trị là trung bình cộng của left và right.  Tạo ra hai mảng con mới, ta lưu các giá trị từ đầu tới mid của mảng A vào mảng con A1, và lưu các giá trị từ mid + 1 tới right vào mảng con A2.  Sau đó, gọi đệ quy hàm mergeSort để tiếp tục chia các mảng con thành những mảng con nhỏ hơn, với các tham số left và right tương ứng.  Sau khi có được hai mảng con tăng dần, ta gọi hàm merge để trộn chúng lại với nhau sao cho ta được một mảng đã sắp xếp tăng dần.  Ở đây, ta sẽ trộn hai mảng con A1 và A2 với số lượng phần tử lần lượt là mid – left + 1 và right – mid vào mảng A kể từ vị trí left trong mảng A. |

* Tiếp theo, ta cài đặt hàm newArray dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| newArray(A, left, right)  n = right - left + 1  j = 0  newA = new int[n]  for i from left to right  newA[j] = a[i]  j = j + 1  end for  return newA  end | Hàm newArray nhận 3 tham số là mảng A cần chia, left là chỉ số của phần tử đầu và right là chỉ số phần tử cuối.  Khai báo biến n là số lượng phần tử trong mảng con ta sẽ tạo, và khởi tạo giá trị cho n là right – left + 1.  Khai báo biến chạy j và khởi tạo giá trị cho nó là 0.  Khai báo mảng một chiều newA với số lượng phần tử là n.  Lần lượt gán các giá trị trong đoạn từ vị trí left đến vị right của mảng A và cho newA  Cuối cùng, trả về mảng con mới newA và kết thúc hàm. |

* Cuối cùng, ta cài đặt hàm merge dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| merge(A, A1, N1, A2, N2)    i1 = 0, i2 = 0  j = 0  while i1 < n1 and i2 < n2  if A1[i1] <= A2[i2]  A[j] = A1[i1]  j = j + 1  i1 = i1 + 1  else  a[j] = a2[i2];  j = j + 1  i2 = i2 + 1  end while    while i1 < n1  a[j] = a1[i1]  j = j + 1  i1 = i1 + 1  end while  while i2 < n2  a[j] = a2[i2]  j = j + 1  i2 = i2 + 1  end while  end | Hàm merge nhận 6 tham số đầu vào, bao gồm mảng A để lưu kết quả trộn, A1 và A2 là hai mảng con cần trộn, N1 và N2 lần lượt là số lượng phần tử của hai mảng con A1 và A2, và left là vị trí trong mảng A và tại đó ta bắt đầu thực hiện việc lưu kết quả  Khai báo hai biến đếm i1 và i2, lần lượt dành cho hai mảng con A1 và A2, và khởi tạo giá trị của chúng là 0.  Khai báo biến chạy j dùng để xác định vị trí mà ta cần lưu kết quả trong mảng A. Ở đây ta khởi tạo j là 0.  Chạy vòng lặp while để lần lượt duyệt các phần tử trong hai mảng con A1 và A2.  Nếu phần tử thứ i1 trong mảng A1 bé hơn phần tử i2 trong mảng A2, thì ta gán A[j] = A1[i1]. Đồng thời, tăng j và i1 lên một đơn vị.  Ngược lại, ta gán A[j] = A2[i2]. Đồng thời, tăng j và i2 lên một đơn vị.  Nói cách khác, trong mỗi lần lặp, ta đều có được một cặp giá trị A1[i1] và A2[i2]. Lúc này, ta cố gắng đưa phần tử nhỏ hơn vào mảng kết quả A nhằm đảm bảo mảng A vẫn tăng dần.  Vòng lặp bên trên sẽ kết thúc khi tất cả các phần tử của một trong hai mảng con A1 và A2 đều đã được chuyển vào mảng A.  Còn lại những phần tử thừa chưa được thêm vào, ta dùng vòng lặp while để đưa chúng vào mảng A.  Lý do mà những phần tử này bị thừa ra là vì chúng lớn hơn các phần tử đã được thêm vào A nên sau khi so sánh, chúng bị bỏ qua.  Mặt khác, do chúng đều đã được sắp xếp tăng dần nên khi thêm vào mảng A, ta vẫn đảm bảo được tính tăng dần. |

**4.3. Đánh giá thuật toán:**

* Độ phức tạp về thời gian:
* Trường hợp tốt nhất: **O(*n.logn*)**
* Trường hợp trung bình: **O(*n.logn*)**
* Trường hợp xấu nhất: **O(*n.logn*)**
* Độ phức tạp về không gian: do trong quá trình thực hiện, thuật toán yêu cầu thêm bộ nhớ nào để lưu trữ các mảng con, nên độ phức tạp về không gian là **O(*n*)**.
* Độ ổn định của thuật toán: **ổn định**. Các phần tử có cùng giá trị sau khi được sắp xếp sẽ không hoán đổi vị trí với nhau.

**5. Thuật toán sắp xếp nhanh – Quick Sort:**

**5.1. Ý tưởng:**

* + - Thuật toán sắp xếp Quick Sort là một thuật toán sử dụng đệ quy theo phương pháp chia để trị.
    - Ở thuật toán này, ta chọn một phần tử trong mảng làm điểm đánh dấu. Ta gọi nó là pivot. Thuật toán sẽ thực hiện chia mảng thành các mảng con dựa vào pivot đã chọn.
    - Mấu chốt chính của thuật toán Quick Sort là việc phân đoạn mảng đang xét thành hai mảng con xoay quanh một phần tử ta chọn làm pivot, sao cho sau khi hoàn thành việc phân đoạn này, ta được một mảng con chỉ gồm các phần tử nhỏ hơn pivot nằm bên trái pivot và một mảng con chỉ gồm các phần tử lớn hơn pivot nằm ở bên phải pivot.
    - Cụ thể hơn, với một mảng đã cho và một phần tử đặt là pivot. Ta tìm cách đặt pivot vào đúng vị trí của mảng đã sắp xếp bằng cách di chuyển tất cả các phần tử của mảng mà nhỏ hơn pivot sang bên trái vị trí của pivot, và di chuyển tất cả các phần tử của mảng mà lớn hơn pivot sang bên phải vị trí của pivot. Khi đó ta sẽ có 2 mảng con: mảng bên trái của pivot (bao gồm các phần tử lớn hơn pivot) và mảng bên phải của pivot (bao gồm các phần tử lớn hơn pivot).
    - Tiếp tục gọi đệ quy, ta lặp lại các thao tác chọn pivot và phân đoạn với mỗi mảng con.
    - Sau khi kết thúc, ta được một mảng đã sắp xếp tăng dần.

**5.2. Thuật toán:**

* + - Ta có mảng A với N phần tử.
    - Dựa theo ý tưởng trên, ta triển khai thuật toán Quick Sort bằng 2 hàm con:
    - Hàm quickSort: dùng để sắp xếp mảng và gọi đệ quy để tiếp tục sắp xếp các mảng con.
    - Hàm partition: ta gọi hàm này bên trong hàm quickSort, nhằm thực hiện công việc phân đoạn và chọn ra pivot thích hợp để chia mảng thành các mảng con.
    - Ta cài đặt hàm quickSort dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| quickSort(A, left, right)  if (right <= left)  return  p = partition(A, left, right)  quickSort(A, left, p - 1)  quickSort(A, p + 1, right)  end | Hàm quickSort nhận 3 tham số đầu vào, với A là mảng một chiều cần được sắp xếp tăng dần, left là vị trí phần tử đầu của mảng và right là vị trí phần tử cuối cùng trong mảng A.  Nếu right ≤ left, tức nếu mảng có ít hơn hoặc bằng một phần tử, ta kết thúc hàm.  Nếu không, ta thực hiện khai báo biến p. Biến p này sẽ lưu vị trí pivot thích hợp mà ta sẽ dùng làm mốc để tiếp tục chia mảng thành các mảng con và gọi đệ quy. Ngoài ra, hàm partition còn thực hiện việc phân đoạn mảng A xung quanh p sao cho phần mảng con nằm bên trái p chỉ bao gồm những phần tử nhỏ hơn p, và phần mảng con nằm bên phải p chỉ bao gồm các phần tử lớn hơn p.  Sau khi chọn được p thích hợp, ta tiến hành gọi đệ quy để sắp xếp cho từng mảng con. Nếu để ý, khi gọi đệ quy, ta không chọn phần tử A[p] vì lúc này vị trí pivot này đã là thích hợp, và ta chỉ đang cần thực hiện sắp xếp với các mảng con bên trái A[p] hoặc bên phải A[p].  Kết thúc hàm. |

* + - Ta cài đặt hàm partition dựa theo mã giả sau đây

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| partition(A, left, right)  temp = random number from left to right  HoanVi(a[temp], a[right])  pivot = right  low = left  high = right – 1  while (true)  while low<=high  and A[low]<A[pivot]  low = low + 1  end while  while high>=low  and A[high]>A[pivot]  high = high - 1  end while  if low >= high  break  HoanVi(a[low], a[high])  low = low + 1  high = high – 1  end while  HoanVi(a[low], a[right])  return low  end | Hàm partition nhận 3 tham số đầu vào, với A là mảng một chiều cần phân đoạn, left là vị trí phần tử đầu của mảng và right là vị trí phần tử cuối cùng trong mảng A.  Thông thường, ta sẽ chọn pivot là phần tử phải cùng, tức A[right]. Tuy nhiên, nếu làm như thế, giả sử ta rơi vào trường hợp xấu nhất, tức là khi mảng đã được sắp xếp tăng sẵn, thuật toán này sẽ không còn “nhanh” nữa do độ phức tạp sẽ tăng lên, gây mất nhiều thời gian chạy hoặc thậm chí thuật toán sẽ không chạy nếu N tương đối lớn  Để giảm tối đa khả năng rơi vào trường hợp xấu nhất, ta sẽ hoán đổi phần tử A[right] với một phần tử ngẫu nhiên trong mảng, rồi sau đó ta mới tiến hành chọn pivot.  Khai báo biến low và high dùng làm các biến chạy khi duyệt mảng. Từ hai biến chạy này, tại mỗi vòng lặp phía sau, ta luôn có được một cặp phần tử để thực hiện phép so sánh. Do biến chạy sẽ chạy từ hai đầu của mảng con trước pivot và ta chọn A[right] là pivot nên ta khởi tạo giá trị cho low là left, và high là right – 1.  Ta khởi tạo một vòng lặp while để di chuyển lần lượt các biến low và high về một vị trí nào đó ở giữa mảng. Bên trong, ta sử dụng 2 vòng lặp while.  Vòng while thứ nhất, ta dùng để di chuyển dần dần biến low về giữa cho tới khi giá trị của low bằng high hoặc khi A[low] ≥ A[pivot]  Tương tự, vòng while thứ hai dùng để di chuyển dần dần biến high về giữa cho tới khi giá trị của high bằng low hoặc khi A[high] ≤ A[pivot].  Sau khi hai vòng lặp while kết thúc, ta kiểm tra bằng một câu lệnh if, nếu như vị trí low ≥ vị trí right, ta thoát khỏi vòng lặp do lúc này đã thực hiện xong.  Đến đoạn này, do dòng lệnh if đó nên điều kiện khiến cho hai vòng lặp while bên trên dừng lại là do có một trong hai phần tử A[low] và A[high] đang xét nằm sai vị trí mà ta muốn sắp xếp, tức là A[low] ≥ A[pivot] và A[high] ≤ A[pivot]. Điều này xung đột với mục tiêu của hàm partition. Do đó, ta hoán đổi chúng với nhau.  Sau khi hoán vị chúng, ta tiếp tục xét cặp ở bên trong cho đến khi kết thúc vòng lặp while, tức khi low ≥ high.  Sau khi kết thúc vòng lặp while, lúc này low ≥ high, nên ta hóa đổi vị trí của A[low] với A[pivot] nhằm đem pivot làm mốc chia đôi mảng. Ngay lúc này, pivot sẽ chia mảng A làm hai mảng con thỏa mãn yêu cầu của hàm partition, mảng con bên trái pivot chỉ bao gồm các phần tử nhỏ hơn pivot và mảng con bên phải pivot chỉ bao gồm các phần tử lớn hơn pivot.  Ta trả về giá trị là giá trị của biến low. Giá trị này sẽ được dùng trong hàm quickSort bên trên.  Kết thúc hàm. |

**5.3. Đánh giá thuật toán:**

* Độ phức tạp về thời gian:
  + - Trường hợp tốt nhất: **O(*n.logn*)**
    - Trường hợp trung bình: **O(*n.logn*)**
    - Trường hợp xấu nhất: **O(*n2*)**. Điều này xảy ra khi mảng được sắp xếp tăng dần sẵn và ta chọn vị trí pivot là phần tử cuối cùng của mảng. Lúc này, hàm partition sẽ chia mảng ra làm hai mảng con với kích thước lần lượt là N – 1 và 1 trong suốt quá trình chạy, dẫn đến việc đệ quy quá nhiều lần gây phát sinh lỗi nếu kích thước mảng đủ lớn (thực nghiệm cho thấy, N ≥ 10000 sẽ gây phát sinh lỗi). Trong phạm vi bài tập này, ta chọn pivot là một phần tử ngẫu nhiên trong mảng, do đó xác suất xảy ra trường hợp xấu nhất là rất thấp và càng thấp hơn nữa khi kích thước mảng càng lớn. Tuy nhiên không vì thế mà độ phức tạp của thuật toán (nếu áp dụng theo cách của bài này) trong trường hợp xấu nhất là O(*n.logn*), do vẫn có khả năng trường hợp xấu nhất sẽ xảy ra mặc dù rất rất nhỏ.
* Độ phức tạp về không gian: trong quá trình thực hiện, thuật toán không yêu cầu thêm bộ nhớ nào để lưu trữ các mảng con (trừ các biến chạy và biến tạm cần thiết) tuy nhiên, do tính chất được cài đặt bằng đệ quy nên chương trình chạy sẽ tiêu tốn thêm bộ nhớ stack dùng cho đệ quy. Vì thế, độ phức tạp về không gian là **O(*logn*)**.
* Độ ổn định của thuật toán: **không ổn định**. Trong một số trường hợp, các phần tử có cùng giá trị sau khi được sắp xếp có thể bị hoán đổi vị trí với nhau.

**6. Thuật toán sắp xếp vun đống – Heap Sort:**

**6.1. Ý tưởng:**

* + - Thuật toán sắp xếp vun đống là thuật toán sắp xếp dựa trên các phép so sánh, và được cài đặt bằng cấu trúc dữ liệu có tên gọi Binary Heap – đống nhị phân. Cách hoạt động của thuật toán cũng tương tự như Selection Sort, trong đó ta sẽ tìm phần tử lớn nhất và đặt nó ở cuối. Lặp đi lặp lại cho đến hết mảng. Trước hết ta đi tìm hiểu một vài khái niệm cơ bản.
    - Đống nhị phân (Binary Heap) là một cây nhị phân hoàn chỉnh (tức cây nhị phân mà tại mỗi tầng trừ những nút lá, đều được lấp đầy và tất cả các nút đều nằm bên trái nhất có thể) mà trong đó các giá trị được lưu sao cho giá trị của nút cha sẽ lớn hơn (gọi là đống lớn – Max heap) hoặc nhỏ hơn (gọi là đống bé – Min Heap) giá trị của hai nút con. Cấu trúc này có thể được biểu diễn dưới dạng cây nhị phân hoặc dưới dạng mảng.
    - Ở đây, ta sẽ biểu diễn cấu trúc này dưới dạng mảng, do cây nhị phân hoàn chỉnh có thể dễ dàng biểu diễn được dưới dạng này, và nó cũng sử dụng bộ nhớ hiệu quả hơn. Nếu dữ liệu trong nút cha được lưu ở phần tử A[i], dễ dàng suy ra được hai nút con lần lượt sẽ là A[2i + 1] và A[2i + 2].
    - Quay trở lại ý tưởng của thuật toán, để sắp xếp mảng tăng dần, trước hết, ta xây dựng một đống lớn – Max Heap dựa vào mảng A cho trước. Ngay lúc này, phần tử lớn nhất sẽ được lưu ở gốc. Ta tiến hành thay thế nó với phần tử cuối cùng của đống (đây thực chất là đưa phần tử lớn nhất về cuối như đã đề cập bên trên), sau đó giảm kích thước của đống đi 1. Cuối cùng, ta tạo đống với gốc của cây nhị phân này. Ta lặp đi lặp lại việc tạo đống cho tới khi kích cỡ đống còn 1 trở xuống.
    - Cách tạo đống: việc tạo đống chỉ có thể áp dụng cho 1 nút khi và chỉ khi các nút con của nó đã được tạo đống. Do đó, việc tạo đống sẽ được cài đặt bằng cách sử dụng đệ quy.

**6.2. Thuật toán:**

* Ta có mảng A với N phần tử.
* Dựa theo ý tưởng trên, ta triển khai thuật toán Heap Sort bằng 2 hàm con:
* Hàm heapSort: dùng để sắp xếp mảng và gọi hàm tạo đống - hàm heapify.
* Hàm heapify: ta gọi hàm này bên trong hàm heapSort, nhằm thực hiện công việc tạo đống.
* Ta cài đặt hàm heapSort dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| heapSort(A, N)  for i from N/2 - 1 down to 0  heapify(A, N, i)  for i from N – 1 down to 1  HoanVi(A[0], A[i])  heapify(A, i, 0)  end for  end for  end | Hàm heapSort nhận 2 tham số là mảng A cần sắp xếp và N là số lượng phần tử của mảng.  Sử dụng vòng lặp for để tạo đống. Do ta đang cố tạo đống từ một mảng một chiều, đống này thực chất là một cây nhị phân, nên ta không thể duyệt mảng từ cuối, mà phải duyệt từ giữa mảng.  Ta tạo đống từ mảng A, với N phần tử, tại gốc i. Gốc i này sẽ đóng vai trò như là gốc trong hàm heapify (sẽ đề cập bên dưới). Nói chung, phần tử lớn nhất của đống sau khi tạo sẽ nằm tại gốc i này.  Sử dụng vòng lặp for để trích xuất ra từng phần tử từ đống.  Lúc này A[0] là gốc, ta hoán đổi A[0] và A[i] để di chuyển gốc về nút cuối cùng.  Ta tạo đống từ mảng A, với i phần tử, tại đỉnh 0  Kết thúc hàm. |

* Ta cài đặt hàm heapify dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| heapify(A, N, i)  k = i  left = 2 \* i + 1  right = 2 \* i + 2  if left < N and A[left] > A[k]  k = left  if right < N && A[right] > A[k]  k = right  if k <> i  HoanVi(A[i], A[k])  heapify(A, N, k)  end | Hàm heapSort nhận 3 tham số là mảng A cần sắp xếp, N là số lượng phần tử của mảng và i là gốc của đống.  Khai báo biến k và khởi tạo nó như gốc của đống.  Khai báo biến left để chứa chỉ số của node con trái, khởi tạo giá trị cho left là 2\*i + 1.  Khai báo biến right để chứa chỉ số của node con phải, khởi tạo giá trị cho left là 2\*i + 2.  Nếu left vẫn nằm trong mảng và nút con trái lớn hơn gốc hiện tại, gán k là left.  Nếu right vẫn nằm trong mảng và nút con phải lớn hơn gốc hiện tại, gán k là right.  Nếu biến k sau khi thực hiện 2 câu lệnh if bên trên khác với gốc ban đầu, hoán vị A[i] và A[k].  Sau đó, gọi đệ quy để tiếp tục tạo đống với mảng A có N phần tử, và gốc là k.  Kết thúc hàm. |

**6.3. Đánh giá thuật toán:**

* Độ phức tạp về thời gian:
  + - Trường hợp tốt nhất: **O(*n.logn*)**
    - Trường hợp trung bình: **O(*n.logn*)**
    - Trường hợp xấu nhất: **O(*n.logn*)**
* Độ phức tạp về không gian: trong quá trình thực hiện, thuật toán không yêu cầu thêm bộ nhớ nào để lưu trữ các mảng con (trừ các biến chạy và biến tạm cần thiết). Vì thế, độ phức tạp về không gian là **O(1)**.
* Độ ổn định của thuật toán: **không ổn định**. Trong một số trường hợp, các phần tử có cùng giá trị sau khi được sắp xếp sẽ có thể hoán đổi vị trí với nhau do quá trình tạo đống (heapify).

**7. Thuật toán sắp xếp chèn nhị phân – Binary Insertion Sort:**

**7.1. Ý tưởng:**

* + - Thuật toán sắp xếp chèn nhị phân là một phiên bản cải tiến của thuật toán sắp xếp chèn đã đề cập bên trên.
    - Thuật toán sắp xếp chèn nhị phân được cài đặt với cùng ý tưởng: ta duyệt từng phần tử và chia mảng đang xét thành hai mảng con: phần mảng từ đầu đến vị trí trước phần tử đang xét (phần mảng này đã được sắp xếp sẵn) và phần còn lại của mảng (chưa được sắp xếp). Tại phần tử đang xét, ta tìm cách chèn nó vào đúng vị trí trong mảng con đầu tiên (mảng này được sắp xếp sẵn) sao cho mảng con đã xếp đó vẫn đảm bảo tính tăng dần.
    - Tuy nhiên, vị trí chèn thích hợp mà ta cần tìm nằm trong một mảng con đã sắp xếp tăng dần. Vì thế, thay vì sử dụng tìm kiếm tuyến tính để tìm vị trí chèn thích hợp, ta sẽ sử dụng thuật toán tìm kiếm nhị phân.
    - Khi kết thúc, ta sẽ được một mảng đã sắp xếp tăng dần.

**7.2. Thuật toán:**

* Ta có mảng A với N phần tử.
* Dựa theo ý tưởng trên, ta triển khai thuật toán sắp xếp chèn nhị phân bằng 2 hàm con:
* Hàm binarySearch: ta gọi hàm này bên trong hàm binaryInsertionSort dùng để tìm kiểm vị trí chèn thích hợp bằng thuật toán tìm kiếm nhị phân.
* Hàm binaryInsertionSort: hàm sắp xếp chính, dùng để duyệt mảng và chèn phần tử thích hợp.
* Ta cài đặt hàm binarySearch dựa theo mã giả sau đây:

|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| binarySearch(A, x, left, right)  int mid  while (left <= right)    mid = (right - left)/2  if x = a[mid] return mid  if x < a[mid]  right = mid – 1  else  left = mid + 1  end while  return left  end | Hàm binarySearch nhận 4 tham số đầu vào, bao gồm mảng A cần tìm vị trí, x là phần tử cần tìm, left là chỉ số của phần tử đầu tiên trong mảng A, và right là chỉ số của phần tử cuối cùng trong mảng A.  Khai báo biến mid là chỉ số của phần tử chính giữa khoảng left và right ta cần xét.  Ta dùng vòng lặp while để duyệt mảng trong khoảng left và right. Do đó, hiển nhiên nếu left > right thì ta dừng vòng lặp.  Do mid là chỉ số của phần tử chính giữa left và right, ta khởi tạo giá trị cho mid là (right + left) / 2.  Nếu phần tử chính giữa, tức A[mid] bằng với x, thì ta trả về giá trị của mid  Nếu x < A[mid], tức phần tử ta cần tìm nằm ở nửa đầu mảng A, tức nó nằm trong khoảng từ A[left] đến A[mid – 1].  Ngược lại, nếu x > A[mid], tức phần tử ta cần tìm nằm ở nửa sau mảng A, tức nó nằm trong khoảng từ A[mid + 1] đến A[right].  Lặp đi lặp lại cho đến khi dừng.  Lúc này left sẽ là vị trí thích hợp để chèn giá trị x vào mảng A, do đó ta trả về left và kết thúc hàm. |

* Ta cài đặt hàm binaryInsertionSort dựa theo mã giả sau đây:

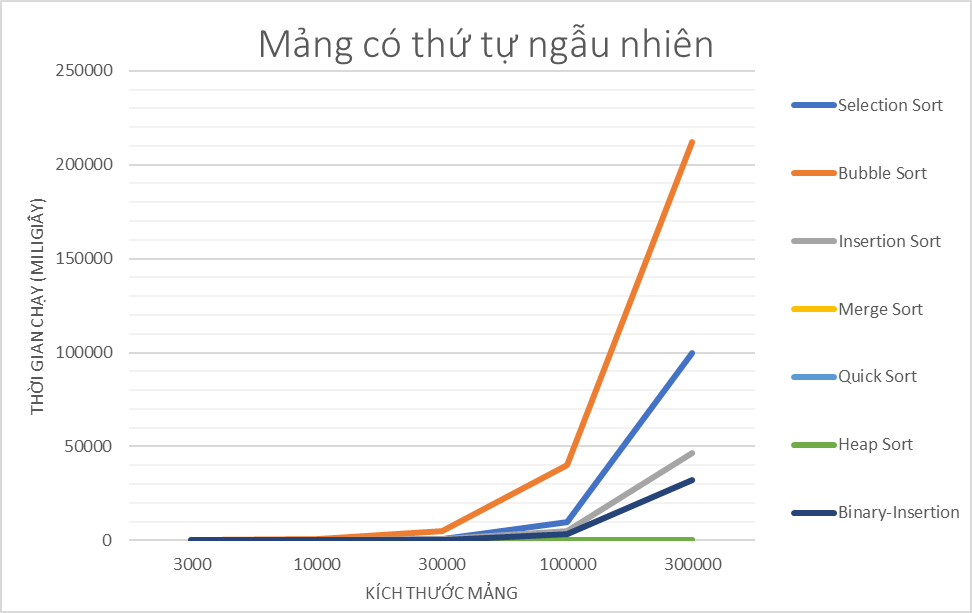
|  |  |
| --- | --- |
| **Mã giả** | **Giải thích** |
| binaryInsertionSort(A, N)  for i from 1 to N – 1  j = i – 1  k = A[i]  L=binarySearch(A, k, 0, j)  while j >= L  A[j + 1] = A[j]  j = j - 1  end while  A[j + 1] = k  end for  end | Hàm binaryInsertionSort nhận 2 tham số đầu vào, bao gồm mảng A cần sắp xếp và N là số lượng phần tử của mảng A.  Trước hết, ta duyệt mảng từ A[1] đến A[N – 1]. Tương tự như ở thuật toán sắp xếp chèn, ta duyệt mảng từ A[1] thay vì A[0] là bởi trước tiên ta muốn tạo một mảng con đã được sắp xếp tăng sẵn và ta mặc định ngầm mảng con chỉ có 1 phần tử là mảng tăng, do đó ta lấy phần tử A[0] làm mảng con tăng sẵn.  Khi tìm được vị trí chèn thích hợp, ta không chỉ đơn giản gán phần tử tại vị trí đó là A[i], mà còn phải thực hiện dời các phần tử trong mảng. Do đó, t khai báo biến j là biến chạy để thực hiện không việc này, và khởi tạo giá trị của j là i – 1 (do ta muốn chèn vào mảng con đứng trước A[i]).  Khai báo biến k làm biến lưu trữ tạm cho giá trị của phần tử A[i].  Khai báo biến L là vị trí chèn thích hợp của phần tử có giá trị là k trong khoảng 0 đến j của mảng A.  Sau khi tìm được vị trí chèn thích hợp là L, ta dùng vòng lặp while để thực hiện dời phần tử nhằm tạo chỗ trống cho việc chèn phần tử A[i] vào mảng con đã sắp xếp.  Nếu j không ở vị trí chèn thích hợp (tức j ≠ L), thì ta dời phần tử sang bên phải bằng cách gán A[j + 1] = A[j]. Ở đây ta không cần lo rằng A[i] sẽ mất giá trị, do trước đó ta đã gán k = A[i] để lưu giá trị của A[i]. Sau đó ta giảm j để tiếp tục xét.  Sau khi vòng lặp while kết thúc cũng là lúc j < L, tuy nhiên vị trí cần chèn thích hợp lại là L, do đó ta gán A[j + 1] = k (do giá trị A[i] trước đó đã được lưu vào biến k). Ở đây, ta không thiết lập điều kiện dừng của vòng lặp while là j > L rồi gán A[j] = k vì ta cần chèn A[i] vào vị trí L, nên A[L] phải được dời sang bên phải nhằm tạo chỗ trống. Nếu ta thiết lập điều kiện dừng là j > L, sẽ gây mất giá trị ban đầu của A[L] do A[L] không được dời qua bên phải, dẫn đến kết quả sai.  Ta lặp đi lặp lại cho đến khi hết mảng.  Kết thúc hàm. |

**7.3. Đánh giá thuật toán:**

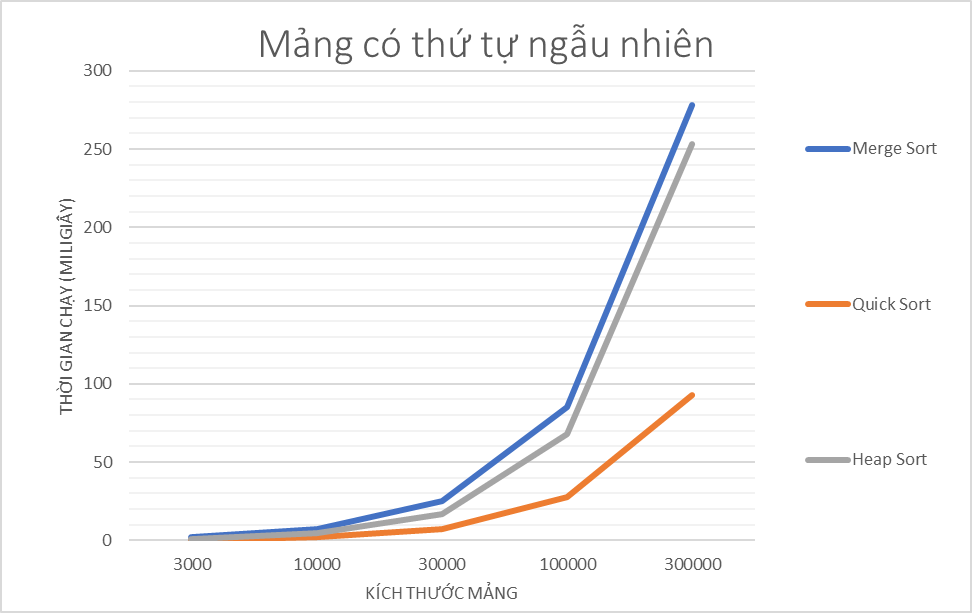
* Độ phức tạp về thời gian:
  + - Trường hợp tốt nhất: **O(*n*)**. Trường hợp này xảy ra khi mảng ta cần sắp xếp đã được sắp xếp tăng sẵn.
    - Trường hợp trung bình: **O()**
    - Trường hợp xấu nhất: **O()**
* Độ phức tạp về không gian: do trong quá trình thực hiện, thuật toán không yêu cầu thêm bất kỳ bộ nhớ nào (trừ các biến tạm, biến chỉ số phần tử cần thiết phải có) nên độ phức tạp về không gian là **O(1)**.
* Độ ổn định của thuật toán: **ổn định**. Các phần tử có cùng giá trị sau khi được sắp xếp sẽ không bị hoán đổi vị trí với nhau.

**PHẦN 2: KẾT QUẢ THỰC NGHIỆM VÀ NHẬN XÉT**

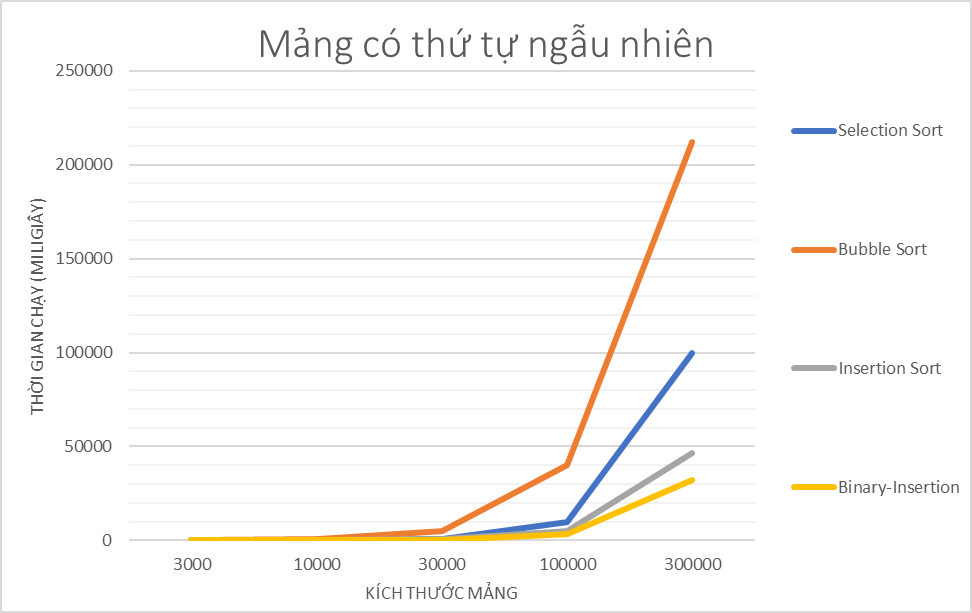
**1. Mảng được sắp xếp ngẫu nhiên:**



Với dữ liệu đầu vào là mảng một chiều có thứ tự ngẫu nhiên, ta nhận thấy:

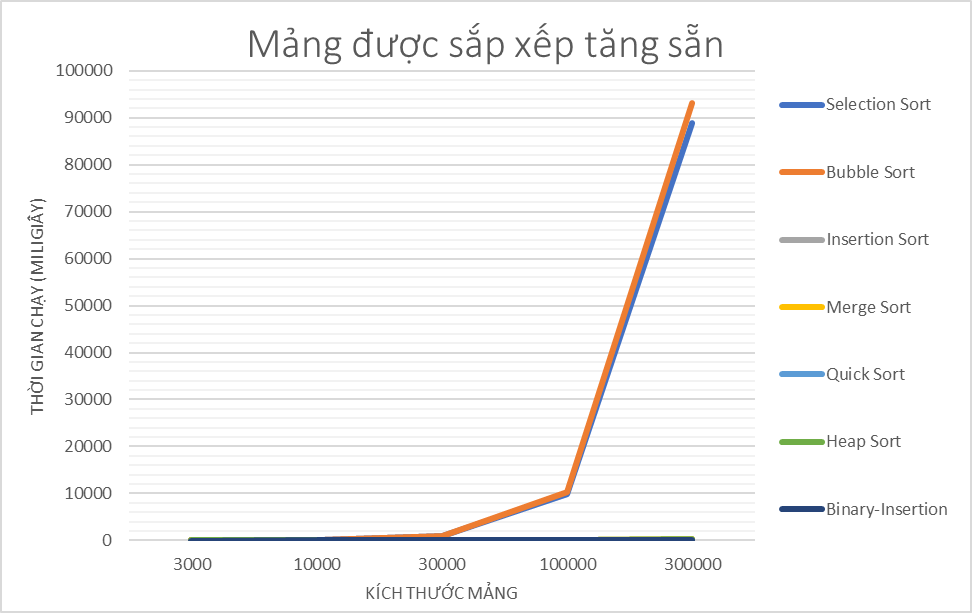


* + - Trong tất cả 7 thuật toán sắp xếp đã cài đặt, Quick Sort, Heap Sort và Merge Sort là ba thuật toán có thời gian chạy nhanh nhất. Điều này khá dễ hiểu, bởi vì độ phức tạp của ba thuật toán sắp xếp trên trong trường hợp mảng có thứ tự ngẫu nhiên này là O(*n.logn*).
    - Cả ba đều có thời gian chạy rất nhanh, thường dưới 0.3 giây. Trong đó, Quick Sort là thuật toán chạy nhanh nhất trong tất cả, với thời gian chạy khi mảng có 300 000 phần tử là 93 miligiây (ms), so sánh với Merge Sort và Heap Sort thì con số này là hơn 250 ms.
    - Mặc dù đều có độ phức tạp là O(*n.logn*), lý do khiến Quick Sort chạy nhanh hơn 2 thuật toán còn lại là vì vòng lặp bên trong Quick Sort (vòng lặp trong hàm phân đoạn – partition) đơn giản hơn và thường được cài đặt một cách có hiệu quả. Ngoài ra, dù được cài đặt bằng đệ quy theo phương pháp chia để trị, Quick Sort không yêu cầu thêm vùng nhớ như Merge Sort, và nó không thực hiện những phép hoán vị phần tử không cần thiết như Heap Sort. Tuy nhiên, xét về độ ổn định thì Merge Sort lại là thuật toán sắp xếp ổn định (tức các phần tử có cùng giá trị sau khi được sắp xếp sẽ không bị hoán đổi vị trí với nhau), trong khi Quick Sort thì không.



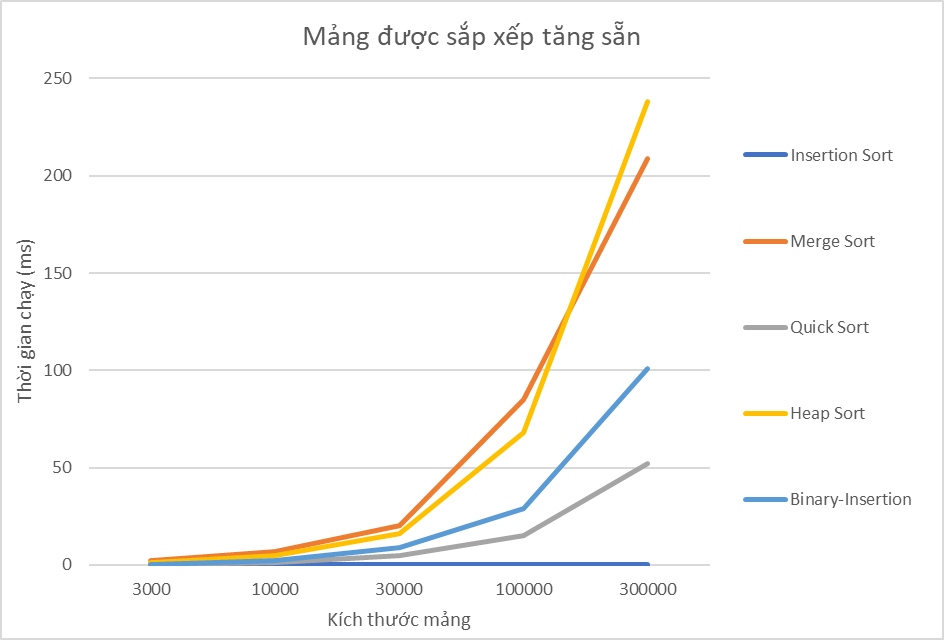
* + - Trong 4 thuật toán sắp xếp còn lại, bao gồm: Selection Sort, Bubble Sort, Insertion Sort và Binary-Insertion Sort. Cả 4 thuật toán này đều có độ phức tạp là O(*n2*) nên hiển nhiên thời gian chạy lâu hơn rất nhiều lần so với 3 thuật toán còn lại.
    - Trong 4 thuật toán với độ phức tạp O(*n2*) thì Binary-Insertion Sort có thời gian chạy ngắn nhất. Lý do là bởi vì Binary-Insertion Sort thực hiện ít phép so sánh hơn và ít phép hoán vị hơn so với những thuật toán sắp xếp khác. Nó chỉ đơn giản tìm vị trí thích hợp để đặt phần tử. Ngoài ra, ở Binary-Insertion Sort, ta cũng thấy được sự cải thiện về thời gian so với Insertion Sort, tiết kiệm hơn 30% thời gian chạy nhờ áp dụng tìm kiếm nhị phân thay vì tìm kiếm tuyến tính.
    - Ta cũng dễ dàng thấy được, Bubble Sort là thuật toán có thời gian chạy lâu nhất trong 4 thuật toán với độ phức tạp O(*n2*), lên đến hơn 210 giây (gần 4 phút) với kích thước mảng là 300 000. Lý do là bởi vì Bubble Sort thực hiện rất nhiều phép so sánh và rất nhiều phép hoán vị phần tử so với những thuật toán sắp xếp khác. Để làm rõ hơn vấn đề này, hãy lấy Selection Sort làm ví dụ. Thông qua cài đặt thuật toán, ta thấy được Selection Sort cũng thực hiện rất nhiều phép so sánh để tìm phần tử nhỏ nhất, tuy nhiên nó chỉ thực hiện hoán vị phần tử một lần sau khi tìm được phần tử nhỏ nhất đó, trong khi Bubble Sort thực hiện phép hoán vị khá thường xuyên. Do đó Selection Sort có thời gian chạy chỉ khoảng 100 giây, nhanh hơn Bubble Sort gấp đôi với cùng kích thước mảng.

**2. Mảng được sắp xếp tăng sẵn**



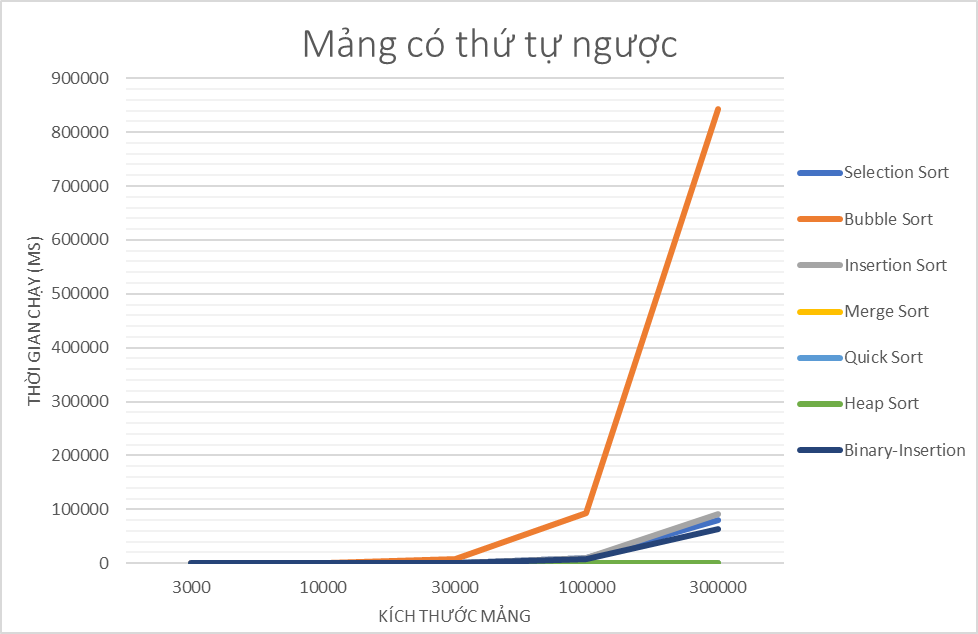
Với dữ liệu đầu vào là mảng một chiều có thứ tự ngẫu nhiên, ta nhận thấy:

* + - Bubble Sort và Selection Sort có thời gian chạy lâu nhất và gần như tương đương nhau. Và giữa hai thuật toán này thì Bubble Sort lại chạy lâu hơn, dẫn đến việc nó là thuật toán chạy chậm nhất trong 7 thuật toán.
    - Điều này có thể được lý giải như sau: mặc dù không thực hiện việc hoán vị phần tử, nhưng cả hai thuật toán đều thực hiện rất nhiều phép so sánh hai phần tử, dẫn đến thời gian chạy lâu hơn.

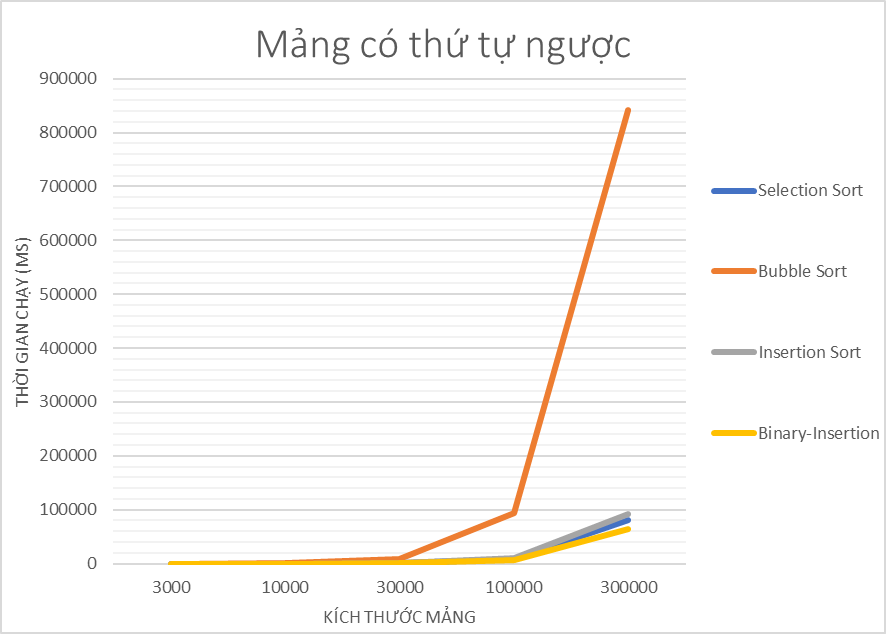


* + - Mặt khác, 5 thuật toán sắp xếp còn lại bao gồm: Insertion Sort, Merge Sort, Quick Sort, Heap Sort và Binary-Insertion Sort có thời gian chạy tương tương nhau và đều dưới 0.25 giây.
    - Khá bất ngờ vì Insertion Sort và Binary-Insertion Sort đều có độ phức tạp là O(*n2*) nhưng lại đứng trong nhóm này. Và Insertion Sort là thuật toán chạy nhanh nhất trong tất cả, với thời gian cho cả 5 kích thước mảng đều gần như tức thì, xấp xỉ 0 ms.
    - Lý do khiến Insertion Sort nhanh như vậy là bởi vì trong trường hợp mảng đã được sắp xếp tăng sẵn, vòng lặp giúp tìm vị trí chèn phù hợp tại mỗi lần lặp đều không được thực thi do không thỏa điều kiện. Do đó, hàm insertionSort chỉ đơn giản duyệt qua mảng này, làm cho độ phức tạp của thuật toán trong trường hợp này là O(*n*), khiến nó nhanh hơn hẳn các thuật toán sắp xếp khác.

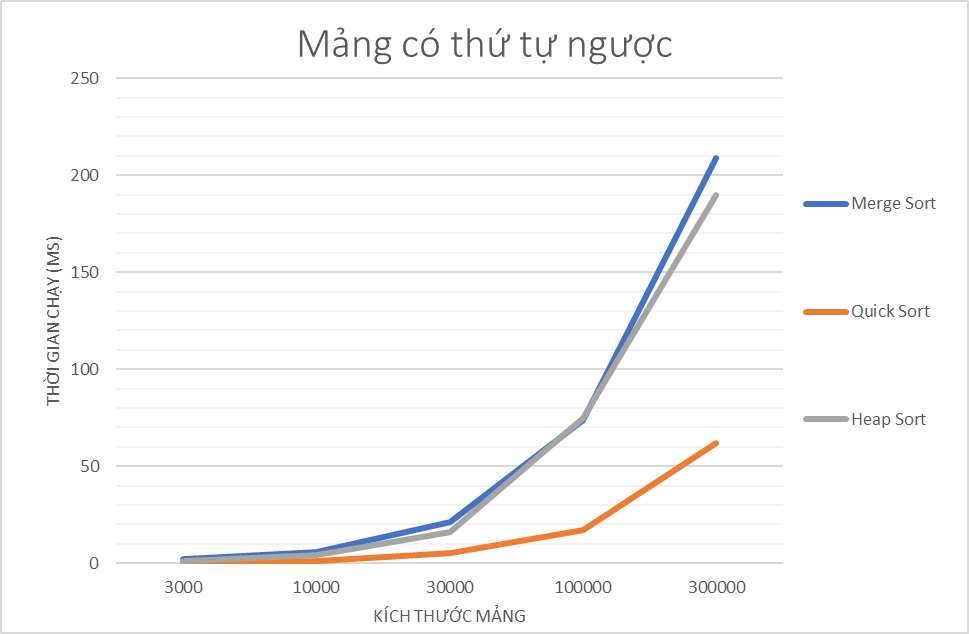
**3. Mảng có thứ tự ngược**



Với dữ liệu đầu vào là mảng có thứ tự ngược, tức các phần tử trong mảng được sắp xếp giảm dần, ta nhận thấy:

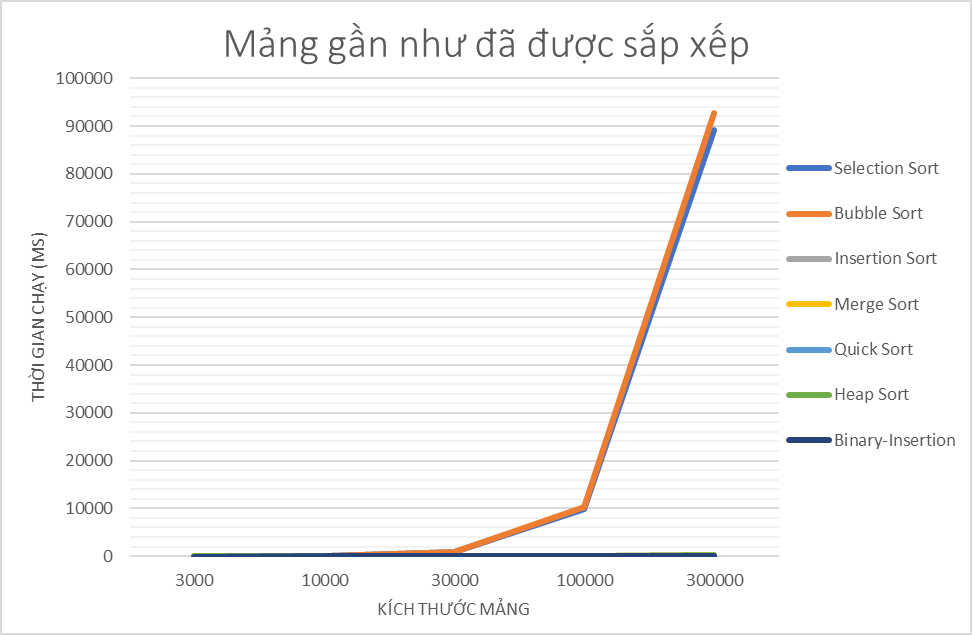


* + - Trước hết, ta xét 4 thuật toán sắp xếp với độ phức tạp O(*n2*). Ta thấy Bubble Sort là thuật toán chạy chậm nhất, trong trường hợp dữ liệu đầu vào là mảng có thứ tự ngược.
    - Lý do là bởi vì, Bubble Sort phải thực hiện rất nhiều phép hoán vị phần tử trong suốt quá trình chạy. Có thể nói tại mỗi lần lặp, thuật toán đều phải thực hiện phép hoán vị phần tử, dẫn đến thời gian chạy rất lâu, lên đến hơn 840 giây (14 phút) với kích thước mảng là 300 000.
    - Binary-Insertion Sort cũng là thuật toán chạy nhanh nhất trong 4 thuật toán trên nhờ áp dụng tìm kiếm nhị phân, khiến cho vòng lặp bên trong vòng lặp duyệt mảng chỉ có độ phức tạp là O(*logn*) thay vì O(*n*).

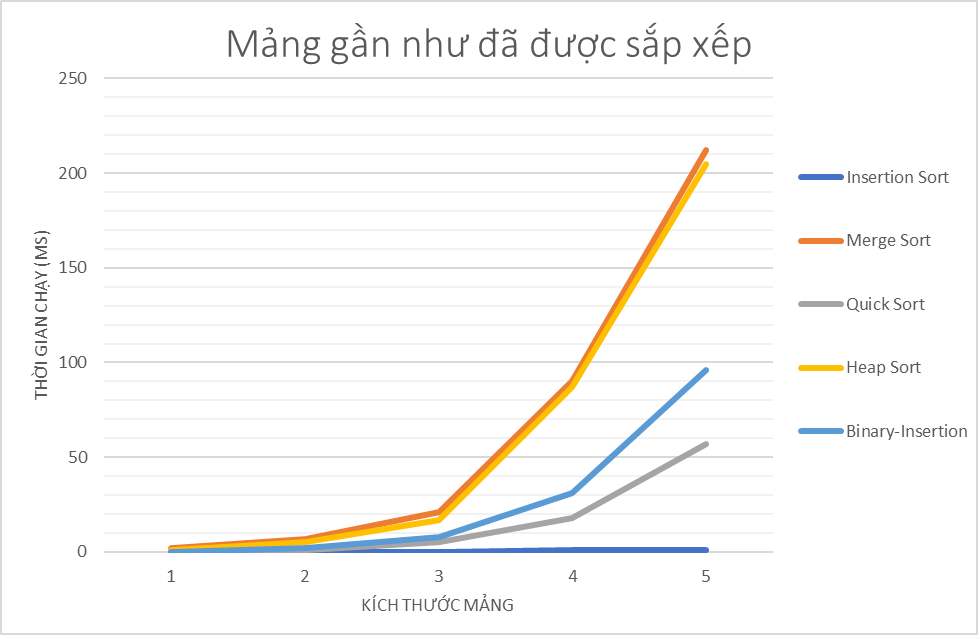


* + - Ta xét tiếp 3 thuật toán sắp xếp với độ phức tạp O(*n.logn*) còn lại. Ta thấy Quick Sort là thuật toán chạy nhanh nhất, với thời gian chạy chỉ hơn 60 ms với kích thước mảng là 300 000.
    - Điều này đã được lý giải trong trường hợp mảng có thứ tự ngẫu nhiên bên trên. Do vòng lặp bên trong hàm phân đoạn – partition đơn giản hơn và thường được cài đặt một cách có hiệu quả.
    - Ngoài ra, dù được cài đặt bằng đệ quy theo phương pháp chia để trị, Quick Sort không yêu cầu thêm vùng nhớ như Merge Sort, và nó không thực hiện những phép hoán vị phần tử không cần thiết như Heap Sort. Điều này dẫn đến thời gian chạy của Quick Sort trong trường hợp này rất ngắn.

**4. Mảng gần như được sắp xếp**



* Với dữ liệu đầu vào là mảng một chiều với các phần tử gần như được sắp xếp, ta nhận thấy:
* Tương tự như trường hợp mảng đầu vào đã được sắp xếp tăng sẵn, Bubble Sort và Selection Sort có thời gian chạy lâu nhất và gần như tương đương nhau. Và giữa hai thuật toán này thì Bubble Sort lại chạy lâu hơn, dẫn đến việc nó là thuật toán chạy chậm nhất trong cả 7 thuật toán.
* Điều này cũng đã được lý giải bên trên: mặc dù thực hiện rất ít phép hoán vị phần tử, cả hai thuật toán đều thực hiện rất nhiều phép so sánh hai phần tử, dẫn đến thời gian chạy lâu hơn.



* + - Mặt khác, 5 thuật toán sắp xếp còn lại bao gồm: Insertion Sort, Merge Sort, Quick Sort, Heap Sort và Binary-Insertion Sort có thời gian chạy tương tương nhau và đều dưới 0.25 giây. Và Insertion Sort là thuật toán chạy nhanh nhất trong tất cả, với thời gian cho cả 5 kích thước mảng đều gần như tức thì, xấp xỉ 0 ms.
    - Lý do khiến Insertion Sort nhanh như vậy là bởi vì trong trường hợp mảng gần như đã được sắp xếp tăng sẵn, vòng lặp giúp tìm vị trí chèn phù hợp tại mỗi lặp phải thực thi rất ít lần. Dù nó không khiến độ phức tạp của Insertion Sort trong trường hợp này chỉ còn O(*n2*) như trong trường hợp mảng được sắp xếp tăng sẵn, nhưng nó cũng khiến cho thời gian chạy giảm đi rất nhiều, thậm chí nhanh hơn cả Quick Sort.

**5. Nhận xét chung cho cả 7 thuật toán sắp xếp với 4 trạng thái dữ liệu đầu vào:**

* + - Nhìn chung, Bubble Sort là thuật toán chạy chậm nhất và Quick Sort là thuật toán chạy nhanh nhất với tất cả 4 trạng thái dữ liệu đầu vào.
    - Đương nhiên, Quick Sort không phải lúc nào cũng nhanh nhất trong tất cả, bởi vì trong trường hợp mảng được sắp xếp tăng sẵn và mảng gần như được sắp xếp, Insertion Sort mới là thuật toán chạy nhanh nhất. Mặc dù vậy, Quick Sort đảm bảo được thời gian chạy của nó gần như tương tương nhau mà không phụ thuộc vào trạng thái dữ liệu đầu vào, nên ta công nhận Quick Sort là nhanh nhất.
    - Trường hợp mảng được sắp xếp tăng sẵn và mảng gần như được sắp xếp cũng là trường hợp tốt nhất của Insertion Sort và Binary-Insertion Sort, khiến độ phức tạp thuật toán giảm xuống còn O(*n*).
    - Xét về độ ổn định trong thời gian chạy của các thuật toán:
    - Selection Sort, Merge Sort, Quick Sort, Heap Sort có thời gian chạy ổn định nhất. Với cả 4 trạng thái dữ liệu đầu vào, mỗi thuật toán đều có thời gian chạy không chênh lệch quá lớn, khoảng chênh lệnh rất nhỏ, chỉ tính bằng miligiây.
    - Bubble Sort, Insertion Sort và Binary-Insertion Sort có thời gian chạy không ổn định và phụ thuộc vào trạng thái của dữ liệu đầu vào. Điển hình như Insertion Sort và Binary-Insertion Sort, nếu chúng gặp trường hợp tốt nhất (mảng được sắp xếp tăng sẵn hoặc gần như được sắp xếp tăng) thì thời gian chạy gần như tức thì, các trường hợp khác thì thời gian chạy khá lâu. Hay Bubble Sort, nếu gặp trường hợp xấu nhất (mảng có thứ tự ngược) thì thời gian chạy lâu hơn nhiều lần.
    - Xét về độ ổn định của thuật toán sắp xếp:
    - Các thuật toán sắp xếp ổn định: Merge Sort, Bubble Sort, Insertion Sort và Binary-Insertion Sort. Các phần tử có cùng giá trị sau khi được sắp xếp sẽ không hoán đổi vị trí với nhau.
    - Các thuật toán sắp xếp không ổn định: Selection Sort, Quick Sort và Heap Sort. Trong một số trường hợp, các phần tử có cùng giá trị sau khi được sắp xếp sẽ hoán đổi vị trí với nhau.

**TÀI LIỆU THAM KHẢO**

1. Thuật toán sắp xếp Selection Sort minh họa code sử dụng C++

<https://nguyenvanhieu.vn/thuat-toan-sap-xep-selection-sort/>

2. Selection Sort - <https://www.geeksforgeeks.org/selection-sort/>

3. Insertion Sort - <https://www.geeksforgeeks.org/insertion-sort/>

4. Thuật toán Quick Sort – Sắp xếp nhanh

<https://nguyenvanhieu.vn/thuat-toan-sap-xep-quick-sort/>

5. Heap Sort - <https://www.geeksforgeeks.org/heap-sort/>

6. Heapsort Algorithm

<https://www.interviewcake.com/concept/java/heapsort>

7. Binary-Insertion Sort

<https://ducmanhphan.github.io/2019-05-24-Binary-Insertion-sort/>

8. Category: Stable Sorts - <https://en.wikipedia.org/wiki/Category:Stable_sorts>

9. Binary Insertion Sort and Complexity

<https://stackoverflow.com/questions/42564936/binary-insertion-sort-and-complexity>

10. Merge Sort Space and Time Complexity

<https://stackoverflow.com/questions/10342890/merge-sort-time-and-space-complexity>

11. Why does Quick Sort use O(log(n)) extra space?

<https://stackoverflow.com/questions/12573330/why-does-quicksort-use-ologn-extra-space>

12. Why is Quick Sort faster in average than others?

<https://stackoverflow.com/questions/4289024/why-is-quicksort-faster-in-average-than-others>