THUẬT TOÁN SẮP XẾP

Báo cáo giữa kỳ

PHAN LỘC SƠN

Cấu trúc Dữ liệu và Giải thuật
19CNTN

Khoa Công nghệ Thông tin
Đại học Khoa học Tự nhiên
Việt Nam
23 - 11 - 2020

Phần I
Phân Tích

Các thuật toán sẽ làm việc

1. Selection Sort
2. Insertion Sort
3. Binary-Insertion Sort
4. Bubble Sort
5. Shaker Sort
6. Shell Sort
7. Heap Sort
8. Merge Sort
9. Quick Sort
10. Counting Sort
11. Radix Sort
12. Flash Sort

1 Selection Sort

(a) Ý tưởng

Ở lượt thứ i, tìm phần tử nhỏ nhất trong mảng i -> n và đặt vào vị trí i, tìm phần tử nhỏ nhất bằng cách duyệt tuần tự!

(b) Giải thích thuật toán

- Ở lượt thứ i, mảng từ 0 $\rightarrow$ i đã được sắp xếp, các phần tử này luôn nhỏ hơn các phần tử chưa được sắp xếp bên phải.
- Như vậy chỉ cần đặt phần tử nhỏ nhất ở mảng chưa được sắp xếp (bên phải) rồi lặp lại, ta sẽ được mảng sắp xếp.

(c) Mã giả

```
function SelectionSort
  arr : array of items
  n   : length of arr

  for i = 0; i < n-1; i++ do:
    pos_min = i;
    for j = i + 1; j < n; j++ do
      if arr[j] < arr[min_idx] then
```

```
9           Cập nhật lại pos_min = j;
10        end for
11        swap(arr[i], arr[min_pos]);
12    end for
```

(d) Độ phức tạp và Không gian bộ nhớ sử

Độ phức tạp:

- Trường hợp tốt nhất: $O(n^2)$.
- Trường hợp xấu nhất: $O(n^2)$.
- Trung bình: $O(n^2)$.
  $\rightarrow$ Selection Sort luôn duyệt $\frac{n*(n+1)}{2}$ lần, vậy nên độ phức tạp của nó luôn là $O(n^2)$ trong mọi trường hợp.

Không gian bộ nhớ sử dụng:

- Mọi trường hợp: $O(1)$. $\rightarrow$ Selection Sort không sử dụng thêm không gian bộ nhớ khác trong quá trình thực thi.

2 Insertion Sort

(a) Ý tưởng

Ở lượt thứ i, mảng 1 -> i đã được sắp xếp, chèn phần tử thứ i + 1 vào mảng 1 -> i bằng cách so sánh tuần tự arr[i+1] và các phần tử trước nó, lặp lại tới khi i = n.

(b) Giải thích thuật toán

- Tại lượt thứ i, mảng từ 0 -> i - 1 đã được sắp xếp, ta đưa phần tử thứ j = i chèn vào mảng đã được sắp xếp bằng cách so sánh lần lượt arr[j] và các phần tử trước nó (nếu arr[j] gặp phần tử bé hơn nó thì dừng lại).
- Như vậy, sau mỗi lượt, số phần tử của mảng đã được sắp xếp tăng lên 1, lặp lại như thế n - 2 lần ta có mảng được sắp xếp!

(c) Mã giả

```
1  function InsertionSort
2    arr : array of items
3    n   : length of arr
4
5    for i = 1; i < n; ++i do:
6      Lưu tạm = arr[i];
7      j = i - 1;
8      while (j >= 0 và arr[j] > Lưu tạm) do:
9        arr[j + 1] <- arr[j];
10       --j;
11     end while
12     arr[j + 1] = tạm;
13   end for
```

(d) Độ phức tạp và Không gian bộ nhớ sử

Độ phức tạp:

- Trường hợp tốt nhất: $O(n)$.
- Trường hợp xấu nhất: $O(n^2)$.
- Trung bình: $O(n^2)$.

Không gian bộ nhớ sử dụng:

- Mọi trường hợp: $O(1)$.

3 Binary - Insertion Sort

(a) Ý tưởng

Ý tưởng giống như Insertion Sort, nhưng Binary-Insertion Sort tìm vị trí đúng của phần tử bằng Binary Search -> chèn thẳng vào đó.

(b) Giải thích thuật toán

- Như trình bày ở trên, Binary-InsertionSort có ý tưởng giống với InsertSort, ở lượt thứ i, mảng 0 -> i - 1 đã được sắp xếp.
- Sau đó, dùng BinarySearch để tìm ra vị trí đúng của arr[i] trong mảng 0 -> i - 1 (index).
- Dịch chuyển mảng [index + 1, i - 1] sang phải 1 chỉ số.
- Đưa giá trị arr[i] vào arr[index].
- Như vậy ta sẽ được 1 mảng mới được sắp xếp, có độ dài lớn hơn mảng cũ 1 đơn vị, lặp lại như thế n - 2 lần ta sẽ được mảng được sắp xếp!

(c) Mã giả

```
function BinarySearch
  arr   : array of items
  left  : left index of search-array
  right : right index of search-array
  x     : search value

  if right >= left then:        // Nếu mảng có >= 1 phần tử
    mid = (right + left) / 2;
    if arr[mid] = x then trả về vị trí mid;
    if arr[mid] < x then Tìm kiếm từ mid + 1 -> right
    if arr[mid] > x then Tìm kiếm từ left -> mid - 1
  end if

  //  Nếu mảng không còn phần tử nào nữa
  if x > a[left] then:
    Trả về left + 1;
  else:
    Trả về left;
  end if

function BinaryInsertionSort
```

```
22    arr    : array of items
23    n      : length of array
24
25    for i = 0; i < n - 1; ++i do:
26      Lưu tạm = arr[i];
27      j = i - 1;
28
29      vị trí = BinarySearch(arr, 0, j, tạm); // tìm kiếm vị trí đúng của
            Lưu tạm
30
31      // Dịch chuyển giá trị arr[vị trí], arr[vị trí +1],...arr[j]
32      đến vị trí [vị trí + 1], [vị trí + 2],... [j + 1]
33      while (j >= vitri) do:
34        arr[j + 1] = arr[j];
35        j--;
36      end while
37      // Đưa biến tạm lưu vào arr[vị trí]
38      arr[j+1] = Lưu tạm
39    end for
```

(d) Độ phức tạp và Không gian bộ nhớ sử

Độ phức tạp:

- Trường hợp tốt nhất: $O(nlog(n))$.
- Trường hợp xấu nhất: $O(n^2)$.
- Trung bình: $O(n^2)$.

Không gian bộ nhớ sử dụng:

- Mọi trường hợp: $O(1)$.

4 Bubble Sort

(a) Ý tưởng

Duyệt từ đầu tới cuối, đổi chỗ 2 phần tử liên tiếp nhau nếu số trước lớn hơn số sau, lặp lại như thế cho tới khi mảng được sắp xếp (lúc duyệt không có 2 phần tử nào đổi chỗ nữa).

(b) Giải thích thuật toán

- Xuất phát từ i = 0, bắt đầu so sánh arr[i] và arr[i + 1], nếu arr[i] lớn hơn thì đổi chỗ chúng (đẩy số lớn hơn ra sau)
- Tiếp tục công việc đẩy số lớn hơn ra sau, ta sẽ đẩy được số lớn nhất ra cuối cùng.
- Như vậy, sau mỗi lượt, ta sắp xếp được 1 phần tử (lớn nhất) ra sau, và làm giảm mảng chưa được sắp xếp xuống 1 phần tử.
- Vậy, ta lặp lại n - 2 lần thì ta sẽ được mảng được sắp xếp!

(c) Mã giả

```
function BubbleSort
  arr : array of items
  n   : length of arr

  có thay đổi = false;
  for i = 0; i < n - 1; ++i do:
    có thay đổi = false
     for  j = 0; j < n - i - 1; ++j do:
      // Nếu arr[j] lớn hơn phần tử ở ngay sau nó
        if arr[j] > arr[j + 1] then:
          swap(arr[j + 1], arr[j])
        có thay đổi = true;
        end if
      end for
    //Nếu có thay đổi, thoát vòng lặp (i)
    if not (có thay đổi) break the for loop
  end for
```

(d) Độ phức tạp và Không gian bộ nhớ sử

Độ phức tạp:

- Trường hợp tốt nhất: $O(n)$.
- Trường hợp xấu nhất: $O(n^2)$.
- Trung bình: $O(n^2)$.

Không gian bộ nhớ sử dụng:

- Mọi trường hợp: $O(1)$.

5 Shaker Sort

(a) Ý tưởng

Cơ bản giống như Bubble Sort, Bubble Sort sắp xếp từ đầu tới cuối, Shaker Sort sắp xếp từ đầu tới cuối rồi từ cuối về lại đầu, mỗi lượt nhận được 1 số lớn nhất và 1 số bé nhất, lặp lại cho tới khi mảng được sắp xếp.

(b) Giải thích thuật toán

- Giống với Bubble Sort, đầu tiên, ta dùng kỹ thuật của Bubble Sort đẩy số lớn nhất ra sau cuối, cập nhật lại vị trí cuối cùng của dãy chưa được sắp xếp.
- Tiếp theo, ta đẩy số nhỏ nhất trong mảng chưa được sắp xếp ra đầu dãy của nó (dãy chưa được sắp xếp) và cập nhật lại vị trí đầu tiên của dãy chưa được sắp xếp.
- Tiếp tục công việc trên, ta dần giảm số phần tử của mảng đi 2 mỗi lượt.
- Tới khi mảng chưa được sắp xếp không còn phần tử nào (vị trí sau $<$ vị trí trước) kết thúc thuật toán!

(c) Mã giả

```
function ShakerSort
  arr : array of items
  n   : length of arr

  đánh dấu trái = 0, đánh dấu phải = n - 1
  while đánh dấu trái nhỏ hơn đánh dấu phải do:
    pos_swapped = l;  // vị trí cuối cùng của quá trình sắp xếp
    // Sắp xếp bên phải
    for i = l; i < r; ++i do:
      //Nếu arr[j] lớn hơn phần tử ở ngay sau nó
      if arr[j] > arr[j + 1] then:
        swap(a[i], a[i + 1]);
        pos_swapped = i;
      end if
    end for

    r = pos_swapped;
    // Sắp xếp bên trái
    for i = r; i > l; --i do:
      // Nếu arr[j] bé hơn phần tử ở ngay trước nó
      if arr[j] < arr[j - 1] then:
        swap(a[i], a[i - 1]);
        pos_swapped = i;
      end if
    end for
    l = pos_swapped;
  end while
```

(d) Độ phức tạp và Không gian bộ nhớ sử

Độ phức tạp:

- Trường hợp tốt nhất: $O(n)$.
- Trường hợp xấu nhất: $O(n^2)$.
- Trung bình: $O(n^2)$.

Không gian bộ nhớ sử dụng:

- Mọi trường hợp: $O(1)$.

6 Shell Sort

(a) Ý tưởng

Bắt đầu với số k $<$ n, phân hoạch thành các tập a_{i+mk}, i = 0 -> k - 1, dùng Insert Sort sắp xếp các phần tử trong từng phân hoạch, lặp lại công việc trên với số k nhỏ hơn số k ở bước lặp trước, kết thúc việc sắp xếp sau khi thực hiện phép sắp xếp với k = 1.

(b) Giải thích thuật toán

- Có nhiều cách xây dựng thuật toán từ các công thức khác nhau, ở đây chúng ta dùng công thức interval = interval * 3 + 1

- Bắt đầu từ interval = 1, Xây dựng số interval như trên lớn nhất có thể mà không vượt qua n.
- Phân hoạch tập trên thành interval tập, các phần tử chung 1 tập thì có chung số dư với interval.
- Dùng InsertionSort để sắp xếp từng phân hoạch (InsertionSort là thuật toán rất tốt để sắp xếp trên 1 mảng nhỏ)
- Giảm interval = (interval - 1) / 3 và thực hiện lại các bước trên, cho tới khi interval = 0 thì kết thúc thuật toán
- Vì interval luôn luôn sẽ tới 1 lúc được nhận giá trị interval = 1 (theo cách xây dựng xuất phát từ 1) nên mảng luôn được sắp xếp theo InsertionSort thực sự (vì interval = 1), nên mảng chắc chắn được sắp xếp trước khi kết thúc thuật toán.
- Thuật toán này là 1 nâng cấp cho InsertionSort vì làm giảm sự bước đổi chỗ của các số nhỏ mà nằm gần cuối mảng!

(c) Mã giả

```
// Giải thuật này sử dụng Knuth's Formula với interval = interval * 3 +
   1, interval là số lớn nhất không vượt quá n/3.
function ShellSort
  arr : array of items
  n   : length of arr

  số tập chia = 1;
  while (số tập chia < n / 3) do:
    số tập chia = số tập chia * 3 + 1;
  end while

  while(số tập chia > 0) do:
    // Sắp xếp các tập phân hoạch
    for i = số tập chia; i < n; ++i do:
      Lưu tạm = arr[i]
      j = i - số tập chia
      // Bắt đầu thao tác chèn, dịch chuyển các phần tử sang phải
      while j >= số tập chia && Lưu tạm < arr[j] do
        arr[j + số tập chia] = arr[j];
        j -= số tập chia;
      end while
      // Sau khi dịch chuyển các phần tử sang phải, chèn giá trị vào
      arr[j + số tập chia] = lưu tạm
    end for
    // cập nhật lại số tập chia
    số tập chia = (số tập chia -1)/3
  end while
```

(d) Độ phức tạp và Không gian bộ nhớ sử

Độ phức tạp:

- Trường hợp tốt nhất: $O(n)$.
- Trường hợp xấu nhất: Tuỳ thuộc vào cách chọn dãy "số tập chia", với Knuth's Formula, là $O(n^{\frac{3}{2}})$.

- Trung bình: $O(n^{\frac{3}{2}})$.

Không gian bộ nhớ sử dụng:

- Mọi trường hợp: $O(1)$.

7 Heap Sort

(a) Ý tưởng

Môt cấu trúc Heap (max) là cấu trúc có arr[i] >= arr[2i] và arr[i] > arr[2i + 1], như vậy, arr[1] luôn lớn nhất, Heap Sort xây dựng cấu trúc Heap bằng cách đổi chỗ arr[i] bằng số lớn nhất của 1 trong 3 số trên. sau khi xây dựng xong Heap, vì arr[1] là số lớn nhất, đổi arr[1] ra sau mảng và lăp lại việc xây dựng cấu trúc Heap, ta dần có được mảng sắp xếp.

(b) Giải thích thuật toán

- Một cấu trúc Heap (max) là 1 cấu trúc có arr[i] > arr[2i] và arr[i] > arr[2i + 1] nếu chúng tồn tại,
- Vì thế, để xây dựng cấu trúc Heap, ta đi từ nửa mảng về đầu mảng, nếu phát hiện 1 cặp 3 phần tử arr[i], arr[2i], arr[2i + 1] không thoả, ta sẽ cập nhật lại giá trị của arr[i] thành max 3 số, tiếp tục xét tiếp bộ liên quan tới arr[i] (là arr[j = $\frac{i}{2}$] và arr[i ± 1]), j = i cho tới khi bộ arr[i], arr[2i], arr[2i + 1] đúng điều kiện trên.
- Sau khi xây dựng được cấu trúc Heap (max), phần tử đầu tiên luôn là phần tử lớn nhất mảng, lúc này ta đổi vị trí phần tử đầu và phần tử cuối mảng → mảng chưa được sắp xếp giảm đi 1!
- Lặp lại công việc (xây dựng lại Heap và đổi vị trí phần tử đầu và cuối của mảng), mỗi lượt ta giảm số phần tử của mảng chưa được sắp xếp đi 1, lặp lại n - 1 lần ta sẽ được mảng được sắp xếp!

(c) Mã giả

```
1  // Hàm xây dựng cấu trúc Heap
2  function Sift
3    arr : array of items
4    l : start position of array
5    r : end position of array
6    n   : length of arr
7
8    x = arr[l];
9    i = l, j = 2 * i
10   while j <= r do:
11     if j < r then:
12       if arr[j] < arr[j + 1] then:
13         ++j // arr[j] là số lớn hơn
14     if x >= arr[j] then:
15       break the while loop
16     arr[i] = arr[j]
17     i = j
18     j = 2 * i
19   end while
20   arr[i] = x
```

```
21
22  // Hàm sắp xếp
23  function HeapSort
24    arr : array of items
25    n   : length of arr
26
27    l = n / 2;
28
29    // Xây dựng mảng a có dạng heap
30    while l >= 0 do:
31      Sift(arr, l, n - 1);
32      --l;
33    end while
34
35    // Chuyển số lớn nhất về cuối mảng và sort lại mảng 0->t-1
36    t = n - 1
37    while t > 0 do:
38
39      swap(arr[0], arr[t])
40      --t;
41      Sift(arr, 0, t)
42    end while
```

(d) Độ phức tạp và Không gian bộ nhớ sử

Độ phức tạp:

- Trường hợp tốt nhất: $O(nlog(n))$.
- Trường hợp xấu nhất: $O(nlog(n))$.
- Trung bình: $O(nlog(n))$.
  $\rightarrow$ Độ phức tạp của HeapSort ổn định xung quanh $O(nlog(n))$.

Không gian bộ nhớ sử dụng:

- Mọi trường hợp: $O(1)$.

8 Merge Sort

(a) Ý tưởng

Giả sử ta đã có 2 mảng đã được sắp xếp, ta chỉ cần duyệt 2 mảng từ đầu tới cuối, liên tục so sánh 2 phần tử ở đầu 2 mảng và di chuyển vào mảng chung, ta sẽ được mảng sắp xếp. Merge Sort sắp xếp mảng có 2^i (i = 1) phần tử -> nối lại thành mảng 2^{i+1} phần tử -> lặp lại cho tới khi $2^i > n$.

(b) Giải thích thuật toán

- Hàm merge ghép 2 mảng đã được sắp xếp lại với nhau bằng cách duyệt tuần tự các phần tử của 2 mảng từ trái sang phải.
- MergeSort liên tục tách mảng làm 2 phần, đến khi mảng không có phần tử nào thì dừng lại và dùng merge (vì lúc này các mảng đều chỉ có 1 phần tử nên mảng đã trở thành mảng tăng dần), liên tục ghép 2 mảng được sắp xếp lại với nhau, cuối cùng ta được mảng cần được sắp xếp.

(c) Mã giả

```
// Hàm ghép mảng con về mảng chính
function merge
  arr : array of items
  l : start position of array
  r : end position of array
  m   : mid, break the array at this position

  n1 = m - l + 1
  n2 = r - m
  i = 0, j = 0, k = 0
  L = new int[n1]
  R = new int[n2]

  // Chép vào mảng phụ
  for i = 0; i < n1; ++i do:
    L[i] = arr[l + i]
  end for

  for j = 0; j < n2; ++j
    R[j] = arr[m + j + 1]
  end for

  // Chép từ mảng phụ về mảng chính
  i = j = k = 0;
  while i < n1 && j < n2 do:
    if L[i] <= R[j] then
      arr[l + k] = L[i]
      k++
      i++
    else
      arr[l + k] = R[j]
      k++
      j++
    end if
  end while

  // Chép các phần tử còn sót lại
  while i < n1 do:
    arr[l + k] = L[i]
    k++
    i++
  end while

  while j < n2 do:
    arr[l + k] = R[j];
    k++;
    j++;
  end while

function MergeSort
  arr : array of items
  l   : start position of array
  r   : end position of array
```

```
54
55    if (l < r)
56      m = (l + r)/2
57      // Chia thành 2 mảng con
58      mergesort(arr, l, m);
59      mergesort(arr, m + 1, r)
60      // Ghép lại
61      merge(arr, l, m, r)
62    end if
```

(d) Độ phức tạp và Không gian bộ nhớ sử dụng

Độ phức tạp:

- Trường hợp tốt nhất: $O(nlog(n))$.
- Trường hợp xấu nhất: $O(nlog(n))$.
- Trung bình: $O(nlog(n))$.

Không gian bộ nhớ sử dụng:

- Trường hợp xấu nhất: $O(n)$.

9 Quick Sort

(a) Ý tưởng

Chia mảng thành 2 phần bằng 1 khoá pivot (tuỳ cách chọn), bé hơn pivot di chuyển sang trái pivot, lớn hơn di chuyển sang phải. Lặp lại công việc trên với 2 mảng đã chia cho tới khi mảng được sắp xếp.

(b) Giải thích thuật toán

- Hàm partition chia mảng đã cho thành 2 phần, 1 phần bé hơn khoá pivot nằm phía trái pivot, 1 phần nằm bên phải khoá pivot nằm ở bên phải khoá (2 mảng này chưa được sắp xếp) và trả về vị trí của pivot.
- Sau khi partition mảng xong, ta nhận được 2 mảng [l, pivot_index - 1] và [pivot_index + 1, r], ta chỉ cần sắp xếp lại 2 mảng con này.
- Như vậy, chỉ cần liên tục chia mảng con thành các mảng con nhỏ hơn bằng QuickSort, ta sẽ được mảng được sắp xếp.

(c) Mã giả

```
1  // Xử lý từng mảng con
2  function partition
3    arr : array of items
4    l : start position of array
5    r : end position of array
6
7    // Chọn khoá pivot là phần tử bên trái mảng
8    pivot = arr[l]
9    i = l, j = r + 1
10   do:
```

```
    // Duyệt từ bên trái và bên phải để tìm 2 phần tử cần đổi vị trí
       với nhau (bé hơn pivot < pivot < lớn hơn pivot)
    do i++; while arr[i] < pivot
    do j--; while arr[j] > pivot
    swap(arr[i], arr[j])
  while i < j
    swap(arr[i], arr[j])
    swap(arr[j], arr[l]);
  return j


function QuickSort
  arr : array of items
  l : start position of array
  r : end position of array
  if l < r then:
    // Chia ra thành 2 mảng con theo và sort từng mảng
    s = partition(arr, l, r)
    quicksort(arr, l, s - 1)
    quicksort(arr, s + 1, r)
  end if
```

(d) Độ phức tạp và Không gian bộ nhớ sử dụng

Độ phức tạp:

- Trường hợp tốt nhất: $O(nlog(n))$.
- Trường hợp xấu nhất: $O(n^2)$.
- Trung bình: $O(nlog(n))$.

Không gian bộ nhớ sử dụng:

- Trường hợp xấu nhất: $O(n)$.

10 Counting Sort

(a) Ý tưởng

Đếm số lần xuất hiện của các phần tử và lưu vào mảng tĩnh -> Tìm vị trí cuối cùng mà giá trị arr[i] được đặt vào và dặt vào nó -> được mảng đã sắp xếp.

(b) Giải thích thuật toán

- CountingSort đếm số lần xuất hiện của các giá trị trong mảng, lưu vào mảng đếm, sau đó, Tìm vị trí cuối cùng mà giá trị arr[i] được đặt vào và dặt vào nó, ta sẽ được mảng được sắp xếp.
- Ở đây, CountingSort sử dụng kỹ thuật: đối với 1 giá trị bất kỳ cộng số lần xuất hiện của giá trị đó và số lần xuất hiện của các giá trị nhỏ hơn nó, sẽ được vị trí cuối cùng mà nó đặt đặt vào mảng (lúc mảng đã sắp xếp xong) để xác định vị trí cần để đặt vào.

(c) Mã giả

```
function CountingSort
  arr : array of items
  n : length of array

  // Tìm khoảng giá trị của mảng và chuẩn bị sắp xếp
  Min = min(arr, n)
  Max = max(arr, n)
  N = max - min + 1;
  f = new int[Max - Min + 1];
  memset(f, 0, n)

  // Đếm số lần xuất hiện của  arr[i] trong mảng và lưu vào f[arr[i] -
     Min]
  for i = 0; i < n; ++i do:
    ++f[arr[i] - Min]
  end for

  // Đến đây, f[i] có nghĩa là vị trí cuối cùng của giá trị i + Min
     trong mảng nếu nó được sắp xếp
  for i = 1; i < N; ++i
    f[i] += f[i - 1];
  end for

  int* brr = new int[n];
  // Sắp xếp!
  for i = n - 1; i >= 0; --i
    // Đưa arr[i] vào vị trí cuối cùng nó xuất hiện, trừ giá trị f[i]
       (vị trí cuối cùng) đi 1.
    brr[f[arr[i] - Min] - 1] = arr[i];
    --f[arr[i] - Min];
  end for

  // Copy vao lai arr
  memcpy(arr, brr, n)
```

(d) Độ phức tạp và Không gian bộ nhớ sử dụng

Độ phức tạp:

- Mọi trường hợp: $O(n + k)$ với $k = Max - Min$.

Không gian bộ nhớ sử dụng:

- Trường hợp xấu nhất: $O(n + k)$.

11 Radix Sort

(a) Ý tưởng

Sắp xếp mảng bằng chữ số, bắt đầu từ hàng đơn vị, ở mỗi lượt, chỉ sắp xếp theo chữ số ở hàng đang xét -> lặp lại cho tới khi đạt tới chữ số hàng cao nhất, do tính ổn định của thuật toán ở mỗi lượt, cuối cùng ta được mảng được sắp xếp.

(b) Giải thích thuật toán

- Dùng CountingSort để sắp xếp các số theo chữ số của hàng chỉ định.
- Bắt đầu từ hàng nhỏ nhất là hàng đơn vị đến hàng lớn nhất, mỗi lần sắp xếp, vì CountingSort giữ được tính ổn định khi sắp xếp và các phép sắp xếp sau là phép sắp xếp hàng cao hơn phép sắp xếp trước (... > trăm > chục > đơn vị) nên cuối cùng, thuật toán hướng đến việc sắp xếp dãy số theo kiểu số nào có "hàng số lớn"lớn hơn thì được trước các số có "hàng số lớn"nhỏ hơn.
- Ví dụ 5678, 1452, 230 sẽ được sắp xếp theo hàng số lớn nhất là hàng nghìn -> hàng trăm -> hàng chục -> hàng đơn vị. Vì thế ta được 5678 > 1452 > 230.

(c) Mã giả

```
function sort
  arr : array of items
  k : current digit
  n : length of array

  // f lưu số lần xuất hiện của các phần tử (chữ số thứ k)
  // b lưu mảng đã sắp xếp theo chữ số thứ k
  f[10] = 0
  b[10] = 0

  for i = 0; i <= n; ++i do:
    f[digit(arr[i], k)]++
  end for

  for i = 1; i <= 9; ++i do:
    f[i] += f[i - 1]
  end for

  for i = n; i >= 1; i-- do:
    j = digit(arr[i], k)
    b[f[j]] = arr[i]
    f[j]--
  end for

  for i = 0; i <= 9; ++i do:
    arr[i] = b[i]
  end for

function RadixSort
  arr : array of items
  n : length of array

  max = max(arr, n);
  d = log(max) / log(10) + 1;

  for k = 0; k < d; ++k do:
    sort(arr, k, n);
  end for
```

(d) Độ phức tạp và Không gian bộ nhớ sử dụng

Độ phức tạp:

- Độ phức tạp xung quanh: $O(n+m)$, với m là số chữ số tối đa của phần tử trong mảng

Không gian bộ nhớ sử dụng:

- Trường hợp xấu nhất: $O(n+m)$.

12 Flash Sort

(a) Ý tưởng

Flash Sort tính toán chia tập dữ liệu thành các phân lớp, sau đó duyệt từ trái sang phải đổi chỗ lần lượt các phần tử để đưa nó về đúng phân lớp của mình. Cuối cùng, tác giả quyết định dùng Insertion Sort để sắp xếp các phân lớp và ta được mảng được sắp xếp.

(b) Giải thích thuật toán

- FlashSort gồm 3 bước: Phân loại các phần tử theo m lớp, phân hoạch các phần tử về lớp của nó và Sắp xếp các phân lớp.
- Giai đoạn 1: Phân loại các phần tử theo m lớp.
  Theo thực nghiệm, với $m = 0.43n$ cho kết quả tốt nhất
  Tạo 1 mảng L có m phần tử dùng để lưu số phần tử của mỗi phân lớp.
  Với phần tử arr[i], phân lớp mà arr[i] thuộc về là:

$$k = \left[\frac{(m-1)(arr[i]-min)}{max-arr[i]}\right] + 1$$

  Như vậy, phân lớp ở thứ tự nhỏ hơn thì mỗi phần tử của nó nhỏ hơn phần tử thuộc phân lớp có thứ tự lớn hơn
  Tính mảng L, và tìm vị trí mà phần tử cuối của phân lớp k bằng kỹ thuật của CountingSort:

$$L[i] += L[i-1]$$

- Giai đoạn 2:
  Duyệt qua mảng, nếu tìm thấy phần tử chưa đúng phân lớp: Đổi chỗ nó với phần tử có vị trí L[k] (k là phân lớp của phần tử chưa đúng) -> phân lớp được phần tử đó thành công, giảm L[k] đi 1 đơn vị, và phân lớp phần tử vừa bị đổi chỗ.
  Lặp lại công việc trên cho tới khi tất cả các phần tử đều được phân lớp thành công.
- Giai đoạn 3:
  Trong mỗi phân lớp, dùng InsertionSort để sắp xếp lại phân lớp đó, InsertionSort là thuật toán rất tốt để sắp xếp lại mảng nhỏ.
- Cuối cùng ta có mảng đã được sắp xếp!

(c) Mã giả

```
function Flash
  arr : array of items
  n : length of array

  // Tìm khoảng giá trị của mảng
  Min = min(arr, n)
  Max = max(arr, n)

```

```
  // Tạo các phân lớp, số phân lớp m = 0.43n làm Flash Sort chạy với
     thời gian tốt nhất
  m = 0.43n

  // mảng l lưu số phần tử của các phân lớp
  l = new int[m]
  memset(l, 0, m)
  for i = 0; i < n; ++i do:
    k = (m - 1) * (arr[i] - Min) / (Max - Min)
    ++l[k]
  end for

  l[0] = 0
  for i = 1; i < m; ++i do:
    l[i] += l[i - 1]
  end for

  count = 0, j = 0, k = m - 1
  while count < n do:
    while j > l[k] do:
      ++j
      k = (m - 1) * (arr[i] - Min) / (Max - Min)
    end while
  temp = a[j]

  while j <= l[k] do:
    k = (m - 1) * (arr[i] - Min) / (Max - Min)
    swap(temp, arr[l[k]])
    --l[k]
    ++count
  end while

  // Insertion Sort cho các phân lớp
  for i = 1; i < m; ++i do:
    for p = l[i] - 1; p > l[i - 1]; --p do:
      if arr[p] > arr[p+1] then:
        temp = a[p]
        t = p
        while (temp > a[t + 1])
          a[t] = a[t + 1]
          ++t
        end while
        arr[t] = temp
    end for
  end for
```

(d) Độ phức tạp và Không gian bộ nhớ sử dụng

Độ phức tạp:

- Trường hợp xấu nhất: $O(n^2)$.
- Trường hợp trung bình: $O(n)$.
- Trường hợp tốt nhất: $O(n)$.

Không gian bộ nhớ sử dụng:

- Trường hợp xấu nhất: $O(n)$.

Phần II
Thực nghiệm các giải thuật

Các giải thuật được thực hiện trên các bộ dữ liệu: Ngẫu nhiên, tăng dần, giảm dần, và ngẫu nhiên có hướng tăng dần. Ở mỗi bộ dữ liệu, ta cho số lượng phần tử của mảng lần lượt là: 3000, 10000,30000 ,100000 ,300000. Sau khi thực nghiệm, ta được các kết quả (Đơn vị: nano giây (ns)):

1 SelectionSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	5720700	5228200	5827000	7969100
10000	54715800	59352600	41786700	73584100
30000	458406700	870013100	482763300	471196400
100000	6135026300	6434962500	4381936300	3745167700
300000	56579970600	37907453300	46764247600	31616122000

Bảng số liệu cụ thể thời gian thực nghiệm đối với Selection Sort
→ Phân tích:

- Nhìn vào bảng số liệu, với cùng 1 size, SelectionSort không tỏ ra ưu việt đối với loại dữ liệu nào. Có thể giải thích là SelectionSort luôn duyệt số lần bằng nhau ở mọi loại dữ liệu mà có cùng độ lớn, vì vậy, sự chênh lệch ở đây không lớn và chỉ phụ thuộc vào sự phân bố của dữ liệu đầu vào.
- Ngoài ra, đây là một thuật toán dễ cài đặt, có tính ổn định cao (ổn định ở $O(n^2)$):

  Lặp $i = 0 \rightarrow n-2$, ở mỗi lần duyệt 1 giá trị i, ta cần duyệt $n-i$ lần, tổng là $(n-1)+(n-2)+...+1 = \frac{n(n-1)}{2} \rightarrow O(n^2)$

2 InsertionSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	2264900	4700	4039400	25000
10000	22072900	14300	35652100	251900
30000	141934700	40300	264255100	240700
100000	1565845000	147800	2898466500	320100
300000	13092687600	382600	26154503400	794100

Bảng số liệu cụ thể thời gian thực nghiệm đối với Insertion Sort
→ Phân tích:

- Với cùng 1 ArraySize, ta dễ thấy bộ dữ liệu (về độ nhanh) SortedData > NearlySortedData > RadomData > ReverseData (nhanh hơn rất nhiều!).
- Có thể giải thích như sau, InsertSord là thuật toán di chuyển các phần tử ở vị trí cũ của nó → vị trí đúng so với các phần tử khác trong mảng đã được sắp xếp khi đang thực thi, như vậy:

  Với 1 mảng đã sort sẵn, ta không cần di chuyển các phần tử đi đâu cả, và vì thế SortedData cho thời gian nhanh nhất.

  NearlySortedData cho một bộ dữ liệu gần như tăng dần → ít phần tử cần được di chuyển về vị trí của nó → InsertionSort nhanh trong trường hợp này.

RandomData là trường hợp ngẫu nhiên, vì thế nó chậm hơn trường hợp tốt hơn (NearlySorted-Data) và chậm hơn trường hợp xấu nhất (ReverseData).

ReserveData, đây là trường hợp xấu nhất, ta cần nhiều bước di chuyển các phần tử về vị trí của nó (vì các phần tử ở cách xa vị trí đúng của nó nhất).

- Đánh giá thuật toán:

  Trường hợp tốt nhất: SortedData: Mỗi lần duyệt thực hiện 1 lần $\rightarrow O(n)$.

  Trường hợp xấu nhất: ReserveData: Mỗi lần duyệt $i : 1 \rightarrow n-1$ cần duyệt $i-1$ lần, như vậy, cần: $0+1+2+...+(n-2) = \frac{(n-2)*(n-1)}{2} \rightarrow O(n^2)$

  Trường hợp trung bình: RandomData: Mỗi phần tử nằm cách vị trí của nó $\frac{1}{2}(1+i)$, suy ra, trường hợp này cần $\frac{1}{2}(1+2+...+(n-1)) = \frac{1}{4}n(n-1) \rightarrow O(n^2)$

3 BinaryInsertionSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	2057500	94500	2833900	109800
10000	12915500	447800	25100900	406100
30000	112358600	1299800	217686600	1396200
100000	1315585100	3868200	2421905100	5510200
300000	10859202400	13884700	27029597100	11401800

Bảng số liệu cụ thể thời gian thực nghiệm đối với BinaryInsertion Sort
$\rightarrow$ Phân tích:

- Vì cơ bản dựa trên InsertionSort, Binary-InsertionSort có độ nhanh khi sắp xếp các bộ dữ liệu SortedData > NearlySortedData > RadomData > ReverseData.

- Có thể thấy Binary-InsertionSort có ưu thế với InsertionSort ở RandomData và ReverseData và có vẻ chậm hơn ở SortedData và NearlySortedData. Có thể giải thích, vì ở SortedData và NearlySortedData, Insertion Sort không cần làm gì nhiều, cơ bản các phần tử đã ở gần vị trí của nó rồi nên không cần tìm kiếm nhị phân để tìm vị trí của nó làm mất thêm thời gian. Còn ở RandomData, ReverseData thì Binary-InsertionSort nhỉnh hơn vì lúc này dữ liệu phân bố rời rạc hoặc xấu, Binary Search gây được hiệu quả search của mình khi vị trí cần tìm ở xa (hoặc không biết đang ở đâu, không phải ở gần như NearlySortedData).

- Đánh giá thuật toán:

  Trường hợp tốt nhất: SortedData: Mỗi lần duyệt thực hiện 1 lần tìm kiếm (trả về vị trí cuối mảng đã được sắp xếp) $\rightarrow O(log(n))$, thực hiện trên tất cả phần tử $\rightarrow O(nlog(n))$.

  Trường hợp xấu nhất: ReserveData: Mỗi lần duyệt phải swap từ phần tử về vị trí của nó, tốn $2*(n+(n-2)+...) \in O(n^2)$

  Trường hợp trung bình: RandomData: Mỗi phần tử nằm cách vị trí của nó $\frac{1}{2}(1+i)$, vẫn cần phải swap số lần: $\frac{1}{2}(1+2+...+(n-1)) = \frac{1}{4}n(n-1) \rightarrow O(n^2)$

- Nhận xét: thuật toán tăng hiệu năng InsertionSort, nhưng với đánh giá chủ quan, thuật toán này dở hơn InsertionSort ở SortedData và NearlySortedData, và cũng dở hơn các thuật toán khác ở RandomData và ReverseData.

4 BubbleSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	13489200	4500	14261400	4097200
10000	152992200	7300	108604500	40655300
30000	1692928300	21600	850776100	386364200
100000	17816415800	95700	9126831000	1243267300
300000	1.57653E+11	300500	83468398100	6594016900

Bảng số liệu cụ thể thời gian thực nghiệm đối với Bubble Sort
→ Phân tích:

- Vì cơ bản dựa trên InsertionSort, Binary-InsertionSort có độ nhanh khi sắp xếp các bộ dữ liệu SortedData > NearlySortedData > RadomData > ReverseData.
- Cụ thể:

  SortedData: rất nhanh, vì trong thuật toán có kiểm tra swapped xem mảng đã được sắp xếp chưa và kết thúc luôn!

  NearlySortedData: chậm hơn rất nhiều so với ShortedData, chứng tỏ thuật toán này rất chậm.

  RandomData và ReverseData: Chạy rất chậm.
- Đánh giá thuật toán:

  Trường hợp tốt nhất: SortedData: Duyệt qua 1 lần mảng, không thấy cặp phần tử nào được đổi chỗ → dừng thuật toán → độ phức tạp $Ø(n)$.

  Trường hợp xấu nhất: ReserveData: Duyệt i từ 0 đến n - 2, mỗi lần duyệt tốn n - i - 1 bước, vậy tổng cộng tốn: $n(n-1) - (1+2+...+n-2) - n \in O(n^2)$.

  Trường hợp trung bình: Các phần tử để về đúng vị trí cần swap ít nhất $i/2$ lần (trường hợp trung bình), vậy cần ít nhất $2 \times \frac{1}{2}(1+2+...+\frac{n}{2}) \in O(n^2)$.
- Nhận xét: Thuật toán rất chậm vì phải duyệt tất cả cho tới khi mảng được sắp xếp.

5 ShakerSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	9117200	3800	9897300	75400
10000	134861700	7300	92557200	187800
30000	1135764500	26400	838623800	961000
100000	11687014200	100500	9397646400	1493500
300000	1.04862E+11	588600	86585035800	5443300

Bảng số liệu cụ thể thời gian thực nghiệm đối với Shaker Sort
→ Phân tích:

- Là phiên bản nâng cấp của Bubble Sort, cải thiện tốc độ rất đáng kể
- Có thể giải thích là vì sau mỗi lần Sort, ta sắp xếp lại chiều ngược và làm giảm độ dài mảng từ cả 2 đầu, không cần phải duyệt lại từ đầu như bubble sort.
- Đánh giá thuật toán:

  Trường hợp tốt nhất: SortedData: Giải thích giống như BubbleSort, $O(n)$.

  Trường hợp trung bình và xấu nhất: Các phần tử để về đúng vị trí cần swap ít nhất $i/2$ lần (trường hợp trung bình, các lần swap là swap với số bên cạnh), vậy cần ít nhất $2 \times \frac{1}{2}(1+2+...+\frac{n}{2}) \in O(n^2)$.

- Nhận xét: Cải tiến tăng tốc độ đáng kể.

6 ShellSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	204700	21100	41100	83800
10000	813000	94500	147900	222300
30000	3472800	274300	506600	1284600
100000	11601300	1110900	2033200	2173800
300000	34680400	3806100	5442300	4214900

Bảng số liệu cụ thể thời gian thực nghiệm đối với Shell Sort
→ Phân tích:

- Là thuật toán lợi dụng tốc độ của InsertionSort ở các tập dữ liệu nhỏ
- Phân hoạch thành các phân lớp lớn, dùng InsertionSort sắp xếp các phân lớp đó, rồi lại tổng hợp và phân hoạch thành các phân lớp nhỏ hơn, lợi dụng InsertionSort khiến ShellSort trở thành 1 thuật toán sắp xếp tương đối nhanh.
- Đánh giá thuật toán:

  Trường hợp tốt nhất: $O(n)$.

  Trường hợp trung bình: $O(n^{\frac{3}{2}})$

  Trường hợp xấu nhất: Tuỳ thuộc vào cách chọn dãy "số tập chia", với Knuth's Formula, là $O(n^{\frac{3}{2}})$.
- Nhận xét: Tốc độ nhanh.

7 HeapSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	263200	221200	218100	210300
10000	1363500	1206300	735500	742300
30000	3446000	1919200	2143200	2361600
100000	13819300	9813300	9288100	8815400
300000	35219200	18532700	19446200	19296000

Bảng số liệu cụ thể thời gian thực nghiệm đối với Heap Sort
→ Phân tích:

- Các bộ dữ liệu có thời gian sắp xếp gần tương đương như nhau.
- Lợi dụng cấu trúc Heap để xây dựng từ từ các phần tử từ lớn tới bé
- Thời gian để tạo cấu trúc Heap ban đầu: $O(nlog(n))$.
- Sau đó, mỗi lần đổi số lớn nhất ra sau mảng, để xây dựng lại cấu trúc Heap chỉ cần $O(log(n))$. Lặp lại n lần, chính là $O(nlog(n))$.
- Đánh giá thuật toán:

  Mọi trường hợp: $O(nlog(n))$.
- Nhận xét: Nhanh và rất ổn định, không cần dùng thêm bộ nhớ.

8 MergeSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	1925300	1348400	1510500	696700
10000	5918300	2690800	2510200	3564500
30000	9927700	10097200	9561700	9774800
100000	41005000	22707400	21885900	21137600
300000	91449000	65761300	86607500	81739100

Bảng số liệu cụ thể thời gian thực nghiệm đối với Merge Sort
→ Phân tích:

- Thuật toán nhanh, lợi dụng việc tách mảng thành 2 mảng con, sắp xếp chúng và nối lại.
- Như vậy, có $O(log(n))$ lần tách mảng, khi nối mảng n phần tử thì cần $O(n)$ lần, như vậy độ phức tạp của MergeSort là $O(nlog(n))$ với mọi trường hợp.
- Đánh giá thuật toán:

  Mọi trường hợp: $O(nlog(n))$.
- Nhận xét: Nhanh và rất ổn định, cần dùng thêm bộ nhớ (lúc tạo mảng tạm để nối 2 mảng con lại, cần $O(n)$).

9 QuickSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	316600	192300	207900	192500
10000	930800	838000	776900	631700
30000	3110000	1741200	2005400	2362300
100000	12127300	5843300	5738900	7005800
300000	34364400	18476800	23608000	18161300

Bảng số liệu cụ thể thời gian thực nghiệm đối với Quick Sort
→ Phân tích:

- Là 1 trong những thuật toán nhanh nhất hiện nay, nhìn vào bảng, ta thấy tất cả bộ dữ liệu đều cần rất ít thời gian để sắp xếp.
- QuickSort chọn khoá pivot, chia mảng thành 2 mảng con (lớn hơn pivot và nhỏ hơn pivot), và tiếp tục QuickSort trên 2 mảng con tạo thành.
- Đánh giá thuật toán:

  Mọi trường hợp: Việc chia mảng liên tiếp thành 2 mảng con, trung bình tốn $O(log(n))$, và việc duyệt các phần tử để tìm 2 phần tử sai vị trí qua pivot (để đổi chỗ), tốn trung bình $O(n)$, như vậy độ phức tạp của QuickSort là $O(nlog(n))$.

10 CountingSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	64200	33900	33300	33300
10000	164700	145500	131800	104200
30000	420300	280000	268200	313100
100000	2016900	1740700	1284900	1355800
300000	6188100	3763700	3485900	4722100

Bảng số liệu cụ thể thời gian thực nghiệm đối với Counting Sort
→ Phân tích:

- Nhanh đối với bộ dữ liệu có khoảng giá trị nhỏ, đối với các bộ dữ liệu có khoảng giá trị lớn, dùng Counting Sort để sắp xếp là 1 thảm hoạ.
- Vì Size là cố định và nhỏ, nên thời gian sắp xếp của các bộ dữ liệu cùng kích thước chênh lệch không lớn và đều rất nhanh.
- Đánh giá thuật toán:

  Mọi trường hợp: Counting sort chỉ dùng 4 vòng for, 2 vòng for duyệt qua mảng (n phần tử), 2 vòng for duyệt qua k (mảng được tạo ra có độ rộng bằng khoảng giá trị của mảng), nên độ phức tạp của thuật toán là $O(n + k)$.
- Đánh giá: với khoảng dữ liệu nhỏ, Counting Sort chạy tuyến tính!

11 RadixSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	251700	218900	228100	300200
10000	792300	530900	531700	744600
30000	2434000	2334700	2866900	2030600
100000	8532600	6747500	8846100	8458500
300000	29651300	28846500	27340500	30638500

Bảng số liệu cụ thể thời gian thực nghiệm đối với Radix Sort
→ Phân tích:

- Nhanh và có tính ổn định. Là thuật toán dựa vào sự sắp xếp các chữ số thập bằng bằng cách sử dụng Counting Sort để sắp xếp mảng.
- Thời gian sắp xếp của các bộ dữ liệu khác nhau cùng kích thước là tương đối gần nhau và nhanh.
- Đánh giá thuật toán:

  Mọi trường hợp: RadixSort sử dụng Counting Sort để sắp xếp từng hàng thập phân, như vậy, độ phức tạp là $O(n + 10) = O(n)$ (vì khoảng giá trị của chữ số thập phân chỉ từ 0 đến 9), sắp xếp k lần (k là số chữ số của số lớn nhất trong mảng), nên độ phức tạp của RadixSort là $O(nk)$
- Đánh giá: Thuật toán chạy nhanh và có tính ổn định, ngoài ra, thời gian gần như nhau với tất cả các bộ dữ liệu.

12 FlashSort

Size (n)	RandomData	SortedData	ReverseData	NearlySortedData
3000	215700	46500	44900	106400
10000	990900	182200	164300	418700
30000	4201800	688100	581900	1242800
100000	13318100	2230000	2025600	2739000
300000	47364000	7386300	6766000	7657400

Bảng số liệu cụ thể thời gian thực nghiệm đối với Flash Sort
→ Phân tích:

- Tác giả phân hoạch mảng thành m (m chọn trước) phân lớp và tìm các đưa tất cả các phàn tử về đúng phân lớp của nó, sau đó, trong từng phân lớp, dùng InsertionSort (hiệu quả với mảng dữ liệu nhỏ) để sắp xếp trong từng phân lớp đó.
- Thuật toán rất hiệu quả, thời gian sắp xếp của 4 bộ dữ liệu đều nhỏ. Ngoài ra, đối với SortedData và NearlySortedData, vì các phần tử gần như đã nằm trong phân lớp đúng của nó khi sắp xếp ít phải xử lý các phần tử đó, thời gian chương trình chạy nhanh hơn RandomData và ReverseData.
- Đánh giá thuật toán:

  Trường hợp tốt nhất: SortedData: các phần tử đã đúng phần lớp -> duyệt qua 1 lần để biết tất cả các phần tử đều đúng phân lớp -> InsertSort cũng duyệt qua mỗi phần tử 1 lần -> $O(n)$

  Trường hợp xấu nhất: Chỉ ra 1 trường hợp: phần tử đầu là bé nhất, phần tử cuối là lớn nhất, các phần tử ở giữa đầu và cuối được xếp theo thứ tự giảm dần, khác nhau và cách nhau 1 đoạn cực nhỏ so với max - min, như vậy, n - 2 phần tử này nằm trong 1 phân lớp, việc phân lớp xảy ra trên $O(n)$, việc sắp xếp InsertionSort diễn ra trên $O((n-2)^2)$, như vậy, trường hợp xấu nhất có độ phức tạp là $O(n^2)$.

  Trường hợp trung bình: $O(n)$, nghe thầy Phương bảo thế chứ em chưa có thời gian tìm hiểu.
- Đánh giá: Thuật toán áp dụng nhiều ưu điểm của các thuật toán khác, là 1 trong những thuật toán sắp xếp nhanh nhất.

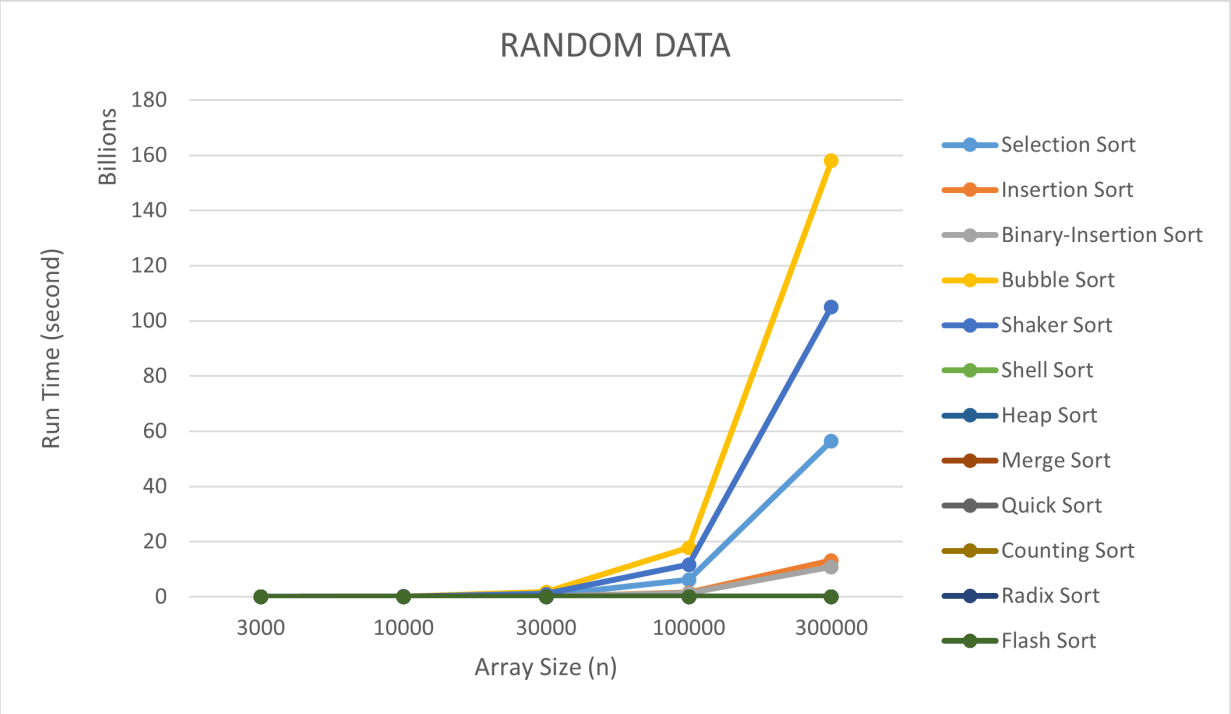
So sánh các giải thuật

1 Mảng dữ liệu ngẫu nhiên

Thời gian được tính theo nanosecond:

Size (n)	3000	10000	30000	100000	300000
Selection Sort	5720700	54715800	458406700	6135026300	56579970600
Insertion Sort	2264900	22072900	141934700	1565845000	13092687600
Binary-Insertion Sort	2057500	12915500	112358600	1315585100	10859202400
Bubble Sort	13489200	152992200	1692928300	17816415800	1.58E+11
Shaker Sort	9117200	134861700	1135764500	11687014200	1.05E+11
Shell Sort	204700	813000	3472800	11601300	34680400
Heap Sort	263200	1363500	3446000	13819300	35219200
Merge Sort	251700	792300	2434000	8532600	29651300
Quick Sort	316600	930800	3110000	12127300	34364400
Counting Sort	64200	164700	420300	2016900	6188100
Radix Sort	5720700	54715800	458406700	6135026300	56579970600
Flash Sort	215700	990900	4201800	13318100	47364000

Thời gian các thuật toán sắp xếp thành công mảng với trường hợp Mảng dữ liệu ngẫu nhiên.
Từ đó ta có biểu đồ sau:



→ Phân tích: Đối với bộ dữ liệu ngẫu nhiên.

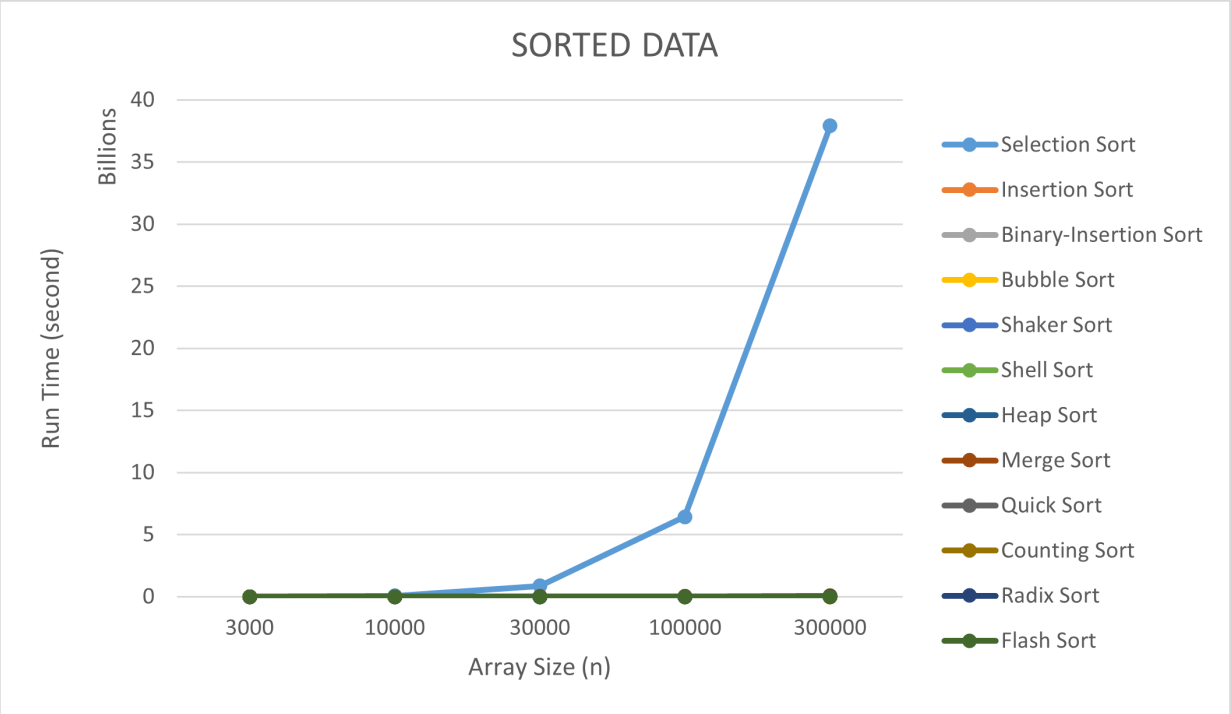
- Nhanh: ShellSort, Heap Sort, Merge Sort, QuickSort, Counting Sort, Radix Sort, Flash Sort.
- Trung bình: Selection Sort, Binary-Insertion Sort.
- Chậm: Bubble Sort, Shaker Sort.
- Nhanh nhất: Counting Sort: phân bố dữ liệu ở khoảng nhỏ, Counting Sort hiệu quả với tốc độ gần bằng tuyến tính.
- Nhanh nhất: Bubble Sort duyệt quá nhiều nên chậm nhất

2 Mảng dữ liệu tăng dần

Thời gian được tính theo nanosecond:

Size (n)	3000	10000	30000	100000	300000
Selection Sort	5228200	59352600	870013100	6434962500	37907453300
Insertion Sort	4700	14300	40300	147800	382600
Binary-Insertion Sort	94500	447800	1299800	3868200	13884700
Bubble Sort	4500	7300	21600	95700	300500
Shaker Sort	3800	7300	26400	100500	588600
Shell Sort	21100	94500	274300	1110900	3806100
Heap Sort	221200	1206300	1919200	9813300	18532700
Merge Sort	1348400	2690800	10097200	22707400	65761300
Quick Sort	192300	838000	1741200	5843300	18476800
Counting Sort	33900	145500	280000	1740700	3763700
Radix Sort	218900	530900	2334700	6747500	28846500
Flash Sort	46500	182200	688100	2230000	7386300

Thời gian các thuật toán sắp xếp thành công mảng với trường hợp Mảng dữ liệu tăng dần.
Từ đó ta có biểu đồ sau:



→ Phân tích: Đối với bộ dữ liệu tăng dần.

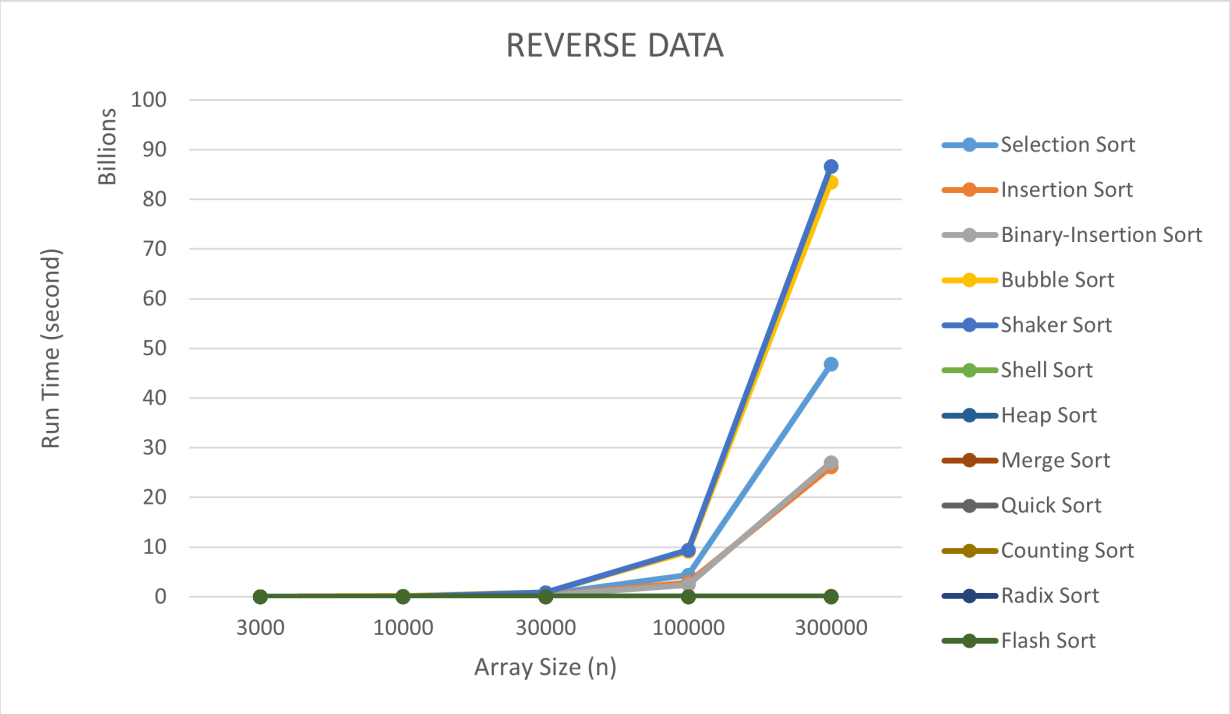
- Nhanh: Tất cả các thuật toán trừ Selection Sort.
- Chậm: Selection Sort.
- Selection Sort luôn phải duyệt theo độ phức tạp $O(n^2)$, Bubble Sort, Shaker Sort vì có biến kiểm tra mảng đã được sắp xếp chưa nên thời gian chạy nhanh.

3 Mảng dữ liệu giảm dần

Thời gian được tính theo nanosecond:

Size (n)	3000	10000	30000	100000	300000
Selection Sort	5827000	41786700	482763300	4381936300	46764247600
Insertion Sort	4039400	35652100	264255100	2898466500	26154503400
Binary-Insertion Sort	2833900	25100900	217686600	2421905100	27029597100
Bubble Sort	14261400	108604500	850776100	9126831000	83468398100
Shaker Sort	9897300	92557200	838623800	9397646400	86585035800
Shell Sort	41100	147900	506600	2033200	5442300
Heap Sort	218100	735500	2143200	9288100	19446200
Merge Sort	1510500	2510200	9561700	21885900	86607500
Quick Sort	207900	776900	2005400	5738900	23608000
Counting Sort	33300	131800	268200	1284900	3485900
Radix Sort	228100	531700	2866900	8846100	27340500
Flash Sort	44900	164300	581900	2025600	6766000

Thời gian các thuật toán sắp xếp thành công mảng với trường hợp Mảng dữ liệu giảm dần.
Từ đó ta có biểu đồ sau:



→ Phân tích: Đối với bộ dữ liệu giảm dần.

- Nhanh: Flash Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort.
- Trung bình: Selection Sort, Insertion Sort, Binary-Insertion Sort.
- Chậm: Selection Sort, Bubble Sort, Shaker Sort.
- Nhanh nhất: Counting Sort, vì khoảng giá trị các phần tử trong mảng nhỏ nên Counting Sort tiệm cận tuyến tính, ngoài ra, Flash Sort cho thời gian chạy rất nhanh cũng cho thấy lợi ích của việc phân hoạch mảng thành các mảng con nhỏ và dùng Insertion Sort.
- Chậm nhất: Selection Sort, Bubble Sort cho thấy sự chậm chạm khi dữ liệu không phải là dữ liệu được sort sẵn.

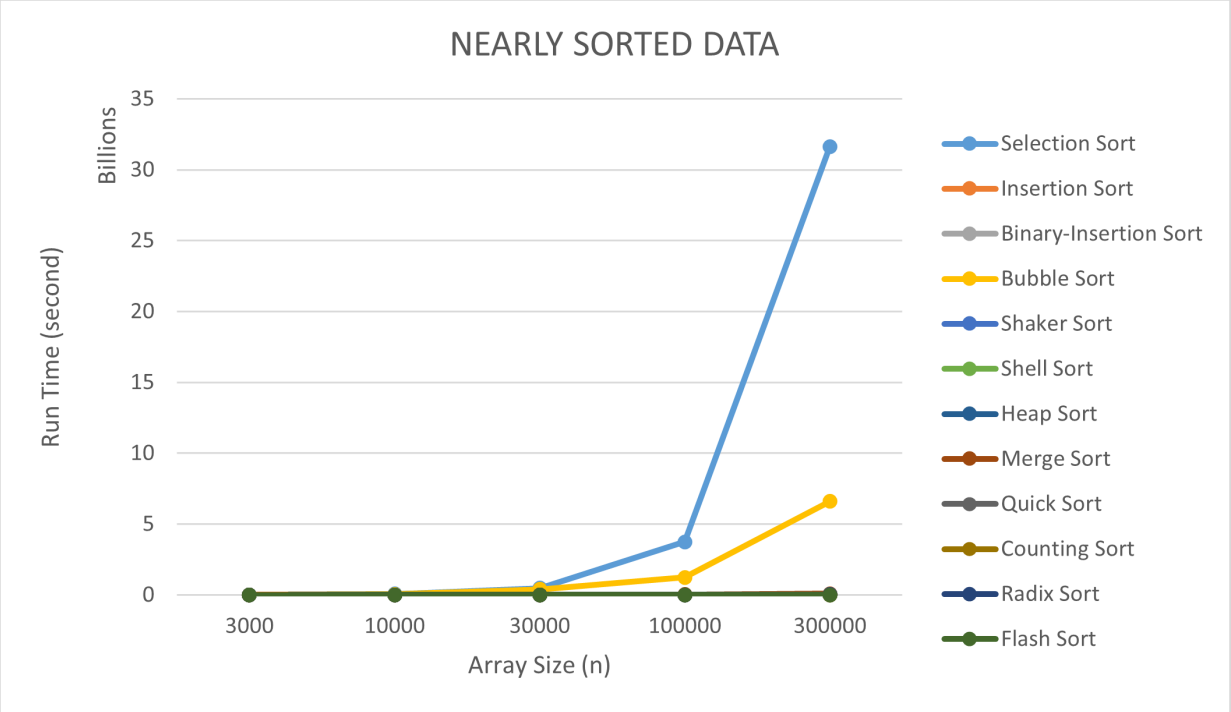
4 Mảng dữ liệu ngẫu nhiên có xu hướng tăng dần

Thời gian được tính theo nanosecond:

Size (n)	3000	10000	30000	100000	300000
Selection Sort	7969100	73584100	471196400	3745167700	31616122000
Insertion Sort	25000	251900	240700	320100	794100
Binary-Insertion Sort	109800	406100	1396200	5510200	11401800
Bubble Sort	4097200	40655300	386364200	1243267300	6594016900
Shaker Sort	75400	187800	961000	1493500	5443300
Shell Sort	83800	222300	1284600	2173800	4214900
Heap Sort	210300	742300	2361600	8815400	19296000
Merge Sort	696700	3564500	9774800	21137600	81739100
Quick Sort	192500	631700	2362300	7005800	18161300
Counting Sort	33300	104200	313100	1355800	4722100
Radix Sort	300200	744600	2030600	8458500	30638500
Flash Sort	106400	418700	1242800	2739000	7657400

Thời gian các thuật toán sắp xếp thành công mảng với trường hợp Mảng dữ liệu ngẫu nhiên có xu hướng tăng dần.

Từ đó ta có biểu đồ sau:



→ Phân tích: Đối với bộ dữ liệu ngẫu nhiên có xu hướng tăng dần.

- Nhanh: Tất cả thuật toán trừ Selection Sort và Bubble Sort.
- Trung bình: ShakerSort.
- Chậm: Selection Sort và Bubble Sort.
- Ở các bộ dữ liệu mà các phần tử có xu hướng tăng dần, đa số các thuật toán đều sắp xếp được nhanh mảng.
- Ở cả 4 bộ dữ liệu, Selection Sort luôn là thuật toán có tốc độ sắp xếp chậm.

Kết luận

- Một số thuật toán sắp xếp nhanh: ShellSort, HeapSort, MergeSort, QuickSort, RadixSort, FlashSort.
- Một số thuật toán có nhanh và có thời gian trung bình khá ổn là: HeapSort, MergeSort, QuickSort, RadixSort, FlashSort.
- CountingSort dùng để sắp xếp mảng có khoảng nhỏ rất tốt, nhưng dùng để sắp xếp mảng có khoảng giá trị lớn thì là tai nạn.
- Insertiong Sort rất nhanh (nhanh hơn cả QuickSort) khi sắp xếp mảng có ít phần tử.
- BubbleSort, ShakerSort rất chậm, trong đa số trường hợp (trừ trường hợp SortedData thì BubbleSort nhận biết ngay).