Các thuật toán sắp xếp và tìm kiếm

Mục tiêu

Hiểu, cài đặt, và phân tích độ phức tạp của một số thuật toán sắp xếp và tìm kiếm.

Bài toán sắp xếp

Input: Mảng A[1..n] gồm n phần tử (A[1], A[2], …, A[n]).

Output: Một hoán vị A' của mảng A để $A'[1] \leq A'[2] \leq \ldots \leq A'[n]$.

Một số thuật toán sắp xếp

Sắp xếp chọn (Selection Sort)

Sắp xếp chèn (Insertion Sort)

Sắp xếp trộn (Merge Sort)

Sắp xếp nhanh (Quick Sort)

Bài toán tìm kiếm

Input: Mảng A[1..n] gồm n phần tử (A[1], A[2], …, A[n]) và phần tử x.

Output: vị trí của phần tử x trong mảng A (trả về -1 nếu không tìm thấy).

Một số thuật toán tìm kiếm

Tìm tuyến tính (Linear Search)

Tìm nhị phân (Binary Search)

Mã giả (Pseudo-Code)

Trong bài này, ta sẽ sử dụng một ngôn ngữ tựa Pascal dùng để mô tả thuật toán.

Các phép toán

Gán : ←

So sánh : =, >, <, >=, <=, !=

Các cấu trúc

Cấu trúc chọn

if … **then** … [**else** …]

Cấu trúc lặp

while … **do**

do … **while**

for … **to/downto** … **do**

Các lệnh khác

Trả giá trị về: **return** <giá trị/biểu thức>

Gọi hàm, thủ tục: <Tên hàm/thủ tục> (<danh sách đối số>)

I. Sắp xếp

1.1. Sắp xếp chọn (Selection Sort)

Ý tưởng thuật toán

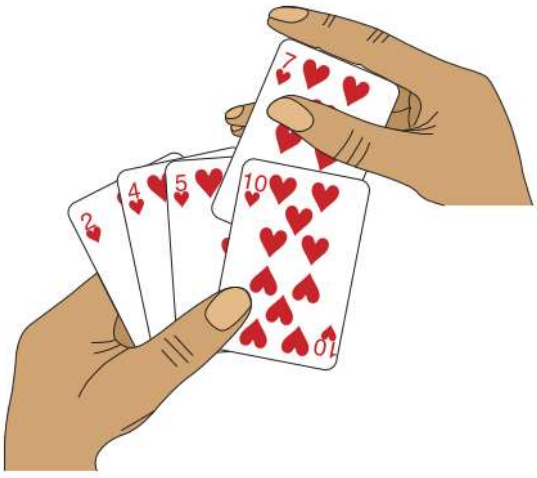
Tại bước thứ i, tìm vị trí k của phần tử nhỏ nhất trong đoạn A[i..n] và hoán vị A[k] với A[i].

```
for i ← 1 to n - 1 do
    // Tìm vị trí k của phần tử nhỏ nhất trong mảng A[i..n].
    k ← i
    for j ← i + 1 to n do
        if (A[j] < A[k]) then
            k ← j
    // Hoán vị A[k] và A[i]
    Exchange(A[k], A[i])
```

1.2. Sắp xếp chèn (Insertion Sort)

Thuật toán này dựa trên ý tưởng sắp bài.

- Bước 1. Úp các lá bài ở trên bàn, trên tay không có lá bài nào.
- Bước 2. Lấy lần lượt từng lá bài trên bàn và chèn vào vị trí thích hợp trên tay. Để tìm vị trí thích hợp, so sánh lá bài mới lấy lên từ bàn với các lá bài đã ở trên tay theo thứ tự từ phải sang trái.
- Bước 3. Lặp lại B2 cho đến khi trên bàn không còn lá bài nào.



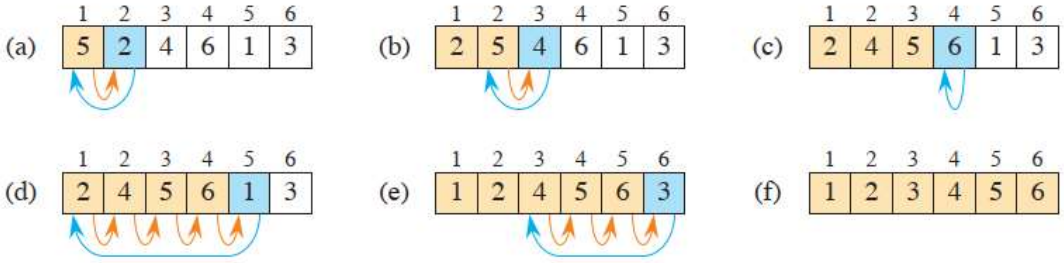
Thao tác sắp bài

Ý tưởng thuật toán

Tại bước thứ i, chèn phần tử key = A[i] vào đoạn A[1..i-1] đã sắp xếp tăng (của mảng A[1..n]) sao cho đoạn A[1..i] cũng sắp xếp tăng.

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Hình sau minh họa từng bước thuật toán với mảng A[1..6] = {5, 2, 4, 6, 1, 3}.



Phân tích độ phức tạp (thời gian thực hiện) của thuật toán xắp sếp chèn

Giả sử t_i là số lần mà điều kiện của vòng lặp while ở dòng 5 được kiểm tra với mỗi giá trị $i = 2, 3, ..., n$. Khi thoát khỏi vòng lặp **while** hay **for** thì điều kiện kiểm tra của vòng lặp thực hiện nhiều hơn các lệnh trong thân vòng lặp một lần. Giả sử mỗi lệnh mất một thời gian c_i để thực hiện. Ta có bảng phân tích như sau:

INSERTION-SORT(A, n)		*cost*	*times*
1	**for** $i = 2$ **to** n	c_1	n
2	$key = A[i]$	c_2	$n-1$
3	// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4	$j = i-1$	c_4	$n-1$
5	**while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6	$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7	$j = j-1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8	$A[j+1] = key$	c_8	$n-1$

Thời gian thực hiện thuật toán được tính bằng tổng thời gian thực hiện tất cả các lệnh. Ta tính được thời gian thực hiện T(n) như sau:

$$T(n) = nc_1 + (n-1)(c_2 + c_4 + c_8) + \left(\sum_{i=2}^{n} t_i\right)c_5 + \left(\sum_{i=2}^{n}(t_i - 1)\right)(c_6 + c_7)$$

Xét một input với kích cỡ cho trước, thời gian thực hiện của thuật toán phụ thuộc vào input này. Ví dụ: với thuật toán InsertionSort ở trên, **trường hợp tốt nhất là khi mảng đã được sắp xếp tăng dần**. Với mỗi $i = 2, 3, ..., n$ thì a[j] ≤ *key* (dòng 5) khi j có giá trị khởi đầu là $i - 1$, do đó, $t_i = 1$, với mọi $i = 2, 3, ..., n$.

$$T(n) = nc_1 + (n-1)(c_2 + c_4 + c_8) + (n-1)c_5$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Khi mảng đã được sắp xếp giảm dần thuật toán phải so sánh a[i] với toàn bộ các phần tử của mảng con a[1.. $i - 1$], do đó, t_i = i với mọi $i = 2, 3, ..., n$.

$$T(n) = nc_1 + (n-1)(c_2 + c_4 + c_8) + \left(\sum_{i=2}^{n} i\right)c_5 + \left(\sum_{i=2}^{n}(i-1)\right)(c_6 + c_7)$$

$$T(n) = nc_1 + (n-1)(c_2 + c_4 + c_8) + \left(\frac{n(n+1)}{2} - 1\right)c_5 + \left(\frac{n(n-1)}{2}\right)(c_6 + c_7)$$

$$T(n) = \left(\frac{c_5 + c_6 + c_7}{2}\right)n^2 + \left(\frac{2(c_1 + c_2 + c_4 + c_8) + c_5 - c_6 - c_7}{2}\right)n - (c_2 + c_4 + c_5 + c_8)$$

Như vậy, thời gian thực hiện trong trường hợp xấu nhất của thuật toán là một hàm bậc 2 theo n. Thông thường, ta hay quan tâm đến trường hợp thực hiện xấu nhất của thuật toán vì khi đó ta sẽ xác định được chặn trên về độ phức tạp (thời gian chạy) của thuật toán cho bất kỳ input nào. Nghĩa là ta biết rằng nó chắc chắn không thể chậm hơn. Mặt khác, trường hợp xấu nhất là trường hợp thường gặp trong thực tế (ví dụ tìm một phần tử không có trong danh sách sẽ dẫn đến trường hợp xấu nhất và điều này rất thường gặp). Cuối cùng, trường hợp trung bình thường cũng có độ phức tạp giống như trường hợp xấu nhất.

Xét **trường hợp trung bình**, mảng ban đầu là ngẫu nhiên và gồm n số, ta cần xác định mất bao lâu cho việc tìm được vị trí của a[i] để chèn nó vào mảng con a[1.. $i - 1$]. Tính trung bình, sẽ có một nửa số phần tử của mảng con a[1.. $i - 1$] bé hơn a[i] vì vậy ta cần kiểm tra i/2 lần. Do đó, t_i = i/2 với $i = 2, 3, ..., n$. Tính tương tự như trên ta sẽ thấy

T(n) cũng là một hàm bậc 2 theo n. Trong nhiều bài toán, trường hợp trung bình thường khó xác định và người ta thường phân tích trường hợp xấu nhất.

1.3. Sắp xếp trộn (Merge Sort)

Thuật toán sắp xếp trộn dựa trên phương pháp "***chia để trị***" (***divide and conquer***). Phương pháp này gồm các bước sau:

- ***Chia*** (***divide***): Bài toán ban đầu được chia thành các bài toán con cùng loại.
- ***Trị*** (***conquer***): Các bài toán con được giải quyết một cách đệ qui. Nếu kích cỡ của bài toán con đủ nhỏ thì giải quyết nó bằng cách thông thường.
- ***Kết hợp*** (***combine***): Kết hợp lời giải cho các bài toán con vào lời giải cho bài toán ban đầu.

Ý tưởng thuật toán

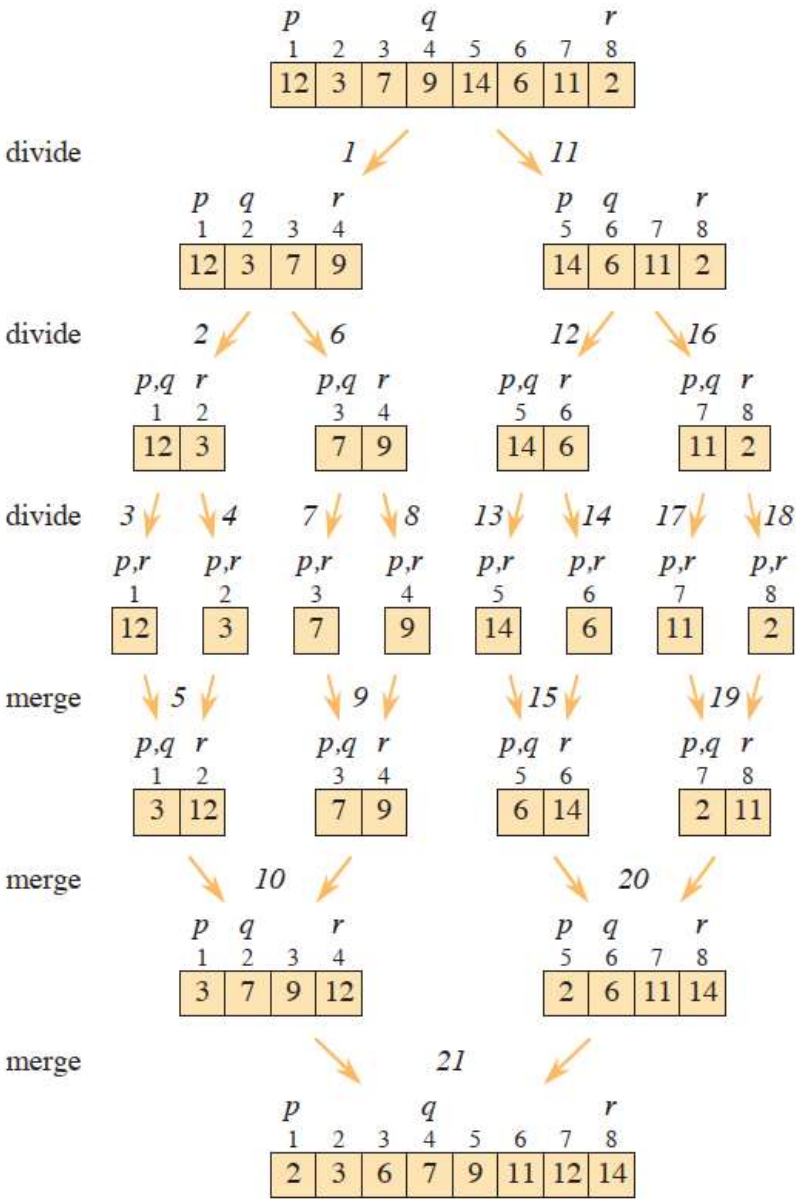
- ***Chia*** (***divide***): Chia mảng cần sắp xếp ban đầu thành 2 mảng con.
- ***Trị*** (***conquer***): Gọi đệ qui việc sắp xếp 2 mảng con cho đến khi kích thước mảng con chỉ còn một phần tử (mảng chỉ gồm một phần tử thì đã được sắp).
- ***Kết hợp*** (***combine***): Trộn 2 mảng con đã được sắp xếp để cho ra mảng kết quả.

Mã giả (***Pseudo-code***)

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Thủ tục `Merge(a, left, mid, right)` trộn 2 dãy đã sắp tăng a[left … mid] và a[mid + 1 … right] thành dãy sắp tăng a[left … right]. Thủ tục này dùng 2 mảng tạm để lưu giá trị 2 mảng con của mảng ban đầu.

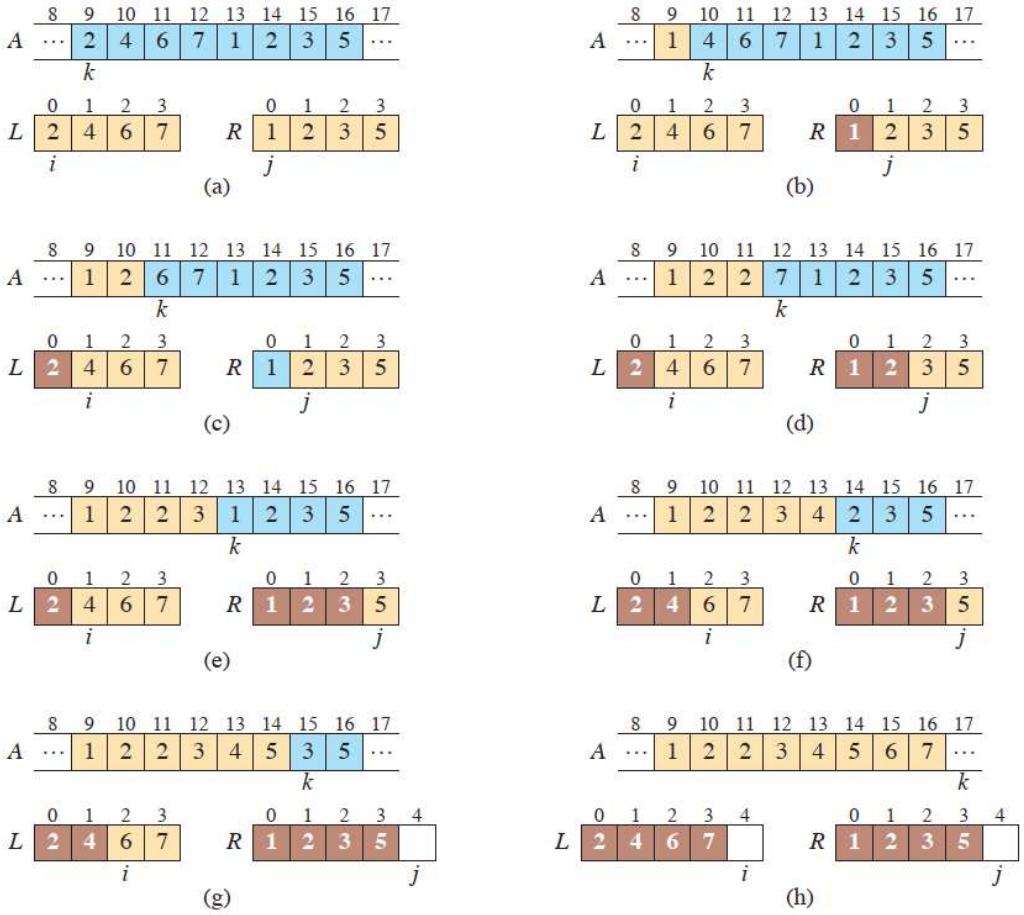
Hình bên dưới minh họa từng bước của thuật toán MERGE-SORT với mảng A = {12, 3, 7, 9, 14, 6, 11, 2}.


p q r
1 2 3 4 5 6 7 8
12 3 7 9 14 6 11 2
divide
1
11
p q r
1 2 3 4
12 3 7 9
p q r
5 6 7 8
14 6 11 2
divide
2
6
12
16
p,q r
1 2
12 3
p,q r
3 4
7 9
p,q r
5 6
14 6
p,q r
7 8
11 2
divide
3
4
7
8
13
14
17
18
p,r
1
12
p,r
2
3
p,r
3
7
p,r
4
9
p,r
5
14
p,r
6
6
p,r
7
11
p,r
8
2
merge
5
9
15
19
p,q r
1 2
3 12
p,q r
3 4
7 9
p,q r
5 6
6 14
p,q r
7 8
2 11
merge
10
20
p q r
1 2 3 4
3 7 9 12
p q r
5 6 7 8
2 6 11 14
merge
21
p q r
1 2 3 4 5 6 7 8
2 3 6 7 9 11 12 14

Chi tiết thuật toán MERGE để trộn hai mảng con được minh họa ở hình sau:

```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //      copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_R
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //      remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Hình sau minh họa từng bước của thuật toán MERGE khi gọi MERGE(A, 9, 12, 16) với mảng A = {2, 4, 6, 7, 1, 2, 3, 5}.



1.4. Sắp xếp nhanh (Quick Sort)

Thuật toán sắp xếp nhanh cũng dựa trên phương pháp "***chia để trị***".

Ý tưởng thuật toán

- ***Chia*** (***divide***): Mảng cần sắp xếp ban đầu được phân hoạch thành 2 mảng con A[p … q] và a[q + 1 … r] sao cho mỗi phần tử của đoạn a[p … q] đều bé hơn hoặc bằng mọi phần tử của đoạn a[q + 1 … r]. Chỉ số q được tính dựa vào giá trị của phần tử được chọn để phân hoạch.
- ***Trị*** (***conquer***): Sắp 2 mảng con a[p … q] và a[q + 1 … r] một cách đệ qui.
- ***Kết hợp*** (***combine***): Không làm gì cả (vì các mảng con đã được sắp trong quá trình phân hoạch).

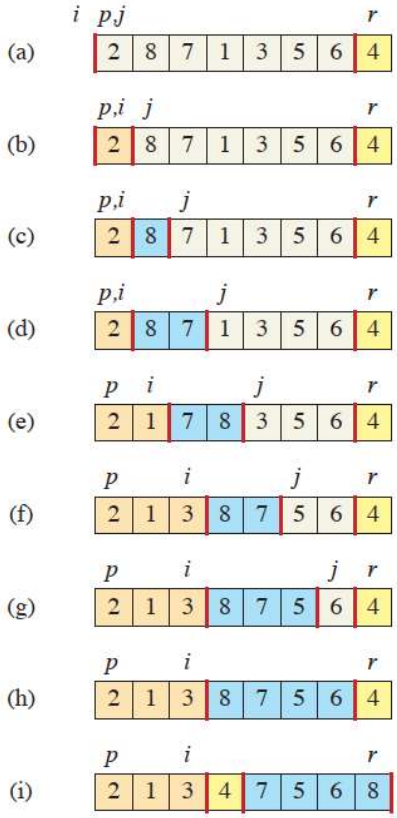
***Mã giả* (*Pseudo-code*)**

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Hàm `PARTITION(A, p, r)` phân hoạch mảng A dựa vào phần tử cuối `A[r]`.

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                     // index of a new slot in the low side
6          exchange A[i] with A[j]       // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

Hình sau minh họa thuật toán PARTITION với mảng A = {2, 8, 7, 1, 3, 5, 6, 4}.



1.5. Bài tập áp dụng

1. Cài đặt thuật toán sắp xếp SelectionSort, InsertionSort, MergeSort ở trên bằng ngôn ngữ lập trình cụ thể.
    a. Chạy các thuật toán với mảng cụ thể a = {2, 4, 5, 7, 1, 2, 3, 6}.
    b. Thay đổi thuật toán để trở thành sắp xếp mảng giảm dần.
    c. Tạo một mảng ngẫu nhiên gồm n = 10^5 phần tử kiểu số nguyên có giá trị trong khoảng từ 1 đến 10^9. Thêm vào thuật toán trên các đoạn code để đếm số phép gán và số phép so sánh. Thực hiện chạy các thuật toán trên 10 lần (dùng vòng lặp for) và xuất kết quả số phép gán, số phép so sánh, và thời gian chạy ra file. Bạn có thể dùng đoạn mã sau để tính thời gian chạy.

```
#include <chrono>
using namespace std::chrono;

auto start = system_clock::now();
// Your Code to Execute //
auto end = system_clock::now();
auto duration = duration_cast<milliseconds>(end – start);
cout << duration.count() << endl; // "mili seconds"
```

    d. Làm tương tự ***câu c*** nhưng thêm vào thuật toán sắp xếp các đoạn code để đếm số phép gán số học, số phép so sánh số học, số phép gán khóa, số phép so sánh khóa (phần tử của mảng), và thời gian chạy.
2. Làm tương tự câu 1 cho thuật toán QuickSort:
    a. Viết lại thuật toán sắp xếp QuickSort nếu phần tử được chọn làm phân hoạch là phần tử cuối của mảng.
    b. Viết lại thuật toán sắp xếp QuickSort nếu phần tử được chọn làm phân hoạch là phần tử đầu của mảng.
    c. Viết lại thuật toán sắp xếp QuickSort nếu phần tử được chọn làm phân hoạch là một phần tử tại một vị trí ngẫu nhiên trong mảng.
3. Phân tích độ phức tạp của các thuật toán SelectionSort, MergeSort, và QuickSort.

II. Tìm kiếm

2.1. Tìm kiếm tuần tự (Linear Search)

Ý tưởng thuật toán

Duyệt qua từng vị trí i của mảng a và kiểm tra xem A[i] có bằng x hay không.

***Mã giả* (*Pseudo-code*)**

```
for i ← 1 to n do
      if A[i] = x then
            return i
return -1
```

2.2. Tìm kiếm nhị phân (Binary Search)

Thuật toán này được áp dụng khi mảng a đã được sắp xếp, tức là khi mảng A đã thỏa mãn $A[1] \le A[2] \le \ldots \le A[n]$. Khi đó, nếu $x > A[i]$ thì x chỉ có thể xuất hiện trong đoạn A[i+1..n], ngược lại, nếu $x < A[i]$ thì x chỉ có thể xuất hiện trong đoạn A[1..i-1]. Dựa trên nhận xét này, ta có một cách hiệu quả hơn để tìm x mà không cần phải duyệt tuần tự từ đầu mảng đến cuối mảng (hay ngược lại) như thuật toán trên.

Ý tưởng thuật toán

So sánh x với phần tử giữa mảng A[m],

- Nếu x = A[m] thì xuất m và kết thúc,
- Nếu x < A[m] thì tìm x trong đoạn a[left..m-1],
- Nếu x > A[m] thì tìm x trong đoạn a[m+1..r].

Mã giả* (*Pseudo-code*) *– dùng đệ quy

```
BinarySearch(A[left..right], x)
      if right >= left
            mid ← left + (right – left)/2
            if x < a[mid]
                  return BinarySearch(A[1eft..mid-1, x])
            else if x > A[mid]
                  return BinarySearch(A[mid+1..right],x)
            else
                  return mid
      return -1   // Duyệt hết mảng A mà không tìm thấy x.
```

Mã giả* (*Pseudo-code*) – *dùng vòng lặp

```
BinarySearch(A[left .. right], x)
      do
            mid ← (left + right)/2
            if x = A[mid] then
                  return mid
            else if x < A[mid] then
                  right ← mid - 1
            else
                  left ← mid + 1
      while left <= right
      return -1   // Duyệt hết mảng A mà không tìm thấy x.
```

2.3. Bài tập áp dụng

1. Cài đặt thuật toán tìm kiếm LinearSearch và BinarySearch ở trên bằng một ngôn ngữ lập trình cụ thể.
    a. Chạy thuật toán với mảng cụ thể A = {2, 4, 5, 7, 1, 2, 3, 6} và x = 4, 8.
    b. Tạo một mảng ngẫu nhiên gồm n = 10^5 phần tử kiểu số nguyên có giá trị trong khoảng từ 1 đến 10^9. Thêm vào thuật toán trên các đoạn code để đếm số phép so sánh khóa (giữa phần tử của mảng với x). Thực hiện chạy các thuật toán trên 10 lần (dùng vòng lặp for) và xuất số phép so sánh, thời gian chạy ra file.
2. Phân tích độ phức tạp của các thuật toán LinearSearch và BinarySearch.

Tài liệu tham khảo

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms, Forth Edition*, The MIT Press, 2022.