ĐỒ ÁN NHẬP MÔN PHÂN TÍCH ĐỘ PHỨC TẠP THUẬT TOÁN

Sinh viên thực hiện:	GV hướng dẫn:
1712919 Lê Văn Vũ	Thầy **Trần Đan Thư** Thầy **Vũ Quốc Hoàng**

MỤC LỤC

Các nguồn tham khảo trong bài làm:

https://voer.edu.vn/c/cac-phuong-phap-sap-xep-co-ban/7c75a38d/6885d36d
https://www.includehelp.com/algorithms/radix-sort-and-its-algorithm.aspx
https://www.bigocheatsheet.com/
https://www.cse.wustl.edu/~sg/CSE241_FL07/hw3-practice-sols.pdf
https://www.includehelp.com/algorithms/radix-sort-and-its-algorithm.aspx
https://cs.fit.edu/~pkc/classes/writing/hw13/luis.pdf
Và một số nguồn khác

BẢNG TÓM TẮT:

Tên	Tốt nhất	Trung bình	Xấu nhất	ĐPT không gian (xấu nhất)	Phương pháp
Select sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)	Selection
Bubble sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)	Exchanging
Quick sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))	Partitioning
Merge sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)	Merging
Radix sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)	

I. SELECTION SORT:

1. Số phép gán, so sánh:

```
void SelectionSort(int a[],int N ){                                    P
 int   min; // chỉ số phần tử nhỏ nhất trong dãy hiện hành
        for (int  i=0; i<N-1 ; i++)                                    Q
            for(int j = i+1; j <N ; j++)                               R
                    if (a[j ] < a[min])
                        min = j;
                Hoanvi(a[min], a[i]);
        }
}
```

Trường hợp	Số phép so sánh	Số phép gán
Tốt nhất	$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$	0
Xấu nhất		$3.\sum_{i=1}^{n-1}(n-i)=3.\frac{n(n-1)}{2}$

2. Độ phức tạp thời gian:

Xét đoạn mã giả sau:

	Chi phí (cost)	Số lần thực thi (times)
N ← length[A]	C1	1
For j ← 1 to n-1	C2	N-1
Do smallest ← j	C3	N-1
For i ← j+1 to n	C4	$\sum_{j=1}^{n-1}(N-j+1)=\frac{1}{2}N^2+\frac{3}{2}N-2$
≈N2/2 so sánh, do if A[i]<A[smallest]	C5	$\sum_{j=1}^{n-1}(N-j)=\frac{1}{2}N^2+\frac{1}{2}N-1$

Then smallest ← i	C6	$\sum_{j=1}^{n-1}(N-j)=\frac{1}{2}N^2+\frac{1}{2}N-1$
≈N hoán vị, hoán vị A[j] ↔ A[smallest]	C7	N-1

Do đó, độ phức tạp thời gian = **O(n^2)**

II. BUBBLE SORT:

```
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}
```

1. Số phép so sánh, hoán vị:

Trường hợp	Số phép so sánh	Số phép hoán vị
Tốt nhất	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$

2. Độ phức tạp:

```
int bubble_sort(int arr[]){
     int n=arr.length; ---------------->1
     Int temp; ----------------------->1
for(int i=n-1; i>=1; i--){-----------> 1+n+n-1=2n
  for(int j=0;j<i;j++){--------------->(n-1)+(x+1)+x
     if(arr[j]>arr[j+1]{-------------> 3x
         //swap
          temp=arr[j];----------------> 2x
          arr[j]=arr[j+1];------------> 2x
          arr[j+1]=temp;--------------->2x
  }}}
  return arr;------------------------> 1
}
T(n)=1+1+2n+(n-1)+x+1+3x+2x+2x+2x+1= 11x+3n+3
x= 1+2+3+…+(n-1)=(n-1)(n-1+1)/2 = n(n-1)/2
```

$$T(n) = \frac{11}{2}n^2 - \frac{11}{2}n + 3n + 3$$
$$\Rightarrow T(n) \in O(n^2)$$

III. QUICK SORT

Mã giả:

```
function quicksort(array)
        if length(array) ≤ 1
            return array // an array of zero or one elements is already sorted
        select and remove a pivot element pivot from 'array'
        // see '#Choice of pivot' below
        create empty lists less and greater
        for each x in array
        if x ≤ pivot then append x to less
        else append x to greater
        return concatenate(quicksort(less), list(pivot), quicksort(greater))
        // two recursive calls
```

1. Phân tích trường hợp trung bình sử dụng phép truy toán (recurrences):

- Trong TH "không cân bằng" nhất, một lần gọi QuickSort = O(n) + (2 lần gọi đệ quy trên danh sách kích thước 0 và n-1). Do đó: T(n) =O(n) + T(0) + T(n-1) = O(n) + T(n-1)

- Còn trong trường hợp "cân bằng" nhất: T(n) =O(n) + 2T($\frac{n}{2}$)

Suy ra: **T(n) = O(nlogn)**

Chứng minh: Giả sử không có trùng lặp như có thể được xử lý với xử lý trước và sau xử lý thời gian tuyến tính, hoặc xem xét các trường hợp dễ dàng hơn các phân tích. Khi đầu vào là một hoán vị ngẫu nhiên, thứ hạng của pivot là ngẫu nhiên từ 0 đến n-1. Sau đó, các phần kết quả của phân vùng có kích thước i và n-i-1 và i là ngẫu nhiên từ 0 đến n-1. tính trung bình trên tất cả các phân chia có thể có và số lượng so sánh cho phân vùng là n-1. Số lượng phép so sánh được ước tính:

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-1-1))$$

Sử dụng phép truy toán, ta có: $C(n) = 2n\ln n = 1{,}39n\log_2 n$

(Điều này có nghĩa là, trung bình, quicksort chỉ thực hiện kém hơn khoảng 39% so với trường hợp tốt nhất của nó. *(Hay nó gần hơn trường hợp tốt nhất hơn trường hợp xấu nhất)*).

2. Phân tích QuickSort ngẫu nhiên:

Với mỗi lần thực hiện Quicksort tương ứng với cây tìm kiếm nhị phân (BST) sau:
+ Pivot ban đầu là nút gốc (Node);
+ Pivot của nửa bên trái là gốc của cây con bên trái,
+ Pivot của nửa bên phải là gốc của cây con bên phải,
Và tương tự các pivot khác

Số phép so sánh của Quicksort bằng với số lượng so sánh trong xây dựng BST bằng một chuỗi các phần chèn thêm. Vì vậy, số lượng so sánh trung bình cho ngẫu nhiên Quicksort bằng với chi phí trung bình của việc xây dựng BST khi các giá trị được chèn $(x_1, x_2, ..., x_n)$ tạo thành 1 hoán vị ngẫu nhiên.

- Xét một BST được tạo bằng cách chèn một chuỗi $(x_1, x_2, ..., x_n)$ của các giá trị hình thành một hoán vị ngẫu nhiên. Gọi C là chi phí tạo BST, ta có:

$C = \sum_i \sum_{j<i}$ (cho dù trong quá trình chèn x_i có so sánh với x_j)

Theo tuyến tính của kì vọng, $E(C) = \sum_i \sum_{j<i}$ (cho dù trong quá trình chèn x_i có so sánh với x_j)

Sửa i và j<i, giá trị $(x_1, x_2, ..., x_j)$ sau khi được sắp xếp, xác định khoảng j+1.

Từ $(x_1, x_2, ..., x_n)$ là 1 hoán vị ngẫu nhiên, $(x_1, x_2, ..., x_j, x_i)$ cũng là 1 hoán vị ngẫu nhiên,

→ Xác suất x_i liền kề với x_j là $\frac{2}{j+1}$

Vì vậy: $E(C) = \sum_i \sum_{j<i} \frac{2}{j+1} = O(\sum_i \log i) =$ **O(nlogn)**

3. Độ phức tạp thời gian:

- Với n=1, T(n) =a

- n>1, ta có:

$$T(n) = 2T(\frac{n}{2}) + n$$

$$= 2.(2T(\frac{n}{4}) + \frac{n}{2}) + n$$

$$= 2^2 T(\frac{n}{2^2}) + n + n$$

$$= 2^2 (2T(\frac{n}{2^3}) + \frac{n}{4}) + n + n$$

$$= 2^3 T(\frac{n}{2^3}) + n + n + n$$

$$= ...$$

$$= 2^k T(\frac{n}{2^k}) + nk = nT(\frac{n}{n}) + n\log n$$

$$= na + n\log n = \Theta(n\log n)$$

IV. MERGE SORT

***Mã giả:**

```
MERGESORT:
Input: List L (contains N elements)
Output: List L (ascending order)
MERGESORT(LIST L):
1.  if(N > 1)
2.     (List left, List right) <- DIVIDE(L)
3.     Left  <- MERGESORT(left)
4.     right <- MERGESORT(right)
5.     L <- MERGE(left, right)

MERGE:
Input: List A (sorted ascending order),
List B (sorted ascending order)
Output: List L (sorted ascending order)
MERGE(List left, List right)
6.  i = j = k = 0
7.  while(i < LEN(left) and j < LEN(right))
8.     if(left[i] < right[j])
9.        L[k++] = left[i++]
10.    else L[k++] = right[j++]
11. if(i == LEN(left)) append(L, rest(left))
12. else append(L, rest(right))
```

1. Phân tích MergeSort:

- Để đơn giản hóa, ta giả sử n là lũy thừa của 2, tức là: $n=2^k$
- Đặt T (n) biểu thị thời gian chạy hợp nhất trong trường hợp xấu nhất trên một mảng gồm n phần tử, Ta có:

$$T(n) = c_1 + T(\frac{n}{2}) + T(\frac{n}{2}) + c_2 n$$

$$= 2T(\frac{n}{2}) + (c_1 + c_2 n)$$

Hay $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

- Ta có thể thấy rằng sự tái diễn có giải pháp $\Theta(n \log_2 n)$ bằng cách nhìn vào cây đệ quy (Recursion Tree):
  tổng số cấp trong cây đệ quy là $\log_2 n + 1$ và mỗi cấp có chi phí thời gian tuyến tính.
- Nếu **n#2^k:**

$$T(n) = \begin{cases} \Theta(1) \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor + \Theta(n) \end{cases}$$

2. Giải quyết các lần lặp bằng cách lặp: (Solving Recurrences by iteration)

- Ví dụ, ta giải quyết: $T(n) = 8T(n/2) + n^2$ (với T(1)=1)

$$T(n) = n^2 + 8T(n/2)$$

$$= n^2 + 8(8T(\frac{n}{2^2}) + (\frac{n}{2})^2)$$

$$= n^2 + 8^2 T(\frac{n}{2^2}) + 8(\frac{n^2}{4}))$$

$$= n^2 + 2n^2 + 8^2 T(\frac{n}{2^2}) = n^2 + 2n^2 + 8^3 (T(\frac{n}{2^3}) + (\frac{n}{2^2})^2)$$

$$= n^2 + 2n^2 + 8^3 T(\frac{n}{2^3}) + 8^2 (\frac{n^2}{4}) = n^2 + 2n^2 + 2^2 n^2 + 8^3 T(\frac{n}{2^3}) = \ldots$$

$$= n^2 + 2n^2 + 2^2 n^2 + 2^3 n^2 + \ldots$$

- Độ sâu đệ quy: Mất bao lâu (bao nhiêu lần lặp) cho đến khi bài toán con có kích thước không đổi?

I lần khi $\frac{n}{2^i} = 1 \Rightarrow i = \log n$

- Kì hạn cuối cùng là gì? $8^i T(1) = 8^{\log n}$

$$T(n) = n^2 + 2n^2 + 2^2 n^2 + 2^3 n^2 + \ldots + 2^{\log n - 1} n^2 + 8^{\log n}$$

$$= \sum_{k=0}^{\log n - 1} 2^k n^2 + 8^{\log n}$$

$$= n^2 \sum_{k=0}^{\log n - 1} 2^k + (2^3)^{\log n}$$

- $\sum_{k=0}^{\log n - 1} 2^k$ là 1 tổng hình học (geometric sum), ta có $\sum_{k=0}^{\log n - 1} 2^k = \Theta(2^{\log n - 1}) = \Theta(n)$

$$T(n) = n^2 . \Theta(n) + n^3 = \Theta(n^3)$$

3. Độ phức tạp thời gian:

	MergeSort(A){
C1	n ←length(A) If(n<2) return mid ← n/2 left ← array of size(mid) right ← array of size(n-mid)
C2.n	for i←0 to mid -1 left[i]←A[i] for i ← mid to n-1 right[i - mid]←A[i]
T(n/2)	MergeSort(left)
T(n/2)	MergeSort(right)
C3.n+C4	MergeSort(left, right, A) }

$$T(n)=\begin{cases} c \\ 2T(n/2)+(c_2+c_3)n+(c_1+c_2) \end{cases}$$

Ta có: $$T(n)=\begin{cases} c \\ 2T(n/2)+c'n+c'' \end{cases}$$

$$T(n)=2\left[2T(\frac{n}{4})+c'.\frac{n}{2}\right]+c'n$$

$$=4T(n/4)+2c'.n$$

$$=4\left[2T(n/8)+c'.n/4\right]+2c'n$$

$$=8T(n/8)+3c'n=16T(n/16)+4c'n$$

$$=2^kT(\frac{n}{2^k})+kc'n$$

$$(k=\log_2 n)$$

$$=2^{\log_2 n}T(1)+\log_2 c'n$$

$$=nc+c'n\log n$$

$$=\Theta(n\log n)$$

V. Radix sort:

Đoạn mã Radix sort:

```
radix_sort(sortPP, endPP, offset)
char**sortPP; // Array of pointers to keys
char**endPP;// Address beyond end of array
int offset;// Byte offset In key to sort on
{
    extern int keysize;// Size of each key
examined
    extern unsigned slot_size[256];// Zero on
each entry to routine
```

```
{
    register int c;
    register char *key, *key1P;

    head = slot_list[head];// End partition
done automatically
    do {
        PP = slotPP[head];//Last unsorted
partition
```

```
    static int slot_head;// Head of linked list
    char slot_list[256];// Offset of next node
in list
    char **slotPP[256];// Pointer to each
partition

    register char **pP;
    register int head;
    register int off;

    off = offset;
    while (1) {
        register char**sPP;// Register copy of
sortPP
        register int keys;// Number of keys to
sort

        PP = endPP;
        sPP = sortPP;//Place parameters In
registers
        keys - PP - sPP;// Number of keys to sort

        if (keys < 16)return;

        do {
            head = («—PP)[off];
            slot_slze[head]++;
        } while (PP > sPP);
        if (slot_slze[head] = keys) {
            slot_slze[head] = 0; //All keys
have same byte value
            if (++off >= keysize) return;
            continue;
        }
    }
    register int size; //Size of a partition
    register int i; //Counter being examined
    head = 255; //End of list marker
    i = -1;
    do { /* For each occurring byte value — */
        while (Ksize = slot size[++i]));
        slot_list[l] = head;
        head = 1;
        slotPP[i] = PP;
        PP += size;
    } while (PP < endPP);
    slot head = head;
}
```

```
        while (slot_slze[head]) {//While this
partition not formed
            keyP ~= *PP;
            do {
                c = keyP[off];
                slot_slze[c]--;
                PP = slotPP[c] ++;
                key1P = *PP;
                *PP = keyP;
                keyP = key1P;
            } while (c != head);
            PP++;
        }
        head = slot_list[head];
    } while (head != 255);
    head = slot_head;
    slot_size[head] = 0;
}
if (++off >= keysize)return;// No subsequent
bytes in key

head = slot + list[head];
do {
    PP = slotPP[head];//Start of next partition
    radlx_sort(PP, endPP, off);//Sort last
unsorted partition
    endPP = PP;// End of previous partition
    head = slot_list[head];//Move back through
partitions
} while (head != 255);//— until only the first
not done * /
//Now sort first partition
```

Giả sử các loại cơ số với n=N key cố định, mỗi khóa chứa m byte và các byte này được gán các giá trị ngẫu nhiên từ 1 bảng chữ cái của các kí hiệu p.

→ Số lượng byte trên mỗi key khoảng $\log_p N$, bất cứ khi nào có các key và riêng biệt và ta có: $\log_p N \le m$.

- Tuy nhiên, vì ta cố gắng sắp xếp bộ ít hơn 16 key, do đó giảm xuống khoảng $\log_p \frac{N}{16}$. Trong trường hợp xấu nhất, điều này tăng lên m. Radix Sort kiểm tra từng byte đáng kể trong mỗi khóa nhiều nhất 2 lần; một lần khi đếm tần số các giá trị byte và lần còn lại khi phân vùng trên các giá trị byte này. Do đó, ta có thể mong đợi thự hiện $< c_1 N \log_p \frac{N}{16}$ hoạt động khi đếm

byte và trao đổi các key pointers, trong TH xấu nhất sẽ thực hiện ít hơn c1mN (với 1 hằng số nhỏ c1).
- Xét số lần thuật toán tạo, sau đó sử dụng một danh sách liên kết để phân vùng key:
+ Khi p=1: không xảy ra.
+ p<1: số lượng các node dự kiến trong 1 trie chứa các key ngẫu nhiên được rút ra từ bảng chữ cái p>1 là N/lnp, và tương ứng với số lần thử dự kiến cho các key phân vùng. Tuy nhiên, vì ta không phân vùng các bộ có ít hơn 16 key nên nó giảm xuống khoảng $\frac{N}{16\ln p}$ (khi tất các các key là khác biệt).

- Nếu các key luôn được phân vùng thành tập con r và có $r^k \le r^m$ key, sau đó có $\sum_{i=0}^{k-1} r^i$ các bước phân vùng được thực hiện, hay $\sum_{i=0}^{k-1} \frac{r^i}{r_k}$ các bước phân vùng trên mỗi key. Khi k tăng đến m, số bước phân vùng trên mỗi key do đó cũng tăng, bất kể giá trị r.

Với s>r, hãy so sánh $\sum_{i=-k}^{-1} r^i > \sum_{i=-k}^{-1} s^i$. Do đó, số lượng các bước phân vùng trên mỗi khóa lớn hơn khi phân vùng r^k key, r way so với khi phân vùng s^k key s way. Khi $r^k < s^k$ $\Rightarrow$ Số bước phân vùng tối đã trên mỗi key được giải phóng khi r min và k=m.

$\rightarrow$ Th xấu nhất thì $\sum_{i=-m}^{-1} 2^i = 1 - 2m \approx 1$ bước phân vùng xảy ra trên mỗi key. Tuy nhiên, vì các bộ ít hơn 16 key không được phân vùng, do đó giảm xuống 1 bước phân vùng trên 16 key.

==> Độ phức tạp hoạt động của thuật toán: $N*(c_1*\min(m, \log_p(\frac{N}{16})) + c_2 \frac{1}{16\ln p}))$

(Trong TH hợp xấu nhất, kết quả tăng đến $N*(c_1*m + c_2/16))$
==> Do đó, kết quả mong đợi là **O(m*N)** với mọi p cố định.

*** Độ phức tạp thời gian:**