**Đồ án Nhập Môn Phân Tích Độ Phức Tạp Thuật Toán**

**ĐỀ TÀI: Đánh Giá Độ Phức Tạp 5 Thuật Toán Yêu Thích**

Họ Và Tên: Lê Quang Nam

MSSV: 1712603

**Các thuật toán được trình bày**:

1. Thuật toán sắp xếp chọn (Selection Sort)
2. Thuật toán sắp nhanh (Quick Sort)
3. Thuật toán sắp xếp trộn (Merge Sort)
4. Thuật toán tìm kiếm nhị phân (Binary Search)
5. Thuật toán tìm đường đi ngắn nhất (Thuật toán Floyd)
6. Thuật toán sắp xếp chọn – Selection Sort

* Ý tưởng của thuật toán:

Chọn phần tử nhỏ nhất trong dãy cần sắp xếp và hoán đổi vị trí của phần tử nhỏ nhất và phần tử đầu tiên. Sau đó làm như vậy cho các phần tử tiếp theo cho đến khi chỉ còn 1 phần tử cuối cùng. Thì dãy trên đã được sắp xếp tăng dần. Còn sắp xếp giảm dần thì chọn phần tử lớn nhất và hoán đổi với vị trí đầu tiên.

* Thuật toán:

for (i = 0; i < n-1; i++)

{

min\_idx = i;

for (j = i+1; j < n; j++)

if (arr[j] < arr[min\_idx])

min\_idx = j;

swap(arr[min\_idx], arr[i]);

}

* Đánh giá độ phức tạp thuật toán:
* Số phép so sánh: Ta thấy vòng lặp for chạy n lần phần tử cần sắp xếp. Để tìm được phần tử nhỏ nhất đầu tiên, thì ta cần duyệt qua n phần tử trong dãy, cứ mỗi lần duyệt qua 1 phần tử (so sánh i và j) thì ta tốn 1 phép so sánh, vậy với n phần tử thì tốn (n-1) phép so sánh. Sau đó nó sẽ hoán vị với phần tử đầu tiên. Sau khi hoán vị phần tử đầu tiên thì tiếp tục với phần tử tiếp theo, ta cần duyệt qua (n-1) phần tử trong dãy, vậy ta tốn (n-2) phép so sánh. Cứ như vậy cho đến khi còn phần tử cuối cùng.
* Số phép so sánh: S = (n-1) + (n-2) + (n-3) + …. + (n-(n-1)) =

n\*(n-1)/2 = (n2-n)/2 = O(n2)

* Số phép gán: Ta có mỗi lần hoán vị swap thì ta có 3 lần gán.
* Số phép gán: G = (n-1) + (n-1) + 3n(n-1) = 5n(n-1) = 5n2 – 5n = O(n2)
* Độ phức tạp T(n) = Số phép gán + Số phép so sánh = (n2-n)/2 + 5n2 – 5n = 6n2 - 12n = O(n2)
* Kết luận: Với thuật toán trên thì độ phức tạp là O(n2) trong 3 trường hợp
* Trường hợp tốt nhất: O(n2)
* Trường hợp trung bình: O(n2)
* Trường hợp xấu nhất: O(n2)

1. Thuật toán sắp xếp nhanh – Quick Sort

* Ý tưởng thuật toán:
* QuickSort chia mảng thành hai danh sách bằng cách so sánh từng phần tử của danh sách với một phần tử được chọn được gọi là phần tử chốt. Những phần tử nhỏ hơn hoặc bằng phần tử chốt được đưa về phía trước và nằm trong danh sách con thứ nhất, các phần tử lớn hơn chốt được đưa về phía sau và thuộc danh sách con thứ hai. Cứ tiếp tục chia như vậy tới khi các danh sách con đều có độ dài bằng 1.
* Sau 1 lượt phân hoạch ta có:

+ V0 … Vj < x, phân hoạch tiếp V0… Vj.

+ Vj+1 Vi-1 = x

Vi … Vn-1 > x, phân hoạch tiếp Vi… Vn-1

* Thuật toán:

|  |
| --- |
| {  int x=a[(l+r)/2];  int i=l;  int j=r;  do {  while (a[i]<x)  i++;  while (a[j]>x)  j--;  if (i<=j)  swap(a[i++], a[j--]);  } while (i<j);  if (l<j)  quicksort(a,l,j);  if (i<r)  quicksort(a,i,r);  } |

* Đánh giá độ phức tạp thuật toán:

Ta nhận thấy hiệu quả của thuật toán phụ thuộc vào việc chọn giá trị mốc (hay phần tử chốt).

* Trường hợp tốt nhất của thuật toán – x nằm ở vị trí medium:

+ Lấy vị trí mid là phần tử nằm ở giữa.

+ Mảng 1 và mảng 2 sẽ có kích thước bằng nhau.

T(n) = t(k) + t(n-k) + cn trong đó:

+ Ta gọi t(k) là thời gian sắp xếp mảng với k phần tử

+ t(n – k) là thời gian sắp xếp mảng với (n – k) phần tử

+ cn là tổng thời gian sắp xếp lại mảng

Ta có hai mảng như sau:

+ Mảng 1: Có n/2 phần tử

+ Mảng 2: Có n/2 phần tử

* T(n) = t(n/2) +t(n/2) + cn

= 2\*t(n/2) + cn

= 2[2t(n/4) + c\*(n/2)] + cn

= 22t(n/4) + 2cn

= 22[2t(n/8) + c\*(n/4)] + 2cn

= 23t(n/8) + 3cn

…

= 2kt(n/2k) + kcn (\*)

(\*) sẽ dừng khi (n/2k) = 1

⬄ n/2k = 1 ⬄ n = 2k ⬄ k = log2n

Thay k vào (\*) ta được:

2kt(n/2k) + kcn

= 2^(log2n) \* t [n/2^(log2n)] + log2n\*cn

= 2^(log2n) \* t (1) + log2n\*cn

Ta có: 2^(log2n) = n

= nt (1) + log2n\*cn = n + cn\*log2n = O(n) + O(nlog2n)

* Độ phức tạp cho trường hợp tốt nhất là: O(nlog2n)
* Trường hợp xấu nhất của thuật toán – x nằm ở vị trí min hoặc vị trí max:

+ Ta lấy vị trí đầu hoặc cuối – chọn vị trí đầu và ta có hai mảng như sau:

+ Mảng 1: phần tử đầu tiên – 1 phần tử

+ Mảng 2: gồm (n-1) phần tử

T(n) = t(k) + t(n-k) + cn trong đó:

+ Ta gọi t(k) là thời gian sắp xếp mảng với k phần tử

+ t(n – k) là thời gian sắp xếp mảng với (n – k) phần tử

+ cn là tổng thời gian sắp xếp lại mảng

* T(n) = t(n-1) + t(1) + cn

= [t(n-2) + t(1) + c(n-1)] + t(1) + cn

= t(n-2) + 2t(1) + c[(n-1) + n]

= [t(n-3) + t(1) + c(n-2)] + 2t(1) + c[(n-1) + n]

= t(n-3) + 3t(1) + c[(n-2) + (n-1) + n]

…

= t(n-k) + kt(1) + c[(n – k +1) + (n – k) + …+(n-2) + (n-1) + n]

=t(n-k) + kt(1) + c()

= t(n-n+1) + (n-1)t(1) + c()

= t(1) + (n-1)t(1) + c()

= nt(1) + c()

= n + c()

= n + c[(n-2)\*(n+1)/2]

= n + c[(n2 – n - 2 )/2]

= O(n) + O(n2)

* Độ phức tạp của thuật toán với trường hợp xấu nhất là: O(n2)
* Trường hợp trung bình – x nằm ở vị trí bất kỳ trừ 2 trường hợp trên:

+ Mảng 1: có k phần tử

+ Mảng 2: có n-k phần tử

* T(n) = T(n - k) + T(k) + cn

= [T(n – k -1) + T(k) + c(n-1)] + T(k) + cn

= T(n-k-1) + 2T(k) + c[(n-1) + n]

= [T(n-k-2) + T(k) + c(n-2)] + 2T(k) + c[(n-1) + n]

= T(n-k-2) + 3T(k) + c[(n-2) + (n-1) + n]

…

= T(n – k – i) + (i +1)T(k) + c[(n – k) + …+(n-2) + (n-1) + n]

= T(n – k – i) + (i+1)T(k) + c()

= T(n – n + k – i ) + (i+1)T(k) + c()

= T(k-i) +(i+1)T(1) + c()

= T(1) + kT(1) + c()

= (k+1)T(1) + c()

= O(n) + O(nlog2n)

* Độ phức tạp của thuật toán với trường hợp trung bình là: O(nlog2n)

1. Thuật toán sắp xếp Merge Sort

* Ý tưởng thuật toán:

- Merge sort là một thuật toán chia để trị. Thuật toán này chia mảng cần sắp xếp thành 2 nửa. Tiếp tục lặp lại việc này ở các nửa mảng đã chia. Sau cùng gộp các nửa đó thành mảng đã sắp xếp. Hàm merge() được sử dụng để gộp hai nửa mảng. Hàm merge(arr, l, m, r) là tiến trình quan trọng nhất sẽ gộp hai nửa mảng thành 1 mảng sắp xếp, các nửa mảng là arr[l…m] và arr[m+1…r] sau khi gộp sẽ thành một mảng duy nhất đã sắp xếp.

- Trộn trực tiếp: đây là phương pháp trộn đơn giản nhất. Việc phân hoạch dãy ban đầu đơn giản như sau: Với dãy ban đầu có n phân tử, ta cứ phân hoạch thành n dãy con. Vì rằng mỗi dãy con chỉ có 1 phần tử nên nó là dãy có thứ tự. Cứ mỗi lần tách – trộn, chiều dài của dãy con sẽ được nhân đôi.

\* Thuật toán:

|  |
| --- |
| function merge (l, r)  if l >= r then return fi {1}  m <- [(r + l)/2]; {1}  merge (l, m-1); {T(n/2) }  merge (m, r); {T(n/2) }  i <- l; {1}  j <- m; {1}  k <- l; {1}  while(k<=r) do { n }  if(a[i] <=a[j] and i < m or j > r) { n }  then b[k] = a[i]; k<- k+1; i <- i+1; { n }  else b[k] = a[j]; k <- k+1; j <- j +1; { n }  endw;  for I = l, . . . , r do a[k] <- b[k] { n } |
|  |

* Độ phức tạp thuật toán:
* T(n) = 5 + 2\*T(n/2) + 10n + 2n

= 5 +2\*(5 + 2\*T(n/4) + 5n + n) + 10n + 2n

= 5 + 2\*(5 + 2\*T(n/4) + 6n) + 12n

= 15 + 4\*T(n/4) + 24n

= 15 + 4\*(5 + 2\*T(n/8) + 12n/4) + 24n

= 35 + 8\*T(n/8) + 36n

= 2k \* T(n/2k) + 12n\*k + 5\*(1 + 2 + … + k) {1}

Điều kiện dừng của thuật là khi n = 2k ⬄ k = log2n

Thay k = log2n vào {1} ta được:

2k \* T(n/2k) + 12n\*k + 5\*(1 + 2 + … + k) {1}

= n\*T(n/n) + 12n\* log2n + 5k(k+1)/2

= n + 12n\* log2n + 5 log2n (log2n +1)/2

= n +12n log2n + (5 log2n3 + 5 log2n)/2

Ta thấy n log2n > log2n > n nên độ phức tạp thuật toán là O(nlogn)

* Kết luận: Độ phức tạp của thuật toán merge sort là O(nlogn) đối với 3 trường hợp:
* Trường hợp xấu nhất: O(nlogn).
* Trường hợp trung bình: O(nlogn).
* Trường hợp tốt nhất: O(nlogn).

1. Thuật toán tìm kiếm nhị phân – Binary Search

* Ý tưởng thuật toán:

Do tính chất mảng đã sắp xếp, công việc tìm kiếm phần tử x có thể triển khai như sau:

* Xét đoạn mảng arr [left…right] cần tìm kiếm phần tử x. Ta so sánh x với phần tử ở vị trí giữa của mảng (mid = (left + right)/2). Nếu:
* Nếu phần tử arr[mid] = x. Kết luận và thoát chương trình.
* Nếu arr[mid] < x. Chỉ thực hiện tìm kiếm trên đoạn

arr [mid+1…right].

* Nếu arr[mid] > x. Chỉ thực hiện tìm kiếm trên đoạn arr [left…mid-1].
* Thuật toán:
* 4 left = 0;
* 5 right = n - 1;
* 6
* 7 **while** (left <= right) { {k}
* 8 mid = (left + right) / 2; {k}
* 9 **if** (x == a[mid]) {k}
* 10 **return** mid;
* 11 **else** **if** (x < a[mid]) {k}
* 12 right = mid - 1; {k}
* 13 **else** **if** (x > a[mid]) {k}
* 14 left = mid + 1; {k}
* 15 endw;
* 16 }
* 17 **return** -1;
* Độ phức tạp thuật toán;
* Với thuật toán tìm kiếm nhị phân thì mỗi khi ta so sánh x với mid thì số lượng phần tử còn lại sẽ là 1 nữa so với lúc chưa so sánh.
* Với n là số lượng phần tử và x là phần tử cần tìm kiếm ta được:
* Với điều kiện dừng là số lượng phần tử trong dãy không nhỏ hơn 1 phần tử.
* Khi thực hiện tìm kiếm ta có các bước lần lượt: (Ta làm tròn n/2, n/4, … là số nguyên khi n là số lẻ)

+ Lần 1: Số phần tử còn lại trong mảng: n/2 = n/21 phần tử

+ Lần 2: Số phần tử còn lại trong mảng: n/4 = n/22 phần tử

+ Lần 3: Số phần tử còn lại trong mảng: n/8 = n/23 phần tử

+ . . .

+ Lần k -1: Số phần tử còn lại trong mảng: n/k – 1= n/2k-1 phần tử

+ Lần k: Số phần tử còn lại trong mảng: n/k = n/2k phần tử

Với điều kiện dừng là số phần tử còn lại trong mảng là 1 nên

n/k = n/2k =1 <=> n = 2k => k = log2n

* Số phép gán: 2 + k + 2k = 2 + 3k = 2 + 3 log2n = O(logn)
* Số phép so sánh: k + k + k + k = 4k = 4 log2n = O(logn)
* Độ phức tạp trong trường hợp xấu nhất:

T(n) = 2 + 11k = 2 + 11 log2n = O(logn)

1. Thuật toán Floyd – Thuật toán tìm đường đi ngắn nhất

* Ý tưởng thuật toán:

- Thuật toán này cho phép chúng ta tìm đường đi ngắn nhất giữa mọi cặp đỉnh.

- Nếu đỉnh k nằm trên đường đi ngắn nhất từ đỉnh i tới đỉnh j thì đoạn đường từ i tới k và từ k tới j phải là đường đi ngắn nhất từ i tới k và từ k tới j tương ứng. Do đó ta sử dụng ma trận A để lưu độ dài đường đi ngắn nhất giữa mọi cặp đỉnh.

– Ban đầu ta đặt A[i,j] = C[i,j], tức là ban đầu A chứa độ dài đường đi trực tiếp (không đi qua đỉnh nào cả).

– Sau đó thực hiện n lần lặp, sau lần lặp thứ k, ma trận A sẽ chứa độ dài đường đi ngắn nhất giữa mọi cặp đỉnh chỉ đi qua các đỉnh thuộc tập {1,2,..,k}. Như vậy, sau n lần lặp ta nhận được ma trận A chứa độ dài các đường đi ngắn nhất giữa mọi cặp đỉnh của đồ thị.

– Ký hiệu Ak là ma trận A sau lần lặp thứ k, tức là Ak[i,j] là độ dài đường đi ngắn nhất từ i đến j chỉ đi qua các đỉnh thuộc {1, 2,.., k}. Ak[i,j] được tính theo công thức như sau: Ak[i,j] = min {Ak -1[i,j], Ak-1[i,k] + Ak-1[k,j]}.

– Trong quá trình lặp ta phải lưu lại vết đường đi, tức là đường đi ngắn nhất đi qua các đỉnh nào. Khi đó ta sử dụng mảng phụ P[nxn], trong đó P[i,j] lưu đỉnh k nếu đường đi ngắn nhất từ i đến j đi qua đỉnh k. Ban đầu P[i,j]=0 với mọi i,j, vì lúc đó đường đi ngắn nhất là đường đi trực tiếp, không đi qua đỉnh nào cả.

* Thuật toán:

|  |  |
| --- | --- |
| void Floyd (int a, int b)  {  int max = tongthiethai(); {1}   |  | | --- | | for (int i=0; i<n; i++){ {n +1}  for (int j=0; j<n; j++) {n + 1 }  {  if (G[i][j]) [P]  A[i][j] = G[i][j]; {x}  else A[i][j] = max;  P[i][j] = -1; {x}  }  } |   for (int k=0; k<n; k++) {n + 1}  {  for (int i=0; i<n; i++) {n +1 }  for (int j=0; j<n; j++) {n + 1}  if (A[i][j] > A[i][k] + A[k][j])  {  A[i][j] = A[i][k] + A[k][j]; {y}  P[i][j] = k ; {y}  }  }  } |

* Độ phức tạp thuật toán:
* Số phép gán: G = 1 + 2\* + 2\*

= 1 + 2\*n(n+1)/2 + 2n(n+1)/2

= 1 + 2n2 + n = O(n2)

* Số phép so sánh:

S = (n+1) \*(n+1) + 2\*(n+1) \*(n+1) + (n+1)3 + (n+1)3

= n2 + 2n +1 + 2n2 + 4n + 2 + n3 + 3n2 + 3n +1 + n3 + 3n2 + 3n +1

= 2n3 + 9n2 + 12n + 5 = O(n3)

* Độ phức tạp của thuật toán trong trường hợp xấu nhất: O(n3)