Introduction to Data Structures and Algorithms

Design by Minh An

Email: anvanminh.haui@gmail.com

First example

- Tìm dãy con lớn nhất
  - Cho dãy số s = (a_1, a_2, …, a_n)
  - Một dãy con là s(i,j) = (a_i, a_{i+1}, …, a_j)
  - Tổng w(s(i, j)) = $\sum_{k=i}^{j} a_k$
  - Yêu cầu: Tìm dãy con có tổng lớn nhất
- Ví dụ:
  - s = (-2, 11, -4, 13, -5, 2)
  - Dãy con lớn nhất là s(2,4) = (11, -4, 13)
  - Có tổng w(s(2,4)) = 20

Direct algorithm

- Duyệt tất cả các dãy con có thể có
- Đưa ra dãy con có tổng lớn nhất

```
long algo1 (long a[], int n){
   long max = a[0];
   for (int i = 0; i < n; i ++){
      for (int j = i; j < n; j ++){
         int s = 0;
         for (int k = i; k <= j; k ++)
            s = s + a[k];
         max = max < s ? s : max ;
      }
   }
   return max ;
}
```


Direct algorithm (Faster)

- **Lưu ý:** $\sum_{k=i}^{j} a[k] = a[j] + \sum_{k=i}^{j-1} a[k]$

```
long algo2 (long a[], int n){
   long max = a [0];
   for (int i = 0; i < n; i ++){
         int s = 0;
         for (int j = i; j < n; j ++){
            s = s + a[j];
            max = max < s ? s : max ;
      }
   }
   return max ;
}
```

Recursive algorithm

- Chia dãy s thành 2 dãy con s1 và s2
- Dãy con lớn nhất có thể là dãy s1 hoặc dãy s2.

```
long maxSeq (int i, int j){
   if (i == j) return a[i];
   int m = (i+j)/2;
   long ml = maxSeq (i,m);
   long mr = maxSeq (m+1,j);
   long maxL = maxLeft (i,m);
   long maxR = maxRight (m+1,j);
   long maxLR = maxL + maxR ;
   long max = ml > mr ? ml : mr;
   max = max > maxLR ? max : maxLR ;
   return max ;
}
long algo3 (int a[], int n){
   return maxSeq (0,n -1);
}
```


Recursive algorithm

```
long maxLeft (int i, int j){
   long maxL = a[j];
   int s = 0;
   for (int k = j; k >= i; k --){
      s += a[k];
      maxL = maxL > s ? maxL : s;
   }
   return maxL ;
}
long maxRight (int i, int j){
   long maxR = a[i];
   int s = 0;
   for (int k = i; k <= j; k ++){
      s += a[k];
      maxR = maxR > s ? maxR : s;
   }
   return maxR ;
}
```

Dynamic programming – Quy hoạch động

- **Nguyên tắc**
  - **Chia bài toán thành các bài toán con, cùng dạng.**
  - **Sử dụng lời giải của các bài toán con để tìm lời giải cho bài toán ban đầu.**
  - **Tính trước lời giải của các bài toán con và lưu vào bộ nhớ (thường là một mảng).**
  - **Lấy lời giải của các bài toán con (ở trong mảng đã tính trước) để giải bài toán ban đầu.**

Dynamic programming

- **Bài toán dãy con lớn nhất**
  - **Chia:**
    - **s_i là tổng của dãy con lớn nhất của dãy ($a_1, a_2, \ldots, a_i$).**
  - **Tổng hợp kết quả:**
    - **$s_1 = a_1$**
    - **$s_i = \max\{s_{i-1} + a_i, a_i\}, \ \forall i = 2, 3, \ldots, n$**
    - **Kết quả của bài toán ban đầu là $\max\{s_1, s_2, \ldots, s_n\}$**
  - **Số phép toán cơ bản là n (*best algorithm*).**

Dynamic programming

```
long algo4 (int a[], int n){
  long max = a [0];
  int s[n];
  s[0] = a[0];
  max = s[0];
  for (int i = 1; i < n; i ++){
    if(s[i-1] > 0)
      s[i] = s[i-1] + a[i];
    else
      s[i] = a[i];
    max = max > s[i] ? max : s[i];
  }
  return max ;
}
```


Analysis of algorithms - Đánh giá thuật toán

- **algo1:** $T(n) = \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$
- **algo2:** $T(n) = \frac{n^2}{2} + \frac{n}{2}$
- **algo3:**
  - **Đếm số phép toán cộng**

$$T(n) = \begin{cases} 0 & if\ n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n & if\ n > 1 \end{cases}$$

  - **Vậy $T(n) = nlog_2 n$**
- **algo4:** $T(n) = n$

Analysis of algorithms

- **Thời gian trong trường hợp xấu nhất: thời gian chạy nhiều nhất đối với mọi bộ dữ liệu vào kích thước n.**
- **Thời gian trong trường hợp tốt nhất: thời gian chạy ít nhất đối với mọi bộ dữ liệu vào kích thước n.**
- **Thời gian trung bình: thời gian chạy trung bình của mọi bộ dữ liệu vào.**

Order of growth - Tốc độ tăng

- **Chỉ quan tâm tới cấp của hàm *T(n)***
- **Bỏ qua các hệ số**
- **Ví dụ:**

  $T(n) = an^3 + bn^2 + cn + d = \Theta(n^3)$

  Θ - Đọc là ô lớn

Asymptotic notations - Các ký hiệu tiệm cận

- **Cho hàm g(n), ta có:**
  - $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 \mid 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall n \ge n_0\}$
  - $O(g(n)) = \{f(n) : \exists c, n_0 \mid f(n) \le cg(n), \forall n \ge n_0\}$
  - $\Omega(g(n)) = \{f(n) : \exists c, n_0 \mid cg(n) \le f(n), \forall n \ge n_0\}$
- **Ví dụ:**
  - $10n^2 - 3n = \Theta(n^2)$
  - $10n^2 - 3n = O(n^3)$
  - $10n^2 - 3n = \Omega(n)$

Analysis of algorithms

- **Thử nghiệm đánh giá thuật toán**
  - **Viết chương trình biểu diễn thuật toán.**
  - **Chạy chương trình trên máy với các bộ dữ liệu vào kích thước khác nhau.**
  - **Đo thời gian thực hiện thực tế.**
  - **Ghi nhận các kết quả.**
  - **Để so sánh hai thuật toán cần sử dụng cùng môi trường phần cứng và phần mềm.**

Analysis of algorithms

❖ **Đánh giá tiệm cận**

- **Mô tả thuật toán bằng mã giả (psuedo code).**
- **Tính toán thời gian chạy thuật toán bằng một hàm của kích thước dữ liệu vào.**
- **Biểu diễn hàm tính toán với ký hiệu tiệm cận.**

Analysis of algorithms

❖ **Cấu trúc tuần tự: P và Q là 2 đoạn mã lệnh có cấu trúc tuần tự trong thuật toán.**

- ***Time (P; Q) = Time(P) + Time(Q)* hoặc**
- ***Time(P; Q) = Θ(max(Time(P), Time(Q)))***

❖ **Cấu trúc lặp for: *for i = 1 to m do P(i)***

- **t(i) là độ phức tạp về thời gian của P(i)**
- **Độ phức tạp về thời gian của cấu trúc lặp for là $\sum_{i=1}^{m} t(i)$**

❖ **Cấu trúc lặp while (repeat)**

- **Xác định một hàm của các biến trong cấu trúc lặp mà giá trị của hàm này giảm trong quá trình thực hiện lặp.**
- **Để đánh giá thời gian chạy, ta phân tích độ giảm giá trị của hàm trong quá trình thực hiện vòng lặp.**

Analysis of algorithms

❖ **Ví dụ: binary search**

```
Function BinarySearch(T[1..n], x)
begin
  i ← 1; j ← n;
  while i < j do
    k ← (i + j)/2;
    case
      x < T[k]: j ← k - 1;
      x = T[k]: i ← k; j ← k;
      exit;
      x > T[k]: i ← k + 1;
    endcase
  endwhile
end
```


Analysis of algorithms

❖ **Ví dụ: binary search**

❖ **Chứng minh:**

- *$d = j - i + 1$ (số phần tử của mảng được kiểm tra)*
- *i^*, j^*, d^* tương ứng là giá trị của i, j, d khi kết thúc lặp*

❖ **Ta có:**

- *If $x < T[k]$ then $i^* = i$, $j^* = (i + j)/2 - 1$, $d^* = j^* - i^* + 1 \leq d/2$*
- *If $x > T[k]$ then $j^* = j$, $i = (i + j)/2 + 1$, $d^* = j^* - i^* + 1 \leq d/2$*
- *If $x = T[k]$ then $d^* = 1$*

❖ **Do đó, số lần lặp của vòng lặp là $\lceil \log n \rceil$**

Master theorem – Định lý thợ

- $T(n) = aT(n/b) + cn^k$ with $a \geq 1$, $b > 1$, $c > 0$ là các hằng số.
  - If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$
  - If $a = b^k$ then $T(n) = \Theta(n^k \log n)$ with $\log n = \log_2 n$
  - If $a < b^k$ then $T(n) = \Theta(n^k)$
- Ví dụ:
  - $T(n) = 3T(n/4) + cn^2 \Rightarrow T(n) = \Theta(n^2)$
  - $T(n) = 2T(n/2) + n^{0.5} \Rightarrow T(n) = \Theta(n)$
  - $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$
  - $T(n) = T(3n/7) + 1 \Rightarrow T(n) = \Theta(\log n)$

Sorting

- Biến đổi vị trí của các đối tượng trong một danh sách theo một thứ tự nhất định.
- Việc thiết kế các thuật toán sắp xếp hiệu quả rất quan trọng đối với các thuật toán khác (tìm kiếm, trộn dữ liệu v.v…).
- Mỗi đối tượng có một khóa, danh sách các đối tượng được sắp xếp theo trật tự của khóa.
- Hai phép toán cơ bản được sử dụng trong hầu hết các thuật toán sắp xếp là:
  - Swap(a,b): đảo vị trí của hai đối tượng a, b.
  - Compare(a,b): so sánh khóa của hai đối tượng a, b.

Sorting

- **Một thuật toán sắp xếp được gọi là sắp tại chỗ nếu độ phức tạp bộ nhớ phụ là o(1) – độ phức tạp hằng số (không phụ thuộc vào kích thước của dữ liệu đầu vào).**
- **Một thuật toán sắp xếp được gọi là ổn định nếu nó duy trì trật tự tương đối của các đối tượng theo khóa.**
- **Một thuật toán sắp xếp chỉ sử dụng phép toán so sánh để xác định thứ tự của hai đối tượng gọi là sắp xếp đổi chỗ.**

Insertion Sort

- **Ở lần lặp thứ k (∀k = 1, 2, …, n-1), chèn phần tử thứ k+1 của dãy ban đầu vào vị trí thích hợp của k phần tử đầu tiên đã được sắp xếp.**
- **Kết quả: Sau lần lặp thứ k, dãy có k+1 phần tử đầu tiên được sắp xếp.**

Insertion sort

```
void insertion_sort (int a[], int n){
   int k;
   for (k = 1; k < n; k ++){
      int last = a[k];
      int j = k;
      while (j > 0 && a[j-1] > last){
         a[j] = a[j -1];
         j --;
      }
      a[j] = last ;
   }
}
```


Selection Sort

- Ở lần lặp thứ k, chọn phần tử có giá trị nhỏ nhất trong số các phần tử a[k], a[k+1], …, a[n], giả sử là a[min], đảo giá trị a[k] và a[min] (∀k = 1, 2, …, n-1)

```
void selection_sort (int a[], int n){
   for (int k = 1; k < n; k ++){
      int min = k;
      for(int i = k+1; i <= n; i++)
         if(a[min] > a[i])
            min = i;
      swap(a[k],a[min]);
   }
}
```

Bubble Sort

- Duyệt từ đầu dãy, so sánh và đảo giá trị của hai phần tử liên tiếp nhau nếu chúng trái thứ tự.
- Lặp lại cho đến khi không còn cặp nào cần đảo giá trị.

```
void bubble_sort (int a[], int n){
   bool swapped;
   do {
      swapped = false;
      for (int i = 1; i < n; i++)
         if(a[i] > a[i+1]){
            swap(a[i],a[i+1]);
            swapped = true;
         }
   }while (swapped == true);
}
```


Merge Sort

- Divide – and – conquer.
  - Chia dãy ban đầu thành 2 dãy con mỗi dãy có n/2 phần tử.
  - Gọi đệ quy sắp 2 dãy con.
  - Trộn hai dãy con để được dãy cần sắp.

Merge Sort

```
void merge(a[], bt1, w1, bt2, w2, b[]){
//bt1, bt2: là vị trí biên trái của hai vệt, w1, w2 là độ dài của hai vệt
   i=bt1; j=bt2; bp1=bt1+w1-1; bp2=bt2+w2-1; k=bt1;
//bp1, bp2 là biên phải của hai vệt, k là biên trái của vệt mới trên dãy Z
   while (i<=bp1 && j<=bp2){
     if (a[i]<a[j]) { b[k] = a[i]; i++; k++; }
     else {  b[k] = a[j]; j++; k++;}
   }
   while (i<=bp1) {
     b[k] = a[i]; i++; k++;
   }
   while (j<=bp2) {
     b[k] = a[j]; j++; k++;
   }
}
```


Merge Sort

```
void merge_pass (a[], n, k, b[]){
//1. Khởi tạo các giá trị ban đầu
   cv = n/(2*k); //Số cặp vệt
   s = 2*k*cv; //Số pt có cặp độ dài K
   r = n - s;      //Số pt lẻ cặp
//2. Trộn từng cặp vệt
   for (j=1; j<=cv; j++){
      b1 = (2*j -2)*k; //biên trái của vệt thứ nhất
      merge(a, b1, k, b1+k, k, b);
   }
//3. Chỉ còn một vệt
   if (r<=k)
      for (j=0; j<r; j++)
         b[s+j] = a[s+j];
//4. Còn hai vệt nhưng một vệt có độ dài nhỏ hơn k
   else
      merge(a, s,k, s+k, r-k, b);
}
```

Merge Sort

```
void merge_sort(a[], n)
{
//1. Khởi tạo số phần tử trong một vệt
   k =1;
//2. Sắp xếp trộn
   while (k<n)
   {
   //Trộn và chuyển các phần tử vào dãy b
      merge_pass(a, n, k, b);
   //Trộn và chuyển các phần tử trở lại dãy a
      merge_pass(b, n, 2*K, a);
      k = k*4;
   }
}
```


Quick Sort

- Chọn một phần tử gọi là chốt.
- Sắp xếp dãy sao cho:
  - Các phần tử nhỏ hơn chốt ở trước chốt.
  - Các phần tử lớn hơn chốt ở bên phải chốt.
- Gọi đệ quy với các phần tử ở trước chốt và các phần tử ở sau chốt.

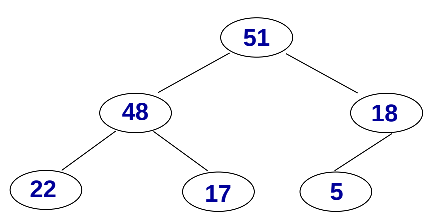
Quick Sort

```
void quick_sort(int a[],int left,int right)
{
  if (left<right)  {
      int k=(left+right)/2;
      int t=a[k];
      int i=left, j=right;
      do{
             while (a[i]<t) i=i+1;
             while (a[j]>t) j=j-1;
             if (i<=j){
                   int tg=a[i]; a[i]=a[j]; a[j]=tg;
                   i=i+1; j=j-1;
             }
      }while (i<=j);
      quick_sort(a,left,j);
      quick_sort(a,i,right);
  }
}
```


Heap Sort

❖ Khái niệm heap

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
51	48	18	22	17	5



Heap Sort

```
void heapify ( int a[], int i, int n){
   int L = 2*i+1;
   int R = 2*i+2;
   int max = i;
   if (L <= n-1 && a[L] > a[i])
      max = L;
   if (R <= n-1 && a[R] > a[max])
      max = R;
   if (max != i){
      swap (a[i],a[max]);
      heapify (a, max ,n);
   }
}
```


Heap Sort

```
void buildHeap (int a[], int n){
   for (int i = n/2-1; i >= 0; i --){
      heapify (a,i,n);
   }
}

void heap_Sort (int a[], int n){
   buildHeap (a,n);
   for (int i = n; i > 1; i --){
      swap (a[0],a[i-1]);
      heapify (a,0,i -1);
   }
}
```