Giới thiệu về Cấu Trúc Dữ Liệu và Thuật Toán

Khoa Công Nghệ Thông Tin,
Trường Đại Học Thủy Lợi.

Ngày 15 tháng 1 năm 2018

Ví dụ đầu tiên

Tìm dãy con có tổng lớn nhất của một dãy số cho trước

- Cho một dãy số $s = \langle a_1, \ldots, a_n \rangle$
- một dãy con là $s(i,j) = \langle a_i, \ldots, a_j \rangle$, $1 \leq i \leq j \leq n$
- tổng

$$w(s(i,j)) = \sum_{k=i}^{j} a_k$$

- Bài toán: tìm dãy con có tổng lớn nhất

Ví dụ

- Cho dãy số: *-2, 11, -4, 13, -5, 2*
- Dãy con có tổng lớn nhất là *11, -4, 13* (tổng của dãy con này là 20)

Thuật toán trực tiếp

- Quét tất cả các dãy con có thể $C_n^2 + n = \frac{n^2+n}{2}$
- Tính và lưu lại tổng của từng dãy con

```
public long algo1(int[] a){
  int n = a.length;
  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;
}
```

Thuật toán trực tiếp

Thuật toán nhanh hơn

- Quan sát: $\sum_{k=i}^{j} a[k] = a[j] + \sum_{k=i}^{j-1} a[k]$

```
public long algo2(int[] a){
  int n = a.length;
  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;
}
```

Chia để trị

- Chia dãy số thành 2 dãy con tại vị trí chính giữa $s = s_1 :: s_2$
- Dãy con lớn nhất có thể là
  - ở trong s_1 hoặc
  - ở trong s_2 hoặc
  - bắt đầu tại một vị trí nào đó trong s_1 và kết thúc tại một vị trí nào đó trong s_2
- Đoạn mã Java:

```
private 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;
}
public long algo3(int[] a){
  int n = a.length;
  return maxSeq(0,n-1);
}
```

Chia để trị

```
private 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;
}
private 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;
}
```

Quy hoạch động

Nguyên tắc chung

- Chia nhỏ: chia bài toán ban đầu thành các bài toán con tương tự
- Lưu trữ: lưu lại lời giải của các bài toán con trong bộ nhớ
- Kết hợp: thành lập lời giải cho bài toán ban đầu bằng cách kết hợp lời giải của các bài toán con được lưu trong bộ nhớ

Dãy con lớn nhất

- Chia nhỏ:
  - Gọi s_i là tổng của dãy con lớn nhất của $a_1, \ldots, a_i$, kết thúc tại vị trí a_i
- Kết hợp:
  - $s_1 = a_1$
  - $s_i = \max\{s_{i-1} + a_i, a_i\}, \forall i = 2, \ldots, n$
  - Lời giải cho bài toán ban đầu là $\max\{s_1, \ldots, s_n\}$
- Số phép tính cơ bản là n (thuật toán tốt nhất)

Quy hoạch động

```java
public long algo4(int[] a){
  int n = a.length;
  long max = a[0];
  int[] s = new int[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;
}
```

Phân tích thuật toán

- Tài nguyên (bộ nhớ, băng thông, CPU, ...) yêu cầu bởi thuật toán
- Quan tâm lớn nhất là thời gian tính toán
- Kích thước đầu vào: số lượng đối tượng trong đầu vào
- Thời gian chạy: được đo theo số lượng phép tính chính được thực thi

Phân tích thuật toán

- thuật toán 1: $T(n) = \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$
- thuật toán 2: $T(n) = \frac{n^2}{2} + \frac{n}{2}$
- thuật toán 3:
  - Đếm số phép cộng ("+") $T(n)$

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

  - Suy ra: $T(n) = n\ log_2 n$
- thuật toán 4: $T(n) = n$

Phân tích thuật toán

- Thời gian chạy tồi nhất: là thời gian chạy dài nhất cho bất kỳ kích thước đầu vào n
- Thời gian chạy tốt nhất: là thời gian chạy ngắn nhất với bất kỳ kích thước đầu vào n
- Thời gian chạy trung bình: phân tích xác suất (với giả thiết của một phân phối đầu vào) tạo nên thời gian chạy mong muốn

Phân tích độ tăng trưởng

- Chỉ xem xét phần tử lớn nhất của hàm
- Bỏ qua các hệ số không đổi
- Ví dụ
  - $an^3 + bn^2 + cn + d = \Theta(n^3)$

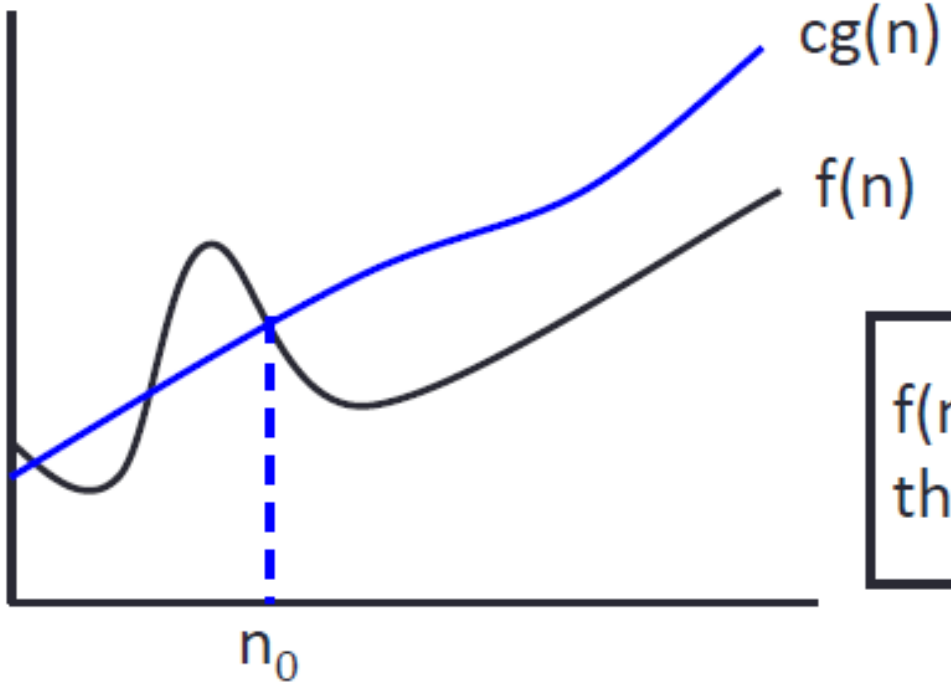
Các ký hiệu tiệm cận

- Cho một hàm $g(n)$, chúng ta ký hiệu:
  - $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$
  - $\mathcal{O}(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ s.t. } f(n) \leq cg(n), \forall n \geq n_0\}$
  - $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ s.t. } cg(n) \leq f(n), \forall n \geq n_0\}$
- Ví dụ
  - $10n^2 - 3n = \Theta(n^2)$
  - $10n^2 - 3n = \mathcal{O}(n^3)$
  - $10n^2 - 3n = \Omega(n)$

Ký hiệu Ô

$f(n) = O(g(n))$

khi và chỉ khi $\exists\ c > 0$ và $n_0 > 0$ sao cho $f(n) \leq cg(n)\ \forall n \geq n_0$

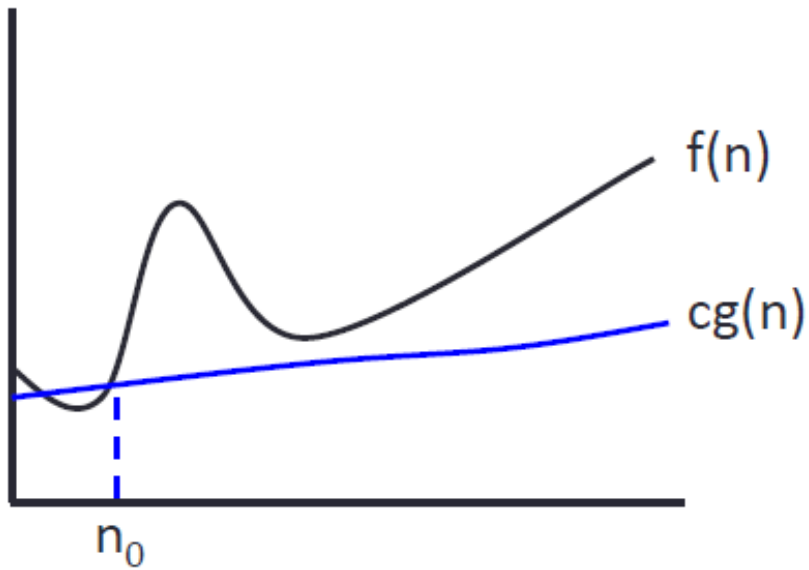


f(n) bị chặn trên bởi g(n) theo nghĩa tiệm cận

Ký hiệu Ô-mê-ga

$f(n) = \Omega(g(n))$

khi và chỉ khi $\exists\ c > 0$ và $n_0 > 0$ sao cho $cg(n) \leq f(n)\ \forall n \geq n_0$



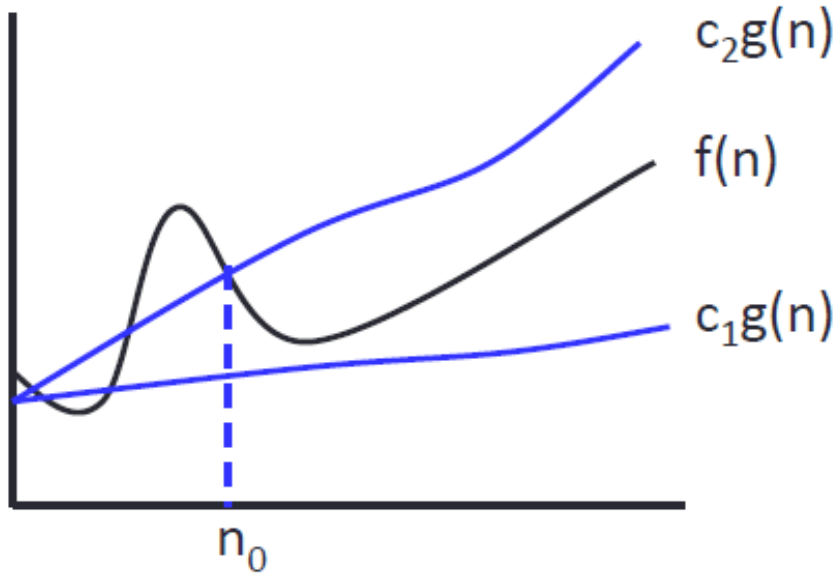
f(n) bị chặn dưới bởi g(n) theo nghĩa tiệm cận

Ký hiệu Tê-ta

$f(n) = \Theta(g(n))$

khi và chỉ khi $\exists\ c_1 > 0, c_2 > 0$ và $n_0 > 0$ sao cho

$c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0$



f(n) có cùng tốc độ tăng với g(n) theo nghĩa tiệm cận

Phân tích thuật toán

Các nghiên cứu thực nghiệm

- Viết một chương trình cài đặt thuật toán
- Chạy chương trình trên một máy tính với những kích thước đầu vào khác nhau
- Đo thời gian chạy thực tế
- Biểu diễn kết quả

Phân tích thuật toán

Điểm yếu của các nghiên cứu thực nghiệm

- Cần phải cài đặt thuật toán, đôi khi rất khó
- Kết quả không thể chỉ ra thời gian chạy của một đầu vào khác không được thực nghiệm
- Để so sánh hai thuật toán, bắt buộc phải sử dụng cùng môi trường phần cứng và phần mềm

Phân tích thuật toán

Phân tích thuật toán tiệm cận

- Sử dụng miêu tả ở mức cao của thuật toán (mã giả)
- Xác định thời gian chạy của một thuật toán như một hàm của kích thước đầu vào
- Biểu diễn hàm này với những ký hiệu tiệm cận

Phân tích thuật toán

- Cấu trúc tuần tự: P và Q là hai phân đoạn của thuật toán
  - $\text{Time}(P; Q) = \text{Time}(P) + \text{Time}(Q)$ hoặc
  - $\text{Time}(P; Q) = \Theta(max(Time(P), Time(Q)))$
- vòng lặp **for**: **for** $i = 1$ to m **do** $P(i)$
  - $t(i)$ là độ phức tạp thời gian của $P(i)$
  - độ phức tạp thời gian của vòng lặp **for** là $\sum_{i=1}^{m} t(i)$

vòng lặp **while** (**repeat**)

- Xác định một hàm cho các biến của vòng lặp sao cho hàm này giảm trong quá trình lặp
- Để đánh giá thời gian chạy, chúng ta phân tích cách mà hàm này giảm trong quá trình lặp

Ví dụ: tìm kiếm nhị phân trên một mảng đã được sắp xếp tăng dần

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

Phân tích thuật toán

Ví dụ: tìm kiếm nhị phân trên một mảng đã được sắp xếp tăng dần
Ký hiệu

- $d = j - i + 1$ (số lượng phần tử của mảng được khảo sát)
- i^*, j^*, d^* là giá trị tương ứng của i, j, d sau một vòng lặp

Chúng ta có

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

Do đó, số lần lặp của vòng lặp là $\lceil logn \rceil$

Định lý Master

$T(n) = aT(n/b) + cn^k$ với $a \geq 1, b > 1, c > 0$ là hằng số

- Nếu $a > b^k$, thì $T(n) = \Theta(n^{log_b a})$
- Nếu $a = b^k$, thì $T(n) = \Theta(n^k logn)$ với $logn = log_2 n$
- Nếu $a < b^k$, thì $T(n) = \Theta(n^k)$

Example

- $T(n) = 3T(n/4) + cn^2 \Rightarrow T(n) = \Theta(n^2)$
- $T(n) = 2T(n/2) + n^{0.5} \Rightarrow T(n) = \Theta(n)$
- $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$
- $T(n) = T(3n/7) + 1 \Rightarrow T(n) = \Theta(logn)$

Sắp xếp

- Đặt các phần tử của một danh sách theo một thứ tự nhất định
- Thiết kế những thuật toán sắp xếp hiệu quả là rất quan trọng để sử dụng cho những thuật toán khác (như *tìm kiếm, trộn, ...*)
- Mỗi đối tượng được gắn với một khóa và các thuật toán sắp xếp làm việc trên những khóa này
- Hai thao tác cơ bản được sử dụng nhiều nhất bởi những thuật toán sắp xếp
  - Swap(a, b): hoán đổi giá trị của hai biến a và b
  - Compare(a, b): trả về
    - *true* nếu a đứng trước b trong thứ tự được xem xét
    - *false*, nếu ngược lại
- Không mất tính tổng quát, giả sử chúng ta cần sắp xếp một danh sách các số theo thứ tự không giảm

- Một thuật toán sắp xếp được gọi là **in-place** nếu kích thước của bộ nhớ thêm được yêu cầu bởi thuật toán là $\mathcal{O}(1)$ (không phụ thuộc vào kích thước của mảng đầu vào)
- Một thuật toán sắp xếp được gọi là **Ổn định** nếu nó giữ nguyên thứ tự tương đối của các phần tử với những khóa bằng nhau
- Một thuật toán sắp xếp chỉ sử dụng phép so sánh để quyết định trật tự giữa hai phần tử được gọi là **Thuật toán sắp xếp dựa trên so sánh**

Sắp xếp chèn

- Ở bước lặp k, đặt phần tử thứ k của danh sách ban đầu vào đúng vị trí của danh sách đã được sắp xếp của k phần tử đầu tiên ($\forall k = 1, \ldots, n$)
- Kết quả: sau bước lặp thứ k, chúng ta có một danh sách được sắp xếp của k phần tử đầu tiên của danh sách ban đầu

```
void insertion_sort(int a[], int n){
  int k;
  for(k = 2; k <= n; k++){
    int last = a[k];
    int j = k;
    while(j > 1 && a[j-1] > last){
      a[j] = a[j-1];
      j--;
    }
    a[j] = last;
  }
}
```

Sắp xếp chọn

- Đặt phần tử nhỏ nhất của danh sách ban đầu ở vị trí đầu tiên
- Đặt phần tử nhỏ thứ hai của danh sách ban đầu ở vị trí thứ hai
- Đặt phần tử nhỏ thứ ba của danh sách ban đầu ở vị trí thứ ba
- ...

```
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]);
  }
}
```

Sắp xếp nổi bọt

- Duyệt từ vị trí ban đầu của danh sách: so sánh và đổi chỗ hai phần tử liền nhau nếu chúng không theo đúng thứ tự
- Lặp lại bước duyệt cho đến khi không còn phép đổi chỗ nữa

```
void bubble_sort(int a[], int n){
  int swapped;
  do{
    swapped = 0;
    for(int i = 1; i < n; i++)
    if(a[i] > a[i+1]){
      swap(a[i],a[i+1]);
      swapped = 1;
    }
  }while(swapped == 1);
}
```

Sắp xếp trộn

Chia để trị

- Chia danh sách ban đầu gồm n phần tử thành hai danh sách nhỏ hơn với $n/2$ phần tử
- Thực hiện đệ quy sắp xếp trộn hai danh sách này
- Trộn hai danh sách đã được sắp xếp với nhau

Sắp xếp trộn

```
void merge(int a[], int L, int M, int R){
  // merge two sorted list a[L..M] and a[M+1..R]
  int i = L;// first position of the first list a[L..M]
  int j = M+1;// first position of the second list a[M+1..R]
  for(int k = L; k <= R;k++){
    if(i > M){// the first list is all scanned
      TA[k] = a[j]; j++;
    }else if(j > R){// the second list is all scanned
      TA[k] = a[i]; i++;
    }else{
      if(a[i] < a[j]){
        TA[k] = a[i];   i++;
      }else{
        TA[k] = a[j]; j++;
      }
    }
  }
  for(int k = L; k <= R; k++)
    a[k] = TA[k];
}
```

Sắp xếp trộn

```
void merge_sort(int a[], int L, int R){
  if(L < R){
    int M = (L+R)/2;
    merge_sort(a,L,M);
    merge_sort(a,M+1,R);
    merge(a,L,M,R);
  }
}
```

Sắp xếp nhanh

- Lấy một phần tử, được gọi là **điểm chốt**, từ danh sách ban đầu
- Sắp xếp lại danh sách sao cho:
  - Tất cả phần tử nhỏ hơn **điểm chốt** sẽ đứng trước nó
  - Tất cả phần tử lớn hơn hoặc bằng **điểm chốt** sẽ đứng sau nó
- Ở đây, **điểm chốt** được cố định ở vị trí đúng của nó trong danh sách sắp xếp cuối cùng
- Đệ quy việc sắp xếp này với danh sách con ở trước **điểm chốt** và danh sách con ở sau **điểm chốt**

Sắp xếp nhanh

```
void quick_sort(int a[], int L, int R){
  if(L < R){
    int index = (L+R)/2;
    index = partition(a,L,R,index);
    if(L < index)
      quick_sort(a,L,index-1);
    if(index < R)
      quick_sort(a,index+1,R);
  }
}
```

Sắp xếp nhanh

```
int partition(int a[], int L, int R, int indexPivot){
  int pivot = a[indexPivot];
  // put the pivot in the end of the list
  swap(a[indexPivot],a[R]);
  // store the right position of pivot
  // at the end of the partition procedure
  int storeIndex = L;

  for(int i = L; i <= R-1; i++){
    if(a[i] < pivot){
      swap(a[storeIndex],a[i]);
      storeIndex++;
    }
  }
  // put the pivot in the right position
  // and return this position
  swap(a[storeIndex],a[R]);

  return storeIndex;
}
```

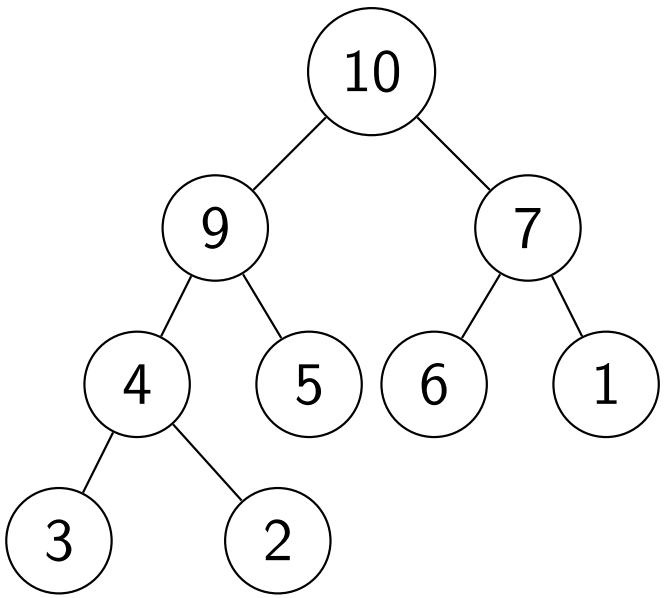
Sắp xếp vun đống

Sắp xếp một danh sách $A[1..N]$ theo thứ tự không giảm

1. Xây dựng môt đống từ $A[1..N]$
2. Loại bỏ phần tử lớn nhất và đặt nó vào vị trí thứ N của danh sách
3. Cấu trúc lại đống từ $A[1..N-1]$
4. Loại bỏ phần tử lớn nhất và đặt nó vào vị trí thứ $N-1$ của danh sách
5. ...

Sắp xếp vun đống - Cấu trúc đống

- Thuộc tính hình dạng: cây nhị phân đầy đủ với mức L
- Thuộc tính đống: mỗi nút lớn hơn hoặc bằng nút con của nó (đống-lớn-nhất)



1	2	3	4	5	6	7	8	9
10	9	7	4	5	6	1	3	2

Sắp xếp vun đống

- Đống tương ứng với một danh sách $A[1..N]$
  - Gốc của cây là $A[1]$
  - Nút con bên trái của nút $A[i]$ là $A[2 * i]$
  - Nút con bên phải của nút $A[i]$ là $A[2 * i + 1]$
  - Chiều cao là $logN + 1$
- Các thao tác
  - Xây-dựng-đống-lớn-nhất: xây dựng đống từ danh sách ban đầu
  - Chỉnh-đống-lớn-nhất: sửa cây nhị phân sau để nó trở thành Đống-lớn-nhất
    - ★ Một cây với gốc $A[i]$
    - ★ $A[i] < max(A[2 * i], A[2 * i + 1])$: thuộc tính đống không thỏa mãn
    - ★ Những cây con ở vị trí $A[2 * i]$ và $A[2 * i + 1]$ là Đống-lớn-nhất

Sắp xếp vun đống

```
void heapify(int a[], int i, int n){
  // array to be heapified is a[i..n]
  int L = 2*i;
  int R = 2*i+1;
  int max = i;
  if(L <= n && a[L] > a[i])
    max = L;
  if(R <= n && a[R] > a[max])
    max = R;
  if(max != i){
    swap(a[i],a[max]);
    heapify(a,max,n);
  }
}
```

Sắp xếp vun đống

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

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

Sắp xếp kiểu lược

Một thuật toán sắp xếp tương đối đơn giản và nhanh được thiết kế ban đầu bởi Dobosiewicz vào năm 1980. Sau đó nó được thiết kế lại bởi Lacey và Box vào năm 1991. Sắp xếp kiểu lược cải tiến trên nền sắp xếp nổi bọt.

- Sắp xếp nổi bọt: bất kỳ hai phần tử so sánh đều có khoảng cách là 1;
- Sắp xếp kiểu lược: khoảng cách có thể lớn hơn 1 và giảm xuống qua mỗi lần lặp theo yếu tố co lại: [kích thước đầu vào / yếu tố co lại, kích thước đầu vào / yếu tố co lại^2, kích thước đầu vào / yếu tố co lại^3, . . . , 1];
- Yếu tố co lại ảnh hưởng lớn đến hiệu quả của sắp xếp kiểu lược. Giá trị **1.3** được gợi ý như là một yếu tố co lại lý tưởng bởi tác giả của bài báo gốc sau khi kiểm tra thực nghiệm với trên 200,000 danh sách ngẫu nhiên.
- Bước cuối cùng của việc sắp xếp tương đương với một thuật toán nổi bọt, nhưng so với thời gian này thì một thuật toán nổi bọt sẽ hiệu quả.

Sắp xếp kiểu lược

```
1  void comb_sort(int a[], int n){
2    gap = n // Initialize gap size
3    shrink = 1.3; // Set the gap shrink factor
4    bool sorted = false;
5
6    while ((gap > 1) || (sorted == false)) {
7      // Update the gap value for a next comb
8      gap = floor(gap / shrink);
9      if (gap < 1)   gap = 1;
10     int i = 0;
11     if (gap == 1)  sorted = true;
12
13     // A single "comb" over the input list
14     while ( i + gap < n) { // See Shell sort for a similar idea
15       if (a[i] > a[i+gap]) {
16         swap(a[i], a[i+gap]);
17         if (gap == 1) sorted = false;
18          //If this assignment never happens within the loop,
19          //then there have been no swaps and the list is sorted.
20       }
21       i := i + 1
22     }
23   }
24 }
```

Cấu trúc dữ liệu

- Danh sách
- Ngăn xếp
- Hàng đợi

Danh sách

- Tập hợp những đối tượng được sắp xếp theo một thứ tự tuyến tính
- Mảng
  - Cấp phát liên tục
  - Truy cập các phần tử thông qua chỉ số
- Danh sách liên kết
  - Các phần tử không cần thiết được cấp phát liên tục
  - Con trỏ được sử dụng để liên kết các phần tử với nhau
  - Truy cập các phần tử thông qua con trỏ

Ngăn xếp

- Một danh sách có thứ tự trong đó tất cả phép chèn và xóa đều được thực hiện tại vị trí đầu tiên (được gọi là **đỉnh** ngăn xếp)
- Nguyên tắc: phần tử cuối cùng được chèn vào trong ngăn xếp phải là phần tử đầu tiên bị xóa đi (**Last-In-First-Out**)
- Các thao tác
  - Push(x, S): đẩy một phần tử x vào trong ngăn xếp S
  - Pop(S): xóa một phần tử khỏi ngăn xếp S, và trả về phần tử này
  - Top(S): trả về phần tử ở đỉnh của ngăn xếp S
  - Empty(S): trả về giá trị *true* nếu ngăn xếp S là rỗng

Hàng đợi

- Một danh sách có thứ tự trong đó thao tác chèn được thực hiện tại vị trí cuối cùng (được gọi là **đuôi**) và thao tác xóa được thực hiện tại vị trí đầu tiên (gọi là **đầu**)
- Nguyên tắc: phần tử đầu tiên được chèn vào hàng đợi phải là phần tử đầu tiên được xóa đi (**First-In-First-Out**)
- Ứng dụng: các phần tử không được xử lý ngay lập tức mà chúng phải được xử lý theo thứ tự FIFO
  - Các gói dữ liệu được lưu trữ trong hàng đợi trước khi được truyền đi trên internet
  - Dữ liệu được chuyển đổi không đồng bộ giữa hai tiến trình: bộ đệm vào ra, đường ống, ...
  - Hàng đợi máy in, hàng đợi tổ hợp phím (như khi chúng ta gõ trên bàn phím), ...

Hàng đợi

- Các thao tác
  - Enqueue(x, Q): đẩy một phần tử x vào trong hàng đợi Q
  - Dequeue(Q): xóa một phần tử khỏi hàng đợi Q, và trả về phần tử này
  - Head(Q): trả về phần tử ở vị trí đầu của hàng đợi Q
  - Tail(Q): trả về phần tử ở vị trí đuôi của hàng đợi Q
  - Empty(Q): trả lại giá trị *true* nếu hàng đợi Q là rỗng

Thư viện Java

- Danh sách
  - ArrayList (mảng động): get(int index), size(), remove(int index), add(int index, Object o), indexOf(Object o)
  - LinkedList (danh sách liên kết đôi): remove, poll, element, peek, add, offer, size
- Ngăn xếp
  - push, pop, size
- Hàng đợi
  - LinkedList
  - remove, poll, element, peek, add, offer, size
- Tập hợp (Set)
  - Tập hợp các phần tử
  - Methods: add, size, contains
- Ánh xạ (Map)
  - Ánh xạ một đối tượng (khóa) với một đối tượng khác (giá trị)
  - Phương thức: put, get, keySet

Ví dụ

```java
package week2;

import java.util.ArrayList;

public class ExampleArrayList {

  public ExampleArrayList(){
    ArrayList<Integer> L = new ArrayList();
    for(int i = 1; i <= 10; i++)
      L.add(i);
    for(int i = 0; i < L.size(); i++){
      int item = L.get(i);
      System.out.print(item + " ");
    }
    System.out.println("size of L is " + L.size());
  }
  public static void main(String[] args) {
    ExampleArrayList EAL = new ExampleArrayList();
  }
}
```

Ví dụ

```java
package week2;
import java.util.HashSet;
public class ExampleSet {

  public void test(){
    HashSet<Integer> S = new HashSet();
    for(int i = 1; i <= 10; i ++)
      S.add(i);
    for(int i: S){
      System.out.print(i + " ");
    }
    System.out.println("S.size() = " + S.size());
    System.out.println(S.contains(20));
  }
  public static void main(String[] args) {
    ExampleSet ES = new ExampleSet();
    ES.test();
  }
}
```

Ví dụ

```
package week2;
import java.util.LinkedList;
import java.util.Queue;
public class ExampleQueue {
  public static void main(String[] args) {
    Queue Q = new LinkedList();
    /*
     * Q.element(): return the head of the queue without removing it.
     * If Q is empty, then raise exception
     * Q.peek(): return the head of the queue without removing it.
     * If Q is empty, then return null
     * Q.remove(): remove and return the head of the queue.
     * If Q is empty, then raise exception
     * Q.poll(): remove and return the head of the queue.
     * If Q is empty, then return null
     * Q.add(e): add an element to the tail of Q.
     * If no space available, then raise exception
     * Q.offer(e): add an element to the tail of Q.
  * If no space available, then return false. Otherwise, return true
     */
    for(int i = 1; i <= 10; i++)  Q.offer(i);
    while(Q.size() > 0){
      int x = (int)Q.remove();
      System.out.println("Remove " + x + ", head of Q is " + Q.peek())
    }
  }
```

Ví dụ

```
package week2;

import java.util.*;
public class ExampleStack {
  public ExampleStack(){
    /*
     * S.push(e): push an element to the stack
     * S.pop: remove the element at
     *        the top of the stack and return it
     */
    Stack S = new Stack();
    for(int i = 1; i <= 10; i++)
      S.push(i + "000");
    while(S.size() > 0){
      String x = (String)S.pop();
      System.out.println(x);
    }
  }
  public static void main(String[] args) {
    ExampleStack S = new ExampleStack();
  }
}
```

Ví dụ

```java
package week2;

import java.util.HashMap;
public class ExampleHashMap {

  public ExampleHashMap(){
    HashMap<String, Integer> m = new HashMap<String, Integer>();
    m.put("abc",1);
    m.put("def", 1000);
    m.put("xyz", 100000);
    for(String k: m.keySet()){
      System.out.println("key = " + k + " map to " + m.get(k));
    }
  }
  public static void main(String[] args) {
    ExampleHashMap EHM = new ExampleHashMap();
  }
}
```

Ví dụ

```
package week2;
import java.util.Scanner;
import java.util.HashMap;
import java.io.File;
public class CountWords {
  public CountWords(String filename){
    HashMap<String, Integer> count = new HashMap<String, Integer>();
    try{
      Scanner in = new Scanner(new File(filename));
      while(in.hasNext()){
        String s = in.next();
        if(count.get(s) == null)
        count.put(s, 0);
        count.put(s, count.get(s) + 1);
      }
      for(String w: count.keySet())
        System.out.println("Word " + w + " appears " + count.get(w) +
      in.close();
    }catch(Exception ex){
      ex.printStackTrace();
    }
  }
  public static void main(String[] args) {
    CountWords  CW = new CountWords("data\\week2\\CountWords.txt");
  }
}
```

Kiểm tra biểu thức dấu ngoặc

```
private boolean match(char c, char cc){
  if(c == '(' && cc == ')') return true;
  if(c == '[' && cc == ']') return true;
  if(c == '{' && cc == '}') return true;
  return false;
}
public boolean check(String expr){
  Stack S = new Stack();
  for(int i = 0; i < expr.length(); i++){
    char c = expr.charAt(i);
    if(c == '(' || c == '{'|| c == '[')
      S.push(c);
    else{
      if(S.size() == 0) return false;
      char cc = (char)S.pop();
      if(!match(cc,c)) return false;
    }
  }
  return S.size() == 0;
}
```

Bài toán bình nước

Có hai bình nước, một bình có thể chứa được a ga lông, một bình có thể chứa được b ga lông (a, b là hai số nguyên dương). Có một máy bơm nước với lượng nước không giới hạn. Không bình nước nào được gán nhãn đánh dấu các mực nước. Bằng cách nào bạn có thể lấy được chính xác một bình nước c ga lông (c là một số nguyên dương, $c <= a$ hoặc b)

Bài toán bình nước

- Vấn đề tìm kiếm
- Kí hiệu (x, y): lượng nước trong hai bình
- Xác định các trạng thái lân cận
  - $(x, 0)$
  - $(0, y)$
  - (a, y)
  - (x, b)
  - $(a, x + y - a)$ if $x + y >= a$
  - $(x + y, 0)$ if $x + y < a$
  - $(x + y - b, b)$ if $x + y >= b$
  - $(0, x + y)$ if $x + y < b$
- Trạng thái cuối cùng: (c, y) hoặc (x, c)

Bài toán bình nước

Algorithm 1: WaterJug(a, b, c)

```
Q ← ∅;
Enqueue((a, b), Q);
while Q is not empty do
    (x, y) ← Dequeue(Q);
    foreach (x', y') ∈ neighboring states of (x, y) do
        if (x' = c ∨ y' = c) then
            Solution();
            BREAK;
        else
            Enqueue((x', y'), Q);
```

Bài toán bình nước

```
package week2;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

class Pair{
  public int x;
  public int y;
  public Pair prev;
  public Pair(int x, int y){
    this.x = x; this.y = y;
  }
  public String toString(){
    return "("+ x + "," + y + ")";
  }
}
```

Bài toán bình nước

```
public class WaterJug {

  private int a;
  private int b;
  private HashMap<Integer, Boolean> visited =
      new HashMap<Integer, Boolean>();
  private Queue Q = new LinkedList();


  private int convert(int x, int y){
    return x * (b+1) + y;
  }
  public WaterJug(int a, int b){
    this.a = a; this.b = b;
  }
  private void checkAndAdd(int x, int y, Pair p){
    int c = convert(x,y);
    if(visited.get(c) == null){
      visited.put(c, true);
      Pair q = new Pair(x,y);
      q.prev = p;
      Q.add(q);
    }
  }
```

Bài toán bình nước

```
public void solve(int e){
  Pair start = new Pair(0,0);
  start.prev = null;
  Q.add(start);
  visited.put(convert(0,0), true);
  Pair target = null;
  while(Q.size() > 0){
    Pair p = (Pair)Q.remove();
    if(p.x == e || p.y == e){
      target = p;
      break;
    }
    checkAndAdd(p.x,0,p);
    checkAndAdd(0,p.y,p);
    checkAndAdd(p.x,b,p);
    checkAndAdd(a,p.y,p);
    if(p.x + p.y <= b) checkAndAdd(0,p.x+p.y,p);
    else checkAndAdd(p.x + p.y - b,b,p);
    if(p.x + p.y <= a) checkAndAdd(p.x+p.y,0,p);
    else checkAndAdd(a,p.x+p.y-a,p);
  }
```

Bài toán bình nước

```
    if(target == null){
      System.out.println("NOT FOUND!!!!!!!!!!!!");
      return;
    }
    Pair p = target;
    Stack S = new Stack();
    while(p.prev != null){
      S.add(p);
      p = p.prev;
    }
    S.add(p);

    while(S.size() > 0){
      System.out.println(S.pop());
    }
  }
  public static void main(String[] args) {
    WaterJug WJ = new WaterJug(3,4);
    WJ.solve(2);
  }
}
```