# BÀI TẬP CHƯƠNG 1-3

Họ và tên: Lương Thiện Chí

MSSV: 1610304

## Chương I

**Câu 3:** C(n) là độ phức tạp tính toán của giải thuật, tác vụ phân tích : so sánh

a) Trường hợp phân tích: xấu nhất
   - Khi phần tử đầu tiên của mảng là phần tử lớn nhất. Tổng số thao tác so sánh của thủ tục MAXMIN chính là số lần thân vòng lặp được thực thi:
     C(n) = (n-1) + (n-1) = 2n-2

b) Trường hợp phân tích: tốt nhất
   - Mảng được sắp xếp theo thứ tự từ bé đến lớn.Nên điều kiện A[j] > max luôn đúng, do đó tổng số thao tác so sánh là C(n) = n - 1

c) Trường hợp phân tích: trung bình với n = 3
   - Mảng có 3 phần tử, gọi phần tử nhỏ nhất, phần tử trung bình, phần tử lớn nhất lần lượt là 1, 2, 3. Từ 3 số có 6 hoán vị xảy ra, ta có các trường hợp sau:

- 123: Tổng số thao tác so sánh là 1+1 = 2
- 132: Tổng số thao tác so sánh là 1+2 = 3
- 213: Tổng số thao tác so sánh là 2+1 = 3
- 231: Tổng số thao tác so sánh là 1+2 = 3
- 312: Tổng số thao tác so sánh là 2+2 = 4
- 321: Tổng số thao tác so sánh là 2+2 = 4
   - Giả sử mỗi hoán vị có xác suất đồng đều = 1/6
   - Vậy độ phức tạp trung bình của giải thuật MAXMIN là:
     C(3) = (4+4+3+3+3+2).(1/6) = 19/6

**Câu 4:**

- Thông số để thiết lập hàm: n
- Tác vụ phân tích: so sánh
- Trường hợp phân tích: xấu nhất
- Độ phức tạp của giải thuật là tổng số thao tác so sánh trong vòng lặp while
- Gọi k là số lần thực hiện phép gán ở trong vòng lặp.

- j=1, tổng số thao tác so sánh là 1
- j=j*b = 1*b = b, tổng số thao tác so sánh là 1+1 = 2
- j=j*b = b*b = $b^2$, tổng số thao tác so sánh là 2+1 = 3

  ….
- j= j*b = $b^{k-1}$*b = $b^k$, tổng số thao tác so sánh là k+1

- Điều kiện để vòng lặp thực thi là $j <= n$ ⇔ Để thoát khỏi vòng lặp thì $j > n$ => tốn 1 thao tác so sánh.
  Ta có $j<=n$ ⇔ $b^k <= n$ => $k >= \log_b(n)$.
  ⇨ Tổng số thao tác so sánh là:$C(n) = k+1 +1 = \log_b(n) + 2$
  ⇨ Độ phức tạp giải thuật là O(logn)

  **Câu 7**:

  Đặt $N = 2^n$

  $C_N = 2C_{N/2} + N + 1$

  ⇔ $C(2^n) = 2C(2^{n-1}) + 2^n + 1$

  $= 2(2C(2^{n-2}) + 2^{n-1} + 1) + 2^n + 1 = 2^2.C(2^{n-2}) + 2.2^n + 1+ 2^1$

  …..

  $= 2^nC_1 + n.2^n + (1+ 2^1 + 2^2 + … + 2^{n-1})$

  $= 0 + n.2^n + 2^n -1$

  $= (n+1).2^n – 1$

  ⇨ $C_N = (\log_2N + 1).N – 1 = N\log_2N + N – 1$

  **Câu 8:**

  $C_n = c + C_{n-1}$ ($C_1 = d$, $n > 1$)

  $= c + c + C_{n-2} = 2c + C_{n-2}$

  $= 3c + C_{n-3}$

  …

  $= (n-1).c + C_1$

  $= c(n-1) + d$

  Vậy $C_n = cn – c + d$

  **Câu 11:**

  $C_N = 4C_{N/2} + N^2$ ($N>= 2$, $C_1 = 1$)

  Đặt $N = 2^n$

  $C(2^n) = 4C(2^{n-1}) + 2^{2n} = 4C(2^{n-1}) + 4^n$

  $= 4.(4C(2^{n-2}) +4^{n-1}) + 4^n = 4^2C(2^{n-2}) + 4^n + 4^n = 4^2C(2^{n-2}) + 2.4^n$

$= 4^3C(2^{n-3}) + 3.4^n$

…

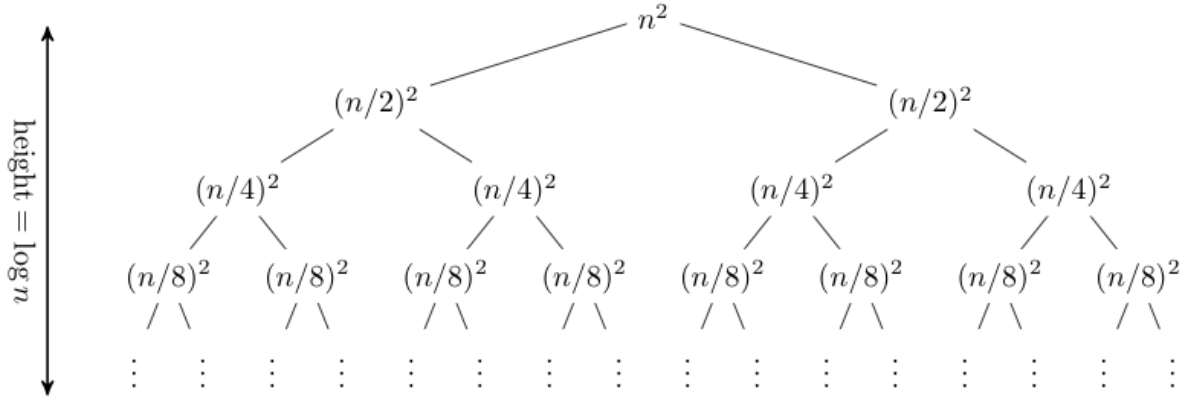
$= 4^nC(2^0) + n.4^n = 4^nC_1 + n.4^n$

$= (n+1).4^n$

⇨ $C_N = (\log_2 N + 1).N^2 = N^2\log_2 N + N$

**Câu 12:**

a) Cây đệ quy




b) $C_N = 2C_{N/2} + N^2$

Đặt $N = 2^n$

$C(2^n) = 2C(2^{n-1}) + 2^{2n}$

$= 2.(2C(2^{n-2}) + 2^{2n-2}) + 2^{2n} = 2^2C(2^{n-2}) + 2^{2n-1} + 2^{2n}$

$= 2^3C(2^{n-3}) + 2^{2n-2} + 2^{2n-1} + 2^{2n}$

…

$= 2^nC(2^0) + 2^{n+1} + 2^{n+2} + \ldots + 2^{2n-1} + 2^{2n}$

$= C_1.2^n + (1 + 2^1 + 2^2 + \ldots 2^n + 2^{n+1} + \ldots + 2^{2n-1} + 2^{2n}) - (1 + 2^1 + 2^2 + \ldots + 2^n)$

$= C_1.2^n + (2^{2n+1} - 1) - (2^{n+1} - 1)$

$= C_1.2^n + 2.2^{2n} - 2.2^n$

⇨ $C_N = C_1.N + 2N^2 - 2N$

$\sim O(N^2)$

**Chương II:**

**Câu 2:** Giải thuật QuickSort, với pivot là phần tử phải cùng (rightmost), ngôn ngữ C++

```
#include <iostream>
using namespace std;
#define swap(a, b) {int temp = a; a = b; b = temp; } // Hằng hoán vị
template <class T>  // Ép kiểu bất kì
void QuickSort(T* pD, int N) {
    if (N<2) return;
    int iP = 0; //Lựa chọn pivot là phần tử phải cùng
    int iL = 0, iR = N - 1;
    while (iL < iR) {
        while (pD[iL] < pD[iP]) iL++;
        while (pD[iR] >= pD[iP] && iR > iL) iR--;
        if (iL < iR) {
            swap(pD[iL], pD[iR]);
            if (iP == iL) iP = iR;
            else if (iP == iR) iP = iL;
        }
    }
    if (iP != iL) swap(pD[iP], pD[iL]);
    QuickSort(pD, iL);
    QuickSort(pD + iL + 1, N - iL - 1);
}
```

**Câu 13:**

a) Thuật toán Closest-pair trên  dùng để xác định cặp phần bất kì, sao cho độ chênh lệch của chúng là nhỏ nhất.

b) Độ phức tạp của giải thuật Closest-pair:

- Quicksort: nlog(n)
- Vòng lặp for j=2 to n : n-1
- Vòng lặp for j=2 to n-1: n-2

⇨ Độ phức tạp của giải thuật là : nlog(n) + 2n-3 ~ O(nlog(n))

**Chương III**

**Câu 1:**

- Giải thuật Brute-force ( Ngôn ngữ C++)

```
int MinLocation(int arr*, int N){
    int min = 0;
    for (i = 1; i < N; i++){
        if(arr[i] < arr[min]) min = i;
    }
```

```
    return min;
}
```

+ Độ phức tạp của giải thuật trên là tổng số thao tác so sánh trong vòng lặp:

⇨ $C_N = N-1$

- Giải thuật divide-conquer (Ngôn ngữ C++)

```
int MinLocation(int arr*, int N){
    if (N=1) return 1;
    else{
        int temp:= MinLocation(arr, N-1);
        if (arr[temp] <= arr[N]) return temp;
        else return N;
    }
}
```

+ Hệ thức truy hồi của giải thuật trên là : $C_N = C_{N-1} + 1$ với n>1, C

+ Độ phức tạp của giải thuật trên : O(N)

**Câu 5: Giải thuật DFS có kiểm tra xem đồ thị có chu trình hay không**

```
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], bool *rs);  // used by isCyclic()
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // to add an edge to graph
    bool isCyclic();    // returns true if there is a cycle in this graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}
```

```
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
        // Mark the current node as visited and part of recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if ( !visited[*i] && isCyclicUtil(*i, visited, recStack) )
                return true;
            else if (recStack[*i])
                return true;
        }

    }
    recStack[v] = false;  // remove the vertex from recursion stack
    return false;
}
// Returns true if the graph contains a cycle, else false
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for(int i = 0; i < V; i++)
    {
        visited[i] = false;
        recStack[i] = false;
    }
    // Call the recursive helper function to detect cycle in different
    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}
```

- Độ phức tạp của giải thuật: O(V+E)