2013147513 조영재

1. We say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. Since $d(n)$ is $O(f(n))$, there exists $c > 0$ such that $d(n) \leq cf(n)$ for $n \geq n_0$. Next, we can get $ad(n) \leq acf(n)$ by multiplying a to the both side. So, we can say $ad(n)$ is $O(f(n))$ such that $f(n)$ which has ac as coeffiient.

2. Let us write definition of big-O again, in the form of this question's literal.

$d(n) \leq c_1 f(n)$ for $n \geq n_0$, $c_1 > 0$

$e(n) \leq c_2 g(n)$ for $m \geq m_0$, $c_2 > 0$

By adding two inequality, we can get

$d(n)+e(n) \leq c_1 f(n)+c_2 g(n) \leq C(f(n)+g(n))$ for some $C \geq c_1, c_2$ and $N \geq m+n$.

So we can call $d(n)+e(n)$ is $O(f(n)+g(n))$.

3. My algorithm is

First, sort an array in descending order.

Second, select first five entries. That are the five largest elements.

In second step, selecting the first five entries is $O(1)$ because that is not related to the size of n. So this runtime it is negligible. To minimize runtime, I will use the fastest sorting algorithm, radix sort, and its runtime would be full runtime of my algorithm.

Runtime of radix sort is $O(dn)$, d means maximum length of number.

4. Input : A set array.

Output : Print subsets of the input set.

Subset of a set can be obtained by using recursive algorithm. For example, let $S=\{1,2,3\}$. Subsets of S are subsets of $S'=\{1,2\}$ and elements which are added $\{3\}$ to each subset. Next, we can obtain subsets of $S'=\{1,2\}$ in the same way. Eliminating one of element, $\{2\}$, subsets of $S'=\{1,2\}$ are the subsets of $S''=\{1\}$ and elements which are added $\{2\}$ to each subset.

In short, subsets can be categorized by two group, one is subsets without a specific element of set S, the other is subsets with a specific element of set S. The former one can be found by using recursive method, the latter one can be found by adding specific element to the each obtained subsets in the former recursive method.

subset_main.java

```java
import java.util.*;

public class subset_main {

    static int counter = 0;
    static Vector<String> subset = new Vector<String>();

    static void subset_exe(char[] arr)
    {
        if(arr.length == 1)
        {
            subset.addElement(" ");
            subset.addElement(Character.toString(arr[0]));
            print_subset();
            return;
        }

        else
        {
            char[] next_arr = Arrays.copyOfRange(arr,0,arr.length - 1);
            subset_exe(next_arr);
            add_element(arr[arr.length-1]);
            print_subset();
        }

    }

    static void print_subset()
    {
        for( ; counter<subset.size();counter ++)
        {
            System.out.println("{" + subset.get(counter)+ "}");
        }
    }

    static void add_element(char c)
    {
        for(int i=0; i<counter; i++)
            if(i==0)
                subset.addElement(Character.toString(c));
            else
                subset.addElement(subset.get(i) + "," + c);
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        char[] src = {'1','2','3','4','5'};

        subset_exe(src);
    }
}
```

5. reverse() method

```
class ListNode
{
    public String data;
    public ListNode next;

    public ListNode(){}
    public ListNode(ListNode n)
    {
        this.data = n.data;
        this.next = n.next;
    }
}

static ListNode head;

void reverse(ListNode CurrentNode)
{
    if(CurrentNode.next == null)
    {
        head = CurrentNode;
        return;
    }

    reverse(CurrentNode.next);
    CurrentNode.next.next = CurrentNode; // same as Following.next = CurrentNode;
    CurrentNode.next = null;
}
```

Input : ListNode which pointed by head pointer.

Output : Reversed singly linked list.

1. First, check if this node points null. If so, execute first if statement which is exit condition. First if statement make head pointer point current node which points null point.

2. Second, if it doesn't satisfy exit condition, call recursive method whose input is CurrentNode.next, the next node of current node.

3. If returned by recursively called method, convert next node pointer to current node and make current node pointer to null.

6. Input : An array of comparable elements.

Output : An array of value $S[i]$

1. Use one counter variable, C, initialized as 0.
2. Compare one and the next entries from the first two entries to the last two entries. In comparing, increase counter by 1 until $X[j] \leq X[i]$ satisfies. If $X[j] \leq X[i]$ does not be satisfied,
3. If $X[j] \leq X[i]$ does not be satisfied, save counter value into a stack. and set counter as 0.
4. From that point, restart counting until $X[j] \leq X[i]$ is satisfied. If $X[j] \leq X[i]$ does not be satisfied again, compare counter value and previous counter value in the stack. Select a bigger one and save the value in the stack.
5. Number of operations of the best case will be n times, and those of worst case will be $2n$ times. So big-O will be $O(n)$.