HTML Entity Parser

JAVA Code (https://leetcode.com/problems/html-entity-parser/discuss/655736/JAVA-Hashmap-Solution**)**

Approach 1: HashMap

```
class Solution {
    public String entityParser(String text) {

        HashMap<String, String> map = new HashMap<String, String>();
        map.put("&quot;", "\"");
        map.put("&apos;", "\'");
        map.put("&amp;", "&");
        map.put("&gt;", ">");
        map.put("&lt;", "<");
        map.put("&frasl;", "/");

        StringBuffer sb = new StringBuffer();
        for(int i = 0; i < text.length(); i++) {
            char ch = text.charAt(i);
            if(ch == '&') {
                for(int j = i + 1; j < text.length(); j++)
                    if(text.charAt(j) == ';') {
                        String s = text.substring(i, j + 1);
                        if(map.containsKey(s)) sb.append(map.get(s));
                        else sb.append(s);
                        i = j;
                        break;
                    }
            }
            else {
                sb.append(ch);
            }
        }

        return sb.toString();
    }
}
```

Approach 2: One Line Code

```
class Solution {
    public String entityParser(String text) {

        return text.replace("&quot;", "\"").replace("&apos;",
"'").replace("&gt;", ">").replace("&lt;", "<").replace("&frasl;",
"/").replace("&amp;", "&");
    }
}
```

CPP Code

```
class Solution {
public:
    string entityParser(string text) {
        m["&quot;"] = "\"";
        m["&apos;"] = "\'";
        m["&amp;"] = "&";
        m["&gt;"] = ">";
        m["&lt;"] = "<";
        m["&frasl;"] = "/";
        for(int i = text.size()-4;i>=0;i--){
            string sub4 = text.substr(i,4);
            string sub5 = text.substr(i,5);
            string sub6 = text.substr(i,6);
            string sub7 = text.substr(i,7);
            if(m.count(sub4)) text.replace(i,4,m[sub4]);
            if(m.count(sub5)) text.replace(i,5,m[sub5]);
            if(m.count(sub6)) text.replace(i,6,m[sub6]);
            if(m.count(sub7)) text.replace(i,7,m[sub7]);
        }
        return text;
    }
private:
    unordered_map<string,string> m;
};
```

Addition of Two Numbers

Approach 1: Elementary Math

Intuition

Keep track of the carry using a variable and simulate digits-by-digits sum starting from the head of list, which contains the least-significant digit.

Figure 1. Visualization of the addition of two numbers: 342 + 465 = 807342+465=807. Each node contains a single digit and the digits are stored in reverse order.

Algorithm

Just like how you would sum two numbers on a piece of paper, we begin by summing the least-significant digits, which is the head of l1l1 and l2l2. Since each digit is in the range of 0 \ldots 90…9, summing two digits may "overflow". For example 5 + 7 = 125+7=12. In this case, we set the current digit to 22 and bring over the carry = 1carry=1 to the next iteration. carrycarry must be either 00 or 11 because the largest possible sum of two digits (including the carry) is 9 + 9 + 1 = 199+9+1=19.

The pseudocode is as following:

- Initialize current node to dummy head of the returning list.
- Initialize carry to 00.
- Initialize pp and qq to head of l1l1 and l2l2 respectively.
- Loop through lists l1l1 and l2l2 until you reach both ends.
    - Set xx to node pp's value. If pp has reached the end of l1l1, set to 00.
    - Set yy to node qq's value. If qq has reached the end of l2l2, set to 00.
    - Set sum = x + y + carrysum=x+y+carry.
    - Update carry = sum / 10carry=sum/10.
    - Create a new node with the digit value of (sum \bmod 10)(summod10) and set it to current node's next, then advance current node to next.
    - Advance both pp and qq.
- Check if carry = 1carry=1, if so append a new node with digit 11 to the returning list.
- Return dummy head's next node.

Note that we use a dummy head to simplify the code. Without a dummy head, you would have to write extra conditional statements to initialize the head's value.

Take extra caution of the following cases:

Test case	Explanation
l1=[0,1]l1=[0,1] l2=[0,1,2]l2=[0,1,2]	When one list is longer than the other.

Test case	Explanation
l1=[]l1=[] l2=[0,1]l2=[0,1]	When one list is null, which means an empty list.
l1=[9,9]l1=[9,9] l2=[1]l2=[1]	The sum could have an extra carry of one at the end, which is easy to forget.

JAVA Code (https://leetcode.com/problems/add-two-numbers/discuss/655730/JAVA-Clean-Code-Solution**)**

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        int carry = 0;
        ListNode head = null, tail = null;
        while(l1 != null && l2 != null) {
            int sum = l1.val + l2.val + carry;
            ListNode ptr = new ListNode(sum % 10);
            carry = sum / 10;
            if(head == null) {
                head = tail = ptr;
            }
            else {
                tail.next = ptr;
                tail = ptr;
            }
            l1 = l1.next;
            l2 = l2.next;
        }

        while(l1 != null) {
            int sum = l1.val + carry;
            ListNode ptr = new ListNode(sum % 10);
            carry = sum / 10;
            if(head == null) {
                head = tail = ptr;
            }
            else {
                tail.next = ptr;
                tail = ptr;
            }
            l1 = l1.next;
        }

        while(l2 != null) {
            int sum = l2.val + carry;
            ListNode ptr = new ListNode(sum % 10);
            carry = sum / 10;
            if(head == null) {
                head = tail = ptr;
            }
            else {
                tail.next = ptr;
                tail = ptr;
            }
            l2 = l2.next;
        }

        if(carry != 0) {
            ListNode ptr = new ListNode(carry);
```

```
            if(head == null) {
                head = tail = ptr;
            }
            else {
                tail.next = ptr;
                tail = ptr;
            }
        }

        return head;
    }
}
```

CPP Code

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        int currSum = 0;
        ListNode* prev = new ListNode(-1); // one node before first node
        ListNode* curr = prev;
        while(l1 || l2) {
            int val1 = l1 ? l1->val : 0;
            int val2 = l2 ? l2->val : 0;
            currSum += (val1 + val2);
            curr->next = new ListNode(currSum % 10);
            curr = curr->next;
            currSum /= 10; // carrier
            l1 = l1 ? l1->next : nullptr;
            l2 = l2 ? l2->next : nullptr;
        }
        if(currSum) curr->next = new ListNode(1);
        return prev->next;
    }
};
```

Python Code

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode, carry: int = 0) ->
ListNode:
        val1 = l1 and l1.val or 0
        val2 = l2 and l2.val or 0
        carry, val3 = divmod(val1 + val2 + carry, 10)
        node = ListNode(val3)
        l1 = l1 and l1.next
        l2 = l2 and l2.next
        if l1 or l2 or carry:
            node.next = self.addTwoNumbers(l1, l2, carry)
        return node
```

Complexity Analysis

- Time complexity : O(\max(m, n))O(max(m,n)). Assume that mm and nn represents the length of l1l1 and l2l2 respectively, the algorithm above iterates at most \max(m, n)max(m,n) times.
- Space complexity : O(\max(m, n))O(max(m,n)). The length of the new list is at most \max(m,n) + 1max(m,n)+1.

Follow up

What if the the digits in the linked list are stored in non-reversed order? For example:

(3 \to 4 \to 2) + (4 \to 6 \to 5) = 8 \to 0 \to 7(3→4→2)+(4→6→5)=8→0→7

Warmer Temperature

Approach #1: Next Array [Accepted]

Intuition

The problem statement asks us to find the next occurrence of warmer temperatures. Because temperatures can only be in `[30, 100]`, if the temperature right now says, `T[i] = 50`, we only need to check for the next occurrence of `51`, `52`, ..., `100` and take the one that occurs soonest.

Algorithm

Let's process each `i` in reverse (decreasing order). At each `T[i]`, to know when the next occurrence of say, temperature `100` is, we should just remember the last one we've seen, `next[100]`.

Then, the first occurrence of a warmer value occurs at `warmer_index`, the minimum of `next[T[i]+1], next[T[i]+2], ..., next[100]`.

JAVA Code

```
class Solution {
    public int[] dailyTemperatures(int[] T) {
        int[] ans = new int[T.length];
        int[] next = new int[101];
        Arrays.fill(next, Integer.MAX_VALUE);
        for (int i = T.length - 1; i >= 0; --i) {
            int warmer_index = Integer.MAX_VALUE;
            for (int t = T[i] + 1; t <= 100; ++t) {
                if (next[t] < warmer_index)
                    warmer_index = next[t];
            }
            if (warmer_index < Integer.MAX_VALUE)
                ans[i] = warmer_index - i;
            next[T[i]] = i;
        }
        return ans;
    }
}
```

Python Code

```
class Solution(object):
    def dailyTemperatures(self, T):
        nxt = [float('inf')] * 102
        ans = [0] * len(T)
        for i in xrange(len(T) - 1, -1, -1):
            #Use 102 so min(nxt[t]) has a default value
```

```
            warmer_index = min(nxt[t] for t in xrange(T[i]+1, 102))
            if warmer_index < float('inf'):
                ans[i] = warmer_index - i
            nxt[T[i]] = i
        return ans
```

Complexity Analysis

- Time Complexity: O(NW)O(NW), where NN is the length of `T` and WW is the number of allowed values for `T[i]`. Since W = 71W=71, we can consider this complexity O(N)O(N).
- Space Complexity: O(N + W)O(N+W), the size of the answer and the next array.

Approach #2: Stack [Accepted]

Intuition

Consider trying to find the next warmer occurrence at `T[i]`. What information (about `T[j]` for `j > i`) must we remember?

Say we are trying to find `T[0]`. If we remembered `T[10] = 50`, knowing `T[20] = 50` wouldn't help us, as any `T[i]` that has its next warmer occurrence at `T[20]` would have it at `T[10]` instead. However, `T[20] = 100` would help us, since if `T[0]` were `80`, then `T[20]` might be its next warmest occurrence, while `T[10]` couldn't.

Thus, we should remember a list of indices representing a strictly increasing list of temperatures. For example, `[10, 20, 30]` corresponding to temperatures `[50, 80, 100]`. When we get a new temperature like `T[i] = 90`, we will have `[5, 30]` as our list of indices (corresponding to temperatures `[90, 100]`). The most basic structure that will satisfy our requirements is a *stack*, where the top of the stack is the first value in the list, and so on.

Algorithm

As in *Approach #1*, process indices `i` in descending order. We'll keep a `stack` of indices such that `T[stack[-1]] < T[stack[-2]] < ...`, where `stack[-1]` is the top of the stack, `stack[-2]` is second from the top, and so on; and where `stack[-1] > stack[-2] > ...`; and we will maintain this invariant as we process each temperature.

After, it is easy to know the next occurrence of a warmer temperature: it's simply the top index in the stack.

Here is a worked example of the contents of the `stack` as we work through `T = [73, 74, 75, 71, 69, 72, 76, 73]` in reverse order, at the end of the loop (after we add `T[i]`). For clarity, `stack` only contains indices `i`, but we will write the value of `T[i]` beside it in brackets, such as `0 (73)`.

- When `i = 7, stack = [7 (73)]. ans[i] = 0.`
- When `i = 6, stack = [6 (76)]. ans[i] = 0.`
- When `i = 5, stack = [5 (72), 6 (76)]. ans[i] = 1.`
- When `i = 4, stack = [4 (69), 5 (72), 6 (76)]. ans[i] = 1.`
- When `i = 3, stack = [3 (71), 5 (72), 6 (76)]. ans[i] = 2.`
- When `i = 2, stack = [2 (75), 6 (76)]. ans[i] = 4.`
- When `i = 1, stack = [1 (74), 2 (75), 6 (76)]. ans[i] = 1.`
- When `i = 0, stack = [0 (73), 1 (74), 2 (75), 6 (76)]. ans[i] = 1`

CPP Code

```
vector<int> dailyTemperatures(vector<int>& T) {
        int n = T.size();
        stack<int> s;
        vector<int> ans(n, 0);
        for (int i = 0; i < n; ++i) {
                while (!s.empty() and T[s.top()] < T[i]) {
                        int j = s.top(); s.pop();
                        ans[j] = i - j;
                }
                s.push(i);
        }
        return ans;
}
```

JAVA Code

```
class Solution {
   public int[] dailyTemperatures(int[] T) {
      int[] ans = new int[T.length];
      Stack<Integer> stack = new Stack();
      for (int i = T.length - 1; i >= 0; --i) {
         while (!stack.isEmpty() && T[i] >= T[stack.peek()]) stack.pop();
         ans[i] = stack.isEmpty() ? 0 : stack.peek() - i;
         stack.push(i);
      }
      return ans;
   }
}
```

Python Code

```
class Solution(object):
    def dailyTemperatures(self, T):
        ans = [0] * len(T)
        stack = [] #indexes from hottest to coldest
        for i in xrange(len(T) - 1, -1, -1):
            while stack and T[i] >= T[stack[-1]]:
                stack.pop()
            if stack:
                ans[i] = stack[-1] - i
            stack.append(i)
        return ans
```

Complexity Analysis

- Time Complexity: O(N)O(N), where NN is the length of `T` and WW is the number of allowed values for `T[i]`. Each index gets pushed and popped at most once from the stack.
- Space Complexity: O(W)O(W). The size of the stack is bounded as it represents strictly increasing temperatures.

Reverse Nodes in K Groups

Required Knowledge : Linked Lists

Time Complexity : O(N)

Approach :

- Iterate the linked list from head till node and while iterating keep pushing the elements into stack. Once the stack size reaches k, pop elements one by one and keep track of the previously popped node. Point the next pointer of prev node to top element of stack. While doing these process be careful to handle head node and tail node.
- These approach has O(N) time complecity and O(K) space complexity.
- Space complexity could be further improved by not using a stack and directly reversing k group element by changing their next pointer one by one.
- **JAVA Code: (**https://leetcode.com/problems/reverse-nodes-in-k-group/discuss/655715/JAVA-Stack-Solution)

class Solution {
public ListNode reverseKGroup(ListNode head, int k) {

```
    ListNode ptr = head;
  while(ptr != null) {
          ListNode temp = ptr;
          Stack<Integer> stack = new Stack<>();
         int i = 0;
          for(; i < k && temp != null; i++) {
                  stack.push(temp.val);
                  temp = temp.next;
          }
         if(i != k) break;
          while(!stack.isEmpty()) {
                  ptr.val = stack.pop();
                  ptr = ptr.next;
          }
  }

  return head;

}
```

}

Python Code

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        #if the list is empty
        if not head:
            return
        #finding total length of list
        cur = head
        length = 0
        while cur:
            length += 1
            cur = cur.next

        # returning head if k is greater than total length of the
list
        if k > length :
            return head

        #reversing the first k nodes seperately and assigning head
        prev = None
        cur = head
        count = 1
        while cur and count <= k:
            temp = cur.next
            cur.next = prev
            prev = cur
            cur = temp
            count += 1
        temp1 = head # temporary variable to assign prev of next
reversing
        head.next = cur
        head = prev
        prev = temp1

        # reversing next subsequent k nodes
        limit = k+k # to keep track of reversing within length of the
list
        while limit <= length:
            dummy_prev = prev
            dummy_cur = cur
            count = 1
            while cur and count <= k:
                temp = cur.next
                cur.next = prev
                prev = cur
                cur = temp
                count += 1
            dummy_prev.next = prev
            dummy_cur.next = cur
            prev = dummy_cur
            limit += k

        return head
```

CPP Code:

```
ListNode* reversethis(ListNode* head)
   {
       if(head==NULL)return NULL;
       ListNode* curr=head;
       ListNode* prev=NULL;
       ListNode* nxt=NULL;
       while(curr)
       {   nxt=curr->next;
           curr->next=prev;
           prev=curr;
           curr=nxt;
       }
       return prev;
   }
   ListNode* reverseKGroup(ListNode* head, int k) {
       if(head==NULL)return NULL;
       ListNode* root=head;
       ListNode* prev=NULL;
       while(root)
       {
          ListNode* start=root;
          ListNode* t1=start;
          int cnt=0;
           while(cnt<k-1)
           {
               if(t1)t1=t1->next;
               else return head;
               cnt++;
           }
           if(t1==NULL)return head;
           ListNode* t2=t1->next;
           t1->next=NULL;
           ListNode* t3=reversethis(start);
           if(prev==NULL)
           {
               head=t3;
           }
           else
           {
               prev->next=t3;
           }
           ListNode* t4=NULL;
           if(prev==NULL)
           {
               t4=head;
           }
           else
           {
               t4=prev;
           }
```

```
            while(t4->next)
            {
                t4=t4->next;
            }
            t4->next=t2;
            prev=t4;
            root=t4->next;
        }
        return head;
    }
```