Contents

1 Trees

1.1 Fenwick trees

Provides efficient methods for calculation and manipulation of prefix sums. Excels over **Segment Trees** on ease of coding and memory usage. Yet there are problems for which the Segment tree is a better choice.

```
//Lowbit function. Helps determine dependent and responsible nodes.
int lb(int i){
    return (i & -i);
}
//Get the prefix sum [1 - idx]. If sum [i , j] is necessary,
//use as get(j) - get(i-1)
int get(int *tree, int idx){
    int s = 0;
    for(int i=idx;i;i-=lb(i)) s += tree[i];
    return s;
}
//Update a value on the tree and let responsible nodes
//know about the change
void update(int *tree, int idx, int val){
    for(int i=idx;i<M;i+=lb(i))tree[i] += val;
}
```

1.1.1 Field test: Interval Product

Source:*Latin America Regional Contest - 2012*
Up to 10^5 numbers are given. One must update any value or answer if the product between all numbers in any interval $[i, j]$ is positive, negative or zero.

```
#include<iostream>
#include<cstring>

using namespace std;

const int M = 10050;
int v[M],n[M],z[M];

int lb(int i){
    return (i & -i);
}

```

```
13 int get(int *tree, int idx){
14     int s = 0;
15     for(int i=idx;i;i-=lb(i)) s += tree[i];
16     return s;
17 }
18
19 void update(int *tree, int idx, int val){
20     for(int i=idx;i<M;i+=lb(i))tree[i] += val;
21 }
22
23
24 int main(){
25     //freopen("int.in" , "r", stdin);
26
27     int N,K;
28     while(cin>>N>>K){
29         memset(n, 0, sizeof(n));
30         memset(z, 0, sizeof(n));
31         for(int i=1;i<=N;i++){
32             cin>>v[i];
33             if(v[i] < 0)
34                 update(n, i, 1);
35             else if(!v[i]) update(z,i,1);
36         }
37         char o;
38         int a,b;
39         for(int i=0;i<K;i++){
40             cin>>o>>a>>b;
41             if(o == 'C'){
42                 if(v[a] < 0)
43                     update(n,a,-1);
44                 else if(!v[a]) update(z,a,-1);
45                 v[a] = b;
46                 if(v[a] < 0)update(n,a,1);
47                 else if(!v[a]) update(z,a,1);
48             }
49
50             else{
51                 int zz = get(z,b) - get(z,a-1);
52                 int nn = get(n,b) - get(n,a-1);
53                 cout << (zz?'0':(nn & 1)?'-':'+');
54             }
55         }
```

```
56          cout << endl;
57      }
58
59      return 0;
60  }
```

1.2 Tries

A Trie (*its name is an infix of re**trie**val*) is a powerful structure that can find or insert strings in $O(L)$ time where L is the length of the string.
I know, it's a shitty intro but Tries are a really useful data structure that allows answering important questions about the nature of a dictionary. They're usually the answer to questions as *how to implement autocompletion on a web browser?* or *given a misspelled word, what's the most probable word that could have been intended?*

```
1   int n; //How many nodes do we have so far?
2   struct Trie{
3       int words, prefix;
4       int e[26]; //links to nodes by characters of the alphabet
5       Trie():words(0),prefix(0){memset(this->e, 0, 26*sizeof(int));}
6   };
7
8   Trie tree[1048576]; //Why 2^20? Why not?
9
10  void add(Trie &t, const string &word, int index = 0){
11      if(index == word.size())
12          t.words++,t.prefix++;
13      else{
14          t.prefix++;
15          int k = word[index] - 'a';
16          if(t.e[k] == 0){
17              t.e[k] = ++n;
18              tree[n] = Trie();
19          }
20          add(tree[t.e[k]], word, index+1);
21      }
22  }
23
24  bool present(const Trie &t, const string &w, int index=0){
25      if(index == w.size())
26          return t.words;
27      int k = w[index] - 'a';
28      if(!t.e[k])
```

```
29          return false;
30      return present(tree[t.e[k]], w, index+1);
31  }
```

1.2.1 Field test: Cellphone Typing

Source:*Latin America Regional Contest - 2012*
A set of words is given as the dictionary of a cellphone auto-complete algorithm. Return the average number of keystrokes a user must use to write the words in the dictionary.

```
1   #include<iostream>
2   #include<cstring>
3   #include<iomanip>
4   #include<vector>
5
6   using namespace std;
7
8   int n;
9   struct Trie{
10      int words, prefix;
11      int e[26];
12      Trie():words(0),prefix(0){memset(this->e, 0, 26*sizeof(int));}
13  };
14
15  Trie tree[1048576];
16
17  void add(Trie &t, const string &word, int index = 0){
18      if(index == word.size())
19          t.words++,t.prefix++;
20      else{
21          t.prefix++;
22          int k = word[index] - 'a';
23          if(t.e[k] == 0){
24              t.e[k] = ++n;
25              tree[n] = Trie();
26          }
27          add(tree[t.e[k]], word, index+1);
28      }
29  }
30
31  int strokes(const Trie &t, const string &word, int index=0, int q = 0){
32      if(index == word.size())
```

```
        return q;
    int k = word[index] - 'a';
    if(t.prefix == tree[t.e[k]].prefix){
        return strokes(tree[t.e[k]], word, index+1, q);
    }
    else{
        return strokes(tree[t.e[k]], word, index+1, q+1);
    }
}

int main(){
    freopen("cell.in", "r" , stdin);
    Trie t;
    for(int z;cin>>z;){
        n = 0;
        memset(t.e,0,sizeof(t.e));
        vector<string> w(z);
        for(int i=0;i<z;i++){
            cin>>w[i];
            add(t,w[i]);
        }

        int ans = 0;
        for(int i=0;i<w.size();i++)
            ans += strokes(t,w[i]);
        cout << fixed << setprecision(2) << ans/(double)z<<endl;
    }
    return 0;
}
```

1.3 Segment Trees

A Segment Tree is a heap-like data structure that allows update/query operations in logarithmical time. They can help solve a wider range of problems than Fenwick Trees but use more memory (about 4 times more) and take a little more time to be coded.

```
/*
    Segment Tree implementation.
    Remember to call all functions with 0-based indexes!
    Did I say you should call all functions with 0
    (ZERO ZEEEERO) based indexes?
*/

```

```
8  #define M 100000
9  #define left(x) (2*x+1)
10 #define right(x) (2*x+2)
11
12 int a[M];
13 int tree[4*M + 1]; //Yup, no lg's or anything. 4*M.
14
15 void init(int node, int l, int r){
16     if(l == r){
17         tree[node] = a[l];
18         return;
19     }
20     int m = (l + r)>> 1;
21     init(left(node), l, m);
22     init(right(node), m+1,r);
23
24     tree[node] = tree[left(node)] * tree[right(node)];
25 }
26
27 int query(int node, int l, int r, int p, int q){
28     if(q < l || r < p) return 1; //The identity value.
29     if(p<=l && r <= q)
30         return tree[node];
31     int m = (l + r)>> 1;
32     return query(left(node), l, m, p, q) * query(right(node), m+1, r, p, q);
33 }
34
35 void update(int node, int l, int r, int p, int val){
36     if(p < l || r < p)return;
37     if(l == r){
38         tree[node] = val;
39         return;
40     }
41     int m = (l + r) >> 1;
42     update(left(node), l, m, p, val);
43     update(right(node), m+1, r, p, val);
44     tree[node] = tree[left(node)] * tree[right(node)];
45 }
```

1.3.1 Field test: Interval Product

Source:*Latin America Regional Contest - 2012*
The same problem solved in 1.1.1 with Fenwick Trees but using Segment Trees.

Up to 10^5 numbers are given. One must update any value or answer if the product between all numbers in any interval $[i, j]$ is positive, negative or zero.

```
#include<cstdio>
#include<iostream>
#include<vector>
#include<cmath>

using namespace std;

#define M 100000
#define left(x) (2*x+1)
#define right(x) (2*x+2)

int a[M];
int tree[4*M + 1];

void init(int node, int l, int r){
    if(l == r){
        tree[node] = a[l];
        return;
    }
    int m = (l + r)>> 1;
    init(left(node), l, m);
    init(right(node), m+1,r);

    tree[node] = tree[left(node)] * tree[right(node)];
}

int query(int node, int l, int r, int p, int q){
    if(q < l || r < p) return 1; //The identity value.
    if(p<=l && r <= q)
        return tree[node];
    int m = (l + r)>> 1;
    return query(left(node), l, m, p, q) * query(right(node), m+1, r, p, q);
}

void update(int node, int l, int r, int p, int val){
    if(p < l || r < p)return;
    if(l == r){
        tree[node] = val;
        return;
    }
    int m = (l + r) >> 1;
```

```
42      update(left(node), l, m, p, val);
43      update(right(node), m+1, r, p, val);
44      tree[node] = tree[left(node)] * tree[right(node)];
45  }
46
47  int f(int val){
48      return val < 0 ? -1 : val != 0;
49  }
50
51  int main(){
52      freopen("i.in" , "r" , stdin);
53      //freopen("i.out" ,"w" , stdout);
54      for(int n,k;cin>>n>>k;){
55          for(int i=0,x;i<n;i++){
56              cin>>x;
57              a[i] = f(x);
58          }
59          init(0,0,n-1);
60          for(int i=0;i<k;i++){
61              char o;
62              int p,q;
63              cin>>o>>p>>q;
64              if(o =='C')
65                  update(0,0,n-1,p-1,f(q));
66              else{
67                  int x = query(0,0,n-1,p-1,q-1);
68                  cout << (x==0?'0':x<0?'-':'+');
69              }
70          }
71          cout << endl;
72      }
73
74  }
```

1.4 Huffman Trees

Huffman coding is a compression algorithm that takes into account the frequency or probability of appearance of each symbol to create a space efficient compression scheme.

1.4.1 Field test: Entropy

Source:*Live Archive - 2088*
Determine the compression ratio between an ASCII encoded string (8bits per character) and an optimal compression (via huffman coding).

```
#include<iostream>
#include<queue>
#include<string>
#include<cstdio>

using namespace std;

struct Node{
    int id,freq,parent;
    bool operator<(const Node &n) const{
        return this->freq > n.freq;
    }
    Node(){id = freq = parent = 0;}
    friend ostream& operator<<(ostream &o, const Node &n){
        o << "[" << (n.id) << ", " << (n.freq) << "]";
    }
};

int main(){

    freopen("p10.in", "r", stdin);

    string str;
    while(cin>>str && str!="END"){
        Node node[100];
        for(int i=0;i<str.size();i++){
            int x = str[i] == '_' ? 26 : str[i] - 'A';
            node[x].freq++;
            node[x].id = node[x].parent = x;
        }
        priority_queue<Node> Q;
        for(int i=0;i<27;i++){
            if(node[i].freq)
                Q.push(node[i]);
        }

        if(Q.size() == 1){
            printf("%d %d %.1f\n", str.size() * 8, str.size(), 8.0);
```

```
            continue;
        }

        int n = 27;

        while(Q.size() > 1){
            Node u,v;
            u = Q.top();
            Q.pop();
            v = Q.top();
            Q.pop();
            node[n].freq = u.freq + v.freq;
            node[n].id = node[n].parent = n;
            node[u.id].parent = node[v.id].parent = n;
            Q.push(node[n++]);
        }

        int ans = 0;

        for(int i=0;i<27;i++){
            if(node[i].freq){
                int depth = 0;
                Node u = node[i];
                while(u.parent != u.id){
                    u = node[u.parent];
                    depth++;
                }
                ans+=node[i].freq * depth;
            }
        }

        printf("%d %d %.1f\n" , 8 * str.size(), ans, 8.0 * str.size() / ans);
    }
    return 0;
}
```