Team Notebook: UPF Programming Force

Contents

1 Data Structures

1.1 Binary Indexed Tree

```
/* Binary indexed tree. Supports cumulative sum queries in O(log n) */
#define N (1<<18)
LL bit[N];

void add(LL* bit,int x,int val) {
    for(; x<N; x+=x&-x)
        bit[x]+=val;
}

LL query(LL* bit,int x) {
    LL res=0;
    for(;x;x-=x&-x)
        res+=bit[x];
    return res;
}
```

1.2 Square Root Trick

```
/* Partitions an array in sqrt(n) blocks of size sqrt(n) to support
 * O(sqrt(n)) range sum queries, O(sqrt(n)) range sum updates, and O(1)
 * point updates */
void update(LL *S, LL *A, int i, int k, int x) {
    S[i/k] = S[i/k] - A[i] + x;
    A[i] = x;
}

LL query(LL *S, LL *A, int lo, int hi, int k) {
        int sum=0, i=lo;
        while((i+1)%k != 0 && i <= hi)
                sum += A[i++];
        while(i+k <= hi)
                sum += S[i/k], i += k;
        while(i <= hi)
                sum += A[i++];
        return sum;
}
```

2 Dynamic Programming

2.1 TSP

```
// TSP in O(n^2 * 2^n). Subset is bitmask, Cost is cost.
Cost distances[N][N], tsp_memoize[1 << (N+1)][N], const
    sentinel=-0x3f3f3f3f;
#define TSP(subset, i) (tsp_memoize[subset][i] == sentinel ? tsp(subset,
    i) : tsp_memoize[subset][i])
```

```
Cost tsp(const Subset subset, const int i) {
        Subset without = subset ^ (1 << i);
        Cost minimum = numeric_limits<Cost>::max();
        for(int j=0; j<n_operas; j++) {
                if(j==i || (without & (1 << j)) == 0)
                        continue;
                Cost v = TSP(without, j);
                v += distances[i][j];
                if(v < minimum)
                        minimum = v;
        }
        return tsp_memoize[subset][i] = minimum;
}

v = tsp_memoize[1<<i][i] = v - price_save[i];
for(int i=0; i<n_operas; i++)
        tsp(0xffff >> (16 - n_operas), i);
```

3 Graphs

3.1 Maximum Bipartite Matching

```
/* Input: VVI with 1 if connected, 0 if not. mr and mc have the matches
 * for each side. From Stanford University's notebook. */
typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
    for (int j = 0; j < w[i].size(); j++) {
        if (w[i][j] && !seen[j]) {
            seen[j] = true;
            if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
                mr[i] = j;
                mc[j] = i;
                return true;
            }
        }
    }
    return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
    mr = VI(w.size(), -1);
    mc = VI(w[0].size(), -1);

    int ct = 0;

    for (int i = 0; i < w.size(); i++) {
        VI seen(w[0].size());
        if (FindMatch(i, w, mr, mc, seen)) ct++;
    }
    return ct;
}
```

4 Math

4.1 Complex Numbers

```
// Complex number class, from Stanford's Notebook. Required for FFT
struct cpx {
        cpx(){}
        cpx(double aa):a(aa){}
        cpx(double aa, double bb):a(aa),b(bb){}
        double a, b;
        double modsq(void) const { return a * a + b * b; }
        cpx bar(void) const { return cpx(a, -b); }
};
cpx operator +(cpx a, cpx b) { return cpx(a.a + b.a, a.b + b.b); }
cpx operator *(cpx a, cpx b) {
        return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}
cpx operator /(cpx a, cpx b) {
        cpx r = a * b.bar();
        return cpx(r.a / b.modsq(), r.b / b.modsq());
}
cpx EXP(double theta) { return cpx(cos(theta),sin(theta)); }
```

4.2 FFT

```
// from Stanford's notebook:
    https://web.stanford.edu/~liszt90/acm/notebook.html
```

```
// in:    input array
// out:   output array
// step:  {SET TO 1} (used internally)
// size:  length of the input/output {MUST BE A POWER OF 2}
// dir:   either plus or minus one (direction of the FFT)
// RESULT: out[k] = \sum_{j=0}^{size - 1} in[j] * exp(dir * 2pi * i * j *
    k / size)
const double two_pi = 4 * acos(0);
void FFT(cpx *in, cpx *out, int step, int size, int dir)
{
  if(size < 1) return;
  if(size == 1)
  {
    out[0] = in[0];
    return;
  }
  FFT(in, out, step * 2, size / 2, dir);
  FFT(in + step, out + size / 2, step * 2, size / 2, dir);
  for(int i = 0 ; i < size / 2 ; i++)
  {
    cpx even = out[i];
    cpx odd = out[i + size / 2];
    out[i] = even + EXP(dir * two_pi * i / size) * odd;
    out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) / size)
        * odd;
  }
}
```