Índice

BGL-UBA - Reference

1. Cosas

1.1. Segment Tree (Guty)

```
// Nodo del segment tree
struct Nodo{
  tint x;
  Nodo (tint xx)
    x = xx;
};
// Operacion del segment tree : tiene que ser ASOCIATIVA
Nodo op (Nodo n1, Nodo n2){
  return Nodo(n1.x+n2.x);
}
vector<Nodo> buildSegTree (vector<Nodo> &v ){
  // Completa el tamanho
  tint k = 4, n = v.size();
  while (k < 2*n)
    k *= 2;
  // Rellena las hojas
  vector<Nodo> seg (k, Nodo(0));
  forn(i,n)
    seg[k/2+i] = v[i];
  // Completa los padres
  while (k > 0) {
    seg[(k-1)/2] = op(seg[k-1],seg[k-2]);
    k -= 2;
  }
  return seg;
}
// i es el indice de [0,n) en el arreglo original
// Nodo es lo que queremos poner ahora como hoja
void update(tint i, Nodo nodo,vector<Nodo> &seg){
  tint k = seg.size()/2 + i;
  seg[k] = nodo;
  while (k > 0){
    seg[k >> 1] = op(seg[k],seg[k^1]);
    k /= 2;
  }
}
Nodo queryAux(tint k, tint l, tint r, tint i, tint j, vector<Nodo> &seg){
  if (i <= l && r <= j)
    return seg[k];
```

```
  if (r <= i or l >= j)
    return 0; // Aca va el NEUTRO de la funcion "op"
  Nodo a = queryAux(2*k,l,(l+r)/2,i,j,seg);
  Nodo b = queryAux(2*k+1,(l+r)/2,r,i,j,seg);
  return op(a,b);

}
// i,j son los indices del arreglo del que se hace la query
// la query se hace en [i,j)
Nodo query(tint i, tint j, vector<Nodo> &seg){
  return queryAux(1,0,seg.size()/2,i,j,seg);
}
int main(){
  tint n = 15;
  vector<Nodo> v (n, Nodo(0));
  forn(i,n)
    v[i] = Nodo((3*(i+1))%7 - 9*(i-4)%13);
  vector<Nodo> seg = buildSegTree(v);
  imprimirVector(v);
  cout << query(3,11,seg).x << "\n";
  return 0;
}
```

1.2. Inverso modular Primo y PotLog (Guty)

```
const tint nmod = 1000000007; // o el primo que deseamos

vector<tint> desBaseB (tint n, tint b){ // Calcula el desarrollo de n en base b
  if (n == 0)
    return {0};
  vector<tint> des;
  while (n > 0){
    des.push_back(n%b);
    n /= b;
  }
  reverse(des.begin(),des.end());
  return des;
}
tint potLogMod (tint x, tint y){ // Calcula: (x^y) mod nmod
  tint ans = 1;
  while (y > 0){
    if (y % 2)
      ans = (x * ans) % nmod;
    x = (x * x) % nmod;
    y /= 2;
  }
  return ans;
}
tint invLog(tint a){// Solo funciona si nmod es primo y devuelve un numero b tal
     que: (a*b) = 1 mod nmod
  return potLogMod(a,nmod-2);
}
```

1.3. Dijkstra (Guty)

```
const int INF = 1000000000; // Aca va una cota que funque para el problema en
    vez de 1000000000
struct Arista{
  tint v1,v2,peso;
  Arista(tint vv1, tint vv2, tint pp){
    v1 = vv1;
    v2 = vv2;
    peso = pp;
  }
};
bool operator < (Arista a1, Arista a2){
  return make_tuple(a1.peso,a1.v1,a1.v2) > make_tuple(a2.peso,a2.v1,a2.v2);
}
vector<tint> dijkstra (vector<vector<tint> > &ladj, vector<vector<tint> > &w,
    tint s){ // Devuelve un vector d, tal que d[v] es la minima distancia de "s
    " a "v"
  tint n = ladj.size();
  vector<tint> d (n,INF);
  priority_queue<Arista> v;
  d[s] = 0;
  for(auto vecino : ladj[s])
    v.push(Arista(s,vecino,w[s][vecino]));
  while (!v.empty()){
    if (d[v.top().v2] == INF or d[v.top().v1] + v.top().peso <= d[v.top().v2]){
      Arista e = v.top();
      v.pop();
      d[e.v2] = d[e.v1] + e.peso;
      for(auto vecino : ladj[e.v2])
        v.push(Arista(e.v2,vecino,w[e.v2][vecino]));
    }else
      v.pop();
  }
  return d;
}
```

1.4. Edmonds Karp (Guty)

```
// Hay que tener definidas de antemano:
```

```
// capacidades: Una matriz que en el lugar (i,j) guarda la capacidad que une al nodo i con j.
// ladj: Para cada nodo tiene la lista de vecinos. Notar que al comenzar el codigo se agregan las aristas que faltan para la red residual
// flow: Se debe dar un flujo inicial de 0
// flowPath: Guarda en el lugar (i,j) la cantidad de flujo que efectivamente pasa por la arista que une i con j. Inicialmente debe ser una matriz de ceros.

const tint INF = 999999999999;

tint maxFlow (vector<vector<tint> > &capacidades, vector<vector<tint> > &ladj, tint qNodos, tint source, tint terminal){
  tint flow = 0;
  vector<vector<tint> > flowPath (qNodos, vector<tint> (qNodos,0));
  tint capacityFound = -1;
  forn(i, ladj.size())
    for (auto &a : ladj[i])
      if (capacidades[i][a] != 0)
        ladj[a].push_back(i);
  while (capacityFound != 0){
    vector<tint> path (qNodos,-1);
    // Aca empieza el bfs

    path[source] = -2;
    capacityFound = 0;
    vector<tint> pathCapacity (qNodos,INF); // Aca va una cota para el flujo del problema
    deque<tint> visit = {source};
    while (!visit.empty()){
      tint actual = visit.front();
      visit.pop_front();
      for (auto vecino : ladj[actual]){
        if (capacidades[actual][vecino] > flowPath[actual][vecino] && path[vecino] == -1 ){
          path[vecino] = actual;
          pathCapacity[vecino] = min(pathCapacity[actual],capacidades[actual][vecino] - flowPath[actual][vecino]);
          if (vecino != terminal)
            visit.push_back(vecino);
          else{
            capacityFound = pathCapacity[vecino];
            visit.clear();
            break;
          }
        }
      }
    // Aca termina el bfs
    if (capacityFound == 0)
      break;
    flow += capacityFound;
    tint v = terminal;
    while (v != source){
      tint u = path[v];
      flowPath[u][v] += capacityFound;
      flowPath[v][u] -= capacityFound;
      v = u;
    }
  }
  return flow;
}
```

1.5. Trie (Jonaz)

```
const int MAXN = 100000;
int NODS = 1;
struct trie {
    map<char, int> sig;
    char c;
    bool final;
};
trie t[MAXN];
void resetNode(int i){
    t[i].sig.clear();
    t[i].final = false;
}
void initT(){
    forn(i,nods){
        resetNode(i);
    }
    NODS = 1;
}
void insertar(string st){
  int pos=0;
  forn(i,st.size()){
    if(t[pos].sig.find(st[i]) == t[pos].sig.end()){
      t[pos].sig[st[i]] = NODS;
      resetNode(nods);
      t[nods].c = st[i];
      NODS++;
    }
    pos = t[pos].sig[st[i]];
```

```
  }
    t[pos].final = true;
}
```

1.6. Geometra Basica Vectorial (Guty)

```
struct Punto{
  double x,y;
  Punto (double xx, double yy){ x = xx; y = yy; }
};
Punto operator + (Punto p1, Punto p2){
  return Punto(p1.x+p2.x, p1.y+p2.y);
}
Punto operator - (Punto p1, Punto p2){
  return Punto(p1.x-p2.x, p1.y-p2.y);
}
Punto operator * (ldouble k, Punto p){
  return Punto(k*p.x, k*p.y);
}
double operator * (Punto p1, Punto p2){
  return p1.x*p2.x + p1.y*p2.y;
}
double norma(Punto p){
  return sqrt(p*p);
}
double pcruz (Punto p1, Punto p2){
  return p1.x*p2.y - p2.x*p1.y;
}
double areaTriangulo(Punto a, Punto b, Punto c){
  return abs(pcruz(b-a,c-a))/2.0;
}
double areaParalelogramo(Punto a, Punto b, Punto c){
  return abs(pcruz(b-a,c-a));
}
```

1.7. Criba y Factorizacion (Jonaz)

```
const int MAXN = 10000000;  // hasta ahi se la re banca, y uno mas tambien
// si p[i] = 0      => i es primo
// si p[i] = j != 0 => j divide a i (y j primo)
int p[MAXN];
// arma la criba
void criba() {
  for(int i=4; i<MAXN; i+=2) p[i]=2;
  for(int i=3; i<MAXN; i+=2)
    if(!p[i]) for(int j=2*i; j<MAXN; j+=i) p[j] = i;
}
```

```
// devuelve una factorizacion de N del tipo <primo,exponente>
map<int,int> factorizar(int N){
    map<int,int> res;
    while(p[N]){
        res[p[N]]++;
        N /= p[N];
    }
    res[N]++;
    return res;
}
// devuelve vector con los factores primos que dividen a N en orden
vector<int> factorizarLista(int N){
    vector<int> res;
    while(p[N]){
        res.push_back(p[N]);
        N /= p[N];
    }
    res.push_back(N);
    // puede omitirse si no interesa que esten ordenados
    sort(res.begin(), res.end());
    return res;
}
```