Apunte ICPC

9 de octubre de 2017

Índice general

Notas previas

0.1. Abreviaciones utilizadas

```
typedef long long ll;
//en ciertos casos es necesario cambiar int por ll
typedef vector<int> vi;
typedef vector<vector<int> > vvi;
typedef pair<int,int> ii;
typedef vector<vector<ii> > vvii;              //util para grafos
typedef pair<pair<int,int>,int> iii;
#define mp(x,y) make_pair(x,y)
#define pb(x) push_back(x)
```

Capítulo 1

Estructuras de datos

1.1. Fenwick Tree

Nota: Ambas implementaciones tienen rangos entre 1 a n.

1.1.1. Actualizaciones por rango, consultas puntales

```
struct FenwickTree{
  vi FT;
  FenwickTree(int N){
     FT.resize(N+1,0);
  }

  int query(int i){
     int ans = 0;
     for(;i;i-=i&(-i)) ans += FT[i];
     return ans;
  }

  int query(int i, int j){
     return query(j)-query(i-1);
  }

  void update(int i, int v){
     for(;i<FT.size();i+=i&(-i)) FT[i] += v;
  }

  void update(int i, int j, int v){
     update(i,v); update(j+1,-v);
  }
};
```

1.1.2. Actualizaciones puntuales, consultas por rango

La consulta $query(a, b)$ corresponde a la sumatoria de los elementos entre los índices a y b.

```
struct FenwickTree {
  vi ft;
  FenwickTree(){}
  FenwickTree(int n){
    ft.assign(n + 1, 0);
  }

  int query(int b) {
    int sum = 0;
    for (; b; b -= b&(-b)) sum += ft[b];
    return sum;
  }

  int query(int a, int b) { \\RSQ
    return query(b) - (a == 1 ? 0 : query(a - 1));
  }

  void update(int k, int v) {                    // note: n = ft.size() - 1
    for (; k < (int)ft.size(); k += k&(-k)) ft[k] += v;
  }
};
```

1.2. Union-Find

Utilizada para trabajar conjuntos disjuntos. Sirve para encontrar componentes conexas en grafos no dirigidos.

```
class UnionFind {
private:
  vi p, rank, setSize;
  int numSets;
public:
  UnionFind(int N) {
        setSize.assign(N, 1); numSets = N; rank.assign(N, 0);
        p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
        if (!isSameSet(i, j)) { numSets--;
        int x = findSet(i), y = findSet(j);
        // rank is used to keep the tree short
        if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }
        else                   { p[x] = y; setSize[y] += setSize[x];
                             if (rank[x] == rank[y]) rank[y]++; } } }
  int numDisjointSets() { return numSets; }
  int sizeOfSet(int i) { return setSize[findSet(i)]; }
};
```

Capítulo 2

Grafos

2.1. Single source shortest path

2.1.1. Dijkstra

Utilizamos la representacion vvii con pares (vecino,peso)

```
asd
```

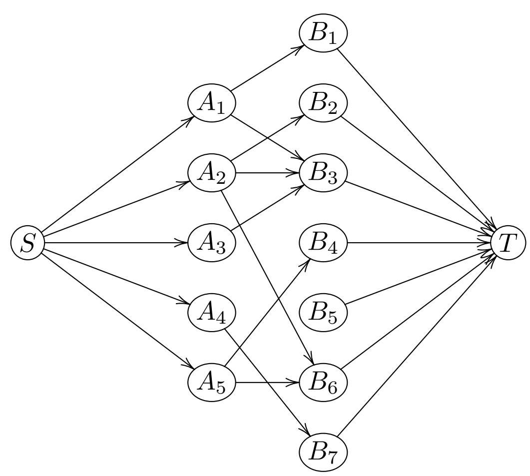
Capítulo 3

Flujo

3.1. Problemas de asignación

3.1.1. Bipartite matching

Tenemos dos conjuntos A y B, donde cada elemento de A es compatible con ciertos elementos de B. Además, tenemos la condición de que podemos asociar cada elemento de A con a lo más un solo elemento de B. Bipartite matching nos permite saber la cantidad máxima de asociaciones posibles.



Modelamiento utilizado. Todas las aristas llevan 1 de flujo.

Capítulo 4

Programación dinámica

Capítulo 5

Contenido adicional

5.1. Usar en caso de emergencia

GOD BLESS OUR SAVIOUR

Índice alfabético