GRAFOS

Floyd-Warshall: Distância

Dá a distância de todos os nós para todos (ele incluso). O(n³).

```
// INITIALIZATION OF DIST MATRIX
dist.assign(n, vector<int>(n, inf));
for (int i = 0; i < n; i++)
{
  for (auto node : adj[i])
  {
    dist[i][node] = 1;
  }
  dist[i][i] = 0;
}

// FLOYD-WARSHALL
for (int k = 0; k < n; k++)
{
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < n; j++)
    {
      dist[i][j] = min(dist[i][j], dist[i][k]
+ dist[k][j]);
    }
  }
}
```

Dijkstra: Distância

Dá a distância de um nó para todos os outros. O(n²).

```
   vector<long long> dijkstra(int n, int o,
int h)
   {
      vector<long long> dist(n, INF);
      vector<bool> processed(n, false);
      priority_queue<pair<long long, int>> q;
      dist[o] = 0;
      q.push({0, o});
      int u, v;
      while (!q.empty())
      {
         u = q.top().second;
         q.pop();
         if (processed[u])
            continue;
         processed[u] = true;
         for (auto e : adj[u]) // Percorrendo
as arestas de u
         {
            v = e.v;
            if (dist[u] + e.w < dist[v])
            {
               dist[v] = dist[u] + e.w;
               q.push({-dist[v], v});
            }
         }
      }
      return dist;
   }
```

Ford-Fulkerson: Fluxo Máximo

Dá o fluxo máximo que pode haver num grafo de V nós, do nó s até t. O(n * max(valor de uma aresta)). Pode não ser eficiente para arestas com valores muito grandes.

```
bool bfs(int s, int t, int V)
{
  // Initialize visited array
  bool visited[V];
  memset(visited, 0, sizeof(visited));
  // Initialize BFS search
  queue<int> q;
  q.push(s);
  visited[s] = true;
  parent[s] = -1;
  // Standard BFS Loop
  while (!q.empty())
  {
    int u = q.front();
    q.pop();
    for (int v = 1; v <= V; v++)
    {
      if (!visited[v] && rGraph[u][v] > 0)
      {
        // Set the parent in the BFS search to
later retrieve the path
        parent[v] = u;
        // If we find the target node, no need
to search anymore
        if (v == t)
        {
          return true;
        }
        q.push(v);
        visited[v] = true;
      }
    }
  }
  // If we didn't find target node, return
false
  return false;
}
// Returns the maximum flow from s to t in the
given graph
int fordFulkerson(int s, int t, int V)
{
  int u, v;
  int max_flow = 0; // Initialize flow as 0
  // Iteration: augment the flow while there
is path from source to target
```

```
  while (bfs(s, t, V)) // #1 - Execute the BFS
to see if there is a path
  {
    // #2 - Get the flow of the path, which is
its smallest residual value
    int path_flow = INT_MAX;
    for (v = t; v != s; v = parent[v])
    {
      u = parent[v];
      path_flow = min(path_flow,
                      rGraph[u][v]);
    }
    // #3 Update the flow of the path
(decrementing residual value in direct
    // paths and augmenting in inverse ones)
    for (v = t; v != s; v = parent[v])
    {
      u = parent[v];
      rGraph[u][v] -= path_flow;
      rGraph[v][u] += path_flow;
    }
    // Add path flow to overall flow
    max_flow += path_flow;
  }
  // Return the overall flow
  return max_flow;
}
```

Euler Tour

Faz um “passeio” pelo grafo com DFS, mantendo uma lista com os nós passados que é o “tour”. Sempre que retorna ao parent, adiciona o parente novamente no tour. Complexidade: DFS. Utilidade: problemas em que é necessário encontrar o menor ancestral comum entre dois nós, o que é uma forma rápida de calcular a distância entre eles. O grafo deve ser uma árvore, ao que parece.

```
vector<pair<int, int>> adj[(int)1e5 + 1];
vector<int> eTour;
int visited[(int)1e5 + 1] = {0};

#Simple DFS
void eulerTour(int u)
{
  visited[u] = 1;
  eTour.push_back(u);
  for (auto v : adj[u])
  {
    if (!visited[v.first])
    {
      eulerTour(v.first);
      eTour.push_back(u);
    }
  }
}
```

ESTRUTURAS

SegTree com Lazy Propagation

Otimização da SegTree que permite tanto fazer consultas em intervalos em O(logn) quanto atualizar intervalos inteiros em O(logn).

```
int lazy[(int)4e5 + 5], tree[(int)4e5 + 5];
#define v_base INT_MAX

void unlazy(int index, int ss, int se)
{
  if (lazy[index] == v_base)
    return;
  tree[index] = min(tree[index], lazy[index]);

  if (ss != se)
  {
    lazy[index * 2 + 1] = lazy[index];
    lazy[index * 2 + 2] = lazy[index];
  }

  lazy[index] = v_base;
}

void updateRangeUtil(int index, int ss, int
se, int us,
                     int ue, int value)
{
  unlazy(index, ss, se);

  // out of range
  if (ss > se || ss > ue || se < us)
    return;

  // Current segment is fully in range
  if (ss >= us && se <= ue)
  {
    lazy[index] = min(lazy[index], value);
    unlazy(index, ss, se);
    return;
  }

  // If not completely in rang, but overlaps,
recur for
  // children,
  int mid = (ss + se) / 2;
  updateRangeUtil(index * 2 + 1, ss, mid, us,
ue, value);
  updateRangeUtil(index * 2 + 2, mid + 1, se,
us, ue, value);

  // And use the result of children calls to
update this
  // node
  tree[index] = min(tree[index * 2 + 1],
tree[index * 2 + 2]);
}

void update(int n, int us, int ue, int value)
```

```
{
  updateRangeUtil(0, 0, n - 1, us, ue, value);
}

int getQueryUtil(int ss, int se, int qs, int qe, int index)
{
  unlazy(index, ss, se);

  // Out of range
  if (ss > se || ss > qe || se < qs)
    return v_base;

  // If this segment lies in range
  if (ss >= qs && se <= qe)
    return tree[index];

  // If a part of this segment overlaps with the given
  // range
  int mid = (ss + se) / 2;
  return min(getQueryUtil(ss, mid, qs, qe, 2 * index + 1),
             getQueryUtil(mid + 1, se, qs, qe, 2 * index + 2));
}

int query(int n, int qs, int qe)
{
  return getQueryUtil(0, n - 1, qs, qe, 0);
}

void buildSegTree(vi arr)
{
  f(i, 0, arr.size() * 4)
  {
    tree[i] = v_base;
    lazy[i] = v_base;
  }

  f(i, 0, arr.size())
      update(arr.size(), i, i, arr[i]);
}
```

Sparse-Table: Pré-processamento

Custa O(nlogn) para o pré-processamento. Não pode ser alterada depois (tabela estática).

```
long long st[K + 1][MAXN];
std::copy(array.begin(), array.end(), st[0]);
for (int i = 1; i <= K; i++)
  for (int j = 0; j + (1 << i) <= N; j++)
    st[i][j] = st[i - 1][j] + st[i - 1][j + (1 << (i - 1))];
```

Sparse table: Query

Custa O(logn) para a consulta. Permite queries de soma, máximo, mínimo, ou outro operador comutativo.

```
// R = end of query range
// L = Start of query range
long long sum = 0;
for (int i = K; i >= 0; i--)
{
  if ((1 << i) <= R - L + 1)
  {
    sum += st[i][L];
    L += 1 << i;
  }
}
```

Sparse table: Min-query optimized

Com essa otimização, custa O(1).

```
int lg[MAXN + 1];
lg[1] = 0;
for (int i = 2; i <= MAXN; i++)
  lg[i] = lg[i / 2] + 1;

int i = lg[R - L + 1];
int minimum = min(st[i][L], st[i][R - (1 << i) + 1]);
```

Disjoint Set: Find

Permite encontrar facilmente a que conjunto disjunto um nó pertence, identificado pelo “parente” do conjunto. O(1).

```
find(int x)
{
  if (pai[x] == x)
  {
    return x;
  }
  return pai[x] = find(pai[x]);
}
```

Disjoint Set: Join

Mescla os conjuntos de dois nós. Possui otimizações para ser praticamente O(1).

```
join(int x, int y)
{
  x = find(x);
  y = find(y);

  if (x == y)
  {
    return;
  }

  if (peso[x] < peso[y])
```

```
  {
    pai[x] = y;
  }
  else if (peso[y] < peso[x])
  {
    pai[y] = x;
  }
  else
  {
    pai[x] = y;
    peso[y]++;
  }
}
```

Segment Tree

Uma implementação mais compacta de SegTree. O (logn).

```
ll tree[(int)4e5 + 5];
#define v_base INT_MAX

void update(int n, int i, long long value)
{
  i += n;
  tree[i] = value;
  i >>= 1;
  while (i > 0)
  {
    tree[i] = min(tree[2 * i], tree[2 * i +
1]);
    i >>= 1;
  }
}

long long query(int n, int i, int j)
{
  i += n, j += n;
  long long minV = v_base;
  while (i <= j)
  {
    if (i % 2 != 0)
      minV = min(minV, tree[i++]);
    if (j % 2 == 0)
      minV = min(minV, tree[j--]);
    i /= 2, j /= 2;
  }
  return minV;
}

void buildSegTree(vector<int> &v)
{
  int n = v.size();
  for (int i = 0; i < 4 * n; i++)
    tree[i] = v_base;
  for (int i = 0; i < n; i++)
    update(n, i, v[i]);
}
```

Fenwick Tree – Obter Soma

Outra estrutura para consultas. Não permite consultas de min e max. Estática. O(logn).

```
   int getSum(int tree[], int index)
   {
      int sum = 0;
      index++;
      while (index > 0)
      {
         sum += tree[index];
         index -= index & (-index);
      }
      return sum;
   }
```

Fenwick Tree – Atualizar

O(logn).

```
void updateTree(int tree[], int n, int index,
int val)
   {
      index++;
      while (index <= n)
      {
         tree[index] += val;
         index += index & (-index);
      }
   }
```

MATH

Soma de quadrados de 1 a N

```
long long sum = n * (n + 1) * (2 * n + 1) / 6;
```

Exponenciação binária

Permite fazer exponenciação rapidamente.
O(logn), sendo n o expoente.

```
long long bin_pow(long long a, long long b)
{
  long long res = 1;
  while (b > 0)
  {
    if (b & 1)
    {
      res = res * a;
    }
    a *= a;
    b >>= 1;
  }
  return res;
}
```

GCD euclidiano

Permite computer o MDC, ou GCD, rapidamente. O(logn).

```
int gcd(int a, int b)
{
  if (b == 0)
    return a;
  else
    return gcd(b, a % b);
}
```

Bignum – Input

Técnica para trabalhar com números muito grandes. Os números são vectors de long long. Lê em complexidade O(n).

```
   for (int i = (int)s.length(); i > 0; i -=
9) {
      if (i < 9)
         a.push_back(atoi(s.substr(0,
i).c_str()));
      else
         a.push_back(atoi(s.substr(i - 9,
9).c_str()));
   }
```

Bignum – Output

Printa o bignum.

```
printf("%d", a.empty() ? 0 : a.back());
for (int i = (int)a.size() - 2; i >= 0; --i)
  printf("%09d", a[i]);
```

Bignum – Addition

Realiza adição entre dois bignums.

```
   int carry = 0;
   for (size_t i = 0; i < max(a.size(),
b.size()) || carry; ++i)
   {
      if (i == a.size())
         a.push_back(0);
      a[i] += carry + (i < b.size() ? b[i] :
0);
      carry = a[i] >= base;
      if (carry)
         a[i] -= base;
   }
```

Bignum - Remove leading zeros

Remove os zeros à esquerda (necessário após alguns processos. Na dúvida, lança após todos).

```
while (a.size() > 1 && a.back() == 0)
      a.pop_back();
```

Bignum – Subtraction

Faz subtração entre dois big nums

```
int carry = 0;
   for (size_t i = 0; i < b.size() || carry;
++i)
   {
      a[i] -= carry + (i < b.size() ? b[i] :
0);
      carry = a[i] < 0;
      if (carry)
         a[i] += base;
   }
   while (a.size() > 1 && a.back() == 0)
      a.pop_back();
```

Bignum – Division

Faz divisão entre dois big nums

```
int carry = 0;
   for (int i = (int)a.size() - 1; i >= 0;
--i)
   {
      long long cur = a[i] + carry * 1ll *
base;
      a[i] = int(cur / b);
      carry = int(cur % b);
   }
   while (a.size() > 1 && a.back() == 0)
      a.pop_back();
```

Bignum - Multiplication by SHORT integer

Multiplica um bignum por um inteiro pequeno (normal).

```
int carry = 0;
   for (size_t i = 0; i < a.size() || carry;
++i)
   {
      if (i == a.size())
         a.push_back(0);
      long long cur = carry + a[i] * 1ll * b;
      a[i] = int(cur % base);
      carry = int(cur / base);
   }
   while (a.size() > 1 && a.back() == 0)
      a.pop_back();
```

Bignum - Multiplication by BIGNUM

Multiplica dois bignum.

```
lnum c(a.size() + b.size());
```

```
   for (size_t i = 0; i < a.size(); ++i)
      for (int j = 0, carry = 0; j <
(int)b.size() || carry; ++j)
      {
         long long cur = c[i + j] + a[i] * 1ll
* (j < (int)b.size() ? b[j] : 0) + carry;
         c[i + j] = int(cur % base);
         carry = int(cur / base);
      }
   while (c.size() > 1 && c.back() == 0)
      c.pop_back();
```

Multiplicação de matrizes

Multiplica duas matrizes (vetores de vetores de long long). Complexidade: $O(n^3)$.

```
vector<vll> multiplyMatrices(vector<vll> A,
vector<vll> B)
{
   vector<vll> C(A.size(), vll(B.size(), 0));
   for (int i = 0; i < A.size(); i++)
   {
      for (int j = 0; j < B[0].size(); j++)
      {
         for (int k = 0; k < B.size(); k++)
            C[i][j] = (C[i][j] + (A[i][k] *
B[k][j]) % M) % M;
      }
   }
   return C;
}
```

NUMBER THEORY

Sieve of Eratosthenes

Computa todos os números primos até n. Quem estiver com 0 na sieve é primo. Os outros vão estar com seu último divisor primo. O(n*logn*logn).

```
vector<int> getSieve(int n)
{
  // Inicializa tudo com 0. Ao final, quem
ainda tiver 0 vai ser primo.
  vector<int> sieve(n + 1, 0);
  // Em geral 0 e 1 não são usados.
  sieve[0] = 1;
  sieve[1] = 2;
  for (int x = 2; x <= n; x++)
  {
    if (sieve[x])
      continue;
    for (int u = 2 * x; u <= n; u += x)
    {
      if (!sieve[u])
        sieve[u] = x;
    }
  }
  return sieve;
}
```

Extended Euclidean Algorithm

Além de calcular o GCD, calcula x e y para resolver a identidade de Bézout. Permite solucionar equações diofantinas.

```
int extEuclid(int a, int b, int &x, int &y)
{ // pass x and y by ref
  int xx = y = 0;
  int yy = x = 1;
  while (b)
  { // repeats until b == 0
    int q = a / b;
    int t = b;
    b = a % b;
    a = t;
    t = xx;
    xx = x - q * xx;
    x = t;
    t = yy;
    yy = y - q * yy;
    y = t;
  }
  return a; // returns gcd(a, b)
}
```

COMBINATORICS & PROBABILITY

Bitmask

Através de um inteiro, que é a “máscara”, codifica todas as possíveis combinações de um conjunto de t elementos. Complexidade: O(2 ^ n), apenas para listar todas as combinações. No exemplo abaixo, printa-se cada combinação com os elementos presentes.

```
// codifica todas as combinações possíveis
for (int mask = 1; mask < (1 << t); mask++)
{
  cout << "{ ";
  for (int i = 0; i < t; i++)
  {
    if ((mask >> i) & 1)
```

```
    {
      cout << i + 1 << " ";
    }
  }
  cout << "}" << endl;
}
```

Submasks de uma mask

Consegue listar todas as submasks de uma mask (no for interno). Ou seja, combinações que estão contidas dentro da combinação que a mask atualmente representa. Complexidade: O(3^n).

```
for (int m = 1; m < (1 << n); m++)
   {
      for (int s = m; s; s = (s - 1) & m)
      {
         //
      }
   }
```

Fatorial MOD p

Pré-computa os fatoriais MOD p em O(MAXN), e disponibiliza uma função para obtê-lo.

```
ll fat[MAXN + 10];


void initFact() {

    fat[0] = 1;

    for(ll i = 1; i <= MAXN; i++)

        fat[i] = (i * fat[i-1]) % MOD;

}


ll fact(ll n) {

    if (n < 0) return 1;

    return fat[n];

}
```

Mod

Calcula o mod, corrigindo valores negativos para positivos.

```
ll mod(ll n, ll m) {

    return ((n % m) + m) % m;
```

```
}
```

Little Fermat's Theorem (Inverso Modular)

Calcula o inverso modular de um número módulo um primo grande p. Pode ser usado como o equivalente de multiplicar um número n por 1/n, ou seja, dividir por ele, em problemas de combinatória que envolvam números muito grandes.

```
ll inv(ll a, ll p) {
    return bin_pow(a, p - 2, p);
}
```

Combinations MOD p with factorial and inverse

Calcula C(n, k) MOD p com fatoriais pré-calculados e função de inverso modular. Complexidade: O(logp), mesma do inverso, mesma do bin_pow.

```
ll C(ll n, ll p, ll m) {

    if (n < p) return 0;

       return (((fact(n)  * inv(fact(p),
m)) % m) * inv(fact(n - p), m)) % m;

}
```

Combinations with Lucas's Theorem

Permite calcular C(n, k) % p, em que n e k podem ser maiores que p. Roda em O(p + logn * logp)

```
ll C(ll n, ll k) {

   if (n < k) return 0;

   if (n >= MOD) return (C(n%MOD, k%MOD)
* C(n/MOD, k/MOD)) % MOD;

             return      (((fact[n]      *
inv(fact[k]))%MOD)  *  inv(fact[n-k]))  %
MOD;

}
```

Catalan's Numbers

Obtém os números de Catalan % p até MAX_N - 1, em tempo O(MAX_N * log(p)).

```
ll Cat[MAX_N];
```

```
  Cat[0] = 1;

   for  (int n = 0; n < MAX_N-1; ++n)  //
O(MAX_N log p)

    Cat[n+1]  =  ((4*n+2)%p  *  Cat[n]%p  *
inv(n+2)) % p;

  cout << Cat[100000] << "\n";
```

Derangement's Count

Um "derangement" é uma permutação em que todos os elementos estão fora de sua posição original. Conta quantas permutações são derangements. Tempo: O(n).

```
ll der(int n) {

    int der[n + 1] = {0};


    der[0] = 1;

    der[1] = 0;

    der[2] = 1;


    for (int i = 3; i <= n; i++)

         der[i] = (i - 1) * (der[i - 1]
+ der[i - 2]);

    return der[n];

}
```

DYNAMIC PROGRAMMING

Mochileiro

Clássico problema da mochila. Complexidade: O(n * v_max), onde n é o tamanho do vetor e v_max é o peso máximo da mochila.

```
#define MAXN 101
#define MAXV 10001
int dp[MAXN][MAXV];

void mochileiro(int n, int peso[], int
valor[])
{
  // Inicializa caso base (sem itens)
  for (int i = 0; i <= MAXV; i++)
    dp[0][i] = 0;
  // Calcula DP
  for (int i = 1; i <= n; i++)
  {
    for (int j = 0; j <= MAXV; j++)
    {
      dp[i][j] = dp[i - 1][j];
      if (j - peso[i] >= 0)
        dp[i][j] = max(dp[i][j], dp[i - 1][j -
peso[i]] + valor[i]);
    }
  }
}
```

Nº de Operações para virar palíndromo

Problema similar à distância de edição entre duas strings. Complexidade: O(n²) aproximadamente.

```
#define MAXN 1001
int dp[MAXN][MAXN];

int getNumOps(string s, int i, int j)
{
  if (dp[i][j] != -1)
    return dp[i][j];
  if (i >= j)
    return dp[i][j] = 0;
  if (s[i] == s[j])
    return dp[i][j] = getNumOps(s, i + 1, j -
1);
  int minOps = getNumOps(s, i + 1, j);
  minOps = min(minOps, getNumOps(s, i, j -
1));
  minOps = min(minOps, getNumOps(s, i + 1, j -
1));
  return dp[i][j] = minOps + 1;
}
```

OUTROS

Busca Binária

Dispensa apresentações. Pode ser adaptada pra buscar em funções e com números reais.

```
int binSearch(int v[], int l, int r, int t)
{
  if (l > r)
    return -1;
  int mid = (l + r) / 2;
  if (v[mid] == t)
    return mid;
  if (t > v[mid])
    return binSearch(v, mid + 1, r, t);
  return binSearch(v, l, mid - 1, t);
}
```

Busca Binária em função - Achar primeiro True

```
bool ok(int x) {
```

```
    //Implementar critério de busca...

}

int binSearch(int l, int r)

{

    int mid = l + (r - l) / 2;

    bool resultOk = ok(mid);

    if (l == r) {

        if (resultOk) return mid;

        else return -1;

    }

    if (resultOk)

        return binSearch(l, mid);

    else

        return binSearch(mid + 1, r);

}
```

---

intSNIPPET

```
    "Competitive Programming": {

        "prefix": "cp",

        "body": [

        "#include <bits/stdc++.h>",

        "using namespace std;",

        "",

        "/* TYPES  */",

        "#define ll long long",

        "#define pii pair<int, int>",

        "#define pll pair<long long, long long>",

        "#define vi vector<int>",

        "#define vll vector<long long>",

        "#define mii map<int, int>",

        "#define si set<int>",

        "#define sc set<char>",
```

```
        "",
        "/* FUNCTIONS */",
        "#define f(i,s,e) for(long long int i=s;i<e;i++)",
        "#define cf(i,s,e) for(long long int i=s;i<=e;i++)",
        "#define rf(i,e,s) for(long long int i=e-1;i>=s;i--)",
        "#define pb push_back",
        "",
        "/* UTILS */",
        "#define MOD 1000000007",
        "#define PI 3.1415926535897932384626433832795",
        "",
        "/* PRINT */",
        "template <class T>",
         "void print_v(vector<T> &v) { cout << \"{\"; for(auto x : v) cout << x <<
\",\"; cout << \"\\b}\";}",
        "",
        "void solve();",
        "",
        "int main() {",
        "    ios_base::sync_with_stdio(false);",
        "    cin.tie(NULL);",
        "",
        "solve();",
        "",
        "    return 0;",
        "}",
        "",
        "void solve() {",
        "",
        "}"
        ]
    }
```