3ª Aula Prática – Algoritmos de Divisão e Conquista – Resolução

1. Nearest Points

a)

```
/**
 * Brute force algorithm O(N^2).
 */
Result nearestPoints_BF(vector<Point> &vp) {
    Result res; // initialized with MAX_DOUBLE by default constructor
    for (unsigned i = 0; i < vp.size(); i++) {
            Point p = vp[i];
            for (unsigned j = i + 1; j < vp.size(); j++) {
                    double dist = p.distance(vp[j]);
                    if (dist < res.dmin)
                            res = Result(dist, p, vp[j]);
            }
    }
    return res;
}
```

b) **c)** (blue background)

```
/**
 * Auxiliary function to find the nearest points in "vp" between indices left and
 * right (inclusive), when the points are sorted by the y coordinate. "res" contains
 * initially the best solution found so far, and the final result in the end.
 */
static void npByY(vector<Point> &vp, int left, int right, Result &res)
{
    for (int i = left; i < right; i++) {
            Point p = vp[i];
            for (int j = i + 1; j <= right; j++) {
                    if (vp[j].y - p.y > res.dmin)
                            break;
                    else {
                            double d = p.distance(vp[j]);
                            if (d < res.dmin)
                                    res = Result(d, p, vp[j]);
                    }
            }
    }
}

/**
 * Recursive divide and conquer algorithm. Finds the nearest points in "vp" between
 * indices left and right (inclusive), using at most numThreads.
 */
static Result np_DC(vector<Point> &vp, int left, int right, int numThreads) {
   // Base case of a single point (or none): no solution, so distance is MAX_DOUBLE
   if (right <= left)
      return Result();

   // Base case of two points
   if (right - left == 1)
      return Result(vp[left].distance(vp[right]), vp[left], vp[right]);

   // Divide in halves (left and right) and solve them recursively,
```

```
    // possibly in parallel (in case numThreads > 1)
    int middle = (right + left)/2;
    Result resL, resR;
    if (numThreads > 1) {
       thread t([&](){resL = np_DC(vp, left, middle, numThreads / 2);});
       resR = np_DC(vp, middle + 1, right, numThreads / 2);
       t.join();
    }
    else {
       resL = np_DC(vp, left, middle, 1);
       resR = np_DC(vp, middle + 1, right, 1);
    }

    // Select the best solution from left and right
    Result res = resL.dmin < resR.dmin? resL : resR;

    // Determine the strip area around middle point
    double midX = (vp[middle].x + vp[middle+1].x) / 2;
    int sLeft = middle, sRight = middle+1;
    while (sLeft > left && vp[sLeft-1].x > midX - res.dmin)
       sLeft--;
    while (sRight < right && vp[sRight+1].x < midX + res.dmin)
       sRight++;

    // Order points in strip area by Y coordinate
    sortByY(vp, sLeft, sRight);

    // Calculate nearest points in strip area (using npByY function)
    npByY(vp, sLeft, sRight, res);

    // Reorder points in strip area back by X coordinate
    sortByX(vp, sLeft, sRight);

    return res;
}
```

d)

Invariante: `res` tem a distância entre os dois pontos (distintos) mais próximos, em que um dos pontos está no conjunto `C1={vp[0],…, vp[i-1]}` e o outro em `vp`.

- ✓ Verifica-se inicialmente (`i=0`), pois `C1` é vazio, logo o mínimo não está definido, o que é representado pela distância `MAX_DOUBLE` em `res` (definido no construtor sem argumentos).
- ✓ É mantido em cada iteração, pois procura um novo mínimo entre o ponto `vp[i]` e `vp[i+1],…, vp[vp.size()-1]` (não adiantaria analisar pontos anteriores), e de seguida incrementa `i`, logo o invariante verifica-se para o novo `i`.
- ✓ No final (`i=vp-size()`), garante a pós-condição (`res` tem a distância mínima entre todos os pontos), pois no final `C1` tem todos os pontos do vetor.

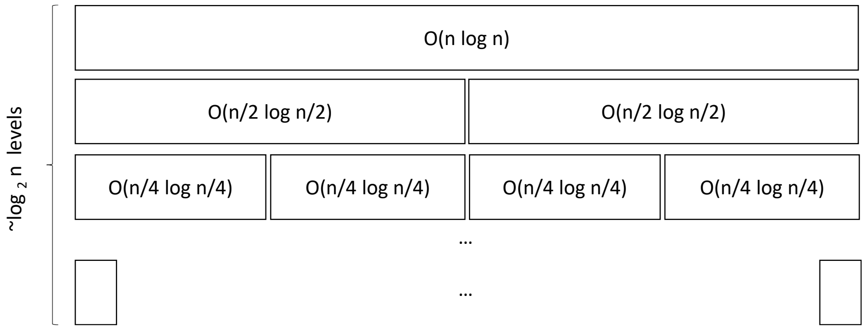
Variante: `vp.size() – i  (i=0, …, vp.size())`

- ✓ Inteiro, pois `vp.size()` e `i` são inteiros
- ✓ Não negativo, pois `i` no máximo vale `vp.size()`
- ✓ Estritamente decrescente, pois `i` é decrementado em cada iteração

e) Em cada chamada, sem contar as chamadas recursivas, a função `np_DC` tem de ordenar a faixa intermédia e de seguida analisá-la. Sendo `n` o tamanho do vetor analisado pela função `np_DC`, no limite a faixa intermédia pode ter esse tamanho (é o que acontece nos últimos casos de teste com x constante).

Assumindo que `std::sort` executa em tempo `O(n log n)`, será esse o tempo gasto na ordenação. A análise da faixa pela função `npByY`, conforme explicado no enunciado, tem um tempo de execução `O(n)`. Portanto o tempo dominante é `O(n log n)`.

Considerando agora as chamadas recursivas, temos a situação ilustrada na figura abaixo.



A soma em cada linha é igualmente de ordem `O(n log n)`. Como o nº de linhas (profundidade de recursão) é aproximadamente $\log_2 n$, o tempo total fica $O(n \log^2 n)$ (sem indicação de base), c.q.d.

f)

```
/*
 * Improved brute force algorithm, that first sorts points by X axis.
 */
Result nearestPoints_BF_SortByX(vector<Point> &vp) {
    Result res;
    sortByX(vp, 0, vp.size()-1);
    for (unsigned i = 0; i < vp.size() && res.dmin > 0; i++) {
        Point p = vp[i];
        for (unsigned j = i + 1; j < vp.size(); j++) {
            if (vp[j].x - p.x > res.dmin)
                break;
            double dist = p.distance(vp[j]);
            if (dist < res.dmin)
                res = Result(dist, p, vp[j]);
        }
    }
    return res;
}
```