Francisco Braz

1-)

Explicação do algoritmo: Inicialmente ordenamos o vetor em relação a coordenada x. Em seguida criamos um vetor Y e inicializamos todas as posições com infinito. Ou seja, nós não criamos e ordenamos um vetor Y com os pontos, nós simplesmente inicializamos todas as posições dele em um loop “para”, ou seja, gastamos O(n).
Logo em seguida chamamos nosso mergeSort para dividir o vetor de pontos até chegar nos casos bases. Após retornamos dos casos bases vamos construindo nosso vetor Y com a função intercala.

OBS: expressões como “ |Y[j].y - Y[i].y| ”, significa que estou pegando o valor absoluto.

a-)

```
algoritmo pontosProximos (V, n)
{-Entrada: vetor V de n pontos, com coordenadas x e y, em ordem
arbitrária
Saída:menor distância entre um par de pontos-}
início

  heapSort(V, n)
  para i = 0 até n faça
  início
     Y[i].x := ∞
     Y[i].y := ∞
  fim
  resp := mergeSort(V, Y, 0, n, n 0)

  função maxHeapify(T, n, i): vazio
  início
     esq := (2*i) + 1
     dir := (2*i) + 2
     maior := 0
     se esq <= n && T[esq].x > T[i].x então maior := esq
     senão se esq <= n && T[esq].x == T[i].x && T[esq].y > T[i].y então
          maior := esq
```

```
  senão maior := i

  se dir <= n && T[dir].x > T[maior].x então maior := dir
  senão se dir <= n && T[dir].x == T[maior].x && T[dir].y > T[maior].y
  então maior := dir


  se maior != i
      Ponto aux = T[i]
      T[i] = T[maior]
      T[maior] = aux
       maxHeapify (T, n, maior)
fim


função buildHeap( T[], n, ): vazio
início
  para i = n/2 até i >= 0 faça maxHeapify(T, n i)
fim

função heapSort(T[], n) :vazio
início
  buildHeap(T, n)
  para i = n até i >= 0 faça
  início
      aux := T[0]
      T[0] = T[i]
      T[i] = aux
      maxHeapify(T, n-1, 0)
  fim
fim

função min(x, y): double
início
  if(x < y) então retorna x
  senão retorna y
fim
```

```
função calcDist(a, b): double
início
    r1 := (b.x - a.x)
    r2 := (b.y - a.y)
    dist := sqrt((r1*r1) + (r2*r2))
    retorna dist
fim


função verificaFronteira(V, m, Y, k, d): double
início
   j := i+1
   enquanto j<=m && |Y[j].y - Y[i].y| < d então
   início
        d := min(calcDist(Y[j], Y[i]), d)
        j = j  + 1
   fim
   retorna d
fim

função intercala(V, Y, esq, meio, dir, tamY): double
início
    tamA := m-esq+1; tamB := dir-meio;
    para i = 1 até tamA faça A[ i ] := Y[esq + 1]
    para j = 1 até tamB faça B[ j ] := Y[meio + 1 + j]

    i := 1; j := 1; k := esq
    enquanto (i < tamA) && (j < tamB) faça
    início
         se A[i].y <= B[j].y então {Y[k] := A[i]; i := i+1}
         senão {Y[k] := B[j]; j := j+1}
         k := k+1;
   fim

   enquanto i < tamA faça {Y[k] := A[i]; i := i+1; k := k+1}
   enquanto j < tamB faça {Y[k] := B[j]; j := j+1; k:= k+1}
```

```
    tamY = tamY + 1
    retorna tamY
  fim

  função mergeSort(S, Y, esq, dir, tamY): double
  início
     cap := 0
     se dir - esq = 0 então retorna ∞
     senão se dir - esq = 1 então retorna calcDist(S[esq], S[dir])
     senão
     início
          meio := esq + ((dir - esq)/2)
          d1 := mergeSort(S, Y, esq, meio, tamY)
          d2 := merseSort(S, Y, meio+1, dir, tamY)
          tamY := intercala(S, Y, esq, meio, dir, tamY)
          d := min(d1, d2)
          para i = 0 até i <= n faça
          início
               se |S[i].x - S[meio].x| < d então
               início
                    Aux[cap] := S[i]
                    cap := cap + 1
               fim
          fim
          d = verificaFronteira(Aux, cap - 1, Y, 2 + tamY-1, d)
     retorna d
  fim
  retorna resp
fim
```

b-) Bem, nas aulas anteriores já calculamos tanto a complexidade do mergeSort quanto do Heapsort. A questão aqui é a ordenação do vetor Y. Como a questão pede para não ordenar o vetor Y com o heapSort em um pré processamento, nós simplesmente inicializamos todas as posições do vetor Y com infinito para que não atrapalhe no cálculo da distância. Ou seja apenas percorremos as posições do vetor Y utilizando um loop “para”. Assim nós conseguimos manter a

complexidade já vista em aula, O(n logn), pois como estamos ordenando nosso vetor Y com o intercala, nós não precisaamos chamar um heapSort na função “verificaFronteira” ,para ordenar o vetor Y, como acontece na versão do algoritmo $O(n \log^2 n)$.

c-) pontosProximos.c