Micael Andrade Dos Santos

1. Questão

   Um edifício é representado por uma tripla: (pExtremoEsq, Altura, pExtremoDir)

   Uma linha do Horizonte é representado por uma lista de pontos: [(x1, y1), (x2, y2), (x3, y3)]

   1. `Extruturação por Indução` → `Indução Forte`

      Caso Base 1: Nenhum edifício é dado como entrada para o algoritmo, retorna lista do horizonte `vazia`.

      Caso Base 2: Apenas um edifício é dado como entrada: retorne [(pExtremoEsq.x, Altura) , (pExtremoDir.x, 0)]

      HIP: O problema pode ser resolvido para $\leq \lceil n/2 \rceil$ edifícios, ou seja, sabemos encontrar a linha do horizonte para metade dos edifícios.

      Caso Geral: Resolver o problema para n dados. Para n dados a estratégia é a seguinte:

      1. Dividir o problema próximo do meio.
      2. Como temos duas metades $n/2$, logo podemos aplicar a HIP em cada metade para encontrar a linha do horizonte.
      3. Encontra-se a solução final intercalando às duas linhas do horizonte encontrados no passo anterior.

   2. `Algoritmo`

```
algoritmo findSkyline(E, esq, dir)
{-
  ENTRADA: Vetor E representando uma lista de edifícios com ponteirospara o início e final da lista.
  SAÍDA: Vetor H de pontos representando a linha do horizonte.
-}
inicio
    se(n = 0) então // Caso base 1
        retorne []
    se( ⌈(esq + dir)/2⌉ = 1) então
        retorne [( E[1].pExtremoEsq.x, E[1].altura), (E[1]pExtremoDir.x, 0)]

    -- Para Ns prédios
    meio = ⌊(esq+dir)/2⌋
    findSkyline(E, esq, meio)
    findSkyline(E, meio+1, dir)

    --Combinando as soluções
     retorne merge_lines(X,esq, meio, dir)
fim
```

```
algoritmo merge_lines(E, esq, dir)
  p_l = p_r = 0 -- Ponteiros para monitorar lista esquerda e direita
```

```
  curr_y = left_y = right_y = 0
  saida  -- Lista encadeada onde será armazenado os pontos da linha do horizonte.

  enquanto p_l < n_l and p_r < n_r faça
    inicio
      -- Captura os primeiros pontos superiores de dois horizontes.
      point_l := E[esq][p_l]
      point_r := E[dir][p_r]

      --Capturando um ponto da linha do horizonte esq
      se (point_l.x < point_r.x) então
        x := point_l.x
        left_y := point_l.y
        p_l := p_l + 1
      senão
        x := point_r.x
        right_y := point_r.y
        p_r := p_r + 1

      max_y = max(left_y, right_y) -- Escolhendo a maior linha do horizonte entre os dois.

      se (curr_y <> max_y) então -- Houve mudança no horizonte
              insereNaSaida(x, max_y, saida)
              curr_y = max_y
    fim
    append_skyline(p_l, E, n_l, curr_y)
    append_skyline(p_r, E, n_r, curr_y)

    retorne saida

fim
```

```
{-
  ENTRADA: Coordenada X e Y
-}
procedimento insereNaSaida(x, y, s):
  -- Se a mudança no horizonte não for vertical adiciona o novo ponto.
  se s <> null || s.ultimoElemento.x != x:
      s.insere((x, y))
  senão:
      s.ultimoElmento.y := y
```

```
{-
  ENTRADA: Ponteiro P, listaEsqOuDir, tamanho da lista, y tual do horizonte
  Utilizado para copiar o restante dos itens.
-}
procedimento append_skyline(p, lst, n, curr_y)
inicio
  enquanto p < n faça
      x := lst[p].x
      y := lst[p].y
      p += 1
      se (curr_y != y) então
          insereNaSaida(x, y)
          curr_y := y
fim
```

3. `Complexidade`

T(findSkyline) temos que no caso base 1 e 2 temos que `T(findSkyline) = C`
No caso geral dividimos o vetor ao meio, assim como ocorre no mergeSort, ou seja, T(n/2) + T(n/2) = `2T(n/2) + T(merge_lines)`

Temos que `merge_line` contém apenas um laço simples com operações elementares o qual será executado de forma linear, logo `T(merge_lines)` = O(n), onde n é o tamanho de uma das listas. Assim, ficamos com a seguinte fórmula de recorrência: `2T(n/2) + O(n)` note que essa mesma recorrência já foi resolvida quando estudamos o mergeSort, e sabemos que sua complexidade é `O(nlogn)`.

4. `Comparação com MergeSort`

   Assim como Udi Manber relata em seu livro, esse algoritmo de linha do horizonte é bastante semelhante ao mergeSort, pois aqui também é utilizado uma estratégia por divisão e conquista. Ou seja, quebramos o problema em duas partes para depois combiná-las.

---

3. Questão

   `Ideia`

   1. Ordenar as coordenadas por extremo de Y (EX_BOTTOM, EX_TOP) e coloque em uma extrutura T
   2. Varrer os segmentos ordenados de cima para baixo
   3. Se encontrar um ponto EX_TOP ele pode conter um segmento contido, então guarde esse ponto em V.
   4. Se encontrar um ponto EX_BOTTOM, verifique se o mesmo contém a mesma coordenada X dos segmentos guardados e sua coordenada Y está entre o range. Se sim, contamos uma interseção e removemos
      o segmento com EX_BOTTOM das candidatas.
      Repitir os passos 3 e 4 enquanto T não estiver vazio.

---

2. Questão

   a. Ideia

   Podemos ordenar os pontos em ordem **decrescente** pela coordenada `Y`, e caso dois pontos tenham a mesma coordenada `Y`, podemos escolher a maior coordenada `X` como critério de desempate.

   Dessa forma, o primeiro ponto da lista ordenada é um ponto `maximal`, então basta comparar a coordenada `x` de todos os próximos pontos apenas com esse maximal, caso o ponto Pi tenha coordenada Pi.x ≥ maximal.x, então Pi também entra no conjunto de pontos maximais.
   Aqui utilizei o `heapSort(adaptado)` para realizar a ordenação.

   b. Algoritmo

```
{-
  ENTRADA: Conjunto de pontos P com n pontos.
  SAÍDA: Conjunt M de pontos maximais.
```

```
-}
algoritmo (P,n)
inicio
  j := 1 -- Ponteiro para representar onde o próximio ponto maximal será armazenado em M
  M[1...n] -- Array do conjunto de pontos maximais
  Heapsort(P,n) -- Ordene todos os pontos com Y como critério (Em caso de desempate escolha o maior X)
  pointMax := P[1] --Com o vetor ordenado estamos pegando o ponto maximal que está na posição 1.
  M[j] := pointMax -- Adicionando no conjunto de maximais
  j := j+1
  para i=2 até n faça
    se(P[i].x >= pointMax.x) então
      M[j] := P[i] -- Encontrei outro ponto maximal
      j := j+1

fim
```

Logo a baixo está a função usada como critério de ordenação. Logo basta chamar essa função no `RearranjeMaxHeap` passando o vetor, o filho e seu respectivo pai.

```
{-
  ENTRADA: Vetor que está sendo ordenadao, e os indíces dos elementos comparados no heap.
  SAÍDA: O índice da menor coordenada Y e caso tenha duas coordenadas Y captura aquela com menor X
-}
procedimento comparador(hl, l, i):
    if(hl[l].y < hl[i].y):
        return l
    elif(hl[l].y > hl[i].y):
        return i
    else:
        if(hl[l].x <= hl[i].x):
            return l
        else:
            return i
```

c. `Complexidade`

Observe que realizamos uma ordenação por meio do `Heapsort` temos que a função utilizada como critério de ordenação é `T(comparador)=O(1)` pois é feita apenas operações elementares em seu corpo. Sendo assim, a complexidade do HeapSorte é a mesma que vimos nas aulas ateriores, ou seja, `O(nlogn)`.

Além disso, temos que o `para` que compara as coordenadas x com o ponto máximo consome Σ(i=2)(n) = (n-2-1) = (n-3) = `O(n)`. Logo, `T(pontosMaxiamais)` = O(n) + O(nlogn) = $O(nlogn)$

---

5. Questão

`Ideia`

`Caso Base 1`: Temos apenas um retângulo. Trivial, basta retorna a área desse retângulo.

`Caso Base 2`: Temos 2 tetângulos. Calcula a área total dos retângulos desconsiderando sua interseção caso haja.

`HIP`: Suponha que saibamos resolver o problema para ≤ ⌈n/2⌉ retângulos. Logo, para metade dos dados sabemos calcular a área entre dois retângulos desconsiderando sua interseção.

`Caso Geral`: Resolver o problema para n retângulos.

1. Dividir o problema pela metade.
2. Aplicar a `HIP` em cada metade dos dados.

---

6. Questão

`Ideia`

Podemos usar o algoritmo de `graham` para construção da envoltória convexa. Assim que tivermos uma envoltória convexa podemos remover do nosso conjunto de pontos todos os pontos que fazer parte da nossa envoltória convexa. Além disso, precisamos de um contador para contabilizar a profundidade dos pontos que foram removidos.

Depois precisamos calcular novamente a envoltória para os pontos remanescentes. Podemos fazer isso até que não restem pontos em nosso conjunto.

`NOTA:` Podemos rotular os pontos como inteiros, onde o index representa o ponto e o valor representa sua profundidade.

`Algoritmo`

```
{-
  ENTRADA: Conjunto P de pontos.
  SAÍDA: Vetor R contendo a profundidade dos pontos.
-}
algoritmo profundidadePontos
    profundidade = 0
    R -- Vetor que armazena a profundidade de todos os pontos.
    enquanto pontos <> Null -- Ou seja, enquanto existir pontos em nosso conjunto
        H -- Vetor representando nossa envoltória
        graham(P, H)
        para i = 1 até  H.tamanho faça
            se(H[i] <> Null) então -- Verificando se H[i] faz parte da envoltória.
                pontos.remove(H[i])  --Removendo do meu conjunto de pontos.
                R[H[i]] := profundidade
        profundidade += 1
    return R
```

`Complexidade`

`T(enquanto)=Σ(p=1)(n)=O(n)` e `T(paraInterno)=Σ(i=1)(m)=O(m)` tal que m é o tamanho da envoltória. Logo ficamos com `O(n)*O(nlogn)` = `O(n^2*logn)`