Algoritmos y Estructuras de Datos 2
Parcial 4

Leandro Carreira
669/18

Documento online: https://docs.google.com/document/d/1Vj76q07aYvwFVg1i9osum9DHRv-vlz2ccWmxJPVsAHg/edit?usp=sharing

---

Ej. 1. Sorting

Se tiene un arreglo P con n puntos de coordenadas enteras ($P[i] = (x_i, y_i)$ con $x_i, y_i \in \mathbb{Z}$), todos distintos. Sabemos que en P hay a lo sumo $\frac{n}{\log n}$ puntos que están fuera del círculo de centro $(0, 0)$ y radio n.

Dar un algoritmo estable de tiempo $O(n)$ que ordene los puntos según su distancia al origen $(0, 0)$. Por ejemplo: si tenemos $P = [(1, 0), (-3, 4), (0, 1), (2, -1)]$, el resultado debe ser $[(1, 0), (0, 1), (2, -1), (-3, 4)]$.

Demostrar que el algoritmo propuesto efectivamente ordena el arreglo, que es estable y que cumple con la cota de complejidad pedida.

Resolución:

Sé que voy a tener que calcular distancias (o medida equivalente) para cada uno de los n puntos, tarea con complejidad O(n)

Suponiendo que ya tengo las distancias calculadas, divido el análisis en dos partes:

Primero investigo qué forma tiene la cota de puntos exteriores a la circunferencia de radio n:

- Para n ≤ 1, la función se indetermina, por lo que es necesario que n sea mayor o igual a 2.

- En el siguiente gráfico, se observa como para valores de n por debajo de 10, todos los puntos pueden estar por afuera de la circunferencia, pero a partir de 11 inclusive, la cantidad de puntos que pueden estar afuera de la circunferencia pueden ser A LO SUMO un número estrictamente menor a la cantidad total de puntos.

  Esto da una intuición que los puntos exteriores NO crecen a la misma velocidad que n.

  Grafico:
  - En Naranja función en R
  - En Rojo función con valores en Z
  - En Violeta, función identidad

  (notar que n se representa con la letra x)

1 $\frac{x}{\log x}$

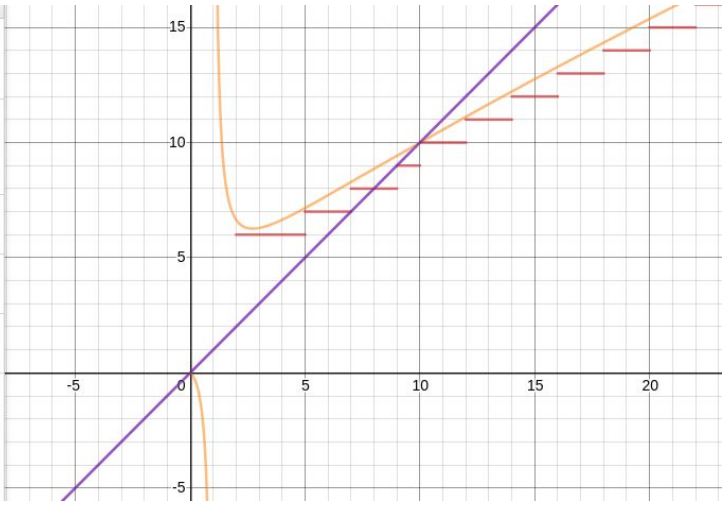
2 $\text{floor}\left(\frac{\text{floor}(x)}{\log(\text{floor}(x))}\right)\{x \geq 2\}$

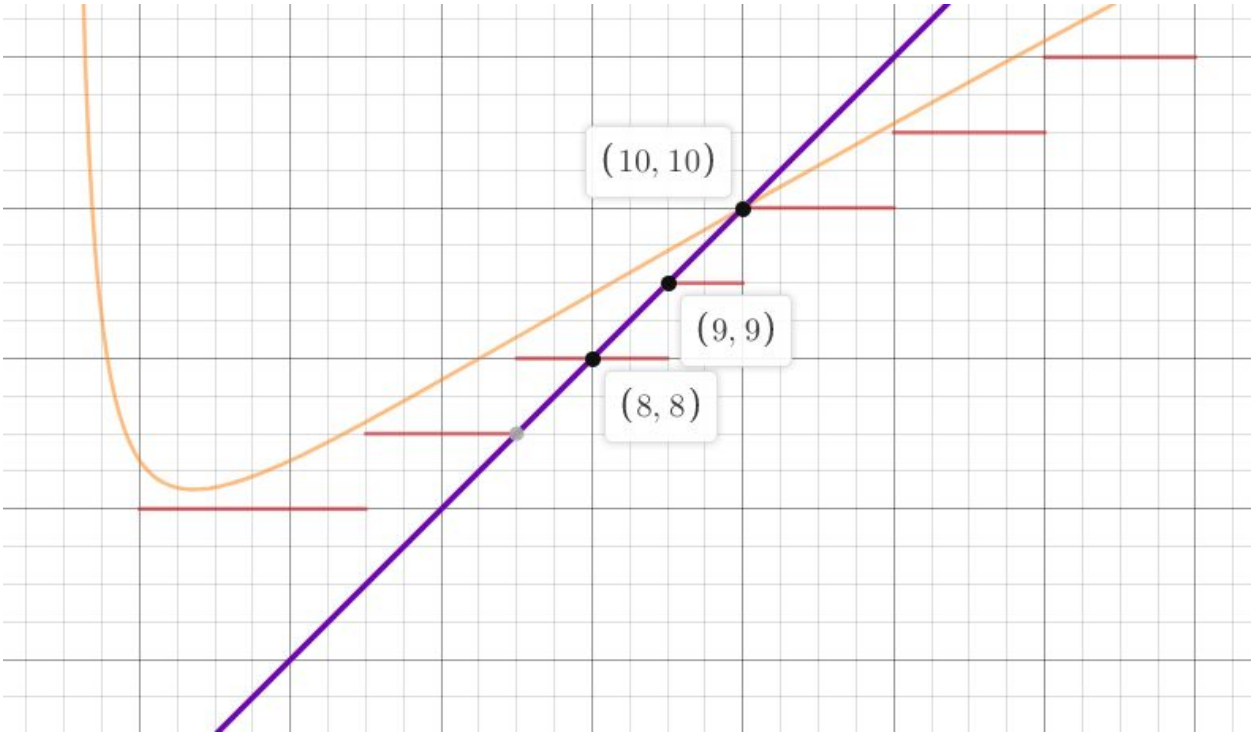
3 $y = x$

4

5

6





Con esta intuición en mente, puedo ser más formal y probar cual es la cota de complejidad para un algoritmo de ordenamiento estable como Merge Sort, con complejidad O(n log n) en condiciones generales:

Sea n la cantidad total de puntos como define el enunciado, llamo n_c a la cantidad de **puntos exteriores al círculo de radio n**, por lo tanto:

$n_c \leq n / \log n$

Pues n / log n puede no ser entero, por lo que n_c se redondea hacia abajo (no hay “pedacitos” de puntos).

Con esto, calculo la cota de complejidad del Merge Sort, pero solo para los puntos exteriores al círculo, que es:

$O(n_c \log n_c)$

Reemplazando con el peor caso ($n_c = n / \log n$)

$$n_c \log n_c = \frac{n}{\log n} * \log \frac{n}{\log n}$$

Por prop de log:

$$n_c \log n_c = \frac{n}{\log n} * (\log n - \log(\log n))$$
$$n_c \log n_c = \frac{n}{\log n} * \log n - \frac{n}{\log n} * \log(\log n)$$
$$n_c \log n_c = n - n * \frac{\log(\log n)}{\log n}$$

Y como log(log n) / log n es **menor a 1** para todo n en Z mayor o igual a 2

$$n_c \log n_c < n$$

Por lo tanto, la complejidad de ordenar los puntos por afuera del círculo de radio n puede acotarse por:

O(n)

Ya se cómo ordenar los puntos por afuera del círculo con la cota de complejidad solicitada, faltan los de adentro.

Para el interior de la circunferencia, uso el dato de que **las coordenadas están en Z**:

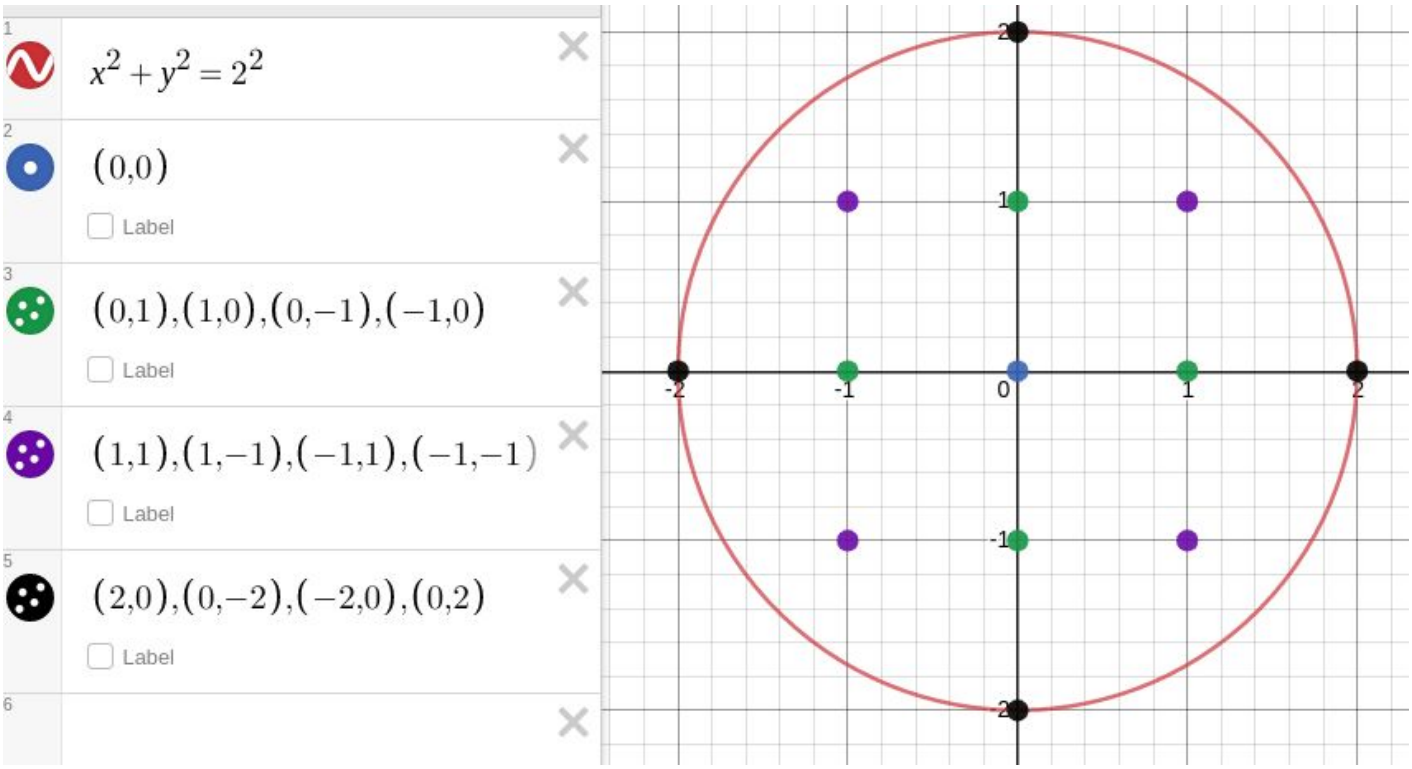
Como las coordenadas están en Z, y n es un número finito natural, entonces **la cantidad de distancias posibles al centro también estará acotada**.

Por ejemplo:

para **n=2**, habrá **13 puntos**, cuyas distancias (diferenciadas en colores) serán 4 **distintas**:

$Distancia \in \{0,\ 1,\ \sqrt{2},\ 2\}$

Dadas al resolver $D(x,y) = \sqrt{x^2+y^2}$



Dado que existen 2*n+1 posibles valores en x para un círculo de radio n, con n la distancia máxima posible, puedo usar la suma de los valores absolutos de las coordenadas como unidad de medida para establecer el órden de los puntos.

Juntando todo, se qué:

Calcular todas las distancias me lleva **O(n)**

Ordenar las coordenadas exteriores a la circunferencia se acota por **O(n)** con **Merge Sort**

Ordenar las coordenadas interiores (y el borde) de la circunferencia se acota por **O(n)** usando **Bucket Sort** sobre los (como máximo) **n** elementos dentro de la circunferencia.

Algoritmo:

```
coordenada es tupla de int con signo

ordenarPuntos(inout P: arreglo(coordenada))
      int n ← tam(P)                                                    O(1)
      lista(coordenada) Afuera ← Vacía()                                      O(1)
      arreglo(lista(coordenada)) Adentro ← crearArreglo(2*n+1)                O(n)

      // Inicializo buckets
      for i ← 0 to 2*n+1 do                                             O(n)
            Adentro[i] ← Vacía()                                              O(1)
      end for

      // Recorro todos los puntos
      for i ← 0 to n do                                                       O(n)
            // Calculo distancia
            int d ← (P[i].prim)^2 + (P[i].segu)^2                       O(1)
            if d > n^2 then
                  // Punto fuera del círculo
                  AgregarAtrás(Afuera, P[i])                                  O(1)
            else
                  // Punto dentro del círculo
                  // Uso suma de valores absolutos como medida de distancia/orden
                  int j ← abs(P[i].prim) + abs(P[i].segu)                     O(1)
                  AgregarAtrás(Adentro[j], P[i])                              O(1)
            end if
      end for

      // Ordeno los puntos exteriores usando un MergeSort que computa y ordena
      // la suma de los cuadrados de las componentes de cada tupla en O(1)
      // MergeSort recibe arreglo para acceder a sus elementos en el tiempo deseado.
      // Ordena por distancia cuadrada. Más información debajo de algoritmo.
      lista(coordenada) listaDeExteriores ← MergeSort(deListaAArreglo(Afuera))
                                                                              O(n)

      // Transformo arreglo de puntos interiores a lista
      lista(coordenada) listaDeInteriores ← deArregloALista(Adentro)
                                                      O(Longitud(listaDeInteriores))

      // Concateno listas en orden creciente de distancias y
      // convierto a Arreglo para devolver el mismo tipo de entrada
      P ← concatenarListasYConvertirAArreglo(listaInteriores, listaDeExteriores)
                                                                              O(n)
```

```
deListaAArreglo(in A: lista(coordenada), out O: arreglo(coordenada))
     arreglo(coordenada) O ← crearArreglo(Longitud(A))          O(Longitud(A))
     for i ← 0 to Longitud(A) do                                O(Longitud(A))
          O[i] ← A[0]                                           O(1)
          // Borro primer elemento de A
          Fin(A)                                                O(1)
     end for

deArregloALista(in A: arreglo(lista(coordenada)), out O: lista(coordenada))
     lista(coordenada) O ← Vacía()                              O(1)
     for i ← 0 to tam(A) do                                     O(tamA)
          concatenar(O, A[i])                                   O(n)
     end for

concatenar(inout A: lista(coordenada), in B: lista(coordenada))
     for i ← 0 to Longitud(B) do                                O(Longitud(B))
          AgregarAtrás(A, B[i])                                 O(1)
     end for

concatenarListasYConvertirAArreglo(inout A: lista(coordenada),
                                   inout B: lista(coordenada),
                                     out C: arreglo(coordenada))
     arreglo(coordenada) C ← crearArreglo(Longitud(A) + Longitud(B))       O(n)
     int longA ← Longitud(A)                                               O(1)
     int longB ← Longitud(B)                                               O(1)
     // Nota: longA + longB = n
     for i ← 0 to longA do                                          O(longA)
          C[i] ← A[0]                                                    O(1)
          // Borro primer elemento de A
          Fin(A)                                                         O(1)
     end for
     for i ← 0 to longB do                                          O(longB)
          C[longA + j] ← A[0]                                            O(1)
          // Borro primer elemento de B
          Fin(B)                                                         O(1)
     end for
```

Sobre MergeSort

MergeSort() recibe una lista de coordenadas para las cuales computa sus componentes al cuadrado en una primer pasada por todos los puntos n*O(1), los guarda en un

diccionario con clave coordenada y valor distancia cuadrado n*O(copia(int)) ≡ n*O(1), y utiliza estas distancias para comparar coordenadas de forma rápida.

Agrega una complejidad de O(n) al principio del algoritmo, que ya se encuentra acotada por la complejidad del algoritmo de O(n) para la cantidad de datos exteriores a la circunferencia (demo de esto al comiendo del ejercicio).

O sea, O(n) + O(n) ≡ O(n + n) ≡ O(n)

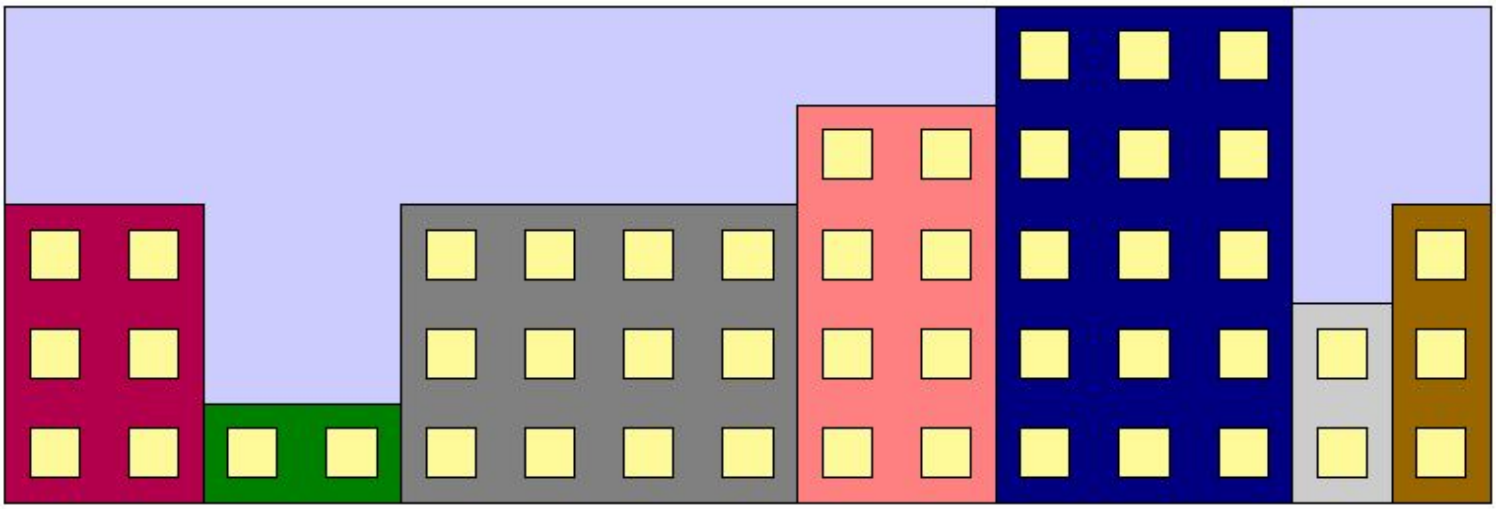
Ej. 2. Dividir y conquistar

Durante la cuarentena, un fotógrafo quiere sacar la foto perfecta para subir a las redes sociales. Su idea es retratar la vida en la nueva cotidianeidad, vista a través de las ventanas.

Los edificios que ve desde su ventana son rectángulos, uno inmediatamente al lado de otro, sin superposiciones. Por una regulación municipal, todos los edificios de la ciudad tienen una altura máxima de 30m.

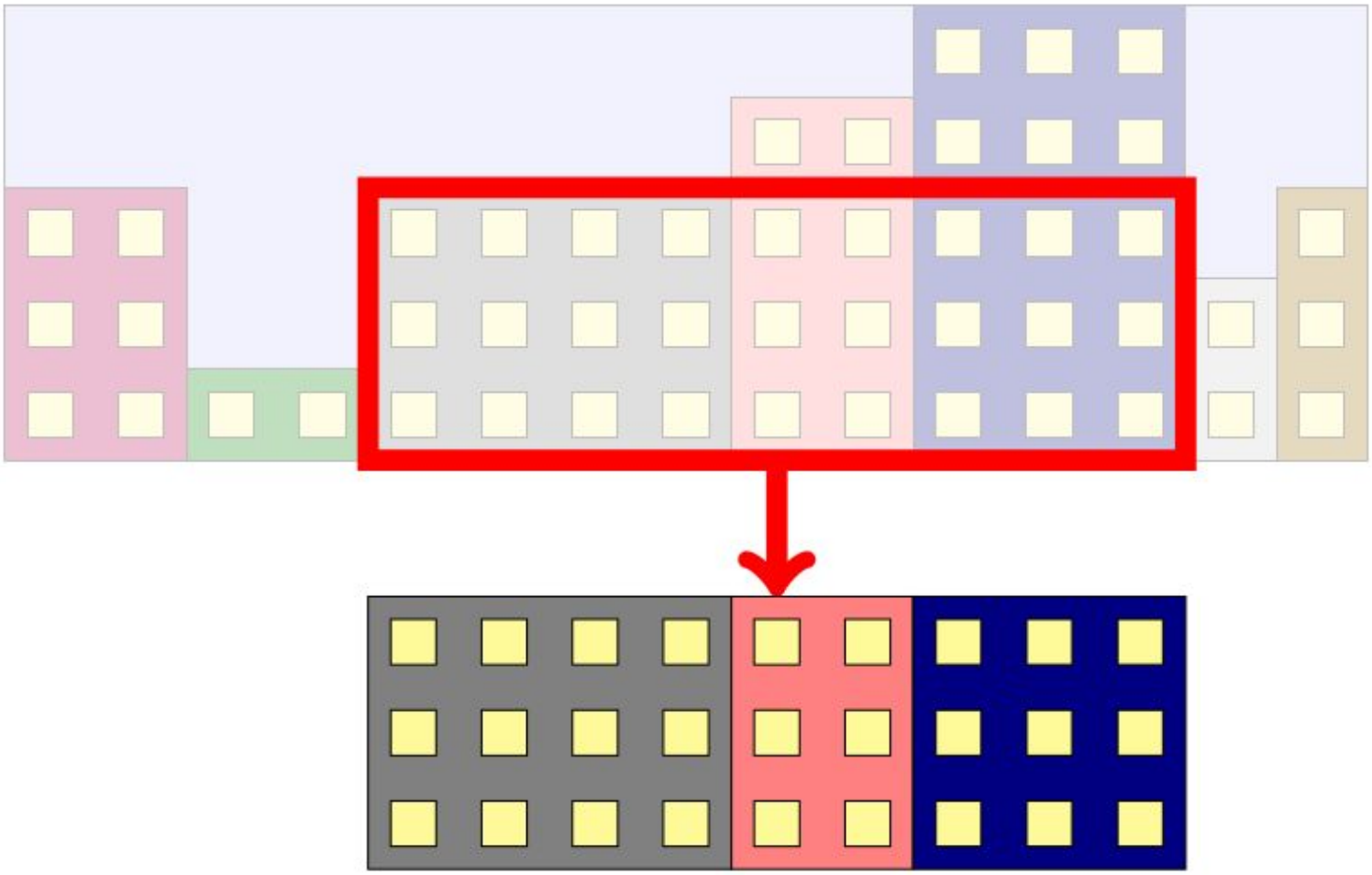
El fotógrafo nos contactó por videollamada para que lo ayudemos a determinar cuál es la foto de mayor área que puede tomar de modo tal que sólo se vean edificios (sin ninguna porción de cielo).

Por ejemplo, si la vista desde su ventana fuera[1]:



la foto de mayor área que puede tomar es:

[1] Dibujo a escala, suponer que el primer edificio mide 2m de ancho y 3m de alto.



La información proporcionada está en el siguiente formato: un arreglo A de n pares (w_i, h_i), de modo que w_i y h_i representan, respectivamente, el ancho y la altura (en metros) del i-ésimo edificio contando desde la izquierda. Para cada i, los valores w_i y h_i son números enteros positivos, y $h_i \leq 30$. La respuesta al problema debe ser *el área* de la mayor foto que puede tomar.

Por ejemplo, una entrada posible es: $A = [(57, 2), (1, 1), (1, 28), (1, 30), (1, 29), (1, 29), (1, 29), (57, 1)]$. En este caso, el primer edificio contando desde la izquierda mide 57m de ancho y 2m de alto. La solución (que se obtiene con un rectángulo de 5m de ancho y 28m de alto) debería ser 140m^2.

Se pide:

- Describir un algoritmo (que utilice la técnica **Divide & Conquer**) que resuelva el problema y cuya complejidad sea $O(n \cdot \log n)$.
- Demostrar que el algoritmo es correcto.
- Demostrar que su complejidad es efectivamente $O(n \cdot \log n)$.

No supe cómo resolver el problema :(

Intenté de varias formas pero no supe como mantener la información de si estoy en un caso de un edificio aislado maximo, o una suma de edificios consecutivos.

Porque SIEMPRE existe la posibilidad de que alguno de los edificios mida 1 de altura, pero 50 millones de ancho, por lo que no puedo decidirme qué arrastrar en el árbol de recursión, hasta haber comparado todas las entradas.

Agregar más casos base no sirvió.

A continuación uno de los intentos.

```
ÁreaAdyacenteMáxima(A) → int
      return obtenerÁreaYMin(A).prim

obtenerÁreaYMin(A) → tupla(int, int)
      if |A| = 1 then
            int base ← A[0].prim                                  O(1)
            int minH ← A[0].segu                                        O(1)
            return tupla(base, minH)
      end if

      b1, h1 ← ÁreaAdyacenteMáxima(A[0 .. n/2])                         T(n/2)
      b2, h2 ← ÁreaAdyacenteMáxima(A[n/2 .. n])                         T(n/2)
      b3, h3 ← ÁreaAdyacenteAlCentro(A)                                 O(n)

      int base ← b1 + b2 + b3                                           O(1)
      int minH ← min(h1, h2, h3)                            O(1)

      return (base * minH, minH)

ÁreaAdyacenteAlCentro(A) → tupla(int, int)
      bd, hd ← ÁreaHaciaDerecha(A[n/2 .. n])                            O(n)
      bi, hi ← ÁreaHaciaIzquierda(A[0 .. n/2])                          O(n)
      int base ← bd + bi                                                O(1)
      int minH ← min(hd, hi)                                            O(1)

      return (base * minH, minH)

ÁreaHaciaIzquierda(A) → tupla(int, int)
      n ← |A|                                               O(1)
      int base ← 0                                          O(1)
      int minH ← 50                                         O(1)
      int bestArea ← 0                                      O(1)
      int bestMinH ← 50                                     O(1)
      for i = 0 to n-1 do                                   O(n)
            base ← base + A[i].prim                         O(1)
            minH ← min(minH, A[i].segu)                     O(1)
            if bestArea < base * minH then                  O(1)
                  bestArea ← base * minH                    O(1)
                  bestMinH ← minH                   O(1)
            end if
      end for
      return (bestArea, bestMinH)
```

```
ÁreaHaciaIzquierda(A) → tupla(int, int)
     n ← |A|                                                    O(1)
     int base ← 0                                               O(1)
     int minH ← 50                                              O(1)
     int bestArea ← 0                                           O(1)
     int bestMinH ← 50                                          O(1)
     for i = 0 to n-1 do                                        O(n)
          base ← base + A[n-i].prim                             O(1)
          minH ← min(minH, A[n-i].segu)                         O(1)
          if bestArea < base * minH then                        O(1)
               bestArea ← base * minH                           O(1)
               bestMinH ← minH                          O(1)
          end if
     end for
     return (bestArea, bestMinH)
```

Analisis de complejidad

a=2 // Partes con las que trabajo
c=2 // En cuánto “parto” el vector de entrada

con f(n) = O(n)

T(n) = 2 * T(n/2) + O(n)

cuyo costo será de O(n log n) dado por analizar el arbol de recursión:
     $O(Sumatoria_{i=0 \text{ to } k}(2^i * n/2^i)) \equiv O(Sumatoria_{i=0 \text{ to } k}(n)) \equiv O(n * k)$

Donde k es la altura del árbol, o sea que k=log n

Por lo tanto, la complejidad es O(n * log n)

Usando el Teorema Maestro, como f(n) = O(n), y a=c=2, tanto la llamada recursiva como la parte no recursiva tienen un costo equivalente, por lo que estamos en el segundo caso donde la complejidad es:
     $O(n^{\log_c(a)} * \log n) \equiv O(n^{\log_2(2)} * \log n) \equiv O(n * \log n)$