



PROYECTO

FUNDAMENTOS DE ANÁLISIS & DISEÑO DE ALGORITMOS

JESÚS ALEXANDER ARANDA

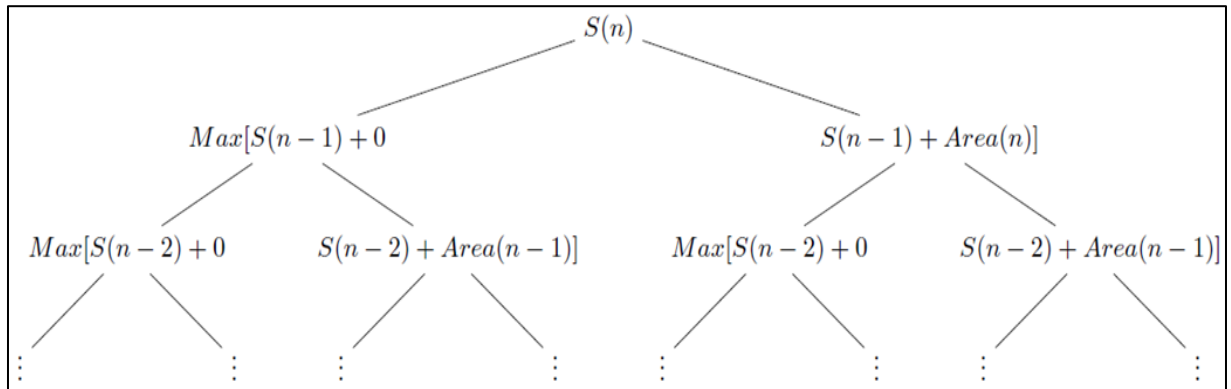
Santiago Hernández Arias	1631281
Cristian Camilo Vallecilla Cuellar	1628790
Esneider Arbey Manzano Arango	1628373
Christian Camilo Taborda Campiño	1632081

Ingeniería de Sistemas

Programación Divide & Vencerás:

- Estructura recursiva de la solución:

Analizando el problema, tenemos una serie de n puntos de impacto, nuestra idea es tomar las duplas que contienen la coordenada de tiro y radio de impacto, y analizar si este posible impacto es válido, es decir no se sale del área de bombardeo, lo siguiente será donde aplicamos la parte recursiva, tomaremos el máximo entre la función pasando $n - 1$ puntos de impacto sin tomar el punto n y pasando $n - 1$ puntos de impacto tomando el punto n , se puede ver básicamente de la siguiente manera:



Con S la función hipotética que resolverá nuestro problema. Retornaremos los puntos de impacto que suman más área, teniendo en cuenta la condición inicial.

- Algoritmo:

El algoritmo que puede resolver nuestro problema con ancho N y altura M del campo de bombardeo sería el siguiente:

```
impactos(I,S,X,R,N,M){  
    C = S  
    if(I.length() == 0) then:  
        return S  
    else:  
        suma = X + R  
        xp = I[0].pos  
        rp = I[0].R  
        if(suma <= (xp-rp) and (M/2 + rp) <= M and (xp+rp) <= N) then:  
            C.push_back(I[0])  
            C[0] = C[0] + area(rp)  
            I.erase(0)  
            return max(impactos(I,S,X,R,N,M), impactos(I,C,xp,rp,N,m))  
        else:  
            I.erase(0)  
            return impactos(I,S,X,R,N,M)
```

Donde I es el conjunto con los blancos, S es el conjunto para la solución, X es la posición horizontal del blanco actual, R es el radio del blanco actual, N es la dimensión horizontal del cuadro y M es su dimensión vertical. Donde la función *max* toma el área almacena en la primera posición del vector soluciones o copia y devuelve el que tenga mayor área.

- Eficiencia en tiempo:

La complejidad del algoritmo se deriva de la complejidad de las dos partes que lo conforman, ordenar el arreglo de blancos y recorrerlo recursivamente a medida que se maximiza el área de las decisiones tomadas.

Si analizamos la complejidad de ordenar los blancos usando mergesort obtenemos la siguiente ecuación recurrencia:

$$T(n) = 2T\left(\frac{n}{2}\right) + (n + 1)$$

Donde el primer sumando representa la cantidad de llamados recursivos y la segunda el costo de combinar las soluciones. Resolviendo la ecuación por método maestro nos queda:

$$T(n) = O(n \log n)$$

Pues:

$$n^{\log_2 2} = \theta(n)$$

Si analizamos el costo de recorrer el arreglo recursivamente, cada vez se realizan 2 llamados recursivos de si mismo, ambos con un arreglo de tamaño $n - 1$, tenemos entonces $2T(n - 1)$, lo demás, son factores constantes, nuestra ecuación de recurrencia es:

$$T(n) = 2T(n - 1) + O(1)$$

Resolviendo:

$$T(n) = 2(2T(n - 2) + 1) + 1 = 4T(n - 2) + 2 + 1$$

$$T(n) = 8T(n - 3) + 4 + 2 + 1$$

\vdots

$$2^k T(n - k) + \sum_{i=0}^{k-1} 2^i$$

Con la condición de parada, cuando recibimos el arreglo vacío $T(0) = 1$ y $n - k = 0$ obtenemos:

$$T(n) = 2^n + \sum_{i=0}^{n-1} 2^i$$

Resolviendo la sumatoria:

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Obtenemos finalmente:

$$T(n) = 2^n + \sum_{i=0}^{n-1} 2^i = 2^n + 2^n - 1$$

$$T(n) = 2^{n+1} - 1$$

Finalmente, tenemos que el costo computacional total está dado por:

$$T(n) = n \log n + (2(2^n) - 1)$$

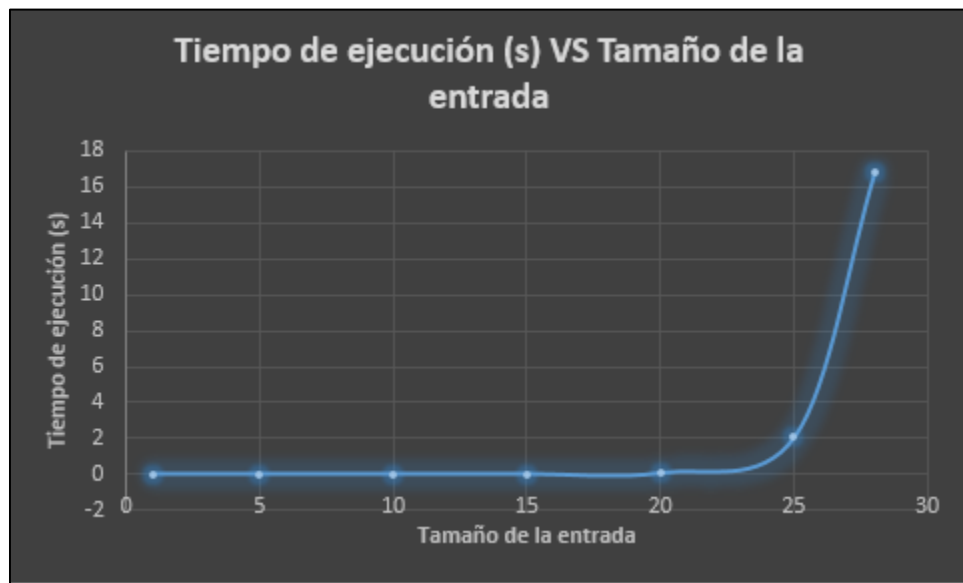
Donde el factor dominante es exponencial, dando como resultado:

$$T(n) = O(2^n)$$

Tomando algunas pruebas con entradas de distintos tamaños y promediando los tiempos medidos se obtuvieron los siguientes resultados:

Tamaño de la entrada	Tiempo de ejecución (s)
1	0
5	0,00005
10	0,000266667
15	0,001833333
20	0,054366667
25	2,066116667
28	16,79433333

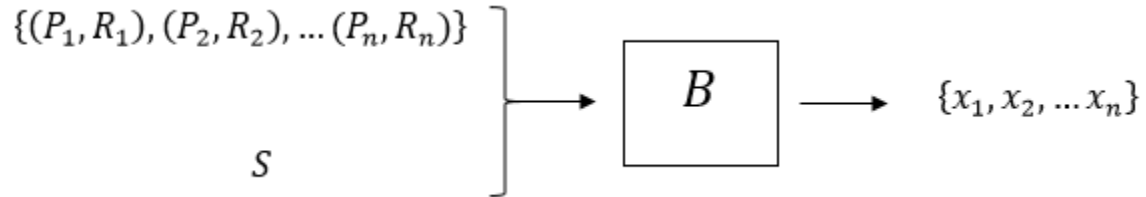
La siguiente gráfica representa los resultados obtenidos en la tabla anterior:



Analizando el comportamiento de la gráfica se observa que el crecimiento sigue un orden exponencial, así como el orden de crecimiento de la complejidad del algoritmo calculada anteriormente.

Programación Dinámica:

Analizando el problema tenemos lo siguiente:

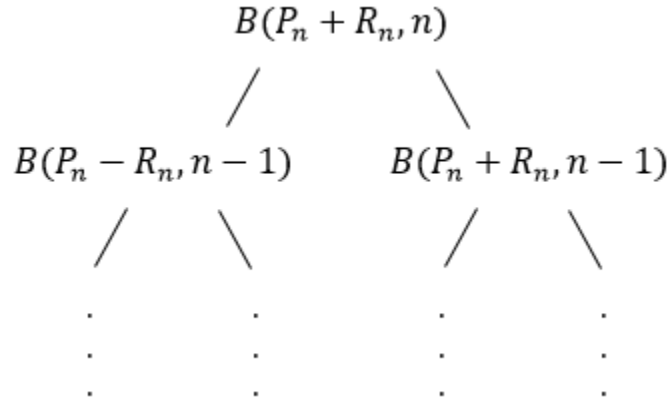


Teniendo a N y M como las dimensiones del cuadro donde se ubican los blancos, las entradas son un conjunto de blancos $\{b_1, b_2, \dots, b_n\}$, donde cada blanco es una dupla conformada por su posición en el eje horizontal P y su radio R , y un número S que me indica la posición más lejana de todas aquellas donde termina el área de cada blanco para servir como criterio de solapamiento. Es necesario asumir que P_i y R_i son números enteros para cada i . Además se debe cumplir que $0 \leq S \leq N$, $0 < P_i < N$, $P_i + R_i \leq N$, y $P_i - R_i \geq 0$ para cada i . La salida es un conjunto de valores binarios tal que $x_i \in \{0,1\}$, donde $1 \leq i \leq n$. El objetivo de la función B es seleccionar aquellos blancos que suman la mayor área sin solaparse entre ellos, es decir:

$$\max \sum_{1 \leq i \leq n} x_i (\pi R_i^2) \text{ tal que } x_i (P_i + R_i) \leq S \text{ para cada } i$$

- Estructura de la solución óptima:

Dependiendo del blanco actual que estemos evaluando y la posición donde se encuentre el criterio de solapamiento, habrá un máximo de dos posibles subproblemas. Si ilustramos de forma recursiva la estructura de la solución tenemos lo siguiente:



El anterior árbol puede llegar a tener dos aristas por nodo como máximo, y el problema se sigue dividiendo hasta llegar a los casos triviales, donde se elegirá el camino con mayor beneficio. Los casos triviales serían aquellos donde no nos quedan blancos por evaluar o donde el criterio de solapamiento es 0.

Sea $B(S', n')$ un problema de bombardeo considerando los primeros n' blancos y con un criterio de solapamiento S' . Como el problema principal toma desde los n primeros blancos hasta los 0 primeros blancos y con un criterio de solapamiento desde S hasta 0, la cantidad de subproblemas es: $(S + 1)(n + 1)$. Sea $S(B(S', n'))$ la solución del problema $B(S', n')$, Tenemos lo siguiente:

$$S(B(S', n')) = \{x_1, x_2, \dots, x_{n'}\}$$

Si $x_{n'} = 1$:

$$S(B(P_{n'} - R_{n'}, n' - 1)) = \{x_1, x_2, \dots, x_{n'-1}\}$$

Si $x_{n'} = 0$:

$$S(B(S', n' - 1)) = \{x_1, x_2, \dots, x_{n'-1}\}$$

- Valor de la solución de manera recursiva:

Sea $V(S', n')$ la función que calcula el valor de la solución $S(B(S', n'))$, entonces:

➤ Si $S' = 0$ o $n' = 0$:

$$V(S', n') = 0$$

➤ Si $P_{n'} + R_{n'} > S'$:

$$V(S', n') = V(S', n' - 1)$$

➤ Si $P_{n'} + R_{n'} \leq S'$:

$$\max(V(S', n' - 1), \pi(R_{n'}^2) + V(P_{n'} - R_{n'}, n' - 1))$$

- Algoritmo para llenar la matriz de valores:

Un aspecto importante acerca del algoritmo es que recibe el arreglo de blancos ordenado, tomando como criterio de ordenamiento la posición sobre el eje horizontal de cada blanco. Para ello se utilizó el algoritmo de mergesort, ya que su complejidad es $n \log n$, el cual no sería un factor dominante al analizar la complejidad del algoritmo principal que se visualizará a continuación. El siguiente algoritmo se encargará de llenar la matriz de valores y una matriz auxiliar para la construcción de la solución a la vez:

```

beneficios(B,V,A):
  m = V.rows
  n = v.columns
  for x=0 to m step +1 do:
    for y=0 to n step +1 do:
      if(x == 0 or y == 0) then:
        V[x][y] = 0
        A[x][y] = 0
      else:
        S = B[y-1].P + B[y-1].R
        if(S <= x) then:
          V[x][y] = max(V[x][y-1],
                         $\pi(B[y-1].R)(B[y-1].R) + V[B[y-1].P - B[y-1].R][y-1]$ ,
                        aux)
          if(aux == 2) then:
            A[x][y] = 1
          else:
            A[x][y] = 0
        else:
          V[x][y] = V[x][y-1]
          A[x][y] = 0

```

Donde B es un arreglo con los n blancos a evaluar, V es la matriz que me va a almacenar todos los valores de las soluciones dadas a los subproblemas presentados y A es la matriz auxiliar que va a contener valores binarios para permitir la construcción de la solución. Las dimensiones de ambas matrices son $(S + 1)(n + 1)$, donde $S = P_n + R_n$ y n la cantidad de blancos a evaluar.

La función *max* utilizada no sólo retorna el máximo, también recibe una variable *aux* que indica cuál de los valores evaluados fue elegido como el máximo, su algoritmo es:

```

max(A,B,&aux):
  if(A > B) then:
    aux = 1
    return A
  else:
    aux = 2
    return B

```

- Algoritmo para determinar la solución óptima:

El siguiente algoritmo determina cuál fue el subconjunto de blancos que permitieron maximizar el área de alcance de los misiles imprimiendo en pantalla uno por uno:

```
imprimirSolucion(A,B):  
    m = A.rows  
    n = A.columns  
    print("Los blancos seleccionados son: ")  
    while(n > 0):  
        if(A[m][n] == 1) then:  
            print(B[n-1])  
            m = B[n-1].P - B[n-1].R  
            n--  
        else:  
            n--
```

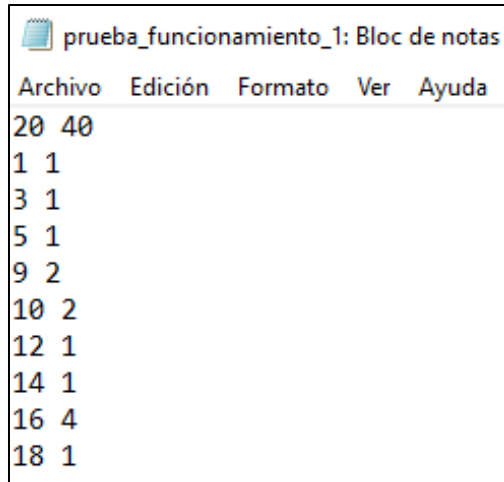
Donde A es la matriz auxiliar que contiene los valores binarios usados para determinar la solución y B es el arreglo con los n blancos a evaluar.

- Modificando el problema:

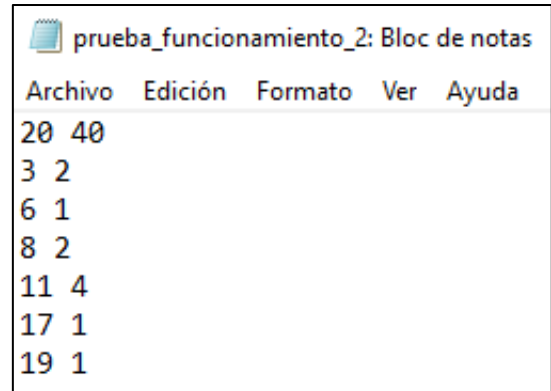
Si modificamos el problema de tal forma que no se busque maximizar el área de impacto de los misiles sino maximizar la cantidad de misiles que se disparan, lo único que cambiaría en el enfoque de la programación dinámica es el valor calculado para llenar la matriz de valores, pues en vez de sumar el área que proporciona cada blanco evaluado, estaríamos sumando un 1 para indicar que se dispara un misil. Al final, el algoritmo haría exactamente lo mismo, seleccionar el subconjunto de blancos que permitan disparar la mayor cantidad de misiles sin solaparse entre ellos.

- Ejemplos de pruebas:

Entradas:



Archivo	Edición	Formato	Ver	Ayuda
20	40			
1	1			
3	1			
5	1			
9	2			
10	2			
12	1			
14	1			
16	4			
18	1			



Archivo	Edición	Formato	Ver	Ayuda
20	40			
3	2			
6	1			
8	2			
11	4			
17	1			
19	1			

Salidas:

```
Ingrese el nombre del archivo: prueba_funcionamiento_1.txt
Los blancos seleccionados son: (16,20) (10,20) (5,20) (3,20) (1,20)
El área máxima de daño es: 72.2566
El tiempo total de ejecución es: 0
```

```
Ingrese el nombre del archivo: prueba_funcionamiento_2.txt
Los blancos seleccionados son: (19,20) (17,20) (11,20) (6,20) (3,20)
El área máxima de daño es: 72.2566
El tiempo total de ejecución es: 0
```

- Eficiencia en tiempo:

La complejidad del algoritmo se deriva de la complejidad de las tres partes que lo conforman, ordenar el arreglo de blancos, llenar la matriz de valores y construir la solución.

Si analizamos la complejidad de ordenar los blancos usando mergesort obtenemos el siguiente costo:

$$T(n) = n \log n$$

Si analizamos la complejidad de llenar la matriz de valores nos damos cuenta que la cantidad de operaciones realizadas depende directamente de la cantidad de iteraciones que realicen los bucles *for*, pues dentro y fuera de los bucles tenemos una cantidad constante de operaciones. El tiempo de ejecución está dado por:

$$T(n) = \sum_{x=0}^m \sum_{y=0}^n 1 = \sum_{x=0}^m (n+1) = (m+1)(n+1) = O(mn)$$

Donde m corresponde a la cantidad de filas de la matriz y n corresponde a la cantidad de columnas de la matriz.

Si analizamos la complejidad de determinar la solución óptima nos damos cuenta que la cantidad de operaciones realizadas depende directamente de la cantidad de iteraciones que realice el bucle *while*, pues dentro y fuera del bucle tenemos una cantidad constante de operaciones. Como dentro del *while* siempre hay un decremento en la variable n y la condición de parada está cuando $n = 0$, la cantidad de iteraciones realizadas por el bucle será n . El tiempo de ejecución está dado por:

$$T(n) = \sum_{i=1}^n 1 = n$$

Donde n corresponde a la cantidad de blancos que se tuvieron en cuenta para la evaluación de los posibles subconjuntos.

Finalmente, tenemos que el costo computacional total está dado por:

$$T(n) = n \log n + (m + 1)(n + 1) + n$$

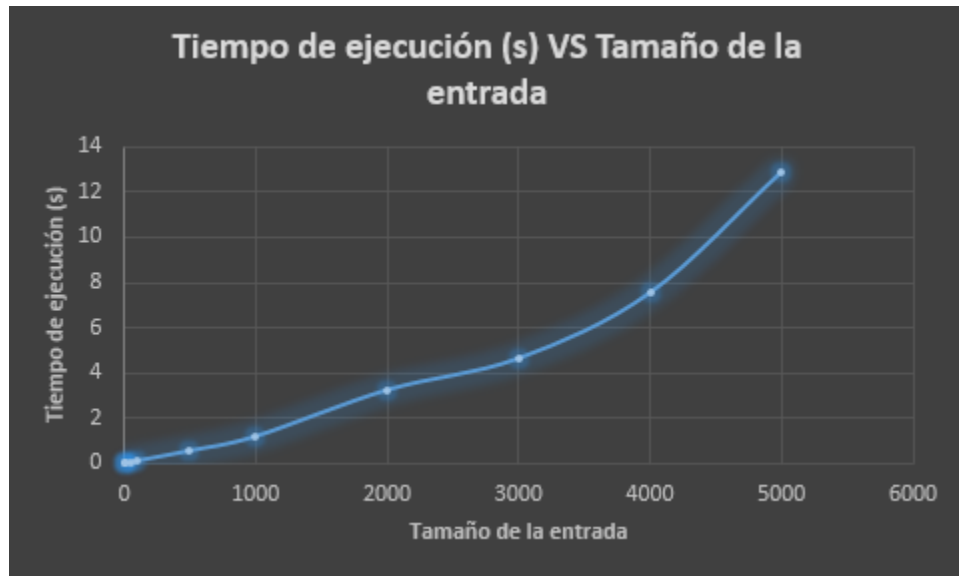
Donde el factor dominante es polinomial, dando como resultado:

$$T(n) = (m + 1)(n + 1)$$

Tomando algunas pruebas con entradas de distintos tamaños y promediando los tiempos medidos se obtuvieron los siguientes resultados:

Tamaño de la entrada	Tiempo de ejecución (s)
1	0
5	0,0155
10	0,0155
50	0,0545
100	0,10525
500	0,547
1000	1,183
2000	3,2333
3000	4,6287
4000	7,53
5000	12,8666

La siguiente gráfica representa los resultados obtenidos en la tabla anterior:



Analizando el comportamiento de la gráfica se observa que el crecimiento sigue un orden polinomial, así como el orden de crecimiento de la complejidad del algoritmo calculada anteriormente.

Programación Voraz:

- Criterio de escogencia local (heurística):

El criterio de escogencia es la cercanía del centro del bombardeo con el punto $(0, M/2)$, es decir que, para armar la solución, se va a escoger de entre los puntos de ataque provistos aquel cuyo valor de centro sea el más pequeño (el que esté más a la izquierda). A continuación, se escogerá nuevamente, de entre los puntos de ataque restantes, el siguiente punto teniendo en cuenta el mismo principio, siempre que éste no se solape el último punto añadido a la solución (factibilidad). Basta esta única comparación puesto que, al solaparse con otro círculo diferente al último, necesariamente se solaparía también con el último, y así se evita comparar con todos los círculos del conjunto solución.

La idea detrás del algoritmo es que después de escoger un círculo para la solución, el recurso, que en este caso es la trinchera, se vea lo menos mermada posible para que así se disponga de más espacio para seleccionar nuevos puntos de ataque. Además, al escoger el círculo con el centro más a la izquierda, se busca también que el espacio no aprovechado sea mínimo.

- Algoritmo:

El algoritmo tiene los conjuntos blancos y solución. El conjunto solución está vacío inicialmente. Lo primero es ordenar el conjunto blancos utilizando mergesort, de manera ascendente respecto al centro de los blancos. El primer blanco de este conjunto ya ordenado se añade a la solución. Después, se revisan los demás blancos. Como el conjunto está ordenado ya, simplemente se revisa cada vez si el blanco actual se solapa con el último elemento del conjunto solución, y de no solaparse se añade a él. Pseudocódigo:


```

voraz(B,S):
  if (B.size()<=1) then:
    return B
  mergesort(B)
  S += B[0]
  for k=1 to B.size() - 1 step +1 do:
    if (not(revisarSolapamiento(S[S.size()-1], B[k])) then:
      S += B[k]
  return S

```

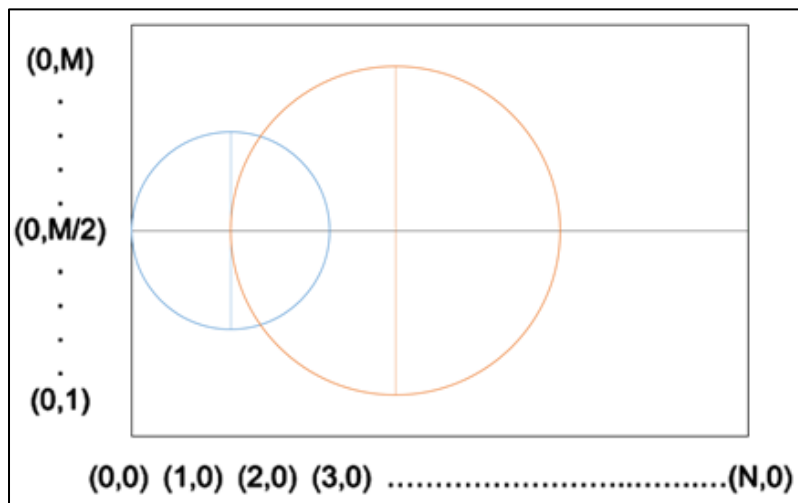
Donde B es el conjunto con los blancos y S es el conjunto para la solución. La función *revisarSolapamiento* recibe dos blancos y devuelve *true* si hay solapamiento entre ellos, *false* sino.

- Análisis de instancias donde hay solución óptima:

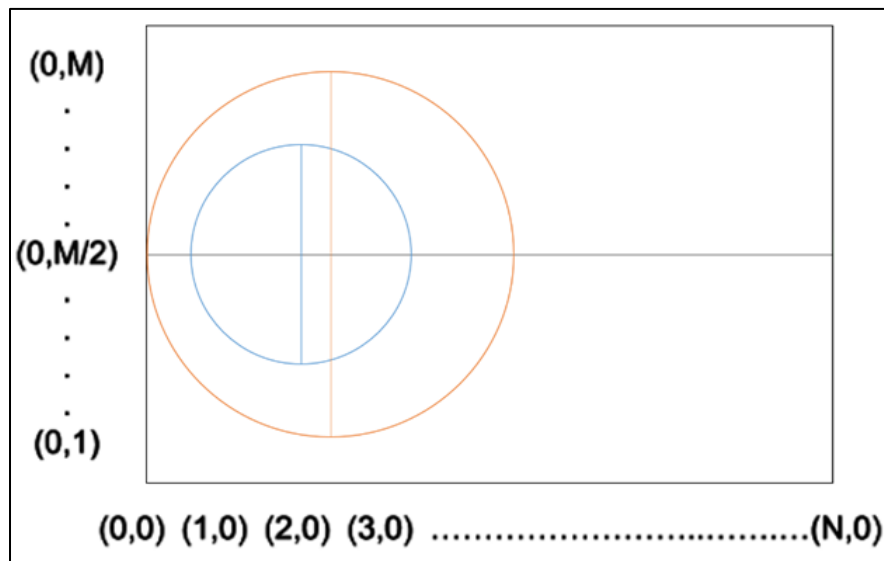
La única instancia en la que se podría tener certeza de que este algoritmo obtendría la solución óptima sería cuando el conjunto solución coincide con el conjunto de blancos. En otras palabras, cuando ocurra que no hay solapamientos y por tanto se pueden seleccionar todos los blancos iniciales.

- Análisis de instancias donde no hay solución óptima:

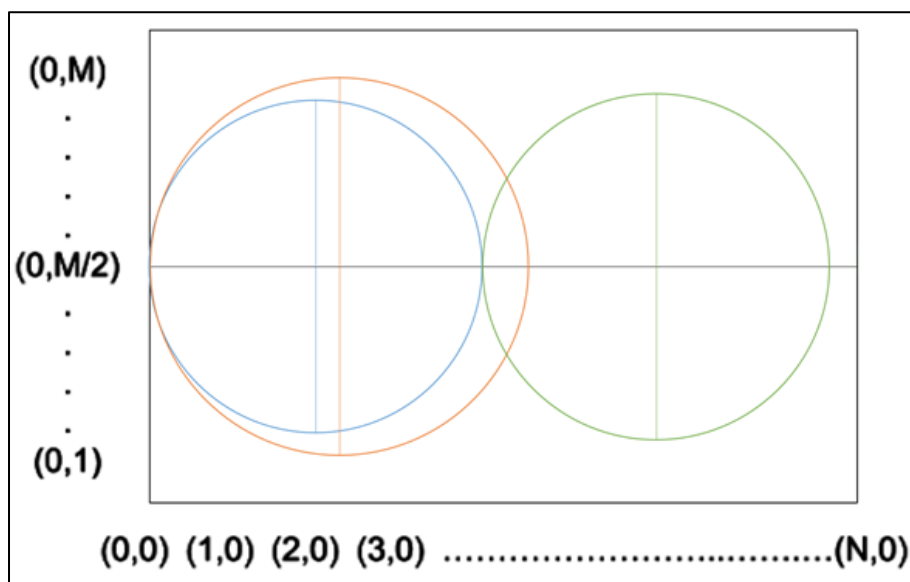
Ejemplos en donde la solución no coincide con la solución óptima:



En la imagen anterior vemos que se tomaría el círculo azul, descartando el rojo que tiene más área. Lo mismo ocurre en el siguiente caso:



Vemos que otro posible criterio de selección local sería escoger cada vez el blanco con mayor área, lo que sí habría acertado en estos ejemplos. No obstante, este fallaría si se tiene:



Se escogería el círculo rojo, descartando el azul y el verde que sumados tienen mayor área. El algoritmo propuesto sí funcionaría. Hay distintos algoritmos voraces además de estos dos, como por ejemplo escoger el

blanco cuyo borde izquierdo estuviera más a la izquierda, pero para cada uno existen contraejemplos. Estos algoritmos voraces no funcionan en este problema por la naturaleza de los subproblemas que se obtienen cuando se selecciona uno de los blancos, en vista de que hay una fuerte dependencia de un subproblema con los demás más pequeños.

- **Modificando el problema:**

Si modificamos el problema de tal forma que no se busque maximizar el área de impacto de los misiles sino maximizar la cantidad de misiles que se disparan, el criterio aplicado anteriormente, además de fallar en el sentido de hallar siempre la solución óptima, fallaría también en el enfoque pues no era éste su propósito. El cambio propuesto sería ordenar el conjunto de blancos ascendentemente en función del área y aplicar el mismo método de selección, añadiendo a la solución el círculo de menor área e ir escogiendo nuevamente, en orden, aquellos blancos que no se solapan con el último blanco adherido a la solución. La idea sería entonces que un blanco de menor área tiene menos probabilidad de solaparse con otros blancos, aumentando la posibilidad de que se pueda seleccionar un mayor número de blancos. Otra opción, quizá más válida, sería ordenar los blancos ascendentemente en función del número de blancos con el que cada uno se solapa, pero esto añadiría al costo el cálculo del número de solapamientos de cada blanco, lo que afectaría significativamente el tiempo que tomaría el proceso conforme la cantidad de blancos a examinar fuese mayor.

- **Eficiencia en tiempo:**

La complejidad del algoritmo se deriva de la complejidad de las dos partes que lo conforman, ordenar el arreglo de blancos y recorrerlo usando el criterio de escogencia como filtro para construir la solución.

Si analizamos la complejidad de ordenar los blancos usando mergesort obtenemos el siguiente costo:

$$T(n) = n \log n$$

Si analizamos el costo de recorrer el arreglo por medio del *for*, cada iteración del proceso de revisar los blancos después del primero es de un costo constante pues simplemente compara la suma de los radios y la distancia entre los centros de los blancos. Su costo sería:

$$T(n) = \sum_{i=1}^{n-1} 1 = (n - 1)$$

Finalmente, tenemos que el costo computacional total está dado por:

$$T(n) = n \log n + (n - 1)$$

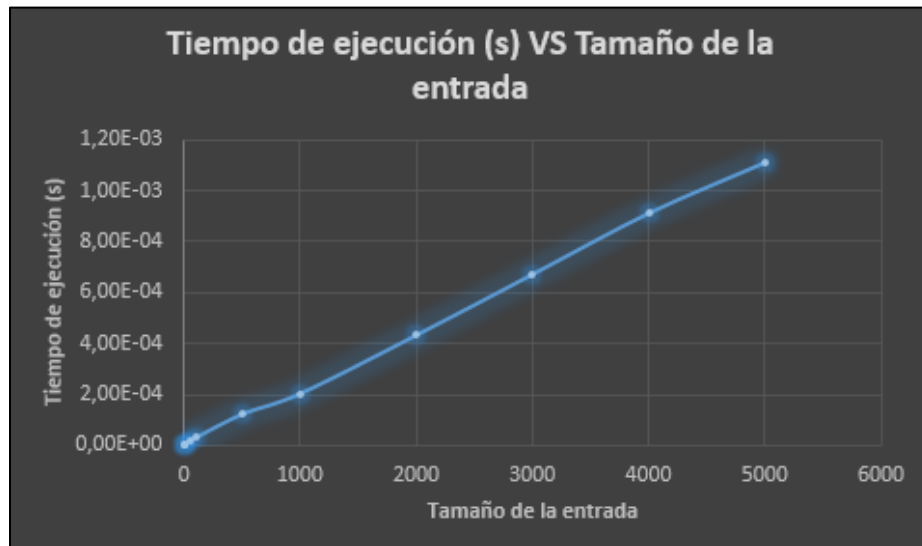
Donde el factor dominante es $n \log n$, dando como resultado:

$$T(n) = O(n \log n)$$

Tomando algunas pruebas con entradas de distintos tamaños y promediando los tiempos medidos se obtuvieron los siguientes resultados:

Tamaño de la entrada	Tiempo de ejecución (s)
1	2,00E-06
5	2,00E-06
10	4,00E-06
50	2,10E-05
100	3,00E-05
500	1,22E-04
1000	2,03E-04
2000	4,33E-04
3000	6,72E-04
4000	9,12E-04
5000	1,11E-03

La siguiente gráfica representa los resultados obtenidos en la tabla anterior:



Analizando el comportamiento de la gráfica se observa que el crecimiento sigue un orden $n \log n$, así como el orden de crecimiento de la complejidad del algoritmo calculada anteriormente.

- Consideraciones Finales:

El ejercicio de abordar este problema por medio de distintas técnicas de programación ha permitido evidenciar cómo el análisis y el diseño de algoritmos juega un papel de capital importancia en la resolución de problemas a través de la computación. La programación dinámica, cuando es aplicable, permite obtener siempre la solución óptima del problema, con el inconveniente de requerir más recursos, tiempo y por ello ser más costosa que una opción como la del enfoque voraz. En un problema como este, en el que cada subproblema es dependiente de otros más pequeños y el cálculo de la solución de cada uno puede necesitarse varias veces, la programación dinámica constituye la solución más adecuada, pues permite almacenar dichas soluciones para evitar calcularlas más de una vez. Podría verse a la programación dinámica como una forma muy refinada del enfoque de divide y vencerás, pues hemos visto cómo el aplicarla ofrece una complejidad mucho más reducida que la de éste último ($O(2^n)$ vs). La programación voraz, en este problema, sacrifica el obtener la solución óptima a cambio de conseguir una complejidad aun menor. Es cierto que, respecto a ésta, el criterio de escogencia local define qué tan aceptable será la solución, pero en el problema se ha observado cómo para distintos criterios se divisan rápidamente contraejemplos con instancias sencillas del problema. Asimismo, algunos criterios (como el de escoger en dependencia de cuántos blancos se solapa uno de ellos) pueden mejorar el resultado de la aproximación voraz, pero aumentando la complejidad y de ser así puede ser mejor ir directamente por el enfoque dinámico ya que se pondría en juego una ventaja de la forma voraz al incrementar la complejidad.

La naturaleza de cada problema dictará qué tan válido o pertinente es un algoritmo, pero con distintas técnicas bien aplicadas podrá responderse a la necesidad de manera consecuente. Así, si el resultado tiene que ser el mejor posible, la programación dinámica es una alternativa de mucho atractivo. Sin embargo, en el escenario de que deba primar la rapidez, limitando la precisión del resultado pero sin llegar a hacerlo inaceptable, un algoritmo voraz con un criterio de escogencia bien pensado e implementado puede ser un camino acertado.