Taller 1 - FADA
Juan Sebastian Velasquez Acevedo
1744936

Parte 1 - Análisis de algoritmos

1. En cada literal se encontrará un algoritmo escrito en pseudocódigo. Para cada algoritmo, debe expresar su orden de complejidad en notaci´on Theta, es decir en términos de Θ( ). La operación básica a considerar en los siguientes algoritmos es print(“Hola Mundo”):

   a. funcion1(n)

```
for(i = 2; i <= 5; i = i * i)
        print("Hola Mundo")
for(i = 1; i <= n; i = i * 5 )
        print("Hola Mundo")
```

   T(n)= $log_5(n)$ -> **Θ**($log_2 n$)

   b. funcion2(n)

```
for(i = 1; i <= n; i = i * 2)
        for(j = 1; j <= n; j = j+j)
                print("Hola Mundo")
```

   T(n)= $\log(n)^2$ -> **Θ**($log_2 n$)

   c. funcion3(n)

```
if(n == 1) then
        { print("Hola Mundo") }
else
        { print("Hola Mundo")
          funcion3(n/2)
          funcion3(n/2)
```

   T(n)= **Θ**($n^{log_3 2}$) por el caso 1 del método maestro

   d. funcion4(n)

```
if(n == 1) then
        { print("Hola Mundo") }
else
```

```
{ print("Hola Mundo")
for(j = 1; j <= 3; j = j + 1)
        funcion4(n/3)}
```

T(n)= **Θ**(n) por método maestro caso 1

2.

```
1.FuncionMisterio(a, b)
2.   if b == 0
3.     return 1
4.   if b == 1
5.     return a
6.   if even(b)
7.     c = FuncionMisterio(a, b/2)
8.     return c*c
9.   else
10.    c = FuncionMisterio(a, b/2)
11.   return c*c*a
```

Donde a y b son números enteros positivos

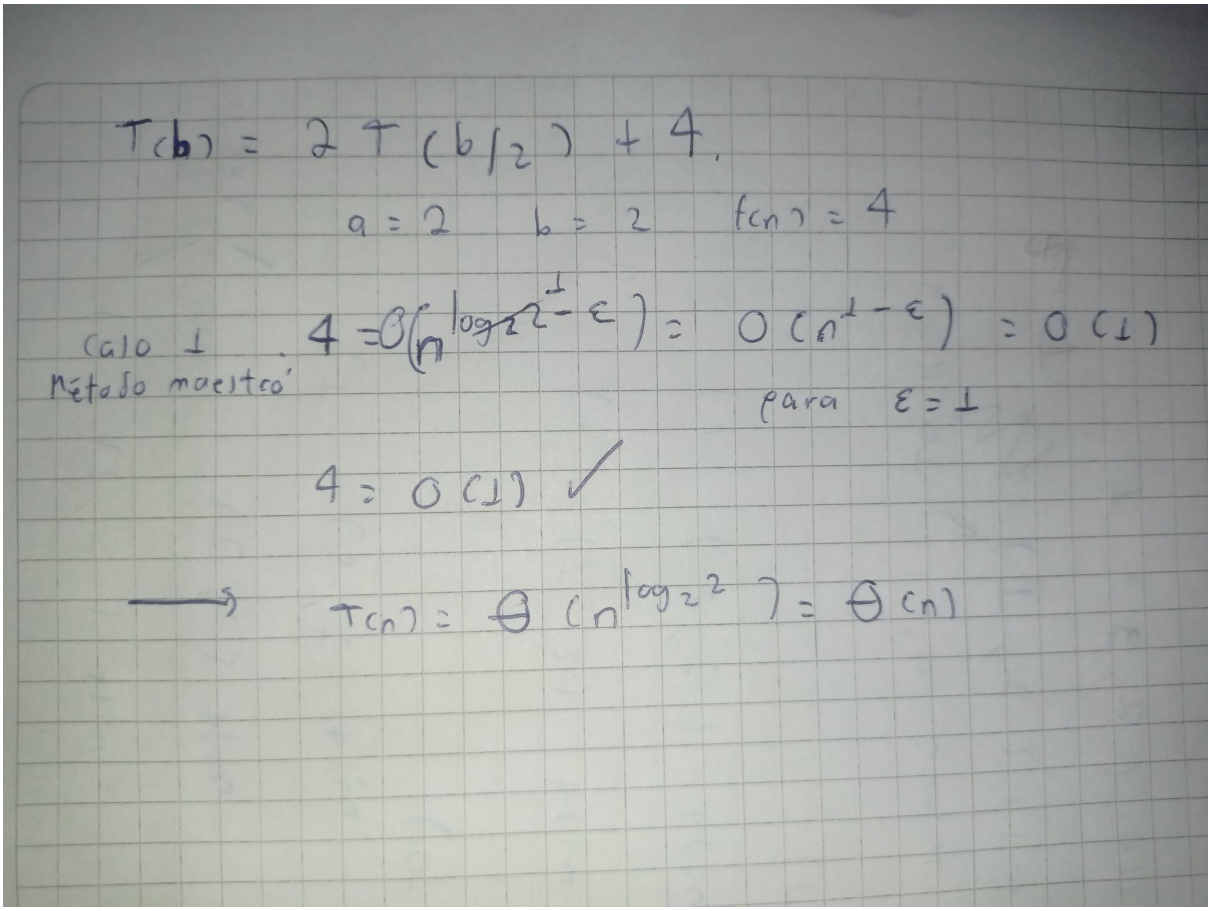
1. Explique qué hace el algoritmo (asuma que existe una función even que recibe un número y devuelve true si es par, false si es impar, y lo hace en O(1)). Asuma que a y b son 2 valores enteros positivos.

El algoritmo básicamente lo que hace es multiplicar el número a, b veces, entonces se podría decir que es un algoritmo que realiza la operación a^b. Donde la estrategia consiste en multiplicar sucesivamente por **a** a partir del valor de **b.** He ir dividiendo por 2 el valor de b hasta que llegue a 1.

2. ¿Cuál es el mínimo de veces que se ejecuta la línea 7? ¿Cuál es el máximo? ¿De qué depende de las veces que se ejecute en las siguientes llamadas al método?

El mínimo de veces que se ejecuta la línea 7 son 0 veces, cuando cuando **b** es de la forma 2^n-1. Que es cuando se quiere elevar **a** al cuadrado. El máximo de veces que se puede ejecutar la línea 7 es $log_2 b$ cuando **b** es de la forma 2^n. La línea 7 se ejecuta dependiendo si en la anterior llamada, la operación (b/2) = even. Considerando que se toma la parte entera de la división.

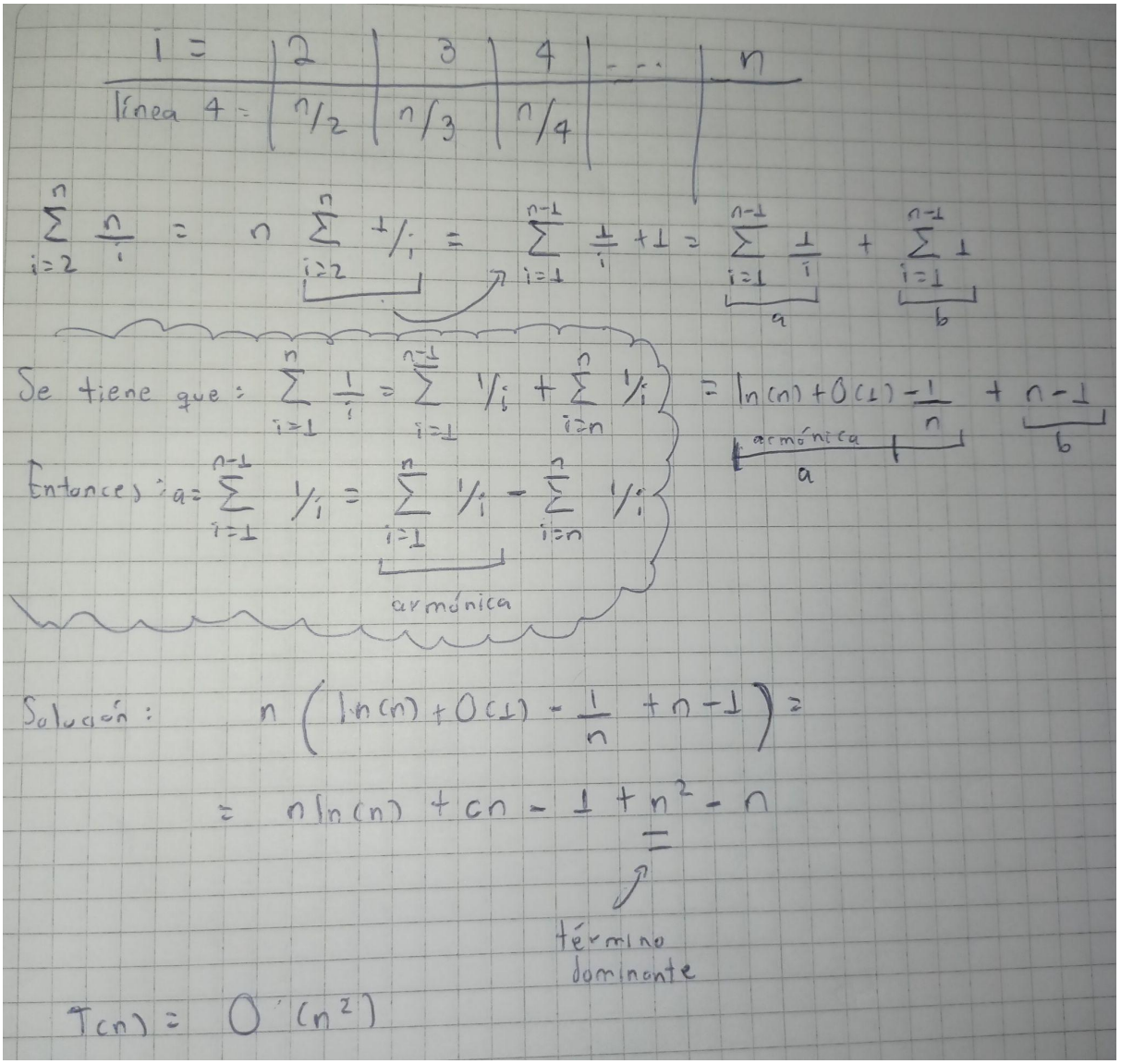
3. Diga la complejidad del algoritmo.



3.

```
1. FuncionMisterio(n, arr[1..n])
2.   for i = 2 to n, i = i + 1
3.     if(arr[ i ] ¡= true)
4.       for j = 2*i to n, j = j + i
5.         arr[ j ] = true
```

Donde n es un entero y arr un arreglo de booleanos de tamaño n, inicializado en false.
Escriba y explique la complejidad del algoritmo presentado:



Parte 2 - Diseño de algoritmos

4.

Solución Planteada:

Una solución a este problema podría ser primero que todo, ordenar el array multidimensional. Primero con respecto al nivel de juego en **A** y si las dos parejas tienen el mismo nivel de juego **A,** entonces se ordena con respecto a su nivel de juego en **B.** Por ejemplo, si tenemos dos parejas: {{3,2},{1,4}}, al ordenarlas: {{1,4},{3,2}} porque primero revisamos si los niveles **A** de cada pareja son iguales, si no, ordena por **A.** Si por ejemplo tenemos dos parejas con el mismo nivel **A**, por ejemplo :{{3,4},{3,1}}, entonces las ordenamos por su nivel **B:** {{3,1},{3,4}}.

Para lo anterior se usa el merge-sort, pero un poco modificado con el fin de que acepte arrays multidimensionales, y que realice el proceso explicado anteriormente.

Luego de tener las parejas ordenadas, se buscaría la capacidad de la primera pareja conformada por el primer jugador y el último jugador: {(a(first) + a(last)), (b(first) + b(last)) }.Tomando en cuenta que: (a(first) + a(last)) = Nivel_A a buscar y (b(first) + b(last)) = Nivel_B a buscar. Luego esta capacidad es la que dictamina cuál es la **capacidad a buscar** por el resto de parejas.

Entonces, ahora se examina el segundo jugador con el penúltimo jugador, comprobando que la capacidad de esta pareja sea igual a la **capacidad a buscar.** Quiere decir que: (a(second) + a(last-1)) = Nivel_A a buscar y (b(second) + b(last-1))=Nivel_B a buscar. Nivel_A y Nivel_B siempre estará dictado por la primera y última pareja, y básicamente lo que se hace, es que el resto de parejas tengan la misma capacidad que la de la primera y última pareja. Siguiendo la lógica de primero-último, segundo-penúltimo, tercero-antepenúltimo y así sucesivamente hasta que no haya más parejas por revisar.

Si todas las capacidades son iguales, entonces se puede armar un torneo justo, si hay alguna que no es igual, entonces **no** se puede armar un torneo justo, esto funciona porque tenemos el arreglo multidimensional ordenado como se explicó previamente.

Pseudocódigo:

En todo el pseudocódigo consideramos dos valores:
variable numParejas
constante sizeParejas=2

El número de parejas asumimos que siempre es par, porque si no lo fuese, no tendría sentido el problema, inmediatamente podríamos deducir que no es un juego justo, porque algún jugador se quedaría sin pareja. El número de parejas varía dependiendo del ejemplo. El sizeParejas, representa el nivel A=Asociada al index 0 y el nivel B= Asociado al index 1, de cada jugador.

Merge-Sort:

```
1
2 mergeSort(int arr[numParejas][sizeParejas], int l, int r)
3   if (l < r)
4     int m = l + (r - l) / 2
5     mergeSort(arr, l, m)
6     mergeSort(arr, m + 1, r)
7     merge(arr, l, m, r)
```

Merge:

```
void merge(int arr[numParejas][sizeParejas], int p, int q, int r)
  int n1 = q - p + 1;
  int n2 = r - q;
  int L[n1][sizeParejas], M[n2][sizeParejas];
  for (int i = 0; i < n1; i++)
    L[i][0] = arr[p + i][0]
    L[i][1] = arr[p + i][1]
  for (int j = 0; j < n2; j++)
    M[j][0] = arr[q + 1 + j][0]
    M[j][1] = arr[q + 1 + j][1]
  int i, j, k
  i = 0
  j = 0
  k = p
  while (i < n1 && j < n2)
    if(L[i][0] == M[j][0])
      if (L[i][1] <= M[j][1])
        arr[k][1] = L[i][1]
        arr[k][0] = L[i][0]
        i++
      else
        arr[k][1] = M[j][1]
        arr[k][0] = M[j][0]
        j++
    else
      if (L[i][0] <= M[j][0])
      arr[k][1] = L[i][1]
      arr[k][0] = L[i][0]
      i++
      else
        arr[k][1] = M[j][1]
        arr[k][0] = M[j][0]
        j++
    k++;
  while (i < n1)
    arr[k][0] = L[i][0]
    arr[k][1] = L[i][1]
    i++
    k++
  while (j < n2)
    arr[k][0] = M[j][0]
    arr[k][1] = M[j][1]
    j++
    k++
```

Función pairs, que determina si es posible tener un juego justo o no, usando el merge-sort previamente presentado:

```
bool pairs(int arr[numParejas][sizeParejas])
  mergeSort(arr, 0, numParejas - 1)
  bool pairs_done=false
  int last=numParejas-1
  int begin=0
  int sumNivelA=arr[begin][0]+arr[last][0]
  int sumNivelB=arr[begin][1]+arr[last][1]
  last--
  begin++
  while(begin<(numParejas/2) && last>=(numParejas/2)){
    int sumNivelA_Actual=arr[begin][0]+arr[last][0]
    int sumNivelB_Actual=arr[begin][1]+arr[last][1]
    if((sumNivelA_Actual==sumNivelA) &&  (sumNivelB_Actual==sumNivelB))
      pairs_done=true
    else
      pairs_done=false
    last--
    begin++
    return pairs_done
```

Complejidad:

La complejidad del merge-sort modificado sigue siendo nlog(n), y la complejidad del resto del código, es **O(n),** porque el índice begin recorre hasta la mitad del array, y el índice last recorre también hasta la mitad. Para poder determinar un juego justo, begin recorre n/2 y last n/2, cuando se analizan todas las parejas, y todas cumplen con lo planteado. Puede ser menor la complejidad, si se detecta que el juego no puede ser justo antes de recorrer todo el array. Pero en el peor de los casos es **O(n).** La complejidad total del algoritmo es O(n)+O(n(log (n))). Por lo tanto, si escogemos la cota superior, la complejidad del algoritmo sería O(n (log (n))).

5.
Solución Planteada:
Una posible solución podría ser, primero que todo, ordenar los arrays de los precios de apartamentos y de dinero de los aplicantes. Ya que esto me ayuda a comparar más fácilmente el dinero tentativo con cada apartamento. Luego de ordenarlos, tener una función que itere hasta que se acaben el número de apartamentos o el número de compradores**.**

Luego, tenemos dos booleanos que nos permiten saber si el precio actual del apartamento ≥ Oferta Mínima o si el precio actual del apartamento es ≤ Oferta Máxima. Si el precio está entre las dos ofertas, significa que se compra el apartamento, y analizamos el siguiente apartamento con el siguiente comprador. Si el precio no está en el intervalo se miran dos casos:

1. Si el precio fue mayor a la oferta mínima, si así fue, entonces analizamos el siguiente comprador con el apartamento actual.

2. Si el anterior caso no fue, entonces quiere decir que el precio fue menor a la oferta máxima. Lo que indica que nos olvidamos de este apartamento actual y cambiamos a analizar el siguiente apartamento con el comprador actual.

Todo el anterior análisis, funciona si y sólo si los arreglos están ordenados de manera ascendente.

Pseudocódigo:

```
1  renta(int m, int n, int k, int M[], int N[])
2    mergeSort(M,0,m-1)
3    mergeSort(N,0,n-1)
4    int i=0
5    int j=0
6    int mayorCantidad=0
7    while(i<m && j<n)
8      bool mayorOfertaMinima = (M[i] >= N[j]-k)
9      bool menorOfertaMaxima = (M[i] <= N[j]+k)
10     if(mayorOfertaMinima && menorOfertaMaxima)
11       i++
12       j++
13       mayorCantidad++
14     else if(mayorOfertaMinima)
15       j++
16     else
17       i++
18     return mayorCantidad
```

Complejidad:

La operación mergeSort fue la que usamos en clase, y esta tiene una complejidad = O(n(log(n))). En este algoritmo lo invocamos dos veces. Entonces esto me da una complejidad m(log(m)) para la primera llamada y n(log(n)) para la segunda llamada.

En el while, si escogemos el peor de los casos, es cuando se acaben los apartamentos para ofrecer **= O(m).** O si ya no hay más demanda de compradores **= O(n).** En cualquier caso, podemos generalizar diciendo que el while tiene una complejidad lineal **O(n)**.

Si sumamos las complejidades = O(m(log(m))) + O(n(log(n))) + O(n), es lo mismo que decir: O(n(log(n))) + O(n(log(n))) + O(n). Entre los anteriores términos, la cota superior la marca nlog(n), ya que nlog(n)≥ n, y representa mejor el comportamiento del algoritmo en el peor de los casos. Es por esto que la complejidad es **O(n(log(n))).**

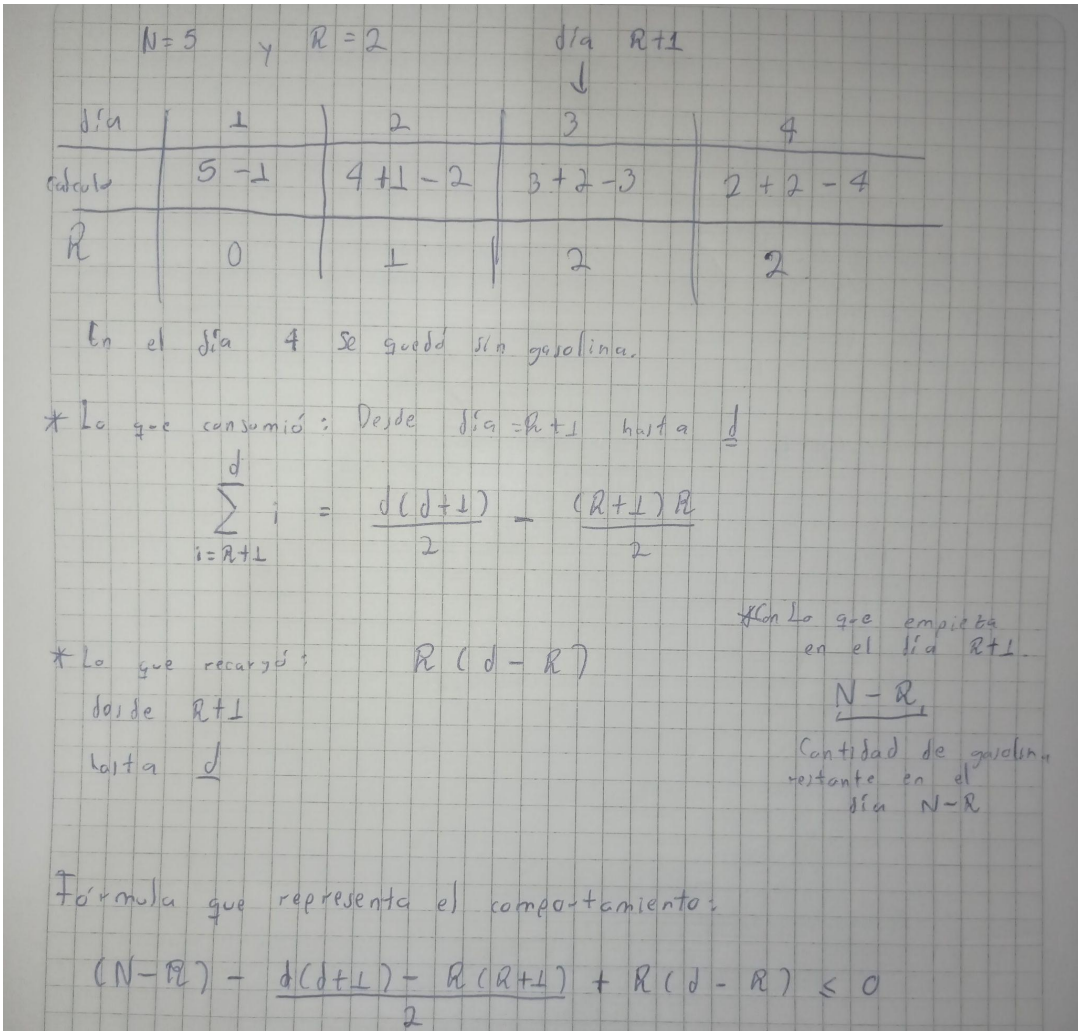
6.

Solución Planteada:

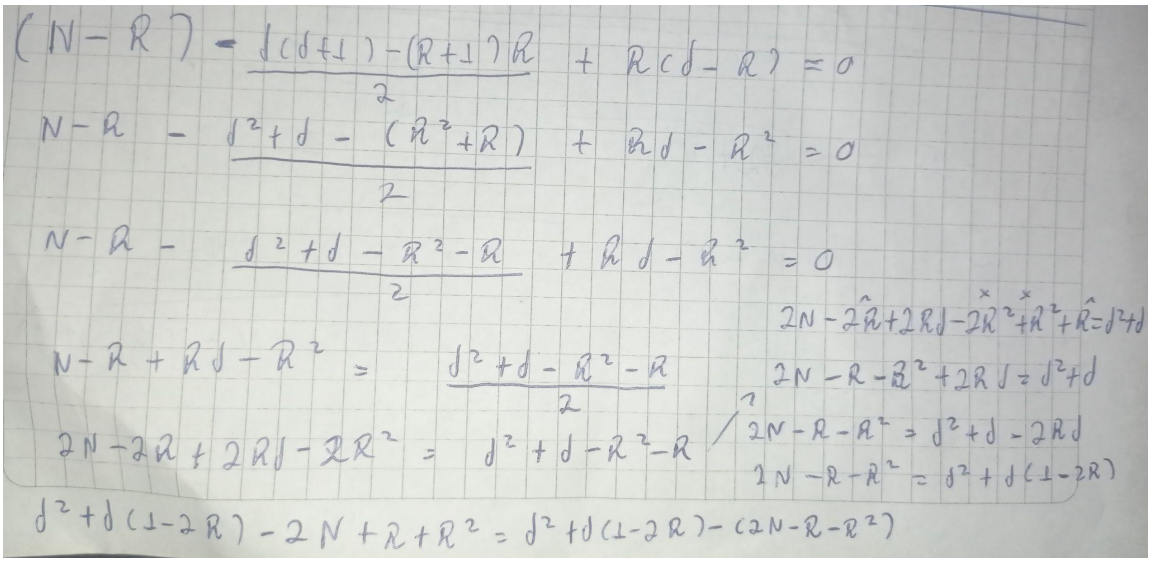
Para este problema, nos indican que tenemos que resolverlo con una complejidad máxima de O(log(n)). Esto nos índica que la solución debe ser veloz y no tan compleja en código. Para esto entonces, se cree necesario plantear una fórmula que relacione la cantidad inicial de gasolina = **N**, la cantidad consumida de gasolina y la cantidad recargada de gasolina, para un día **d**. Para esto se analiza el problema, y encontramos un comportamiento predecible que podemos representar con sumatorias, luego presentar un fórmula y de ahí despejar **d.** El problema es que el comportamiento predecible se empieza a notar desde el día **d=R+1,** siendo R el valor de recarga que me dan en la entrada**.** Entonces se tienen que presentar los comportamientos de las variables anteriormente mencionadas desde el día R+1 hasta un día **d.**

Lo anterior se presenta en los siguientes cálculos:

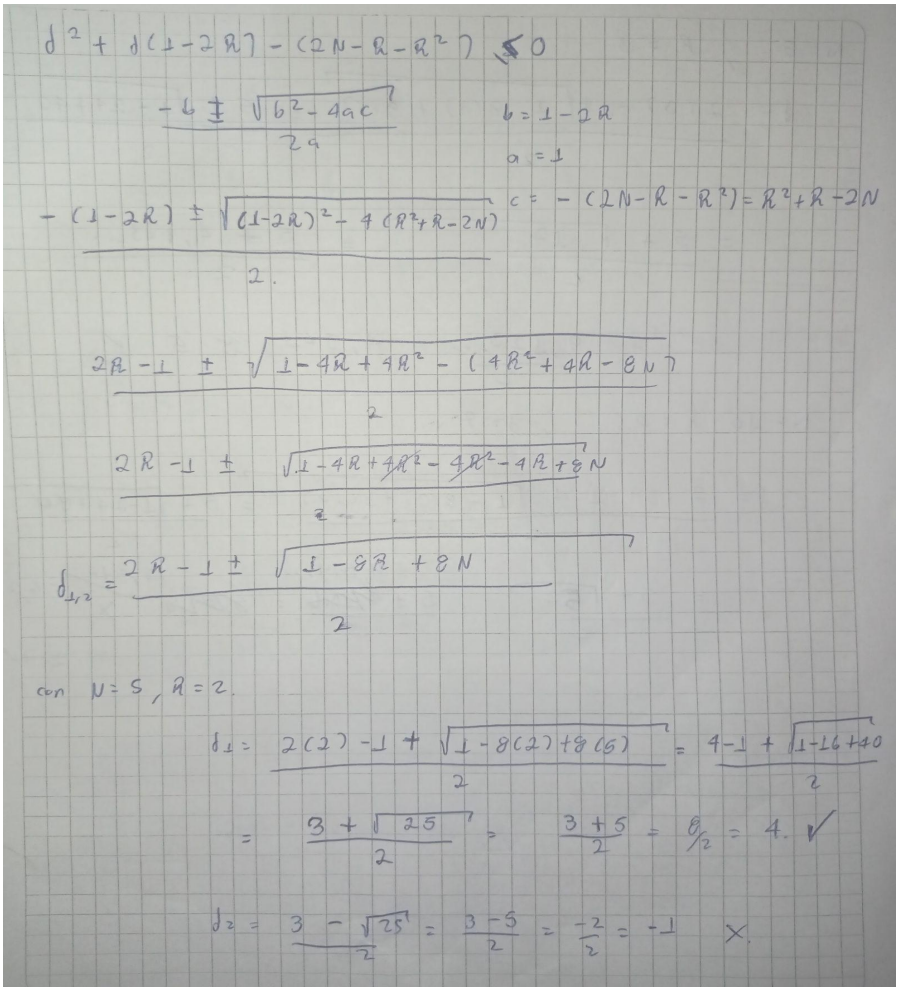
1. Análisis



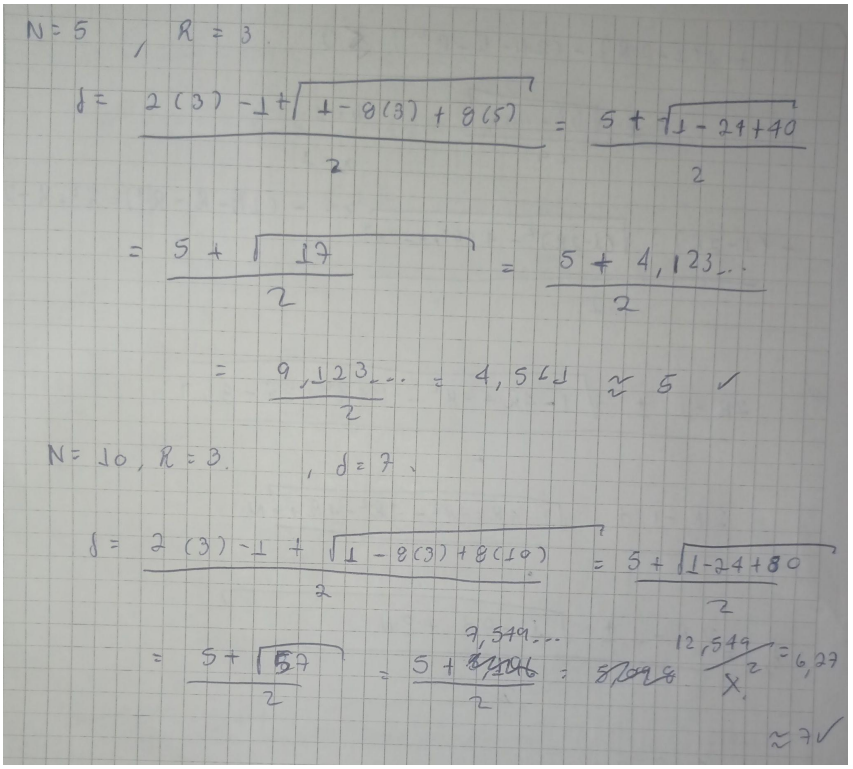
2. Consolidación de la fórmula



3. Despeje de **d** y testeo con R=2 y N=5



4. Otros tests con R=3 y N=5, R=3 y N=10



Como podemos observar, se obtiene una fórmula que me da dos raíces, en este caso escogemos la raíz positiva. Y en los tests, en caso tal que luego de reemplazar los valores,el día me arroje un número decimal, éste se aproxima al número entero mayor más cercano, esto se da porque si se escoge el menor es como estuviese dejando gasolina sin gastar, y la idea es que se gaste por completo sin importar si nos queda en negativo la resta en el día final, que probablemente sería si me varo.

Pseudocódigo:

```
1 dia(r, n)
2   float dia=0
3   float raiz= sqrt((1-(8*r)+(8*n)))
4   dia= (((2*r)-1)+ raiz) /2
5   diaAprox=ceil(dia)
6   diaAprox
```

La función ceil me permite tomar un número decimal, y aproximarlo a su entero más cercano. Si el número a aproximar ya era entero, entonces devuelve el mismo entero.

Complejidad:
Como podemos observar, se logra una complejidad menor a la esperada, ya que si solo planteamos una fórmula y la calculamos, estamos haciendo operaciones meramente atómicas sin recursión ni ciclos. La complejidad de este algoritmo es **O(1)**. Que representa la cota superior de todas las operaciones que se muestran en el pseudocódigo.

Implementación del punto 4,5 y 6 de diseño de algoritmos en el lenguaje de programación c++, y otros archivos.

https://github.com/Odzen/Algorithm-Analysis-and-Design---FADA - Git

https://drive.google.com/drive/folders/11ANbOZITQ8_dEzlt7SS-8iL39Q8RHE2c?usp=sharing - Drive