Campus Santa Fe

Evidencia análisis y diseño de algoritmos

Curso:

Programación de estructuras de datos y algoritmos fundamentales

Alumnos:

Sebastian Enrique Moncada González	A01027028
Alejandro Arouesty Galván	A01782691
Samuel Roberto Acevedo Sandoval	A01026893

Profesor:

Esteban Castillo Juarez

4 de diciembre del 2022

Introducción:

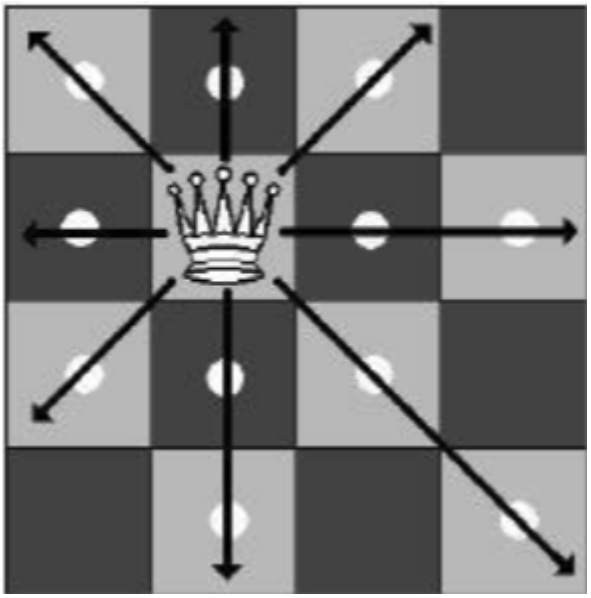
Esta evidencia tiene como propósito evaluar nuestras habilidades en el análisis y diseño de algoritmos con estructuras de datos y los programas a realizar tendrán que ser en uno de los siguientes lenguajes de programación: C, Python o ambos.

Para esta evidencia se propone resolver el problema de las “Ocho reinas (N reinas)” esperando cumplir los siguientes objetivos:

- Una solución en C, Python o ambos basada en una estrategia de **algoritmo de fuerza bruta.**
- Una solución en C, Python o ambos basada en una estrategia de **programación dinámica** o una **estrategia de divide y vencerás.**
- Un análisis algorítmico **específico** y **general** de ambos programas utilizando la notación asintótica **Big-O.**

Metodología:

Cómo se mencionó anteriormente se va a resolver el problema de las ***ocho reinas***, este consiste en colocar ***n*** reinas en un tablero de ajedrez de ***n*** x ***n*** de tal manera que no sea posible que dos reinas se capturen entre sí, es decir, que no estén en la misma fila, columna ni diagonal. Para resolverlo se usarán dos distintos tipos de algoritmos, usando ***fuerza bruta*** y ***programación dinámica*** **o** ***divide y vencerás***. (Riojas Cañari, 2005)



El algoritmo de ***fuerza bruta*** consiste en enumerar sistemáticamente todos los posibles candidatos para la solución y comprobar que cada candidato satisface el enunciado del problema. Si bien este algoritmo es simple de implementar y siempre llegará a una solución, si es que esta existe, los costos de implementación son proporcionales al número de soluciones candidatas.

El algoritmo ***divide o vencerás*** consiste en separar un problema en subproblemas que se parecen al original, de manera recursiva los resuelve y por último los combina para resolver el problema original. Esta técnica es la base de los algoritmos eficientes para casi cualquier tipo de problema como algoritmos de ordenamiento, tales cómo ***quicksort***, ***mergesort*** entre otros.

Sección experimental y resultados:

Programación dinámica:

Primero hay que aclarar que las reinas van a ser tratadas en este código como estructuras; de hecho se tomó como influencia el programa de lista circular hecho previamente. Tiene

funciones básicas como saber si no hay reinas existentes (el equivalente a listaVacia), la cantidad de reinas que hay, crear las reinas en la que se le pasa como parámetro la cantidad a crear y borrar todas las reinas para que después de hacer todo el proceso no ocupen memoria, imprimir las reinas el cual se hace de dos formas: como una lista de coordenadas y como tablero, la cual es 100% por diseño y a la que realmente no le dedicamos tiempo en hacerla dinámica ya que no es indispensable su uso.

Realmente la mayor parte del trabajo y en donde nos enfocamos más fue en las funciones *solveBoard, placeQueen* y *randomizeQueen*. En estas es que se hace todo el proceso para resolver el problema de las 8 reinas y en pocas palabras lo que hacen es que las coordenadas en *x* de las reinas se coloca de manera ordenada en su creación, pero sus coordenadas en *y* se colocan de manera aleatoria, así que se revisa que ninguna reina esté en la misma fila que otra reina ni que entre tenga un pendiente constante con respecto a otra reina, ya que esto significa que está en su diagonal. En caso de que esto suceda se vuelve a calcular de manera aleatoria lo coordenada en *y* de la reina con coordenada en *x* mayor, ya que la comparación siempre se realiza de una reina con las que tienen una coordenada en *x* menor, sin llegar a las que la tienen mayor.

Esto deja un problema y es que a veces sucede que el programa se queda atorado porque llegando a las últimas reinas ya las únicas ubicaciones posibles darían una resolución no exitosa, por lo que se tiene un número limitado de 64 intentos para colocar cada reina en un lugar que satisfaga la solución de problema. De no ser así, se borran las reinas actuales y se vuelven a crear nuevas reinas, haciendo que se reinicie todo el proceso hasta que llegue a un resultado exitoso.

Ejemplo de ejecución:

```
Reinas iniciales:

Reina [1] = (1, 4)
Reina [2] = (2, 6)
Reina [3] = (3, 2)
Reina [4] = (4, 6)
Reina [5] = (5, 7)
Reina [6] = (6, 8)
Reina [7] = (7, 5)
Reina [8] = (8, 2)


0  0  0  0  0  0  0  0
0  0  X  0  0  0  0  X
0  0  0  0  0  0  0  0
X  0  0  0  0  0  0  0
0  0  0  0  0  0  X  0
0  X  0  X  0  0  0  0
0  0  0  0  X  0  0  0
0  0  0  0  0  X  0  0
```

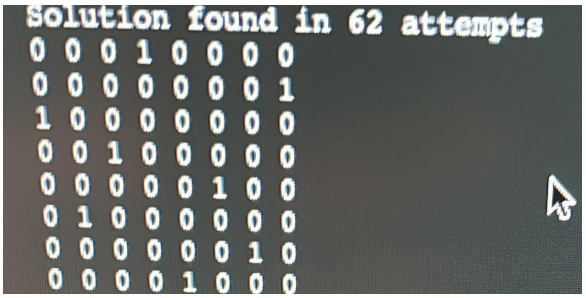
```
Tablero exitoso:

Reina [1] = (1, 5)
Reina [2] = (2, 1)
Reina [3] = (3, 8)
Reina [4] = (4, 6)
Reina [5] = (5, 3)
Reina [6] = (6, 7)
Reina [7] = (7, 2)
Reina [8] = (8, 4)


0  X  0  0  0  0  0  0
0  0  0  0  0  0  X  0
0  0  0  0  X  0  0  0
0  0  0  0  0  0  0  X
X  0  0  0  0  0  0  0
0  0  0  X  0  0  0  0
0  0  0  0  0  X  0  0
0  0  X  0  0  0  0  0
```

Fuerza Bruta:

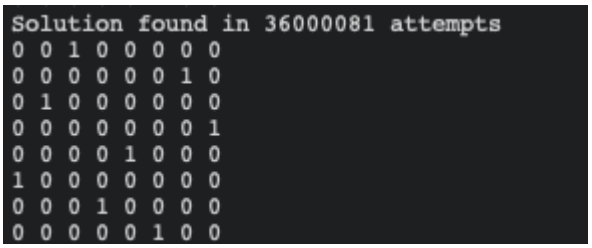
Declaramos nuestro tablero y tenemos la función "*print_b*" para imprimirlo. Después tenemos nuestra función "*check_attack*" para revisar si el número en el tablero es una reina (1) o no (0) y para determinar si está en algún carril de ataque. En este programa estamos jalando i y j como números enteros random y hacemos el chequeo. Cada que se hace un chequeo cuenta como un intento el cual lo vamos agregando a un counter de cuantos intentos llevamos en esa Iteración. Sin embargo nos encontramos con un problema. Había ocasiones donde el programa lograba posicionar 7 reinas en el tablero pero ya no la octava.

Es por eso que implementamos la condición para que cada 1 millón de intentos se reseteara el tablero. De esta forma aseguramos que el programa encontrará una solución eventualmente.

Como podemos observar, hay ocasiones en donde el programa toma muy pocos intentos y hay otros en los que le tomó millones de intentos.



62 intentos

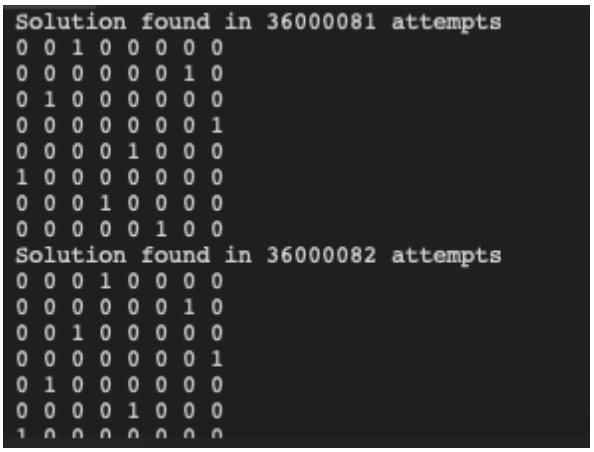


36 millones de intentos

Por último, para comprobar que nuestras condiciones servían de manera correcta, creamos otro tablero llamado "*board_test*" con una posible solución ya implementada.

```
62
63  board_test = [[0,0,0,1,0,0,0,0],
64   [0,0,0,0,0,0,1,0],
65   [0,0,1,0,0,0,0,0],
66   [0,0,0,0,0,0,0,1],
67   [0,1,0,0,0,0,0,0],
68   [0,0,0,0,1,0,0,0],
69   [1,0,0,0,0,0,0,0],
70   [0,0,0,0,0,1,0,0]]
71
```

Y le pedimos al algoritmo que bajo las mismas condiciones nos dijera cuantos intentos después de encontrar la primera solución le tomaba volver a encontrar una solución correcta.



Podemos observar que nuestro programa funciona de manera correcta ya que solo necesito de 1 intento más para “encontrar” la solución correcta con el segundo tablero.

Notación asintótica:

En esta tabla podemos ver una breve explicación de la notación Big-O específica que tienen distintas instrucciones.

ID	Línea de código ejemplo	Notación esp.	Explicación
1	`#include <stdio.h>`	O (1)	Incluir una librería
2	`#define MAXIMO 5`	O (1)	Definir una variable
3	`int valores [MAXIMO]`	O (1)	Declarar una variable no primitiva
4	`int frente`	O (1)	Declarar una variable primitiva
5	`instancia -> frente = 0`	O (1)	Asignación a una variable
6	`(instancia -> frente == instancia -> atras)`	O (1)	Comparación entre dos variables
7	`printf("La cola esta vacia\n")`	O (1)	Imprimir
8	`return 1`	O (1)	Devolver un valor
9	`return`	O (1)	Terminar la ejecución de una función
10	`colaLlena (instancia)`	O (n)	Llamar una función. 'n' es la complejidad general de la función, ya que al llamarla va a tener que realizar esa cantidad de procesos
11	`instancia -> valores [instancia -> atras]`	O (1)	Obtener valor de una lista en un índice específico.
12	`instancia -> atras++;`	O (1)	Cambiar la cantidad numérica que tiene una variable (aumentar o disminuir)
13	`pow(i, 2)`	O (1)	Elevar un número al cuadrado
14	`struct cola instanciaMain`	O (n)	Declarar una estructura. 'n' es la complejidad general de la declaración de la estructura, ya que al ejecutarla va a tener que realizar esa cantidad de procesos
15	`srand(time(NULL))`	O (1)	Se obtiene la seed con la que se van a calcular los números aleatorios
16	`rand() % 5 + 1`	O (1)	Obtiene un número random
17	`scanf("%d", &n);`	O (1)	Se pide al usuario un valor
18	`malloc(sizeof(struct nodo))`	O (1)	Reservar un espacio de memoria
19	`free(temporal)`	O (1)	Borrar y desocupar un espacio de memoria
20	`&&`	O (1)	Operación lógica

Notación asintótica programa de *fuerza bruta* (Python):

inicio:

```
import random      # O(1)

board = [[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0]]    # O(1)
```

Específica: O(2) → General: O(1)

Su notación es constante.

print_b:

```
def print_b(board):
    for i in range(8):
        for j in range(8):
            print(board[i][j], end=" ") # O(n^2+2)
        print() # O(n)
# O(n^2+n+2)
```

Específica: O($n^2 + n + 2$) → General: O(n^2)

Su notación es cuadrática.

check_attack:

```
def check_attack(i, j):
    for k in range(8):
        if board[i][k] == 1 or board[k][j] == 1:
            return True # O(2n+1)
    for k in range(8):
        for l in range(8):
            if (k + l == i + j) or (k - l == i - j):
                if board[k][l] == 1:
                    return True #O(3n^2+2)
# O(3n^2+2n+3)
```

Específica: O($5n^2$+4n+3) → General: O(n^2)

Su notación es cuadrática.

while True:

```
while True:
    i = random.randint(0, 7) # O(1)
    j = random.randint(0, 7) # O(1)

    if not check_attack(i, j): #O(n^2)
        board[i][j] = 1 # O(1)
        counter += 1 # O(1)
        print("counter:", counter) # O(1)
        print_b(board) # O(1)

    counter_try += 1 # O(1)
    if counter_try > 1000000: #O(1)
        counter = 0 #O(1)
        counter_try = 1 #O(1)
        board = [[0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0]] # O(1)

    total_try += 1 # O(1)

    if counter == 8: # O(1)
        print("Solution found in", total_try, "attempts") #O(1)
        print_b(board) #O(n^2)
        break # O(1)

#O(2n^2+16)
```

Específico: O($2n^2$+16) ⟶ General: O(n^2)

Su notación es cuadrática.

inicializando:

```
counter = 0 # O(1)
counter_try = 1 # O(1)
total_try = 1 # O(1)
# O(3)
```

Específica: O(3) ⟶ General: O(1)

Su notación es constante.

Para la otra parte del código con el "***board_test"*** es prácticamente igual lo de arriba, por lo que consideramos que solo repetiriamos y con la información que tenemos ahora podemos decir que la notación del programa en general es de **O(n^2).**

Notación asintótica programa de *programación dinámica o divide y vencerás*

(C) :

Inicio:

```c
#include <stdio.h> // O(1)
#include <stdlib.h> // O(1)
#include <time.h>  // O(1)
```

Específica: O(3) → General: O(1)

Su notación es constante.

struct queen:

```c
struct queen
{
    int x_pos, y_pos; // O(2)
    struct queen *following; // O(1)
    struct queen *previous; //O(1)
}; // O(4)
```

Específica: O(4) → General: O(1)

Su notación es constante.

start:

```c
struct queen *start = NULL; // O(2)
```

Específica: O(2) → General: O(1)

Su notación es constante.

randomNumber:

```
int randomNumber(int start, int end)
{
    return rand() % (end - start + 1) + start; // O(2)
} //O(2)
```

Específica: O(2) ⟶ General: O(1)

Su notación es constante.

noQueens:

```
int noQueens(void)
{
    if (start == NULL) // O(1)
        return 1; // O(1)
    else
        return 0; // O(1)
} // O(3)
```

Específica: O(3) ⟶ General: O(1)

Su notación es constante.

queensAmount:

```
int queensAmount(void)
{
    if (noQueens() == 1) // O(2)
        return 0;  // O(1)
    // O(3)

    struct queen *temporary; // O(1)
    int length = 0; // O(1)
    temporary = start;  // O(1)
    do
    {
        length++; // O(1)
        temporary = temporary->following; // O(1)
    } while (temporary != start);  //O(1
    // O(2n+1)
    return length;  // O(1)
} // O(2n+1+1+3+3) -> O(2n+8)
```

Específica: O(2n+8) ⟶ General: O(n)

Su notación es lineal.

createQueens:

```
void createQueens(int queens_amount)
{
    int i; // O(1)
    for (i = 0; i < queens_amount; i++) // O(3)
    {
        struct queen *new_queen = malloc(sizeof(struct queen)); // O(3)

        if (new_queen == NULL) // O(1)
        {
            printf("No se puede crear una nueva reina"); // O(1)
            return; // O(1)
        } // O(3)

        new_queen->x_pos = i + 1; // O(2)
        new_queen->y_pos = randomNumber(1, 8); // O(1)
        // O(3)
        if (noQueens() == 1) // O(2)
        {
            new_queen->previous = new_queen; //O(1)
            new_queen->following = new_queen; //O(1)
            start = new_queen; //O(1)
        } // O(5)
        else
        {
            new_queen->previous = start->previous; // O(1)
            new_queen->following = start; // O(1)
            start->previous->following = new_queen; // O(1)
            start->previous = new_queen; // O(1)
        } // O(4)
    } // O(18n+3)
} // O(18n+4)
```

Específica: O(18n+4) → General: O(n)

Su notación es lineal.

deleteQueens:

```
void deleteQueens()
{
    if (noQueens() == 1) // O(2)
    {
        printf("La lista está vacía\n"); // O(1)
        return; // O(1)
    } // O(3)

    struct queen *index = start->previous; // O(2)

    while (start != NULL) // O(1)
    {
        if (index == start) // O(1)
        {
            start = NULL; // O(1)
            free(index); // O(1)
            return;  // O(!)
        } // O(4)
        else
        {
            index->previous->following = index->following; // O(1)
            index->following->previous = index->previous;  // O(1)

            struct queen *temporary = index;  // O(1)
            index = index->previous; // O(1)
            free(temporary); // O(1)
        } // O(5)
    } // O(10n+1)
} // O(10n + 1 + 5) -> O(10n+6)
```

Específica: O(10n+6) → General: O(n)

Su notación es lineal.

printQueens1:

```c
void printQueens(void)
{
    printf("\n"); // O(1)

    if (noQueens() == 1) //O(2)
    {
        printf("No existen reinas a mostrar\n"); // O(1)
        return; //O(1)
    } //O(4)

    int i; // O(1)
    struct queen *temporary; // O(1)

    temporary = start; //O(1)
    int length = queensAmount(); //O(n+1)

    for (i = 0; i < length; i++) //O(3)
    {
        printf("Reina [%d] = (%d, %d)\n", temporary->x_pos, temporary->x_pos, temporary->y_pos); // O(1)
        temporary = temporary->following; // O(1)
    } // O(2n+3)

    printf("\n"); // O(1)
} // O(3n+13)
```

Específica: O(3n+13) ⟶ General: O(n)

Su notación es lineal.

printQueens2:

```
void printQueens2(void)
{
    printf("\n"); // O(1)
    if (noQueens() == 1) // O(2)
    {
        printf("No existen reinas a mostrar\n"); //O(1)
        return; //O(1)
    } //O(4)

    int i, j; //O(2)
    struct queen *temporary = start; //O(2)

    for (i = 0; i < 8; i++) //O(3)
    {
        for (j = 0; j < 8; j++) //O(3)
        {
            if (temporary->y_pos == i + 1) //O(2)
            {
                if (temporary->x_pos == j + 1) //O(2)
                    printf("X  "); //O(1)
            }
            else
                printf("0  "); //O(1)
            temporary = temporary->following; //O(1)
        }  // O(7n+3)
        temporary = start; //O(1)
        printf("\n"); // O(1)
    } // O(7n^2+5n+3)
    printf("\n"); //O(1)
} // O(7n^2+5n+13)
```

Específica: O($7n^2$+5n+13) ⟶ General: O(n^2)

randomizeQueen:

```
void randomizeQueen(struct queen *changed_queen, struct queen *checkpoint)
{
    struct queen *index = start; // O(2)
    changed_queen->y_pos = randomNumber(1, 8); //O(2)
    // O(4)

    while (index != checkpoint->following) // O(1)
    {
        if (index->y_pos == changed_queen->y_pos) // O(1)
        {
            changed_queen->y_pos = randomNumber(1, 8); // O(2)
            index = start; // O(1)
        }
        else
            index = index->following; // O(1)
    } // O(5n+1)
} // O(5n + 1 + 4) -> O(5n+5)
```

Específica: O(5n+5) ⟶ General: O(n)

Su notación es lineal.

placeQueen:

```
int placeQueen (struct queen *checkpoint)
{
    int x_diff, y_diff, flag = 0, cntr = 1; // O(6)
    float slope; //O(1)
    struct queen *index = start; // O(1)
    // O(8)
    while (index != checkpoint) // O(1)
    {
        if (cntr > 64) // O(1)
            return 1; // O(1)
        // O(2)
        x_diff = index->x_pos - checkpoint->x_pos; // O(2)
        y_diff = index->y_pos - checkpoint->y_pos; // O(2)
        slope = (float)y_diff / (float)x_diff;  // O(4)
        // O(8)
        if (y_diff == 0 || slope == 1 || slope == -1) //O(5)
        {
            flag = 1; //O(1)
            checkpoint->y_pos = randomNumber(1, 8); // O(2)
            index = start; // O(1)
            cntr ++; // O(1)
        } // O(10)
        else
        {
            flag = 0; //O(1)
            index = index->following; //O(1)
        } //O(2)
    } // O(22n+1)
    return 0; //O(1)
} // O(22n+1+1+8) -> O(22n+10)
```

Específica: O(22n+10) ⟶ General: O(n)

Su notación es lineal.

solveBoard:

```
void solveBoard(void)
{
    int i, repeat = 0; // O(3)
    struct queen *index = start; // O(2)
    //O(5)
    for (i = 0; i < 8; i++) // O(3)
    {
        repeat = placeQueen(index); // O(n+1)

        if (repeat == 0) // O(1)
            index = index->following; // O(1)
        // O(2)
        else
        {
            deleteQueens(); // O(n)
            createQueens(8); // O(n)
            index = start; // O(1)
            i = -1; // O(1)
        } // O(2n+2)
    } // O(3n^2 + (2 + 2 + 1)n + 3) -> O(3n^2+5n+3)
    return; // O(1)
} // O(3n^2+5n+3+1+5) -> O(3n^2+5n+9)
```

Específica: O($3n^2 + 5n + 9$) → General: O(n^2)

Su notación es cuadrática.

main:

```
int main(void)
{
    srand(time(NULL)); // O(1)

    createQueens(8); // O(n)
    printf("\nReinas iniciales:\n"); //O(1)
    printQueens(); // O(n)
    solveBoard(); // O(n^2)
    printf("Tablero exitoso:\n"); //O(1)
    printQueens(); // O(n)
    deleteQueens(); // O(n)
} // O(n^2+4n+3)
```

Específica: O(n^2+4n+5) → General: O(n^2)

Su notación es cuadrática.

Después de analizar el programa podemos decir que este tiene una complejidad de **O(n^2).**

Conclusiones:

La ventaja de usar el algoritmo de fuerza bruta es que tienes garantizado encontrar una solución al problema. Sin embargo, el programa tiene una gran desventaja la cual pudimos apreciar en varias ocasiones. Su fundamento es muy simple; probar todas las combinaciones posibles hasta encontrar la solución. Esto, naturalmente, presenta un desafío en la mayoría de problemas ya que el número de soluciones candidatas puede llegar a ser extremadamente alto. Esto conlleva a que el programa tenga una notación asintótica muy alta y que el tiempo de ejecución sea, por lo general, también muy alto.

Si bien hacer un programa de fuerza bruta puede ayudar mucho a la sencillez y velocidad al programarlo, en casos como este un programa así puede llevar muchísimo tiempo para ejecutarse, lo que hace que no sea factible ni siquiera usarlo. Aquí es en donde entra la gran ventaja de los programas dinámicos: pueden reducir en gran manera el tiempo de ejecución del programa; a pesar de que lleve más tiempo programarlo, ya sólo con poder probarlo en poco tiempo es una ventaja muy significativa.

En general consideramos que nos llevamos un gran aprendizaje derivado de este trabajo, pues pudimos aprender a diferenciar y a utilizar estos 2 tipos de soluciones para resolver programas y así en un futuro poder determinar cuál de estos nos servirá más para utilizar en cualquier problema que se nos presente. Además aprendimos a poder determinar la complejidad de los programas que utilizamos o programamos gracias a la notación asintótica "Big-O" y así poder determinar si es que se puede hacer algo más para mejorar esta.

Referencias

Riojas Cañari A. C., (2005) *Conceptos, algoritmo y aplicación al problema de las N-reinas* Recuperado de https://sisbib.unmsm.edu.pe/bibvirtualdata/monografias/basic/riojas_ca/cap4.pdf

Khan Academy (s.f) *Algoritmos de divide y vencerás* Recuperado de https://es.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms

Búsqueda de fuerza bruta (s.f.) Recuperado de https://hmn.wiki/es/Brute-force_search