Análisis de la práctica Tablero de Ajedrez

El funcionamiento del programa implementado se describe a continuación:

Dadas n piezas de tipo $x_0, \dots, x_{n-1} \in X = \{reina, rey, torre, peon, caballo, alfil\}$ y un tablero de $n\,x\,n$ casillas, la posición inicial que ocupara cada pieza dependerá de la posición de la pieza anterior, a menos de que sea la pieza_0 que estará en (0,0). Es decir, la pieza_1 se colocará en la siguiente casilla de donde se encuentre la pieza_0. Entonces, la pieza_1 está en (0,1) y así sucesivamente.

El recorrido que hacen las piezas una vez que están en una posición inicial estará haciéndose por medio de una función que se encargará de mover la última pieza casilla a casilla, primeramente. Y una vez que haya llegado a la ultima casilla del tablero esta ya no debe moverse, pero esto indicará que debe moverse una pieza anterior. Lo último causara que la pieza que ya había llegado a la ultima casilla del tablero, se posicione enseguida de la pieza que se acaba de mover.

Con un ejemplo estará mas claro. A continuación, se muestra como lo hace con 3 piezas. Ojo, no estamos contemplando el tipo de pieza, solamente vemos como es que se van moviendo las piezas dentro del tablero.

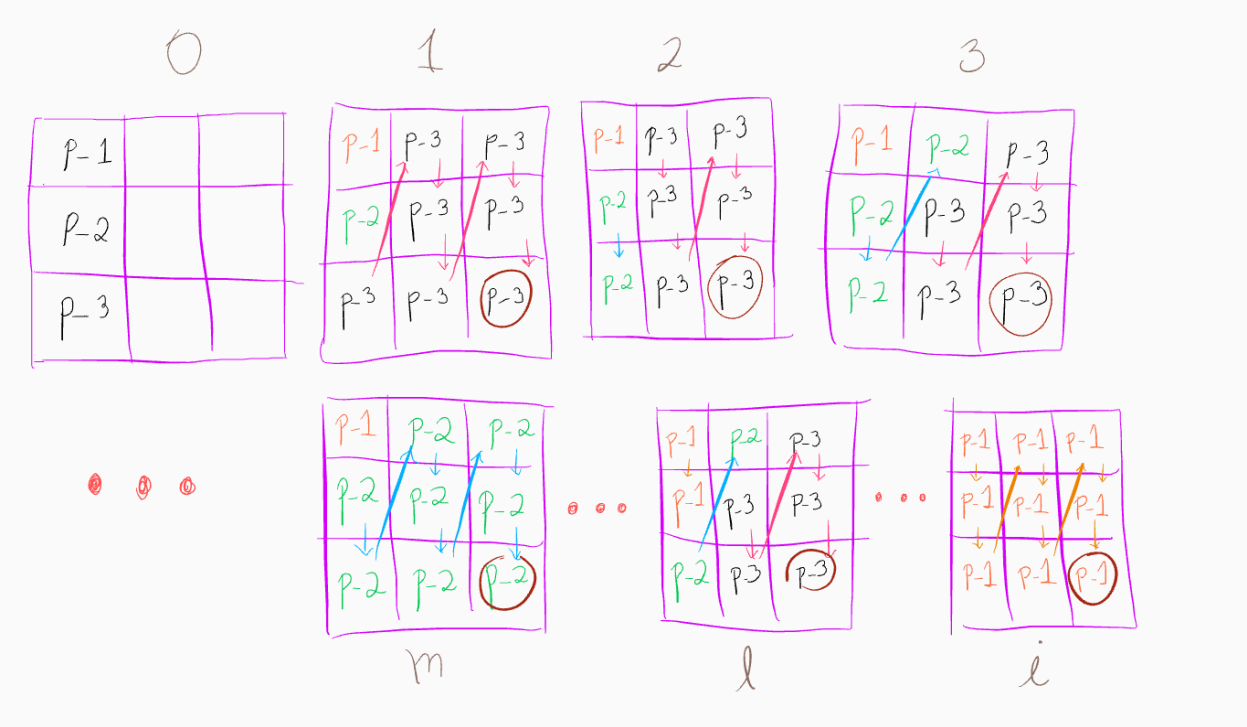


Imagen_1

Cada que se coloca una pieza se estará validando su posición, además de que se ejecutaran otras funciones. Su complejidad será analizada mas adelante.

Primeramente, analizaremos como se van encontrando las soluciones, con el siguiente árbol y el mismo ejemplo que se mostró del tablero $3\ x\ 3$:

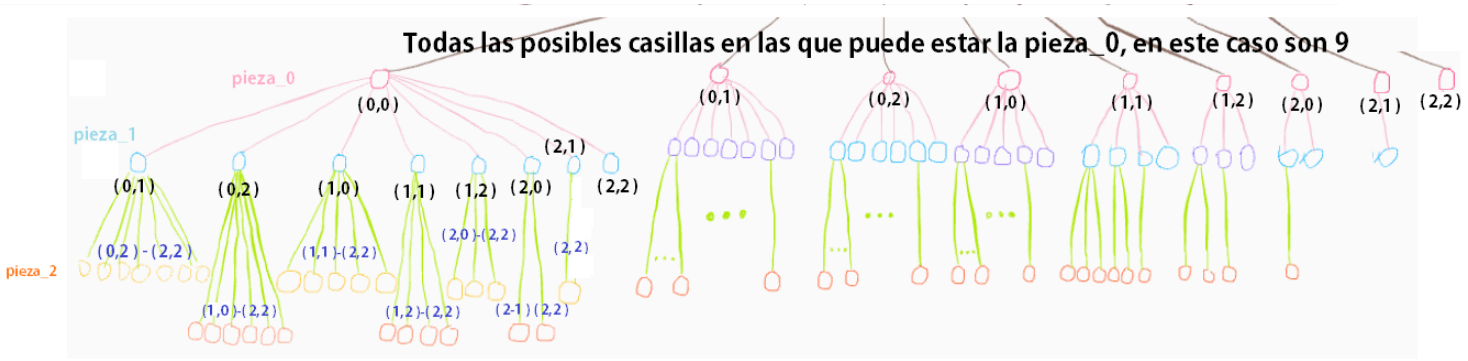
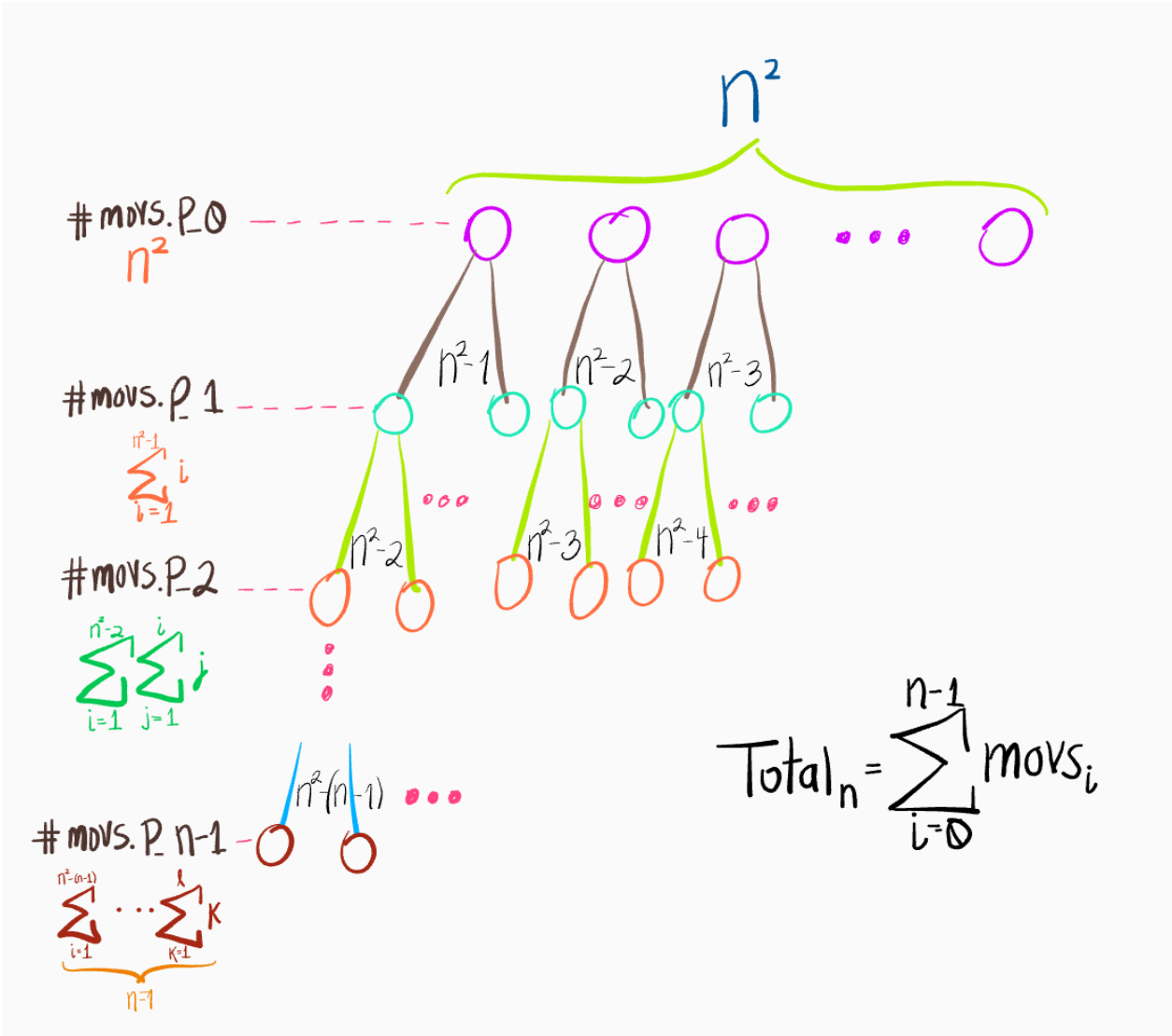


Imagen_2

Y de forma general se ve de la siguiente forma:



Por lo que la cantidad de llamadas que hace recursivamente para encontrar las soluciones de n piezas son:

$$total_n = n^2 + \underbrace{\sum_{i=1}^{n^2-1} i}_{\text{pieza 0}} + \underbrace{\sum_{i=1}^{n^2-2}\sum_{j=1}^{i} j}_{\text{pieza 1}} + \cdots + \underbrace{\sum_{i=1}^{n^2-(n-1)} \cdots \sum_{k=1}^{l} k}_{\text{pieza 2}} \quad \text{pieza n-1}$$

Por ejemplo, cuando se tienen solo dos piezas la cantidad de veces que llamará a la función $movePiece$ será 10 veces, para encontrar todas las posibles soluciones. A continuación, se muestra una tabla donde se puede apreciar como crece el numero de llamadas.

Cantidad de piezas	Cantidad de llamadas
2	10
3	129
4	2516
5	68405

Pero, ¿qué relación existe entre la cantidad de piezas y la cantidad de llamadas?

La verdad es que, haciendo algunas pruebas con ayuda de la calculadora, porque yo ya no vi la forma de simplificar la expresión de arriba, más que sustituir la última suma de cada termino con su expresión equivalente:

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

Y encontré una aproximación, aunque varía mucho, pero me ayudo a darme una idea de la complejidad de mi algoritmo. La relación entre los valores de una columna y otra, solamente tomando en cuenta los 4 datos que muestro en la tabla:

$$cantidadLlamadas_n = n^{n+1+\frac{1}{10}k}$$

Donde 0 < k < 10, solamente así lo pude aproximar mas a los datos que tengo en la tabla.

En el peor de los casos, cuando las piezas tengan una posición inicial como la que se muestra en el tablero_0 de la imagen_1, la complejidad de buscar todas las soluciones posibles es $O(n^{n+1+\frac{1}{10}k})$, o sea que la complejidad es exponencial. Tomar en cuenta que por cada que se lleva a cabo se ejecutan más

```
void movePiece(int piece, int pieceMat[][3], int pieceNum, int *typePiece, int *count){
    if(pieceMat[piece-1][1]!=-1)//si aun no se le ha asignado un lugar dentro del tablero
        pieceNextSquare(piece, pieceMat, pieceNum);//mueve las piezas a la siguiente casilla
    else
        positionInitial(piece, pieceMat, pieceNum);//lo pongo en una posicion inicial

    int val = validatePosition(piece, pieceMat, pieceNum, typePiece); //validara que no me maten y que no mate
```

funciones.

La primera función de la que se apoya para ir moviendo la pieza por todo el tablero es:

```
93  void pieceNextSquare(int piece, int pieceMat[][3], int pieceNum){
94      int x = pieceMat[piece-1][1], y = pieceMat[piece-1][2];
95
96      if(x == pieceNum-1 && y == pieceNum-1){
97          //lo ponemos enseguida de su anterior cuando ya se encuentre en la casilla (pieceNum-1, pieceNum-1)
98          int xPrev = pieceMat[piece-2][1], yPrev = pieceMat[piece-2][2];
99          if( yPrev == pieceNum-1 )
100             pieceMat[piece-1][2]=0;
101         else
102             pieceMat[piece-1][2]=yPrev+1;
103         if( pieceMat[piece-1][2]==0 )
104             pieceMat[piece-1][1]=xPrev+1;
105         else
106             pieceMat[piece-1][1]=xPrev;
107     }else{
108         //lo ponemos en la siguiente casilla
109         if( y == pieceNum-1 )
110             pieceMat[piece-1][2]=0;
111         else
112             pieceMat[piece-1][2]+=1;
113         if( pieceMat[piece-1][2] == 0 )
114             pieceMat[piece-1][1]+=1;
115     }
116 }
```

$T_{pieceNextSquare} = T_{Linea94} + T_{Linea96}$
$T_{Linea94} = 6$
$T_{Linea96} = T_{condicional1} + f\{T_{linea98-106}, T_{linea109-114}\}$
$T_{linea98-106} = 6 + T_{condicional2} + f\{T_{linea100}, T_{linea102}\} + T_{condicional3} + f\{T_{linea104}, T_{linea106}\}$
$T_{linea98-106} = 6 + 2 + f\{2,3\} + 2 + f\{3,2\}$

Como sacaremos el peor de los tiempos, tomaremos el máximo de $f\{x,y\}$:

$T_{linea98-106} = 6 + 2 + 3 + 2 + 3 = 16$
$T_{linea109-114} = T_{condicional4} + f\{T_{linea110}, T_{linea112}\} + T_{condicional5} + T_{linea114}$
$T_{linea109-114} = 2 + f\{2,3\} + 2 + 3 = 7 + \max\{2,3\} = 10$
$T_{Linea96} = 5 + f\{16, 10\} = 5 + \max\{16,10\} = 21$

Finalmente, tenemos:

$T_{pieceNextSquare} = 4 + 16 = 20$

Por lo que la complejidad en el peor de los casos de la función $pieceNextSquare$ es $\boldsymbol{O(1)}$.

La siguiente función que se lleva a cabo es la de inicializar, pero solamente se lleva a cabo n veces, es decir, mientras que las piezas no estén en el tablero.

```
270 void positionInitial(int piece, int pieceMat[][3], int pieceNum){
271     if(piece == 1){//si la pieza 1 no ha sido acomodada la pongo en la primera casilla del tablero
272         pieceMat[piece-1][1]=0;
273         pieceMat[piece-1][2]=0;
274     }else{//si es cualquier otra pieza, la pongo enseguida del anterior
275         int xPrev = pieceMat[piece-2][1];
276         int yPrev = pieceMat[piece-2][2];
277         if(yPrev==pieceNum-1)
278             pieceMat[piece-1][2]=0;
279         else
280             pieceMat[piece-1][2]=yPrev+1;
281         if(pieceMat[piece-1][2]==0)
282             pieceMat[piece-1][1]=xPrev+1;
283         else
284             pieceMat[piece-1][1]=xPrev;
285     }
286     return;
287 }
```

Calculemos su complejidad en el peor de los casos:

$T_{positionInitial} = T_{condicional1} + f\{T_{lineas272-273}, T_{lineas275-284}\}$
$T_{lineas272-273} = 4$
$T_{lineas275-284} = 6 + T_{condicional2} + f\{T_{linea278}, T_{linea280}\} + T_{condicional3} + f\{T_{linea282}, T_{linea284}\}$
$T_{lineas275-284} = 6 + 2 + max\{2,3\} + 2 + max\{3,2\} = 6 + 2 + 3 + 2 + 3 = 16$
$T_{positionInitial} = 1 + max\{4,16\} = 1 + 16 = 17$

Por lo que la complejidad en el peor de los casos de la función $positionInitial$ es $\boldsymbol{O(1)}$.

La siguiente función es una de las mas importantes del codigo, ya que es la que me indicara la situación de la pieza en la casilla que se le posiciono.

Inicialmente tenemos algunas declaraciones y la activación de una bandera en caso necesario:

```
117 int validatePosition(int piece, int pieceMat[][3], int pieceNum, int *typePiece){
118     int x = pieceMat[piece-1][1], y = pieceMat[piece-1][2], lim = 0, xaux, yaux;
119     //movimientos del caballo, hacia adelante "hx-hy" & hacia atras "hxx-hyy"
120     int hx[] = {1, 2, 2, 1}, hy[] = {-2, -1, 1, 2}, hxx[] = {-1, -2, -2, -1}, hyy[] = {-2, -1, 1, 2};
121     //movimientos del rey, hacia adelante "rx-ry" & hacia atras "rxx-ryy"
122     int rx[] = {0, 1, 1, 1, 0}, ry[] = {-1, -1, 0, 1, 1}, rxx[] = {0, -1, -1, -1, 0}, ryy[] = {-1, -1, 0, 1, 1};
123
124     if(x == pieceNum-1 && y == pieceNum-1)// estas en la ultima casilla del tablero
125         lim = 1;
```

Para posteriormente, en el codigo, validar cada pieza que ya se encuentre en el tablero para asegurarnos de que no la maten y que no mate a ninguna otra pieza que ya esta puesta. Si pasa cualquiera de las dos se tendrá que mover de lugar.
Como estamos analizando el orden en el peor de los casos, tomaremos en cuenta que se trata de un tablero que estará ocupado solo por reinas, ya que es la pieza que más validaciones tiene que hacer.

La validación se hace en dos partes, dos $switch()$. El primer $switch()$ checa que las piezas que ya están puestas no maten a la que acaban de posicionar, por lo que se encuentra dentro de un ciclo $for()$.

```
126     for(int i=0; i<piece-1; i++){
127
128         int xv = pieceMat[i][1], yv = pieceMat[i][2];
129         //checar si alguna pieza no me mata
130         switch(typePiece[i]){
131             case 1:{//Reina
132                 if(x == xv || y == yv)
133                     return (lim==1)?-1:0;
134                 xaux = xv; yaux = yv;
135                 while( xaux < pieceNum && yaux < pieceNum ){
136                     xaux++; yaux++;
137                     if(xaux == x && yaux == y)
138                         return (lim==1)?-1:0;
139                 }
140                 xaux = xv; yaux = yv;
141                 while( xaux < pieceNum && yaux >= 0 ){
142                     xaux++; yaux--;
143                     if( xaux == x && yaux == y )
144                         return (lim==1)?-1:0;
145                 }
146                 break;
```

$T_{validatePosition} = T_{linea118} + T_{linea120} + T_{linea122} + T_{linea124} + T_{validarPeorCaso-Reina}$

$T_{validarPeorCaso-Reina} = T_{ini} + (M+1)T_{comparacion} + MT_{codigo} + MT_{actualiz}$

El peor de los casos, la reina que se tiene que posicionar es la última, es decir, es la $reina_n$. Por lo que $n-1$ reinas ya están en el tablero. Entonces $M = n-1$.

$T_{comparacion} = 2$

$T_{codigo} = 4 + T_{evaluacionVar} + T_{case1Reina} + T_{case2Reina}$

$T_{case1Reina} = T_{if} + 2 + T_{while1} + 2 + T_{while2}$

$T_{if} = T_{condicion-if} + T_{codigo-if} = 3 + 3 = 6$

$T_{condicion-if} = 3$

$T_{codigo-if} = 3\ (comparación, elección\ y\ return)$

Para el análisis de los ciclos $while()$ veamos como funciona. Este primer $while\ ()$ verifica la diagonal positiva de todas las reinas, por lo que tomaremos el caso donde se encuentra en la casilla media de la última columna.

Entonces la diagonal tendrá $\frac{n}{2}$ casillas que verificar. Lo considerare de esta forma porque de forma general es incierto poner una posición adecuada con la cual calcular la complejidad de verificar las diagonales, ya que si ponemos que tiene n casillas que verificar, quiere decir que estamos en las esquinas y entonces no tendrá diagonal negativa y desequilibra el funcionamiento real del programa ya que no entraría en el segundo $while\ ()$. Y eso no ocurre la mayoría de las veces.

Por lo anterior, $K = \frac{n}{2}$.

$T_{while1} = (K+1)T_{condicion-while1} + KT_{codigo-while1}$

$T_{condicion-while1} = 3$

$T_{codigo-while1} = 4 + T_{if-w} = 4 + 6 = 10$

$T_{if-w} = 3 + 3 = 6$

$T_{while1} = \left(\frac{n}{2} + 1\right)3 + \frac{n}{2}10 = \frac{9}{2}n + 5n = \frac{19}{2}n$

Como $K = \frac{n}{2}$ entonces, $T_{while1} = T_{while2}$

Regresando a la expresión del $T_{caseReina}$ para sustituir los datos que ya tenemos:

$$T_{case1Reina} = 6 + 2 + \frac{19}{2}n + 2 + \frac{19}{2}n = 19n + 10$$
$$T_{case1Reina} = T_{case2Reina} = 19n + 10$$

Y como el codigo de $case1Reina$ del primer $switch()$ es similar al $case2Reina$ del segundo $switch()$. Solo que se invierten los papeles, es decir, a los que estaremos verificando que no maten es a los $n-1$ elementos ya puestos.

$$T_{codigo} = 4 + 1 + 19n + 10 + 19n + 10 = 38n + 25$$

Como $M = n - 1$, entonces:

$$T_{validarPeorCaso-Reina} = 1 + (n - 1 + 1) \cdot 2 + (n - 1)(38n + 25) + (n - 1) \cdot 2$$
$$T_{validarPeorCaso-Reina} = 1 + 2n + 38n^2 - 38n + 25n - 25 + 2n - 2 = 38n^2 - 9n - 26$$

Para finalmente calcular la complejidad de la función completa, en el peor de los casos:

$$T_{validatePosition} = T_{linea118} + T_{linea120} + T_{linea122} + T_{linea124} + T_{validarPeorCaso-Reina}$$

$$T_{linea118} = 2\,restas + 5\,declaraciones + 3\,inicializaciones = 10$$
$$T_{linea120} = 16\,inicializaciones + 4\,declaraciones = 20$$
$$T_{linea122} = 20\,inicializaciones + 4\,declaraciones = 24$$
$$T_{linea124} = T_{if} = T_{condicion} + T_{codigo} = 5 + 1 = 6$$

$$T_{validatePosition} = 10 + 20 + 24 + 6 + 38n^2 - 9n - 26 = 38n^2 - 9n + 34$$

Para el caso de las reinas, tenemos una complejidad $O(38n^2)$. Pongo el coeficiente porque es un número significativo.

A la función que imprime las soluciones, también le sacare su complejidad:

```
243  void printSolution(int pieceMat[][3], int pieceNum,int* typePiece){
244      printf("Encontre solucion\n");
245      int x=0, y=0, k=0;
246      int mat[pieceNum][pieceNum];
247      for( int i=0; i<pieceNum; i++ ){
248          for( int j=0; j<pieceNum; j++ ){
249              x = pieceMat[k][1]; y = pieceMat[k][2];
250              if(x==i && y ==j){
251                  mat[i][j]=typePiece[k];
252                  (k==pieceNum-1)?k=0:k++;
253              }else
254                  mat[i][j]=0;
255          }
256      }
257      for(int i=0; i<pieceNum; i++){
258          for(int j=0; j<pieceNum; j++)
259              printf("%d ", mat[j][i]);
260          printf("\n");
261      }
262      return;
263  }
```

$T_{printSolution} = T_{printf} + 7 + T_{linea247} + T_{linea257}$
$T_{linea247} = T_{for-externo} = T_{ini} + (M+1)T_{comparacion} + MT_{codigo-forExt} + MT_{actualiz}$

Para este caso $M = n$, entonces:

$T_{linea247} = 2 + (n+1) + nT_{codigo-forExt} + 2n$
$T_{codigo-forExt} = T_{for-interno} = T_{ini} + (M+1)T_{comparacion} + MT_{codigo-forInt} + MT_{actualiz}$
$T_{codigo-forExt} = 2 + (n+1) + nT_{codigo-forInt} + 2n$
$T_{codigo-forInt} = 2 + T_{if} = 2 + T_{condicional} + \max\{T_{lineas251-252}, T_{linea254}\}$
$T_{codigo-forInt} = 2 + 3 + \max\{1 + 1 + \max\{1,2\}, 1\} = 5 + \max\{4,1\} = 9$

Sustituyendo en $T_{codigo-forExt}$:

$T_{codigo-forExt} = 2 + (n+1) + 9n + 2n = 12n + 3$

Luego:

$T_{linea247} = T_{for-externo} = 2 + (n+1) + n(12n+3) + 2n = 12n^2 + 6n + 3$

Finalmente, calcularemos $T_{linea257}$. Donde $M = n$:

$T_{linea257} = T_{for-externo} = T_{ini} + (M+1)T_{comparacion} + MT_{codigo-forExt} + MT_{actualiz}$
$T_{linea257} = T_{for-externo} = 2 + (n+1) + nT_{codigo-forExt} + 2n$
$T_{codigo-forExt} = T_{for-interno} + T_{printf} = 2 + (n+1) + nT_{codigo-forInt} + 2n + T_{printf}$
$T_{codigo-forInt} = T_{printf}$

Sustituyendo:

$T_{codigo-forExt} = T_{for-interno} + T_{printf} = 2 + (n+1) + nT_{printf} + 2n + T_{printf} = (n+1)T_{printf} + 3n + 3$
$T_{linea257} = T_{for-externo} = 2 + (n+1) + n\left((n+1)T_{printf} + 3n + 3\right) + 2n$

$T_{linea257} = T_{for-externo} = (n^2 + n)T_{printf} + 3n^2 + 6n + 3$

$T_{printSolution} = T_{printf} + 7 + 12n^2 + 6n + 3 + (n^2 + n)T_{printf} + 3n^2 + 6n + 3$
$T_{printSolution} = T_{printf} + 7 + 12n^2 + 6n + 3 + nT_{printf} + n^2T_{printf} + 3n^2 + 6n + 3$
$T_{printSolution} = 15n^2 + 12n + 13 + n^2T_{printf} + (n + 1)T_{printf}$

El orden de la complejidad de la función $printSolution$ es de $O(n^2)$.

Programación Dinámica con las Torres

```
51          if(*count == 1 && onlyRook(typePiece, pieceNum)){
52              for(int i=0; i<pieceNum; i++)
53                  permut[i] = pieceMat[i][1];
54              do{
55                  for(int i=0; i<pieceNum; i++)
56                      pieceMat[i][2]=permut[i];
57                  printSolution(pieceMat, pieceNum, typePiece);
58                  (*count)++;
59              }while(next_permutation(permut, permut+pieceNum));
60              if(*count==0)
61                  printf("ATENCION: No encontramos solucion en ese orden\n");
62              else
63                  printf("Encontramos %d solucion(es)\n", (*count)-=1);
64              return;
65          }
```

El movimiento de las torres no es muy complicado. Cada torre se pone en una fila y en una columna distinta.

Las soluciones se pueden crear a partir de la primera ya que el resto serán las permutaciones de las filas de la solución_1.

Como ya conocemos como sacar todas las soluciones, uso la función de $c++$ llamada $next\_permutation$ donde a partir de que guarde la primera solución, principalmente la posición de las y de cada torre se generaran las siguientes. Por lo que se esta ahorrando que todas las piezas visiten todo el tablero.

Permutaciones:

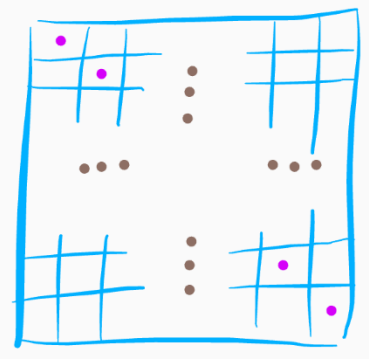
El total de permutaciones que se estarán imprimiendo son:

$$totalPermutacionesTorres = n!$$

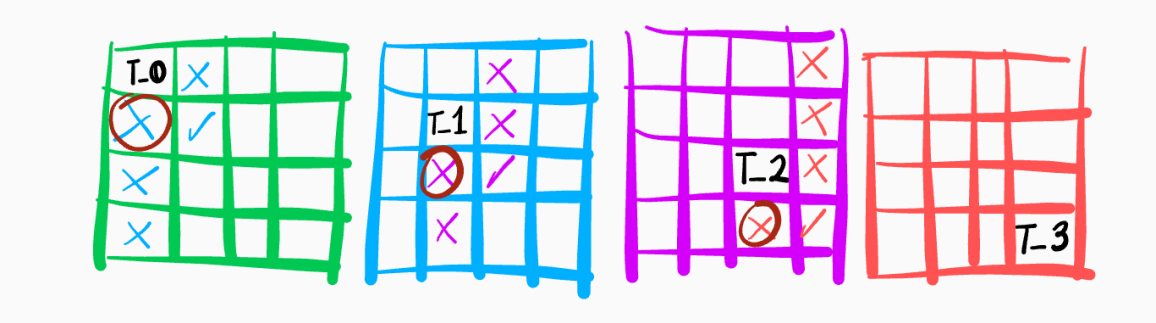
A partir de esto, veremos la complejidad de aplicar esta técnica.

```
Tablero: 3
1.Reina 2.Torre 3.Alfil 4.Peon 5.Caballo 6.Rey
Tipo de piezas: 2 2 2
Encontre solucion
2 0 0
0 2 0
0 0 2
Encontre solucion
2 0 0
0 0 2
0 2 0
Encontre solucion
0 2 0
2 0 0
0 0 2
Encontre solucion
0 0 2
2 0 0
0 2 0
Encontre solucion
0 2 0
0 0 2
2 0 0
Encontre solucion
0 0 2
0 2 0
2 0 0
Encontramos 6 solucion(es)
```

Primeramente, se encuentra la primera solución y veamos que la primera solución es similar no importando la cantidad de torres a acomodar en el tablero. Las torres se acomodarán de forma diagonal en un inicio.



Para formar esta solución recordemos como funciona el recorrido por el tablero.



Todas las piezas en un principio se encuentran sin posición alguna en el tablero. La torre_0 toma la posición $(0,0)$ del tablero. La torre_1 se posiciona bajo la torre_0, es decir, en la casilla $(0,1)$ y la función que valida nos dice que no es correcta su posición por lo que la seguimos recorriendo hasta que cambie de columna. Una vez que cambio de columna se encuentra en $(1,0)$, pero también está en una posición incorrecta y la bajamos una vez más, llegando así a una casilla donde no corre peligro.

Es momento de comenzar a buscarle una casilla correcta a la torre_2. El primer lugar que tomara es la coordenada $(1,2)$, que es la posición que esta enseguida de donde quedo la torre_1. Para encontrar un lugar correcto ocurrirá lo que paso con torre_1. Llegara a la casilla $(2,2)$ que es correcta y comenzara el recorrido de la torre_3 hasta que llegue a la casilla $(3,3)$, recordando que su camino comenzara en $(2,3)$.

Por lo que la cantidad de llamadas que se hacen a la función $movePiece(\,)$ es:

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

Entonces, hasta el momento tenemos que la complejidad de encontrar las soluciones de las torres con esta técnica, es del orden de $O(n^2)$, pero no es la complejidad final.

Después entrara en el codigo que se mostro en un inicio de este apartado. Vemos que se tiene el apoyo de una función llamada $onlyRook$, que se presenta a continuación:

```
86  bool onlyRook(int* typePiece, int pieceNum){
87      for(int i=0; i<pieceNum; i++)
88          if(typePiece[i]!=2)
89              return false;
90      return true;
91  }
```

$T_{onlyRook} = T_{for} + T_{linea90}$
$T_{for} = T_{ini} + (M+1)T_{comparacion} + MT_{codigofor} + MT_{actualiz}$
$T_{codigofor} = T_{if} = T_{comparacion-if} + T_{codigo-if} = 1 + 1 = 2$
$T_{for} = 2 + (n+1) + 2n + 2n = 3 + 5n$
$T_{onlyRook} = 3 + 4n + 1 = 5n + 4$

Ahora, veremos el codigo para formar las soluciones restantes.
Donde la variable $pieceNum$ representa la cantidad de torres que se desean a acomodar en el tablero.

$T_{torres} = T_{for52} + T_{doWhile} + T_{if-else} + 1$

$T_{for52} = T_{ini} + (M+1)T_{comparacion} + MT_{codigo} + MT_{actualiz}$
$T_{doWhile} = U(T_{for55} + T_{linea57} + T_{linea58})$
$T_{if-else} = T_{comparacion} + \max\{T_{linea61}, T_{linea63}\}$

$T_{for52} = 2 + (n+1) + nT_{codigofor52} + 2n = 2 + (n+1) + n + 2n = 4n + 3$
$T_{codigofor52} = 1$

La cantidad de veces que se ejecuta el ciclo $do\ While()$ será $n!$, entonces $U = n!$
Hay que tomar en cuenta que la función $next\_permutation$ tiene una complejidad lineal.

$T_{doWhile} = (n!+1)\left(T_{for55} + T_{linea57} + T_{linea58}\right) + (n!+1)(T_{comparacion})$
$T_{for55} = T_{ini} + (M+1)T_{comparacion} + MT_{codigofor55} + MT_{actualiaz}$
$T_{for55} = 2 + (n+1) + n + 2n = 4n + 3$
$T_{linea57} = T_{printSolution} = n^2$
$T_{linea58} = 2$
$T_{comparacion} = T_{next\_permutation} = n$
$T_{doWhile} = (n!+1)(4n + 3 + n^2 + 2) + n(n!+1)$
$= 4n \cdot n! + 4n + 3n! + 3 + n^2 \cdot n! + n^2 + 2n! + 2 + n \cdot n! + n$
$= n^2 \cdot n! + 5n! + 5n \cdot n! + n^2 + 5n + 5$

Por lo que la complejidad total de encontrar las soluciones de las torres mediante la Programación Dinámica es:

$T_{torres} = T_{primeraSolucion} + T_{permutaciones} = n^2 + n^2 \cdot n! + 5n! + 5n \cdot n! + n^2 + 5n + 5$
$= 2n^2 + n^2 \cdot n! + 5n! + 5n \cdot n! + 5n + 5$

Finalmente, el orden para hallar las soluciones mediante la programación dinámica solo con torres es $O(2n^2 + n^2 \cdot n!)$.

Se anexarán *screenshot* donde se mostrará la mejora en esta implementación con programación dinámica. Se utilizo la plataforma *CodeChef* que muestra el tiempo que tomo la ejecución del codigo y la cantidad de memoria que se usó. Para una salida aceptable en la plataforma *Codechef* se omitió en ambos programas la impresión de cada solución y solamente se imprime la cantidad total de soluciones encontradas.

nombre del programa	nombre de las imágenes	técnica
npieces.cpp	8TorresProgramacionDinamica.jpg 9TorresProgramacionDinamica.jpg 10TorresProgramacionDinamica.jpg 11TorresProgramacionDinamica.jpg (la última cantidad de torres que acepta es 11)	programación dinámica – divide y vencerás
npieces.c	8TorresDivideyVenceras.jpg (la última cantidad de torres que acepta es 8)	divide y vencerás - backtracking

También se anexarán ambos códigos.

MI IDEA DE LA APLICACIÓN DE HEURÍSTICA VORAZ EN LA SOLUCIÓN

Cuando al programa el usuario le indica que quiere saber cuáles son las soluciones de n piezas aleatorias, el programa lo que hace es tomar tal cual el orden como fueron ingresadas.

Por ejemplo, la entrada que se muestra en la siguiente imagen:

```
Tablero: 5
1.Reina 2.Torre 3.Alfil 4.Peon 5.Caballo 6.Rey
Tipo de piezas: 2 3 1 4 6
```

El programa tomara primero la **torre** y la acomodara, después acomodara al **alfil**, seguirá la **reina**, después al **peón** y finalmente acomodara al **rey**, pero la búsqueda no será completa, ya que si hacemos las permutaciones de las piezas se encuentran más soluciones. Es decir, que la búsqueda no se hará una sola vez si no que se llevara a cabo $n!$ veces.

Por ejemplo, en la siguiente ejecución vemos cómo cambia la cantidad de soluciones, siendo las mismas piezas:

```
C:\Users\sel19\Documents\desktop\semestre5\analisisAlgoritmos\practicas\practicas>a
Tablero: 5
1.Reina 2.Torre 3.Alfil 4.Peon 5.Caballo 6.Rey
Tipo de piezas: 2 3 1 4 6
Encontramos 38 solucion(es)

C:\Users\sel19\Documents\desktop\semestre5\analisisAlgoritmos\practicas\practicas>a
Tablero: 5
1.Reina 2.Torre 3.Alfil 4.Peon 5.Caballo 6.Rey
Tipo de piezas: 6 4 1 2 3
Encontramos 42 solucion(es)

C:\Users\sel19\Documents\desktop\semestre5\analisisAlgoritmos\practicas\practicas>
```