Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de casos

Ejercicios 03

Luis Fernando Resendiz Chavez
2019630547

Edgardo Adrian Franco Martinez
Análisis de algoritmos

Algoritmo 01

Funcion complejidad temporal

En este algoritmo los principales elementos que nos deben de importar son cuales y cuantos elementos tiene el arreglo A, ya que las instrucciones que el algoritmo realizará van a depender de la forma en que estén acomodados los elementos de dicho arreglo.
Podemos empezar diciendo que sin importar cual sea el orden de los elementos del arreglo A, se realizarán las siguientes operaciones:

+1 asignacion a m1
+1 comparación entre arreglos
+2 asignaciones a las variables m2 y m3
+1 asignación a i

Operaciones dentro del while:

(n-4)+1 comparaciones en la condición del if
3(n-4) asignaciones en el peor caso, cuando m1 sea menor que A[i]
2(n-4) asignaciones en el caso promedio, cuando m2 sea menor que A[i]
(n-4) asignaciones en el mejor caso, cuando m3 sea menor que A[i]

Para que termine el while, la variable i se va incrementando de uno en uno, por lo tanto hay que contar las siguientes operaciones:

(n-4) asignaciones a i
(n-4) operaciones de suma a i

Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n) = 4(n - 4) + 6$

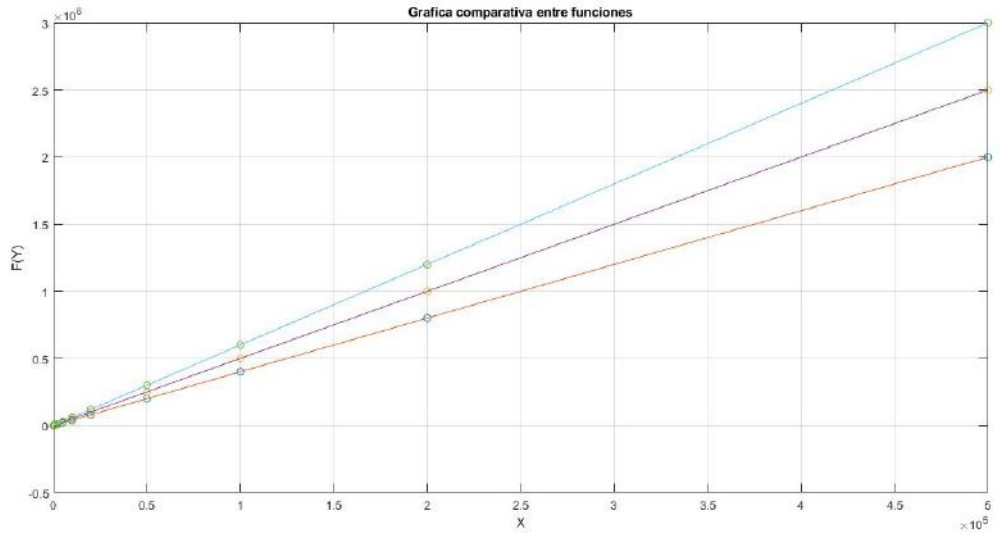
Funcion complejidad temporal para el caso promedio

$f(n) = 5(n - 4) + 6$

Funcion complejidad temporal para el peor caso

$f(n) = 6(n - 4) + 6$

Gráfica comparativa



Código

```
int sumaCuadratica3Mayores(int A[], int n) {
    if(A[1] > A[2] && A[1] > A[3]) { // 3, 2, 1
        m1 = A[1];
        if(A[2] > A[3]) {
            m2 = A[2];
            m3 = A[3];
        } else {
            m2 = A[3];
            m3 = A[2];
        }
    } else if(A[2] > A[1] && A[2] > A[3]) { // 1 3 2
        m1 = A[2];
        if(A[1] > A[3]) {
            m2 = A[1];
            m3 = A[3];
        } else {
            m2 = A[3];
            m3 = A[1];
        }
    } else { // 1, 2, 3
        m1 = A[3];
        if(A[1] > A[2]) {
            m2 = A[1];
            m3 = A[2];
        } else {
            m2 = A[2];
```

```
            m3 = A[1];
        }
    }

    i = 4;

    while(i <= n) {
        if(A[i] > m1) {
            m3 = m2;
            m2 = m1;
            m1 = A[i];
        } else if(A[i] > m2) {
            m3 = m2;
            m2 = A[i];
        } else if(A[i] > m3) {
            m3 = A[i];
        }

        i = i + 1;
    }

    return pow(m1 + m2 + m3, 2);
}
```

Algoritmo 02

Funcion complejidad temporal

En este algoritmo el tamaño del problema es la longitud de n, sin embargo lo que hará que nuestra función temporal cambie, será la distribución de los números del arreglo, ya que la condición if que tenemos en el código sucederá más veces si los números están más desordenados.
Independientemente del tamaño de A (n)
(n-1) iteraciones en el primer for y por cada una se hará lo siguiente:
(n-1) iteraciones el el segundo for y por cada una se hará lo siguiente:
- (n-1) comparaciones en el if
- 3xy asignaciones, siendo "x" el numero de numeros desordenados, "y" la distancia entre la posición inicial y la posición ordenada.

En el mejor de los casos el arreglo dado ya está ordenado de mayor a menor.
En el peor de los casos el arreglo está ordenado de menor a mayor.
El caso promedio dependerá de la distribución de los elementos en el arreglo.

Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n) = 2(n-1)^2$

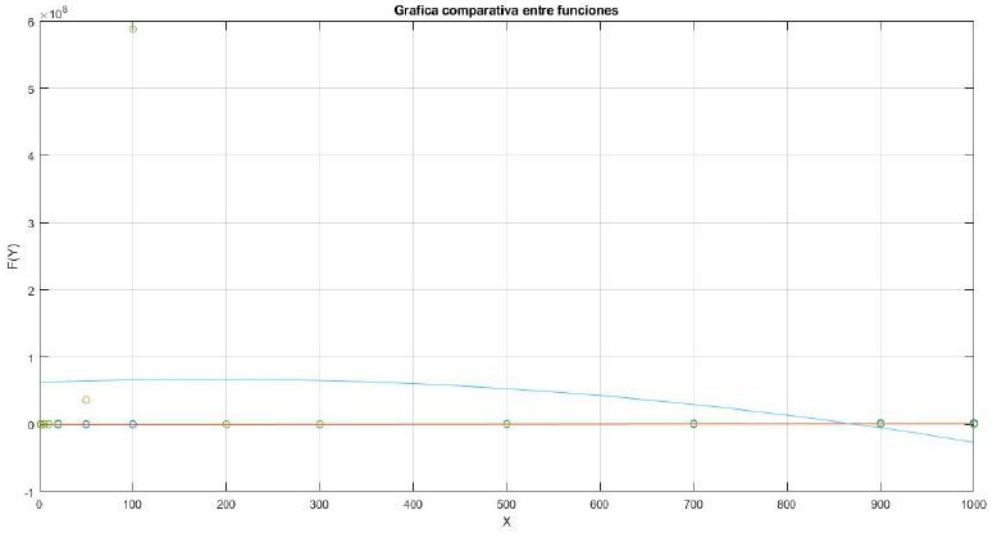
Funcion complejidad temporal para el caso promedio

$f(n) = 2(n-1)^2 \cdot 3xy$

Funcion complejidad temporal para el peor caso

$f(n) = 2(n-1)^2 \cdot 3xy$

Gráfica comparativa



La gráfica tiende a 0, porque los valores son muy grandes y se desborda con n muy grandes.

Código

```
void OrdenamientoIntercambio(A, n) {
    for(i = 0; i < n-1; ++i) {
        for(j = i+1; j < n; ++j) {
            if(A[j] < A[i]) {
                temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
        }
```

```
        }
}
```

Algoritmo 03

Funcion complejidad temporal

Sin importar los valores de m y n:
+3 asignaciones a, b y residuo.
Dentro del while se realizarán:
- r comparaciones
- 3r operaciones de asignación
- r operaciones de módulo

Siendo r el número de residuos entre a y b que no son 0, más una en donde el residuo si es 0 y el while termina.

En el mejor de los casos el número mayor es divisible por el número menor.

El peor de los casos es cuando ambos números pertenecen a la serie de fibonacci,
y son consecutivos, se harán m operaciones de módulo, siendo m el número de dígitos del menor número entre ellos.

El caso medio dependerá de los números dados, si los números no son divisibles entre ellos.

Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n) = 3r + 3$

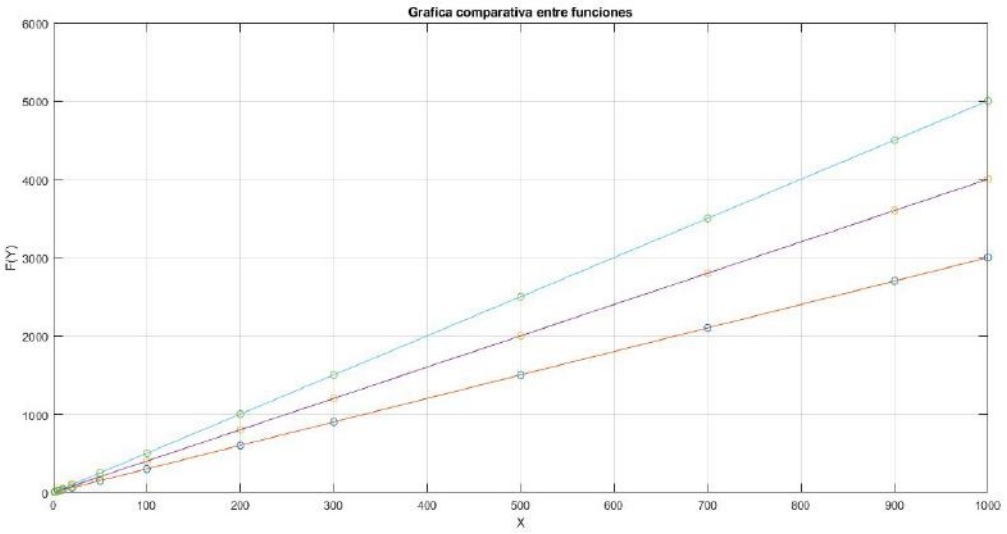
Funcion complejidad temporal para el caso promedio

$f(n) = 4r + 3$

Funcion complejidad temporal para el peor caso

$f(n) = 5r + 3$

Gráfica comparativa



Código

```
int MaximoComunDivisor(m, n) {
    int a = max(n, m);
    int b = min(n, m);
    int residuo = 1;

    while(residuo > 0) {
        residuo = a%b;
        a = b
        b = residuo;
    }
    int MaximoComunDivisor = a;
    return MaximoComunDivisor;
}
```

Algoritmo 04

Funcion complejidad temporal

En este algoritmo el tamaño del problema está definido por la longitud del arreglo dado, el if que está dentro de ambos for se ejecutara en el peor de los casos n^2 veces, esto dependerá de cómo estén distribuidos los números en dicho arreglo.

Sin embargo, sin importar de qué tamaño sea, ni cómo están distribuidos los elementos del arreglo, el primer for se ejecutara 3 veces, y las operaciones de asignación dentro del if, dependerá de la distribución de los números.
Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n) = 2n^2 - 3n + 1$

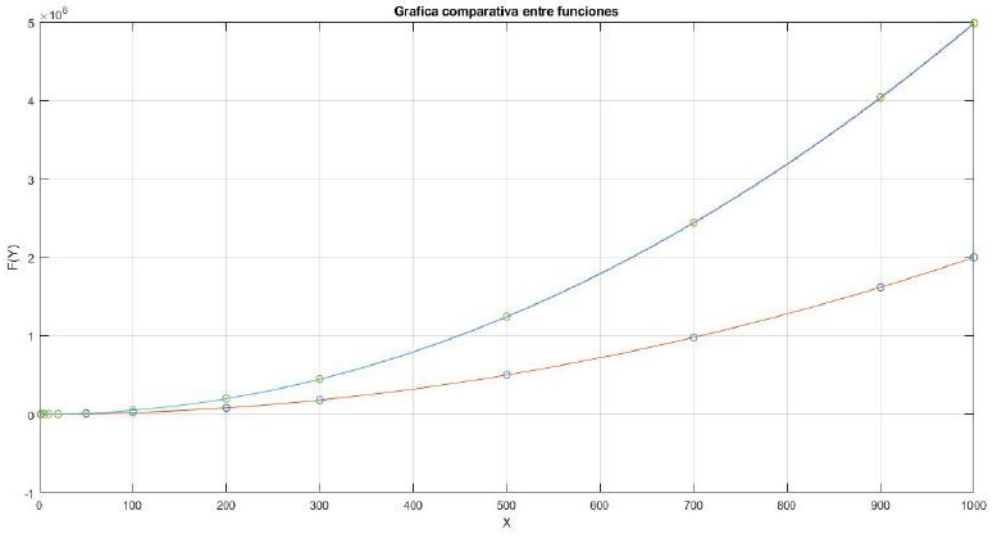
Funcion complejidad temporal para el caso promedio

$f(n) = \frac{5}{4}(5n^2 - 20n + 24)$

Funcion complejidad temporal para el peor caso

$f(n) = 5n^2 - 9n - 5$

Gráfica comparativa



Código

```
int SumaCuadratica3MayoresV2(int A[], int n) {
    for(int i = 0; i < 3; ++i) {
        for(int j = 0; j < n-1-i; ++j) {
            if(A[j] > A[j+1]) {
                int aux = A[j];
                A[j] = A[j+1];
                A[j+1] = aux;
            }
```

```
            }
        }
}
```

Algoritmo 05

Funcion complejidad temporal

Este algoritmo ya es conocido y ya lo hemos analizado anteriormente en equipo, podemos comenzar con que sin importar el tamano del arreglo, la función realizara la creacion y asignacion de las variables auxiliares y por lo menos comprobará una vez si entra al for, también podemos observar que gracias a la variable que nos dice si hubo cambios o no en cierta iteración, la complejidad temporal de esta función dependerá de la disposición de los números en el arreglo, ya que en el mejor de los casos, el arreglo ya está ordenado y solo tendrá que reiterar una vez en el primer arreglo, hacer las comparaciones en el if y nunca entraría a realizar las demás operaciones, por lo tanto saldría a la siguiente iteración del primer for. En el peor de los casos, el arreglo está ordenado pero de mayor a menor, lo tanto en todas las iteraciones si entraría en el if y por lo tanto, las operaciones que harias eran mínimo n^2.
Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n) = 3n - 3$

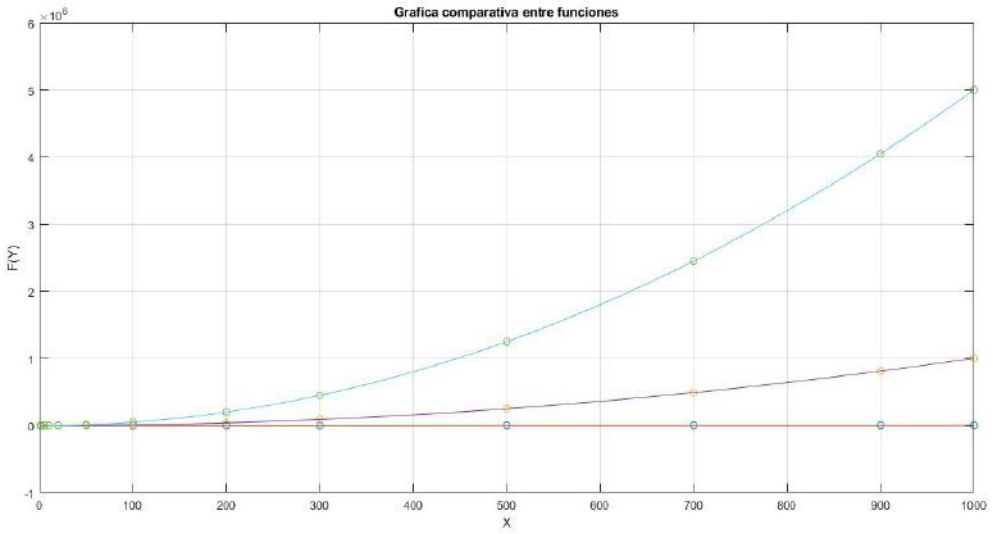
Funcion complejidad temporal para el caso promedio

$f(n) = n^2 + n$

Funcion complejidad temporal para el peor caso

$f(n) = 5n^2 + n - 45$

Gráfica comparativa



Código

```
void BurbujaOptimizada(int A[], int n) {
    char cambios = 's';
    int i = 0;
    while(i < n-1 && cambios != 'n') {
        cambios = 'n';
        for(int j = 0; j <= (n-2)-i; ++j) {
            if(A[j] < A[j+1]) {
                int aux = A[j];
                A[j] = A[j+1];
                A[j+1] = aux;
                cambios = 's';
            }
        }
        i = i + 1;
    }
}
```

Algoritmo 06

Funcion complejidad temporal

En este algoritmo, que incluso ya hemos analizado anteriormente, su complejidad temporal sera la misma sin importar el numero de elementos en el arreglo, n^2, por lo tanto la

complejidad temporal en su mejor, peor y promedio caso dependera 100% de la distribucion de los numeros dentro del arreglo, ya que si todos estan en desorden o estar ordenados pero de forma inversa, siempre entrara en la condicion del if, por lo tanto hara todas las operaciones posibles en el arreglo.
Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n) = 2n^2 - 3n - 2$

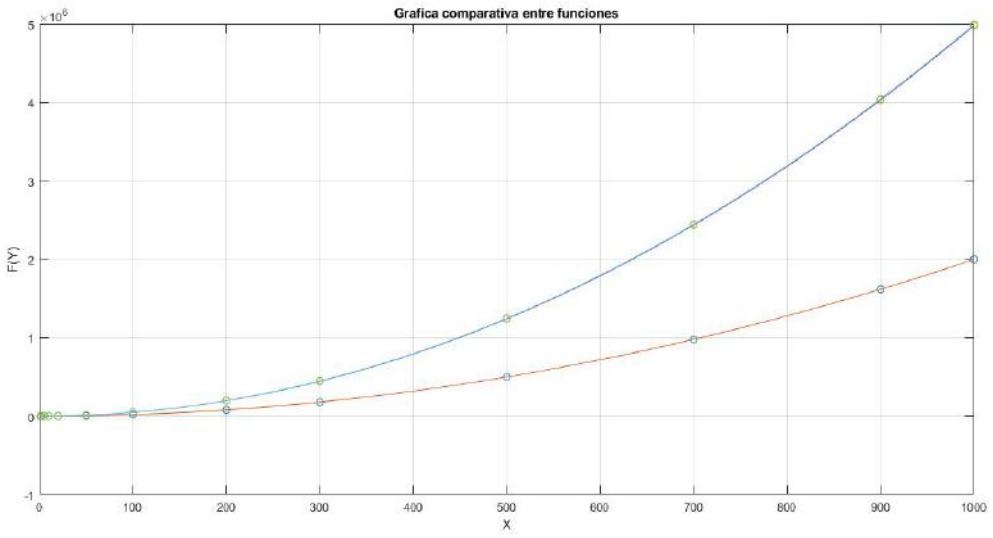
Funcion complejidad temporal para el caso promedio

$f(n) = \frac{9}{8}(5n^2 - 20n + 24)$

Funcion complejidad temporal para el peor caso

$f(n) = 5n^2 - 9n - 8$

Gráfica comparativa



Código

```
void BurbujaSimple(int A[], int n) {
    for(int i = 0; i <= n-2; ++i) {
        for(int j = 0; j <= (n-2)-i: ++j) {
            if(A[j] < A[j+1]) {
                int aux = A[j];
                A[j] = A[j+1];
                A[j+1] = aux;
```

```
                    }
                }
            }
        }
```

Algoritmo 07

Funcion complejidad temporal

Este algoritmo evalúa si un número es primo buscando todos sus divisores desde 1 hasta el mismo, si únicamente hay 2 divisores entonces si lo es, caso contrario no lo es, entonces el tamaño del problema está dado por el número a evaluar, ya que entre más grande sea, más operaciones realizará nuestro algoritmo, sin embargo, es necesario saber que también intervienen otros parámetros, en este caso es el if que está dentro del ciclo for, ya que en el caso de que se cumpla su condición, nuestro algoritmo hará más operaciones.
Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n) = n$

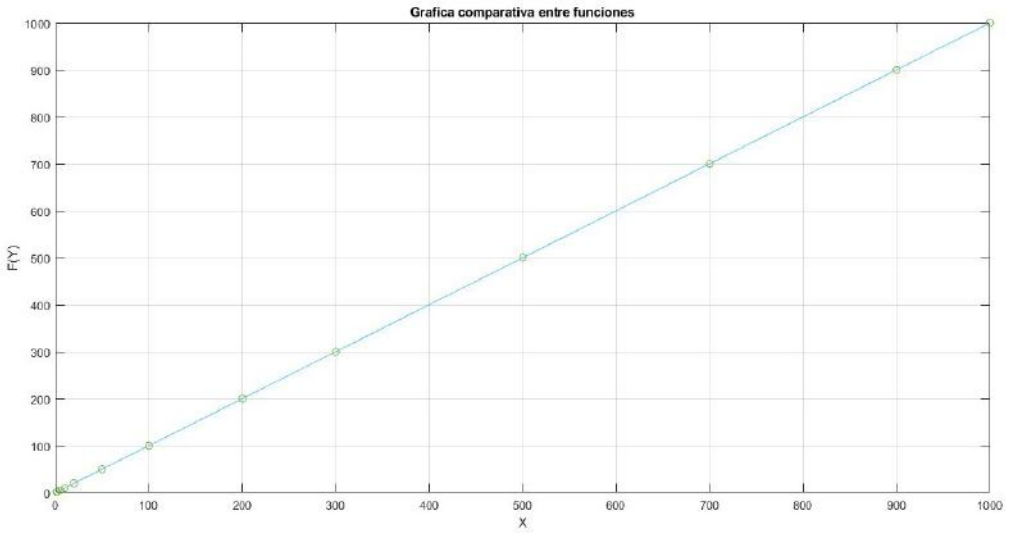
Funcion complejidad temporal para el caso promedio

$f(n) = n$

Funcion complejidad temporal para el peor caso

$f(n) = n$

Gráfica comparativa



Código

```
void ValidaPrimo() {
    int n;
    scanf("%d", &n);
    int divisores = 0;
    if(n > 0) {
        for(i = 1; i <= n; ++i) {
            if(n % i == 0) {
                divisores = divisores + 1;
            }
        }
    }

    (divisores == 2) ? printf("s") : printf("n");
}
```

Algoritmo 08

Funcion complejidad temporal

Este algoritmo busca las frecuencias de los números dados en un arreglo, el tamaño del problema depende de cuántos números hay en el arreglo, también depende de cuántos números hay iguales dentro del mismo, ya que en el caso de que haya más de uno igual, el arreglo realiza más operaciones.

Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n) = n^2 + 5n - 3$

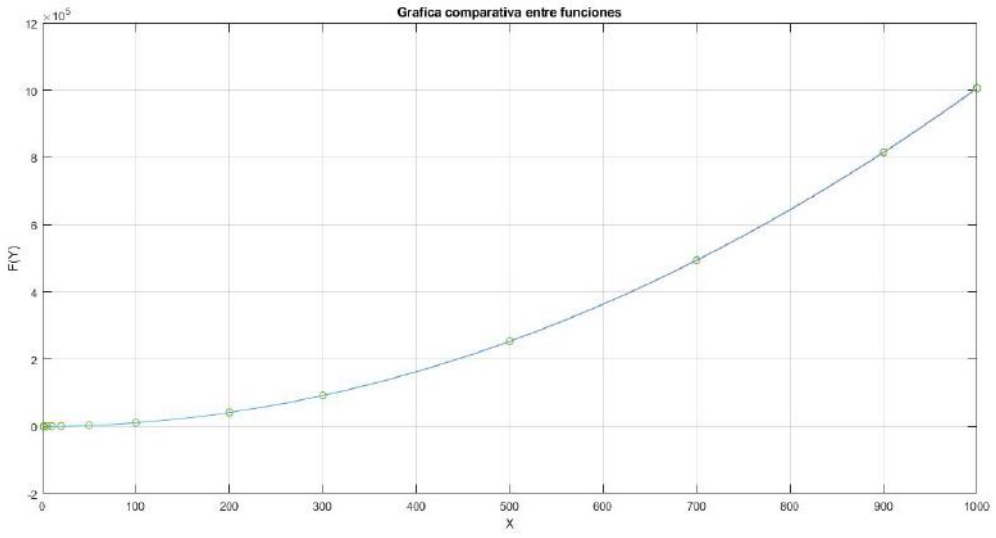
Funcion complejidad temporal para el caso promedio

$f(n) = n^2 + 6n + 4$

Funcion complejidad temporal para el peor caso

$f(n) = n^2 + 7n + 4$

Gráfica comparativa



Código

```
void FrecuenciaMinNumeros() {
    int n;
    scanf("%d", &n);
    int A[n];
    int i = 1;
    while(i <= n) {
        scanf("%d", &A[i]);
        i = i + 1;
    }

    int f = 0;
```

```
    i = 1;

    while(i <= n) {
        int ntemp = A[i];
        int j = 1;
        int ftemp = 0;
        while(i <= n) {
            if(ntemp == A[j]) {
                ftemp = ftemp + 1;
            }
            j = j + 1;
        }

        if(f < ftemp) {
            f = ftemp;
            num = ntemp;
        }

        i = i + 1;
    }
}
```

Algoritmo 09

Funcion complejidad temporal

Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n,m) = n - m + 1$

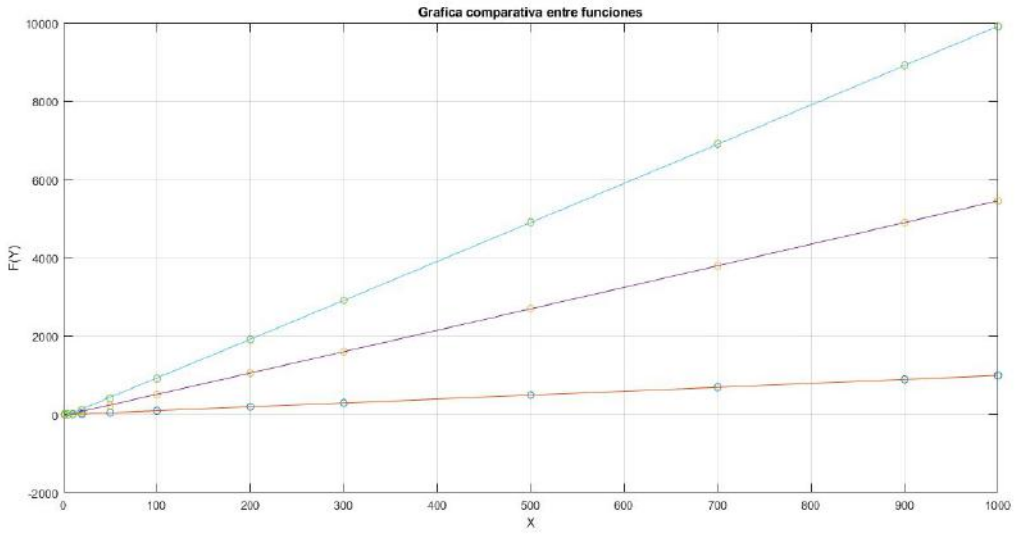
Funcion complejidad temporal para el caso promedio

$f(n,m) = \frac{n+nm-m^2+1}{2}$

Funcion complejidad temporal para el peor caso

$f(n,m) = m(n - m + 1)$

Gráfica comparativa



Código

```
void search(char* pat, char* txt) {
    int M = strlen(pat);
    int N = strlen(txt);

    for(int i = 0; i <=  N - M; i++) {
        int j;
        for(j = 0; j < M; ==j) {
            if(txt[i = j] != pat[j]) {
                break;
            }
        }
        if(j == M) {
            printf("Pattern found at index %d \n", i);
        }
    }
}
```

Algoritmo 10

Funcion complejidad temporal

Tomando en cuenta que las operaciones básicas de un pila que se utilizan en este algoritmo son pop(), push(), entonces analizando el algoritmo, en el mejor de los casos es cuando la pila ya está ordenada de manera no decreciente, por lo tanto las únicas operaciones que se hacen son a la pila auxiliar. También, en el peor de los casos es justamente cuando los elementos de la pila están ordenados de manera creciente, por lo tanto para cada x en la pila se tiene que vaciar hacia la pila temporal, y regresar los datos, antes de meter a x a temp. Entonces podemos notar que la complejidad aumenta y bastante.
Por otro lado, el caso medio se obtiene generalizando la idea del algoritmo y calculando sus probabilidades de las instancias de la pila dada.
Por lo tanto, las funciones de complejidad temporal para el mejor, peor y promedio las podemos definir como sigue:

Funcion complejidad temporal para el mejor caso

$f(n) = 2n$

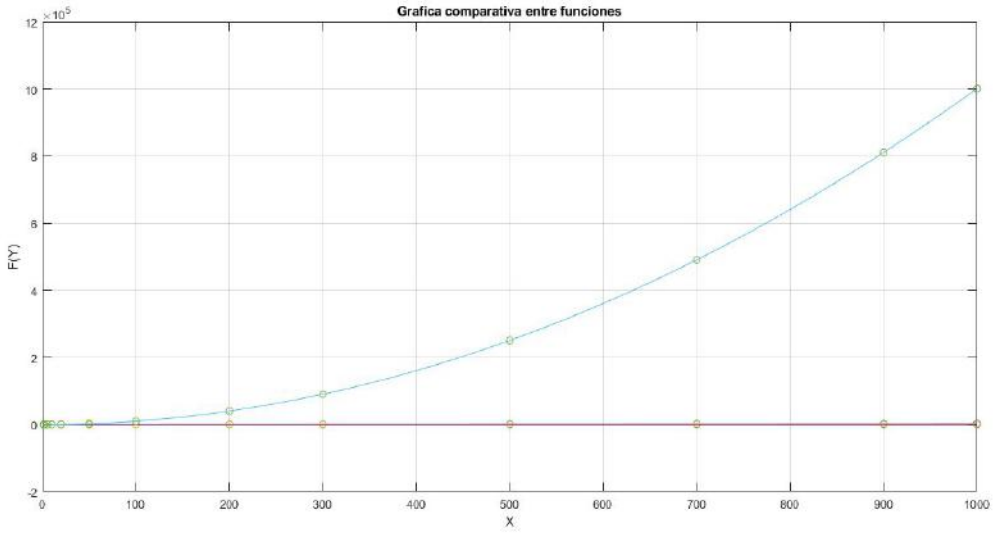
Funcion complejidad temporal para el caso promedio

$f(n) = \frac{1}{2}(n^2 + 3n)$

Funcion complejidad temporal para el peor caso

$f(n) = n^2 + n$

Gráfica comparativa



Código

```
stack<int> sortStack(stack<int> &input) {
    stack<int> tmpStack;
    while(!input.empty()) {
        int tmp = input.empty();
        input.pop();

        while(!tmpStack.empty() && tmpStack.top() > tmp) {
            input.push(tmpStack.top());
            tmpStack.pop();
        }

        tmpStack.push(tmp);
    }

    return tmpStack;
}
```