.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica para el Reino de la Tierra

13 de Octubre de 2025

Agustin Bermudez	Mateo Gonzalez Pautaso	Tiago Calderón
111863	111699	111894

1. Análisis del problema

En el comienzo del desarrollo del problema, consideramos la posibilidad de plantear la solución al problema del enunciado, inspirándonos en algoritmos ya conocidos y vistos en clase. El primero de ellos fue el problema de la 'mochila', donde modelaríamos un arreglo bidimensional con (x_i, f_i), pero, por la naturaleza del mismo, una implementación así cuenta con limitaciones a la hora de saber cuánto poder acumulado uno tiene en un minuto i.

Otra idea propuesta fue plantearlo como el problema de 'Tú a Londres y yo a California', donde tendríamos dos arreglos que en cada índice i representarían una decisión tomada en ese minuto. Lo habríamos modelado de forma que uno de los arreglos siempre se forzara a atacar en el último minuto y el otro a no hacerlo. Si bien esto no tiene mucho sentido, ya que en un arreglo nunca atacarías y en el otro siempre atacarías, nos sirvió como un primer acercamiento a la idea de forzar las decisiones para alcanzar un óptimo.

1.1. Solución propuesta

Finalmente, utilizamos ideas provenientes del 'Problema del cambio' y de 'Osvaldo, el especulador'. Este último resultó especialmente útil ya que nos permitió darnos cuenta de que convenía usar un array en el que cada posición representara un minuto, calculando en cada uno cuál es la máxima cantidad de bajas enemigas que se pueden alcanzar forzándose a atacar en este minuto y analizando los óptimos anteriores.

De esta manera, se puede plantear la ecuación de recurrencia obtenida, considerando un arreglo que almacena los valores óptimos, es decir, la máxima cantidad de enemigos abatidos en cada minuto. Los casos base de la ecuación son:

- $i = 0$: Representa el minuto 0, por lo tanto el óptimo es 0 enemigos atacados por el hecho de que aún no hubo ningún ataque enemigo ni poder acumulado.
- $i = 1$: En el minuto 1, la mejor decisión que se puede tomar va a ser atacar a los enemigos que llegaron.

Sabiendo esto y tomando ideas del 'Problema del cambio', llegamos al razonamiento siguiente: En cada minuto i me fuerzo a atacar y, por esto, debo buscar en que minuto me conviene hacer el último ataque para maximizar la cantidad de enemigos abatidos hasta este minuto. Plasmando esto en una ecuación de recurrencia, nos queda lo siguiente:

$$OPT[i] = max(OPT[t] + min(f_{i-t}, x_i))\forall t < i \tag{1}$$

Detalles de la ecuación: Para calcular el óptimo del minuto i voy a recorrer todos los minutos anteriores al actual y verificar con cuál, también habiendo forzado a atacar en ese otro minuto, se maximiza la cantidad de bajas enemigas. Esto también considerando que el haber atacado en un minuto previo me reducirá la acumulación de poder.

Sea x_i la cantidad de enemigos que llegan en el minuto i y f_i la cantidad de poder acumulado hasta dicho minuto, definimos $OPT[i]$ como la máxima cantidad de enemigos abatidos al finalizar este minuto, forzando un ataque en el mismo. El parámetro t representa el minuto del último ataque más reciente, y $OPT[t]$ guarda cuánto es el máximo de derribados hasta ese t. Por eso, la cantidad de enemigos que se pueden derribar es el óptimo en el minuto t agregándole los derribos por el ataque en el minuto actual que se representa con el mínimo entre el poder acumulado desde el minuto t y x_i (se usa el mínimo porque no puedo abatir más enemigos de los que llegan ni tampoco más de los que me permite mi poder).

2. Detalles del algoritmo

2.1. Implementación

A continuación se encuentra la implementación, en Python, de la solución propuesta en los apartados anteriores. Esta implementación consiste en un algoritmo de Programación Dinámica que busca maximizar la cantidad de enemigos abatidos según las oleadas provenientes de los mismos y la función acumuladora de poder.

Nuestro algoritmo presenta las características típicas de un enfoque de Programación Dinámica: evalúa de forma iterativa los posibles estados y aplica una lógica de decisión que busca maximizar el valor óptimo acumulado basándose en los óptimos de subproblemas anteriores (un subproblema en este algoritmo se representa con un n más pequeño, es decir, una menor cantidad de minutos)

En cada iteración, el algoritmo calcula el mejor resultado posible para el minuto actual i, considerando todos los minutos previos $t < i$ y determinando cuál maximiza la cantidad total de enemigos abatidos hasta ese punto, de acuerdo con la función acumuladora de poder.

El siguiente fragmento de código, en Python, muestra la función desarrollada, que implementa estos pasos, aplicando la ecuación de recurrencia, y devuelve tanto el valor óptimo final como las decisiones que conducen a él, obtenidas mediante la reconstrucción de la solución.

```python
def maximizar_kills(x, f):
    n = len(x)
    if n == 0: return 0

    KILLS = [0] * (n + 1)

    for i in range(1, n + 1):
        max_kills = -1

        for t in range(0, i):
            ataque = KILLS[t] + min(x[i - 1], f[(i - t) - 1])

            if ataque > max_kills:
                max_kills = ataque

        KILLS[i] = max_kills

    decisiones = reconstruir_solucion(x, f, KILLS)
    return KILLS[n], decisiones
```

2.2. Reconstrucción

Una vez que se tienen los óptimos para cada minuto, se le pasan todos los arreglos (el de los enemigos que arriban en cada minuto, el de la función acumuladora de poder y el que tiene los óptimos para cada minuto) a la función *reconstruir_solucion* que se encargará de explicitar los días en los que se decidió atacar y en los que se decidió esperar para acumular poder.

En el siguiente fragmento de código se puede observar dicha función:

```
def reconstruir_solucion(x, f, KILLS):
    n = len(KILLS)
    if n == 0: return []

    decisiones = []

    i = n - 1
    anterior = i    # En la iteracion "anterior" siempre se ataca

    while i > 0:

        if anterior == i:

            for t in range(i - 1, -1, -1):
                if KILLS[i] == KILLS[t] + min(x[i - 1], f[(i - t) - 1]):
                    decisiones.append("Atacar")
                    anterior = t
                    break

        else:
            decisiones.append("Cargar")

        i -= 1

    decisiones.reverse()
    return decisiones
```

La lógica de la reconstrucción parte de que se sabe que en el último día siempre se va a atacar debido a que al ser el último, nunca va a convenir 'Cargar' para intentar asesinar más soldados en el futuro, esto también es válido aunque tenga la menor cantidad de poder acumulada(haber atacado en el anteúltimo minuto). Desde ahí, se recorre el arreglo de óptimos de atrás para adelante utilizando una variable auxiliar llamada *anterior* que nos dirá la próxima batalla (en la iteración) en la que se debe atacar. En caso de que el índice en *anterior* no coincida con el de la iteración, significa que en el minuto actual se decidió Cargar.

Finalmente se utiliza la función *reverse* (complejidad $O(n)$) para que el arreglo de decisiones quede bien ordenado, debido a que el arreglo de óptimos se recorre al revés.

2.3. Complejidad

2.3.1. Construcción de la solución

La complejidad del algoritmo está determinada por los dos bucles for que la componen. El bucle externo recorre el rango $[1, n+1)$ (ya que el for es exclusivo), mientras que el interno recorre $[0, i)$, siendo i el valor actual del índice externo. Debido a esto, el total de iteraciones que habremos recorrido es aproximadamente $n * \frac{(n+1)}{2}$ y esto desembocaría en una complejidad de $O(n^2)$.

Mediciones de complejidad para la construcción

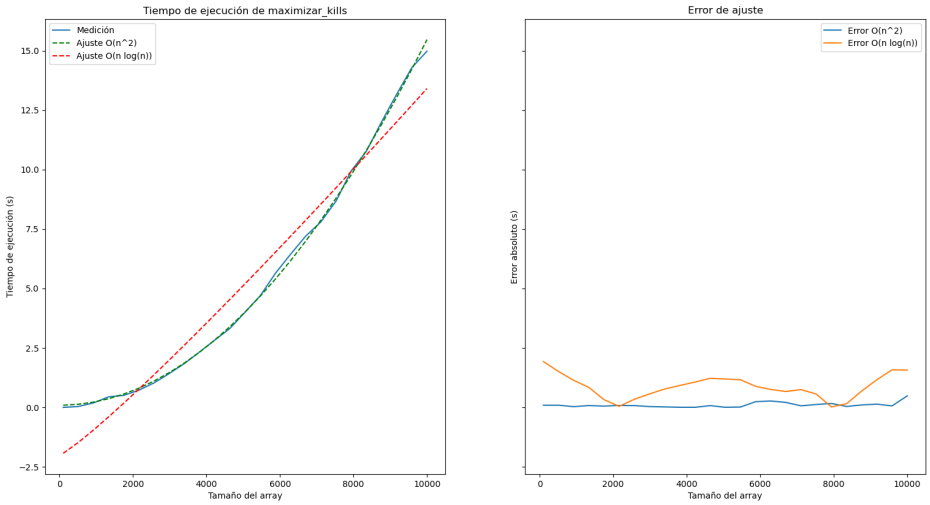

Figura 1: Resultados con $f \in [1, 100000]$ y $x \in [1, 100000]$

2.3.2. Reconstrucción de la solución

El algoritmo de reconstrucción va a recorrer el arreglo de óptimos desde el fin hacia el inicio, buscando la decisión por la construcción tomada en cada iteración. Cuando se llega a un minuto en el que se realizó un ataque, se recorre linealmente, desde el fin hasta el inicio, el arreglo hasta encontrar el minuto anterior en el que también se realizó un ataque. Este recorrido para buscar el próximo valor de la variable *anterior* comienza desde el valor actual de la misma y termina de iterar una vez encontrado el mismo, mediante el uso de *break*. Por lo que en el total de todas las iteraciones del *while* solo se termina recorriendo 1 vez el arreglo en el bucle interno for.

Esto nos deja con una complejidad en el recorrido del arreglo de la construcción de $O(2n)$, es decir, complejidad $O(n)$.

Luego de iterar el arreglo para verificar las decisiones en cada minuto, se realiza un reverse sobre el arreglo reconstruido. Esta operación tiene una complejidad $O(n)$

En conclusión, la complejidad total de la reconstrucción de la solución es $O(n)$.

Mediciones de complejidad para la reconstrucción

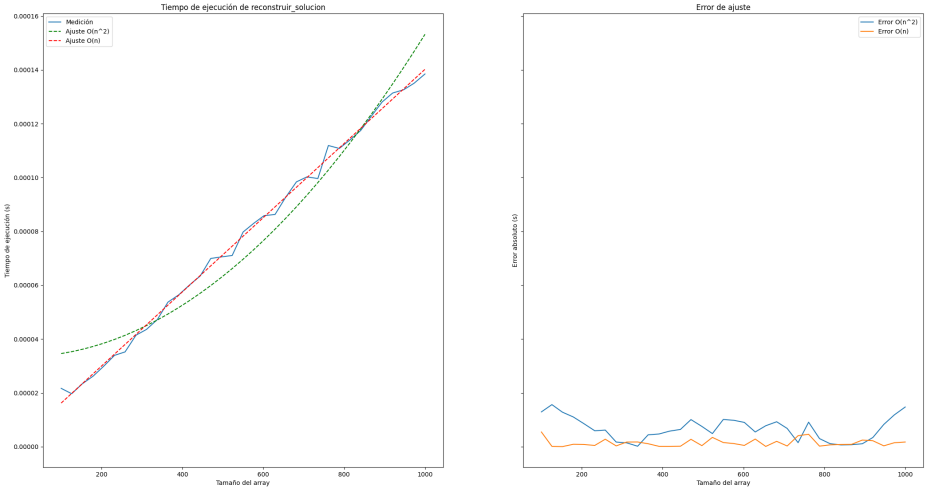

Figura 2: Resultados con $f \in [1, 100000]$ y $x \in [1, 100000]$

2.3.3. Complejidad total

Por ende, la complejidad del algoritmo será $O(n^2)$, dado que dentro de la construcción se cuenta con una complejidad $O(n^2)$, que absorbe la complejidad de la reconstrucción, que es $O(n)$.

Todas las imágenes que respectan a las mediciones de tiempo del algoritmo en conjunto están en la sección 5.

3. Análisis de optimalidad del algoritmo

3.1. Demostración de optimalidad

La demostración de la optimalidad del algoritmo va a ser realizada mediante el método de inducción.

Primero se considera el caso base de la inductividad, $i = 1$, es decir el primer minuto.

Al realizar un ataque en el minuto $i = 1$, sabemos que la única posibilidad para el minuto en que se realizó el último ataque es que no haya habido ninguno, es decir, $t = 0$. Por lo que el número máximo de tropas eliminadas que podemos conseguir se representa como $min(x_1, f(1))$.

Aplicado a la ecuación de recurrencia quedaría de la forma:

$$OPT[1] = max(OPT[0] + min(f(1), x_1)) \tag{2}$$

Que es efectivamente, el óptimo en el primer minuto. Cabe aclarar que el valor $OPT[0]$ es un caso base constante con valor 0, esto por la naturaleza de la ecuación de recurrencia.

Paso inductivo, ataque en el minuto i.

Supongamos que para todo minuto t anterior al actual $t < i$, es cierto que el valor $OPT[t]$ es efectivamente el valor máximo de tropas eliminadas que se pueden alcanzar para ese minuto t en concreto habiendo atacado en t.

La ecuación de recurrencia va a buscar el valor máximo entre el óptimo de cada minuto t agregándole la cantidad de bajas que se pueden realizar en el minuto i con el poder que se viene acumulando desde el minuto t. Es decir, va a tomar la forma:

$$(mejor\ posible\ hasta\ t\ con\ último\ ataque\ en\ t) + min(x_i,\ f(i\ -\ t)) \tag{3}$$

Por la hipótesis inductiva, sabemos que el primer término es $OPT[t]$, desde ahí, se observa que todo valor alcanzable con el último ataque en el minuto t se encuentra acotado

$$OPT[t] <= OPT[t] + min(x_i,\ f(i\ -\ t)) \tag{4}$$

Recorriendo todos los minutos t, sabemos que ninguna estrategia en i puede superar a:

$$max(\ OPT[t] + min(x_i,\ f(i\ -\ t))\ )\ \forall\ t < i \tag{5}$$

Que es exactamente la ecuación presente en la ecuación de recurrencia aplicada en el algoritmo, ya que esa es la definición de $OPT[i]$. Se puede concluir que $OPT[i]$ es el valor máximo de tropas eliminadas que se puede conseguir en el minuto i habiendo atacado en i, seleccionando como ataque anterior al minuto t que maximiza la expresión.

Conclusión de la inducción.

Sabiendo que el paso inductivo funciona desde la hipótesis de que todos los minutos anteriores al actual cuentan con su valor óptimo calculado y que los casos base de $i = 0$ y $i = 1$ son siempre óptimos, el paso inductivo se traslada a $i = 2$, cumpliendo la optimalidad en ese minuto, luego hasta $i = 3$ y siguiendo la cadena hasta llegar al minuto final $i = n$.

Por lo tanto, el algoritmo es óptimo y calcula la máxima cantidad de bajas posibles hasta el minuto n proporcionado.

3.2. Corroboración por backtracking

Utilizamos un algoritmo que cumple lo pedido por el enunciado utilizando Backtracking para comparar nuestros resultados con los del implementado mediante programación dinámica.

El algoritmo de backtracking implementado es el siguiente:

```
def _maximizar_kills_bt(x, f, n, i, ult_atq, mejor_estado):
    if i > n:
        return 0    # kills de aca en adelante

    key = (i, ult_atq)
    if key in mejor_estado:
        return mejor_estado[key]    # Ya esta calculado el maximo para este estado

    # Rama 1: no ataco y sigo cargando
    no_ataco_ahora = _maximizar_kills_bt(x, f, n, i + 1, ult_atq, mejor_estado)

    # Rama 2: ataco en este minuto
    j = i - ult_atq
    kills_i = min(x[i - 1], f[j - 1])
    ataco_ahora = kills_i + _maximizar_kills_bt(x, f, n, i + 1, i, mejor_estado)

    max_kills = max(no_ataco_ahora, ataco_ahora)
    mejor_estado[key] = max_kills
    return max_kills

def maximizar_kills_bt(x, f):
    n = len(x)
    mejor_estado = {}
    return _maximizar_kills_bt(x, f, n, 1, 0, mejor_estado)
```

Dada la naturaleza del problema, al usar el algoritmo por backtracking sin ninguna optimización, resultaba en unos tiempos de ejecución muy grandes. Para evitar esto, se usa un diccionario que va a ir memorizando la mejor cantidad de kills que se pueden obtener dado un momento específico (formado por el minuto actual y el minuto del último ataque). Al tener memorizada la mayor cantidad de kills que se pueden obtener en cierto estado, no hace falta recalcularlo en algunas de las ramas de recursión, lo que mejora sustancialmente los tiempos de ejecución.

Se compararon las salidas de los archivos provistos por la cátedra, una por backtracking y otra por programación dinámica, observando los siguientes resultados:

Archivo	Programación Dinámica	Backtracking
5.txt	1413	1413
10.txt	2118	2118
10_bis.txt	1237	1237
20.txt	11603	11603
50.txt	3994	3994
100.txt	7492	7492
200.txt	4230	4230
500.txt	15842	15842
1000.txt	4508	4508
5000.txt	504220	504220

Cuadro 1: Comparación entre Programación Dinámica y Backtracking

Nota sobre las ejecuciones: El tiempo de ejecución del algoritmo para 1000.*txt* es de aproximadamente 5 minutos y para el archivo 5000.*txt* es de aproximadamente 25 minutos. Ademas para poder ejecutar estos archivos hay que modificar el depth recursion limit, pues superan el que viene por defecto en Python.

4. Optimalidad del algoritmo cuando varían los valores

4.1. Verificación de optimalidad con datasets propios

Nota: Si bien los datasets generados son aleatorios, con la librería Random de Python, se pueden ajustar sus valores mediante el uso de una semilla.

- Resultados con $f \in [1, 100000]$ y $x = 100$
- Resultados con $x \in [1, 100000]$ y $f = 100$
- Resultados con $x, f \in [1, 100000]$
- Resultados con $x, f \in [1, 100000]$, con x ordenado
- Resultados con $x, f \in [1, 100000]$, con x ordenado al revés
- Resultados con $x, f \in [1, 200]$

En todos los apartados listados, los resultados de la implementación del algoritmo por Programación Dinámica y por Backtracking coincidieron. Por lo tanto, podríamos deducir en base a esto que la variabilidad de los valores no afecta la optimalidad del algoritmo.

4.2. Verificación de optimalidad con datasets de la catédra

4.2.1. Comparación con resultados conocidos

Para testear nuestro código, utilizamos los set de datos y las salidas esperadas proporcionadas:

Archivo	Tropas eliminadas
5.txt	1413
10.txt	2118
10_bis.txt	1237
20.txt	11603
50.txt	3994
100.txt	7492
200.txt	4230
500.txt	15842
1000.txt	4508
5000.txt	504220

Cuadro 2: Cantidad de tropas eliminadas en cada dataset de la cátedra.

Para esto implementamos ciertas pruebas que comparaban los resultados. Las mismas están implementadas en funciones de Python del estilo:

```
def test_10():
    path_set = "./datasets/10.txt"
    x, f = parser(path_set)
    resultado, _ = maximizar_kills(x, f)
    assert resultado == 2118, "El resultado del dataset de n = 10 no coincide con la salida de catedra"
    print(f"Cantidad de tropas eliminadas 10.txt = {resultado}")
```

que se pueden encontrar dentro del repositorio. La salida corrobora todos los valores y no falla en ningún *assert*

5. Tiempos del algoritmo con distintos valores

Metodología de las pruebas: Para tener una medición lo más fiel posible, decidimos fijar el valor de cantidad de ejecuciones, por caso, en 10. Con esta cantidad de ejecuciones se realiza un promedio de los tiempos de ejecución. Los generadores de datasets para las pruebas están hechos usando la librería estándar de Python Random.

Para verificar que la complejidad sea efectivamente $O(n^2)$ decidimos agregar al gráfico un ajuste contra la función $O(nlogn)$, para así poder contrastarlas y estudiar su comportamiento.

Se usó una escala lineal con tamaños para el arreglo de batallas entre [100, 10000], con una cantidad de 25 puntos en el gráfico. Se decidieron usar estos parámetros debido a que aumentar las ejecuciones por caso o el intervalo de tamaños posibles para las listas x y f reflejaba un costo muy grande para computar.

5.1. Mediciones con los valores de f fijos

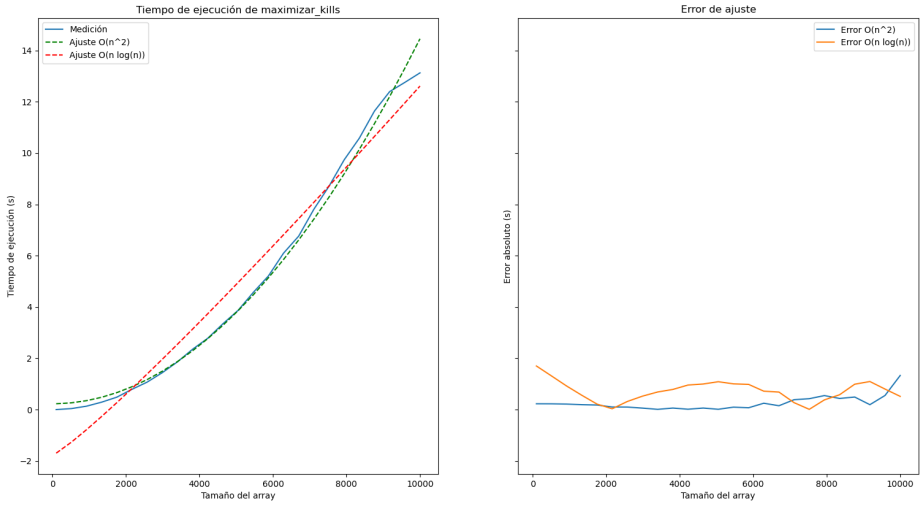

Figura 3: Resultados con $x \in [1, 100000]$ y $f = 100$

5.2. Mediciones con los valores de x ordenado al revés

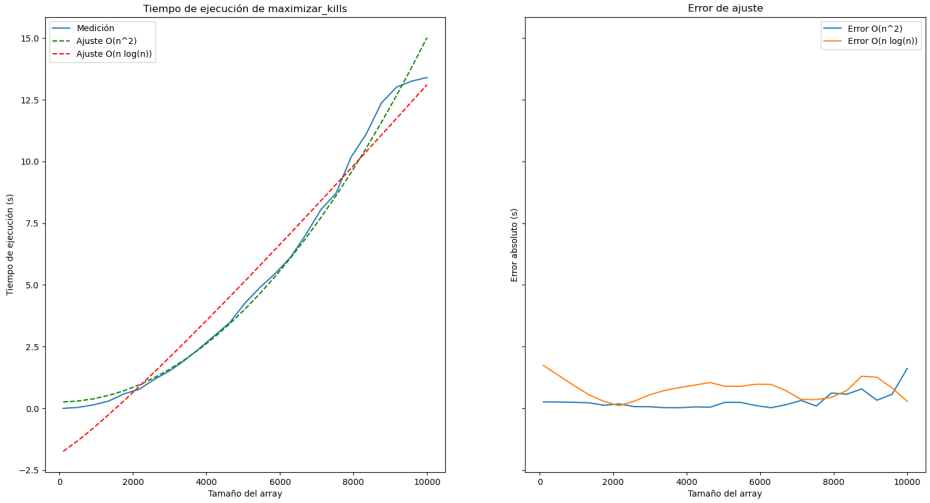

Figura 4: Resultados con $x, f \in [1, 100000]$

5.3. Mediciones con los valores de x ordenados

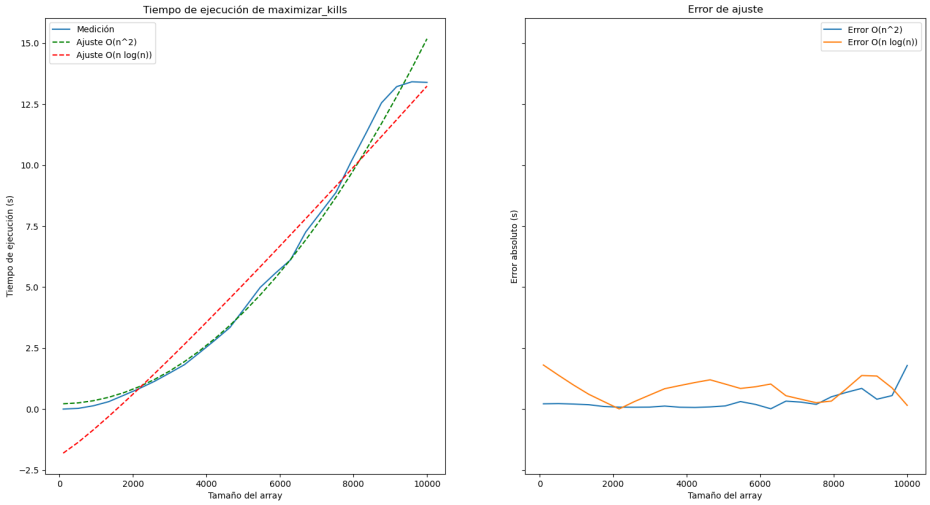

Figura 5: Resultados con $x, f \in [1, 100000]$

5.4. Mediciones con los valores de x aleatorios

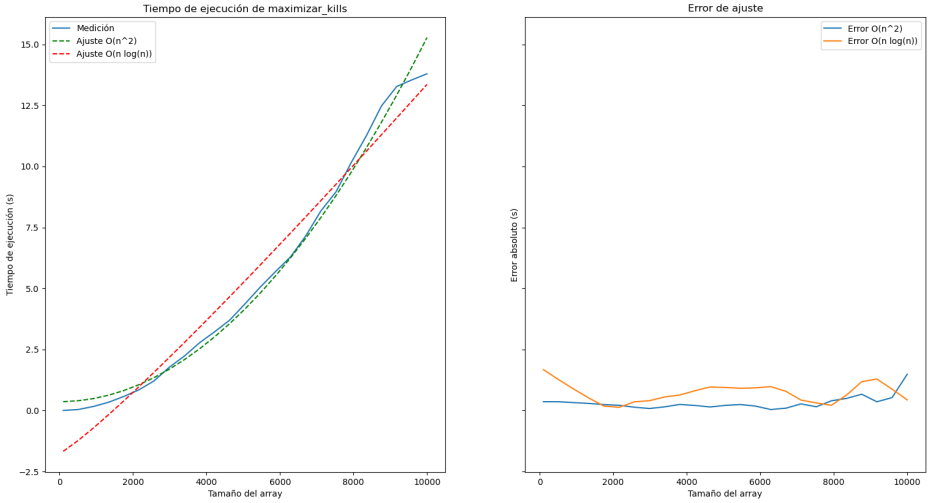

Figura 6: Resultados con $x, f \in [1, 100000]$

5.5. Mediciones con los valores de x fijos

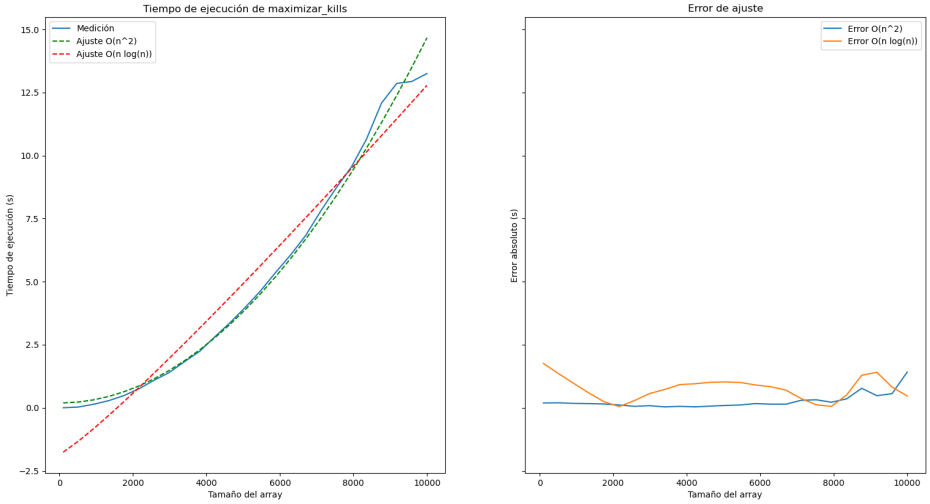

Figura 7: Resultados con $f \in [1, 100000]$ y $x = 100$

5.6. Mediciones con los valores de x y f en un menor rango

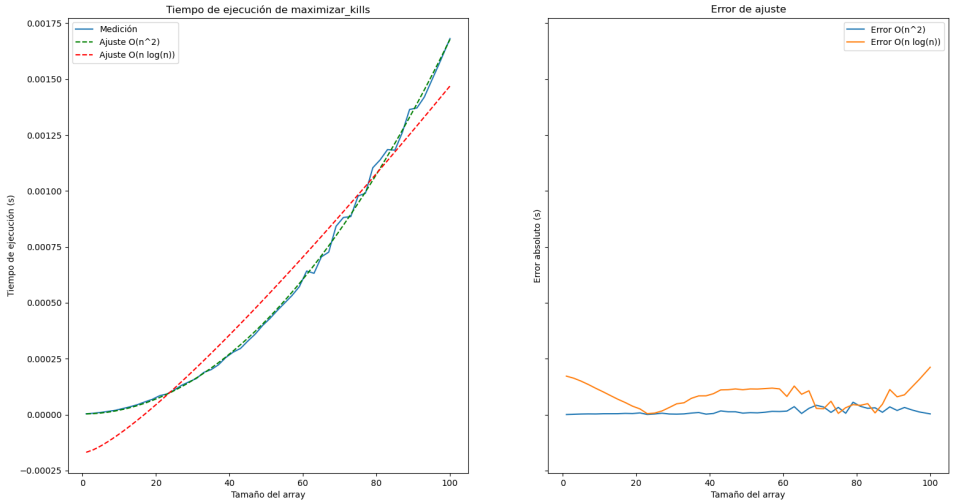

Figura 8: Resultados con $x, f \in [1, 200]$

5.7. Conclusión

Basándonos en las imágenes presentadas, concluimos que la variabilidad de los valores de x y de f no afecta a los tiempos de nuestro algoritmo debido a que en todos los casos, los gráficos de ajuste quedaron muy parecidos. Además los errores por ajuste se minimizan ajustando para $O(n^2)$ y no se observa un aumento claro del mismo en ninguno de los gráficos presentados.

Los resultados experimentales muestran que la variación en los valores de entrada (x y f) no tiene un impacto significativo en los tiempos de ejecución. Esto se debe a que el algoritmo siempre realiza las mismas operaciones para cada i, independientemente de los valores específicos de las listas de entrada, ya que la función $min(x[i-1], f[(i-t)-1])$ tiene costo constante.