REPORTE TAREA 2
ALGORITMOS Y COMPLEJIDAD

«Más allá de la notación asintótica: Análisis experimental de algoritmos de diferentes paradigmas y sus rendimientos.»

Sergio Cárcamo Naranjo

24 de noviembre de 2025

11:13

Resumen

El presente informe detalla la implementación y análisis comparativo de distintas técnicas algorítmicas para resolver un problema de optimización de productividad en la empresa ficticia CppCorp. El problema consiste en particionar una fila de empleados en equipos contiguos para maximizar la productividad total, la cual se define por reglas específicas de lenguajes de programación y contribuciones individuales.

El objetivo principal es comparar el rendimiento y la calidad de las soluciones obtenidas mediante cuatro enfoques diferentes. Se implementó una solución óptima mediante Fuerza Bruta (Backtracking) que explora todo el espacio de soluciones. Paralelamente, se desarrolló la solución óptima eficiente utilizando Programación Dinámica.

Además, se implementaron dos heurísticas Greedy subóptimas , diseñadas para encontrar soluciones rápidas pero no necesariamente óptimas. El análisis se centra en medir experimentalmente el tiempo de ejecución y el uso de memoria de cada algoritmo frente a un conjunto de casos de prueba con distintos tamaños de entrada.

Los resultados obtenidos permiten cuantificar el trade-off entre la optimalidad de una solución y su eficiencia computacional. Se busca demostrar la inviabilidad de la fuerza bruta para entradas grandes, la eficiencia de la programación dinámica para encontrar el óptimo, y analizar qué tan cercanas al óptimo son las soluciones entregadas por las heurísticas greedy.

Índice

1. Introducción

El **Análisis y Diseño de Algoritmos** es un pilar fundamental de las Ciencias de la Computación, enfocado en el desarrollo y estudio de métodos eficientes para resolver problemas computacionales. Una clase importante de estos son los problemas de optimización, donde se busca encontrar la mejor solución entre un vasto conjunto de posibilidades. Frecuentemente, un mismo problema de optimización puede ser abordado mediante distintas estrategias o paradigmas algorítmicos (como fuerza bruta, programación dinámica o heurísticas greedy), cada uno con diferentes perfiles de rendimiento y garantías de optimalidad.

Este informe aborda un problema de optimización específico, presentado en el contexto de la empresa CppCorp. El desafío consiste en maximizar la productividad total de la compañía dividiendo a sus empleados, organizados en una fila, en equipos que corresponden a segmentos contiguos no vacíos. La productividad de cada equipo se define por una regla que depende del lenguaje de programación más frecuente entre sus miembros . Este escenario define un problema de partición óptima, donde encontrar la solución global requiere evaluar de forma inteligente las sub-particiones que la componen.

El objetivo principal de este trabajo es implementar y comparar cuatro enfoques distintos para resolver dicho problema. Acorde a los objetivos de la Tarea 2, se busca **comparar diferentes técnicas de resolución de un mismo problema**, evaluando su tiempo de ejecución y uso de memoria. Las implementaciones desarrolladas abarcan:

- Un algoritmo de Fuerza Bruta (mediante backtracking) que explora el espacio completo de soluciones.
- Una solución óptima eficiente mediante Programación Dinámica.
- Dos heurísticas Greedy subóptimas, diseñadas para encontrar soluciones rápidas pero no necesariamente óptimas.

La pregunta que guía este informe es: **¿Cuál es el costo computacional (en tiempo y memoria) de garantizar la optimalidad (Fuerza Bruta, DP) frente a la eficiencia de una solución aproximada (Greedy) para este problema de partición?** El aporte de este trabajo radica en el análisis experimental de estas implementaciones, midiendo su rendimiento en diversos casos de prueba. Adicionalmente, se analizará la calidad de las soluciones greedy para "determinar qué tan cercanas están al resultado óptimo", ilustrando así el balance fundamental entre eficiencia y optimalidad en el diseño de algoritmos.

2. Diseño y Análisis de Algoritmos

2.1. Fuerza Bruta

2.1.1. Explicación

La estrategia de fuerza bruta para este problema consiste en explorar todas las particiones posibles de la fila de n empleados en equipos contiguos no vacíos. Para n empleados, existen $n-1$ posibles **cortes** (posiciones entre dos empleados). Cada corte puede existir o no, lo que resulta en un total de 2^{n-1} formas distintas de particionar la fila.

El algoritmo implementado utiliza Backtracking, una técnica que construye la solución de forma recursiva. Se define una función que, dado un índice inicial start, prueba todas las combinaciones posibles para el primer equipo. Este equipo puede ser el segmento [start, j], donde j varía desde start hasta $n-1$.

Para cada j posible, el algoritmo:

- Calcula la productividad del segmento [start, j] usando la función `productividad_equipo`.
- Se llama recursivamente para encontrar la productividad máxima del resto de la fila, es decir, los empleados de $j+1$ hasta $n-1$.
- Suma ambos resultados.
- La función retorna el valor máximo encontrado entre todas las elecciones posibles de j.

Relación de recurrencia: Podemos definir la solución óptima mediante una relación de recurrencia. Sea $P(i)$ la máxima productividad total que se puede obtener al formar equipos con los empleados en el segmento $[i, n-1]$ (desde el empleado i hasta el final). El objetivo es calcular $P(0)$.

La recurrencia se define como:

$$P(i) = \max_{i \le j < n} \{\text{productividad\_equipo}(i, j) + P(j+1)\}$$

- **Caso Base**: $P(n) = 0$. Si no quedan empleados ($i = n$), la productividad a obtener es 0.
- **Paso Recursivo**: Para encontrar la productividad máxima $P(i)$, probamos todos los puntos de corte j posibles. Para cada j, formamos un equipo $[i, j]$ y sumamos su productividad (calculada según las reglas del enunciado) al resultado óptimo de particionar el resto de los empleados, $P(j+1)$. Tomamos el máximo de todas estas opciones.

El pseudocódigo para este algoritmo está en el apéndice.

2.1.2. Complejidad Temporal

La complejidad temporal de este enfoque es exponencial. Analizando la recurrencia $P(i)$, vemos que para calcular $P(i)$ se realizan $n-i$ llamadas recursivas (a $P(i+1), P(i+2), \ldots, P(n)$). El número total de llamadas recursivas sigue la recurrencia $T(k) = 1 + \sum_{m=1}^{k} T(k-m)$, donde k es el número de empleados restantes. Esto resulta en $O(2^n)$ llamadas totales.

Dentro de cada llamada `resolver_backtracking(i, n)`, el bucle para j desde i hasta n-1 invoca a `productividad_equipo(i, j)`. Esta función auxiliar itera sobre los $j-i+1$ elementos del segmento para calcular frecuencias y sumas. En el peor de los casos, esto toma $O(n)$ tiempo. Dado que se realizan $O(2^n)$ llamadas y cada una implica un bucle que a su vez llama a una función $O(n)$, la complejidad temporal total es $O(n \cdot 2^n)$. Esto coincide con la expectativa experimental de que el algoritmo se vuelve intratable para valores de n grandes (como $n > 30$).

2.1.3. Complejidad Espacial

La complejidad espacial está determinada por la profundidad máxima de la pila de recursión. La secuencia de llamadas más profunda ocurre cuando se elige consistentemente el segmento más pequeño (de un solo empleado), por ejemplo: $P(0) \to P(1) \to P(2) \to \cdots \to P(n)$. La profundidad máxima de la pila de llamadas es, por lo tanto, n. Cada llamada en la pila almacena variables locales (como i, j, mejor_total). La función `productividad_equipo` utiliza estructuras de datos (mapas) que, en el peor caso (un segmento de tamaño n con n lenguajes distintos), podrían ocupar $O(n)$. Sin embargo, esta memoria se libera al retornar de la función. La memoria que persiste a lo largo de la recursión es la de la pila de llamadas. Por lo tanto, la complejidad espacial es $O(n)$ para almacenar la pila de recursión.

2.2. Greedy

2.2.1. Algoritmo Greedy 1

Para este problema, el enfoque **Greedy** consiste en pararse en el primer empleado disponible (start) y decidir dónde **cortar** para formar el primer equipo. Una vez que se forma el equipo [start, end], el proceso se repite desde end + 1 hasta que todos los empleados estén en un equipo. La diferencia entre las dos heurísticas implementadas radica en el criterio para elegir el punto de corte **end**.

Explicación y pseudocódigo: Esta heurística se basa en la decisión localmente óptima más simple: en cada paso, formar el equipo que, por sí solo, genere la mayor productividad posible. El algoritmo comienza en start = 0. Itera probando todos los end posibles (desde start hasta $n-1$) y calcula la productividad del segmento [start, end]. Elige el mejor_end que maximiza este valor. Luego, añade esa productividad al total y repite el proceso comenzando desde mejor_end + 1.

Complejidad Temporal: El algoritmo tiene un bucle mientras que avanza el puntero start. En el peor de los casos, start avanza solo de uno en uno (si el mejor segmento es siempre de tamaño 1), ejecutándose n veces. Dentro de este bucle, hay un bucle para que itera end desde start hasta $n-1$, $O(n)$ veces. Dentro de ese bucle, se llama a `productividad_equipo(start, end)`, que, como vimos, toma O(end - start + 1) = $O(n)$ en el peor caso. La estructura es un bucle anidado (el mientras que avanza start y el para que avanza end) que en total definen $O(n^2)$ pares de (start, end) a probar. Para cada par, se hace un cálculo de $O(n)$. Por lo tanto, la complejidad temporal total es $O(n^3)$.

Complejidad Espacial: El algoritmo almacena el vector de empleados, que ocupa $O(n)$. La función `productividad_equipo` utiliza mapas y vectores auxiliares que en el peor caso pueden ocupar $O(n)$ si todos los empleados tienen lenguajes distintos. No hay recursión. La complejidad espacial es $O(n)$.

2.3. Algoritmo Greedy 2

Explicación: Esta segunda heurística utiliza un criterio Greedy más sofisticado. En lugar de maximizar la productividad absoluta del segmento (lo que podría favorecer segmentos muy largos), busca maximizar la productividad promedio por empleado (densidad). La lógica es idéntica a Greedy 1, pero la decisión de mejor_end se toma calculando la **densidad** como la prod_actual dividido en número de empleados en el segmento. Se elige el mejor_end que maximiza esta métrica.

Complejidad Temporal: El análisis es idéntico al de Greedy 1. La estructura de bucles anidados y la llamada a `productividad_equipo` son las mismas. El cálculo de la densidad (una división) es una operación $O(1)$ que no cambia el orden de complejidad. La complejidad temporal total entonces es $O(n^3)$.

Complejidad Espacial: El análisis es idéntico al de Greedy 1. Se necesita $O(n)$ para almacenar los empleados y $O(n)$ para las estructuras auxiliares dentro de `productividad_equipo`. La complejidad espacial es $O(n)$.

El pseudocódigo para ambos algoritmos está en el apéndice.

2.4. Programación Dinámica

2.4.1. Explicación

La Programación Dinámica (DP) es una técnica que permite encontrar la solución óptima, al igual que la fuerza bruta, pero de manera mucho más eficiente. Resuelve el problema de la fuerza bruta, que recalcula la solución para los mismos subproblemas (por ejemplo, la partición óptima de los empleados $[i, n-1]$) múltiples veces. La DP descompone el problema en subproblemas más pequeños, almacena sus resultados en una tabla y los reutiliza. Esto se conoce como superposición de subproblemas y subestructura óptima. La solución óptima para los n empleados se puede construir a partir de las soluciones óptimas para $k < n$ empleados.

Relación de recurrencia: Definimos un estado $DP[i]$ como la máxima productividad total que se puede obtener al formar equipos usando exactamente los primeros i empleados (es decir, los empleados en los índices 0 a $i-1$). Nuestro objetivo final es encontrar $DP[n]$. Para calcular $DP[i]$, debemos decidir cuál fue el último equipo. Este último equipo debe ser un segmento $[j, i-1]$, donde j puede ser cualquier valor desde 0 hasta $i-1$.

Si el último equipo es $[j, i-1]$, entonces la productividad total será la suma de:

- La productividad de ese último equipo: `productividad_equipo(j, i-1)`.
- La productividad óptima de todos los empleados antes de ese equipo (de 0 a $j-1$), que ya tenemos calculada y almacenada en $DP[j]$.

Como queremos maximizar la productividad, probamos todas las posiciones j posibles para el inicio del último segmento y nos quedamos con la mejor. Entonces, a relación de recurrencia es:

$$DP[i] = \max_{0 \le j < i} \{DP[j] + \texttt{productividad\_equipo}(j, i-1)\}$$

Estructura de datos y orden de cálculo: Se utiliza un arreglo (vector) unidimensional, dp, de tamaño $n+1$, donde $dp[i]$ almacenará el valor de $DP[i]$. En cuanto a la orden de cálculo, para calcular $dp[i]$, necesitamos conocer todos los valores $dp[j]$ donde $j < i$. Por lo tanto, debemos llenar la tabla dp en orden ascendente (método bottom-up), comenzando por el caso base $dp[0]$ y calculando $dp[1], dp[2], \ldots$, hasta $dp[n]$.

El pseudocódigo para este algoritmo está en el apéndice.

2.4.2. Complejidad Temporal

El algoritmo consta de dos bucles anidados:

- El bucle externo itera i desde 1 hasta n, $O(n)$ veces.

- El bucle interno itera j desde 0 hasta $i-1$, $O(i)$ veces, que en el peor caso es $O(n)$.

Dentro del bucle interno, se llama a la función `productividad_equipo(j, i-1)`. Esta función, como se analizó en las secciones anteriores, itera sobre los $i-j$ elementos del segmento, tomando $O(i-j)$ tiempo, que en el peor caso es $O(n)$.

Dado que tenemos tres niveles de dependencia anidada ($O(n) \times O(n) \times O(n)$), la complejidad temporal total es $O(n^3)$. Esto es una mejora exponencial sobre la fuerza bruta ($O(n \cdot 2^n)$) y coincide con la complejidad de las heurísticas greedy implementadas.

2.4.3. Complejidad Espacial

La estructura de datos principal que se almacena es el vector dp de tamaño $n+1$. Por lo tanto, la complejidad espacial es $O(n)$.

Nota: Aunque cada llamada a `productividad_equipo` usa $O(n)$ de espacio temporal para sus mapas, este espacio se libera al retornar y no se acumula, por lo que el espacio persistente del algoritmo sigue siendo $O(n)$.

3. Implementaciones

Link del repositorio:

https://github.com/INF221-20252/INF221-2025-2-TAREA-2-frostodev

4. Experimentos

El hardware utilizado en este estudio consiste de:

- **Procesador**: Intel(R) Core(TM) i5-9400 CPU @ 2.90GHz
- **RAM**: 16 GB DDR4-2666 MT/s Single Channel
- **GPU**: NVIDIA GeForce GTX 1060 6GB
- **SSD**: 256 GB WDC WDS240G2G0C-00AJM0 NVMe
- **HDD**: 14 TB WDC WD142PURP-85B6HY0 SATA

Respecto al software y toolchain usados, estos consisten de:

- Ubuntu 24.04.3 LTS (WSL2) sobre Microsoft Windows 11 Enterprise IoT LTSC
- GNU gcc 13.3.0 (Ubuntu 13.3.0-6ubuntu2 24.04)
- GNU Make 4.3 x86_64-pc-linux-gnu
- Python 3.13.7

4.1. Dataset (casos de prueba)

Para realizar una comparación experimental robusta, fue necesario generar un conjunto de casos de prueba (*dataset*) que evaluara sistemáticamente el rendimiento de los algoritmos bajo distintas condiciones. Siguiendo las especificaciones del Anexo A del enunciado, se implementó un script en Python, `testcases_generator.py`, para crear automáticamente los archivos de entrada.

El diseño de este dataset se centró en la variación de dos parámetros clave: el **cantidad de empleados** (**n**) y los **rangos de productividad** (**A_i y B_i**), manteniendo la cantidad de lenguajes constante entre cada prueba (**Python, C++, Prolog, Scheme, Java**)

4.1.1. Variación de la Cantidad de Empleados (n)

La cantidad de empleados n es la variable principal para analizar la complejidad temporal y espacial. Se seleccionó un conjunto de 13 valores de n con propósitos específicos:

- $n = 5, 6$: Valores pequeños utilizados para validar la correctitud de todos los algoritmos contra los ejemplos proporcionados en el enunciado.
- $n = 15, 18, 20, 25, 28, 30$: Este rango se seleccionó específicamente para analizar el comportamiento de la solución de Fuerza Bruta. Dado que su complejidad es exponencial ($O(n \cdot 2^n)$), se esperaba que el tiempo de ejecución se volviera intratable (superando varios segundos o minutos) dentro de esta ventana. El objetivo era encontrar experimentalmente el "muro exponencial".

- $n = 50, 100, 500, 1000, 5000$: Valores de n considerablemente más grandes. Son completamente inviables para la Fuerza Bruta, pero son esenciales para medir y comparar el rendimiento polinomial ($O(n^3)$) de la Programación Dinámica y las dos heurísticas Greedy. Estos valores se mantienen dentro de los límites del problema ($n < 10^4$). Cabe mencionar que **no se incluyeron casos con** $n > 5000$ debido a que con el hardware actual, se tomaría demasiado tiempo. Sin embargo, con los valores actuales la tendencia debería notarse.

4.1.2. Variación de Rangos de Productividad

Para probar la robustez de las soluciones, especialmente de las heurísticas Greedy, no bastaba con variar n. Por cada valor de n, se generaron 4 casos de prueba distintos (identificados con $i \in [0,3]$ en el nombre `testcases_{n}_{i}.txt`), cada uno con un "perfil"de productividad diferente:

- **Perfil 0 (Largo Mixto)**: $A_i, B_i \in [-10^9, 10^9]$. Corresponde al rango completo especificado en el enunciado. Es el caso estándar.
- **Perfil 1 (Corto Mixto)**: $A_i, B_i \in [-100, 100]$. Un escenario con valores de productividad mucho menores, pero aún mixtos (positivos y negativos).
- **Perfil 2 (Corto Positivo)**: $A_i, B_i \in [1, 100]$. Un escenario de "solo ganancias", donde toda contribución es positiva. Esto podría alterar las decisiones de los algoritmos Greedy, al no tener que gestionar pérdidas.
- **Perfil 3 (A Garantizado Mejor)**: $A_i \in [1, 500]$ y $B_i \in [-500, 0]$. Este perfil impone la lógica de que usar el lenguaje favorito (A_i) es siempre una ganancia, y no hacerlo (B_i) es siempre una pérdida.

Combinando ambas variaciones, el dataset final consiste en 13 valores de $n \times 4$ perfiles = **52 casos de prueba** en total. Este conjunto permite un análisis detallado tanto de la escalabilidad (vs. n) como de la calidad de la solución (vs. perfil de datos).

4.2. Resultados

Basado en los resultados obtenidos de las mediciones temporales, junto a los resultados de productividad calculados por cada algoritmo, podemos derivar los siguientes gráficos, usando el script **plot_generator.py**:

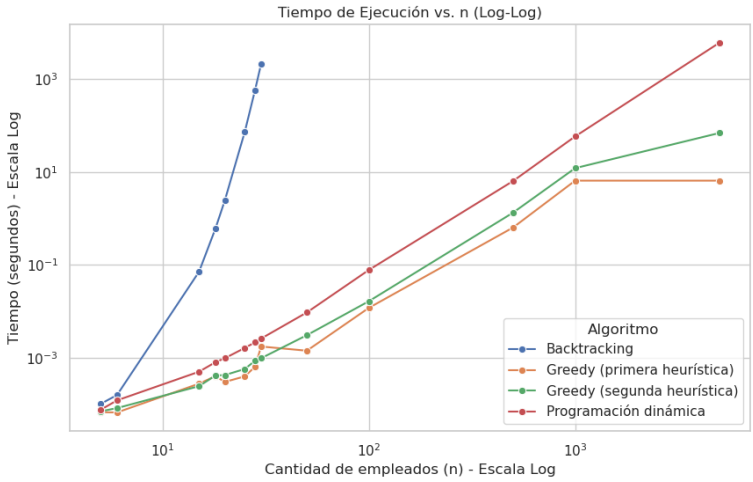

Figura 1: Tiempo de Ejecución vs cantidad de empleados

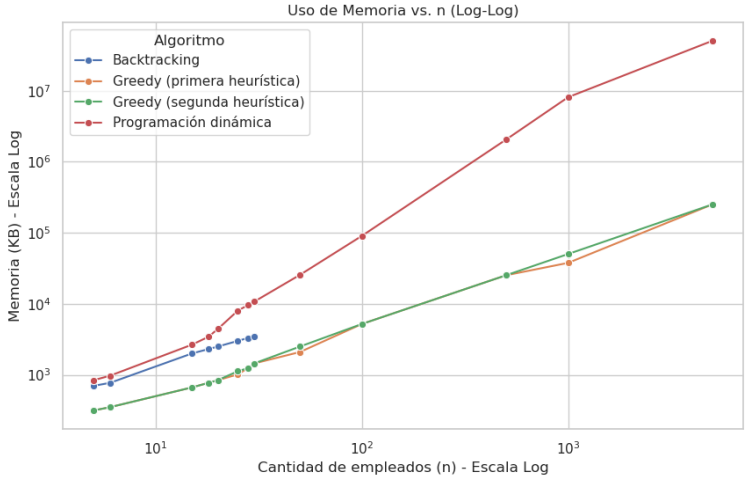

Figura 2: Uso de memoria vs cantidad de empleados

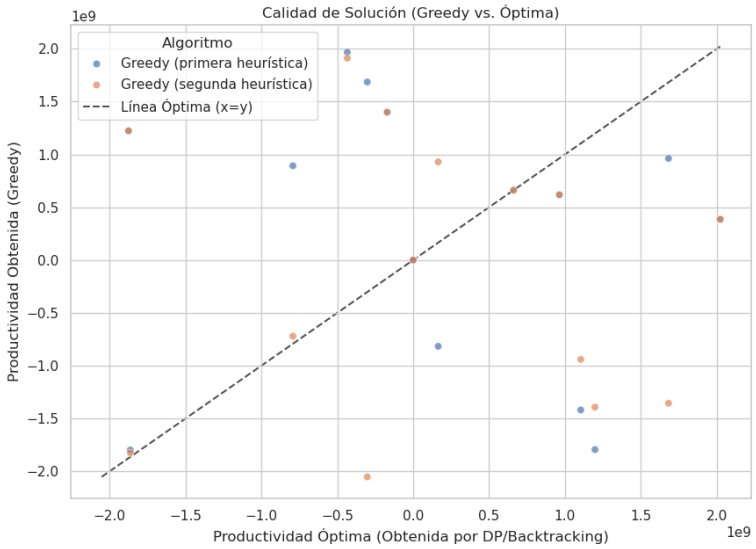

Figura 3: Scatterplot comparativo para calidad de solución

Al analizar los gráficos de rendimiento en escala Log-Log (Figura 1 y Figura 2), podemos extraer conclusiones claras sobre el costo computacional de cada enfoque.

4.2.1. Tiempo de ejecución

El gráfico de tiempo de ejecución confirma dramáticamente las complejidades teóricas.

- **Backtracking (Fuerza Bruta)**: La línea azul muestra un crecimiento que no es lineal en la escala Log-Log, sino que se curva bruscamente hacia arriba. Esto es la firma de una complejidad exponencial. El algoritmo se vuelve intratable rápidamente, y los experimentos se detienen alrededor de $n = 30$, golpeando el muro exponencial esperado. Más allá de ese valor, al volverse inviable, no se probó usando este algoritmo.

- **DP y Greedy**: Las otras tres implementaciones (DP, Greedy 1, Greedy 2) son drásticamente más rápidas. Sus líneas son rectas en la escala Log-Log, confirmando su naturaleza polinomial (analizada como $O(n^3)$). Para valores de n donde Backtracking ya es inviable ($n = 30$), estos algoritmos terminan en menos de un segundo.

4.2.2. Uso de Memoria

- **Backtracking y Heurísticas Greedy**: Las líneas azul (Backtracking), naranja (Greedy 1) y verde (Greedy 2) son casi indistinguibles. Muestran una línea recta con pendiente suave, consistente con una complejidad espacial lineal ($O(n)$)

- **Programación Dinámica**: La línea roja (DP) es la atípica. No solo consume más memoria base, sino que su pendiente en la gráfica Log-Log es visiblemente más pronunciada. Esto sugiere una

complejidad espacial polinomial de orden superior (p.ej., $O(n^2)$), lo cual se alinea con una de las conclusiones clave del informe: **DP consume una gran cantidad de memoria**.

4.2.3. Análisis de Calidad de Solución

El rendimiento no lo es todo; la solución debe ser correcta. La Figura 3 compara la productividad obtenida por las heurísticas (eje Y) contra la solución óptima verdadera obtenida por DP/Backtracking (eje X).

- **La línea punteada (x = y)** representa una solución perfecta (obtenida mediante DP y Backtracking, cuyas producciones calculadas son idénticas en cada caso de prueba).
- Como se observa en el gráfico, **casi ningún punto reposa sobre la línea**. Esto prueba experimentalmente la conclusión de que **Greedy no siempre llega a la solución correcta**.
- Más importante aún, los puntos (tanto de Greedy 1 como de Greedy 2) están extremadamente dispersos y, a menudo, muy lejos de la línea óptima. Por ejemplo, para una solución óptima de $1{,}5x10^9$, una heurística puede devolver $-1{,}7x10^9$.
- Esto responde a la pregunta de investigación: las heurísticas greedy implementadas **no son cercanas al óptimo**, para este problema, sus decisiones locales no logran capturar la estructura de la solución global y entregan resultados muy poco fiables (obviamente depende de las heurísticas elegidas, otras heurísticas pueden comportarse mejor o peor).

5. Conclusiones

Este trabajo implementó y comparó experimentalmente cuatro enfoques algorítmicos para un problema de optimización de particionamiento. El análisis de los resultados nos permite responder a la pregunta de investigación sobre el balance entre la optimalidad y la eficiencia.

- **La Fuerza Bruta (Backtracking) es inviable en la práctica.** A pesar de garantizar la solución óptima, su complejidad temporal exponencial ($O(n \cdot 2^n)$) la vuelve computacionalmente intratable para entradas que superen $n \approx 30$ empleados, como demostró la curva de crecimiento vertical en el gráfico de tiempo (Figura 1).

- **Las heurísticas Greedy son rápidas pero incorrectas.** Los dos algoritmos greedy (Maximizar Segmento y Maximizar Densidad) demostraron ser rápidos, con un rendimiento polinomial $O(n^3)$ similar al de la Programación Dinámica. Sin embargo, el análisis de calidad (Figura 3) reveló que **no garantizan la optimalidad** y, de hecho, sus soluciones son erráticas y a menudo se desvían significativamente del valor óptimo.

- **La Programación Dinámica es la única solución óptima y eficiente.** El enfoque DP es el único que logra el balance deseado: encuentra la **solución óptima garantizada** en un tiempo polinomial ($O(n^3)$), permitiendo resolver el problema para n grandes.

- **El costo de la optimalidad es la memoria.** El principal trade-off de la solución de Programación Dinámica no es el tiempo (comparte el $O(n^3)$ con las heurísticas), sino el **consumo de memoria**. Los gráficos (Figura 2) mostraron que su uso de memoria crece a un ritmo superior (posiblemente $O(n^2)$ debido al almacenamiento de subproblemas) en comparación con el crecimiento lineal ($O(n)$) de los otros tres algoritmos.

En definitiva, se comprueba que para este problema, la Programación Dinámica es el único método robusto, superando la inviabilidad de la fuerza bruta y la incorrección de las heurísticas greedy.

A. Apéndice 1

A.1. Pseudocódigos

A.1.1. Backtracking

```
// Función recursiva de Backtracking
long long resolver_backtracking(const vector<Empleado>& empleados, int start
    , int n) {
  // si ya cubrimos a todos los empleados (start == n), la productividad
      extra es 0
  if (start == n) return 0;

  long long mejor = LLONG_MIN;

  // Probar todos los posibles puntos de corte end desde start hasta el
      final
  for (int end = start; end < n; end++) {
    long long prod_segmento = productividad_equipo(empleados, start, end);

    long long total = prod_segmento + resolver_backtracking(empleados, end +
        1, n);
    mejor = max(mejor, total);
  }

  return mejor;
}

int calcular_productividad_total_backtracking(int n, vector<string> lineas)
    {
  vector<Empleado> empleados;
  empleados.reserve(n);

  // Leer la entrada
  for (auto& linea : lineas) {
    stringstream ss(linea);
    long long A, B;
    string C;
    ss >> A >> B >> C;
    empleados.push_back({A, B, C});
  }

  // Iniciar la recursión desde el empleado 0
  long long resultado = resolver_backtracking(empleados, 0, n);
  return static_cast<int>(resultado);
}
```

Listing 1: Algoritmo de Fuerza bruta

A.1.2. Greedy 1

```
int calcular_productividad_total_greedy1(int n, vector<string> lineas) {
  vector<Empleado> empleados;
  empleados.reserve(n);

  // leer la entrada
  for (auto& linea : lineas) {
    stringstream ss(linea);
    long long A, B;
    string C;
    ss >> A >> B >> C;
    empleados.push_back({A, B, C});
  }

  long long productividad_total = 0;
  int start = 0;

  // Iterar hasta agrupar a todos los empleados
  while (start < n) {
    long long mejor_prod_segmento = LLONG_MIN;
    int mejor_end = start;

    // Búsqueda local, hay que probar todos los posibles finales para el
        segmento actual
    // y quedarse con el que de el valor máximo absoluto
    for (int end = start; end < n; end++) {
      long long prod_actual = productividad_equipo(empleados, start, end);

      if (prod_actual > mejor_prod_segmento) {
        mejor_prod_segmento = prod_actual;
        mejor_end = end;
      }
    }

    // Acá está la decision greedy, confirmamos el segmento [start, mejor_end]
    productividad_total += mejor_prod_segmento;

    // avanzar el inicio para el siguiente equipo
    start = mejor_end + 1;
  }

  return static_cast<int>(productividad_total);
}
```

Listing 2: Algoritmo Greedy con primera heurística

A.1.3. Greedy 2

```
int calcular_productividad_total_greedy2(int n, vector<string> lineas) {
  vector<Empleado> empleados;
  empleados.reserve(n);

  // leer la entrada
  for (auto& linea : lineas) {
    stringstream ss(linea);
    long long A, B;
    string C;
    ss >> A >> B >> C;
    empleados.push_back({A, B, C});
  }

  long long productividad_total = 0;
  int start = 0;

  while (start < n) {
    double mejor_densidad = -1e18; // Inicializar con un valor muy bajo
    int mejor_end = start;

    // busqueda local
    for (int end = start; end < n; end++) {
      long long prod_actual = productividad_equipo(empleados, start, end);
      int num_empleados = end - start + 1;

      // Criterio greedy: Productividad / Cantidad de empleados
      double densidad_actual = static_cast<double>(prod_actual) /
          num_empleados;

      if (densidad_actual > mejor_densidad) {
        mejor_densidad = densidad_actual;
        mejor_end = end;
      }
    }

    // Recalculamos el valor real (entero) del mejor segmento encontrado para
        sumarlo al total
    productividad_total += productividad_equipo(empleados, start, mejor_end);
    start = mejor_end + 1;
  }

  return static_cast<int>(productividad_total);
}
```

Listing 3: Algoritmo Greedy con segunda heurística

A.1.4. Programación Dinámica

```
int calcular_productividad_total_dp(int n, vector<string> lineas) {
  vector<Empleado> empleados;
  empleados.reserve(n);

  // leer entrada
  for (auto& linea : lineas) {
    stringstream ss(linea);
    long long A, B;
    string C;
    ss >> A >> B >> C;
    empleados.push_back({A, B, C});
  }

  // Definición de estado
  // dp[i] almacena la productividad máxima posible considerando SOLO los
      primeros i empleados
  // El objetivo final es encontrar dp[n]
  vector<long long> dp(n + 1);

  // El caso base sería que 0 empleados generan 0 productivdad
  dp[0] = 0;

  // Llenado de la tabla Bottom-Up
  for (int i = 1; i <= n; i++) {
    dp[i] = LLONG_MIN; // Inicializar con un valor enano

    // Transición:
    // Para calcular dp[i], probamos todos los posibles "últimos equipos"
        formados por el segmento [j, i-1]
    // dp[i] será el maximo de (la solución óptima hasta j + lo que aporta el
        nuevo equipo [j, i-1])
    for (int j = 0; j < i; j++) {
      long long prod_ultimo_segmento = productividad_equipo(empleados, j, i -
          1);

      if (dp[j] != LLONG_MIN) { // check de seguridad (aunque dp[0] en teoría
          siempre es válido)
        dp[i] = max(dp[i], dp[j] + prod_ultimo_segmento);
      }
    }
  }

  return static_cast<int>(dp[n]);
}
```

Listing 4: Algoritmo de Programación Dinámica