Universidad del Valle de Guatemala

Facultad de ingeniería

Análisis y Diseño de Algoritmos

Catedrático: Bidkar Pojoy

Proyecto 2

Nelson Eduardo García Bravatti 22434

Joaquín André Puente Grajeda 22296

Guatemala, abril de 2025

Link repositorio:

https://github.com/nel-eleven11/Proy2_AyDA

Problema elegido:

Una montaña rusa de longitud L es una atracción que consta de 2L tramos, puestos uno detrás del otro. Solamente existen dos tipos de tramos: los ascendentes y los descendentes. Cada tramo mide exactamente una unidad de longitud en su componente horizontal y una unidad de longitud en su componente vertical. Es decir, que cada tramo avanza una unidad de longitud ascendente o descendente y una unidad de longitud horizontalmente, en función de si es un tramo ascendente o descendente. Además, las montañas rusas no tienen ningún tramo bajo tierra, y empiezan y acaban al nivel del suelo.

La altura de una montaña rusa es la distancia sobre el nivel del suelo de su punto más alto. Se pide calcular el número de montañas rusas de longitud L y altura H ($1 \leq L, H \leq 200$). El programa recibe T ($T \leq 4 * 104$) pares de enteros L, H y se debe imprimir la solución para cada uno de ellos módulo 10^9 +7.

Problema obtenido de:

https://aprende.olimpiada-informatica.org/algoritmia-dinamica-2

Algoritmos de solución:

Solución con programación dinámica:

#Variables globales:

MOD = 10^9 + 7

MAX_L = 200

MAX_H = 200

Se inicializa la matriz montañas

res_montanias = un array de 3 dimensiones de tamaño (2*MAX_L + 1) x (MAX_H + 1) x (MAX_H + 1) se inicializa todo en 0 menos: res_montanias[0][0][0] = 1

Calculando todas las posibles montañas con DP bottom-up

for t = 1 to 2*MAX_L:

for ch = 0 to MAX_H: # Iteramos sobre todas las posibles alturas

for mh = 0 to MAX_H: # Iteramos sobre todas las posibles alturas máximas alcanzables

```
# Si la altura es 0, el último tramo debe ser descendente
            if ch == 0:
            res_montanias[t][ch][mh] = res_montanias[t-1][1][mh]  # Solo puede ser ascendente el tramo anterior
            # Si es una altura intermedia (0 < ch < mh), el tramo anterior puede ser ascendente o descendente
            elif 0 < ch < mh:
                res_montanias[t][ch][mh] = (res_montanias[t-1][ch-1][mh] + res_montanias[t-1][ch+1][mh]) % MOD
            # Si es la altura máxima (ch == mh), solo puede ser ascendente el tramo anterior
            elif ch == mh:
                res_montanias[t][ch][mh] = (res_montanias[t-1][mh-1][mh] + res_montanias[t-1][mh-1][mh-1]) % MOD
# Responder las consultas
for (L, H) in mon:
La respuesta es res_montanias2*L][0][H], que es el número de montañas rusas de longitud L y altura máxima H
    print(res_montanias[2*L][0][H])
```

Explicación:

Se tiene MOD como convención para manejar valores grandes.

La altura máxima se establece como 200 unidades y la longitud máxima también ya que, para llegar a la altura máxima se necesitan 200 tramos y otros 200 para llegar al nivel del suelo.

Se establece el valor base res_montanias[0][0][0] = 1, porque hay exactamente una forma de tener una montaña rusa sin tramos y a nivel de altura 0.

La matriz se calcula de forma bottom-up desde t = 1 hasta 2*MAX_L. En cada iteración, se recorren todas las posibles alturas ch y las posibles alturas máximas alcanzables mh. Con las siguientes transiciones:

- Altura 0 (ch == 0), el tramo anterior debe ser descendente, por lo que la única transición posible es desde res_montaniast-1][1][mh].
- Altura intermedia (0 < ch < mh), el tramo anterior puede haber sido ascendente o descendente, por lo que la transición sería desde res_montanias[t-1][ch-1][mh] o res_montaniast-1][ch+1][mh].
- Altura máxima ch == mh, el tramo anterior solo puede haber sido ascendente. Aquí la transición es desde res_montanias[t-1][mh-1][mh] y res_montanias[t-1][mh-1][mh-1].

Después de haber precomputado todos los valores de dp, respondemos a cada consulta simplemente consultando el valor de res_montanias[2*L][0][H] para cada par (L, H) que se consulte.

Programa implementando programación dinámica:

https://github.com/nel-eleven11/Proy2_AyDA/blob/main/sol_dp.py

Solución con DaC:

MOD = 10^9 + 7

MAX_L = 200

MAX_H = 200

Matriz 3D para memoization: montanias[t][ch][mh] almacena el número de montañas montanias = arreglo tridimensional de tamaño [2*MAX_L + 1][MAX_H + 1][MAX_H + 1], inicializado a -1

Función recursiva para calcular el número de montañas rusas

Contar_montanias(t, ch, mh):

Caso base: Si no hay tramos, solo hay una forma de estar a nivel 0 y con altura máxima 0

if t == 0:

return 1 if (ch == 0 and mh == 0) else 0

if ch < 0 O ch > MAX_H O mh < 0 O mh > MAX_H

Return 0

if montanias[t][ch][mh] ≠ -1
return montanias[t][ch][mh]

#Caso 1: altura actual es 0 → el tramo anterior debió ser descendente desde altura 1

if ch == 0

result = contar_montanias(t - 1, 1, mh)

Caso 2: estamos en la altura máxima actual (ch == mh)

```
        elif ch == mh
            if mh > 0
                case1 =  contar_montanias(t - 1, mh - 1, mh)
                 case2 = contar_montanias(t - 1, mh - 1, mh - 1)
                 result = (case1 + case2) % MOD

        # Caso 3: estamos en una altura intermedia (0 < ch < mh)
        Elif:
            if: ch > 0 Y mh > 0 :
                 ascend = contar_montanias(t - 1, ch - 1, mh)   # subimos desde ch - 1
                 descend = contar_montanias(t - 1, ch + 1, mh)  # bajamos desde ch + 1
                 result = (ascend + descend) % MOD
    montanias[t][ch][mh] =  result
    Return result

# Función para procesar varias consultas
Calcular_montanias(queries):
    for (L, H) in queries:
        return resultado = contar_montanias(2 * L, 0, H)
```

Explicación:

contar_montanias(t, ch, mh): Representa el número de montañas rusas construidas con t tramos que terminan en la altura ch y cuya altura máxima alcanzada hasta ese momento es mh.

El objetivo es calcular contar_montanias(2*L, 0, H), es decir, el número de montañas rusas de longitud L (2L tramos), que terminan al nivel del suelo (ch = 0) y cuya altura máxima total es H.

Caso base: Si t == 0 (no hay tramos), solo existe una configuración válida.

El algoritmo se va dividiendo con las transiciones, en cada llamada recursiva contar_montanias(t, ch, mh) se considera cuál fue el tramo anterior y desde qué estado pudo venir. Se analizan 3 situaciones según la altura actual ch:

- Caso 1: Estamos en el suelo (ch == 0)
- Caso 2: Estamos en la altura máxima (ch == mh)

Solo es posible haber llegado aquí mediante un ascenso desde altura mh – 1

Hay dos posibilidades:

1. Ya habíamos alcanzado mh anteriormente.
2. Este es el primer momento en que se alcanza mh, por lo tanto, el mh anterior era mh - 1.

- Caso 3: Estamos en una altura intermedia (0 < ch < mh)

  El último tramo pudo ser:

  - Ascendente (desde ch - 1)
  - Descendente (desde ch + 1)

Programa implementando DaC:

https://github.com/nel-eleven11/Proy2_AyDA/blob/main/sol_dac.py

Análisis Teórico:

Dinamic Programming:

Al analizar el programa podemos analizarlo de la siguiente manera:

1) Construcción de la matriz tridimensional res_montanias[t][ch][mh]
   Ya que existe MAX_H y MAX_L = 200 podemos concluir que la inicialización de la matriz tiene un tiempo proporcional al número de elementos
   $T = O(L \cdot H^2)$ osea, O(401 * 201 * 201)
2) Cálculo Bottom-Up
   Tenemos 3 bucles anidados que se encargan de hacer operaciones aritméticas de complejidad O(1) y el número de iteraciones es de O(400 * 201 * 201).
3) Respuesta de consultas
   Ya que las consultas se hacen como un acceso directo a un arreglo. Esto tiene una complejidad de O(1) y para T Consultas tendría una complejidad de O(T).

Por lo tanto, la complejidad total del problema es de O(16,160,400 + T). Pero como MAX_L y MAX_H son constantes se puede simplificar a O(T).

Divide and Conquer:

Por medio del árbol de recursión cada llamada a contar_montanias(t, ch, mh) puede hacer hasta dos llamadas recursivas (como máximo), dependiendo del caso:

- Caso 1 (ch == 0): 1 llamada
- Caso 2 (ch == mh): hasta 2 llamadas

- Caso 3 (0 < ch < mh): 2 llamadas

El parámetro t disminuye en cada llamada: t-1. La profundidad máxima del árbol es t = O(L) ya que, se inicia en 2L y termina en 0. En el peor de los casos, cada nodo genera 2 hijos:

$$T(t) = 2T\,(t-1) + O(1)$$

$$T(t) = 2^t \cdot T(0) + O(2^t)$$

$$T(2L) = O(2^t)$$

$$T(2L) = O(2^{2L}) = O(4L)$$

$$T(L) = O(4^L)$$

El código almacena resultados previos en la estructura montanias[t][ch][mh], evitando recalcular subproblemas. Por lo tanto, cada combinación de (t, ch, mh) se calcula solo una vez.

Tamaño del espacio de estados

t: de 0 a 2L → O(L)

ch: de 0 a H → O(H)

mh: de 0 a H → O(H)

Esto da un total de: $Número\ total\ de\ subproblemas = O(L \cdot H^2)$

El costo de los subproblemas es de: $Costo\ por\ estado = O(1)$

Análisis Empírico:

Figura 1: Código para generar datos

```python
#Se genera el conjunto de datos de prueba
import random

def generar_pares(opcion):
    pares = []
    lista = []

    if opcion == 1:
        # Pares pequeños (L entre 10 y 50)
        for i in range(2, 32):
            L = random.randint(10, 50)
            H = random.randint(10, L)
            lista.append([L, H])

    elif opcion == 2:
        # Pares grandes (L entre 100 y 200)
        for i in range(2, 32):
            L = random.randint(100, 200)
            H = random.randint(100, L)
            lista.append([L, H])

    for i in range(1, len(lista) + 1):
        pares.append(lista[:i])

    return pares
```


Las entradas se generan aleatoriamente en el archivo main.py, sin embargo, se utiliza el mismo conjunto para probar ambas soluciones y se pueden generar varios conjuntos de datos para probar, además, existe la posibilidad de ingresar manualmente un par (L,H) para probar cualquiera de las soluciones.

Dinamic Programming:

Figura 2: Tiempo vs Entradas en DP con conjunto de datos pequeño

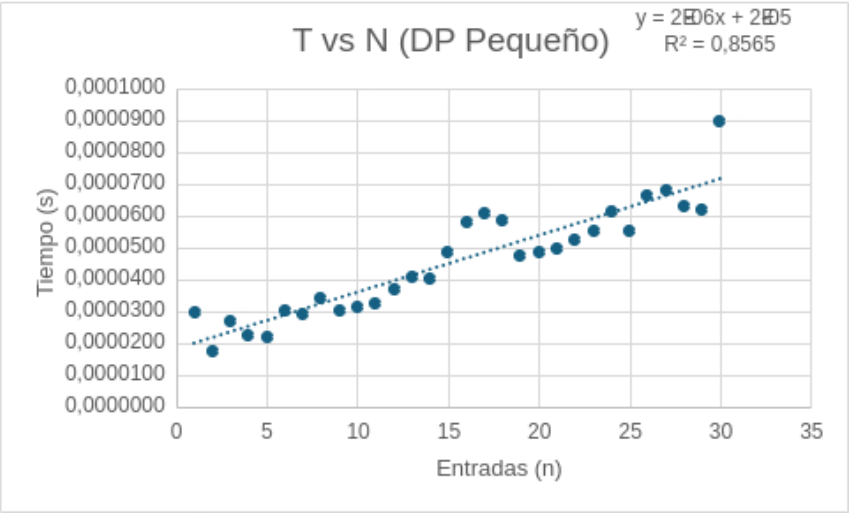
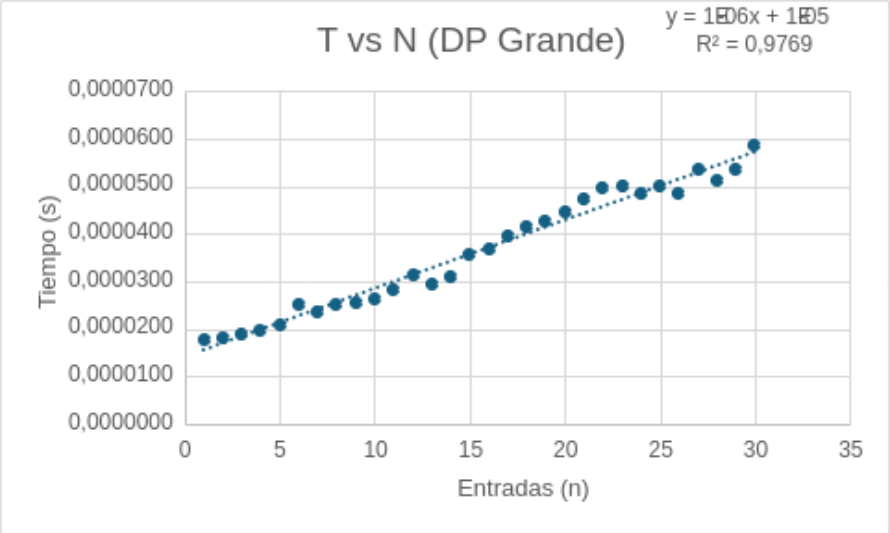


Figura 3: Tiempo vs Entradas en DP con conjunto de datos grande



Divide and Conquer:

Figura 4: Tiempo vs Entradas en DaC con conjunto de datos pequeño

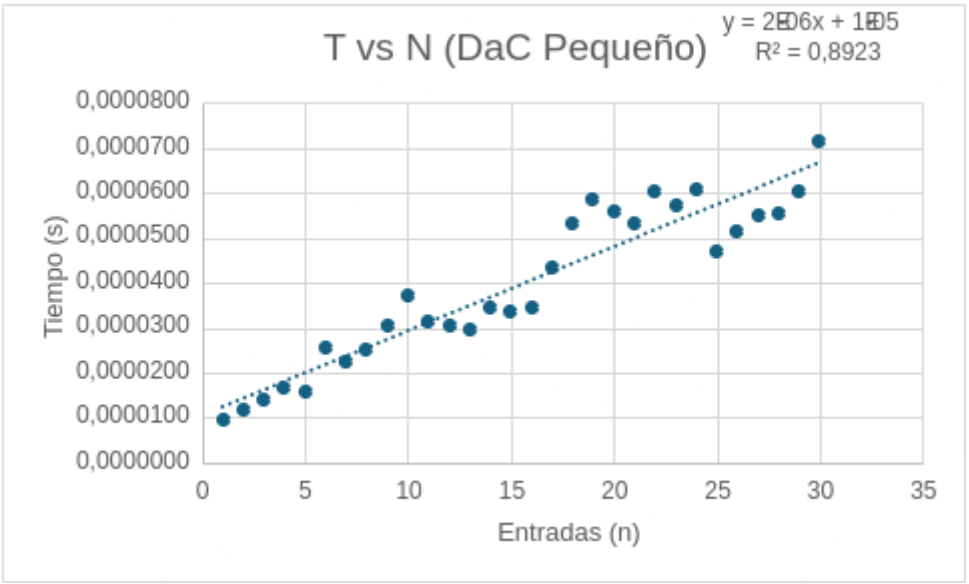
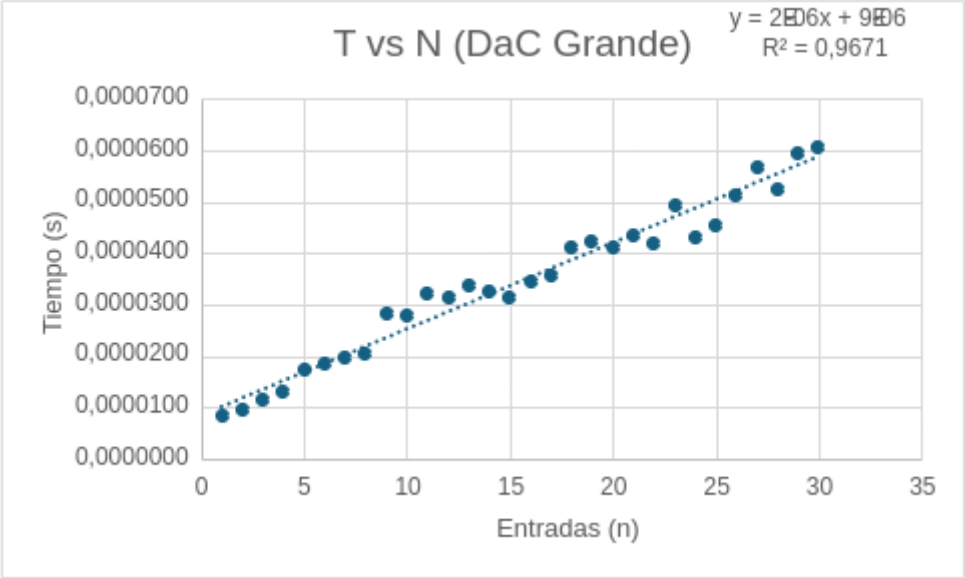


Figura 5: Tiempo vs Entradas en DaC con conjunto de datos grande



AyDA Proyecto 2 - UVG 2025

Las figuras 2 y 4 (datos pequeños) presentan un R^2 menor en comparación de las figuras 3 y 5 (datos grandes), esto puede deberse a que al recibir pares (L,H) cercanos al límite permitidio (50 y 200) estos hacen que se calculen las respuestas para alturas menores y como el conjunto de datos grande tiene un mayor rango de datos, estos se ven beneficiados por esto, teniendo tiempos de ejecución con menor distribución y menos datos atípicos.

Se puede decir que el algoritmo que presenta mejor oportunidad para manejar conjuntos de datos grandes es el de dinamic programming, al tener una menor complejidad al realizar las operaciones sin necesidad de hacer recursividad, y teniendo que hacer menos comparaciones que divide and conquer. Teniendo un R^2 promedio más alto que DaC.

Todas las gráficas validan de manera sólida el análisis teórico de complejidad para ambos algoritmos. Tanto el enfoque bottom-up iterativo como divide and conquer presentan crecimiento lineal cuando la altura H se mantiene constante, lo cual respalda la complejidad $O(L \cdot H^2)$deducida analíticamente. Las diferencias menores observadas en tiempos absolutos se explican por la implementación (recursión vs iteración) y el manejo de memoria.

Bibliografía:

Programación Dinámica (II): Ejemplos más avanzados | Aprende Programación Competitiva. (n.d.). https://aprende.olimpiada-informatica.org/algoritmia-dinamica-2

Soltys, M. (2018). An Introduction to the Analysis of Algorithms. World Scientific Publishing Company.