Análisis del Problema "Breaking Integers"

Adam Bourbahh Romero

26 de junio de 2025

1 Introducción al Problema

El problema "Breaking Integers", también conocido como "Integer Break", es un desafío fascinante en el campo de la **optimización matemática** y la **programación dinámica**. Este problema nos invita a explorar cómo la descomposición de un número entero positivo puede influir drásticamente en el producto de sus partes. El objetivo principal es encontrar la manera más eficiente de dividir un entero n en una suma de al menos dos enteros positivos, de tal forma que el producto de estos enteros resultantes sea el **máximo posible**. Este problema tiene aplicaciones teóricas en la optimización y es un excelente caso de estudio para aplicar técnicas algorítmicas.

—

2 Planteamiento del Ejercicio

Dado un entero positivo n, el problema consiste en descomponerlo en k enteros positivos, denotados como $n_1, n_2, \ldots, n_k$, bajo la estricta condición de que la suma de estos enteros sea igual a n (es decir, $n_1 + n_2 + \cdots + n_k = n$). Además, se requiere que el número de partes k sea al menos 2 ($k \geq 2$), lo que significa que siempre debemos "romper" el entero original en al menos dos fragmentos. Nuestro objetivo final es **maximizar el producto** de estos enteros descompuestos: $n_1 \times n_2 \times \cdots \times n_k$.

Para ilustrar mejor el problema, consideremos algunos ejemplos:

- Si $n = 2$: La única forma de dividir 2 en al menos dos enteros positivos es $1 + 1$. El producto resultante es $1 \times 1 = \mathbf{1}$.
- Si $n = 3$: Tenemos dos opciones principales para romper el número:
  - $1 + 1 + 1$: El producto es $1 \times 1 \times 1 = 1$.
  - $1 + 2$: El producto es $1 \times 2 = \mathbf{2}$.

  En este caso, el producto máximo es **2**.
- Si $n = 10$: Este ejemplo es más complejo y demuestra la necesidad de una estrategia de optimización:

- Una posible descomposición es 3+3+4. El producto es $3 \times 3 \times 4 = \mathbf{36}$.
- Otra descomposición es 3+3+2+2. El producto es $3 \times 3 \times 2 \times 2 = \mathbf{36}$.

Para $n = 10$, el producto máximo que se puede obtener es **36**.

El desafío reside en determinar sistemáticamente cómo dividir n para asegurar que el producto sea siempre el mayor posible, sin tener que probar todas las combinaciones, lo cual sería inviable para valores grandes de n.

—

3 Soluciones Propuestas

A continuación, se exploran dos enfoques distintos para resolver el problema "Breaking Integers", cada uno con su propia lógica y eficiencia.

3.1 Solución 1: Programación Dinámica

Explicación Esta solución aborda el problema utilizando la técnica de la **programación dinámica**, que construye la solución óptima de manera **iterativa**. La idea central es que el producto máximo para un número i puede encontrarse dividiendo i en dos partes, j e $i - j$, y luego multiplicando los productos máximos que ya se calcularon para esas partes. Es crucial comprender que `dp[j]` y `dp[i-j]` ya contienen los valores de los productos máximos si j o $i - j$ fueran los números originales a romper.

- **Casos Base:** Los casos para $n = 2$ y $n = 3$ son fundamentales y se manejan de forma explícita. Para $n = 2$, el problema requiere romperlo en al menos dos enteros, siendo la única opción $1 + 1$, cuyo producto es $1 \times 1 = 1$. Para $n = 3$, la mejor opción es $1 + 2$, con un producto de $1 \times 2 = 2$. Por lo tanto, se retorna $n - 1$ para ambos casos.
- **Arreglo `dp`:** Se inicializa un vector llamado `dp` de tamaño $n + 1$. Cada `dp[i]` se encargará de almacenar el **producto máximo** que se puede obtener al "romper" el entero i. **Inicialización de `dp`:**
  - `dp[1]=1`: Si el "subproblema" es 1, lo tratamos como un factor de 1 para futuras multiplicaciones.
  - `dp[2]=2`: Si el "subproblema" es 2, lo consideramos como el número 2 mismo para multiplicaciones. No lo rompemos en 1×1 porque 2 como factor da un producto mayor.
  - `dp[3]=3`: De manera similar, si el "subproblema" es 3, lo consideramos como 3. Esto es vital, por ejemplo, si estamos calculando `dp[5]` y una división es $2 + 3$, queremos usar $2 \times 3 = 6$, no `dp[2]` $\times dp[3]=$ `2` $\times 3 = 6$. `Es decir, para estos números pequeños, usar el número en sí como factor es mejor que romperlo si es parte de una suma mayor.`

- **Bucle Principal:** Se inicia un bucle que va desde $i = 4$ hasta n. Para cada valor de i, se busca el **producto máximo** al dividir i.
- **Bucle Interno:** Dentro del bucle principal, otro bucle itera con j desde 0 hasta $i/2$. En cada iteración, se evalúa la división de i en dos partes: j e $i - j$. El producto se calcula como `dp[i-j]` $\times$ `dp[j]`. Es fundamental recordar que `dp[j]` y `dp[i-j]` ya guardan los productos máximos óptimos para j e $i - j$ respectivamente.
- **Actualización de `dp[i]`:** El valor `maximo` encontrado durante las iteraciones del bucle interno se asigna a `dp[i]`.
- **Resultado Final:** Al concluir los bucles, `dp[n]` contendrá el producto máximo para el entero n, que es el resultado que se retorna.

Representación Gráfica de Dependencias (para $n = 5$)

Para comprender mejor cómo funciona la programación dinámica, podemos visualizar las dependencias entre los subproblemas. A continuación, se muestra un grafo que ilustra cómo se calcula `dp[5]`, basándose en los valores previamente calculados de `dp[1]` a `dp[4]`.

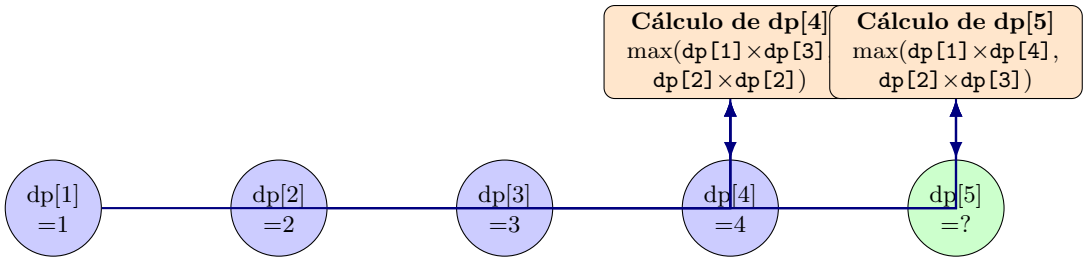

Figure 1: Grafo de Dependencias para la Solución de Programación Dinámica (Ejemplo $n = 5$)

Nodos redondos: Elementos de la tabla `dp` con sus valores precalculados.
Nodos rectangulares (naranjas): Representan el proceso de búsqueda del máximo producto para un `dp[i]`.
Flechas: Indican una dependencia. Los valores de `dp` apuntan a los cálculos que los utilizan, y el cálculo apunta al `dp` que se actualiza.
Ejemplo de Cálculo para `dp[5]`:

- Se consideran todas las divisiones de 5 en dos partes j e $i - j$:
- Para la división $1 + 4$: Producto `dp[1]` $\times$ `dp[4]` $= 1 \times 4 = 4$
- Para la división $2 + 3$: Producto `dp[2]` $\times$ `dp[3]` $= 2 \times 3 = 6$

El producto máximo es **6**, por lo tanto, `dp[5]`=6.

Este grafo muestra que para calcular `dp[i]`, se consideran todas las posibles divisiones de `i` en j y $i - j$, y se utiliza el producto de los valores ya calculados

de `dp` para esas subpartes. El valor final de `dp[i]` es el máximo de todos estos productos.

```cpp
class Solution {
public:
    int integerBreak(int n) {
        using namespace std;
        if(n==2 || n==3) return n-1;
        vector <int> dp(n+1);
        dp[1]=1;
        dp[2]=2;
        dp[3]=3;
       for(int i=4; i<=n; i++){
       int maximo=0;
            for(int j=0; j<=i/2; j++){
                maximo=max(maximo, dp[i-j]*dp[j]);
            }
        dp[i]=maximo;
        }
        return dp[n];
    }
};
```

Listing 1: Código de Solución con Programación Dinámica

—

3.2 Solución 2: Enfoque Basado en la Observación Matemática

Explicación Esta solución se basa en una **observación matemática clave** sobre los factores óptimos que maximizan el producto de una descomposición. Se ha demostrado que, para maximizar el producto, los factores de la descomposición deben ser principalmente **2s y 3s**. De hecho, es **más beneficioso** utilizar tantos 3s como sea posible, ya que 3^k crece más rápido que 2^m para sumas iguales o similares.

- **Casos Base:** Al igual que en la primera solución, los casos para $n = 2$ y $n = 3$ se manejan de manera especial, retornando $n - 1$.
- **Observación Clave y Justificación:** Para cualquier entero $n \geq 4$, la intuición detrás de esta solución radica en las siguientes propiedades:
  - **Evitar el factor 1:** Romper un número en un factor de 1 es subóptimo, ya que $1 \times x = x$. Siempre es mejor mantener el factor x intacto o descomponerlo en factores mayores que 1 para un producto potencialmente mayor.
  - **Factores óptimos: 2s y 3s:** Se puede demostrar que cualquier factor mayor o igual a 4 puede ser descompuesto en 2s y 3s para obtener un producto mayor. Por ejemplo:
    - Si tenemos un factor 4, es mejor reemplazarlo por 2×2, ya que $2 \times 2 = 4$.

* Si tenemos un factor 5, es mejor reemplazarlo por 2×3, ya que $2 \times 3 = 6 > 5$.
* Si tenemos un factor 6, es mejor reemplazarlo por $2 \times 2 \times 2 = 8$ o $3 \times 3 = 9$. Claramente $9 > 6$.

Esto implica que los factores óptimos en la descomposición serán únicamente 2s y 3s.

- **Preferencia por el factor 3:** Entre 2s y 3s, los 3s son generalmente preferibles para maximizar el producto. Por ejemplo, $3 > 2 + 1$ (es mejor 3 que 2×1) y $3 > 1 + 1 + 1$. Además, $3^2 = 9$ mientras que $2^3 = 8$. Para una suma de 6, $3 \times 3 = 9$ es mejor que $2 \times 2 \times 2 = 8$. Esta comparación resalta la eficiencia del factor 3.

- **Estrategia Basada en el Módulo 3:** La estrategia se define en función del residuo de n al dividirlo por 3:
  - **Si $n\%3 == 0$:** Esto significa que n es un múltiplo exacto de 3. En este caso, la estrategia óptima es dividir n en tantas veces 3 como sea posible. El producto máximo se obtiene multiplicando 3 por sí mismo $n/3$ veces, lo que se calcula como $3^{n/3}$.
  - **Si $n\%3 == 1$:** Si n tiene un residuo de 1 al dividirse por 3, significa que $n = 3k+1$ para algún entero k. Si solo usáramos 3s, nos quedaría un 1, lo cual es subóptimo. La mejor estrategia es agrupar este 1 con un 3 para formar un 4, que a su vez se rompe en 2×2. Así, n se transforma en $3 \times (k-1) + 4 = 3 \times (k-1) + 2 + 2$. El producto resultante será $2 \times 2 \times 3^{(n-4)/3}$. (Observa que $k = (n-1)/3$, por lo tanto, $k - 1 = (n-1)/3 - 1 = (n-4)/3$).
  - **Si $n\%3 == 2$:** Si n tiene un residuo de 2 al dividirse por 3, es decir, $n = 3k + 2$, la estrategia óptima es usar un factor 2 y el resto de 3s. El producto será 2×3^k, que se calcula como $2 \times 3^{n/3}$.

```cpp
class Solution {
public:
    int integerBreak(int n) {
        /*Es facil ver que todo n mero mayor que 4 se puede
        expresar como suma de 2s y 3s, y adem s esta es la
    expresion
        optima para tener el producto maximo:
        n%3==0=>S=3+...+3
        n%3==2=>S=3+...+3 + 2
        n%3==1=>S=3+.+3 + 2 + 2 */
        using namespace std;
        if(n==2 || n==3) return n-1;
        if(n%3==0) return pow(3, n/3);
        if(n%3==1) return 2*2*pow(3, (n-4)/3);
        if(n%3==2) return 2*pow(3, n/3);
        return 0;
    }
};
```

Listing 2: Código de Solución Basada en Observación Matemática