Resumen para Algoritmos y Estructuras de Datos III

Basado en esta guía de estudio.

Tomás Spognardi

2 de agosto de 2022

Índice general

Capítulo 1

Técnicas de diseño de algoritmos

1.1. Complejidad

1.1.1. Repaso de complejidad computacional

La complejidad computacional es una técnica de análisis de algoritmos, en particular, de su tiempo de ejecución. Es de carácter *teórico*: se basa en determinar matemáticamente la cantidad de operaciones que llevará a cabo el algoritmo para una instancia de tamaño dado, indendientemente de la máquina sobre la cuál se implementa y el lenguaje utilizado.

Definición informal

La complejidad de un algoritmo es una función $T_A : \mathbb{N} \longrightarrow \mathbb{R}_{>0}$ que representa el tiempo de ejecución en función del tamaño de la entrada. Para distinguir entre entradas de un mismo tamaño, se pueden considerar:

- **Complejidad de peor caso**:[1]

$$T_{\text{peor}}(n) = \text{máx}\,\{t_A(i) \mid I \in I_A, |I| = n\}$$

- **Complejidad de mejor caso**:

$$T_{\text{mejor}}(n) = \text{mín}\,\{t_A(i) \mid I \in I_A, |I| = n\}$$

- **Complejidad de caso promedio**:[2]

$$T_{\text{prom}}(n) = \sum_{I \in I_A, |I|=n} P(I) \cdot t_A(I)$$

Para ciertos algoritmos, conviene hacer un análisis más profundo que distingue entre tipos de instancias particulares al problema.

Estas definiciones no son rigurosas: no contienen ninguna indicación sobre cómo determinar T_A para un algoritmo A, y el "tiempo de ejecución" ni siquiera tiene unidades de medida. Para formalizarlas, es necesario definir un *modelo de cómputo*.

Modelo de cómputo: Máquina RAM

La *Máquina RAM* es una máquina abstracta que funciona como modelo de cómputo. Nos permite modelar computadoras en las que la memoria es suficiente y los enteros involucrados en los cálculos entran en una palabra[3]. Este modelo cuenta con:

- **Memoria Principal**: Una sucesión de celdas numeradas (tantas como se necesiten). Cada una puede guardar un entero de tamaño arbitrario.
- **Registro Acumulador**: un registro especiel se usa como (generalmente primer) operando en las operaciones.

[1] $t_A : I_A \longrightarrow \mathbb{R}_{>0}$ devuelve el tiempo de ejecución para una instancia particular del problema A.
[2] $P(I)$ es la probabilidad de que la entrada sea la instancia I.
[3] Una *palabra* es el tamaño de una celda de memoria

- **Acceso Aleatorio**: Acceso directo a cualquier celda en tiempo constante. También cuenta con direccionamiento indirecto: la dirección accedida puede ser el valor de una celda (un *puntero*).
- **Programa**: Se codifica en una serie de instrucciones, y se almacena en una memoria aparte de la principal. Hay un *contador de programa*, que identifica la próxima a ser ejecutada y puede ser manipulado a través de ciertas instrucciones (*jumps*).

Tanto la entrada como la salida son representadas como una sucesión de celdas numeradas, cada una con un entero de tamaño arbitrario. Para codificar un programa, es necesario definir un *set de instrucciones*. Un ejemplo posible sería el siguiente[4]:

- LOAD `valor` – Carga un valor en el acumulador.
- STORE `valor` – Carga el acumulador en un registro.
- ADD `valor` –Suma el operando al acumulador
- SUB `valor` – Resta el operando al acumulador
- MULT `valor` – Multiplica el operando por el acumulador
- DIV `valor` – Divide el acumulador por el operando
- READ `valor` – Lee un nuevo dato de entrada → operando
- WRITE `valor` – Escribe el operando a la salida
- JUMP `label` – Salto incondicional
- JGTZ `label` – Salta si el acumulador es positivo
- JZERO `label` – Salta si el acumulador es cero
- HALT – Termina el programa

Para calcular la complejidad de un programa, se asume que cada instrucción tiene un tiempo de ejecución constante. En ese caso, se puede definir $t_A(I) =$ suma de los tiempos de ejecución de las instrucciones ejecutadas por el algoritmo A para la instancia I. Esto es casi suficiente para calcular $T_A(n)$: solo resta definir $|I|$, el tamaño de la instacia.

Modelo uniforme

En este modelo, cada **dato individual** ocupa una celda de memoria, y cada operación básica tiene tiempo de ejecución constante. Esto resulta razonable cuando la entrada es una estructura de datos y cada dato entra en una palabra de memoria. Bajo esta suposición, el tamaño de entrada se define como la cantidad de datos individuales de la instancia.

Sin embargo, para algoritmos que operan sobre un entero particular, esta definición no resulta adecuada. Por ejemplo, se puede tomar el siguiente algoritmo, que determina si un número es o no primo:

```
ES-PRIMO(n)
1  for i = 2 to ⌈√n⌉
2      if n ≡ 0  mód i
3          return FALSE
4  return TRUE
```

Según la definición anterior, el tamaño de la entrada de ES-PRIMO es siempre 1, lo cual es anti-intuitivo: sería conveniente poder definir la complejidad de este algoritmo en función del tamaño de n.

Modelo logarítmico

En este caso, el tamaño de la instancia se define como la cantidad de símbolos de un **alfabeto** necesaria para representarla, y el tiempo de ejecución de cada operación elemental depende del tamaño de los operandos (definido de la misma manera). Esto es apropiado para algoritmos que toman como input un número fijo de datos individuales.

Para representar los datos, se suele tomar como alfabeto $\mathbb{B} = \{0, 1\}$, los dígitos binarios. En tal caso, el tamaño de un entero $n \in \mathbb{Z}$ es $L(n) = \lceil \log_2 n \rceil + 1$ bits, mientras que para almacenar una lista de m enteros se necesitan $L(m) + mL(N)$, donde N es el valor máximo posible en la lista.

[4]Este ejemplo no es de ninguna forma minimal, pero es similar a un set de instrucciones RISC para una computadora real.

1.1.2. Notación O

Para comparar tiempos de ejecución entre distintos algoritmos, es conveniente obviar constantes de proporcionalidad y enfocarse en el comportamiento asintótico de las complejidades. Con eso en mente, se definen las clases:

$$\begin{aligned} f \in \mathcal{O}(g) &\iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N} \mid \forall n \geq n_0,\ f(n) \leq c \cdot g(n) \\ f \in \Omega(g) &\iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N} \mid \forall n \geq n_0,\ f(n) \geq c \cdot g(n) \\ f \in \Theta(g) &\iff f \in \mathcal{O}(g) \wedge f \in \Omega(g) \end{aligned}$$

Informalmente, $f \in \mathcal{O}(g)$ implica que f crece a lo sumo tan rápido como g.

Complejidades comunes

- Si un algoritmo es $\mathcal{O}(\log n)$, se dice **logarítmico**.
- Si un algoritmo es $\mathcal{O}(n)$, se dice **lineal**.
- Si un algoritmo es $\mathcal{O}(n^2)$, se dice **cuadrático**.
- Si un algoritmo es $\mathcal{O}(n^3)$, se dice **cúbico**.
- Si un algoritmo es $\mathcal{O}(n^k)$, se dice **polinomial**.
- Si un algoritmo es $\mathcal{O}(k^n)$ $(k > 1)$, se dice **exponencial**.

Además, se tiene:

$$\begin{aligned} \mathcal{O}(n^k) &\subsetneq \mathcal{O}(d^n)\ \forall k, d \in \mathbb{N} \\ \mathcal{O}(\log n) &\subsetneq \mathcal{O}(n^k)\ \forall k \in \mathbb{R}_{>0} \end{aligned}$$

1.1.3. Problemas "bien resueltos" e intractabilidad

Un problema se denomina *bien resuelto* si existe un algoritmo de tiempo polinomial que lo resuelve. Esto se debe a que el tiempo de ejecución de los algoritmos exponenciales crece demasiado rápido: su ejecución puede resultar infactible para valores de n pequeños.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$O(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min
$O(2^n)$	1.00 ms	1.00 sg	17.90 min	12 días	35 años
$O(3^n)$	0.59 sg	58 min	6 años	3855 siglos	2×10^8 siglos!

Tabla de comparaciones de los posibles tiempos de ejecución para distintas clases de complejidad.

Sin embargo, cabe destacar que:

- Si los tamaños de instancias no son muy grandes, un algoritmo exponencial puede ser apropiados.
- Un algoritmo puede ser polinomial, pero con un exponente o una constante demasiado grande para ser aplicado en la práctica.
- Existen ciertos algoritmos con complejidad de peor caso exponencial, pero que en la práctica son muy eficientes (como el método *simplex*).

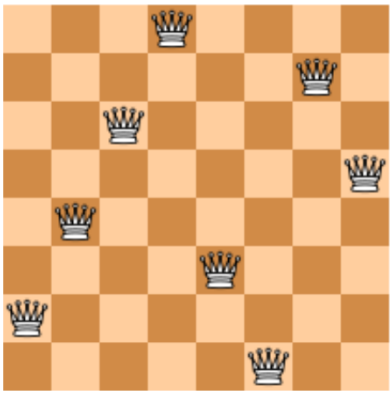
1.2. Backtracking

1.2.1. Fuerza bruta

Un algoritmo de *fuerza bruta* (también llamado de *búsqueda exhaustiva*) analiza todas las posibles configuraciones de la salida, hasta encontrar una que cumple con los requerimientos del problema.

Ejemplo: Problema de las n damas

> **Problema:**
> Ubicar n damas en un tablero de ajedrez de $n \times n$ casillas, de forma que ninguna dama amenace a otra.



Solución posible para el caso $n = 8$

Un posible algoritmo de fuerza bruta sería recorrer todos los posibles subconjuntos de n casillas, verificando si algún par de reinas se amenaza en caso de ser ubicarlas en las casillas del subconjunto. Sin embargo, esto no es muy eficiente: para $n = 8$, implicaría recorrer $\binom{64}{8} = 4{,}426{,}165{,}368$ combinaciones.

Se pueden lograr mejoras aprovechando la estructura del problema: como cada columna debe tener exactamente 1 dama, las configuraciones exploradas pueden representarse como un vector $(a_1, ..., a_n)$, con $a_i \in \{1, ..., n\}$ indicando la fila de la dama que está en la columna i. Además, cada fila tiene exactamente una reina, así que los elementos del vector no se repiten. Por ende, la cantidad de combinaciones se reduce a $n!$, que para el caso $n = 8$ es $8! = 40{,}320$. No obstante, esto puede mejorarse.

1.2.2. Backtracking

El backtracking es una técnica general de diseño de algoritmos que consiste de extender las soluciones parciales $a = (a_1, ..., a_k), k < n$, agregando un elemento a_{k+1} al final del mismo. Si se detecta que S_{k+1}, el conjunto de soluciones que tienen al vector como prefijo, es vacío, se retrocede a la solución anterior. Esto permite descartar configuraciones parciales apenas se determina que no pueden llevar a una solución. Los algoritmos de backtracking siguen el siguiente esquema general:

```
BT(a)
1  if ¬Es-Válida(a)
2       return
3  if Es-Solución(a)
4       Procesar(a)
5       return
6  for a' ∈ Sucesores(a)
7       BT(a')
```

Si solo se busca una solución, esto se puede volver más eficiente usando una variable global *encontró*:

```
BT(a)
 1  if ¬Es-Válida(a)
 2       return
 3  if Es-Solución(a)
 4       sol = a
 5       encontró = true
 6       return
 7  for a' ∈ Sucesores(a)
 8       BT(a')
 9       if encontró
10            return
```

Para que el backtracking sea eficiente, el procedimiento Es-Válida debe ser capaz de identificar algún conjunto de instancias inválidas, y no puede tener una complejidad demasiado grande.

En el problema de las n damas, se puede chequear en cada paso si alguna reina amenaza a la recién agregada. Esto se puede realizar en tiempo lineal, y por la construcción de las soluciones solo hace falta comprobar las amenazas diagonales. Utilizando este algoritmo, cualquier configuración de n elementos que no haya sido rechazada es una solución válida.

Ejemplo: Resolución de Sudokus

> **Problema:**
> Encontrar una asignación de números a casillas que resuelve un Sudoku particular.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Ejemplo de un Sudoku y su solución.

Los Sudokus pueden ser resueltos con un algoritmo de backtracking: las soluciones son extendidas agregando un número a algún casillero vacío. Cuando el nuevo número no cumple alguna de las restricciones del sudoku, la solución es rechazada. A pesar de ser exponencial, este algoritmo es muy eficiente en la práctica.

1.3. Programación Dinámica

1.3.1. Definición

La *programación dinámica* (PD/DP) es otra técnica de diseño de algoritmos. Es similar al *Divide & Conquer*, ya que se basa en dividir el problema en sub-problemas de menor tamaño, resolverlos recursivamente, y combinar las sub-soluciones para formar una solución. La diferencia con este método es que PD se utiliza en casos donde estos sub-problemas suelen superponerse, y aprovecha este hecho al resolverlos una única vez.

Para evitar repetir la resolución de sub-problemas equivalentes, los algoritmos de programación dinámica siguen alguno de estos dos esquemas:

- **Enfoque "top-down"**: Se implementa el algoritmo tradicionalmente, pero los resultados se guardan en una estructura de datos indexada por los parámetros de la llamada (*memoización*). Luego, antes de resolver ejecutar el algoritmo para una llamada, se chequea si sus parámetros están en esta estructura, y en tal caso se devuelve la solución previamente calculada.
- **Enfoque "bottom-up"**: Se resuelven los sub-problemas en un orden que asegura que las llamadas recursivas de cada uno son calculadas antes que este[5], guardando los resultados de cada llamada en una tabla.

[5] Esto representa un ordenamiento topológico del árbol de llamadas de la función (en realidad, del árbol invertido, donde cada nodo tiene una arista apuntando hacia aquellos que lo tienen como sub-problema).

Ejemplo: Cálculo de coeficientes binomiales

> **Problema:**
> Calcular el valor del coeficiente $\binom{n}{k}$, definido como:
>
> $$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

El problema se podría resolver calculando $\binom{n}{k}$ directamente, pero esto se dificulta para valores grandes. Por ejemplo, a pesar de que $\binom{100}{99} = 100$, el valor 100! es un número de 157 cifras, muy por encima del límite de 64 bits utilizados para representar enteros.

Una forma alternativa de realizar la operación sería haciendo uso del siguiente teorema:

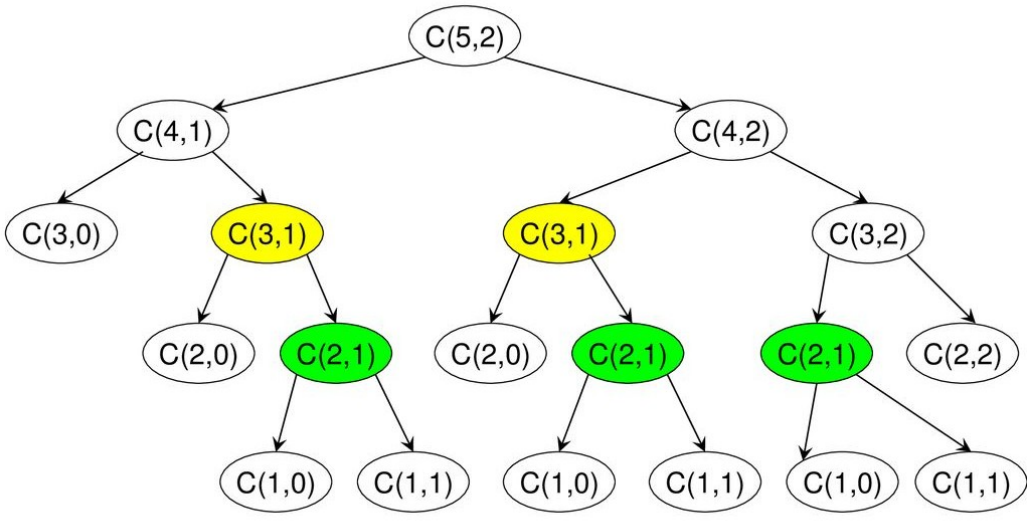
Teorema. *Si $n \geq 0$ y $0 \leq k \leq n$, entonces:*

$$\binom{n}{k} = \begin{cases} 1 & si\ k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & si\ 0 < k < n \end{cases}$$

Esta fórmula recursiva se puede implementar directamente:

```
COMBINATORIO(n, k)
1  if k == 0 ∨ k == n
2      return 1
3  else
4      return COMBINATORIO(n − 1, k − 1) + COMBINATORIO(n − 1, k)
```

Este método tiene una complejidad de $\Omega(\binom{n}{k})$, y evita calcular factoriales, pero podría ser más eficiente, ya que al ejecutarlo se repiten llamadas con los mismos parámetros.



Árbol de llamadas de COMBINATORIO para la instancia $n = 5, k = 2$

Acá es donde entra en juego la programación dinámica. El siguiente algoritmo bottom-up calcula una única vez cada coeficiente necesario:

COMBINATORIO-PD(n, k)

```
1  Inicializar matriz A ∈ ℕ^{n×k}
2  for i = 1 to n
3        A[i][0] = 1
4  for j = 0 to k
5        A[j][j] = 1
6  for i = 2 to n
7        for j = 2 to mín {i − 1, k}
8              A[i][j] = A[i − 1][j − 1] + A[i − 1][j]
```

La complejidad de este método es $\mathcal{O}(nk)$, y $\mathcal{O}(nk) \subseteq \mathcal{O}(n^2)$, ya que $k \leq n$. Además, se puede implementar con una complejidad espacial de $\mathcal{O}(k)$ almacenando solo la fila actual y la anterior de la tabla en el ciclo.

1.3.2. Principio de optimalidad de Bellman

Un problema satisface el *principio de optimalidad de Bellman* cuando para cualquier *sucesión óptima* de decisiones, cada *subsucesión* es a su vez óptima para el subproblema asociado. Esta es una condición necesaria para que aplicar PD sea eficiente.

Ejemplo: Problema de la mochila

Problema:
Dados

- Capacidad $C \in \mathbb{Z}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{Z}_+$ de objetos.
- Peso $p_i \in \mathbb{Z}_+$ del objeto i.
- Beneficio $b_i \in \mathbb{Z}_+$ del objeto i.

Determinar qué objetos se deben incluir en la mochila para **maximizar** el beneficio total, sin **excederse** del peso máximo C. Formalmente, encontrar:

$$\arg\max \left\{ \sum_{s \in S} b_s \mid S \subseteq \{1, ..., n\},\ \sum_{s \in S} p_s \leq C \right\}$$

Para resolver este problema utilizando PD, se puede definir la función $m(k, D)$ como el valor óptimo para el problema considerando solo los primeros k objetos y una mochila con capacidad D. Los valores de esta función pueden ser guardados en una tabla de $n \times C$ posiciones. Los valores se pueden calcular de manera recursiva:

$$m(k, D) = \begin{cases} 0 & \text{si } k = 0 \vee D \leq 0 \\ \max \{m(k-1, D), b_k + m(k-1, D-p_k)\} & \text{en caso contrario} \end{cases}$$

Esto contempla, para cada k dos posibilidades: o bien el objeto de índice k está en la solución óptima, y entonces $m(k, D) = b_k + m(k-1, D-p_k)$, o bien no, en cuyo caso $m(k, D) = m(k-1, D)$.

Si esta función se implementa directamente en un algoritmo de PD (ya sea top-down o bottom-up) utilizando una matriz como estructura de memoización, tanto la complejidad temporal como la espacial son $\mathcal{O}(nC)$. Esta complejidad es *pseudopolinomial*: está acotada por un polinomio, pero este incluye valores númericos del input, no solo el tamaño del mismo.

Solución óptima

Calcular $m(k, D)$ nos da el *valor óptimo*, pero no la *solución óptima*. Para obtener el conjunto de objetos que resulta en ese valor se debe reconstruir a partir de la tabla calculada. El esquema general para la reconstrucción se basa en la observación anterior: recorriendo los índices de atrás para adelante, si $m(k, D) = b_k + m(k-1, D-p_k)$, entonces el valor k está en (alguna) solución. Si no, es porque $m(k-1, D) > b_k + m(k-1, D-p_k)$, es decir, ignorar el objeto k resulta en un mayor beneficio total. Este procedimiento permite obtener el conjunto solución en tiempo lineal (una vez que ya se ejecutó el algoritmo anterior).

Ejemplo: Multiplicación de matrices

> **Problema:**
> Dadas $M_1, M_2, ..., M_n$, calcular:
>
> $$M = M_1 \times M_2 \times ... \times M_n$$
>
> Realizando la menor cantidad de multiplicaciones entre números de punto flotante.

La dificultad de este problema radica en que la cantidad de operaciones realizadas depende de la forma en la que se asocie el producto. Para resolverlo, se puede observar que alguna de las multiplicaciones tiene que ser la última realizada, es decir, para algún i, se deben multiplicar primero las matrices de 1 a i por un lado y las de $i+1$ a n por el otro, y finalmente multiplicar estos 2 resultados. Estos dos sub-problemas ($M_1 \times M_2 \times ... \times M_i$ y $M_{i+1} \times M_{i+2} \times ... \times M_n$) deben ser resueltos, a su vez, de forma óptima.

Luego, suponiendo que las dimensiones de las matrices están dadas por un vector $d \in \mathbb{N}^{n+1}$ tal que $M_i \in \mathbb{R}^{d[i-1] \times d[i]}$, se puede implementar el siguiente algoritmo bottom-up:

MIN-OPERACIONES(d)
1 Inicializar la matriz $m \in \mathbb{N}^{n \times n}$
2 **for** $i = 1$ **to** n
3 $\quad m[i][i] = 0$
4 **for** $i = 1$ **to** $n - 1$
5 $\quad m[i][i+1] = d[i-1]d[i]d[i+1]$
6 **for** $s = 2$ **to** $n - 2$
7 $\quad$ **for** $i = 1$ **to** $n - s$
8 $\quad\quad m[i][i+s] = \min \{m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s] \mid i \leq k < i+s\}$

En este caso, $m[i][j]$ representa la cantidad mínima de operaciones necesarias para calcular $M_i \times M_{i+1} \times ... \times M_j$, y por ende el valor óptimo es $m[1][n]$. Para obtener la secuencia de multiplicaciones, se puede emplear un procedimiento similar al del ejemplo anterior.

1.4. Algoritmos Golosos

1.4.1. Heurísticas

Una *heurística* para un problema dado es un procedimiento computacional que intenta obtener soluciones de "buena calidad" para el mismo. Por ejemplo, para un problema de optimización, una heurística obtendría una solución con un valor cercano al óptimo.

Un algoritmo A es ϵ-aproximado cuando:

$$\left| \frac{x_A - x^*}{x^*} \right| \leq \epsilon$$

Donde x^* es el valor óptimo, y x_A es el resultado del algoritmo.

Un ejemplo práctico es el algoritmo de Christofides y Serdykuov, un algoritmo $\frac{1}{2}$-aproximado para instancias del problema del viajante de comercio que forman un espacio métrico (las distancias son simétricas y obedecen la desigualdad triangular). Lo notable de este algoritmo es que tiene complejidad polinómica, siendo el TSP un problema NP-Completo.

1.4.2. Algoritmos golosos

Los *algoritmos golosos* se basan en construir una solución para un problema seleccionando en cada la "mejor" alternativa, sin considerar (o haciéndolo débilmente) las implicancias posteriores de esa selección. Habitualmente, proporcionan heurísitcas sencillas para los problemas de optimización, produciendo soluciones razonables (aunque subóptimas) en tiempos eficientes. Sin embargo, existen casos donde la solución que generan es óptima.

Ejemplo: Problema de la mochila

A pesar de haber resuelto el problema anteriormente, un enfoque goloso puede proveer soluciones (subóptimas) con mayor eficiencia temporal. El esquema general es agregar a la mochila el objeto i que...

1. ...tenga el mayor beneficio b_i.
2. ...tenga el menor peso p_i.
3. ...maximice $\frac{b_i}{p_i}$ (la "densidad").

Se puede demostrar que, si se corre el algoritmo goloso 2 veces, una con el primer criterio y otra con el segundo, alguno de los resultados tiene un valor de al menos la mitad de la solución óptima. Esto hace al procedimiento un algoritmo $\frac{1}{2}$-aproximado, y se puede implementar en tiempo $\mathcal{O}(n \log n)$ si se ordenan los elementos previamente (es aún más eficiente usar una colas de prioridad implementadas con heap).

Por otro lado, si cambia el problema, permitiendo poner una fracción de cada elemento en la mochila, el algoritmo goloso que utiliza el tercer criterio devuelve soluciones óptimas.

Ejemplo: Problema del cambio

> **Problema:**
> Dado un monto m y un conjunto de denominaciones $d_1, ..., d_k$, encontrar la mínima cantidad de monedas necesarias para obtener el valor m.

Para encontrar soluciones (no necesariamente óptimas) de este problema, se puede emplear un algoritmo goloso simple: en cada paso, seleccionar la moneda de mayor valor que no exceda el monto restante.

```
DAR-CAMBIO(D, m)
1  suma = 0
2  M = {}
3  while suma < m
4        próxima = máx {d | d ∈ D, d ≤ m}
5        M = M ∪ {próxima}
6        suma = suma + próxima
7  return M
```

Para ciertos conjuntos de denominaciones, como el tradicional ($\{1, 5, 10, 25, 50\}$), este algoritmo siempre devuelve soluciones óptimas, mientras que para otros no (en $D = \{1, 5, 10, 12\}, m = 21$, el algoritmo devuelve un conjunto de 6 monedas cuando la solución óptima tiene 3).

El algoritmo es goloso porque en cada paso selecciona la moneda de mayor valor posible, sin preocuparse que esto puede llevar a una mala solución, y nunca modifica una decisión tomada.

Ejemplo: Tiempo de espera total en un sistema

> **Problema:**
> Un servidor tiene n clientes que puede atender en cualquier orden, y el tiempo necesario para atender al cliente i es $t_i \in \mathbb{R}_+$. Encontrar un orden de atención que minimice el tiempo de espera total de todos los clientes.

Si se denota $I = (i_1, ..., i_n)$ al orden de atención, el tiempo de espera total T se puede calcular de la siguiente manera:

$$T = t_{i_1} + (t_{i_1} + t_{i_2}) + ... = \sum_{k=1}^{n} (n - k) t_{i_k}$$

Se puede plantear el siguiente algoritmo goloso: En cada paso, atender al cliente pendiente que tenga el menor tiempo de atención. La idea detrás de ese criterio es que el tiempo de los clientes que son atendidos primero tendrá que ser esperado por todos los demás, así que lo ideal es que sea el mínimo. Formalmente, la solución $I = (i_1, ..., i_n)$ es una que cumple $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, ..., n - 1$.

En este caso, la solución que proporciona el algoritmo resulta ser óptima. Por otro lado, la complejidad temporal es $\mathcal{O}(n \log n)$, ya que el procedimiento es equivalente a ordenar a los clientes por tiempo de espera.

1.5. Algoritmos Probabilísticos

Un *algoritmo probabilístico* es uno que emplea un grado de aleatoriedad en su ejecución. Los efectos de esta aleatoriedad pueden variar: en algunos casos solo varía el tiempo de ejecución, mientras que en otros la salida tiene una probabilidad de ser incorrecta.

1.5.1. Algoritmos numéricos

Un *algoritmo numérico probabilístico* es uno que aproxima la solución a un problema matemático. Estos algoritmos suelen ser adaptaciones aleatorizadas de algoritmos clásicos, como el método de cuadratura bayesiana para la integración numérica, o el de optimización bayesiana para problemas de optimización.

1.5.2. Algoritmos de Monte Carlo

Los *algoritmos de Monte Carlo* son aquellos que proporcionan una respuesta que tiene cierta probabilidad (típicamente baja) de ser incorrecta. En general, si estos algoritmos se corren varias veces, la probabilidad de que la respuesta obtenida sea correcta aumenta (asumiendo independencia entre las distintas ejecuciones). Un ejemplo de estos algoritmos sería el test de primalidad de Solovay-Strassen, que siempre identifica a números primos correctamente, pero tiene una probabilidad menor a $\frac{1}{2}$ de devolver una respuesta falsa para los compuestos.

1.5.3. Algoritmos de Las Vegas

Los *algoritmos de Las Vegas* siempre devuelven una respuesta cuando terminan, pero su tiempo de ejecución es aleatorio (potencialmente infinito). Un ejemplo podría ser un algoritmo para el problema de n damas que chequea configuraciones aleatorias hasta encontrar una que satisface las restricciones.

1.5.4. Algoritmos de Sherwood

Los *algoritmos de Sherwood* son algoritmos que aleatorizan procesos determinísticos, habitualmente aquellos que tienen una gran diferencia entre el peor caso y el promedio. El ejemplo clásico es el algoritmo de quicksort con pivote seleccionado aleatoriamente.