Algorítmica

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Algorítmica

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2023-2024

Índice general

Introducción

El siguiente documento no es sino el mero resultado proveniente de la amalgación proficiente de un cúmulo de notas, todas ellas tomadas tras el transcurso de consecutivas clases magistrales, primordialmente —empero, no exclusivamente— de teoría, en simbiosis junto con una síntesis de los apuntes originales provenientes de la asignatura en la que se basan los mismos. Como motivación para la asignatura, introducimos a continuación un par de problemas que sabremos resolver tras la finalización de esta:

Ejercicio (Parque de atracciones)**.** Disponemos de un conjunto de atracciones

$$A_1, A_2, \ldots, A_n$$

Para cada atracción, conocemos la hora de inicio y la hora de fin. Podemos proponer varios retos de programación acerca de este parque de atracciones:

1. Seleccionar el mayor número de atracciones que un individuo puede visitar.
2. Seleccionar las atraciones que permitan que un visitante esté ocioso el menor tiempo posible.
3. Conocidas las valoraciones de los usuarios, $val(A_i)$, seleccionar aquellas que garanticen la máxima valoración conjunta en la estancia.

Ejercicio. Una empresa decide comprar un robot que deberá soldar varios puntos (n) en un plano. El software del robot está casi terminado pero falta diseñar el algoritmo que se encarga de decidir en qué orden el robot soldará los n puntos. Se pide diseñar dicho algoritmo, minimizando el tiempo de ejecución del robot (este depende del tiempo de soldadura que es constante más el tiempo de cada desplazamiento entre puntos, que depende de la distancia entre ellos). Por tanto, deberemos ordenar el conjunto de puntos minimizando la distancia total de recorrido.

Nociones de conceptos

A lo largo de la asignatura, será común ver los siguientes conceptos, los cuales aclararemos antes de empezar la misma:

- Instancia: Ejemplo particular de un problema.
- Caso: Instancia de un problema con una cierta dificultad.

Generalmente, tendremos tres casos:

- El mejor caso: Instancia con menor número de operaciones y/o comparaciones.
- El peor caso: Instancia con mayor número de operaciones y/o comparaciones.
- Caso promedio. Normalmente, será igual al peor caso.

Para notar la eficiencia del peor caso usaremos $O(\cdot)$, mientras que para el mejor caso, $\Omega(\cdot)$.

Diremos que un algoritmo es *estable* en ordenación si, dado un criterio de ordenación que hace que dos elementos sean iguales en cuanto a orden, el orden de stos vendrá dado por el primero se que introdujo en la entrada.

Ejemplo. Dado el criterio de que un número es menor que otro si es par, ante la instancia del problema: $1, 2, 3, 4$. La salida de un algoritmo de ordenación estable según este criterio será:

$$2, 4, 1, 3$$

Sin embargo, un ejemplo de salida que podría dar un algoritmo no estable sería:

$$4, 2, 3, 1$$

Los datos se encuentra ordenados pero no en el orden de la entrada.

Algoritmos de ordenación

A continuación, un breve reapso de algoritmos de ordenación:

- Burbuja es el peor algoritmo de ordenación.
- Si tenemos pocos elementos, suele ser más rápido un algoritmos simple como selección o inserción. Entre estos, selección hace muchas comparaciones y pocos intercambios, mientras que inserción hace menos comparaciones y más intercambios. Por tanto, ante datos pesados con varios registros, selección será mejor que insercción.
- Cuando se tienen muchos elementos, es mejor emplear un algoritmo de ordenación del orden $n \log(n)$.

1. Eficiencia de Algoritmos

La asignatura se centrará en eficiencia basada en el tiempo de ejecución (no en la eficiencia en cuanto espacio, memoria usada por el programa). Para calcular la eficiencia de un algoritmo, tenemos tres métodos:

- Método empírico: donde se mide el tiempo real.
- Método teórico: donde se mide el tiempo esperado.
- Método híbrido: tiempo teórico evitando las constantes mediante resultados empíricos.

Proposición 1.1 (Principio de Invarianza)**.** *Dadas dos implementaciones $I1$, $I2$ de un algoritmo, el tiempo de ejecución para una misma instancia de tamaño n, $T_{I1}(n)$ y $T_{I2}(n)$, no diferirá en más de una constante multiplicativa. Es decir, $\exists K > 0$ que verifica:*

$$T_{I1}(n) \leqslant K \cdot T_{I2}(n)$$

Por este teorema, podremos despreciar las constantes. En un principio, se asumiriá que operaciones básicas como sumas, multiplicaciones, ... serán de tiempo constante, salvo excepciones (por ejemplo, multiplicaciones de números de 100000 dígitos).

Definición 1.1 (Notación O)**.** Se dice que un algoritmo A es de orden $O(f(n))$, donde f es una función $f : \mathbb{N} \to \mathbb{R}^+$, cuando existe una implementación del mismo tamaño cuyo tiempo de ejecución $T_A(n)$ es menor igual que $K \cdot f(n)$, donde K es una constante real positiva a partir de un tamaño grande n_0. Formalmente:

$$A \text{ es } O(f(n)) \iff \exists K \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid T_A(n) \leqslant K \cdot f(n) \quad \forall n \geqslant n_0$$

La notación O nos permite conocer cómo se comportará el algoritmo en términos de eficiencia en instancias del caso peor del problema. Como mucho, sabemos que el algoritmo no tardará más de $K \cdot f(n)$ en ejecutarse en el peor de los casos.

Al decir que el algoritmo A es de orden $O(f(n))$, decimos que siempre podemos encontrar una constante positiva K que para valores muy grandes del caso n (a partir de un n_0), el tiempo de ejecución del algoritmo siempre será inferior a $K \cdot f(n)$:

$$T_A(n) \leqslant K \cdot f(n)$$

Ejemplos de órdenes de eficiencia son:

- Constante, $O(1)$.

- Logarítmico, $O(\log(n))$.
- Lineal, $O(n)$.
- Cuadrático, $O(n^2)$.
- Exponencial, $O(a^n)$.
- $\vdots$

Proposición 1.2 (Principio de comparación)**.** *Para saber si dos órdenes $O(f(n))$ y $O(g(n))$ son equivalentes o no, aplicamos las siguientes reglas:*

$$O(f(n)) \equiv O(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = K \in \mathbb{R}^+$$
$$O(f(n)) > O(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$
$$O(f(n)) < O(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Entendiendo que un órden es menor que otro si es mejor, es decir, más rápido en el caso asintótico.

Ejemplo. Si tenemos dos algoritmos A y B con órdenes de eficiencia $O(n^2)$ y $O((4n+1)^2+n)$ respectivamente, tratamos de ver qué algoritmos es más eficiente:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{(4n+1)^2+n)} = \lim_{n\to\infty} \frac{n^2}{(16n^2+1+2\cdot 4n\cdot 1)+n} = \lim_{n\to\infty} \frac{1}{16}$$

Gracias a la Proposición **??**, tenemos que los algoritmos A y B son equivalentes.

Ejemplo. En esta ocasión, tenemos a dos algoritmos A y B con órdenes de eficiencia de $O(2^n)$ y $O(3^n)$, respectivamente.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$$

Por la Proposición **??**, A es más eficiente que B.

Ejemplo. El algoritmo A tiene una eficiencia $O(n)$ y el algoritmo B tiene una eficiencia de $O(n\log(n))$. Buscamos cuál es más eficiente.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{n\log(n)} = \lim_{n\to\infty} \frac{1}{\log(n)} = 0$$

Por lo que A es más eficiente que B, por la Proposción **??**.

Ejemplo. Disponemos de dos algoritmos, A y B con órdenes de eficiencia $O((n^2+29)^2)$ y $O(n^3)$ respectivamente. Intuimos que B es más eficiente que A pero queremos probarlo.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{(n^2+29)^2}{n^3} = \infty$$

Gracias a la Proposción **??**, hemos probado lo que esperábamos; B es más eficiente que A.

Ejemplo. Se quiere probar que $O(\log(n))$ es más eficiente que $O(n)$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{\log(n)} = \lim_{n\to\infty} \frac{10^n}{n} = \infty$$

Por la Proposción **??**, lo acabamos de probar.

Ejemplo. Se quiere dar un ejemplo de que el órden de eficiencia de los logaritmos es equivalente sin importar la base de este. Podemos ver qué sucede con $O(\log_2(n))$ y con $O(\log_3(n))$:

$$\lim_{n\to\infty} \frac{\log_3(n)}{\log_2(n)} = \lim_{n\to\infty} \frac{\ln(2)}{\ln(3)}$$

Por lo que ambos algoritmos tienen el mismo órden de eficiencia.

Definición 1.2 (Notación Ω)**.** Se dice que un algoritmo A es de orden $\Omega(f(n))$, donde f es una función $f : \mathbb{N} \to \mathbb{R}^+$, cuando existe una implementación del mismo tamaño cuyo tiempo de ejecución $T_A(n)$ es mayor igual que $K \cdot f(n)$, donde K es una constante real positiva a partir de un tamaño grande n_0. Formalmente:

$$A \text{ es } \Omega(f(n)) \iff \exists K \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid T_A(n) \geqslant K \cdot f(n) \quad \forall n > n_0$$

La notación Ω nos permite conocer cómo se comportará el algoritmo en términos de eficiencia en instancias del caso mejor del problema. Como poco, sabemos que el algoritmo no t ardará menos de $K \cdot f(n)$ en ejecutarse, en el mejor de los casos.

Definición 1.3 (Notación Θ)**.** Se dice que un algoritmo A es de orden exacto $\Theta(f(n))$, donde f es una función $f : \mathbb{N} \to \mathbb{R}^+$, cuando existe una implementación del mismo tamaño cuyo tiempo de ejecución $T_A(n)$ es igual que $K \cdot f(n)$, donde K es una constante real positiva a partir de un tamaño grande n_0. En este caso, el algoritmo es simultáneamente de orden $O(f(n))$ y $\Omega(f(n))$.

$$A \text{ es } \Theta(f(n)) \iff \exists K \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid T_A(n) = K \cdot f(n) \quad \forall n > n_0$$

Propiedades

A continuación, vemos algunas propiedades de las notaciones anteriormente vistas, todas ellas demostradas en el Ejercicio **??**:

Reflexiva

$$f(n) \in O(f(n))$$

También se da para las notaciones Ω y Θ.

Simétrica

$$f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$$

Suma Si $T_1(n) \in O(f(n))$ y $T_2(n) \in O(g(n))$. Entonces:

$$T_1(n) + T_2(n) \in O(\text{máx}(f(n), g(n)))$$

Producto Si $T_1(n) \in O(f(n))$ y $T_2(n) \in O(g(n))$. Entonces:

$$T_1(n) \cdot T_2(n) \in O(f(n) \cdot g(n))$$

Regla del máximo

$$O(f(n) + g(n)) = \text{máx}(O(f(n)), O(g(n)))$$

Regla de la suma

$$O(f(n) + g(n)) = O(f(n)) + O(g(n))$$

Regla del producto

$$O(f(n) \cdot g(n)) = O(f(n)) \cdot O(g(n))$$

Puede suceder que el tamaño del problema no depende de una única variable n, sino de varias. En estos casos, se analiza de igual forma que en el caso de una variable, pero con una función de varias variables. Conocida una función $f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^+$:

$$A \text{ es } O(f(n,m)) \iff \exists K \in \mathbb{R}^+ \mid T_A(n,m) \leqslant K \cdot f(n,m) \quad \forall n, m \in \mathbb{N}$$

Ejemplo. El órden de eficiencia del algoritmo canónico (el que todos conocemos) de suma de matrices $n \times m$ es de órden $O(n \cdot m)$.

1.1. Análisis de algoritmos

El primer paso a la hora de determinar la eficiencia de un algoritmo es identificar qué parámetro determina el tamaño del problema (n). Posteriormente, tenemos que tener claro como se analiza cada estructura del código:

1. Operaciones elementales.
2. Secuencias de sentencias.
3. Sentencias condicionales.
4. Sentencias repetitivas.
5. Llamadas a funciones no recursivas.
6. Llamadas a funciones recursivas.

Sentencias simples u operaciones elementales

Son aquellas instrucciones cuya ejecución no depende del tamaño del caso, como por ejemlo:

- Operaciones matemáticas básicas (sumas, multiplicaciones, ...).
- Comparaciones.
- Operaciones booleanas.

Su tiempo de ejecución está acotado superiormente por una constante. Su órden es $O(1)$.

Secuencias de sentencias

Constan de la ejecución de secuencias de bloques de sentencias:

```
Sentencia_1;
Sentencia_2;
// etc
Sentencia_r;
```

Suponiendo que cada sentencia i tiene eficiencia $O(f_i(n))$, la eficiencia de la secuencia se obtiene mediante las reglas de la suma y del máximo:

$$O(f_1(n) + f_2(n) + \cdots + f_r(n)) = \text{máx}\,[O(f_1(n)), O(f_2(n)), \ldots, O(f_r(n))]$$

Ejemplo. Un ejemplo que puede parecer confuso es el siguiente:

```
if(n == 10){
    for(int i = 0; i < n; i++){
        cout << "Hola" << endl;
    }
}
```

En este caso, se trata de un código de orden $O(1)$, ya que es para un valor fijo de n, 10. No depende por tanto del tamaño.

1.1.1. Sentencias condicionales

Constan de la evaluación de una condición y la ejecución de un bloque de sentencias. Puede ejecutarse la *Sentencia1* o la *Sentencia2*, en función de la veracidad o falsedad de la condición:

```
if(condicion){
    Sentencia_1;
}else{
    Sentencia_2;
}
```

Peor caso

El órden de eficiencia del peor caso (notación O) viene dado por:

$$O(\text{estructura condicional}) = \text{máx}\,[O(\text{condicion}), O(\text{Sentencia1}), O(\text{Sentencia2})]$$

Demostración. Como justificación para la fórmula, démonos cuenta de que la ejecución de la estructura condicional es igual a una de las siguientes secuencias de instrucciones (dependerá de la condición la ejecución de una o de otra):

```
bool a = condicion;
Sentencia_1;
```

```
bool a = condicion;
Sentencia_2;
```

La notación O trata de buscar el órden del mayor tiempo de ejecución, por lo que buscaremos la secuencia que más tarde de las dos:

$$O(\text{estructura condicional}) = \text{máx}\left[O(\text{Secuencia1}), O(\text{Secuencia2})\right]$$

Usando la relga para secuencias de instrucciones vista anteriormente, podemos expresar cada órden como:

$$\begin{aligned} O(\text{Secuencia1}) &= \text{máx}\left[O(\text{condicion}), O(\text{Sentencia1})\right] \\ O(\text{Secuencia2}) &= \text{máx}\left[O(\text{condicion}), O(\text{Sentencia2})\right] \end{aligned}$$

Por lo que:

$$\begin{aligned} O(\text{estructura condicional}) &= \text{máx}\left[O(\text{Secuencia1}), O(\text{Secuencia2})\right] = \\ &= \text{máx}\left[\text{máx}\left[O(\text{condicion}), O(\text{Sentencia1})\right],\right. \\ &\qquad \left.\text{máx}\left[O(\text{condicion}), O(\text{Sentencia2})\right]\right] = \\ &= \text{máx}\left[O(\text{condicion}), O(\text{Sentencia1}), O(\text{Sentencia2})\right] \end{aligned}$$

□

Mejor caso

El órden de eficiencia del mejor caso (notación Ω) viene dado por:

$$\Omega(\text{estructura condicional}) = \Omega(\text{condicion}) + \text{mín}\left[\Omega(\text{Sentencia1}), \Omega(\text{Sentencia2})\right]$$

Demostración. Buscamos expresar el órden de ejecución en el mejor caso. Al igual que en la justificación anterior, el bloque condicional es equivalante a seleccionar una de dos secuencias:

```
bool a = condicion;
Sentencia_1;
```

```
bool a = condicion;
Sentencia_2;
```

Nos damos cuenta de que la condición siempre se ejecuta y luego se ejecuta una sentencia. Como estamos en el mejor caso, seleccionamos la sentencia que tarde menos tiempo en ejecutarse. Por esto, tenemos que el órden de ejecución es la suma del órden de la condición más el mínimo del órden de las dos sentencias. □

Ejemplo. Supongamos el siguienbte código:

```
if(n % 2 == 1){
    cout << "Es impar";
}else{
    // código de órden n
}
```

Aplicamos las fórmulas anteriormente vistas, sabiendo que la condición y la salida son de órden $O(1)$ al ser sentencias simples:

$$\begin{aligned} O(\text{estructura condicional}) &= \text{máx}\left[O(\text{condicion}), O(\text{Sentencia1}), O(\text{Sentencia2})\right] \\ &= \text{máx}\left[O(1), O(1), O(n)\right] = O(n) \end{aligned}$$

$$\begin{aligned} \Omega(\text{estructura condicional}) &= \Omega(\text{condicion}) + \text{mín}\left[\Omega(\text{Sentencia1}), \Omega(\text{Sentencia2})\right] \\ &= \Omega(1) + \text{mín}\left[\Omega(1), \Omega(n)\right] = \Omega(1) \end{aligned}$$

1.1.2. Sentencias repetitivas

Constan de la evaluación de uan condición y la ejecución de un bloque de sentencias, mientras que dicha condición se cumpla. Tienen la siguiente forma:

```
Mientras(condicion){
    BloqueSentencias;
}
```

Suponiendo que:

- El bloque de sentencias tiene eficiencia $f(n)$.
- La evaluación de la condición tiene eficiencia $g(n)$.
- El bucle se ejecuta $h(n)$ veces.

Entonces, la eficiencia será:

$$O(\text{estructura repetitiva}) = O(g(n) + h(n) \cdot (g(n) + f(n)))$$

Demostración. La condición se comprueba al menos una vez, por ello se suma. Cada iteración tiene un costo de $g(n) + f(n)$. El bucle realiza $h(n)$ iteraciones. □

Ejemplo. Dado el siguiente código, calcular su eficiencia:

```
while(n > 0){
    cout << n;
    n--;
}
```

- Evaluación de la condición: $g(n) = 1$.
- Bloque de sentencias: $f(n) = \text{máx}(O(1), O(1)) = 1$.
- Repeticiones: $h(n) = n$.

$$\begin{aligned} O(\text{estructura repetitiva}) &= g(n) + h(n) \cdot (g(n) + f(n)) \\ &= 1 + n \cdot (1 + 1) = 2 \cdot n + 1 \Rightarrow O(2 \cdot n + 1) \end{aligned}$$

Aplicando la regla del máximo: $O(2 \cdot n + 1) = \text{máx}(O(2 \cdot n), O(1)) = O(2 \cdot n)$. Simplificando la constante: La secuencia repetitiva es $O(n)$.

Bucles for

Constan de la inicialización de una variable, comprobación de una condición y actualización de la variable. Se ejecutará un bloque de sentencias mientras que la condición se cumpla:

```
for(Inicialización; Condición; Actualización){
    BloqueSentencias;
}
```

Suponiendo que:

- El bloque de sentencias tiene eficiencia $f(n)$.
- La evaluación de la condifición tiene eficiencia $g(n)$.
- El bucle se ejecuta $h(n)$ veces.
- La actualización tiene eficiencia $a(n)$.
- La inicialización tiene eficiencia $i(n)$.

La eficiencia de una estructura de este tipo viene dada por:

$$O(\text{for}) = O(i(n) + g(n) + h(n) \cdot (g(n) + f(n) + a(n)))$$

Demostración. La inicialización tiene lugar una vez. La condición se comprueba al menos una vez, por ello se suma. Cada iteración tiene un costo de $g(n)+f(n)+a(n)$. El bucle realiza $h(n)$ iteraciones. □

Ejemplo. Dado el siguiente código, calcular su eficiencia:

```
while(n > 0){
    for(int i = 1; i <= n; i*=2){
        cout << i;
    }
    n--;
}
```

Comenzamos analizando el bucle interno:

- Inicialización: $O(1)$.
- Condición: $O(1)$.
- Actualización: $O(1)$.
- Bloque de sentencias: $O(1)$.
- Veces que se ejecuta: $O(\log_2(n))$.

Eficiencia del bucle:

$$O(1) + O(1) + O(\log_2(n)) \cdot (O(1) + O(1) + O(1)) = O(\log_2(n))$$

Ahora, analizamos el bucle externo:

- Condición: $O(1)$.
- Bloque de sentencias: $O(\log_2(n) + 1)$.
- Veces que se ejecuta: $O(n)$.

Eficiencia del bucle:

$$O(1) + O(n) \cdot (O(1) + O(\log_2(n) + 1)) = O(n \cdot \log_2(n))$$

Despreciando la base del logaritmo:

$$O(n \log(n))$$

1.1.3. Funciones no recursivas

La eficiencia de la función per se se calcula como una secuencia de sentencias o bloques. Por otra parte, la eficiencia de la llamada a la función depende de si sus parámetros de entrada dependen o no del tamaño del problema. Esto se entenderá mejor en los siguientes ejemplos. Dada la siguiente función (que usaremos en varios ejemplos):

```
bool esPrimo(int valor){
    double tope = sqrt(valor);
    for(int i = 2; i <=tope; i++){
        if(valor % i == 0)
            return false;
    }

    return true;
}
```

El cuerpo de la función per se tiene efieciencia $O(\sqrt{n})$, ya que:

$$T(n) = 1 + \sum_{i=2}^{\sqrt{n}} 1 = 1 + \sqrt{n} - 2 + 1 = \sqrt{n}$$

Ejemplo.
```
for(int i = 1; i<n; i++){
    if(esPrimo(1234567))
        cout << i;
}
```

El tiempo en calcular `esPrimo(1234567)` es constante, luego es de eficiencia $O(1)$. Se repite n veces, luego la eficiencia del bucle es de $O(n)$.

Ejemplo.
```
for(int i = 1; i<n; i++){
    if(esPrimo(i))
        cout << i;
}
```

En este caso, el tiempo es:

$$T(n) = \sum_{i=1}^{n-1} \sqrt{i} = \sqrt{1} + \sqrt{2} + \cdots + \sqrt{n-1}$$

Demostraremos que $T(n) \in O(n\sqrt{n})$ usando el Criterio de Stolz. Para ello, calculamos el cociente de términos consecutivos:

$$\left\{\frac{T(n) - T(n-1)}{n\sqrt{n} - (n-1)\sqrt{n-1}}\right\} = \left\{\frac{\sqrt{n-1}}{n\left(\sqrt{n} - \sqrt{n-1}\right) + \sqrt{n-1}}\right\} =$$

$$= \left\{\frac{1}{n\left(\sqrt{\frac{n}{n-1}} - 1\right) + 1}\right\} = \left\{\frac{1}{n\left(\frac{1}{\sqrt{1-\frac{1}{n}}} - 1\right) + 1}\right\} =$$

$$= \left\{\frac{1}{\frac{\left(\frac{1}{\sqrt{1-\frac{1}{n}}} - 1\right)}{\frac{1}{n}} + 1}\right\} \overset{(*)}{\to} \frac{1}{\frac{1}{2} + 1} = \frac{1}{\frac{3}{2}} = \frac{2}{3}$$

Por el Criterio de Stolz, tenemos que:

$$\lim_{n\to\infty} \frac{T(n)}{n\sqrt{n}} = \frac{2}{3} \in \mathbb{R}^+$$

Por tanto, tenemos que la eficiencia es de orden $T(n) \in O(n\sqrt{n})$. En el proceso, en $(*)$, hemos aplicado que:

$$\lim_{n\to\infty} \frac{\left(\frac{1}{\sqrt{1-\frac{1}{n}}} - 1\right)}{\frac{1}{n}} \overset{L'Hôpital}{=} \lim_{n\to\infty} \frac{-\frac{1}{1-\frac{1}{n}} \cdot \frac{1}{2\sqrt{1-\frac{1}{n}}} \cdot \frac{1}{n^2}}{-\frac{1}{n^2}} = \frac{1}{2}$$

Ejemplo.
```
for(int i = 1; i<2000; i++){
    if(esPrimo(i))
        cout << i;
}
```

El tiempo de ejecución no depende de n, siempre tarda lo mismo (es decir, es constante), luego es de orden $O(1)$.

Ejemplo.
```
for(int i = n; i>0; i/=2){
    if(esPrimo(i))
        cout << i;
}
```

Intuitivamente, vemos que la eficiencia es de $O(\log(n)\sqrt{n})$, ya que repite $\log(n)$ veces una orden de eficiencia $O(\sqrt{n})$.

Ejemplo.
```
int LlamarV(int *s, int N){
    for(int i = N-1; i > 0; i = i/2)
        V[i] = V[i]-1;
    return V[0];
}

void Ejemplo(int *v, int N){
    for(int i=0; i<N; i++){
        v[i] = (i*2+20-4*i)/N;
        v[i] = LlamarV(v, N-1)*LlamarV(v, N-2);
    }
}
```

La función `LlamarV` tiene un tiempo de ejecución de $O(\log(n))$. El bucle de `Ejemplo` se repite n veces, luego es de orden $O(n\log(n))$.

Ejemplo.
```
void ejemplo1(int n){
    int i, j, k;
    for(i = 0; i<n; i++){
        for(j = 0; j<n; j++){
            C[i][j] = 0;
            for(k = 0; k<n; k++){
                C[i][j] = A[i][k] - B[k][j];
            }
        }
    }
}
```

Tiene eficiencia $O(n^3)$.

Ejemplo.
```
bool esPalindromo(char v[]){
    bool pal = true;
    int inicio = 0, fin = strlen(v)-1;

    while((pal) && (inicio<fin)){
        if(v[inicio] != v[fin])
            pal = true;
        inicio++;
        fin--;
    }

    return pal;
}
```

Se trata de un algoritmo de orden $O\left(\frac{n}{2}\right) = O(n)$.

Ejemplo.
```
void F(int num1, int num2){
    for(int i = num1; i<= num2; i*=2){
        cout << i << endl;
    }
}
```

Tenemos un algoritmo de dos variables, con $n = num2 - num1$. Se repite $\log(n)$ veces, luego eficiencia:

$$O(\log(n)) = O(\log(num2 - num1))$$

Ejemplo.
```
void F(int* v, int num, int num2){
    int i = -1, j = num2;

    while(i <= j){
        do{
            i++; j--;
        }while(v[i]<v[j]);
    }
}
```

Es un algoritmo que no funciona correctamente en todos los cacos, en un caso puede no llegar a terminar.

1.1.4. Funciones recursivas

Se expresa como una ecuación en recurrencias y el orden de eficiencia es su solución. Primero, suponemos que hay un caso base que se encarga de conocer la solución al problema en un caso menor. Si la solución al caso base la podemos manipular para dar una solución a un caso mayor, tenemos el problema resuelto.

Para solucionar la ecuación en recurrencias, tratamos de buscar la expresión general de la ecuación y luego podremos resolverla, tal y como se elustra en los siguientes ejemplos. Sin emargo, como se apreciará en el Ejemplo **??**, no resuelve todos los algoritmos recursivos. Para ello, deberemos avanzar en teoría.

Ejemplo (Factorial)**.**

```
unsigned long factorial(int n){
    if(n<=1) return 1;
    else return n*factorial(n-1);
}
```

La eficiencia de este algoritmo viene dada por la función $T(n)$:

$$\begin{cases} T(n) = T(n-1) + 1 \\ T(0) = T(1) = 1 \end{cases}$$

Tratamos de dar una expresión general a esta función:

$$\begin{aligned} T(n) &= T(n-1) + 1 = T(n-2) + 2 = T(n-3) + 3 \\ &= T(n-k) + k \quad \forall k \in \mathbb{N} \end{aligned}$$

Tomando $k = n - 1$:

$$T(n) = T(n-(n-1)) + (n-1) = T(1) + n - 1 = \cancel{1} + n - \cancel{1} = n$$

Por lo que $T(n) \in O(n)$.

Ejemplo.

```
int algoritmo(int n){
    if(n <= 1){
        int k = 0;
        for(int = 0; i < n; i++){
            k += k*i;
        }
        return k;
    }else{
        int r1, r2;
        r1 = algoritmo(n-1);
        r2 = algoritmo(n-1);
        return r1*r2;
    }
}
```

La eficiencia viene dada por $T(n)$:

$$\begin{cases} T(n) = 2T(n-1) + 1 \\ T(1) = 1 \end{cases}$$

Tratamos de resolver la ecuación:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ T(n-1) &= 2T(n-2) + 1 \\ T(n-2) &= 2T(n-3) + 1 \end{aligned}$$

$$T(n) = 2T(n-1) + 1 = 2[2T(n-1) + 1] + 1 = 2^2T(n-2) + 2 + 1 = \dots$$
$$= 2^kT(n-k) + \sum_{i=0}^{k-1} 2^i \quad \forall k \in \mathbb{N}$$

Para $k = n - 1$:

$$2^{n-1}T(1) + \sum_{i=0}^{n-2} 2^i = \sum_{i=0}^{n-1} 2^i = 2^{n+1} - 1 \in O(2^n)$$

Ejemplo. Plantedada la siguiente función que define el tiempo de ejecución de un algoritmo, se pide determinar el orden de eficiencia del algoritmo:

$$\begin{cases} T(n) = T(n-2) + 1 \\ T(1) = 1 \\ T(0) = 1 \end{cases}$$

Pasamos a resolver el ejercicio:

$$\begin{aligned} T(n) &= T(n-2) + 1 \\ T(n-2) &= T(n-4) + 1 \\ T(n-4) &= T(n-6) + 1 \end{aligned}$$

$$T(n) = T(n-4) + 1 + 1 = T(n-6) + 3 = T(n-8) + 4 = T(n-10) + 5$$
$$= T(n-2k) + k \quad \forall k \in \mathbb{N}$$

Para $k = \frac{n}{2}$ si n es par:

$$T(n) = T\left(n - \frac{2n}{2}\right) + \frac{n}{2} = T(0) + \frac{n}{2} = 1 + \frac{n}{2} \in O(n)$$

Para $k = \frac{n-1}{2}$ si n es impar:

$$T(n) = T\left(n - \frac{2(n-1)}{2}\right) + \frac{n-1}{2} = T(1) + \frac{n-1}{2} = 1 + \frac{n-1}{2} \in O(n)$$

Ejemplo (Fibonacci)**.** Dada la función del tiempo de ejecución de Fibonacci:

$$\begin{cases} T(n) = T(n-1) + T(n-2) + 1 \\ T(1) = T(0) = 1 \end{cases}$$

Se pide calcular la expresión de la función para determinar su orden de eficiencia.

$$T(n-1) = T(n-2) + T(n-3) + 1$$
$$T(n-2) = T(n-3) + T(n-4) + 1$$

$$T(n) = T(n-1) + T(n-2) + 1 = T(n-2) + T(n-3) + 1 + T(n-3) + T(n-4) + 1 + 1 =$$
$$= T(n-3) + T(n-4) + 1 + 2[T(n-4) + T(n-5) + 1] + 1 + T(n-5) + T(n-6) + 1$$

En resumen, este método no es útil para resolver este problema.

1.2. Ecuación característica

Se usa para resolver ecuaciones recurrentes que salen en análisis de eficiencia de algoritmos. Vamos a ver dos etapas, que corresponden con cómo se solucionan:

- Ecuaciones lineales homogéneas de coeficientes constantes.
- Ecuaciones lineales no homogéneas de coeficientes constantes.

En análisis de algoritmos nunca tendremos ecuaciones homogéneas, ya que es normal tener un +1 por sentencias constantes, usuales en código.

1.2.1. Ecuaciones homogéneas

Raíces distintas

Dada una ecuación del estilo:

$$T(n) = a_1T(n-1) + a_2T(n-2) + \cdots + a_kT(n-k)$$

Tratamos de buscar la solución a la ecuación. Realizamos el cambio de variable $T(n) = x^n$:

$$x^n = a_1x^{n-1} + a_2x^{n-2} + \cdots + a_kx^{n-k}$$

$$\begin{aligned} x^n - a_1x^{n-1} - a_2x^{n-2} - \cdots - a_kx^{n-k} &= 0 \\ = x^{n-k}(x^n - a_1x^{n-1} - a_2x^{n-2} - \cdots - a_k) & \end{aligned}$$

$$p(x) = (x^k - a_1x^{k-1} - a_2x^{k-2} - \cdots - a_k) = 0$$

Se trata de un polinomio de grado k. Suponiendo que $r_1, \ldots, r_k$ son las raíces del polinomio (todas distintas), definimos:

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

Ejemplo (Fibonacci)**.**

$$T(n) = T(n-1) + T(n-2)$$

Se pide determinar el orden de eficiencia de $T(n)$.

$$\begin{aligned} x^n - x^{n-1} - x^{n-2} &= 0 \\ x_2 - x - 1 &= 0 \end{aligned}$$

El polinomio característico es:

$$p(x) = x^2 - x - 1$$

Buscamos raíces:

$$x = \frac{1 \pm \sqrt{1+1}}{2}$$

Luego:

- $r_1 = \dfrac{1+\sqrt{5}}{2}$
- $r_2 = \dfrac{1-\sqrt{5}}{2}$

$$t_n = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Con $c_1, c_2 \in \mathbb{R}$.
Y se tiene que :

$$T(n) \in O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

Raíces con multiplicidad distinta de 1

En caso de que las raíces no sean todas distintas, nuestro polinomio es del estilo:

$$p(x) = (x - r_1)^{m_1}(x - r_2)^{m_2} \cdots (x - r_s)^{m_s}$$

Si r_i es una raíz con multiplicidad m_i, tenemos que:

$$t_n = \sum_{i=1}^{s} \left(\sum_{j=0}^{m_i - 1} c_{ij} r_i^n n^j \right)$$

Ejemplo. Dada la función de tiempo de un algoritmo:

$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3)$$

Se quedaría:

$$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$$

Sustituyo $t_n = x^n$:

$$x^n - 5x^{n-1} + 8x^{n-2} - 4x^{n-3} = 0$$

Y al dividir entre x^{n-3}:

$$x^3 - 5x^2 + 8x - 4 = 0$$

Con raíces:

$$p(x) = (x-2)^2(x-2)$$

Entonces, t_n sería:

$$t_n = c_{11} \cdot 1^n + c_{21} 2^n + c_{22} n 2^n$$

Luego $t_n \in O(n2^n)$.

Ejemplo.

$$T(n) = 2T(n-1) - T(n-2)$$

Sustituimos $T(n) = x^n$:

$$x^n - 2x^{n-1} + x^{n-2} = 0$$
$$(x^2 - 2x + 1)x^{n-2} = 0$$

$$p(x) = x^2 - 2x + 1 = (x-1)^2$$

Luego:

$$t_n = c_{10} 1^n + c_{11} 1^n n$$

Con $c_{10}, c_{11} \in \mathbb{R}$.

1.2.2. Ecuciones no homogéneas

Trabajaremos con ecuaciones del estilo:

$$a_0T(n) + a_1T(n-1) + \cdots a_kT(n-k) = b_1^n p_1(n) + b_2^n p_2(x) + \cdots$$

Con b_i constantes y $p_i(n)$ es un polinomio de grado d_1.

Calculamos el polinomio con la fórmula:

$$p(x) = p_h(x) \prod_{i=1}^{s} (x - b_i)^{i+1}$$

Donde p_h es la ecuación característica de la ecuación homogénea.

De donde extraemos todas sus raíces: $r_1, \ldots, r_k$.

$$t_n = c_1\sqrt{2}^n + c_2(-\sqrt{2})^n + c_3 \cdot 1^n$$

Con c_1, c_2, c_3 constantes.

Ejemplo (Fibonacci)**.**

$$T(n) = 2T(n-1) + T(n-2) + 1$$

Sustituimos $T(n) = t_n$ y buscamos como expresar $1 = b^n p(n)$:

$$t_n - t_{n-1} - t_{n-2} = 1 = 1^n \cdot n^0$$

$$\left(x - \sqrt{2}\right)\left(x + \sqrt{2}\right)(x-1)^1$$

Luego:

$$t_n = c_1\sqrt{2}^n + c_2(-\sqrt{2})^n + c_3 \cdot 1^n$$

Con c_1, c_2, c_3 constantes, luego $T(n) \in O(\sqrt{2}^n)$.

Ejemplo.

$$T(n) = T(n-1) + n$$

Sustituimos:

$$t_n - t_{n-1} = n = 1^n \cdot n^1$$

$$p(x) = (x-1)(x-1)^2 = (x-1)^3$$

Luego:

$$t_n = c_1 1^n + c_2 n 1^n + c_3 n^2 1^n \Rightarrow T(n) = O(n^2)$$

Ejemplo.

$$T(n) - T(n-1) = n + 3^n$$

En este caso, nos podríamos quedar simplemente con la fórmula:

$$T(n) - T(n-1) = 3^n$$

Y el orden de eficiencia sería el mismo.

Lo resolvemos de ambas formas:

$$p(x) = (x-1)(x-3)$$
$$t_n = c_1 1^n + c_3 3^n \Rightarrow T(n) \in O(3^n)$$

Si decidimos mantener la n:

$$p(x) = (x-1)(x-1)^2(x-3) = (x-1)^3(x-3)$$
$$t_n = c_1 1^n + c_2 n 1^n + c_3 3^n \Rightarrow T(n) \in O(3^n)$$

1.2.3. Cambio de variable

Hay problemas que el método anterior no nos permite solucionar. Para ello, hacemos un cambio de variable. No olvidemos cambiarlo al final.

Ejemplo.

$$T(n) = 2T\left(\frac{n}{3}\right) + 1$$

Hacemos el cambio de variable: $n = 3^k$:

$$T(3^k) = 2T(3^{k-1}) + 1$$

$$t_k - 2t_{k-1} = 1$$
$$p(x) = (x-2)(x-1)$$

$$t_k = c_1 2^k + c_2 1^k$$

Con c_1, c_2 constantes

Luego:

$$\begin{aligned} t_n &= c_1 2^{\log_3(n)} + c_2 \\ &\overset{(*)}{=} c_1 n^{\log_3(2)} + c_2 \Rightarrow T(n) \in O\left(n^{\log_3 2}\right) \end{aligned}$$

Donde en $(*)$ he usado que:

$$a^{\log b} = b^{\log a}$$

Ejemplo.

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Haciendo el cambio: $n = 2^k$:

$$t_k - 2t_{k-1} = 2^k k^0$$
$$p(x) = (x-2)(x-2) = (x-2)^2$$
$$t_k = c_1 2^k + c_2 k 2^k$$

Deshago el cambio: $k = \log_2(n)$

$$t_n = c_1 n + c_2 n \log_2(n) \in O(n \log(n))$$

Ejemplo.

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Haciendo el cambio: $n = 2^k$:

$$t_k - 2t_{k-1} = (2^k)^2 k^0 = (2^2)^k = 4^k$$

$$p(x) = (x-2)(x-4)$$
$$t_k = c_1 2^k + c_2 k 4^k = c_1 2^k + c_2 (2^k)^2$$

Deshago el cambio: $k = \log_2(n)$

$$t_n = c_1 n + c_2 n^2 \in O(n^2)$$

Ejemplo.

$$T(n) = aT\left(\frac{n}{6}\right) + n^c$$

Con $a, b, c \in \mathbb{R}$. Se pide calcular la eficiencia del algoritmo en función de a, b, c.
Haciendo el cambio: $n = 6^k$:

$$t_k = at_{k-1} + (b^k)^c = at_{k-1} + (b^c)^k$$
$$t_k - at_{k-1} = (b^c)^k$$

Con polinomio:

$$p(x) = (x-a)(x-b^c)$$

- Si $a = b^c$:

$$p(x) = (x-b^c)^2 = (x-a)^2$$
$$t_k = c_1 a^k + c_2 k a^k = c_1 (b^c)^k + c_2 n (b^c)^k$$
$$t_n = c_1 n^c + c_2 n^c \log_b n \Rightarrow T(n) \in O\left(n^c \log_b(n)\right)$$

- Si $a \neq b^c$:

$$p(x) = (x-b^c)(x-a)$$
$$t_k = c_1 (b^c)^k + c_2 a^k$$
$$t_n = c_1 n^c + c_2 a^{\log_b(n)} \Rightarrow c_1 n^c + c_2 n^{\log_b(a)}$$

  - Si $a > b^c$: $T(n) \in O\left(n^{\log_b a}\right)$
  - Si $a < b^c$: $T(n) \in O(n^c)$

1.2.4. Cambio de recorrido o rango

Hay casos en los que los coeficientes no son constantes:

$$T(n) = nT(n-1)$$
$$T(n) = T^2(n-1)$$
$$T(n) = nT\left(\frac{n}{2}\right)$$

Ejemplo.

$$T(n) = T^2(n-1)$$

Sea $V(n) = \log_2 T(n)$:

$$\log_2(T(n)) = \log_2\left(T^2(n-1)\right) = 2\log_2 T(n-1)$$
$$V(n) = 2V(n-1)$$
$$V_n = 2V_{n-1}$$
$$t_n - 2t_{n-1} = 0$$
$$p(x) = (x-2)$$
$$v_n = c_1 2^n$$
$$t_n = 2^{v_n} = 2^{(c_1 2^n)} = 4^n$$

Teorema 1.3 (Teorema Maestro)**.** *Dada una función que mide el tiempo de ejecución de un algoritmo del estilo:*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Entonces:

- *Si* $\exists \varepsilon > 0 \mid f(n) \in O\left(n^{\log_b(a-\varepsilon)}\right) \Rightarrow T(n) \in \Theta\left(n^{\log_b(a)}\right)$
- *Si* $\exists k \geqslant 0 \mid f(n) \in \Theta\left(n^{\log_b a} \log^k(n)\right) \Rightarrow T(n) \in \Theta\left(n^{\log_b(a)} \log^{k+1}(n)\right)$
- *Si* $\left\{ \begin{array}{c} \exists \varepsilon > 0 \mid f(n) \in \Omega\left(n^{\log_b(a+\varepsilon)}\right) \\ \wedge \\ af\left(\frac{n}{b}\right) \leqslant cf(n) \ \ con\ c < 1 \end{array} \right\} \Rightarrow T(n) \in \Theta(f(n))$

Ejercicio. Se pide determinar el orden de eficiencia de los siguientes algoritmos, conociendo las funciones que nos dan sus tiempos a partir de n:

1. $T(n) = 4T(n-1) - 4T(n-2)$
2. $T(n) = 3T(n-1) - 3T(n-2) + T(n-3)$
3. $T(n) = 2T(n-1) + n + n^2$
4. $T(n) = 5T(n-1) - 6T(n-2) + 4 \cdot 3^n$
5. $T(n) = 2T\left(\frac{n}{2}\right) + \log_2(n)$
6. $T(n) = T\left(\frac{n}{2}\right) \cdot T^2\left(\frac{n}{4}\right)$
7. $T(n) = aT\left(\frac{n}{b}\right) + n^k$

2. Algoritmos Divide y Vencerás

Dado un problema P, lo dividimos en subproblemas que han de ser del mismo tipo: $P = \{P_i\}_{i \in I}$. Encontramos las soluciones de todos estos problemas: $S_i \mid i \in I$ y con estas, las juntamos de forma adecuada para obtener la solución al problema P: $S = \{S_i\}_{i \in I}$. Notemos que podemos hacerlo de forma recursiva, podemos aplicar la técnica de divide y vencerás a los subproblemas. Debemos fijar un caso base, donde el problema sea suficientemente pequeño como para resolverlo.

Ejemplos de algoritmos que implementan la búsqueda binaria son Quicksort, Mergesort, Búsqueda Binaria, ...

Problemas como el de las Torres de Hanoi pueden resolverse mediante esta técnica de resolución de problemas.

Tratamos cuestiones importantes:

- Cómo descomponer un problema en subproblemas.
- Cómo resolver subproblemas.
- Cómo combinar las soluciones.
- ¿Mereció la pena aplicar esta técnica?

Para responder al último punto observemos el siguiente ejemplo, que trata un problema de forma genérica.

Ejemplo. Supongamos un problema P de tamaño n que sabemos que puede resolverse con un algoritmo (básico) A, con:

$$t_A(n) \leqslant cn^2$$

Dividimos P en 3 subproblemas de tamaño $\frac{n}{2}$, isneod cada uno de ellos del mismo tipo que A y consumiendo un tiempo lineal la combinación de sus soluciones: $t(n) \leqslant dn$. Tenemos, por tanto, un nuevo algoritmo B, Divide y Vencerás, que consumirá un tiempo de:

$$t_B(n) = 3t_A\left(\frac{n}{2}\right) + t(n) \leqslant 3t_A\left(\frac{n}{2}\right) + dn \leqslant \left(\frac{3c}{4}\right)n^2 + dn$$

B tiene un tiempo de ejecución mejor que el algoritmo A, ya que reduce la constante oculta.

Si volvemos a reducir el tamaño del subproblema, obtenemos un nuevo algoritmo C, que tendría un tiempo:

$$t_c(n) = \begin{cases} t_A(n) & \text{si } n \leqslant n_0 \\ 3t_C\left(\frac{n}{2}\right) + t(n) & \text{si } n > n_0 \end{cases}$$

C es mejor en eficiencia que los algoritmos A y B ($t_c(n) \leqslant bn^{1,58}$).

Al valor n_0 se le denomina umbral y al algoritmo que se ejecuta debajo de este umbral, `ad hoc` son fundamentales para que la técnica funcione bien.

Requisitos lógicos para usar Divide y Vencerás:

- El problema se tiene que poder reducir a subproblemas del mismo tipo.
- Los subproblemas han de poder resolverse de manera independiente.
- Las soluciones de los subproblemas no deben afectar entre sí, no debe existir solapamiento entre subproblemas.
- Ha de poderse juntar las soluciones de los subproblemas para resolver el problema.

Una forma matemática de entender el punto 3 es decir que los subproblemas deben formar una partición del problema.

Para que la técnica sea eficiente:

- Selección cuidadosa de cuando usar el algoritmo `ad hoc`.
- El número de subproblemas debe ser razonablemente pequeño (para no reducir mucho la eficiencia).
- Los subproblemas deben tener el menor tamaño posible.

Normalmente, al aplicar Divide y Vencerás nos encontraremos con eficiencias del estilo:

$$T(n) = \begin{cases} t(n) & \text{si } n \leqslant n_0 \\ IT\left(\frac{n}{b}\right) + G(n) & \text{si } n > n_0 \end{cases}$$

Donde I es el número de subproblemas, $\frac{n}{b}$ el tamaño de estos y:

$$G(n) = D(n) + C(n)$$

Con $D(n)$ el tiempo de dividir el problema en subproblemas y $C(n)$ el tiempo de combinación. $t(n)$ era el tiempo del algoritmo `ad hoc`.

Ejemplo. Dado un vector de elementos, determinar la posición que ocupa el máximo del mismo.

Proponemos como solución el siguiente código:

```
int Maximo(int* v, int inf, int sup){
    if(sup == inf) return v[inf];
    else{
        int med = (inf+sup)/2;
        int izda = Maximo(v, inf, med);
        int dcha = Maximo(v, med+1, sup);

        return max(izda, dcha);
    }
}
```

Con un tiempo de ejecución de:

$$T(n) = 2T\left(\frac{n}{2}\right) + 1 \in O(n)$$

Otra solución sería:

```
int Maximo2(int* v, int inf, int sup){
    if(sup == inf) return v[inf];
    else{
        int izda = Maximo2(v, inf, sup);

        return max(izda, sup);
    }
}
```

En este caso:

$$T(n) = T(n-1) + 1 \in O(n)$$

Los dos del mismo orden, pero encontramos un problema entre uno y otro: el número de llamadas recursivas:

- `Maximo2` Realiza n llamadas recursivas, pero el tamaño de la pila ocupado es de n.
- `Maximo` Realiza n llamadas recursivas, pero el tamaño de la pila es $\log n$.

El número de los elementos de la pila de `Maximo` puede razonarse dibujándolo con un árbol.

Por tanto, el primer algoritmo que planteamos era mejor ya que usa menos pila (evitando su desbordamiento).

Observación. Notemos que este ejemplo ha sido un caso instructivo, es más eficiente resolverlo de la forma canónica que aplicando Divide y Vencerás.

Ejercicio. Un camión va desde Granada a Moscú siguiendo una ruta determinada. La capacidad del tanque de combustible es C y conocemos el consumo por kilómetros del camión. Conocemos las gasolineras que se encuentran en la ruta y la distantica entre ellas. Se pide minimizar el número de paradas que hace el conductor.

2.1. Ejemplos de Divide y Vencerás

2.1.1. Multiplicación de Enteros Grandes

Comprobar si un número es primo requiere muchas multiplicaciones de enteros grandes (desde dos millones de dígitos). Útil en criptografía. Para resolver este problema, debemos implementar algoritmos efiientes capaces de trabajar con estos valores. Nos encontramos con:

- Método clásico (el que aprendimos en infantil).

- Método basado en Divide y Vencerás.

Método clásico

Multiplicar dos dígitos de n entre sí por el método clásico tiene un costo de ($n \cdot n = n^2$ multiplicaciones más n desplazamientos más $n \cdot 2n$ sumas) $n^2 + n + 2n^2$. Tratamos de reducir su costo mediante Divide y Vencerás.

Divide y Vencerás sencillo

Aplicando Divide y Vencerás, obtenemos el siguiente algoritmo:

Dados dos números $X = 12345678$ e $Y = 24680135$, consideramos:

$$x_i = 1234 \qquad x_d = 5678$$
$$X = x_i \cdot 10^4 + x_d$$

$$y_i = 2468 \qquad y_d = 0135$$
$$Y = y_i \cdot 10^4 + y_d$$

Y combinamos:

$$\begin{aligned} X \cdot Y &= (x_i \cdot 10^4 + x_d)(y_i \cdot 10^4 + y_d) \\ &= x_i \cdot y_i \cdot 10^8 + (x_i \cdot y_d + x_d \cdot y_i) \cdot 10^4 + x_d \cdot y_d \end{aligned}$$

Que, de forma general:

$$\begin{aligned} X &= x_i \cdot 10^{n/2} + x_d \\ Y &= y_i \cdot 10^{n/2} + y_d \\ X \cdot Y &= (x_i \cdot y) \cdot 10^n + (x_i \cdot y_d + x_d \cdot y_i) \cdot 10^{n/2} + x_d \cdot y_d \end{aligned}$$

De eficiencia:

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \in O\left(n^{\log_2 4}\right) = O\left(n^2\right)$$

Este lo vemos en el siguiente pseudocódigo:

```
Funcion DVbasico(X, Y, n){
    if P chico return X * Y;
    else{
        // Dividir
        Obtener xi, xd, yi, yd;

        z1 = DVbasico(xi, yi, n/2);
        z2 = DVbasico(xi, yd, n/2);
        z3 = DVbasico(xd, yi, n/2);
        z4 = DVbasico(xd, yd, n/2);

        // Combinar
        aux = Sumar(z2, z3);
        z1 = DesplazarDcha(z1, n);
        aux = DesplazarDcha(aux, n/2);
        z = Sumar(z1, aux, z4);

        return z;
    }
}
```

Sin embargo, podemos seguir mejorando el algoritmo.

Divide y Vencerás mejorado

Observemos que el cuello de botella del algoritmo anterior está en el número de multiplicaciones, que es de tamaño $n/2$. Por tanto, para mejorar la eficiencia, necesitamos reducir el número de multiplicaciones que hacemos. Podemos plantear el siguiente algoritmo, donde dados dos números X e Y y divididos entre x_i, x_d, y_i e y_d; definimos:

$$r = (x_i + x_d) \cdot (y_i + y_d) = (x_i \cdot y_i) + (x_i \cdot y_d + x_d \cdot y_i) + x_d \cdot y_d$$
$$p = x_i \cdot y_i$$
$$q = x_d \cdot y_d$$

Luego se tiene que:

$$x_i \cdot y_d + x_d \cdot y_i = r - p - q$$

Y podemos calcular que:

$$X \cdot Y = p \cdot 10^n + (r - p - q) \cdot 10^{n/2} + q$$

Donde tenemos una multiplicación de tamaño n. Podemos verlo en pseudocódigo:

```
FuncionDV (X, Y, n){
    if P chico return X * Y;
    else{
        // Dividir
        Obtener xi, xd, yi yd;

        s1 = Sumar(xi, xd);
        s2 = Sumar(yi, yd);
        p = DV (xi, yi, n/2);
        q = DV (xd, yd, n/2);
        r = DV (s1, s2, n/2);

        // Combinar
        aux = Sumar(r, -p, -q);
        p = DesplazarDcha(p, n);
        aux = DesplazarDcha(aux, n/2);
        z = Sumar(p, aux, q);

        return z;
    }
}
```

De eficiencia:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) \in O\left(n^{\log_2 3}\right) \approx O(n^{1,585})$$

2.1.2. Búsqueda binaria

La búsqueda binaria es un algoritmo de búsqueda de un elemento en un vector *ordenado* de tamaño n. Esta consiste en coger el elemento del medio del vector (supongamos que está en la posición k) y comprobar si es o no el elemento deseado. Si lo es, hemos terminado; si no lo es:

- Si el elemento del medio es menor que el deseado, repetimos el procedimiento con el subvector que va desde $k+1$ hasta n.
- Si el elemento del medio es mayor que el deseado, repetimos el procedimiento con el subvector que va desde 0 hasta $k-1$.

Se trata de un algoritmo de eficiencia $O(\log n)$.

Ejercicio. Hágase la demostración del orden de eficiencia de la búsqueda binaria.

2.1.3. Multiplicación de matrices

Dadas dos matrices $A = (a_{ij})_{ij}$ y $B = (b_{ij})_{ij}$ del mismo tamaño, $n \times m$, tratamos de multiplicar A por B para obtener una nueva matriz $C = (c_{ij})_{ij}$. La multiplicación

de matrices se define como:

$$c_{ij} = \sum_{k=1}^{n} (a_{ik} \cdot b_{kj})$$

Para todo índice (i, j) de cada elemento de la matriz C de tamaño $n \times m$.

Implementando la fórmula, se puede ver claramente que es de orden (considerando $n = m$) $O(n^3)$. Para aplicar Divide y Vencerás, vamos a proceder como con la multiplicación de enteros:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = C = A \cdot B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae + bf & ag + bh \\ ce + df & cg + dh \end{pmatrix}$$

Esta fórmula también es cierta cuando $a, \ldots, h$ son matrices. Por tanto, mediante esta fórmula podemos dividir el producto de dos matrices $n \times n$ en 8 subproblemas de tamaños $n/2$. Por lo que tendríamos una eficiencia de tamaño (recordemos que la suma de matrices es de orden n^2):

$$T(n) = 8T\left(\frac{n}{2}\right) + cn^2 \in O\left(n^{\log_2 8}\right) = O(n^3)$$

Algoritmo de Strassen

Dado el problema anterior:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = C = A \cdot B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae + bf & ag + bh \\ ce + df & cg + dh \end{pmatrix}$$

Ahora, definimos:

$$\begin{aligned} P &= (a + d)(e + h) \\ Q &= (c + d)e \\ R &= a(g - h) \\ S &= d(f - e) \\ T &= (a + b)h \\ U &= (c - a)(e + g) \\ V &= (b - d)(f + h) \end{aligned}$$

De esta forma se tiene que:

$$\begin{aligned} r &= P + S - T + V \\ s &= R + T \\ t &= Q + S \\ u &= P + R - Q + U \end{aligned}$$

Y sólo se necesitan 7 multiplicaciones y 18 sumas y restas, en lugar de 8 multiplicaciones y 4 sumas.

La eficiencia de este algoritmo es de:

$$T(n) = 7T\left(\frac{n}{2}\right) + cn^2 \in O\left(n^{\log_2 7}\right) \approx O(n^{2,81})$$

2.2. Umbrales

Por el método de Divide y Vencerás es natural que queramos ejecutar algoritmos de mayor orden de eficiencia para tamaños de problema pequeños (porque rinden mejor) y que para el resto de tamaños se ejecute un algoritmo de mayor eficiencia asintótica. Este tamaño límite (el último tamaño en el que se ejecuta el algoritmo de orden de eficiencia mayor) recibe el nombre de umbral. De esta forma, el algoritmo que se ejecuta para tamaños del problema (n) menores que el umbral ($n \leqslant n_0$), se denomina algoritmo `ad hoc`.

Lo más difícil relativo al umbral es su elección: es difícil hablar del umbral n_0 si no tratamos con implementaciones, ya que gracias a ellas conocemos las constantes ocultas que nos permiten afinar el cálculo de dicho valor. No hay restricciones sobre el tamaño que debe tener el umbral: puede que el mejor umbral sea tal que siempre se ejecute el algoritmo `ad hoc`; o puede que el umbral sea tal que siempre se ejecute el algoritmo de mejor orden asintótico.

2.2.1. Selección de umbrales

Podemos encontrar el umbral de un algoritmo en el que vamos a usar Divide y Vencerś y el `ad hoc` de distintos métodos:

Método empírico

Implementamos ambos algoritmos, el básico (o `ad hoc`) y el Divide y Vencerás. Ejecutamos el problema para varios valores de n, obteniendo las gráficas de los dos algoritmos. Fijarnos en el valor de la abscisa del punto de corte de las gráficas y dicho valor será el umbral. Posteriormente, eligiremos obviamente a cada lado del umbral los algoritmos de menor tiempo de ejecución.

Método teórico

El método es similar: obtenemos las expresiones de los tiempos de ambos algoritmos (sin despreciar constantes) y analizamos las gráficas, buscando algún punto de corte que será el umbral.

Método híbrido

Calculamos las constantes ocultas mediante el enfoque empírico. Calculamos el umbral igualando teóricamente ambas gráficas. Tenemos ya candidado a umbral. Finalmente, probamos valores alrededor de dicho umbral, para determinar el umbral óptimo.

2.3. Algoritmos de ordenación

A la hora de clasificar los algoritmos de ordenación, encontramos:

1. Algoritmos lentos (de orden $\Theta(n^2)$ y ordenación por cambio):
   a) Burbuja.

b) Inserción.

c) Selección.

Se trata e algoritmos más sencillos cuyo comportamiento es malo para tamaños muy grandes.

2. Algoritmos rápidos (de orden $\Theta(n \log n)$):

   a) Heapsort.

   b) Mergesort.

   c) Shellsort.

   d) Quicksort.

   Son algoritmos más complejos que se coomportan bien cuando el tamaño del problema es grande.

2.3.1. Burbuja

Burbuja (*Bubble sort*) es un algoritmo que trata de ordenar los elementos de un vector de la siguiente forma: compara los dos primeros elementos del vector.

- Si están ordenados, pasa a la siguiente pareja de elementos del vector (formada por el segundo y tercer elemento).
- Si no están ordenados, los intercambia y pasa a la siguiente pareja de elementos del vector (formada por el segundo y tercer elemento).

Se repite este procedimiento hasta llegar a los dos últimos elementos del vector. Una vez realizado este procedimiento, volvemos a repetirlo sobre el vector, hasta que este esté ordenado (sabremos que el vector está ordenado cuando al recorrerlo mediante este procedimiento, no se realice ningún intercambio).

```
void bubbleSort(int* v, int N){
    int temp;
    bool ordenado = false;

    int i = 1;
    while(!ordenado && i < N){
        ordenado = true;

        for(int j = 0; j < N - i; j++){
            if(v[j] > v[j+1]){
                temp = v[j];
                v[j] = v[j+1];
                v[j+1] = temp;

                ordenado = false;
            }
        }
```

```
            i++;
        }
    }
```

2.3.2. Inserción

Inserción (*Insertion sort*) es un algoritmo de ordenación que trata de ordenar los elementos de un vector de una forma natural para el ser humano. El procedimiento es el siguiente. Se toma un elemento del vector (por ejemplo, el primero), al ser sólo un elemento, se trata de un conjunto ordenado. Continuamos cogiendo otro elemento del vector (el segundo):

- Si es mayor que el primer elemento, lo colocamos detrás.
- Si es menor que el primer elemento, lo colocamos delante de este.

Continuamos con el siguiente elemento del vector (el tercero):

- Si es mayor que el segundo, lo colocamos detrás.
- Si es menor que el segundo y mayor que el primero, lo colocamos en medio.
- Si es menor que ambos, lo colocamos al inicio.

Repitiendo los pasos, consiste en construir un subconjunto del vector de tamaño k de elementos ordenados, de forma que cogemos un nuevo elemento del vector y tratamos de colocarlo en el conjunto de tamaño k, obteniendo un subconjunto ordenado de tamaño $k+1$ y repetir el proceso hasta llegar a n.

```
void insertionSort(int* v, int N){
    int i, j, temp;

    for(i = 1; i < N; i++){
        temp = v[i];
        j = i - 1;

        while(j >= 0 && temp <= v[j]){
            v[j+1] = v[j];
            j--;
        }

        v[j+1] = temp;
    }
}
```

2.3.3. Selección

Selección (*Selection sort*) es otro algoritmo de ordenación que junto con Inserción constituyen los dos algoritmos más intuitivos (más naturales) del ser humano. Dado

un vector de n elementos, buscamos el mínimo del vector. Posteriormente, colocamos el mínimo en la posición inicial y, dado un vector de $n-1$ elementos restantes (todos salvo el primero), buscmos el mínimo y lo colocamos detrás del mínimo anterior. Repetimos el proceso $n-1$ veces y obtenemos un vector ordenado.

```
void selectionSort(int* v, int N){
    int i, j, pos_min, temp;

    for(i = 0; i < N - 1; i++){
        // buscamos el mínimo
        pos_min = i;
        for(j = i + 1; j < N; j++){
            if(v[j] < v[pos_min]){
                pos_min = j;
            }
        }

        // intercambiamos con el primero
        temp = v[pos_min];
        v[pos_min] = v[i];
        v[i] = temp;
    }
}
```

2.3.4. Mergesort

O también llamado ordenación por mezcla, consiste en dado un vector de n componentes, dividirlo en dos vectores de $n/2$, de forma que ordenamos los dos subvectores y combinamos ambos en un único vector (el vector ya ordenado), sabiendo que los dos subvectores ya se encuentran ordenados (cosa que sabemos hacer). Este proceso puede hacerse de forma recursiva, aplicándolo varias veces sobre el mismo vector.

Podemos encontrar variaciones de este algoritmo, como no dividir por $n/2$, sino por cualquier otro tamaño dependiente de n. Por ejemplo, n/k con k entero.

2.4. Ejercicios

2.4.1. Mediana de un vector

Ejercicio. ¿Cómo calculamos la mediana de un vector algorítmicamente?

2.4.2. Problema de los puntos más cercanos

Ejercicio. Tenemos un vector con una secuencia de valores y queremos saber los dos valores más cercanos.

1. Una primera aproximación es calcular para cada valor del vector las distancias y buscar la mínima. Eficiencia $O(n^2)$.

2. Otra solución es ordenar los datos y ahora recorrer linealmente el vector para buscar la distancia mínima. Eficiencia $O(n \log n)$.

3. Mediante la técnica divide y vencerás sobre el vector del primer punto no vemos cómo resolverlo.

4. Si aplicamos divide y vencerás sobre el vector ordenado, sí que podemos resolver el problema (elegimos o la menor distancia a la izquierda del corte, o la menor a al derecha, o justo los que separamos a cortar). Tenemos un coste de $T(n) = 2T(n/2) + 1$. Otra vez eficiencia $O(n \log n)$.

Ejercicio. Tenemos un conjunto de puntos repartidos a lo largo de un plano y queremos saber cuales son los dos puntos que equidistan entre sí menos que cualquier dos otros.

1. Una primera aproximación es calcular para cada valor del vector de puntos las distancias y buscar la mínima. Eficiencia $O(n^2)$.

2. Otra aproximación es utilizando divide y vencerás: Buscamos una forma de partir los puntos en subproblemas. Nos damos cuenta de que si partimos por la mitad del plano obtenemos una distribución no muy buena de los puntos. Para una mejor aproximación, tratamos de dejar todos los puntos con ordenada menor que el valor medio de las ordenadas de todos los puntos y el resto a otro lado. De esta forma acabamos de dividir el problema en dos subproblemas del mismo tamaño.

   Una vez resueltos los dos subproblemas, nos queda a un lado una distancia mínima de a y al otro una distancia mínima de b. Lo que hacemos es coger $c = \text{mín}\{a, b\}$ y considerar una banda que definimos como: coger de los puntos de la derecha el punto más a la izquierda y poner una tolerancia de c a la izquierda y del punto de la izquierda más a la derecha, poner una tolerancia de c a la derecha.

   a) A su vez, dentro de esta banda podemos compararlos todos con todos. Orden de $O(n^2)$.

   b) También, podemos ordenarlos en la otra coordenada y establecer otra tolerancia, una especie de bola donde estudiar qué puntos son los más cercanos. Orden de $O(n \log^2 n)$. Estos dos órdenes de eficiencia no son satisfactorios, buscamos uno mejor.

3. Otra aproximación de Divide y Vencerás es:

   a) Ordenar por las dos coordenadas todos los puntos en dos vectores.

   b) Separar por una coordenada (como x) ambos conjuntos, obteniendo directamente L_x y R_x (izquierda según x y derecha según x) y en orden lineal L_y y R_y: que son respectivamente los puntos en L_x y R_x ordenados por coordenada y.

Ejercicio. Dada una secuencia de números, definimos pico al valor i tal que $x[i-1] < x[i] > x[i+1]$ y valle al valor j tal que $x[j-1] > x[j] < x[j+1]$. Un pico i es consecutivo a un valle j si ningún valor entre i y j es pico o valle.

1. Dado un vector de enteros que primero es estrictamente creciente y luego estrictamente decreciente, encontrar el máximo del vector.

   Para ello, planteamos cortar por el del centro del vector y ver a sus vecinos: si el vecino de la derecha, aplicamos otra vez el algoritmo de la derecha (y si es por la izquierda, aplicarlo a la izquierda); hasta tener al menos 3 elementos.

3. Algoritmos Greedy

Los algoritmos Greedy son algoritmos que siempre toman la mejor decisión en cada momento, la cual sólo depende de datos locales. Por ejemplo, un algoritmo que busque recorrer n ciudades en el plano podría implementarse mediante la técnica Greedy cogiendo en cada paso la ciudad que se encuentre más cerca de la que estamos. Notemos que de esta forma no obtenemos la solución óptima al problema, aún habiendo tomado la solución óptima en cada caso.

Los algoritmos Greedy tienen como características:

- Se suelen utilizar para resolver problemas de optimización: búsqueda de máximo o mínimo.
- Toman decisiones en función de la información local de cada momento, tomando la mejor alternativa.
- Una vez tomada la decisión, no se replantea en el futuro.
- Son fáciles y rápidos de implementar.
- Los algoritmos Greedy no siempre proporcionan la solución óptima, como ya hemos podido comprobar.

Elementos de un algoritmo voraz:

- Conjunto de candidatos: representa al conjunto de posibles soluciones que se pueden tomar en cualquier momento.
- Conjunto de seleccionadas: representa al conjunto de decisiones tomadas hasta el momento.
- Función solución: determina si se ha alcanzado una solución al problema.
- Función de selección: determina el candidato que proporciona el mejor paso.
- Función de factibilidad: determina si es posible o no llegar a la solución del problema mediante el conjunto de seleccionados.
- Función objetivo: da el valor de la solución alcanzada.

Un algoritmo Greedy puede resolverse de la forma:

- Se parte de un conjunto de candidatos a solución vacío.
- De la lista de candidatos que hemos podido identificar, con la función de selección, se coge el mejor candidato posible.

- Vemos si con ese elemento podríamos llegar a constituir una solución, es decir, si se verifican las condiciones de factibilidad en el conjunto de candidatos.
- Si el candidato anterior no es válido, lo borramos de la lista de candidatos y nunca más se considera.
- Evaluamos la función solución (si no hemos terminado, seleccionamos otro candidato con la función de selección y repetimos el proceso hasta alcanzar una solución).

Normalmente, si conseguimos un algoritmo Greedy óptimo, es probable que sea el mejor algoritmo que resuelve el problema.

Problema de dar cambio

Normalmente, las máquinas que dan cambio están programadas de forma que nos lo devuelven con el mínimo número posible de monedas.

En este caso, el conjunto de candidatos es el conjunto de monedas que tiene disponible la máquina. El conjunto de seleccionados son las monedas que voy incorporando a la solución. Habremos encontrado una solución cuando hayamos llegado a la cifra que se pedía. Si superamos o no llegamos a la cantidad a devolver, nuestra solución no es factible.

Como criterio de selección inicial, podemos elegir la moneda de mayor valor que sea menor o igual que el precio a pagar.

¿Este algoritmo consigue la solución óptima?
Pues si el valor de las monedas corresponde con los billetes y monedas de euro en circulación sí; pero si tenemos otro sistema monetario, quizás no.

Problema de selección de programas

Tenemos un conjunto T de n programas, cada uno con tamaño $t_1, \ldots, t_n$ y un dispositivo de capacidad máxima C, seleccionar el mayor número de programas que pueden meterse en C.

3.1. Heurística Greedy

Hay casos en los que no podemos alcanzar la solución óptima por Greedy, pero nos es útil porque nos da una solución aproximada, como puntos de entrada para otros algoritmos. Los algoritmos Greedy suelen ser eficientes.

Cuando esto sucede, hablamos de *Heurísticas Greedy*. Problemas donde puede aplicarse una heurística Greedy son:

- Problema de coloreo de un grafo.
- Problema de Viajante de Comercio.
- Problema de la Mochila.

3.1.1. Problema de coloreo de un grafo

Dado un grafo, determinar el mínimo número de colores necesarios para colorear todos sus vértices y que no haya dos adyacentes pintados con el mismo color.

Si el grafo no es plano puede llegar a necesitar tantos colores como nodos. Si es plano, vamos a poder pintarlo con 4 caminos.

Problema de cruce de semáforos

Dado un cruce con caminos que pueden ser de sentido único o no, queremos regularlo con semáforos de forma que podamos agilizar al máximo el tráfico. Puede verse como un problema de coloreo de grafo:

- Como vértices del grafo, todos los posibles turnos (desde esta carretera hasta esta).
- Vamos a conectar los vértices cuando dos turnos no sean compatibles.

4. Backtracking y Branch & Bound

Si tenemos un problema que podemos resolver con un conjunto de decisiones, podemos usar *backtracking*.

La solución al problema la representarmos por una $n-$upla, de forma que cada decisión se toma de un conjunto finito de candidatos a seleccionar. Aquí sí que podemos volver atrás tras una decisión, no como en la técnica Greedy. Además, las mejores decisiones a tomar no tenemos por qué tomarlas al inicio. El orden de las decisiones tomadas puede importar o no.

Cuando veamos que una solución es inviable, no seguimos por ella, sino que retrocedemos hasta cierto punto y proseguimos por otras posibles soluciones. Una vez alcanzada una solución, puede que queramos encontrar una mejor. En dicho caso continuamos buscando otra solución. Desde este punto, podemos pensar en los algoritmos *backtracking* de forma similar a los fuerza bruta, pero de forma inteligente.

Branch & Bound es una modificación de este método: Este método busca una solución como en el método de backtracking, pero cada solución tiene asociado un costo y la solución que se busca es una de mínimo costo llamada óptima. Además de ramificar una solución padre (branch) en hijos se trata de eliminar de consideración aquellos hijos cuyos descendientes tienen un costo que supera al óptimo buscado acotando el costo de los descendientes del hijo (bound). La forma de acotar es un arte que depende de cada problema. La acotación reduce el tiempo de búsqueda de la solución óptima al "podar" (pruning) los subarboles de descendientes costosos.

Backtracking busca agilizar algoritmos de un orden grande de forma inteligente.

Coloreo de grafos

Queremos colorear un grafo plano (o mapa) con no más de 4 colores, de forma que nodos (o países) adyacentes tengan distinto color. El problema a resolver depende del criterio a optimizar (si un color es más caro, si queremos usar el mínimo número de colores, ...).

Problema de la mochila

N paquetes de datos a transmitir en un tiempo t_i y con ganancia g_i. Seleccionar el conjujnto de paquetes a transmitir en un tiempo T maximizando la ganancia.

Problema del laberinto

Dado un laberinto, encontrar el camino desde la entrada hasta la salida (puede que queramos optimizar la longitud del camino).

Buscaminas

En la resolución del juego "buscaminas", hay casos donde no podemos asegurar por ciertos algoritmos la existencia de una bomba o no. Puede ser de utilidad emplear algoritmos *backtracking*: suponer que hay una bomba y a partir de ahí decidir si se puede llegar a una solución sin llegar a contradicciones. De otra forma, hay una bomba en dicho sitio.

Esto mismo lo podemos aplicar a resolver sudokus.

4.0.1. Diferencias Backtracking B&B frente a fuerza bruta

La principal diferencia es el uso de funciones de acotación o poda, de forma que si no podemos alcanzar una decisión por alguna rama, cortamos dicha rama (no seguimos por dicha solución, sino que retrocedemos y tomamos decisiones en base a ello).

Generar todas las combinaciones de n bits

Por ejemplo, para 2 bits sería `00`, `01`, `10`, `11`.
Una solución puede ser:

```
void completa_binario(vector<int>& v, int pos){
   if(pos == v.size()){
      imprimeVector(v);
   }else{
      v[pos] = 0;
      completa_binario(v, pos+1);
      v[pos] = 1;
      completa_binario(v, pos+1);
   }
}
```

Del orden $O(2^n)$. Intuitivamente, nos estamos moviendo por un árbol de estados.

Coloreo de grafos

```
void completa_grafo(vector<int>& v, int pos){
   if(pos == v.size()){
      comprueba_factibilidad(v)
   }else{
      v[pos] = 0;
      completa_grafo(x, pos+1);
      v[pos] = 1;
      completa_grafo(x, pos+1);
```

```
            v[pos] = 2;
            completa_grafo(x, pos+1);
            v[pos] = 3;
            completa_grafo(x, pos+1);
        }
    }
```

Del orden $O(4^n)$. La función de factibilidad podría hacer, por ejemplo, que si tenemos dos valores repetidos seguidos como `011`, entonces la solución no es factible, no seguimos por esa rama.

Para mejorar la solución, podemos reducir el espacio de decisiones, lo que agiliza el programa.

Viajante de comercio

```
void completa_kario(vector<int>& v, int pos, int k){
    if(pos == v.size()){
        procesa_vector(v)
    }else{
        for(int j = 0; j < k; j++){
            x[pos] = j;
            completa_kario(v, pos+1, j);
        }
    }
}
```

En todos estos problemas vemos que se establece una estructura de árbol imaginario sobre el conjunto de posibles soluciones. La forma en la que se generan las soluciones es equivalente a realizar un recorrido preorden del árbol. Sólo se procesan las hojas que corresponden con soluciones completas.

Definiciones

Solución parcial. Tupla o vector al que no se le han asignado todos sus componentes.

Función de poda. Evalúa si una solución parcial es viable o no (por no satisfacer las restricciones, porque tenemos una solución mejor o porque no satisface otras restricciones).

Restricciones explícitas. Reglas que restringe las soluciones a un subconjunto de todas las combinaciones.

Restricciones implícitas. Reglas que nos dicen cuándo una solución parcial nos puede llevar a la solución objetivo. Relacionan una decisión a tomar con las anteriores.

Árbol de estados. Árbol imaginario formado.

Estado del problema. Cada uno de los nodos del árbol.

Estado solución. Nodos del árbol que representan una solución al problema.

Estado respuesta. Una solución al problema que satisface restricciones implícitas.

Nodo vivo. Nodo que ya ha sido generado pero del que aún no se han generado todos sus descendientes.

Nodo muerto. Nodo que ha sido generado, y o bien se ha podado o se han generado todos los descendientes.

e-nodo o nodo en expansión. Nodo vivo del que actualmente se están generando los descendientes.

Problema de la suma de subconjuntos

Dados n números positivos y uno más, M, encontrar todos los subconjuntos cuya suma valga M.

Podemos dar la solución con una tupla de longitud variable (que contenga los índices de los elementos que forman parte de la solución) o con una tupla de longitud fija n (que contenga por cada elemento si se selecciona o no).

Problema de las n reinas

Dado un tablero de ajedrez, cómo colocar n reinas (supondremos 8) en el tablero de forma que ninguna pueda atacar a otra (no puede haber dos reinas en la misma fila, columna o diagonal).

- El vector solución puede ser un vector de 8 posiciones de números a elegir (cada uno representa una posición del tablero) entre 1 y 64. Tenemos 64^8 posibilidades.
- Como cada reina sólo puede estar en una fila, podemos tener 8 vectores de longitud 8 donde en cada uno indicamos la posición de la reina de dicha fila. Son 8^8 posibilidades.
- También sabemos que no pueden estar en la misma columna. Reducimos a 8! posibilidades.

4.0.2. Diferencias entre Backtracking y Branch & Bound

Ambos métodos recorren el árbol de estados y ámbos métodos utilizan funciones de poda para eliminar ramas que no conducen a soluciones.

- Backtracking: cuando el nuevo hijo del enodo ha sido generado, este se combierte en enodo hasta explorar todos sus descendientes.
- Branch & Bound: El enodo continúa siéndolo hasta que se han generado todos sus descendientes o se poda. De todos los nodos vivos, se escoge uno (el primero, el mejor, ...).

Si en un problema Greedy da el óptimo, Branch & Bound también lo hará.

5. Programación Dinámica

- Para problemas en los que necesitamos estados anteriores (en fibonacci, para calcular fibonacci(n) necesitamos tener fibonacci($n-1$) y fibonacci($n-2$), y para fibonacci($n-1$) necesitamos, ...).
- En el camino aparecen requisitos que se repiten (necesitamos calcular varias veces fibonacci(k)). En vez de calcularlo todas las veces, calcularlo una sola vez. Para calcular fibonacci(6) necesitamos 5 veces fibonacci(2).
- Antes de calcular el subproblema, mira si lo tienes ya resuelto (si lo tiene, lo usa y si no lo tiene, lo calcula).
- Es necesaria una estructura para almacenar las soluciones a los subproblemas, con la finalidad de ahorrar llamadas recursivas.

Ejemplos de dónde usar programación dinámica son:

- Fibonacci.
- Calcular números combinatorios.
- Calcular potencias naturales.
- Cualquier problema con solapamiento de subproblemas (encontramos subproblemas que se repiten).

Una cosa es la programación dinámica y otra es la memorización:

Memorización. Almacenamos en una estructura (como un diccionario) los resultados.

Programación dinámica. Una vez que los hemos almacenado, buscamos un patrón para ver cómo se completan las soluciones de alguna forma más eficiente (quitando sobrecarga por la recursividad). Buscamos una forma de rellenar la estructura de datos.

Cuando aplicar programación dinámica

Normalmente para problemas de optimización (minimizar o maximizar). La solución al problema la tenemos que ver como un proceso de selección de varias etapas.

- Se aplica a problemas que pueden suponer un alto coste computacional que dispone de subestructuras optimales que se solapan (se repiten a lo largo del cálculo de la solución).

La eficiencia del algoritmo suele ser polinomial. Normalmente suele ser $O(n \cdot m)$ donde n es el tamaño de la estructura de datos y m el tiempo para cada casilla.

Comparación con Divide y Vencerás

Divide y vencerás.

- Se aplica a subproblemas independientes.
- La técnica suele ser descendente.

Programación dinámica.

- Se aplica a subproblemas que se solapan (que se resuelven más de una vez).
- La técnica suele ser ascendente.
- Asegura optimilidad pero puede llevar a un algoritmo que no sea eficiente.

Teorema 5.1 (Principio de Optimalidad de Bellman)**.** *Una solución óptima está compuesta de subsoluciones óptimas.*

Cuando se cumpla el principio, se podrá utilizar la programción dinámica.

Ejemplo. Un ejemplo que no cumple el Principio de Optimalidad de Bellman es el problema del camino más largo entre dos nodos en un grafo.

Por tanto, no podemos aplicar programación dinámica para resolverlo.

5.1. Pasos para desarrollar un algoritmo

1. Plantear la solución a un problema como una secuencia de decisiones.
2. Ver que se verifica el principio de optimalidad **??**.
3. Plantear la solución como una función recursiva y ver la tipología de los subprobelmas.
4. Ver cómo un problema grande se puede calcular a partir de los problemas más pequeños.
5. Tratar de buscar un enfoque ascendente (resolver problemas pequeños y resolver problemas mayores).

Ante un problema del estilo buscar un camino óptimo, es capaz de decir el costo del camino pero no de decir el camino. Para ello:

- O se puede deducir el camino a partir del costo.
- O apuntar en una tabla auxiliar las decisiones tomadas.

5.2. Ejemplos de Programación Dinámica

Ejemplo. Problema de devolver cambio: ¿cómo devolver el cambio con el menor número de monedas?

Podemos considerarlo como un problema de resolución multietápica. Cumple el Principio de Optimalidad de Bellman, luego podemos aplicar programación dinámica para resolver el problema.

Ejemplo. Producto encadenado de matrices. Su solución es similar al árbol binario de búsqueda óptima (por ejemplo, en un árbol de palabras, poner las más frecuentes arriba y las menos abajo).

- Ante una solución que es multiplicar al tun tún, es mejor calcular primero el número de operaciones a realizar y quedarse con la multiplicación que dé el menor.

Se cumple el Principio de Optimalidad de Bellman, luego puede hacerse con programación dinámica.

Buscamos hacer la definición recursiva de la solución. Para ello vemos dónde colocar el primer paréntesis:

$$N[i,j] = 0 \qquad \text{si } i = j$$
$$N[i,j] = \min_{k \in [i,j]} \{N[i,k] + N[k+1,j] + p_{i-1}p_k p_j\} \text{ si } i < j$$

Siendo $N[i,j]$ el costo de multiplicar desde la matriz i-ésima a la j-ésima.

- Tenemos $O(n^2)$ subproblemas.
- Los subproblemas triviales son aquellos en los que $i = j$.

Representamos los $N[i,j]$ en una tabla. Formas de rellenar la tabla:

- Por diagonales.
- De abajo a arriba y de izquierda a derecha.

Una vez rellenada la tabla triangular superior, tenemos el número mínimo de operaciones para calcular el producto.

Tenemos un algoritmo $O(n^3)$. Sabemos el costo mínimo pero no tenemos cuál es la parantización que nos lo da. Para ello, o volvemos sobre nuestros pasos o almacenamos el menor en cada momento (el k de cada paso).

Ejemplo. Subsecuencia común de mayor longitud (LCS): Dadas dos secuencias de símbolos X e Y, buscamos la mayor subsecuencia común a ambas de mayor longitud (pudiendo estar tanto en X como en Y caraceteres intermedios):

$$X = A\ B\ C\ B\ D\ A\ B$$
$$Y = B\ D\ C\ A\ B\ A$$

La mayor subsecuencia común es $B\ D\ A\ B$, aunque también puede ser $B\ C\ B\ A$. Tiene aplicaciones en genética, diferencias de archivos y detección de plagio.

- Un algoritmo de fuerza bruta compararía cualquier subsecuencia de X con símbolos de Y.

  Si $|X| = m$ y $|Y| = n$, hay que contrastar 2^m subsecuencias de X contra los n elementos de Y, o al revés. Siendo:

  $$k = \text{mín}\{n, m\} \qquad t = \text{máx}\{n, m\}$$

  tenemos un orden de $O(t \cdot 2^k)$, en el mejor caso.

Cumple el Principio de Optimalidad de Bellman. Buscamos una secuencia de decisiones a desarrollar: Si meto o no un carácter.

Definimos:

$$X_i = (x_1, \ldots, x_i) \qquad Y_j = (y_1, \ldots, y_j)$$

Considerando que $X = (x_1, \ldots, x_m)$ con $m \geqslant i$ y $Y = (y1, \ldots, y_n)$ con $n \geqslant j$.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{si } x[i] = y[j] \\ \text{máx}\{c[i, j-1], c[i-1, j]\} & \text{si } x[i] \neq y[j] \end{cases}$$

siendo $c[i, j]$ la longitud de la solución óptima al problema de X_i y Y_j.

Por tanto, la solución es $c[n, m]$. Rellenaríamos la matriz por filas o por columnas (ver qué necesitamos para calcular $c[i, j]$, en cada caso). Algoritmo de orden $O(n \cdot m)$.

Para encontrar la solución miramos la solución recursiva que hicimos:

- Si son iguales la columna y la fila, disminuimos en 1 la fila y la columna.
- Si no son iguales, vuelvo al máximo.

Ejemplo. Dado un grafo y dos nodos, buscar el camino mínimo entre ambos.

Secuencia de decisiones a tomar: si consideramos un vértice o no para ser vértice intermedio del camino.

Notación. Notaremos por $D[i, j]$ al camino mínimo entre i y j.

Además, notaremos por $D_k[i, j]$ al camino mínimo entre i y j en un problema de los primeros k nodos del grafo.

De esta forma, $D_0[i, j] < \infty \iff$ hay un camino directo entre i y j.

Definición recursiva:

- Si el camino de i a j por k vértices no pasa por k:

  $$D_k[i, j] = D_{k-1}[i, j]$$

- Si pasar por k:

  $$D_k[i, j] = D_{k-1}[i, k] + D_{k-1}[k, j]$$

De donde:

$$D_k[i,j] = \text{mín}\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\}$$

Tenemos varias matrices, desde 0 hasta $n-1$ (siendo n el tamaño del grafo). Notemos que a la hora de construir la matriz D_k, no alteramos ni la fila k ni la columna k.

Obtenemos el **algoritmo de Floyd**, de eficiencia $O(n^3)$.

Ejemplo. Distancia de edición. Dadas dos cadenas, buscar la transformación de menor coste para llegar de una palabra a otra.

Podemos reemplazar, insertar y borrar caracteres.

Notación. Notamos por $d(i,j)$ a la distancia de edición de transformar la cadena desde el carácter 1 hasta el i hasta la que tenemos por transformar desde el 1 hasta el j.

Definición recursiva:

$$d(i,j) = \begin{cases} d(i-1,j-1) & \text{si} \quad s[i] = t[j] \\ 1 + \text{mín}\{d(i-1,j), d(i,j-1), d(i-1,j-1)\} & \text{si} \quad s[i] \neq t[j] \end{cases}$$

Por ejemplo:

$$d(\text{casa}, \text{costa}) = d(\text{cas}, \text{cost})$$

$$d(\text{casas}, \text{cosa}) = \text{mín} \left\{ \begin{array}{l} 1 + d(\text{casa}, \text{cosa}), \\ 1 + d(\text{casas}, \text{cos}), \\ 1 + d(\text{casa}, \text{cos}) \end{array} \right\}$$

Ejemplo. Problema de caminos mínimos con peso negativo. Estudiamos el algoritmos de Bellman-Ford.

Notación. Notamos por $D_i(v)$ al camino de costo mínimo entre el nodo v (origen) y el t (destino) usando como mucho i arcos.

6. Relaciones de Problemas

6.1. La eficiencia de los algoritmos

Ejercicio 6.1.1. Demostrar las siguientes propiedades:

a) $k \cdot f(n) \in O(f(n)), \qquad \forall k > 0.$

Hemos de ver que existe una constante $c \in \mathbb{R}^+,\ n_0 \in \mathbb{N}$ tal que $k \cdot f(n) \leqslant c \cdot f(n)$ para todo $n \geqslant n_0$. En este caso, podemos tomar $c = k$ y $n_0 = 1$ y se tiene que $k \cdot f(n) \leqslant k \cdot f(n)$ para todo $n \in \mathbb{N}$.

b) $n^r \in O(n^k)$ si $0 \leqslant r \leqslant k$.

Hemos de ver que existe una constante $c \in \mathbb{R}^+,\ n_0 \in \mathbb{N}$ tal que $n^r \leqslant c \cdot n^k$ para todo $n \geqslant n_0$.

Como $0 \leqslant r \leqslant k$, entonces $n^r \leqslant n^k$ para todo $n \in \mathbb{N}$, por lo que podemos tomar $c = 1$ y $n_0 = 1$.

c) $O(n^k) \subset O(n^{k+1})$.

Sea $f(n) \in O(n^k)$; es decir, existe una constante $c \in \mathbb{R}^+,\ n_0 \in \mathbb{N}$ tal que $f(n) \leqslant c \cdot n^k$ para todo $n \geqslant n_0$. Hemos de ver que $f(n) \in O(n^{k+1})$; es decir, que existe una constante $c' \in \mathbb{R}^+,\ n_0' \in \mathbb{N}$ tal que $f(n) \leqslant c' \cdot n^{k+1}$ para todo $n \geqslant n_0'$.

Tomando $c' = c$ y $n_0' = n_0$, se tiene que $f(n) \leqslant c \cdot n^k \leqslant c \cdot n^{k+1}$ para todo $n \geqslant n_0$, por lo que $f(n) \in O(n^{k+1})$.

d) $n^k \in O(b^n) \quad \forall b > 1, k \geqslant 0.$

Hemos de ver que existe una constante $c \in \mathbb{R}^+,\ n_0 \in \mathbb{N}$ tal que $n^k \leqslant c \cdot b^n$ para todo $n \geqslant n_0$. Tomando $c = 1$, tenemos que dicho valor de n_0 existe, ya que:

$$\lim_{n \to \infty} \frac{n^k}{b^n} = 0$$

e) $\log_b n \in O(n^k) \quad \forall b > 1, k > 0.$

Hemos de ver que existe una constante $c \in \mathbb{R}^+,\ n_0 \in \mathbb{N}$ tal que $\log_b n \leqslant c \cdot n^k$ para todo $n \geqslant n_0$. Tomando $c = 1$, tenemos que dicho valor de n_0 existe, ya que:

$$\lim_{n \to \infty} \frac{\log_b n}{n^k} = 0$$

f) Si $f(n) \in O(g(n))$ y $h(n) \in O(g(n))$, entonces $f(n) + h(n) \in O(g(n))$.

Tenemos que:

$$f(n) \in O(g(n)) \Longrightarrow \exists c_1 \in \mathbb{R}^+,\ n_1 \in \mathbb{N} \text{ tal que } f(n) \leqslant c_1 \cdot g(n) \quad \forall n \geqslant n_1,$$
$$h(n) \in O(g(n)) \Longrightarrow \exists c_2 \in \mathbb{R}^+,\ n_2 \in \mathbb{N} \text{ tal que } h(n) \leqslant c_2 \cdot g(n) \quad \forall n \geqslant n_2.$$

Tomando $c = c_1 + c_2$ y $n_0 = \text{máx}\{n_1, n_2\}$, se tiene que:

$$f(n) + h(n) \leqslant c_1 \cdot g(n) + c_2 \cdot g(n) = (c_1 + c_2) \cdot g(n) \quad \forall n \geqslant n_0,$$

g) Si $f(n) \in O(g(n))$, entonces $f(n) + g(n) \in O(g(n))$.

Por el primer apartado, sabemos que $g(n) \in O(g(n))$. Por tanto, usando el apartado anterior, se tiene que $f(n) + g(n) \in O(g(n))$.

h) *Reflexividad*: $f(n) \in O(f(n))$.

Se tiene de forma directa por el primer apartado tomando $k = 1$.

i) *Transitividad*: Si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$, entonces $f(n) \in O(h(n))$.

Tenemos que:

$$f(n) \in O(g(n)) \Longrightarrow \exists c_1 \in \mathbb{R}^+,\ n_1 \in \mathbb{N} \text{ tal que } f(n) \leqslant c_1 \cdot g(n) \quad \forall n \geqslant n_1,$$
$$g(n) \in O(h(n)) \Longrightarrow \exists c_2 \in \mathbb{R}^+,\ n_2 \in \mathbb{N} \text{ tal que } g(n) \leqslant c_2 \cdot h(n) \quad \forall n \geqslant n_2.$$

Por tanto, tomando $c = c_1 \cdot c_2$ y $n_0 = \text{máx}\{n_1, n_2\}$, se tiene que:

$$f(n) \leqslant c_1 \cdot g(n) \leqslant c_1 \cdot c_2 \cdot h(n) = c \cdot h(n) \quad \forall n \geqslant n_0,$$

j) *Regla de la suma*: Si $T1(n)$ es $O(f(n))$ y $T2(n)$ es $O(g(n))$, entonces:

$$T1(n) + T2(n) \in O(\text{máx}\{f(n), g(n)\}).$$

Tenemos que:

$$T1(n) \in O(f(n)) \Longrightarrow \exists c_1 \in \mathbb{R}^+,\ n_1 \in \mathbb{N} \text{ tal que } T1(n) \leqslant c_1 \cdot f(n) \quad \forall n \geqslant n_1,$$
$$T2(n) \in O(g(n)) \Longrightarrow \exists c_2 \in \mathbb{R}^+,\ n_2 \in \mathbb{N} \text{ tal que } T2(n) \leqslant c_2 \cdot g(n) \quad \forall n \geqslant n_2.$$

Tomando $c = \text{máx}\{c_1, c_2\}$ y $n_0 = \text{máx}\{n_1, n_2\}$, se tiene que:

$$T1(n) + T2(n) \leqslant c_1 \cdot f(n) + c_2 \cdot g(n) \leqslant c \cdot \text{máx}\{f(n), g(n)\} \quad \forall n \geqslant n_0,$$

k) *Regla del producto*: Si $T1(n)$ es $O(f(n))$ y $T2(n)$ es $O(g(n))$, entonces:

$$T1(n) \cdot T2(n) \in O(f(n) \cdot g(n)).$$

Tenemos que:

$$T1(n) \in O(f(n)) \Longrightarrow \exists c_1 \in \mathbb{R}^+,\ n_1 \in \mathbb{N} \text{ tal que } T1(n) \leqslant c_1 \cdot f(n) \quad \forall n \geqslant n_1,$$
$$T2(n) \in O(g(n)) \Longrightarrow \exists c_2 \in \mathbb{R}^+,\ n_2 \in \mathbb{N} \text{ tal que } T2(n) \leqslant c_2 \cdot g(n) \quad \forall n \geqslant n_2.$$

Tomando $c = c_1 \cdot c_2$ y $n_0 = \text{máx}\{n_1, n_2\}$, se tiene que:

$$T1(n) \cdot T2(n) \leqslant c_1 \cdot f(n) \cdot c_2 \cdot g(n) = c \cdot f(n) \cdot g(n) \quad \forall n \geqslant n_0,$$

Ejercicio 6.1.2. Expresar, en notación $O(\cdot)$, el orden que tendrí un algoritmo cuyo tiempo de ejecución fuera $f_i(n)$, donde:

1. $f_1(n) = n^2$

   En este caso se tiene que $f_1(n) \in O(n^2)$.

2. $f_2(n) = n^2 + 1000n$

   En este caso, por la regla de la suma, se tiene que $f_2(n) \in O(n^2)$.

3. $f_3(n) = \begin{cases} n & \text{si } n \text{ es par} \\ n^3 & \text{si } n \text{ es impar} \end{cases}$

   En este caso, como $n^3 \geqslant n$ para todo $n \geqslant 1$, se tiene que $f_3(n) \in O(n^3)$.

4. $f_n4(n) = \begin{cases} n & \text{si } n \leqslant 100 \\ n^3 & \text{si } n > 100 \end{cases}$

   En este caso, como se trata de comportamientos asintóticos, se tiene que $f_4(n) \in O(n^3)$.

5. $f_5(n) = (n-1)^3$

   En este caso, por la regla de la suma, se tiene que $f_5(n) \in O(n^3)$.

6. $f_6(n) = \sqrt{n^2 - 1}$.

   En este caso, como $\sqrt{n^2-1} \leqslant n$ para todo $n \geqslant 1$, se tiene que $f_6(n) \in O(n)$.

7. $f_7(n) = \log(n!)$

   Por el Criterio de Stolz, se tiene que:

   $$\left\{\frac{\log(n!)}{n\log n}\right\} = \left\{\frac{\log(n+1)}{(n+1)\log(n+1) - n\log n}\right\} = \left\{\frac{1}{n+1-n\cdot\frac{\log(n)}{\log(n+1)}}\right\} \to \frac{1}{n+1-n} = 1$$

   Por tanto, tenemos que $\log(n!) \in O(n\log n)$.

8. $f_8(n) = n!$

   Claramente, $f_8(n) \in O(n!)$.

Ejercicio 6.1.3. Usando la notación $O(\cdot)$, obtener el tiempo de ejecución de las siguientes funciones:

1. Código Fuente **??** (`ejemplo1`).

```
1  void ejemplo1 (int n)
2  {
3      int i, j, k;
4
5      for (i = 0; i < n; i++)
6          for (j = 0; j < n; j++)
7          {
8              C[i][j] = 0;
9              for (k = 0; k < n; k++)
10                 C[i][j] += A[j][k] * B[k][j];
11         }
12 }
```

Código fuente 1: Función del Ejercicio ?? apartado ??.

En este caso, el tiempo de ejecución de la linea 10 lo podemos acotar por una constante, sea esta c. Entonces, tenemos que el tiempo de ejecución de la función `ejemplo1` es:

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left(1+\sum_{k=0}^{n-1} c\right) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(1+c\cdot(n-1-0+1)) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(1+c\cdot n) =$$
$$= n^2\cdot(1+c\cdot n) \in O(n^3)$$

Se trata del algoritmo de multiplicación de matrices, cuyo tiempo de ejecución es $O(n^3)$.

2. Código Fuente ?? (`ejemplo2`).

```
1  long ejemplo2 (int n)
2  {
3      int i, j, k;
4      long total = 0;
5
6      for (i = 0; i < n; i++)
7          for (j = i+1; j <= n; j++)
8              for (k = 1; k <= j; k++)
9                  total += k*i;
10
11     return total;
12 }
```

Código fuente 2: Función del Ejercicio ?? apartado ??.

En este caso, el tiempo de ejecución de la linea 9 lo podemos acotar por una constante, sea esta c. Entonces, tenemos que el tiempo de ejecución de la

función `ejemplo2` es:

$$T(n) = 1 + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} \sum_{k=1}^{j} c = 1 + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} c \cdot (j - 1 + 1) = 1 + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} c \cdot j =$$
$$= 1 + \sum_{i=0}^{n-1} c \cdot \left(\frac{n(n+1)}{2} - \frac{i(i+1)}{2} \right) = 1 + cn \cdot \frac{n(n+1)}{2} - c \cdot \sum_{i=0}^{n-1} \frac{i(i+1)}{2} =$$
$$= 1 + cn \cdot \frac{n(n+1)}{2} - \frac{c}{2} \sum_{i=0}^{n-1} i^2 - \frac{c}{2} \sum_{i=0}^{n-1} i =$$
$$= 1 + cn \cdot \frac{n(n+1)}{2} - \frac{c}{2} \cdot \frac{(n-1)n(2(n-1)+1)}{6} - \frac{c}{2} \cdot \frac{(n-1)n}{2} \in O(n^3)$$

Por tanto, el tiempo de ejecución de la función `ejemplo2` es $O(n^3)$.

3. Código Fuente **??** (`ejemplo3`).

```
1  void ejemplo3 (int n)
2  {
3      int i, j, x=0, y=0;
4
5      for (i = 1; i <= n; i++)
6          if (i % 2 == 1)
7          {
8              for (j = i; j <= n; j++)
9                  x++;
10             for (j = 0; j < i; j++)
11                 y++;
12         }
13 }
```

Código fuente 3: Función del Ejercicio **??** apartado **??**.

Tenemos que:

$$T(n) = 1 + \sum_{i=1}^{n} \left(2 + \sum_{j=i}^{n} 1 + \sum_{j=0}^{i-1} 1 \right) = 1 + \sum_{i=1}^{n} (2 + (n - i + 1) + i) =$$
$$= 1 + \sum_{i=1}^{n} (n + 3) = 1 + n(n + 3) \in O(n^2)$$

Por tanto, el tiempo de ejecución de la función `ejemplo3` es $O(n^2)$.

4. Código Fuente **??** (`ejemplo4`).

```
1  int ejemplo4 (int n)
2  {
3      if (n <= 1)
4          return 1;
5      else
6          return (ejemplo4(n - 1) + ejemplo4(n-1));
7  }
```

Código fuente 4: Función del Ejercicio ?? apartado ??.

En este caso, el tiempo de ejecución de la función `ejemplo4` es:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n-1)+1 & \text{en otro caso} \end{cases}$$

Opción 1. Desarrolo en series. Desarrollando en series, tenemos que:

$$\begin{aligned} T(n) &= 2T(n-1)+1 = 2(2T(n-2)+1)+1 = 2^2T(n-2)+2+1 = \\ &= 2^3T(n-3)+2^2+2+1 = \ldots = 2^iT(n-i)+2^{i-1}+\ldots+2+1 = \\ &= 2^iT(n-i)+\sum_{j=0}^{i-1}2^j \qquad \forall i \in \mathbb{N}, i \geqslant n-1 \end{aligned}$$

Para $i = n-1$, se tiene que:

$$\begin{aligned} T(n) &= 2^{n-1}T(1)+\sum_{j=0}^{n-2}2^j = 2^{n-1}+\sum_{j=0}^{n}2^j-2^{n-1}-2^n = \\ &= \cancel{2^{n-1}}+\frac{1-2^{n+1}}{1-2}-\cancel{2^{n-1}}-2^n = \\ &= -1+2^{n+1}-2^n \end{aligned}$$

Comprobemos que $T(n) \in O(2^n)$:

$$\lim_{n\to\infty}\frac{T(n)}{2^n} = \lim_{n\to\infty}\frac{-1+2^{n+1}-2^n}{2^n} = \lim_{n\to\infty}\frac{-1}{2^n}+2-1 = 2-1 = 1 \in \mathbb{R}^+$$

Por tanto, se tiene que $T(n) \in O(2^n)$.

Opción 2. Ecuación Característica.

Aplicando el cambio de variable $T(n) = x^n$, en la parte homogénea tenemos que:

$$x^n - 2x^{n-1} = 0 \Longrightarrow x^{n-1}(x-2) = 0$$

Como $x^{n-1} > 0$ para todo $n \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 2$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $1 = 1^n n^0$, por lo que el polinomio característico de la ecuación en diferencias es:

$$p(x) = (x-2)(x-1)$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(n) = x^n = c_1 \cdot 2^n + c_2 \cdot 1^n = c_1 \cdot 2^n + c_2 \in O(2^n)$$

De ambas formas, se tiene que el tiempo de ejecución de la función `ejemplo4` es $O(2^n)$.

5. Código Fuente **??** (`ejemplo5`).

```
1  int ejemplo5 (int n)
2  {
3      if (n == 1)
4          return n;
5      else
6          return (ejemplo5(n/2) + 1);
7  }
```

Código fuente 5: Función del Ejercicio **??** apartado **??**.

En este caso, el tiempo de ejecución de la función `ejemplo5` es:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T\left(\frac{n}{2}\right) + 1 & \text{en otro caso} \end{cases}$$

Opción 1. Desarrolo en series. Desarrollando en series, tenemos que:

$$T(n) = T\left(\frac{n}{2}\right) + 1 = T\left(\frac{n}{2^2}\right) + 1 + 1 = T\left(\frac{n}{2^3}\right) + 1 + 1 + 1 = \ldots =$$
$$= T\left(\frac{n}{2^i}\right) + i \qquad \forall i \in \mathbb{N}, i \geqslant \log_2 n$$

Para $i = \log_2 n$, se tiene que:

$$T(n) = T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n = T(1) + \log_2 n = 1 + \log_2 n \in O(\log n)$$

Por tanto, el tiempo de ejecución de la función `ejemplo5` es $O(\log n)$.

Opción 2. Ecuación Característica.

Aplicando el cambio de variable $n = 2^m$, en la parte homogénea tenemos que:

$$T(2^m) - T(2^{m-1}) = 0$$

Por tanto, la ecuación característica de la ecuación en diferencias es:

$$x^m - x^{m-1} = 0 \Longrightarrow x^{m-1}(x-1) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 1$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $1 = 1^m$, por lo que el polinomio característico de la ecuación en diferencias es:

$$p(x) = (x-1)(x-1) = (x-1)^2$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(2^m) = c_1 \cdot 1^m + c_2 \cdot m \cdot 1^m = c_1 + c_2 \cdot m$$

Deshaciendo el cambio de variable $m = \log_2 n$, se tiene que:

$$T(n) = c_1 + c_2 \cdot \log_2 n \in O(\log n)$$

De ambas formas, se tiene que el tiempo de ejecución de la función `ejemplo5` es $O(\log n)$.

Ejercicio 6.1.4. Resolver las siguientes recurrencias:

a) $T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 2T(n-1)+1 & \text{en otro caso} \end{cases}$

Opción 1. Desarrolo en series.

Desarrollando en series, tenemos que:

$$\begin{aligned} T(n) &= 2T(n-1)+1 = 2(2T(n-2)+1)+1 = 2^2T(n-2)+2+1 = \\ &= 2^3T(n-3)+2^2+2+1 = \ldots = 2^iT(n-i)+2^{i-1}+\ldots+2+1 = \\ &= 2^iT(n-i)+\sum_{j=0}^{i-1}2^j \qquad \forall i \in \mathbb{N}, i \geqslant n \end{aligned}$$

Para $i = n$, se tiene que:

$$\begin{aligned} T(n) &= 2^nT(0)+\sum_{j=0}^{n-1}2^j = \sum_{j=0}^{n}2^j - 2^n = \\ &= \frac{1-2^{n+1}}{1-2} - 2^n = -1+2^{n+1}-2^n \in O(2^n) \end{aligned}$$

Veamos que efectivamente $T(n) \in O(2^n)$:

$$\lim_{n\to\infty}\frac{T(n)}{2^n} = \lim_{n\to\infty}\frac{-1+2^{n+1}-2^n}{2^n} = \lim_{n\to\infty}\frac{-1}{2^n}+2-1 = 2-1 = 1 \in \mathbb{R}^+$$

Por tanto, se tiene que $T(n) \in O(2^n)$.

Opción 2. Ecuación Característica.

Aplicando el cambio de variable $T(n) = x^n$, en la parte homogénea tenemos que:

$$x^n - 2x^{n-1} = 0 \Longrightarrow x^{n-1}(x-2) = 0$$

Como $x^{n-1} > 0$ para todo $n \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 2$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $1 = 1^n n^0$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-2)(x-1)$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(n) = x^n = c_1 \cdot 2^n + c_2 \cdot 1^n = c_1 \cdot 2^n + c_2 \in O(2^n)$$

b) $T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 2T(n-1)+n & \text{en otro caso} \end{cases}$

Opción 1. Desarrolo en series.

Desarrollando en series, tenemos que:

$$\begin{aligned}T(n) &= 2T(n-1)+n = 2(2T(n-2)+n-1)+n = 2^2T(n-2)+2(n-1)+n = \\ &= 2^3T(n-3)+2^2(n-2)+2(n-1)+n = \ldots = \\ &= 2^iT(n-i)+2^{i-1}(n-i+1)+\ldots+2(n-1)+1\cdot n = \\ &= 2^iT(n-i)+\sum_{j=0}^{i-1}2^j(n-j) \qquad \forall i\in\mathbb{N}, i\geqslant n\end{aligned}$$

Para $i=n$, se tiene que:

$$\begin{aligned}T(n) &= 2^nT(0)+\sum_{j=0}^{n-1}2^j(n-j) = \sum_{j=0}^{n}2^j(n-j)-2^n(n-n) = \\ &= \sum_{j=0}^{n}2^jn-\sum_{j=0}^{n}2^jj = n\sum_{j=0}^{n}2^j-\sum_{j=0}^{n}j2^j\end{aligned}$$

Como vemos, la suma de la derecha no es de cálculo sencillo, por lo que vamos a intentar resolver la recurrencia por otro método.

Opción 2. Ecuación Característica.

Aplicando el cambio de variable $T(n)=x^n$, en la parte homogénea tenemos que:

$$x^n-2x^{n-1}=0 \Longrightarrow x^{n-1}(x-2)=0$$

Como $x^{n-1}>0$ para todo $n\in\mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x=2$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $n=1^n\cdot n^1$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x)=(x-2)(x-1)^2$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(n)=x^n=c_1\cdot 2^n+c_2\cdot 1^n+c_3\cdot n\cdot 1^n=c_1\cdot 2^n+c_2+c_3\cdot n\in O(2^n)$$

c) $T(n)=\begin{cases}0 & \text{si } n=0\\ 1 & \text{si } n=1\\ T(n-1)+T(n-2) & \text{en otro caso}\end{cases}$

Aplicamos el cambio de variable $T(n)=x^n$, en la parte homogénea tenemos que:

$$x^n-x^{n-1}-x^{n-2}=0 \Longrightarrow x^{n-2}(x^2-x-1)=0$$

Resolviendo la ecuación de segundo grado, se tiene que:

$$x^2-x-1=0 \Longrightarrow x=\frac{1\pm\sqrt{5}}{2}$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(n)=x^n=c_1\left(\frac{1+\sqrt{5}}{2}\right)^n+c_2\left(\frac{1-\sqrt{5}}{2}\right)^n\in O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

d) $T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ 3T(n-1) + 4T(n-2) & \text{en otro caso} \end{cases}$

Aplicamos el cambio de variable $T(n) = x^n$, en la parte homogénea tenemos que:

$$x^n - 3x^{n-1} - 4x^{n-2} = 0 \Longrightarrow x^{n-2}(x^2 - 3x - 4) = 0$$

Resolviendo la ecuación de segundo grado, se tiene que:

$$x^2 - 3x - 4 = 0 \Longrightarrow x = \frac{3 \pm \sqrt{25}}{2} = \frac{3 \pm 5}{2} = \begin{cases} x_1 = 4 \\ x_2 = -1 \end{cases}$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(n) = x^n = c_1 \cdot 4^n + c_2 \cdot (-1)^n \in O(4^n)$$

e) $T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ 5T(n-1) - 8T(n-2) + 4T(n-3) & \text{en otro caso} \end{cases}$

Aplicamos el cambio de variable $T(n) = x^n$, en la parte homogénea tenemos que:

$$x^n - 5x^{n-1} + 8x^{n-2} - 4x^{n-3} = 0 \Longrightarrow x^{n-3}(x^3 - 5x^2 + 8x - 4) = 0$$

Resolviendo la ecuación de tercer grado, se tiene que:

	1	−5	8	−4
1		1	−4	4
	1	−4	4	0

Figura 6.1: División mediante Ruffini donde se ve que $x = 1$ es una solución.

Por tanto, tenemos que el polinomio característico queda:

$$(x-1)(x^2 - 4x + 4) = (x-1)(x-2)^2$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(n) = x^n = c_1 \cdot 1^n + c_2 \cdot 2^n + c_3 \cdot n \cdot 2^n \in O(n2^n)$$

f) $T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 36 & \text{si } n = 1 \\ 5T(n-1) + 6T(n-2) + 4 \cdot 3^n & \text{en otro caso} \end{cases}$

Aplicamos el cambio de variable $T(n) = x^n$, en la parte homogénea tenemos que:

$$x^n - 5x^{n-1} - 6x^{n-2} = 0 \Longrightarrow x^{n-2}(x^2 - 5x - 6) = 0$$

Resolviendo la ecuación de segundo grado, se tiene que:

$$x^2 - 5x - 6 = 0 \Longrightarrow x = \frac{5 \pm \sqrt{49}}{2} = \frac{5 \pm 7}{2} = \begin{cases} x_1 = 6 \\ x_2 = -1 \end{cases}$$

Respecto a la parte no homogénea, se tiene que $4 \cdot 3^n = 3^n(4n^0)$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-6)(x+1)(x-3)$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(n) = x^n = c_1 \cdot 6^n + c_2 \cdot (-1)^n + c_3 \cdot 3^n \in O(6^n)$$

g) $T(n) = 2T(n-1) + 3^n$.

Aplicamos el cambio de variable $T(n) = x^n$, en la parte homogénea tenemos que:

$$x^n - 2x^{n-1} = 0 \Longrightarrow x^{n-1}(x-2) = 0$$

Como $x^{n-1} > 0$ para todo $n \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 2$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $3^n = 3^n n^0$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-2)(x-1)$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(n) = x^n = c_1 \cdot 2^n + c_2 \cdot 1^n = c_1 \cdot 2^n + c_2 \in O(2^n)$$

h) $T(n) = 2T(n-1) + n + 2^n$.

Aplicamos el cambio de variable $T(n) = x^n$, en la parte homogénea tenemos que:

$$x^n - 2x^{n-1} = 0 \Longrightarrow x^{n-1}(x-2) = 0$$

Como $x^{n-1} > 0$ para todo $n \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 2$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $n + 2^n = 1^n \cdot n + 2^n n^0$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-2)(x-1)^2(x-2) = (x-2)^2(x-1)^2$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(n) = x^n = c_1 \cdot 2^n + c_2 \cdot n \cdot 2^n + c_3 \cdot 1^n + c_4 \cdot n \cdot 1^n = c_1 \cdot 2^n + c_2 \cdot n2^n + c_3 + c_4 \cdot n \in O(n2^n)$$

i) $T(n) = 2T\left(n/2\right) + \log n$.

Aplicamos el cambio de variable $n = 2^m$, y tenemos que:

$$T(2^m) = 2T\left(2^{m-1}\right) + m \log 2$$

Aplicamos el cambio de variable $T(2^m) = x^m$, en la parte homogénea tenemos que:

$$x^m - 2x^{m-1} = 0 \Longrightarrow x^{m-1}(x-2) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 2$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $m \log 2 = 1^m \cdot (\log_2 m^1)$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-2)(x-1)^2$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(2^m) = x^m = c_1 \cdot 2^m + c_2 \cdot 1^m + c_3 \cdot m \cdot 1^m = c_1 \cdot 2^m + c_2 + c_3 \cdot m$$

Deshaciendo el cambio de variable $m = \log_2 n$, se tiene que:

$$T(n) = c_1 \cdot 2^{\log_2 n} + c_2 + c_3 \cdot \log_2 n = c_1 \cdot n + c_2 + c_3 \cdot \log_2 n \in O(n)$$

j) $T(n) = 4T\left(n/2\right) + n$.

Aplicamos el cambio de variable $n = 2^m$, y tenemos que:

$$T(2^m) = 4T\left(2^{m-1}\right) + 2^m$$

Aplicamos el cambio de variable $T(2^m) = x^m$, en la parte homogénea tenemos que:

$$x^m - 4x^{m-1} = 0 \Longrightarrow x^{m-1}(x-4) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 4$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $2^m = 2^m \cdot (m^0)$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-4)(x-2)$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(2^m) = x^m = c_1 \cdot 4^m + c_2 \cdot 2^m$$

Deshaciendo el cambio de variable $m = \log_2 n$, se tiene que:

$$T(n) = c_1 \cdot 4^{\log_2 n} + c_2 \cdot 2^{\log_2 n} = c_1 \cdot n^2 + c_2 \cdot n \in O(n^2)$$

k) $T(n) = 4T\left(n/2\right) + n^2$.

Aplicamos el cambio de variable $n = 2^m$, y tenemos que:

$$T(2^m) = 4T\left(2^{m-1}\right) + 2^{2m}$$

Aplicamos el cambio de variable $T(2^m) = x^m$, en la parte homogénea tenemos que:

$$x^m - 4x^{m-1} = 0 \Longrightarrow x^{m-1}(x-4) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 4$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $2^{2m} = 4^m \cdot (m^0)$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-4)^2$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(2^m) = x^m = c_1 \cdot 4^m + c_2 \cdot m \cdot 4^m$$

Deshaciendo el cambio de variable $m = \log_2 n$, se tiene que:

$$T(n) = c_1 \cdot 4^{\log_2 n} + c_2 \cdot \log_2 n \cdot 4^{\log_2 n} = c_1 \cdot n^2 + c_2 \cdot n^2 \log_2 n \in O(n^2 \log n)$$

l) $T(n) = 2T\left(n/2\right) + n \log n$.

Aplicamos el cambio de variable $n = 2^m$, y tenemos que:

$$T(2^m) = 2T\left(2^{m-1}\right) + 2^m \cdot m \log 2$$

Aplicamos el cambio de variable $T(2^m) = x^m$, en la parte homogénea tenemos que:

$$x^m - 2x^{m-1} = 0 \Longrightarrow x^{m-1}(x-2) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 2$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $2^m \cdot m \log 2 = 2^m \cdot \log_2 m^1$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-2)(x-2)^2 = (x-2)^3$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(2^m) = x^m = c_1 \cdot 2^m + c_2 \cdot m \cdot 2^m + c_3 \cdot m^2 \cdot 2^m$$

Deshaciendo el cambio de variable $m = \log_2 n$, se tiene que:

$$\begin{aligned} T(n) &= c_1 \cdot 2^{\log_2 n} + c_2 \cdot \log_2 n \cdot 2^{\log_2 n} + c_3 \cdot \log_2^2 n \cdot 2^{\log_2 n} = \\ &= c_1 \cdot n + c_2 \cdot n \log_2 n + c_3 \cdot n (\log_2 n)^2 \in O(n \log^2 n) \end{aligned}$$

m) $T(n) = \begin{cases} 1 & \text{si } n = 2 \\ 2T\left(\sqrt{n}\right) + \log n & \text{si } n \geqslant 4 \end{cases}$

Aplicamos el cambio de variable $n = 2^{(2^m)}$, y tenemos que:

$$T\left(2^{(2^m)}\right) = 2T\left(2^{(2^{m-1})}\right) + 2^m$$

Aplicamos el cambio de variable $T\left(2^{(2^m)}\right) = x^m$, en la parte homogénea tenemos que:

$$x^m - 2x^{m-1} = 0 \Longrightarrow x^{m-1}(x-2) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 2$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $2^m = 2^m \cdot m^0$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-2)(x-2) = (x-2)^2$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T\left(2^{(2^m)}\right) = x^m = c_1 \cdot 2^m + c_2 \cdot m \cdot 2^m$$

Deshaciendo el cambio de variable $m = \log_2 \log_2 n$, se tiene que:

$$\begin{aligned} T(n) &= c_1 \cdot 2^{\log_2 \log_2 n} + c_2 \cdot \log_2 \log_2 n \cdot 2^{\log_2 \log_2 n} = \\ &= c_1 \cdot \log_2 n + c_2 \cdot \log_2 n \cdot \log_2 \log_2 n \in O(\log n \cdot \log \log n) \end{aligned}$$

n) $T(n) = \begin{cases} 1 & \text{si } n = 2 \\ 2T\left(\sqrt{n}\right) + \log \log n & \text{si } n \geqslant 4 \end{cases}$

Aplicamos el cambio de variable $n = 2^{(2^m)}$, y tenemos que:

$$T\left(2^{(2^m)}\right) = 2T\left(2^{(2^{m-1})}\right) + m$$

Aplicamos el cambio de variable $T\left(2^{(2^m)}\right) = x^m$, en la parte homogénea tenemos que:

$$x^m - 2x^{m-1} = 0 \Longrightarrow x^{m-1}(x-2) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 2$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $m = 1^m \cdot m^1$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-2)(x-1)^2$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T\left(2^{(2^m)}\right) = x^m = c_1 \cdot 2^m + c_2 \cdot 1^m + c_3 \cdot m \cdot 1^m = c_1 \cdot 2^m + c_2 + c_3 \cdot m$$

Deshaciendo el cambio de variable $m = \log_2 \log_2 n$, se tiene que:

$$\begin{aligned} T(n) &= c_1 \cdot 2^{\log_2 \log_2 n} + c_2 + c_3 \cdot \log_2 \log_2 n = \\ &= c_1 \cdot \log_2 n + c_2 + c_3 \cdot \log_2 \log_2 n \in O(\log \log n) \end{aligned}$$

o) $T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 5T\left(n/2\right) + (n \log n)^2 & \text{si } n \geqslant 2 \end{cases}$

Aplicamos el cambio de variable $n = 2^m$, y tenemos que:

$$T(2^m) = 5T\left(2^{m-1}\right) + (2^m \log 2^m)^2 = 5T\left(2^{m-1}\right) + (m2^m)^2 = 5T\left(2^{m-1}\right) + m^2 \cdot 4^m$$

Aplicamos el cambio de variable $T(2^m) = x^m$, en la parte homogénea tenemos que:

$$x^m - 5x^{m-1} = 0 \Longrightarrow x^{m-1}(x-5) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 5$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $m^2 \cdot 4^m = 4^m \cdot (m^2)$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-5)(x-4)^3$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(2^m) = x^m = c_1 \cdot 5^m + c_2 \cdot 4^m + c_3 \cdot m \cdot 4^m + c_4 \cdot m^2 \cdot 4^m$$

Deshaciendo el cambio de variable $m = \log_2 \log_2 n$, se tiene que:

$$\begin{aligned} T(n) &= c_1 \cdot 5^{\log_2 \log_2 n} + c_2 \cdot 4^{\log_2 \log_2 n} + c_3 \cdot \log_2 \log_2 n \cdot 4^{\log_2 \log_2 n} + \\ &\quad + c_4 \cdot (\log_2 \log_2 n)^2 \cdot 4^{\log_2 \log_2 n} = \\ &= c_1 \cdot (\log_2 n)^{\log_2 5} + c_2 \cdot (\log_2 n)^{\log_2 4} + c_3 \cdot \log_2 \log_2 n \cdot (\log_2 n)^{\log_2 4} + \\ &\quad + c_4 \cdot (\log_2 \log_2 n)^2 \cdot (\log_2 n)^{\log_2 4} = \\ &= c_1 \cdot (\log_2 n)^{\log_2 5} + c_2 \cdot (\log_2 n)^2 + c_3 \cdot \log_2 \log_2 n \cdot (\log_2 n)^2 + \\ &\quad + c_4 \cdot (\log_2 \log_2 n)^2 \cdot (\log_2 n)^2 \end{aligned}$$

Por tanto, tenemos que $T(n) \in O\left((\log(\log(n)) \cdot \log(n))^2\right)$.

p) $T(n) = \sqrt{n} T\left(\sqrt{n}\right) + n, \qquad n \geqslant 4.$

q) $T(n) = \begin{cases} 6 & \text{si } n = 1 \\ nT^2\left(n/2\right) & \text{si } n > 1 \end{cases}.$

r) $T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 4 & \text{si } n = 2 \\ T\left(n/2\right) \cdot T^2\left(n/2\right) & \text{si } n \geqslant 4 \end{cases}$

Ejercicio 6.1.5. El tiempo de ejecución de un Algotimo A viene descrito por la recurrencia

$$T(n) = 7T\left(n/2\right) + n^2$$

Otro algoritmo B tiene un tiempo de ejecución descrito por la recurrencia

$$T'(n) = aT'\left(n/4\right) + n^2$$

¿Cuál es el mayor valor de la constante $a \in \mathbb{R}$ que hace al algoritmo B asintóticamente más eficiente que A?

Estudiamos en primer lugar la eficiencia del algoritmo A. Aplicamos el cambio de variable $n = 2^m$, y tenemos que:

$$T(2^m) = 7T\left(2^{m-1}\right) + 4^m$$

Aplicamos el cambio de variable $T(2^m) = x^m$, en la parte homogénea tenemos que:

$$x^m - 7x^{m-1} = 0 \Longrightarrow x^{m-1}(x-7) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = 7$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $4^m = 4^m \cdot (m^0)$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-7)(x-4)$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(2^m) = x^m = c_1 \cdot 7^m + c_2 \cdot 4^m$$

Deshaciendo el cambio de variable $m = \log_2 n$, se tiene que:

$$T(n) = c_1 \cdot 7^{\log_2 n} + c_2 \cdot 4^{\log_2 n} = c_1 \cdot n^{\log_2 7} + c_2 \cdot n^{\log_2 4} = c_1 \cdot n^{\log_2 7} + c_2 \cdot n^2 \in O(n^{\log_2 7})$$

Estudiamos en segundo lugar la eficiencia del algoritmo B. Aplicamos el cambio de variable $n = 4^m$, y tenemos que:

$$T'(4^m) = aT'\left(4^{m-1}\right) + 16^m$$

Aplicamos el cambio de variable $T'(4^m) = x^m$, en la parte homogénea tenemos que:

$$x^m - ax^{m-1} = 0 \Longrightarrow x^{m-1}(x-a) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = a$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $16^m = 16^m \cdot (m^0)$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x-a)(x-16)$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T'(4^m) = x^m = c_1 \cdot a^m + c_2 \cdot 16^m$$

Deshaciendo el cambio de variable $m = \log_4 n$, se tiene que:

$$T'(n) = c_1 \cdot a^{\log_4 n} + c_2 \cdot 16^{\log_4 n} = c_1 \cdot n^{\log_4 a} + c_2 \cdot n^2 \in O\left(n^{\max\{2, \log_4 a\}}\right)$$

Para que B sea asintóticamente más eficiente que A, es necesario que:

$$\max\{2, \log_4 a\} < \log_2 7 \Longleftrightarrow \log_4 a < \log_2 7 \Longleftrightarrow a < 7^{\log_2 4} = 7^2 = 49$$

Por tanto, los valores que hacen que el algoritmo B sea asintóticamente más eficiente que A son aquellos $a \in \mathbb{R}$ tales que $a < 49$.

Ejercicio 6.1.6. Resuelva la siguiente recurrencia:

$$T(n) = aT\left(\frac{n}{b}\right) + n^k$$

con $a, b, k \in \mathbb{R},\ a \geqslant 1,\ b \geqslant 2,\ k \geqslant 0$.

Aplicamos el cambio de variable $n = b^m$, y tenemos que:

$$T(b^m) = aT\left(b^{m-1}\right) + b^{km} = aT\left(b^{m-1}\right) + (b^k)^m$$

Aplicamos el cambio de variable $T(b^m) = x^m$, en la parte homogénea tenemos que:

$$x^m - ax^{m-1} = 0 \Longrightarrow x^{m-1}(x - a) = 0$$

Como $x^{m-1} > 0$ para todo $m \in \mathbb{N}$, se tiene que la única solución del polinomio caractertístico es $x = a$, con multiplicidad simple. Añadiendo la parte no homogénea, se tiene que $(b^k)^m = (b^k)^m \cdot (m^0)$, por lo que el polinomio caractertístico de la ecuación en diferencias es:

$$p(x) = (x - a)(x - b^k)$$

Por tanto, la solución general de la ecuación en diferencias es:

$$T(b^m) = x^m = c_1 \cdot a^m + c_2 \cdot b^{km}$$

Deshaciendo el cambio de variable $m = \log_b n$, se tiene que:

$$T(n) = c_1 \cdot a^{\log_b n} + c_2 \cdot b^{k \log_b n} = c_1 \cdot n^{\log_b a} + c_2 \cdot n^k \in O\left(n^{\max\{k, \log_b a\}}\right)$$