Algoritmos y estructuras de datos

www.u-tad.com

Grado en Ingeniería en Desarrollo de Contenidos Digitales

Introducción

- Algoritmo
  - Método o proceso para resolver un problema (generalmente iterativo)
  - Conjunto de reglas para efectuar algún cálculo
    - Las reglas deben ser objetivas (no subjetivas) y claras
    - Bien sea a mano o, más frecuentemente, en una máquina
- Algoritmia
  - Ciencia que estudia el concepto, diseño y construcción de algoritmos de forma eficiente
- Estructura de datos
  - Conjunto de datos, relacionados entre sí, que tienen un propósito y uso común
  - Una estructura de datos viene siempre acompañada por varios algoritmos para manipularla
  - Una clase es una estructura de datos (conjunto de sus atributos) más los algoritmos para manipularla (conjunto de métodos)

Ejemplo: número racional

- Un ejemplo típico de estructura de datos (**actividad 2.1**) es un número racional (una fracción)
  - Datos elementales que contiene la estructura de datos: numerador (tipo entero) y denominador (tipo entero)
  - Algoritmos para manipular la estructura de datos:
    - De creación de la estructura de datos, a partir de dos enteros
      - Es lo que comúnmente se llama “Constructor”
    - De creación de un nuevo racional a partir de otros racionales
      - Ej: sumar, multiplicar
    - De obtención de información de la estructura
      - Ej: getNumerador, getDenominador
    - De modificación de la estructura
      - Ej: simplificar. Se haría calculando el máximo común divisor del denominador y el denominador, por ejemplo mediante el algoritmo de Euclides (https://www.youtube.com/watch?v=O2_GOVf0vJ8). Luego se divide el numerador y el denominador entre ese máximo común divisor.
    - De conexión con una interfaz
      - Ej: escribir (en pantalla)
      - Viola los principios de separar interfaz del modelo, por lo cual una estructura de datos no debería incluir este tipo de algoritmos de conexión con la interfaz
        - En su lugar, otras clases o métodos (ej: main) se encargarían de esto. Ej: para escribir por pantalla el racional, main pediría el numerador y el denominador a un objeto de la clase Racional, y a continuación el main imprimiría estos datos por pantalla según un determinado formato

Eficiencia de los algoritmos I

- Eficiencia de un algoritmo
  - Un algoritmo es más eficiente cuanto menos recursos (tiempo y memoria) emplee para la resolución de su tarea, en relación al tamaño de sus datos de entrada
    - Ejemplo: ¿qué es más eficiente en tiempo? Dicho de otra manera: ¿cuál de los dos consigue ordenar una lista más grande en el mismo tiempo?
      - Un algoritmo que ordena una lista de 500 números en 1000 minutos
      - Un algoritmo que ordena una lista de 50 números en 200 minutos

Eficiencia de los algoritmos II

- Los ordenadores son cada vez más potentes, ¿por qué entonces importa tanto la eficiencia?
  - Las ambiciones crecen más rápido que la potencia del hardware, los problemas a resolver son cada vez más complejos
    - Ej: cuanto más ancho de banda y rapidez tenemos en Internet, más vídeos y de más calidad vemos
    - Ej: calcular la trayectoria para ir a la Luna (años 60) es mucho menos complejo que calcularla para ir a Saturno
  - Si el algoritmo es ineficiente, aumentar la velocidad del ordenador no es muy significativo en cuanto a la reducción de tiempo
    - Luego explicaremos por qué

Eficiencia de los algoritmos III

- La eficiencia de los programas no debería entrar en conflicto con los principios de la Ingeniería del Software
  - Los programas deben seguir siendo legibles y bien estructurados, siguiendo máxima cohesión y mínimo acoplamiento
  - Ahorrar algunas variables no hace mucho más eficiente el algoritmo, pero sí disminuye mucho la legilibidad
  - Igualmente, poner menos líneas de código y hacerlas más densas disminuye mucho la legibilidad, pero no aumenta sustancialmente la eficiencia

Eficiencia de los algoritmos IV

- El tiempo de ejecución de un programa depende de:
  - Lo bien hecho que esté el algoritmo base del programa
  - Potencia de hardware
  - El tamaño de los datos de entrada
    - Ej: número de componentes de un array que queremos ordenar
  - La complejidad de los datos de entrada
    - No tardamos lo mismo en ordenar un vector que ya está ordenado (caso mejor), que uno que está ordenado al revés (caso peor), o que uno que está ordenado aleatoriamente (caso medio)

Eficiencia de los algoritmos V

- En cuanto a la complejidad de la entrada:
  - Caso peor
    - Si no nos dicen nada, calculamos la eficiencia en el caso peor
    - Lo normal en cualquier Ingeniería es ponernos en el caso peor
  - Caso medio
    - Habría que calcular la media ponderada de todas las posibles entradas
    - Se necesita, por lo tanto, saber la probabilidad de ocurrencia de cada una, información que no siempre se tiene

Eficiencia de los algoritmos VI

- ¿La memoria es importante?
  - La eficiencia puede ser en tiempo o en memoria
  - Actualmente nos importa mucho más el tiempo que la memoria…
  - … porque hay memoria de sobra (es barata), y se puede conseguir más fácilmente juntando ordenadores, de forma proporcional al número de ordenadores juntados
    - No obstante, juntando ordenadores no siempre se consigue reducir el tiempo de computación en la misma proporción que el número de ordenadores que se juntan

Eficiencia de los algoritmos VII

- ¿Cómo decidir el mejor algoritmo para un problema? Para un mismo problema es posible que haya disponibles varios algoritmos para resolverlo… ¿cuál elegimos?
  - Si solamente tenemos que resolver uno o dos casos pequeños de un problema más bien sencillo, o bien el programa se va a ejecutar pocas veces, podríamos seleccionar el algoritmo más sencillo de programar, o aquél para el que ya exista un programa
  - Pero si tenemos que resolver muchos casos, o el problema es difícil, o el tamaño de la entrada puede llegar a ser muy grande, tendremos que seleccionar de forma más cuidadosa, generalmente el más eficiente
  - Un algoritmo complicado puede no ser conveniente si se necesita mantenimiento (algo que casi siempre ocurre… ¡el 80% del gasto mundial en desarrollo se emplea en mantenimiento!)
  - En los algoritmos numéricos (ej: calcular una integral), la precisión es más importante que la eficiencia
  - Si reducir tiempo y reducir memoria entran en contradicción, elegiremos de forma general el reducir tiempo a costa de aumentar la memoria

Enfoques de decisión I

- Enfoque empírico o «a posteriori»
  - Consiste en programar en la misma máquina los diferentes algoritmos candidatos y elegir el que menos tiempo o memoria gaste (**actividad 2.2**)
  - No es un enfoque muy adecuado porque:
    - Requiere un esfuerzo de programación
      - Lo ideal sería que pudiéramos elegir sólo con el pseudocódigo, sin necesidad de programar nada
    - La ejecución de los algoritmos puede durar mucho
    - Incluso la misma máquina nos puede dar distintos resultados según su carga actual, si estamos en un entorno multitarea

Enfoques de decisión II

- Enfoque teórico (a priori)
  - Determinar matemáticamente la cantidad de recursos necesarios para el algoritmo como función del tamaño de los casos de entrada
  - La ventaja es que no depende de la computadora que se use, ni del lenguaje de programación, ni de las habilidades del programador
  - A partir de ahora nos centraremos en este enfoque

Cálculo de T(n) I

- T(n) será la “función de tiempo” de un algoritmo. Nos dice el tiempo de ejecución del algoritmo:
  - Con una entrada de tamaño n, asumiendo que n es muy grande (que es lo que nos interesa)
  - En el caso peor en cuanto a la complejidad de la entrada
  - El tiempo no será “exacto”, ni será medido en segundos u otra medida…
    - Más adelante aclararemos esto
  - Es un cálculo “a priori”. No es necesario programar ni ejecutar el algoritmo.
- T(n) > 0 para todo valor de n, ya que un algoritmo no puede tardar un tiempo negativo en ejecutarse
- T(n) siempre va a ser una suma de términos
  - Ej: un algoritmo concreto tiene una función de tiempo de $T(n)=\log(n)+3n^2+8n^3$

Cálculo de T(n) II

- Cada “operación elemental” tiene un T(n)=1
- Una “operación elemental” es aquélla operación cuyo tiempo de ejecución no depende de “n”
  - De forma más precisa, una operación cuya duración esté acotada superiormente por un valor constante, independientemente de lo grande o pequeño que éste sea
- Las operaciones elementales suelen ser:
  - Sumas, restas, multiplicaciones, divisiones, módulo, operaciones booleanas, comparaciones, asignaciones a variables o a posiciones de un array, leer de una variable o posición de un array, saltos en el código, llamadas a funciones, terminación de funciones, declaraciones, reservas de trozos de memoria de cualquier tamaño (malloc, new)…

Cálculo de T(n) III

- Cuando tenemos varias líneas de código seguidas en un bloque, se suman los T(n) de cada una
- ¿Cómo se calculan las constantes?
  - En el resultado del bloque, si tenemos un sumando que sea alguna constante mayor que 1, se cambia por 1
    - Posteriormente veremos por qué
    - Ej: $T(n)=5n^2+3n+155$ estaría mal, ya que habría que cambiar el "155" por "1". Por lo tanto quedaría $T(n)=5n^2+3n+1$
  - Igualmente, si tenemos alguna resta del tipo 1 – c (siendo "c" una constante), queda 1, pues realmente el 1 puede ser un número mucho más grande que posteriormente hemos cambiado por un 1
- Ejemplo:
  - Hay multitud de operaciones elementales (declaraciones, asignaciones, sumas, multiplicaciones, llamadas a funciones y a métodos, etc.), pero la suma de todas ellas una constante mayor que 1, la cual cambiamos por 1
  - Por lo tanto, el resultado final del bloque es $T(n)=5n^2+3n+10\log(n)+1$

```
instruccionCompleja1(n); // Tiene T(n) = 5*n^2
int i = 5 + 10;
instruccionCompleja2(n); // Tiene T(n) = 2n + 10*logn
instruccionCompleja3(n); // Tiene T(n) = 1
int j;
j=i*funcionCompleja(n); // funcionCompleja tiene T(n)=n+1
```

Cálculo de T(n) IV

- El T(n) de una condición es el T(n) de la condición más el mayor de los T(n) de cada alternativa (cuando n es muy grande) más 1
  - Escogemos el T(n) más grande de las alternativas porque recordemos que estamos midiendo T(n) para el peor caso
  - El 1 es por el salto que implica cualquiera de las dos alternativas. Si se ejecuta el "if", al terminar hay que saltar hasta el final del "else". Si se ejecuta el "else", es necesario saltar hasta él desde la línea en donde está la evaluación de la condición
- Ejemplo:
  - Condición tiene T(n) = n+1
  - Bloque "if" tiene T(n) = 2n+10log(n)+1
  - Bloque "else" tiene T(n) = $5n^2+1$
    - Su curva es más grande que el T(n) del "if" cuando n es muy grande
  - Por lo tanto, el total es T(n) = $5n^2+n+1$

```
// funcionCompleja tiene T(n)=n+1
if (funcionCompleja(n) == a*b) {
    instruccionCompleja2(n); // Tiene T(n) = 2n + 10*logn
    instruccionCompleja3(n); // Tiene T(n) = 1
    int j = 10 * a + b;
}
else {
    instruccionCompleja1(n); // Tiene T(n) = 5*n^2
    int i = 5 + 10;
}
```

Cálculo de T(n) V

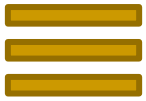
- En los bucles while:
  - Se multiplica las vueltas que da el bucle en el peor caso…
  - … por la suma de la T(n) de la evaluación, la T(n) de cada iteración, y un 1 por el salto hasta la línea en donde está la condición inicial, para volver a evaluarla
  - … y, al resultado anterior, se le suma la evaluación de la última condición, que no se cumple y por lo tanto hace que se termine el bucle
  - … y finalmente se suma 1 por el salto desde la línea en donde está la condición que no se cumple hasta el final del while, para seguir ejecutando las instrucciones que haya después del bucle
- Ejemplo:
  - La condición es T(n)=1 porque es una comparación
  - Cada iteración es $T(n)=5n^2+2n+10\log(n)+1$ (incluyendo el salto hasta el inicio)
  - Se ejecutan n iteraciones siempre
  - Por lo tanto, el total es $T(n)=5n^3+2n^2+10n\log(n)+n+1$

```
// i es inicialmente 0
while (i < n) {
    instruccionCompleja2(n); // Tiene T(n) = 2n + 10*logn
    instruccionCompleja1(n); // Tiene T(n) = 5*n^2
    i++;
}
```

Cálculo de T(n) VI

- Un bucle “do-while” se puede convertir en un bucle “while”, con propósitos de evaluarlo:

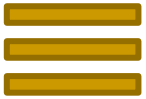
```
hacer
   bloque
mientras (condición)
```



```
bloque
mientras (condición)
   bloque
```

- Un bucle “for” también se puede convertir en un “while”:

```
for (inicio, condición, fin)
   bloque
```



```
inicio
mientras (condición)
   bloque
   fin
```

Cálculo de T(n) VII

- Hay que recordar que las vueltas que da el bucle siempre se calculan en el peor caso
- También hay que tener en cuenta que no siempre se puede calcular algo en función de “n”…
- Y hay que tener cuidado en contar las vueltas que da el bucle, teniendo en cuenta que, si se incluyen ambos límites en la variable que itera, hay que sumar 1 vuelta más
  - En general, es abs(fin-inicio)+1
  - Ejemplo: de 0 a 10, ambos inclusive, hay 11 números, no 10. Es 10-0+1 = 11.
  - Ejemplo: for (int i=20; i>=5; i--) da 16 vueltas porque 20-5+1=16
  - Ejemplo: for (int i=20; i>5; i--) da 15 vueltas porque el 5 no se incluye
- Ejemplo:
  - En el peor caso, siempre elemento < vector[posicionHueco]
  - Por lo tanto el bucle, en el peor caso, se ejecuta posicionHueco+1 veces
  - Como posicionHueco = posición-1, entonces el bucle da “posición” vueltas
  - Cada iteración tiene T(posicion)=1
  - Por lo tanto la función de tiempo final del bucle es T(posicion) = posicion+1

```
// posicionHueco = posicion-1;
while (posicionHueco >= 0 && elemento < vector[posicionHueco]) {
    vector[posicionHueco + 1] = vector[posicionHueco];
    posicionHueco--;
}
```

Cálculo de T(n) VIII

- También es posible que cada vuelta de un bucle tenga diferente T(n)
  - En este caso, hay que usar sumatorios
  - Todas las fórmulas de los sumatorios están en https://es.wikipedia.org/wiki/Sumatorio
  - En el cálculo del valor total del sumatorio, no tengas en cuenta las reglas de sustitución de constantes. Calcula el sumatorio tal cual. Sólo en el resultado es cuando tienes que aplicar las reglas.
    - En el cálculo interno del sumatorio no apliques los cambios de cualquier constante positiva por 1
    - En el cálculo interno del sumatorio tampoco apliques las restas 1-c=1
- Ejemplo:
  - El bucle del anterior ejemplo está dentro de otro bucle superior
  - En el ejemplo anterior hemos calculado que el bucle "while" tiene T(posicion)=posicion+1
  - Por lo tanto, cada iteración del bucle for es T(posicion)=posicion+1 también
  - Por lo tanto, el bucle while tiene:
    - T(n) = 1+$\sum_{posicion=1}^{n-1}(posicion+1)$ = 1+n-1+$\sum_{posicion=1}^{n-1} posicion$=
      $= 1 + n + \frac{(n-1)(n-1+1)}{2} = 1 + n + \frac{(n-1)n}{2} = 1 + n + 0{,}5n^2 - 0{,}5n = 0{,}5n^2 + 0{,}5n + 1$
  - Como vemos, al final la variable "posición" ha desaparecido de la función de tiempo, quedándonos únicamente con "n", que es el auténtico parámetro del algoritmo

```
for (posicion = 1; posicion <= n - 1; posicion++) {
    elemento = vector[posicion];
    posicionHueco = posicion - 1;
    while (posicionHueco >= 0 && elemento < vector[posicionHueco]) {
        vector[posicionHueco + 1] = vector[posicionHueco];
        posicionHueco--;
    }
    vector[posicionHueco + 1] = elemento;
}
```

Cálculo de T(n) IX

- En cuanto a las funciones:
  - Llamar a la función implica un salto hasta el lugar de la memoria en donde empiezan sus instrucciones
    - Este salto es una operación elemental y se computa en el bloque desde el que se llama a la función
  - Cuando la función termina, hay un salto hasta el lugar del código desde que se llamó a la función
    - Este salto es una operación elemental y se computa cuando calculamos el T(n) interno de la función
      - Por lo tanto la T(n) de cualquier función tendrá un +1 al final
      - Por lo tanto, démonos cuenta que el T(n) de funcionCompleja1() y funcionCompleja2() nunca podría ser el que hemos puesto en los ejemplos
  - Al comienzo de la función, se declaran los parámetros formales y se copia en ellos los parámetros reales con los que la función se llama realmente en ese caso
    - Las declaraciones y las copias generalmente serán operaciones elementales, aunque podría no ser así si los parámetros son objetos complejos con constructores y constructores de copia complicados, los cuales podrían tener un tiempo de ejecución que dependiera de n
  - Igualmente, cuando la función devuelve un valor, realmente el valor se copia y es esta copia la que se devuelve
    - Como antes, la copia será casi siempre una operación elemental, pero podría no ser así si implica una copia de un objeto complejo

Cálculo de T(n) X

- También es posible calcular la función de tiempo en función de varios parámetros en vez de uno solo (**actividad 2.3**)
  - Ej: $T(a,b) = 5a^3+2b^2+10b\log(n)+ab+a+1$
  - Las reglas son las mismas que hemos visto
- En general, la función de tiempo de un algoritmo se calcula en función de todos aquéllos parámetros que vemos que influyen en su tiempo de ejecución
  - Tendremos que deducir en cada caso qué parámetros influyen y cuáles no
  - Ten en cuenta que el algoritmo puede ser un método dentro de una clase
    - En este caso, los atributos de la clase también pueden influir en el tiempo que tarda el algoritmo en ejecutarse
      - Ej: “n” podría ser un atributo del objeto, y no necesariamente un parámetro del método

Orden de un algoritmo I

- ¿Qué T(n) son más grandes (y por lo tanto peores) cuando “n” es grande?
  - A continuación se muestra una lista, ordenada de los T(n) más rápidos a los más lentos
  - “c” es una constante cualquiera

Tipo de T(n)	T(n)
Constante	T(n) = 1
Logarítmico	T(n) = log(n)
Lineal	T(n) es un polinomio de grado 1
Cuadráticos	T(n) es un polinomio de grado 2
Cúbicos	T(n) es un polinomio de grado 3
Polinómico	T(n) es un polinomio de grado k
Exponencial	$T(n) = c^n$
Factorial	T(n) = n!
Doblemente exponencial	$T(n) = n^n$

Orden de un algoritmo II

- Según vimos, si el algoritmo es ineficiente, aumentar la velocidad del ordenador no es muy significativo en cuanto a la reducción de tiempo
- Ej: veamos lo que ocurre si multiplicamos por 10 la velocidad de un ordenador (mediante mejoras hardware), ante distintos algoritmos
  - Multiplicar por 10 la velocidad de un ordenador equivale a dejarle 10 veces más tiempo para resolver un problema

T(n) del algoritmo	n conseguida en un tiempo de $T(n)=10^3$ sg	n conseguida en un tiempo de $T(n)=10^4$ sg	Aumento de la n conseguida al multiplicar por 10 el tiempo
$100n$	10	100	900%
$5n^2$	14	45	221%
$n^3/2$	12	27	125%
2^n	10	13	30%

Orden de un algoritmo III

- El orden de un algoritmo es:
  - Su consumo “a priori” de tiempo o de memoria adicional
  - En función del tamaño de los operandos
  - Poniéndonos en el caso peor
  - Cuando n tiende a infinito
    - Cuando el tamaño de los operandos es pequeño, hemos dicho que nos da igual escoger un buen o mal algoritmo
    - Además nos interesa pensar en operandos grandes porque hemos dicho que los problemas son cada vez más complejos

Orden de un algoritmo IV

- Asumiendo que la función T(n) es una suma de términos, el "orden" de T(n) es su término mayor (según la tabla anterior), sin su constante multiplicativa (si la tiene)
  - Ej: si tenemos $T(n) = \log(n)+3n^2+8n^3$, decimos que es de orden n^3 (orden polinómico de grado 3)
  - Ej: si tenemos $T(n) = \log(n)+3n^2+8n^3+2^{2n}$, decimos que es de orden 4^n (orden exponencial)
- Esto es así porque asintóticamente (es decir, cuando n tiende a infinito o es muy grande) el resto de términos y las constantes multiplicativas son despreciables
  - Ahora entendemos por qué, al calcular T(n), cambiábamos todas las constantes por 1… pues en el infinito es igual de despreciable 1000 que 1.

Orden de un algoritmo V

- Si el orden de una T(n) de un algoritmo es una función g(n), decimos que $T(n) \in O(g(n))$
  - Decimos que el algoritmo es de orden temporal O(g(n))
  - También podemos decir que el algoritmo tiene una complejidad computacional en tiempo de O(g(n))
  - O, más abreviadamente, podemos decir que la complejidad temporal del algoritmo es O(g(n))
  - O(g(n)), matemáticamente, es el conjunto de funciones cuyo orden es g(n). Por eso decimos que $T(n) \in O(g(n))$
- Ejemplos:
  - Ej: si tenemos un algoritmo con $T(n) = \log(n)+3n^2+8n^3+1$, decimos que su complejidad temporal es $O(n^3)$
  - Ej: si tenemos un algoritmo con $T(n) = \log(n)+3n^2-8n^3+8\cdot 2^{2n}+1$, decimos que su complejidad temporal es $O(4^n)$
  - Ej: si tenemos un algoritmo con T(n) = 1, entonces decimos que su complejidad temporal es O(1)

Orden de un algoritmo VI

- La “implementación de un algoritmo” es llevarlo a un lenguaje, sistema operativo y máquina concreta
- Definición del “Principio de invarianza”:
  - Los valores de las constantes de T(n) sólo dependen de la implementación del algoritmo, y no del algoritmo en sí mismo
    - Estas constantes se llaman “constantes ocultas”, porque son despreciables en el infinito y por lo tanto no nos interesan, por lo tanto no aparecen en el O(…), están “ocultas”
- Por lo tanto, ¿en qué influye mejorar la implementación (ej: mejorar el hardware)?
  - Obviamente nos permite resolver un problema más rápido, pues las constantes son menores
  - Sin embargo, cuando n es muy grande o tiende a infinito:
    - Sabemos que las constantes son despreciables…
    - … y, por lo tanto, la implementación concreta es despreciable

Ejemplos de orden temporal

- Cálculo de determinantes
  - Basado en el teorema de Laplace: O(n!)
  - Por eliminación de Gauss-Jordan: $O(n^3)$ (muchísimo mejor)
- Ordenación:
  - Algoritmo de selección: $O(n^2)$
  - Algoritmo QuickSort: $O(n \cdot \log n)$
  - Si los componentes del vector están acotados: O(n)
- Operaciones matemáticas:
  - Multiplicar dos números muy grandes (de “m” y “n” cifras respectivamente): $O(m \cdot n)$
  - Sucesión de Fibonacci (0,1,1,2,3,5,8,13,21…): O(i) si se quiere calcular el elemento i-ésimo
- Transformada de Fourier (típico algoritmo que se usa constantemente para telecomunicaciones e inteligencia artificial):
  - Algoritmo clásico: $O(n^2)$
  - Transformada rápida de Fourier: $O(n \cdot \log n)$
  - En algunos casos particulares, se llega a la increíble $O(n/\log n)$

Complejidad espacial I

- Hasta ahora hemos calculado la complejidad temporal de un algoritmo
  - Es decir, el tiempo que tarda cuando el tamaño de su entrada tiende a infinito
- Pero no podemos olvidar el otro gran recurso (aunque sea menos importante que el tiempo) por el cual se mide la eficiencia de los algoritmos: la memoria que se usa

Complejidad espacial II

- Cuando medimos la memoria que usa un algoritmo, nos referimos siempre a la “memoria adicional”
  - Es la memoria extra máxima que el algoritmo necesita en algún momento de su ejecución
  - En “extra” no incluimos la memoria que, de por sí, ocupan los parámetros inicialmente
  - Tampoco incluímos la memoria que ocupe el objeto sobre el que se llama al algoritmo, en el momento inicial del mismo
    - Por lo tanto el constructor de una clase VectorEnteros tendría una complejidad espacial de O(n), pues el array interno de VectorEnteros no existía al empezar a ejecutar el constructor

Complejidad espacial III

- La función M(n) es la homónima a T(n), pero mide la memoria adicional máxima que necesita el algoritmo, cuando n es muy grande, en vez del tiempo de ejecución
- Para calcular M(n), se siguen reglas similares a T(n), con las siguientes salvedades:
  - Ahora estamos contando la memoria extra que se crea o usa, no el tiempo
  - Por lo tanto, por muchas vueltas que dé un bucle, la memoria usada no crece a no ser que en cada iteración se reserve memoria nueva y no se destruya la creada en la iteración anterior
  - Si una instrucción o varias crean o usan una cantidad de memoria acotada superiormente por una constante que no depende de n, entonces ponemos un 1
    - … similar a lo que hacíamos con T(n) cuando ejecutábamos varias operaciones elementales
    - Por ello, el hecho de declarar muchas variables sencillas no influye cuando n es muy grande, y por eso ponemos un 1 en M(n)

Complejidad espacial IV

- De la misma forma que calculábamos el orden de T(n), también calculamos el orden de M(n)
- Si el orden de una M(n) de un algoritmo es una función g(n):
  - Decimos que el algoritmo es de orden espacial O(g(n))
  - O que el algoritmo tiene una complejidad computacional en memoria adicional de O(g(n))
  - O que la complejidad espacial del algoritmo es O(g(n))
- Ejemplos:
  - Ej: si tenemos un algoritmo con $M(n) = \log(n)+3n^2+8n^3+1$, decimos que su complejidad espacial es $O(n^3)$
  - Ej: si tenemos un algoritmo con $M(n) = \log(n)+3n^2-8n^3+8\cdot 2^{2n}+1$, decimos que su complejidad espacial es $O(4^n)$
  - Ej: si tenemos un algoritmo con M(n) = 1, entonces decimos que su complejidad espacial es O(1)

Complejidad espacial V

- Algunos ejemplos (**actividad 2.4**):
  - Ej: el algoritmo de ordenación por selección
    - Sólo necesita memoria adicional acotada por una constante (unas 5 ó 6 variables sencillas)
    - Por lo tanto no necesita memoria adicional cuya cantidad dependa de n
    - Por lo tanto su complejidad espacial es O(1)
  - Ej: tenemos el siguiente algoritmo de ordenación:
    - Crea internamente un array temporal de tamaño n
    - Va poniendo en él sucesivamente los mínimos encontrados en el vector pasado como parámetro
    - Finalmente copia los valores del array temporal al vector pasado como argumento, y borra el array temporal
    - Por lo tanto tiene una complejidad espacial de O(n)