Análisis de algoritmos

Prof. Enrique Vílchez Quesada

Universidad Nacional de Costa Rica

Introducción

El tema de análisis de algoritmos pretende ofrecer al lector, una serie de recursos de naturaleza experimental y teórica, por medio de los cuales se da cabida a la comparación de procedimientos que resuelven un mismo problema. El análisis de algoritmos es un tema interesante que procura brindar herramientas para estudiar el esfuerzo que hace un programa cuando al tener n datos de entrada proporciona un conjunto de datos de salida.

- El análisis de la complejidad de un algoritmo se puede efectuar midiendo los tiempos de ejecución arrojados localmente por un ordenador, al ejecutar cierta cantidad de veces un programa que representa la implementación del algoritmo en algún lenguaje de programación (enfoque experimental)
- También, al encontrar una función $f(n)$ que calcula ese tiempo de ejecución al procesar n datos de entrada en el peor, mejor o caso promedio y posteriormente, al acotar $f(n)$ con otras funciones más simples que la delimitan superior o inferiormente (enfoque teórico).

Nota

La complejidad o análisis de algoritmos busca, por lo tanto, observar cuál sería el comportamiento de un programa en situaciones extremas.

Enfoque experimental

Prof. Enrique Vílchez Quesada

Universidad Nacional de Costa Rica

Enfoque experimental

- La idea de este enfoque para comparar varios algoritmos que resuelven un mismo problema, consiste en poner a correr en un ordenador sus implementaciones asociadas a un lenguaje de programación, midiendo el tiempo que tarda cada uno en ofrecer una salida.

Enfoque experimental

Nota

Se considera un enfoque experimental pues la toma de decisión de cuál método se comporta mejor, se sustenta en seleccionar el algoritmo que muestra un menor tiempo de ejecución la mayor cantidad de experimentos realizados.

Enfoque experimental

- Es decir, los experimentos se traducen en efectuar varias invocaciones sobre los algoritmos a analizar, se miden los tiempos que tardan en devolver un resultado y se contabiliza el número de veces en las que cada uno, se demoró menos. La intención, en este sentido, reside en visualizar una tendencia sobre cuál de los prodemientos constituye un algoritmo más rápido o veloz y por consiguiente, más eficiente en tiempo de ejecución.
- El paquete **VilCretas** facilita el empleo del enfoque experimental mediante dos instrucciones llamadas: `PruebaADA2` y `PruebaADA3`. Consideremos algunos ejemplos.

Example (3.1)

Construya tres métodos S1, S2 y S3 para calcular:

$$S_n = \sum_{i=2}^{n-3} \frac{(-1)^{i+1}}{3\,(i-1)}$$

Donde S1 se debe basar en una lógica recursiva, S2 en un proceso iterativo y S3 en el uso del comando `Sum` de *Wolfram Mathematica*. Determine, experimentalmente, cuál de ellos es más adecuado en términos de tiempo de ejecución.

Solución

Reutilizando las ideas compartidas en el capítulo 1 de este texto, una relación de recurrencia que representa a S_n corresponde a:

$$S_n = S_{n-1} + \frac{(-1)^{n-2}}{3\,(n-4)} \text{ con } S_5 = -\frac{1}{3}\ \forall n,\ n \in \mathbb{N},\ n \geq 5 \tag{1}$$

De 1, se infiere el programa recursivo de cola S1:

In[ ] :=

```
S1[n_, suma_ : 0]:=If[n == 5, -1/3 + suma, S1[n - 1,
suma + (-1)^(n - 2)/(3 (n - 4))]]
```

Solución

También, se pudo haber pensado en una recursividad de pila en lugar del método anterior. Por otro lado, si se desea programar S_n con una estructura de control de repetición (proceso iterativo), por ejemplo, usando un ciclo `For`, en *Mathematica* se procede así:

In[ ] :=

```
S2[n_]:=Module[{suma = 0, i}, For[i = 2, i <= n - 3,
suma = suma + (-1)^(i + 1)/(3 (i - 1)); i++]; suma]
```

Nota

Se observa que el incremento del parámetro `i` que controla el ciclo `For`, se coloca al final de la estructura, aspecto distintivo al compararlo con otros ambientes de programación.

Solución

Finalmente, para S3 se define la función:

In[ ] :=

```
S3[n_]:=Sum[(-1)^(i + 1)/(3 (i - 1)), {i, 2, n - 3}]
```

Empleando un `Table` de *Wolfram*, se puede corroborar en algunos casos particulares, que S1, S2 y S3 resuelven el cálculo de la sumatoria:

In[ ] :=

```
Table[S1[i] == S2[i] == S3[i], {i, 5, 20}]
```

Out[ ] =

{True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True}

Solución

En este punto, nos interesa determinar cuál de los tres métodos es más eficiente. Para ello, el paquete **VilCretas** provee la instrucción `PruebaADA3` que compara los tiempos de ejecución arrojados por tres funciones distintas, con la finalidad de agrupar por parejas el número de veces que tardaron menos tiempo en proporcionar una salida. En este ejercicio:

In[ ] :=

```
PruebaADA3[{S1, S2, S3}, 1500, 5]
```

Out[ ] =

El primer y segundo algoritmo fueron mejores: 130
El primer y tercer algoritmo fueron mejores: 174
El segundo y tercer algoritmo fueron mejores: 243
Se comportaron igual: 953

Solución

- El primer vector de `PruebaADA3` establece los identificadores de los programas a comparar. El parámetro 1500 especifica la cantidad de pruebas o experimentos a realizar y el último argumento 5, precisa el valor de inicio.
- Bajo esta perspectiva, `PruebaADA3` comienza evaluando las funciones `S1`, `S2` y `S3` en $n = 5$, midiendo el tiempo que tardó cada método en retornar un resultado e incrementando un contador asociado a los dos programas que se tornaron más veloces. Posteriormente, `S1`, `S2` y `S3` se evaluán en $n = 6$, repitiendo la selección de las dos funciones más rápidas en tiempo de ejecución e incrementando el contador correspondiente. El proceso se repite de forma sucesiva hasta llegar a la última evaluación en $n = 1500$.

Nota

La salida de `PruebaADA3` devuelve los valores acumulados en los tres contadores encargados de medir localmente la eficiencia por parejas. De hecho, muy probablemente, si el lector en su computadora presiona *shift+enter* sobre `PruebaADA3[{S1, S2, S3}, 1500, 5]`, los valores retornados por *Mathematica* serán diferentes a los ya compartidos. Esto obedece al hecho de que las mediciones de tiempo dependen directamente de las condiciones de hardware de la máquina que se está utilizando y de su estado actual, referido al uso de la memoria temporal y de las unidades de almacenamiento.

Solución

Sin cambiar nada, el alumno puede comprobar cómo varias ejecuciones de `PruebaADA3[{S1, S2, S3}, 1500, 5]`, devolverán mediciones de tiempo diferentes. Pese a ello, aunque en cada grupo de experimentos los contadores registrarán probablemente resultados distintos, lo que se busca en los retornos es observar una tendencia que muestre cuál par de algoritmos en la mayoría de los corrimientos realizados, se están comportando más apropiadamente (tardan menos).

Solución

En este ejemplo y luego de realizar varias ejecuciones de `PruebaADA3[{S1, S2, S3}, 1500, 5]`, se concluye que `S2` y `S3` son los más veloces. De alguna manera era predecible esta conclusión, dado que `S1` es un programa recursivo vinculado a una relación de recurrencia y aunque `S1` es de cola, siempre consume muchos recursos por parte del ordenador.

Solución

Si ahora se desea comparar las funciones `S2` y `S3` en tiempo de ejecución, la librería **VilCretas** posee el comando `PruebaADA2` cuyo funcionamiento es similar a `PruebaADA3`, solo que en este caso, `PruebaADA2` compara únicamente dos algoritmos (de allí el 2 que aparece al final del nombre de la setencia). Luego, en *Wolfram*:

In[ ] :=

```
PruebaADA2[{S2, S3}, 1500, 5]
```

Out[ ] =

El primer algoritmo fue mejor: 110
El segundo algoritmo fue mejor: 193
Se comportaron igual: 1197

Solución

Pese al comportamiento variable de la salida anterior, lo esencial reside en buscar una tendencia, tal y como se explicó con el comando `PruebaADA3`. Al ejecutar varias veces `PruebaADA2[{S2, S3}, 1500, 5]`, se infiere que el mejor algoritmo corresponde a `S3`.

Nota

Las instrucciones `PruebaADA2` y `PruebaADA3` están basadas en el empleo de un comando nativo de *Mathematica* llamado `Timing`. `Timing` calcula tiempos de ejecución en tiempo real. El lector debe comprender que aunque las sentencias `PruebaADA2` y `PruebaADA3` están elaboradas en lenguaje *Wolfram*, en otros ambientes de programación existen funciones similares a `Timing` o los recursos necesarios para poderlas programar. Desde este punto de vista, el enfoque experimental no es algo exclusivo de *Wolfram Language*, sino una forma de trabajo en análisis de algoritmos aplicable en cualquier otro lenguaje de programación.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-41.zip

Explicación en video

https://youtu.be/c7UcIXqVucM

Explicación en video

https://youtu.be/sLIh4CMhe7A

Example (3.2)

Verifique usando un `Table` que los métodos `CalculaProducto` y `CalculaOtroProducto` dados a continuación, retornan los mismos resultados. Compare experimentalmente ambos algoritmos.

```
CalculaProducto[n_]:=Module[{p=1}, While[p<=n, p=p*3];
Return[p]]
CalculaOtroProducto[n_]:=Module[{p=1}, For[i=1, i<=n,
While[p<=i, p=p*3]; i++]; Return[p]]
```

Solución

Al observar el código que caracteriza a las funciones `CalculaProducto` y `CalculaOtroProducto` se aprecia que la variable `p` retornada por ambas, acumula una potencia de 3. Usando la instrucción `Table` de *Mathematica* se puede corroborar en distintos valores de `n`, cómo `CalculaProducto` y `CalculaOtroProducto` devuelven el mismo resultado. Veamos:

In[ ] :=

```
Table[CalculaProducto[n] == CalculaOtroProducto[n], {n, 1,
20}]
```

Out[ ] =

{True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True}

Solución

El estudiante debe apreciar que, pese a la igualdad en las salidas de `CalculaProducto` y `CalculaOtroProducto`, ambos métodos se comportan diferente con relación a los tiempos de ejecución consumidos. `CalculaOtroProducto` itera mayor cantidad de veces al estar conformado por un `For` y por un `While` anidados, en comparación con `CalculaProducto` que solo tiene un ciclo `While` y por esta razón, `CalculaOtroProducto` gasta más tiempo al proporcionar un **Out[ ]**. El comando `PruebaADA2` permite comprobar lo anterior:

In[ ] :=

```
PruebaADA2[{CalculaProducto, CalculaOtroProducto}, 5000, 1]
```

Solución

Se obtiene la siguiete salida:

Out[ ] =

El primer algoritmo fue mejor: 575
El segundo algoritmo fue mejor: 7
Se comportaron igual: 4418
En `PruebaADA2` se han ejecutado cinco mil pruebas de tiempo de ejecución. Al correr varias veces `PruebaADA2[{CalculaProducto, CalculaOtroProducto}, 5000, 1]`, se corrobora la tendencia de asumir como un mejor método a `CalculaProducto`.

Nota

`CalculaProducto` y `CalculaOtroProducto` son funciones incorporadas en el paquete **VilCretas**, razón por la cual dentro del software *Mathematica*, no es necesario crear los métodos para ser utilizados en el comando `PruebaADA2`.

Solución

En ocasiones, la comparación de los tiempos de ejecución locales entre varios métodos, puede efectuarse de manera gráfica. El código expuesto usa la instrucción `RepeatedTiming` de *Wolfram*, que a diferencia del `Timing`, realiza el cálculo de un tiempo promedio de salida. Con `RepeatedTiming` se gesta una lista de tiempos promedio de salida para `CalculaProducto` y `CalculaOtroProducto`, donde luego, mediante el comando `ListLinePlot` de *Mathematica*, se traza una línea a través de ellos, construyéndose una gráfica de tiempos de ejecución para cada algoritmo analizado.

Solución

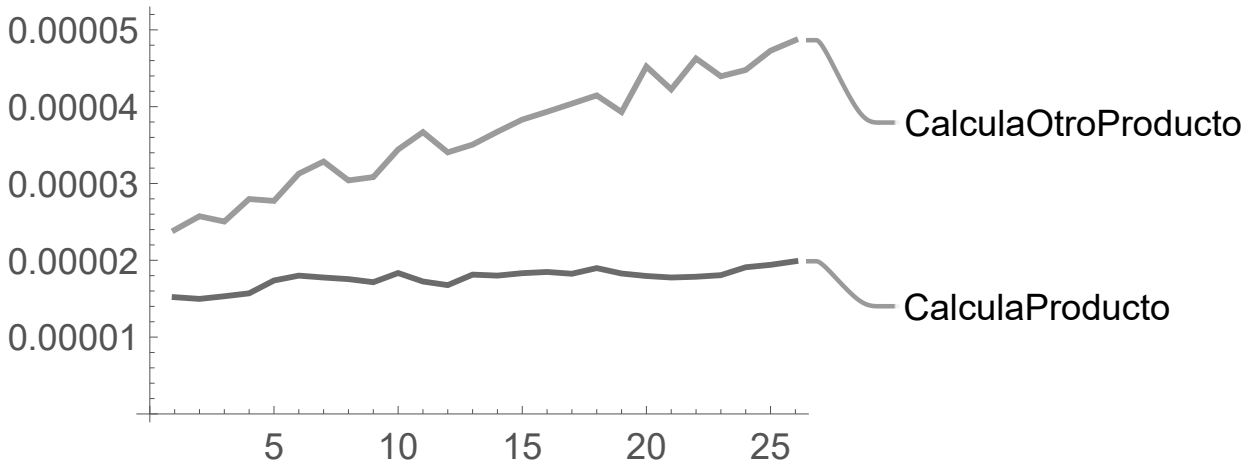
Veamos:

In[ ] :=

```
T1 = Table[RepeatedTiming[CalculaProducto[i]][[1]], {i, 5,
30}];
T2 = Table[RepeatedTiming[CalculaOtroProducto[i]][[1]], {i,
5, 30}];
ListLinePlot[{T1, T2}, PlotLabels -> {"CalculaProducto",
"CalculaOtroProducto"}]
```

Solución

Out[ ] =



En la gráfica se corrobora un menor crecimiento en los tiempos de ejecución para el método `CalculaProducto`, reflejando con ello, una mayor eficiencia.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-42.zip

Example (3.3)

Compare en tiempo de ejecución los programas recursivos `divisores` y `Otrodivisores` elaborados en el ejemplo 1.7 (Ejemplos de programas recursivos), que retornan una lista con todos los divisores de un entero positivo n.

Solución

Recordando la lógica aplicada en el método `Otrodivisores`, en él, se realizan como máximo $\frac{n}{2}$ llamadas recursivas, por lo que, intuitivamente debería tardar menos que la función `divisores`, al realizar esta última, n invocaciones. La sentencia `PruebaADA2` de la librería **VilCretas**, nos faculta a comprobarlo de forma experimental:

In[ ] :=

```
PruebaADA2[{divisores, Otrodivisores}, 500, 1]
```

Out[ ] =

```
El primer algoritmo fue mejor: 18
El segundo algoritmo fue mejor: 35
Se comportaron igual: 447
```

Solución

En este ejercicio, no es posible ejecutar un alto número de mediciones, pues de lo contrario, se da un desbordamiento en la pila de llamadas. Al correr varias veces `PruebaADA2[{divisores, Otrodivisores}, 500, 1]`, se torna una tendencia mayoritaria a concebir en `Otrodivisores`, una función más rápida.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-43.zip

Example (3.4)

Utilice los comandos `Factoriales` y `FactorialesCola` de **VilCretas**, para verificar una mayor eficiencia en la recursividad de cola, al calcular $n!$, $n \in \mathbb{N} \cup \{0\}$.

Solución

Al recurrir al uso de la instrucción `PruebaADA2`, se tiene:

In[ ] :=

```
PruebaADA2[{Factoriales, FactorialesCola}, 1000, 1]
```

Out[ ] =

El primer algoritmo fue mejor: 52
El segundo algoritmo fue mejor: 99
Se comportaron igual: 849

Se reitera al lector que el **Out[ ]** de `PruebaADA2` es variable, por lo que muy posiblemente obtendrá en el software *Wolfram Mathematica*, valores distintos a los mostrados con anterioridad.

Solución

Al ejecutar varias veces `PruebaADA2[{Factoriales, FactorialesCola}, 1000, 1]`, se observa una clara tendencia de considerar a `FactorialesCola` un método más veloz. También, en este ejercicio al igual que en el ejemplo 3, el número de mediciones no debe ser muy elevado, con el objetivo de evitar un desbordamiento.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-44.zip

Example (3.5)

Compare la eficiencia de los métodos de ordenación de datos “burbuja” e “inserción”.

Solución

- En el presente texto se asume que el alumno conoce de previo los métodos de ordenación de la “burbuja” e “inserción”.

Recordar el algoritmo de burbuja

- El algoritmo de la burbuja es un método que ordena una lista L de datos de forma ascendente o descendente. Se basa en tomar inicialmente la lista L y comparar el primer elemento con el segundo, si el primero es mayor que el segundo se intercambian y se continúa ahora comparando el segundo elemento con el tercero. Se sigue de esta misma forma hasta comparar con el último elemento de L.
- El proceso permite dejar el número mayor, o el que posee mayor peso, al final de la lista L. Lo anterior completa una iteración del algoritmo de la burbuja. La segunda iteración se produce al repetir el mismo procedimiento con la lista L, exceptuando su último elemento. En la tercera iteración, se continúa de manera similar con la lista L menos sus dos últimos términos ya ordenados.

Recordar el algoritmo de burbuja

- El procedimiento continúa hasta obtener una lista de longitud uno. Al finalizar, L contiene los datos ordenados ascendentemente (similar a una "burbuja en ascenso"). Se invita al estudiante a analizar, cómo por el método de la burbuja, se pueden ordenar un conjunto de datos de manera descendente.

Y así si ocurriera en otro caso

Solución

Por ejemplo, para ordenar la lista $L = \{9, 37, 19, 17\}$ por burbuja, se procede así:

Iteración 1: se toma toda la lista

$\underbrace{9\ 37}\ 19\ 17$ $9 \not> 37$
$\underbrace{9\ 37}\ 19\ 17$ No se invierten
$9\ \underbrace{37\ 19}\ 17$ $37 > 19$
$9\ \underbrace{19\ 37}\ 17$ Se invierten
$9\ 19\ \underbrace{37\ 17}$ $37 > 17$
$9\ 19\ \underbrace{17\ 37}$ Se invierten
$\underbrace{9\ 19\ 17\ 37}$ Salida

Iteración 2: se emplea toda la lista exceptuando a 37

$\underbrace{9\ 19}\ 17\ 37$ $9 \not> 19$
$\underbrace{9\ 19}\ 17\ 37$ No se invierten
$9\ \underbrace{19\ 17}\ 37$ $19 > 17$
$9\ \underbrace{17\ 19}\ 37$ Se invierten
$\underbrace{9\ 17\ 19\ 37}$ Salida

Iteración 3: se utiliza toda la lista exceptuando a 19 y 37

$\underbrace{9\ 17}\ 19\ 37$ $9 \not> 17$
$\underbrace{9\ 17}\ 19\ 37$ No se invierten
$\underbrace{9\ 17\ 19\ 37}$ Lista ordenada

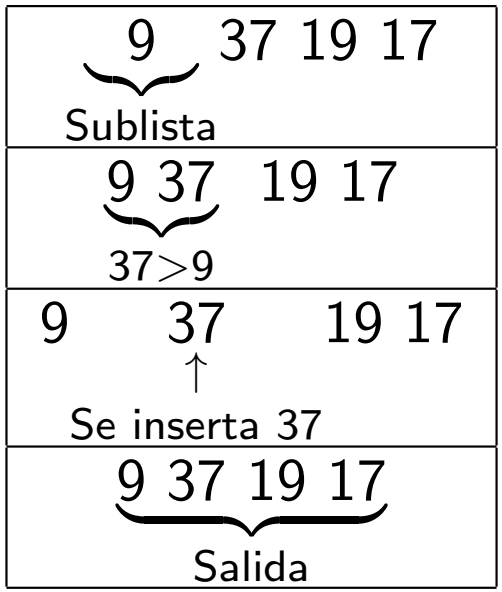
Solución

Por otra parte, el algoritmo de inserción consiste en "insertar" un elemento de la lista L, donde al comparar con los términos de una sublista ya ordenada, se detiene el cotejo hasta que ese elemento sea mayor, en ese instante se inserta, creando una nueva lista L. El proceso se repite hasta llegar al último elemento de L.

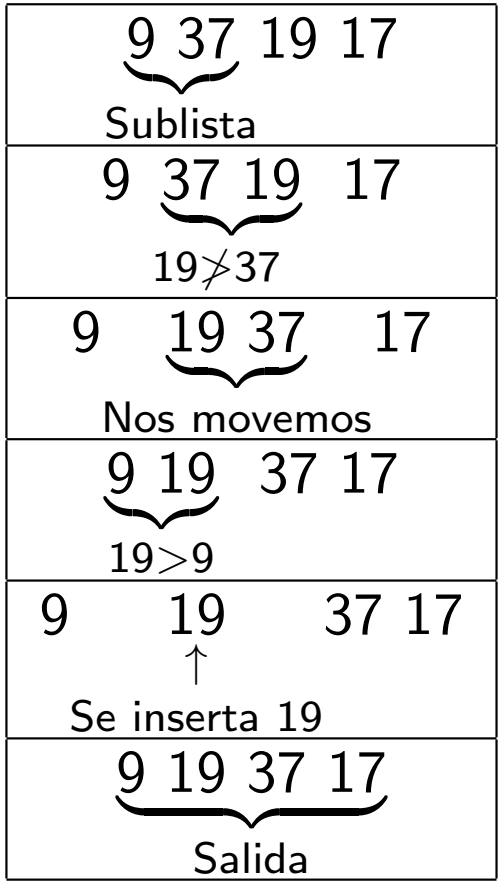
Solución

Retomando el ejemplo de $L = \{9, 37, 19, 17\}$, por inserción, se tendría:

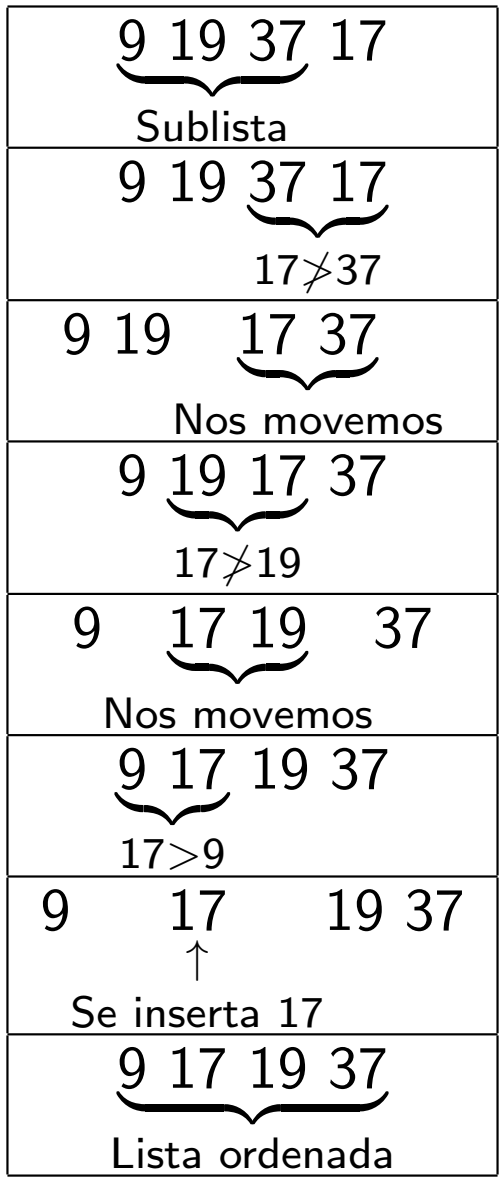
Iteración 1



Iteración 2



Iteración 3


9 19 37 17
Sublista
9 19 37 17
17≯37
9 19 17 37
Nos movemos
9 19 17 37
17≯19
9 17 19 37
Nos movemos
9 17 19 37
17>9
9 17 19 37
Se inserta 17
9 17 19 37
Lista ordenada

Solución

El paquete **VilCretas** trae incorporadas las sentencias `Burbuja` e `Insercion`, siendo estas funciones una implementación de los algoritmos de la burbuja e inserción, respectivamente. Ambas, contienen la opción `code -> True` por si se desea conocer cuál es su código interno de programación.

Nota

El lector debe tener en cuenta que el método de la burbuja realiza en ciertos casos, mayor cantidad de comparaciones, por ese motivo, el algoritmo de inserción se considera más eficiente en tiempo de ejecución.

Solución

Verificando esto mediante el comando `PruebaADA2`, se obtiene:

In[ ] :=

```
PruebaADA2[{Burbuja, Insercion}, 200, 5, 1, lista -> True]
```

Out[ ] =

El primer algoritmo fue mejor: 14
El segundo algoritmo fue mejor: 95
Se comportaron igual: 91

`PruebaADA2` como se aprecia en el **In[ ]**, dispone de la opción `lista -> True` necesaria para indicar que las funciones a analizar, emplean como argumento una lista en sustitución de un valor numérico.

Solución

En este ejercicio, *Wolfram* internamente construye listas pseudoaleatorias de números enteros, comenzando con una de longitud 5 y hasta el tamaño 200. Cada una de esas listas es pasada a las instrucciones `Burbuja` e `Insercion`, midiéndose los tiempos de ejecución al proporcionar una salida.

Descargue un archivo

`https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-45.zip`

- A continuación, se desarrollará el enfoque teórico para analizar la complejidad de un algoritmo. En el tema de análisis de algoritmos es usual tratar de buscar una función $f(n)$ que mida el nivel de esfuerzo en tiempo de ejecución, realizado por un programa al procesar n datos de entrada y a partir de ella, dar un "orden asintótico" al procedimiento.
- Lo anterior quiere decir que, se aproxima el comportamiento de la velocidad de convergencia de la salida del método, por medio de otras funciones más simples y conocidas, por lo general, funciones polinomiales o transcendentales (exponenciales y logarítmicas). Los órdenes de convergencia de un algoritmo se establecen mediante el uso de distintas notaciones llamadas en la literatura "notaciones asintóticas".

Enfoque teórico: Notación asintótica "*O* grande"

Prof. Enrique Vílchez Quesada

Universidad Nacional de Costa Rica

Notación asintótica O grande

- Las notaciones asintóticas constituyen un mecanismo notacional por medio del cual, una función se acota o delimita por encima de su gráfica, por debajo o ambas simultáneamente.

Tienen una relación directa con la complejidad de un algoritmo pues normalmente el análisis de un programa se realiza encontrando una función $f(n)$ que representa su nivel de esfuerzo en unidades de tiempo de ejecución, sea contando el número de comparaciones que realiza o la cantidad de iteraciones o llamadas que ejecuta en el peor, mejor o caso promedio. Al determinar esta función se compara con otras, usualmente: $n!$, n^m con m un número natural, a^n o $\log_a n$, $a \in \boldsymbol{R}^+$, $a \neq 1$.

Notación asintótica O grande

- Un ejemplo de este tipo de proceso, se presentó en el capítulo 2 cuando se determinó una relación de recurrencia que cuenta el número de pasos necesarios para resolver el juego de las torres de *Hanoi*. *Mathematica* nos devolvió como solución la función $f(n) = 2^n - 1 \ \forall n,\ n \in \mathbb{N}$.
- En este ejercicio, $f(n)$ representa el esfuerzo que realiza el algoritmo empleado para resolver el juego con n discos. Al observar la forma del criterio de f es natural concluir que una función que se encuentra por encima de su gráfica es $g(n) = 2^n$, pues $2^n - 1 < 2^n \ \forall n,\ n \in \mathbb{N}$. En la siguiente figura, se han graficado simultáneamente f y g donde se visualiza el comportamiento $f(n) < g(n)$, es decir, la gráfica de la función $g(n)$ se encuentra por encima de la gráfica de la función $f(n)$.

Notación asintótica O grande

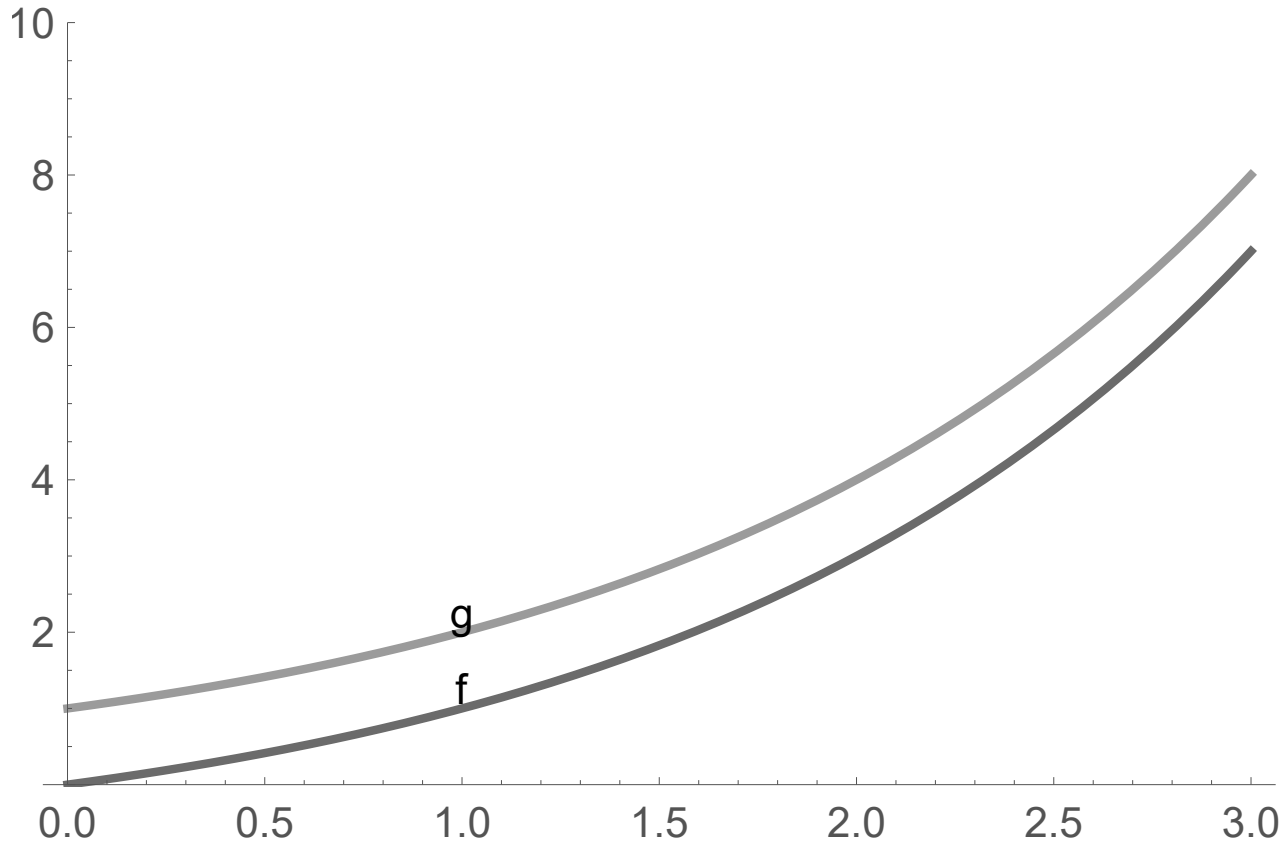


Figura: Comparación gráfica de $f(n) = 2^n - 1$ y $g(n) = 2^n$

Notación asintótica O grande

- Como g se encuentra por encima de f se dice que "acota superiormente" a la función f, o bien, que g es una "cota superior" de f. Esta tendencia asintótica se representa por medio de una notación denominada "O grande". Veamos la siguiente definición.

Definition (3.1)

Sean f y g funciones definidas sobre el conjunto de los números naturales. Se dice que $f(n)$ es de orden a lo sumo $g(n)$ y se denota $f(n) = O(g(n))$, si existe una constante real positiva c, tal que:

$$|f(n)| \leq c\,|g(n)| \;\; \forall n,\; n \in \mathbb{N}$$

o excepto un número finito de ellos. A la notación $O(g(n))$ se le llama notación asintótica "O grande".

Comentario sobre la definición 6

Como ya se observó en la figura 1, cuando f y g son funciones positivas en el conjunto de los números naturales (es decir, sus imágenes dan valores en $\boldsymbol{R}^{+}$), $f(n) = O(g(n))$ significa que f está por debajo de la gráfica de un múltiplo de g $\forall n$, $n \in \mathbb{N}$, o bien, sobre $\mathbb{N}$ exceptuando algunos de sus elementos. La exclusión "exceptuando algunos de sus elementos", se refiere a satisfacer la desigualdad $|f(n)| \leq c\,|g(n)|$ a partir de algún número natural, no necesariamente igual a uno.

Notación asintótica O grande

- Si una función $f(n)$ representa el número de unidades de tiempo necesarias que realiza un algoritmo para procesar n datos de entrada y se satisface $f(n) = O(g(n))$, se dice que su velocidad de convergencia es de orden a lo sumo $g(n)$ y se representa por medio de la notación asintótica $O(g(n))$.
- La notación asintótica $O(g(n))$ describe el nivel de eficiencia del algoritmo analizado. Los algoritmos más veloces son aquellos que poseen en el peor caso una notación asintótica O grande cuya gráfica se encuentra por debajo de la función identidad $y = n$. En este tipo de programas, cuando n toma valores muy grandes, el tiempo de ejecución representado a lo sumo por $g(n)$, no crece tan rápidamente como lo hace la cantidad n de datos procesados, ocasionando que el algoritmo funcione de manera efectiva al tener muchos datos de entrada.

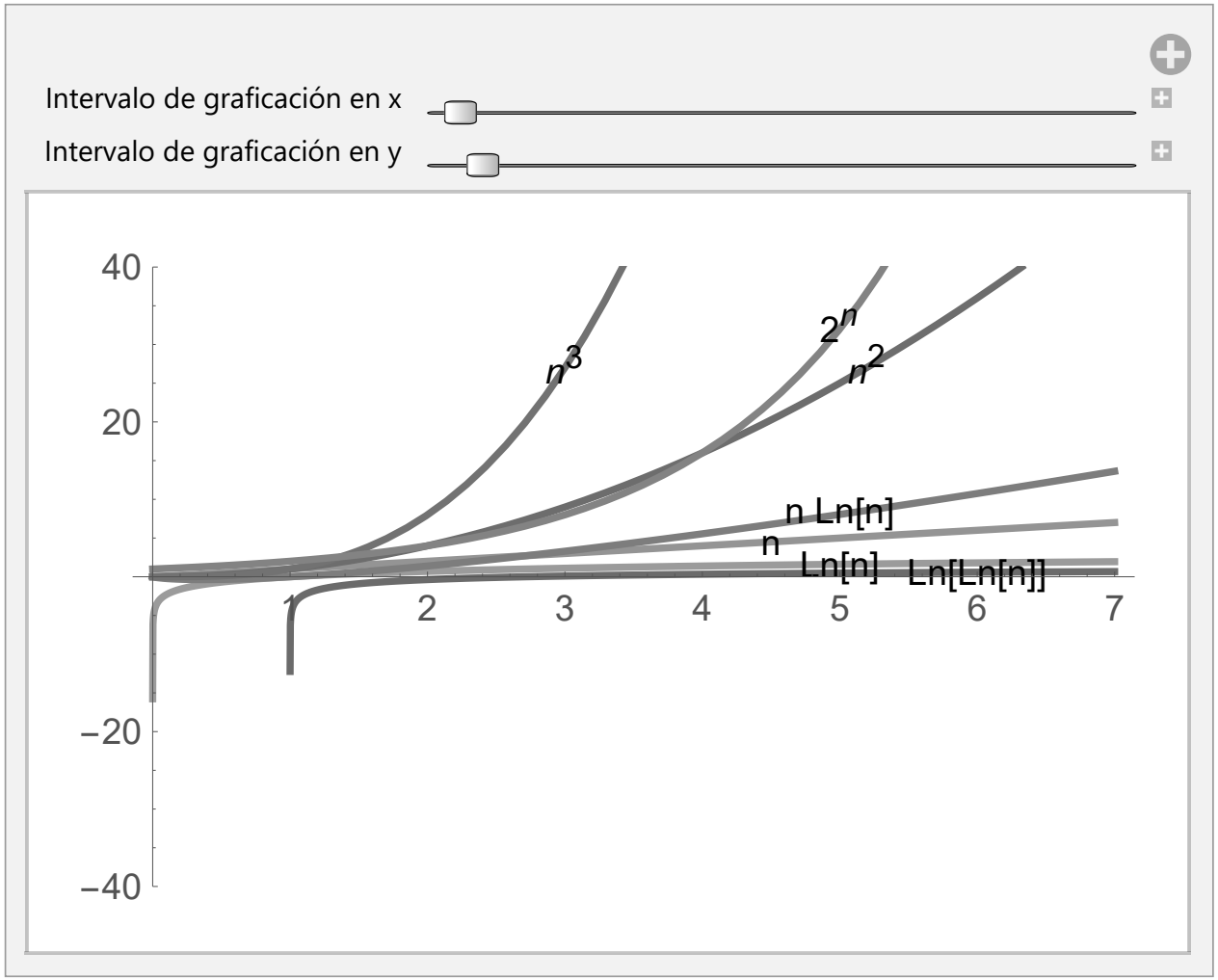
Notación asintótica O grande

- La instrucción `CDFGraficaNOG` de la librería **VilCretas**, facilita comparar, tomando diversos tamaños de n, el comportamiento asintótico de las funciones: $\ln(\ln(n))$, $\ln(n)$, n (función identidad), $n\ln(n)$, n^2, n^3 y 2^n :

In[ ] :=

```
CDFGraficaNOG[1500]
```

- El comando anterior produce la siguiente salida:
  Out[ ] =



- El argumento `1500` en `CDFGraficaNOG` especifica el valor máximo que se puede otorgar a los intervalos de graficación en los ejes coordenados. En otras palabras, si el usuario mueve los deslizadores de la animación, el número máximo posible es 1500. En esta representación, podría interpretarse erróneamente que n^3 es una cota superior de 2^n sobre todo el conjunto de los números naturales.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-46.zip

- Moviendo el deslizador en el eje y, se puede visualizar de la siguiente forma:

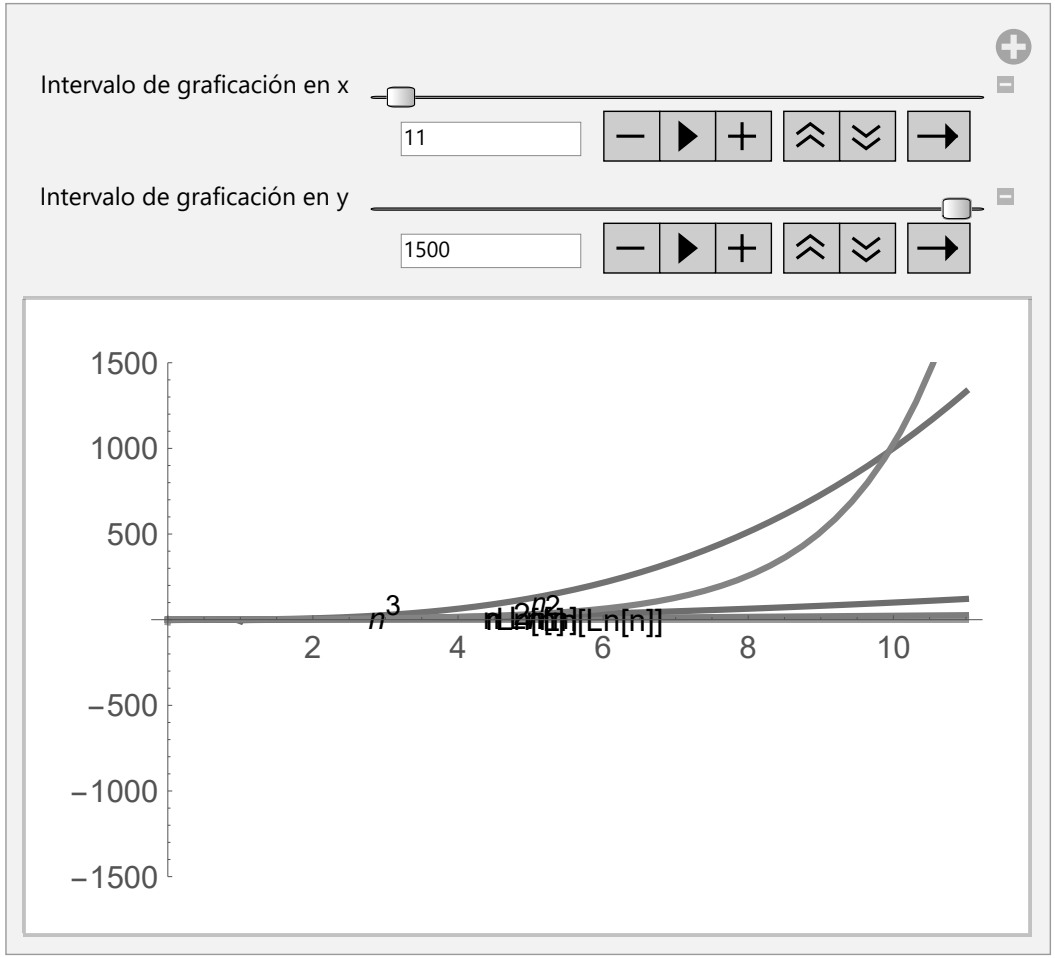


Figura: Funciones 2^n y n^3

- También, la animación generada por `CDFGraficaNOG` se puede reproducir por medio del siguiente *CDF*.

Descargue un archivo

`https://www.escinf.una.ac.cr/discretas/Archivos/CDFs/Notaciones_O_grande.cdf.zip`

Example (3.6)

Demuestre que $n^3 + n^2 + n + 2 = O\left(n^3\right)$.

Solución

La demostración de acuerdo con la definición 6, se fundamenta en encontrar una constante real positiva c, tal que:
$\left|n^3+n^2+n+2\right| \leq c\left|n^3\right| \ \forall n,\ n \in \mathbb{N}$ o exceptuando algunos de ellos.
Una técnica usual para acotar superiormente a una función $f\left(n\right)$ que corresponde a una suma, consiste en sustituir cada sumando por el mayor.
En este ejemplo, como $f\left(n\right)=n^3+n^2+n+2$, se debe reemplazar cada monomio n^j por $n^3\ \forall j,\ j \in \mathbb{N},\ 0 \leq j \leq 3$. Luego:

$$n^3+n^2+n+2=n^3+n^2+n^1+2n^0 \leq n^3+n^3+n^3+2n^3=5n^3 \quad (2)$$

Solución

El 2 se ha multiplicado por n^3 y no sustituido por ese monomio, pues si $n = 1$, $n^3 \not\geq 2$, en otras palabras, no para todo número natural n, se cumple que n^3 es mayor que 2. Por lo tanto:

$$\left|n^3 + n^2 + n + 2\right| \leq 5\left|n^3\right| \ \ \forall n,\ n \in \mathbb{N}$$

La desigualdad anterior es válida por 2 y al ser $f\left(n\right) = n^3 + n^2 + n + 2$ y $g\left(n\right) = n^3$ funciones positivas en $\mathbb{N}$, de donde el alumno debe observar que: $\left|f\left(n\right)\right| = \left|n^3 + n^2 + n + 2\right| = n^3 + n^2 + n + 2$ y $\left|g\left(n\right)\right| = \left|n^3\right| = n^3$. Se concluye que la definición 6 se satisface en este ejercicio con $c = 5$.

Solución

El paquete **VilCretas** integra el comando `CDFGraficaNA`, una instrucción que brinda la posibilidad de visualizar gráficamente el significado de las notaciones asintóticas estudiadas en este texto, incluyendo la O grande. Al ser utilizada `CDFGraficaNA` en este ejemplo:

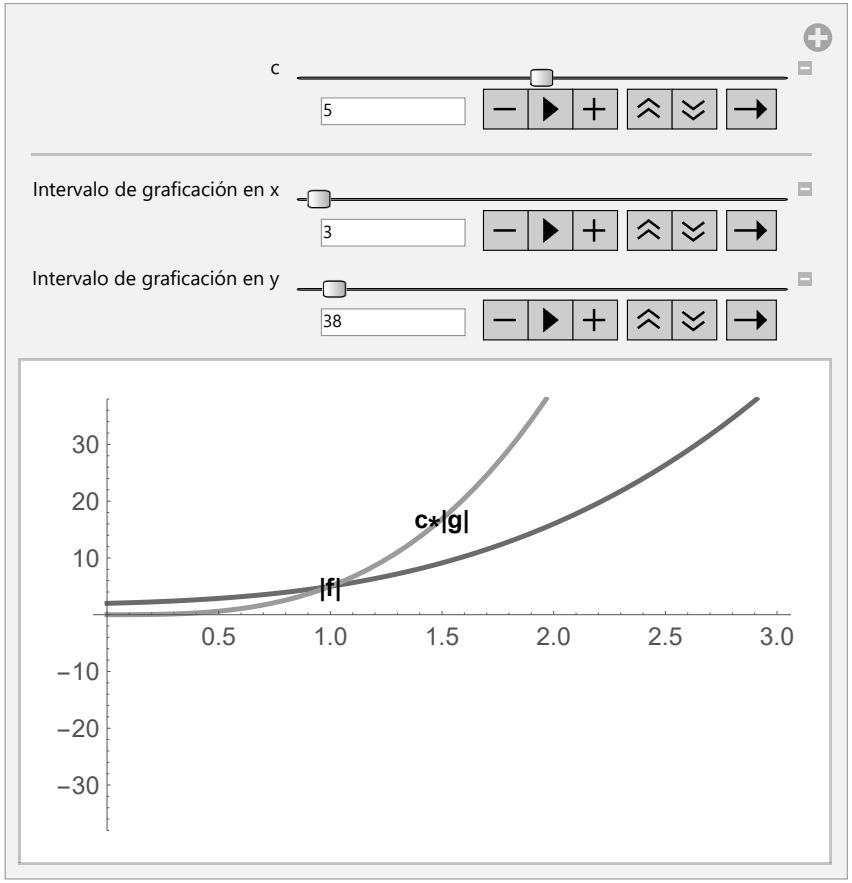
In[ ] :=

```
CDFGraficaNA[{n^3 + n^2 + n + 2, n^3}, 0.01, 10, 1000]
```

Solución

El comando anterior muestra la siguiente salida:

Out[ ] =



Solución

`CDFGraficaNA` recibe entre llaves las funciones $f\left(n\right) = n^3 + n^2 + n + 2$ y $g\left(n\right) = n^3$, la variación del deslizador que representa la constante real positiva c de la definición de O grande, en este caso de 0,01 a 10 y el valor máximo que se puede asignar a los intervalos de graficación en los ejes coordenados, 1000 para el **In[ ]** anterior. La "`NA`" que lleva al final el identificador de la función `CDFGraficaNA` es un acrónimo del término "notación asintótica", es decir, la instrucción genera un documento con un formato computable que grafica una notación asintótica.

Nota

La gráfica compartida en el **Out[ ]** devela que para $c = 5$ y a partir del número natural 1, la función $5\left|g\left(n\right)\right| = 5\left|n^3\right| = 5n^3$ acota superiormente a $\left|f\left(n\right)\right| = \left|n^3 + n^2 + n + 2\right| = n^3 + n^2 + n + 2$. En palabras más sencillas, $g\left(n\right)$ está "por encima" de $f\left(n\right)$. La interpretación gráfica del O grande, por consiguiente, implica que un múltiplo del valor absoluto de la función $g\left(n\right)$ acotará superiormente (estará por encima) al valor absoluto de $f\left(n\right)$. Cabe destacar que la constante real positiva c no es única, si el estudiante por ejemplo, cambia en el objeto dinámico el valor de $c = 5$ por $c = 2$, apreciará el cumplimiento de la cota superior pero a partir del número natural 2, esto no contradice la definición de O grande pues la desigualdad $\left|f\left(n\right)\right| \leq c\left|g\left(n\right)\right|$ puede excluir algunos naturales.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-47.zip

Explicación en video

https://youtu.be/3GIxn1vGZjs

- Lo mostrado de manera particular en el ejemplo 7 es generalizable. Consideremos el siguiente teorema.

Theorem (3.1)

Sea $f(n) = a_j n^j + a_{j-1} n^{j-1} + \cdots + a_0$ *con* $j \in \mathbb{N}$, $a_k \in \mathbb{R}\ \forall k$, $k \in \mathbb{N}$, $0 \leq k \leq j$ *y* $a_j \neq 0$ *entonces* $f(n) = O\left(n^j\right)$.

Comentario sobre el teorema 8

La propiedad expuesta en el teorema 8 es muy utilizada en la práctica. Verbalmente expresa que "cualquier polinomio de grado j es de orden a lo sumo n^j". Por ejemplo:

1. $n^4 + 6n^3 + n^2 - 9 = O\left(n^4\right)$.
2. $6n^5 - n^4 + \frac{1}{2}n^2 + n + 4 = O\left(n^5\right)$.
3. $n^6 + \frac{3}{4}n^5 - n^4 + 7n^3 - 2n^2 - 8n + 5 = O\left(n^6\right)$.

Example (3.7)

Pruebe que $1^j + 2^j + \cdots + n^j = O\left(n^{j+1}\right)$ con $j \in \mathbb{N}$. Grafique utilizando el *software Wolfram Mathematica* la cota superior para $j = 1, 2, \ldots, 10$.

Solución

Al aplicar una metodología de trabajo similar a la del ejemplo 7, se reemplazará cada sumando de $1^j + 2^j + \cdots + n^j$ por n^j, con la intención de acotar superiormente la sumatoria. Por lo tanto:

$$1^j + 2^j + \cdots + n^j \leq \underbrace{n^j + n^j + \cdots + n^j}_{n \text{ veces}} = n \cdot n^j = n^1 \cdot n^j$$

Esto nos implica que:

$$1^j + 2^j + \cdots + n^j \leq n^{j+1}$$

Solución

Como $\left|1^j + 2^j + \cdots + n^j\right| = 1^j + 2^j + \cdots + n^j$ y $\left|n^{j+1}\right| = n^{j+1}$, siendo n un entero positivo, finalmente se concluye:

$$\left|1^j + 2^j + \cdots + n^j\right| \leq \left|n^{j+1}\right| = 1 \cdot \left|n^{j+1}\right| \ \forall n,\ n \in \mathbb{N}$$

Se constata la existencia de la constante real positiva $c = 1$ en este ejercicio, satisfaciéndose la definición 6.

Solución

Con respecto a la gráfica del comportamiento asintótico es importante señalar que el comando `CDFGraficaNA` no es acorde con $f(n) = 1^j + 2^j + \cdots + n^j$ y $g(n) = n^{j+1}$ al depender ambas, de un parámetro j variable desde 1 hasta 10, siendo j un número entero. Para este tipo de funciones, la librería **VilCretas** provee el comando `CDFGraficaNAP` similar a `CDFGraficaNA`, con el detalle de facilitar un quinto argumento que especifica el valor máximo a otorgar al parámetro j, en este ejemplo, $j = 10$. `CDFGraficaNAP` asume un inicio de j en 1. La "P" al final del nombre de la sentencia se refiere al término "parámetro":

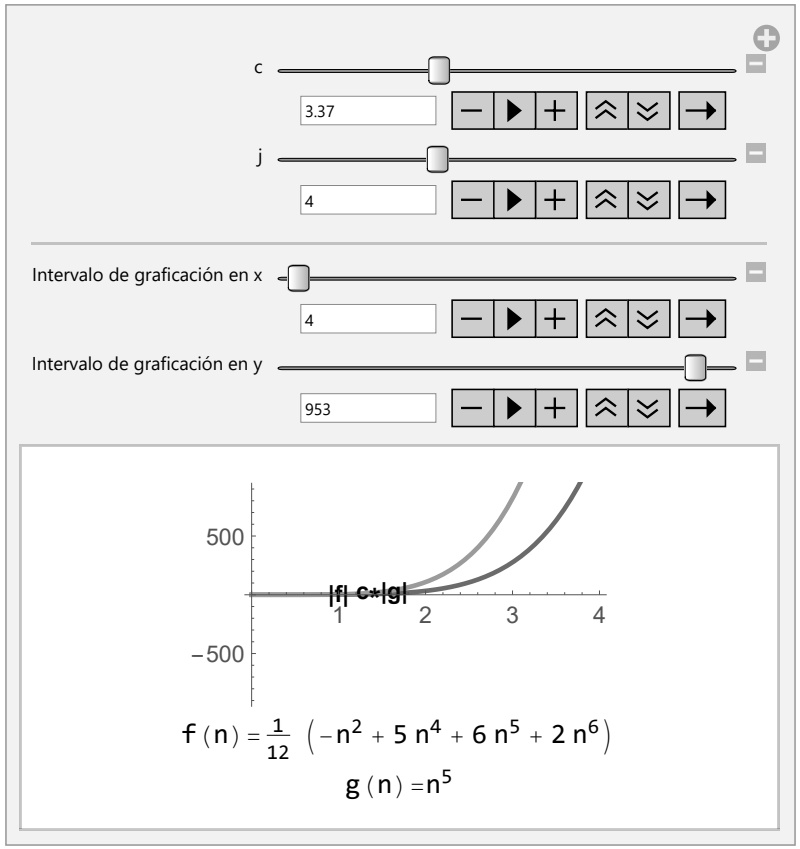
In[ ] :=

```
CDFGraficaNAP[{Sum[i^(j + 1), {i, 1, n}], n^(j + 1)}, 0.01,
10, 1000, 10]
```

Solución

Del comando anterior se obtiene la siguiente salida:

Out[ ] =



Solución

La gráfica mostrada corresponde a $j = 4$, donde cuando $c = 3{,}37$ se visualiza el acotamiento superior de $c\,|g\,(n)|$. La animación despliega nueve gráficas más, al cambiar el valor de j desde 1 hasta 10.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-48.zip

Explicación en video

https://youtu.be/zqQ6JJ23LFw

Example (3.8)

Demuestre $2 + 4 + \cdots + 2^n = O\left(2^n\right)$. Represente mediante el uso del software *Wolfram Mathematica*.

Solución

En el presente ejemplo, la función $f(n)$ es igual a $2+4+\cdots+2^n$, es decir, $f(n)$ es una sumatoria, razón por la cual, se pretendería buscar una cota superior sustituyendo cada sumando por el mayor, en este caso, ese elemento corresponde a 2^n. Si se realiza lo anterior, se obtiene:

$$2+4+\cdots+2^n \leq \underbrace{2^n+2^n+\cdots+2^n}_{n \text{ veces}} = n\cdot 2^n$$

La desigualdad, en consecuencia, nos conduce a afirmar que $2+4+\cdots+2^n = O(n\cdot 2^n)$, sin embargo, el O grande no es coincidente con la notación asintótica que nos interesa demostrar aquí.

Nota

Usualmente la técnica expuesta en los ejemplos 7 y 9 basada en sustituir por un sumando mayor con el objetivo de determinar una cota superior, servirá para realizar la prueba de una notación asintótica O grande, pese a ello, en este ejercicio no está funcionando por lo que se cambiará el modo de resolución. Se recomienda al estudiante observar el procedimiento siguiente como una guía en otros ejemplos donde falle el reemplazo por el sumando mayor.

Solución

La idea ahora, reside en encontrar una fórmula que permita calcular la sumatoria. Algunas veces esta fórmula será fácilmente reconocible y en otras ocasiones presentará un nivel de complejidad mayor. Pese a ello, en este contexto se tiene la ventaja de contar con el comando `Sum` del software *Wolfram Mathematica*. Esta instrucción al ser ejecutada en términos de n es capaz de determinar, casi siempre, una fórmula explícita. Veamos:

In[ ] :=

```
Sum[2^i, {i, 1, n}]
```

Solución

Del comando anterior se obtiene la siguiente salida:

Out[ ] =

2 (-1 + 2^n)

En este sentido:

$$2 + 4 + \cdots + 2^n = 2\left(-1 + 2^n\right) = -2 + 2 \cdot 2^n$$

Donde se aprecia que:

$$-2 + 2 \cdot 2^n = 2 \cdot 2^n - 2 \leq 2 \cdot 2^n$$

En consecuencia:

$$2 + 4 + \cdots + 2^n \leq 2 \cdot 2^n \Rightarrow \left|2 + 4 + \cdots + 2^n\right| \leq 2 \cdot \left|2^n\right| \ \ \forall n,\ n \in \mathbb{N}$$

Solución

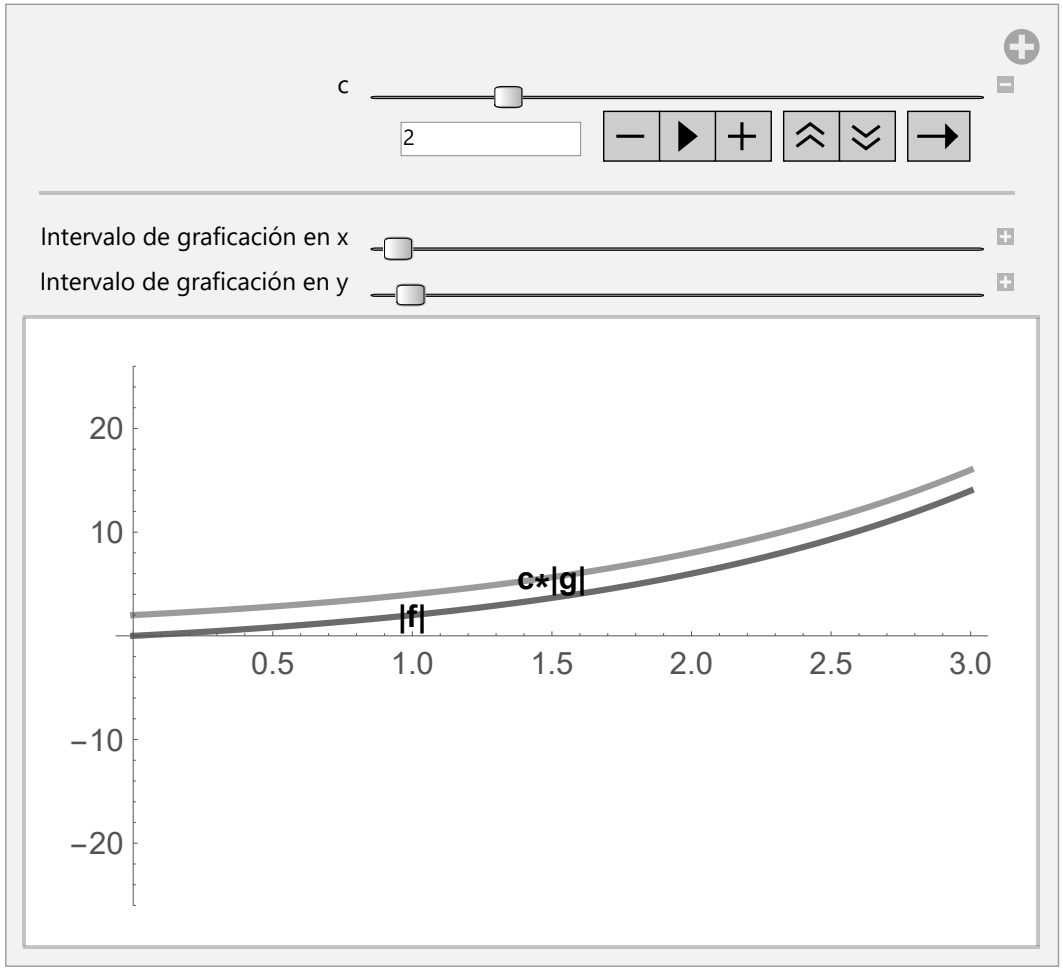
Al ser $f(n) = 2 + 4 + \cdots + 2^n$ y $g(n) = 2^n$ funciones positivas. Finalmente, se ha demostrado en virtud de la definición 6, el O grande buscado con $c = 2$. Gráficamente en *Mathematica*:

In[ ] :=

```
CDFGraficaNA[{Sum[2^i, {i, 1, n}], 2^n}, 0.01, 10, 1000]
```

Solución

Out[ ] =



Solución

Se observa como en $c = 2$, la gráfica de $c\,|g\,(n)| = 2 \cdot |2^n|$ está por encima de $|f\,(n)| = |2 + 4 + \cdots + 2^n|$.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-49.zip

Example (3.9)

Pruebe que $\ln(n!) = O(n \ln n)$. Verifique gráficamente esta igualdad por medio del *software*.

Solución

En este caso como $n! = 1 \cdot 2 \cdot \ldots \cdot n$, por propiedades del logarítmo se tiene:

$$\ln(n!) = \ln(1 \cdot 2 \cdot \ldots \cdot n) = \ln 1 + \ln 2 + \cdots + \ln n$$

Nota

El alumno debe recordar para justificar este primer paso que:

$$\log_a(x \cdot y) = \log_a x + \log_a y$$

Solución

Al sustituir en la sumatoria cada sumando por el término mayor $\ln n$, se infiere:

$$\ln 1 + \ln 2 + \cdots + \ln n \leq \underbrace{\ln n + \ln n + \cdots + \ln n}_{n \text{ veces}} = n \ln n$$

Es importante hacer notar al lector que esta desigualdad se satisface dado que la función logarítmo natural es creciente (su base es el número neperiano $e \approx 2{,}71 > 1$), por esta razón:

$$\begin{aligned} \ln 1 &\leq \ln n \\ \ln 2 &\leq \ln n \\ &\vdots \\ \ln n &\leq \ln n \end{aligned}$$

Solución

Si la base del logaritmo hubiese sido un valor entre 0 y 1, el logaritmo de n no sería el sumando mayor, sino por el contrario, el término más pequeño. Por otra parte, como $|\ln(n!)| = \ln(n!)$ y $|n \ln n| = n \ln n \ \forall n,\ n \in \mathbb{N}$, se concluye:

$$|\ln(n!)| \leq |n \ln n| = 1 \cdot |n \ln n|$$

Es decir, $\ln(n!) = O(n \ln n)$. La técnica de sustitución por el sumando mayor nos ha servido nuevamente. Antes de verificar la igualdad a través del comando `CDFGraficaNA` es importante mencionar que en *Mathematica* la sentencia `Log[n]` calcula el logarítmo natural de `n`. En general, la instrucción `Log[a, n]` devuelve en el *software* el logarítmo en base `a` de `n`.

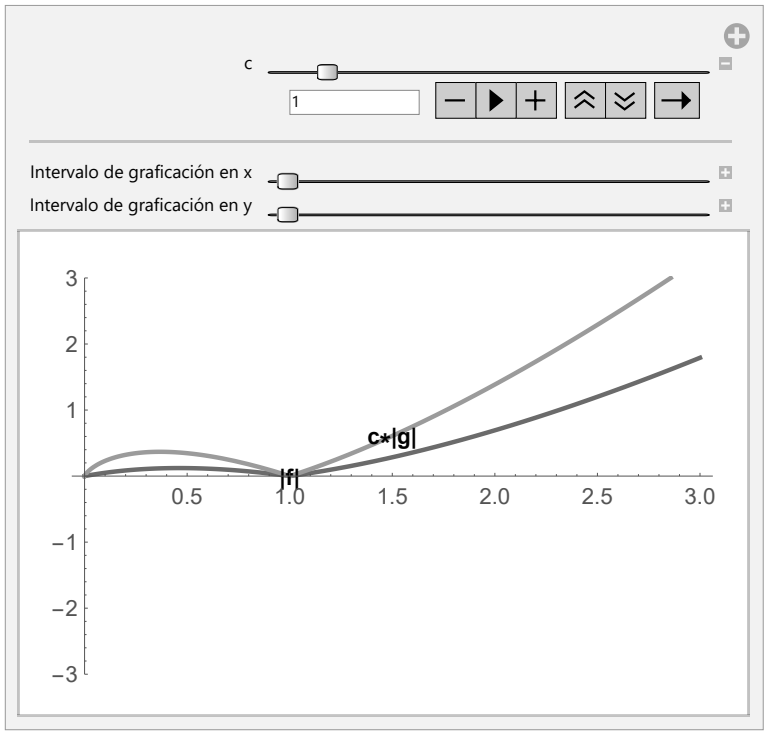
Solución

Luego:

In[ ] :=

```
CDFGraficaNA[{Log[n!], n Log[n]}, 0.01, 10, 1000]
```

Out[ ] =



Solución

En la gráfica, con $c = 1$, se comprueba la notación asintótica, al ser $|g(n)| = |n \ln n|$ una cota superior (está por encima) de $|f(n)| = |\ln(n!)|$.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-50.zip

- Demostrar órdenes de convergencia utilizando la definición 6, puede tornarse una tarea muy laboriosa o compleja dependiendo de las funciones $f(n)$ y $g(n)$ que se tengan. Por este motivo, muchas veces este tipo de pruebas se realizan mediante los atributos de la notación asintótica O grande. El siguiente teorema resume las principales propiedades.

Theorem (3.2)

Sean f, g, h y t funciones definidas sobre el conjunto de los números naturales:

1. $c = O(1)$, $c \in \mathbb{R}$.
2. $f(n) = O(f(n))$ *(propiedad reflexiva).*
3. $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = c$, $c \in \mathbb{R}^+ \cup \{0\} \Rightarrow f(n) = O(g(n))$ *(propiedad del límite).*
4. $f(n) = O(g(n))$ y $h(n) = O(t(n))$:
   1. $f(n) + h(n) = O(Max(|g(n)|, |t(n)|))$ *con Max el máximo entre las funciones* $|g(n)|$ y $|t(n)|$ *(regla de la suma).*
   2. $f(n) \cdot h(n) = O(g(n)) \cdot O(t(n)) = O(g(n) \cdot t(n))$ *(regla de la multiplicación).*

La propiedad 3 del teorema 12 tiene importantes aplicaciones prácticas. Veamos algunos ejemplos.

Example (3.10)

Pruebe que $\frac{(n+1)(4n+1)}{7n+1} = O(n)$.

Solución

La igualdad se satisface si:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)}$$

da como resultado un número real positivo o cero, donde $f(n) = \frac{(n+1)(4n+1)}{7n+1}$ y $g(n) = n$. En *Wolfram Mathematica* un límite se calcula por medio del comando `Limit`, entonces:

In[ ] :=

```
Limit[((n + 1) (4 n + 1))/(7 n + 1)/n, n -> Infinity]
```

Out[ ] =

4/7

Solución

`Infinity` corresponde al símbolo ∞ en el software. Naturalmente, el alumno podría calcular este límite utilizando principios de cálculo diferencial:

$$\lim_{n\to+\infty} \frac{\frac{(n+1)(4n+1)}{7n+1}}{n} = \lim_{n\to+\infty} \frac{4n^2+5n+1}{7n^2+n} = \lim_{n\to+\infty} \frac{4n^2}{7n^2} = \frac{4}{7}$$

En este libro se empleará de forma directa la instrucción `Limit` con la intención de facilitar los procedimientos. Al dar el límite como resultado un número real positivo $(4/7)$, la propiedad 3 del teorema 12 garantiza que $f(n) = O(g(n))$, es decir, $\frac{(n+1)(4n+1)}{7n+1} = O(n)$.

Nota

El alumno podría tener la sesación de que lo realizado usando la propiedad del límite del teorema 12, no corresponde a una demostración, sin embargo, esto es falso, pues el uso de esta regla es tan formal como la utilización de la definición de O grande. A este respecto, también cabe aclarar, que el empleo de esa definición en este ejercicio no es consistente pues la función $f(n) = \frac{(n+1)(4n+1)}{7n+1}$ claramente no permite buscar un sumando mayor a ser reemplazado.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-51.zip

Example (3.11)

Demuestre $2^n = O(n!)$.

Solución

El estudiante debe notar que el uso de la definición 6 no es apropiado pues la función $f(n) = 2^n$ no corresponde a una suma. Nuevamente entonces, se piensa en calcular $\lim_{n\to+\infty} \frac{f(n)}{g(n)}$ siendo $g(n) = n!$. De forma análitica, este cálculo merece cierta cantidad de esfuerzo, pese a ello, la setencia `Limit` ofrece un resultado casi inmediato. En *Mathematica*:

In[ ] :=

```
Limit[2^n/n!, n -> Infinity]
```

Out[ ] =

0

Solución

Finalmente, al ser el valor del límite igual a 0, en virtud de la regla del límite enunciada en el teorema 12, se concluye que $2^n = O(n!)$.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-52.zip

Example (3.12)

Pruebe usando el límite $n! = O\left(n^n\right)$.

Solución

Se observa que $f(n) = n!$ y $g(n) = n^n$, luego en *Wolfram*:

In[ ] :=

```
Limit[n!/n^n, n -> Infinity]
```

Out[ ] =

0

Según la regla del límite, al ser el resultado 0 se concluye $n! = O(n^n)$.

También, en este ejemplo el cálculo del límite requerido $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{n!}{n^n}$ no es una tarea tan sencilla si se llevara a cabo de forma analítica, sin embargo, afortunadamente la solución se ha encontrado de manera directa gracias al uso del software.

Nota

En este ejercicio sí existe la posibilidad de ejecutar la demostración por definición:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \leq \underbrace{n \cdot n \cdot n \cdot \ldots \cdot n}_{n \text{ veces}} = n^n \Rightarrow |n!| \leq 1 \cdot |n^n| \ \ \forall n,\ n \in \mathbb{N}$$

Aunque $n!$ no es una sumatoria se aprecia un procedimiento similar, al corresponder $n!$ a una productoria.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-53.zip

Solución alternativa mediante el límite

- En algunos ejercicios es viable aplicar la definición 6 y también la propiedad del límite del teorema 12 (ver los ejemplos 7, 10, 11 y 15), en otros, solo cabe usar la definición 6 (ver el ejemplo 9) y en otros, resulta apropiado únicamente el uso del límite (ver ejemplos 13 y 14). Solo la práctica y perseverancia brindarán al estudiante los recursos lógicos para elegir el camino correcto en demostraciones de esta naturaleza. La ventaja radica en la baja inversión que se necesitará durante el ensayo y el error del cálculo del límite, al ser *Wolfram Mathematica* quien supere esa complejidad.

Solución alternativa mediante el límite

- Si se retoman los ejemplos 7, 10 y 11 recurriendo al límite:

In[ ] :=

```
{Limit[(n^3 + n^2 + n + 2)/n^3, n -> Infinity],
Limit[Sum[2^i, {i, 1, n}]/2^n, n -> Infinity],
Limit[Log[n!]/(n Log[n]), n -> Infinity]}
```

Out[ ] =

$\{1, 2, 1\}$

Solución alternativa mediante el límite

- Todos los valores del **Out[ ]** son positivos, razón por la cual, quedaría demostrada por esta vía, cada una de las notaciones asintóticas O grande respectivas.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-54.zip

- Las propiedades de la suma y de la multiplicación del teorema 12, son también atributos teóricos muy útiles para hallar una notación asintótica O grande. Consideremos algunos ejemplos.

Example (3.13)

Encuentre una notación O grande tan buena como sea posible para las siguientes funciones:

1. $\left(n! + 3^n\right)\left(n^3 + \ln\left(n^2 + 5\right)\right)$
2. $\left(n \ln n + n^3\right)\left(n^4 - 2\right) + n^2 \ln n$

Solución

1. En este ejercicio se analiza que $n! = O\left(n^n\right)$ por el ejemplo 15, $3^n = O\left(3^n\right)$ y $n^3 = O\left(n^3\right)$ por la propiedad reflexiva del teorema 12, además, $\ln\left(n^2 + 5\right) = O\left(\ln n\right)$ por la regla del límite:

In[ ] :=

```
Limit[Log[n^2 + 5]/Log[n], n -> Infinity]
```

Out[ ] =

2

Solución

En consecuencia, por la propiedad de la suma del teorema 12:

- $\underbrace{n!}_{O(n^n)} + \underbrace{3^n}_{O(3^n)} = O\left(Max\left(\left|n^n\right|, \left|3^n\right|\right)\right) = O\left(Max\left(n^n, 3^n\right)\right)$ al ser funciones positivas $= O\left(n^n\right)$ pues n^n crece más rapidamente que 3^n.
- $\underbrace{n^3}_{O(n^3)} + \underbrace{\ln\left(n^2+5\right)}_{O(\ln n)} = O\left(Max\left(\left|n^3\right|, \left|\ln n\right|\right)\right) = O\left(Max\left(n^3, \ln n\right)\right)$ al ser funciones positivas $= O\left(n^3\right)$ pues n^3 crece más rapidamente que $\ln n$.

Solución

Entonces:

$$\underbrace{(n! + 3^n)}_{O(n^n)}\underbrace{(n^3 + \ln(n^2 + 5))}_{O(n^3)} = O(n^n \cdot n^3) = O(n^{n+3})$$

por la regla del producto del teorema 12.

Al verificar usando el límite:

In[ ] :=

```
Limit[((n! + 3^n) (n^3 + Log[n^2 + 5]))/n^(n + 3), n ->
Infinity]
```

Out[ ] =

0

Esta última comprobación no es necesaria, sin embargo, se ha realizado de forma alterna por la facilidad del cálculo del límite en *Mathematica*.

Solución

2. En este caso se debe notar que $n = O(n)$, $\ln n = O(\ln n)$, $n^2 = O(n^2)$, $n^3 = O(n^3)$ y $n^4 = O(n^4)$ usando la regla reflexiva del teorema 12. Luego:
   - $\underbrace{n \ln n}_{O(n \cdot \ln n)} + \underbrace{n^3}_{O(n^3)} = O\left(Max\left(|n \ln n|, |n^3|\right)\right)$ por la propiedad $4a$ del teorema 12
     $= O\left(Max\left(n \ln n, n^3\right)\right)$ al ser funciones positivas
     $= O\left(n^3\right)$ al crecer n^3 más velózmente en comparación con $n \ln n$.
   - $n^4 - 2 = O\left(n^4\right)$ por el teorema 8.
   - $\underbrace{\left(n \ln n + n^3\right)}_{O(n^3)}\underbrace{\left(n^4 - 2\right)}_{O(n^4)} = O\left(n^3 \cdot n^4\right) = O\left(n^7\right)$ por la regla $4b$ del teorema 12.
   - $\underbrace{n^2}_{O(n^2)} \cdot \underbrace{\ln n}_{O(\ln n)} = O\left(n^2 \cdot \ln n\right) = O\left(n^2 \ln n\right)$ por la propiedad $4b$ del teorema 12.

Solución

De donde, por la regla de la suma:

$$\underbrace{\left(n \ln n + n^3\right)\left(n^4 - 2\right)}_{O(n^7)} + \underbrace{n^2 \ln n}_{O(n^2 \ln n)} = O\left(Max\left(\left|n^7\right|, \left|n^2 \ln n\right|\right)\right) \tag{3}$$
$$= O\left(Max\left(n^7, n^2 \ln n\right)\right) = O\left(n^7\right)$$

Se puede verificar que n^7 crece más rápidamente que $n^2 \ln n$, al tomar $\lim_{n \to +\infty} \frac{n^2 \ln n}{n^7}$ y corroborar su igualdad a 0. En *Mathematica*:

In[ ] :=

```
Limit[(n^2 Log[n])/n^7, n -> Infinity]
```

Out[ ] =

0

Solución

Finalmente, si se desea de manera adicional, comprobar 3 mediante el uso de *Wolfram* y la propiedad del límite del teorema 12, se tiene:

In[ ] :=

```
Limit[((n Log[n] + n^3) (n^4 - 2) + n^2 Log[n])/n^7, n ->
Infinity]
```

Out[ ] =

1

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-55.zip

- La teoría desarrollada en este capítulo tiene como principal objetivo el ánalisis de la complejidad de un algoritmo, es decir, encontrar si es posible una función que represente el tiempo de ejecución en el peor, mejor o caso promedio, determinar su orden asintótico y definir a razón de este orden qué tan eficiente es su lógica de programación. A continuación, se abordarán algunos ejemplos en ese sentido.

Example (3.14)

Halle para el algoritmo de la burbuja una notación asintótica O grande en el peor de los casos.

Solución

Como se mencionó en el ejemplo 5, el método de la burbuja se sustenta en realizar comparaciones. Una función $f(n)$ que mide su tiempo de ejecución en el peor de los casos, por consiguiente, debe contar cuántas comparaciones como máximo realiza el algoritmo al ordenar una lista L con n datos, $n \in \mathbb{N}$. Este conteo no es tan evidente si el lector lo analiza. De allí que se empleará una relación de recurrencia para poderlo efectúar. Si a_n representa el número máximo de comparaciones que ejecuta el algoritmo de la burbuja, se aprecia que ese valor es igual a la cantidad de comparaciones al ordenar $n-1$ datos, es decir, a_{n-1}, más la medida máxima de pasos requieridos para acomodar el último dato.

Solución

La peor situación se presentará si ese último elemento es mayor que todos los demás, en cuyo caso, se necesitarán $n-1$ comparaciones adicionales al finalizar el ordenamiento. Como consecuencia, $a_n = a_{n-1} + n - 1$ con $a_1 = 0$, corresponde al costo en tiempo de ejecución del método de la burbuja. Se asume $a_1 = 0$ pues no hay comparaciones en una lista de longitud 1. Si se resuelve ahora la relación de recurrencia se tendría una función explícita que analiza la complejidad del método de la burbuja.

Solución

Utilizaremos para ello el comando RR:

In[ ] :=

```
RR[{1, n - 1}, {0}, n]
```

Out[ ] =

1/2 (-1 + n) n

De acuerdo con este resultado y por lo establecido en el teorema 8, se concluye:

$$f(n) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O\left(n^2\right)$$

Nota

La función $g(n) = n^2$ constituye la representación asintótica O grande del algoritmo de la burbuja. Esto nos muestra que para listas L donde su longitud n es muy grande, el procedimiento tenderá a ralentizarse. La gráfica exhibida en la página 61, evidencia que n^2 está por encima de la función identidad ($y = n$), por lo que un algoritmo en el orden cuadrático no se comportará adecuadamente si n tiende a crecer.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-56.zip

Example (3.15)

Considere el programa recursivo `factoriales` que calcula el factorial de un número natural o cero. Determine para el peor y mejor caso una notación asintótica O grande.

Solución

Una función $f(n)$ que mide el nivel de esfuerzo realizado por un método recursivo como lo es `factoriales`, debería contar el número total de llamadas o invocaciones que realiza. Recordando el código de `factoriales`:

```
factoriales[n_]:=If[Or[n==0,n==1],Return[1],
                    n*factoriales[n-1]]
```

Solución

En el peor de los casos $n \neq 0$ y $n \neq 1$. ¿Cuántas veces se llama así misma la función `factoriales` en el cálculo de $n!$? En *Wolfram* al correr paso a paso, por ejemplo, lo que hace `factoriales` con $n = 5$, se tiene:

In[ ] :=

```
Factoriales[5, steps -> True]
```

Out[ ] =

Factoriales[5] = 5 * Factoriales[4] = 120
Factoriales[4] = 4 * Factoriales[3] = 24
Factoriales[3] = 3 * Factoriales[2] = 6
Factoriales[2] = 2 * Factoriales[1] = 2
Factoriales[1] = 1
120

Solución

Por lo que, 5! se ha determinado por medio de 5 invocaciones de `factoriales`. El total de pasos ejecutados, si se piensa de una manera más específica, no solo depende en `factoriales` de la cantidad de llamadas, sino también, del número de reemplazos que hace hacia arriba hasta obtener $n!$. En 5! esa cantidad de sustituciones es 5 del mismo modo. En otras palabras, el cálculo de 5! requiere de $2 \cdot 5 = 10$ pasos.

Solución

En general, el alumno puede extrapolar las ideas a cualquier valor de n, $n \in \mathbb{N}$, de donde se intuye la función $f(n) = 2n$. Finalmente, por el teorema 8:

$$f(n) = 2n = O(n)$$

Nota

Un orden lineal O grande como éste, no es ni bueno ni malo pues la forma de crecimiento del tiempo se comporta igual a la cantidad de datos procesados. Por otra parte, el programa `factorialesCola` visto con anterioridad, se diferencia en tiempo de ejecución de `factoriales` al necesitar únicamente las llamadas recursivas para llegar al cálculo (no hay sustituciones hacia arriba), por lo que una función $f(n)$ que estima su nivel de esfuerzo es $f(n) = n = O(n)$. En este caso, el O grande tampoco brinda información adicional para concluir, como se hizo en el ejemplo 4, una mayor eficiencia en el método `factorialesCola`.

Solución

El mejor caso de `factoriales` ocurre cuando $n = 0$, o bien, $n = 1$. Allí, directamente `factoriales` retorna a 1 y al ejecutarse solo el paso de retorno, esto permite concluir un orden de convergencia $O(1)$.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-57.zip

Example (3.16)

Analice a través de la notación asintótica O grande, el tiempo que tarda la peor situación del algoritmo:

```
CalculaSuma[n_] := Module[{p = 0}, For[i = 1, i <= n,
For[j = 1, j <= n, p = p + 2; j++]; i++]; Return[p]]
```

Solución

El programa muestra dos ciclos "for" cuya instrucción interna es una asignación simple `p = p + 2`.

Nota

En un método cualquier asignación que no modifique la variable que controla un ciclo y del mismo modo, cualquier operación aritmética que se realice, son consideradas operaciones simples.

Solución

En dicho caso, el nivel de esfuerzo del método `CalculaSuma` es medible al contar el número de veces que se realiza la asignación `p = p + 2`. Si por ejemplo, se asume de forma particular $n = 3$, ¿Cuántas veces se ejecuta `p = p + 2`? Al tomarse `i = 1` en el primer "for" el segundo itera `p = p + 2`, 3 veces , al tener `i = 2`, de nuevo el segundo "for" itera 3 veces `p = p + 2`, finalmente, en `i = 3`, el primer "for" produce en el segundo "for" anidado, tres asignaciones más de `p = p + 2`. Se concluye $9 = 3 \cdot 3$ asignaciones simples de `p = p + 2`. Visto de forma general, el conteo sobre la cantidad de asignaciones simples corresponde al número de iteraciones ejecutadas por el primer ciclo "for" multiplicado por el número de iteraciones asociadas al segundo ciclo "for", esto es:

$$n \cdot n = n^2 = O\left(n^2\right)$$

Solución

Otra manera de obtener el resultado anterior, consiste en encontrar un O grande para cada ciclo "for" y luego usar la regla de la multiplicación enunciada en el teorema 12. Con respecto a los "for", cada uno recorre un ciclo desde 1 hasta llegar a n, por lo tanto, se encuentran en un orden $O(n)$. El programa finalmente tiene un comportamiento asintótico:

$$O(n) \cdot O(n) = O(n \cdot n) = O\left(n^2\right)$$

En consecuencia, `CalculaSuma` tiene una velocidad de convergencia en el orden $O\left(n^2\right)$, lo cual evidencia que no tendrá un tiempo de salida apropiado cuando n tiende a ser un número grande. Cabe destacar que `CalculaSuma` es una función del paquete **VilCretas** por si el alumno desea correr algunas pruebas directamente en *Mathematica*.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-58.zip

Example (3.17)

Determine un O grande para el peor caso en:

```
CalculaProducto[n_] := Module[{p = 1}, While[p <= n,
p = p*3]; Return[p]]
```

Solución

La complejidad del método `CalculaProducto` en concordancia con lo expuesto en el ejemplo anterior, se analiza encontrando una función que cuente el número de veces que se realiza la asignación simple `p = p*3`. El problema radica en el hecho de no saber de manera directa cuántas veces itera el ciclo "while".

Nota

La forma de resolución que se compartirá tal y como prosigue, constituye una técnica de trabajo cuando se desea analizar un algoritmo que depende de una estructura de control de repetición, donde no se conoce el número de veces que itera una asignación simple.

Solución

Empecemos entonces. En el código, la variable que controla la entrada al ciclo "while" corresponde a p, interesa determinar la forma que toma esa variable despues de "k" iteraciones. Esto se puede determinar construyendo una relación de recurrencia sugerida por la asignación p = p*3, que al inicializarce en 1, configura la recursividad:

$$p_k = 3p_{k-1} \text{ con } p_0 = 1 \tag{4}$$

Solución

Al resolver 4 en *Wolfram Mathematica*:

In[ ] :=

```
RR[{3}, {1}, k, inicio -> 0]
```

Out[ ] =

3^k

Por lo tanto, la variable `p` en el algoritmo `CalculaProducto` toma la forma 3^k después de k iteraciones. Luego, la ejecución de `CalculaProducto` termina cuando $3^k > n$ (se niega la condición de entrada al "while"), por lo que:

$$3^k > n \Rightarrow \ln\left(3^k\right) > \ln n \Rightarrow k \ln 3 > \ln n \Rightarrow k > \frac{\ln n}{\ln 3}$$

Solución

El estudiante no debe perder de vista cómo la variable k es la que mide el tiempo de ejecución de `CalculaProducto` al contar el número de asignaciones simples realizadas en `p = p*3`. Como k es un número entero es posible suponer un valor mínimo sacando la parte entera superior de $\frac{\ln n}{\ln 3}$, o bien, avanzado al entero superior de $\frac{\ln n}{\ln 3}$ si $\frac{\ln n}{\ln 3} \in \mathbb{Z}$, es decir, $k = \frac{\ln n}{\ln 3} + a$, $a \in \mathbb{R}^+$, donde la "a" es una constante que facilita un corrimiento al entero más pequeño mayor que $\frac{\ln n}{\ln 3}$. De esta igualdad, se manifiesta la intuición de asumir un orden asintótico O grande en el $\ln n$.

Solución

Para demostrar $k = O(\ln n)$ se puede recurrir a la propiedad del límite del teorema 12:

In[ ] :=

```
Clear[a]
Limit[(Log[n]/Log[3] + a)/Log[n], n -> Infinity]
```

Out[ ] =

```
1/Log[3]
```

El comando `Clear` nativo del software *Mathematica* limpia de la memoria la variable a. El límite nos ha devuelto $\frac{1}{\ln 3} \in \mathbb{R}^+$ por lo que queda demostrado $k = O(\ln n)$.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-59.zip

Example (3.18)

El método `CalculaOtroProducto` dado a continuación, devuelve la misma salida del programa `CalculaProducto`. ¿Cuál de los dos algoritmos teóricamente proporciona los mejores resultados en tiempo de ejecución?

```
CalculaOtroProducto[n_] := Module[{p = 1}, For[i = 1,
i <= n, While[p <= i, p = p*3]; i++]; Return[p]]
```

Solución

Los algoritmos `CalculaProducto` y `CalculaOtroProducto` son los expuestos en el ejemplo 2. Allí, ya se habia cotejado experimentalmente que `CalculaProducto` consume menos tiempo de ejecución al interar menor cantidad de veces en comparación con `CalculaOtroProducto`.
Nos corresponde en el presente ejercicio, analizar mediante el uso de la notación asintótica O grande, si es posible demostrar una mayor eficiencia en `CalculaProducto`.
El programa `CalculaOtroProducto` está basado en dos estructuras de control de repetición, un "for" y un "while". El "while" alcanza su peor situación cuando `i = n` y bajo ese supuesto, dicho ciclo se reduce al del ejemplo 20, por lo que se tiene un orden a lo sumo $O(\ln n)$. Por otra parte, el "for" se ejecuta n veces, en cuyo caso tiene un orden $O(n)$.

Solución

Luego, por la regla del producto del teorema 12, el método `CalculaOtroProducto` posee un orden de convergencia:

$$O(n) \cdot O(\ln n) = O(n \cdot \ln n) = O(n \ln n)$$

Si recordamos las comparaciones realizadas en la gráfica mostrada por el comando `CDFGraficaNOG[1500]`, $\ln n$ está por debajo de la función $n \ln n$. Esto nos lleva a concluir que el algoritmo más efectivo en tiempo de ejecución es `CalculaProducto`. En este caso, el uso de la notación asintótica O grande sí nos ha permitido comparar (teóricamente) ambos procedimientos.

Example (3.19)

Analice la peor situación de `CalculaOtraSuma`, donde:

```
CalculaOtraSuma[n_]:=Module[{p = 1, i = n}, While[i >= 1,
For[j = 1, j <= i, p += 2; j++]; i = i/Sqrt[2]];
Return[p]]
```

Solución

Al examinar el código que caracteriza a la función `CalculaOtraSuma`, el "for" anidado tiene la peor situación al iniciar las iteraciones, cuando `i = n`, en cuyo caso, un O grande de ese ciclo es $O(n)$. Por otra parte, al analizar el "while" y siguiendo las ideas expuestas en el ejemplo 20, nos interesa conocer la forma que toma la variable `i` encargada de controlar la entrada al ciclo. Para ello, se parte de la asignación `i = i/Sqrt[2]` y el valor inicial `i = n`, construyendo la relación de recurrencia:

$$i_k = \frac{i_{k-1}}{\sqrt{2}} \text{ con } i_0 = n$$

Solución

El estudiante debe recordar que la sentencia `Sqrt` devuelve la raíz cuadrada en el software. Al ser resuelta i_k en *Wolfram Mathematica*:

In[ ] :=

```
RR[{1/Sqrt[2]}, {n}, k, inicio -> 0]
```

Out[ ] =

2^(-k/2) n

Esto nos indica que después de "k" iteraciones la variable $i = 2^{-\frac{k}{2}} n = \frac{n}{2^{\frac{k}{2}}}$.

Solución

Negando la condición de entrada al "while" se obtiene:

$$i < 1 \Rightarrow i_k < 1 \Rightarrow \frac{n}{2^{\frac{k}{2}}} < 1 \Rightarrow n < 2^{\frac{k}{2}}$$

$$\Rightarrow \ln n < \ln\left(2^{\frac{k}{2}}\right) \Rightarrow \ln n < \frac{k}{2}\ln 2$$

$$\Rightarrow \frac{2\ln n}{\ln 2} < k \Rightarrow k > \frac{2\ln n}{\ln 2}$$

Solución

Ahora se asume k como el menor entero posible mayor a $\frac{2 \ln n}{\ln 2}$, es decir, sabemos que existe una constante a, $a \in \mathbb{R}^+$, donde $k = \frac{2 \ln n}{\ln 2} + a$. Se conjetura un $O(\ln n)$ para el ciclo "while" a comprobarse por medio de la propiedad del límite del teorema 12, veamos:

In[ ] :=

```
Clear[a]
Limit[((2 Log[n])/Log[2] + a)/Log[n], n -> Infinity]
```

Out[ ] =

2/Log[2]

Solución

Como $\frac{2}{\ln 2}$ es un número real positivo queda demostrado $k = O(\ln n)$. En conclusión, `CalculaOtraSuma` presenta un orden asintótico O grande:

$$O(\ln n) \cdot O(n) = O(\ln n \cdot n) = O(n \ln n)$$

Esto por la regla de la multiplicación. $O(n \ln n)$ refleja un mal comportamiento en tiempo de ejecución pues $n \ln n$ es una función que está por encima de la identidad (ver gráfica generada por el comando `CDFGraficaNOG[1500]`). Naturalmente este orden es mejor que uno polinomial o exponencial, pero aún así, la función $n \ln n$ crece más rápidamente de lo que lo hace la variable n o el número de datos procesados.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-60.zip

- Los ejemplos 20 y 22 tienen en común el empleo de un ciclo “while” en los métodos analizados y en ambos, al buscar el O grande apareció un logaritmo. Esto podría dar la impresión equivocada de relacionar el “while” con la presencia de un logaritmo en el O grande. El ejercicio que prosigue evidencia que no siempre el “while” implicará un logaritmo en este tipo de notación asintótica.

Example (3.20)

Determine una notación asintótica O grande en la peor situación de:

```
Programa[n_]:=Module[{p = 1}, For[i = 0, i <= n+3,
While[p <= i, p += 2]; i++]; Return[p]]
```

Solución

Es claro que el ciclo "for" se ejecuta:

$$\underbrace{n+3}_{\text{Extremo superior}} - \underbrace{0}_{\text{Extremo inferior}} + 1 = n+4 \text{ veces}$$

Luego:

$$n+4 = O(n) \text{ por el teorema 8}$$

Ahora se centrará la atención en el ciclo "while". La peor situación de ese ciclo ocurre cuando `i = n+3`, es decir:

`While[p <= n+3, p += 2]` con `p = 1` inicialmente

Solución

De donde, se infiere la relación de recurrencia:

$$p_k = p_{k-1} + 2 \text{ con } p_0 = 1$$

En *Wolfram*:

In[ ] :=

```
RR[{1, 2}, {1}, k, inicio -> 0]
```

Out[ ] =

$1 + 2\ k$

Al negar la condición de entrada al ciclo "while", se tiene:

$$p > n+3 \Rightarrow p_k > n+3 \Rightarrow 2k+1 > n+3 \Rightarrow k > \frac{n+2}{2}$$

Solución

En consecuencia, un valor mínimo para k corresponde al menor número entero mayor que $\frac{n+2}{2}$. Por lo tanto, $k = \frac{n+2}{2} + a$, $a \in \mathbb{R}^+$, a una constante, de donde se conjetura un $O(n)$. Para probarlo se utiliza el límite:

In[ ] :=

```
Clear[a]
Limit[((n + 2)/2 + a)/n, n -> Infinity]
```

Out[ ] =

$1/2$

Solución

Como $\frac{1}{2} \in \mathbb{R}^+$, se concluye $k = O(n)$ por el teorema 12. Analizando el método `Programa`, su orden asintótico O grande por la regla del producto es:

$$O(n) \cdot O(n) = O(n \cdot n) = O(n^2)$$

Éste no es un buen orden de convergencia como ya se ha señalado en algunos ejemplos anteriores.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-61.zip

Example (3.21)

Encuentre una notación asintótica O grande en la peor situación de `OtroPrograma`, con:

```
OtroPrograma[n_]:=Module[{p = 1, j = n}, While[j >= 1,
For[i = 1, i <= j, p = p/4; i++]; j = Floor[j/2]];
Return[p]]
```

La instrucción `Floor` en *Wolfram Mathematica* calcula la parte entera inferior o piso.

Solución

Se inicia encontrando un O grande para la peor situación del "for" que ocurre cuando `j = n`. Bajo este supuesto el ciclo anidado "for" es $O(n)$. Con respecto al "while" la variable que controla su entrada es j y ésta se actualiza en cada iteración como la parte entera piso de $\frac{j}{2}$. ¿Qué diferencia implica el detalle del uso de la función parte entera inferior con respecto al procedimiento habitual ya explicado, cuando tenemos un "while"?, la respuesta es ninguna, debido al hecho de que $\frac{j}{2}$ es mayor o igual que $\left\lfloor \frac{j}{2} \right\rfloor$ (el símbolo $\lfloor \_ \rfloor$ representa la parte entera inferior), por lo que si se determina una cota superior para el "while" cambiando $\left\lfloor \frac{j}{2} \right\rfloor$ por $\frac{j}{2}$, será también una cota superior del "while" original.

Solución

Luego, se parte de la relación de recurrencia:

$$j_k = \frac{j_{k-1}}{2} \text{ con } j_0 = n$$

En *Mathematica*:

In[ ] :=

```
RR[{1/2}, {n}, k, inicio -> 0]
```

Out[ ] =

2^-k n

Al negar la condición de entrada en el "while":

$$j < 1 \Rightarrow j_k < 1 \Rightarrow 2^{-k} \cdot n < 1 \Rightarrow \frac{n}{2^k} < 1 \Rightarrow n < 2^k$$

$$\Rightarrow \ln n < \ln 2^k \Rightarrow \ln n < k \ln 2 \Rightarrow \frac{\ln n}{\ln 2} < k \Rightarrow k > \frac{\ln n}{\ln 2}$$

Solución

Se toma ahora $k = \frac{\ln n}{\ln 2} + a$, $a \in \mathbb{R}^+$, a una constante, donde por la propiedad del límite:

In[ ] :=

```
Clear[a]
Limit[(Log[n]/Log[2] + a)/Log[n], n -> Infinity]
```

Out[ ] =

```
1/Log[2]
```

Solución

El valor $\frac{1}{\ln 2}$ es un número real positivo, a razón de ello, $k = O(\ln n)$ y `OtroPrograma` por la regla de la multiplicación del teorema 12, tiene un orden O grande:

$$O(\ln n) \cdot O(n) = O(\ln n \cdot n) = O(n \ln n)$$

Esta notación asintótica devela un mal comportamiento en tiempo de ejecución, cuando n tiende a crecer.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-62.zip

- Además de la notación asintótica O grande, existen otras similares que acotan el esfuerzo realizado por un algoritmo inferiormente y superior e inferiormente de manera simultánea. La sección próxima trata el tema con la intención de ampliar aún más, los métodos de análisis sobre la complejidad de un programa.

Enfoque teórico: otras notaciones asintóticas

Prof. Enrique Vílchez Quesada

Universidad Nacional de Costa Rica

- Si una función $f(n)$ se acota inferiormente por medio de otra función $g(n)$, se le asocia una notación asintótica llamada "omega". Cuando $f(n)$ se acota tanto superior como inferiormente a través de una función $g(n)$, se dice que f tiene un orden "theta" de g. La definición formal se presenta a continuación.

Definition (3.2)

Se dice que una función $f(n)$ con dominio en el conjunto de los números naturales es de orden al menos $g(n)$ y se denota $f(n) = \Omega(g(n))$ si existe una constante real positiva c, tal que:

$$|f(n)| \geq c\,|g(n)| \;\; \forall n,\; n \in \mathbb{N}$$

o excepto un número finito de ellos. A la notación $\Omega(g(n))$ se le llama notación asintótica "omega". Por otra parte, si $f(n) = \Omega(g(n))$ y $f(n) = O(g(n))$ entonces se dice que $f(n)$ es de orden "*theta*" de $g(n)$ y se representa $f(n) = \Theta(g(n))$.

Comentario sobre el uso de las notaciones asintóticas

Si una función $f(n)$ que representa el esfuerzo realizado por un algoritmo es de orden "omega" de $g(n)$, la función $g(n)$ nos da una idea sobre el funcionamiento del algoritmo para el "mejor de los casos". La cota inferior $g(n)$ da una interpretación sobre el comportamiento de un programa cuando $f(n)$ representa el tiempo de ejecución consumido al proporcionarse una salida lo más rápidamente posible, esto es lo que se conoce como el "mejor de los casos". La notación asintótica O grande, por otra parte, tiene una utilidad práctica si $f(n)$ mide el tiempo de recorrido en la "peor situación". Θ se emplea con fines interpretativos principalmente cuando $f(n)$ estudia el "caso promedio" en tiempo invertido por un algoritmo.

- La definición 25 resultará más comprensible al lector después de desarrollar algunos ejemplos demostrativos.

Example (3.22)

Demuestre que $1^j + 2^j + \cdots + n^j = \Theta\left(n^{j+1}\right)$ con $j \in \mathbb{N}$.

Solución

En el ejemplo 9 se probó $f(n) = O\left(n^{j+1}\right)$ siendo $f(n) = 1^j + 2^j + \cdots + n^j$. De acuerdo con la definición 25, falta demostrar $f(n) = \Omega\left(n^{j+1}\right)$. Para encontrar una cota inferior de $f(n)$ el estudiante podría pensar en sustituir cada término de la suma por el menor sumando 1^j, en analogía a la técnica usada al buscar una cota superior, sin embargo, se observa que:

$$1^j + 2^j + \cdots + n^j \geq \underbrace{1^j + 1^j + \cdots + 1^j}_{n \text{ veces}} = n \cdot 1^j = n \cdot 1 = n$$

Esta igualdad permite concluir $f(n) = \Omega(n)$, sin embargo, nuestro interés consiste en demostrar $\Omega\left(n^{j+1}\right)$, por este motivo se utilizará otra estrategia de resolución.

Nota

La técnica de trabajo compartida a continuación normalmente funciona en una demostración Ω donde $f(n)$ constituye una suma.

Solución

Iniciaremos. Se piensa entonces en eliminar de la suma "casi la mitad de los términos" para obtener la cota inferior buscada. Con esta intención se sustraen todos los números enteros previos al entero superior de $\frac{n}{2}$ hasta llegar a n, suponiendo que n es un número impar. Este entero se representa como $\left\lceil \frac{n}{2} \right\rceil$. El símbolo $\lceil \rceil$ hace referencia en la literatura a la función parte entera superior, distinta de $\lfloor \rfloor$, al ser esta última la parte entera inferior. Luego, al estar considerando menos términos de la suma original, se halla una cota inferior:

$$1^j + 2^j + \cdots + n^j \geq \left\lceil \frac{n}{2} \right\rceil^j + \cdots + (n-1)^j + n^j$$

Solución

Ahora, se reemplaza cada sumando por el más pequeño $\left\lceil \frac{n}{2} \right\rceil^j$:

$$1^j + 2^j + \cdots + n^j \geq \left\lceil \frac{n}{2} \right\rceil^j + \cdots + \left\lceil \frac{n}{2} \right\rceil^j + \left\lceil \frac{n}{2} \right\rceil^j$$

La última suma tiene $\frac{n+1}{2}$ elementos, lo cual no resulta tan evidente, pero el estudiante lo puede corroborar usando un ejemplo en particular:

$$1^j + 2^j + 3^j + 4^j + 5^j \geq \left\lceil \frac{5}{2} \right\rceil^j + \cdots + 5^j = \underbrace{3^j + 4^j + 5^j}_{\frac{n+1}{2} = \frac{5+1}{2} = \frac{6}{2} = 3} \geq 3^j + 3^j + 3^j$$

$$= \underbrace{3}_{\frac{n+1}{2} = 3} \cdot 3^j$$

Solución

En general, en este tipo de demostraciones asumiendo un n impar, siempre el número de sumandos que quedarán al comenzar en el entero superior será igual a $\frac{n+1}{2}$. Una prueba formal se podría llevar a cabo por inducción matemática, pese a ello, se omitirá esta formalidad para no sobrecargar la exposición de las ideas. Luego:

$$\left\lceil \frac{n}{2} \right\rceil^j + \cdots + \left\lceil \frac{n}{2} \right\rceil^j + \left\lceil \frac{n}{2} \right\rceil^j = \frac{n+1}{2} \left\lceil \frac{n}{2} \right\rceil^j$$

Solución

De donde:

$$\frac{n+1}{2}\left\lceil\frac{n}{2}\right\rceil^j \geq \frac{n}{2}\left\lceil\frac{n}{2}\right\rceil^j \text{ pues } \frac{n+1}{2} > \frac{n}{2}$$

$$\Rightarrow \frac{n+1}{2}\left\lceil\frac{n}{2}\right\rceil^j \geq \frac{n}{2}\left(\frac{n}{2}\right)^j \text{ pues } \left\lceil\frac{n}{2}\right\rceil^j > \left(\frac{n}{2}\right)^j$$

$$\Rightarrow \frac{n+1}{2}\left\lceil\frac{n}{2}\right\rceil^j \geq \frac{n}{2}\cdot\frac{n^j}{2^j} = \frac{n\cdot n^j}{2\cdot 2^j} = \frac{n^{j+1}}{2^{j+1}}$$

Solución

Se ha demostrado, por consiguiente:

$$1^j + 2^j + \cdots + n^j \geq \frac{n^{j+1}}{2^{j+1}} \Rightarrow \left|1^j + 2^j + \cdots + n^j\right| \geq \frac{1}{2^{j+1}} \left|n^{j+1}\right| \qquad (5)$$

al ser $f(n) = 1^j + 2^j + \cdots + n^j$ y $g(n) = n^{j+1}$ funciones positivas. En el presente ejercicio la constante real positiva c de la definición 25 corresponde a $c = \frac{1}{2^{j+1}}$. Finalmente, $f(n) = \Omega(g(n)) = \Omega\left(n^{j+1}\right)$ y por ende, $f(n) = \Theta\left(n^{j+1}\right)$.

Nota

Si n es par fácilmente se deduce la misma desigualdad establecida en 5.

Solución

En dicho caso, la cota inferior de partida iniciaría en $\frac{n}{2}$ en lugar de $\left\lceil \frac{n}{2} \right\rceil$, al ser $\frac{n}{2}$ un número entero:

$$1^j + 2^j + \cdots + n^j \geq \left(\frac{n}{2}\right)^j + \cdots + (n-1)^j + n^j$$

$$\Rightarrow 1^j + 2^j + \cdots + n^j \geq \left(\frac{n}{2}\right)^j + \cdots + \left(\frac{n}{2}\right)^j + \left(\frac{n}{2}\right)^j$$

Solución

La cantidad de sumandos en:

$$\left(\frac{n}{2}\right)^j + \cdots + \left(\frac{n}{2}\right)^j + \left(\frac{n}{2}\right)^j$$

es igual a $\frac{n}{2} + 1$, lo cual tampoco es evidente, pero sí demostrable (se obviará la prueba), por lo que:

$$1^j + 2^j + \cdots + n^j \geq \left(\frac{n}{2} + 1\right)\left(\frac{n}{2}\right)^j$$

$$\Rightarrow 1^j + 2^j + \cdots + n^j \geq \frac{n}{2}\left(\frac{n}{2}\right)^j \text{ pues } \frac{n}{2} + 1 > \frac{n}{2}$$

$$\Rightarrow 1^j + 2^j + \cdots + n^j \geq \frac{n}{2} \cdot \frac{n^j}{2^j} = \frac{n \cdot n^j}{2 \cdot 2^j} = \frac{n^{j+1}}{2^{j+1}}$$

Llegando al mismo resultado.

Example (3.23)

Pruebe $\ln(n!) = \Theta(n \ln n)$. Realice una gráfica en *Wolfram Mathematica* que permita verificar esta notación Θ.

Solución

Ya demostramos $\ln(n!) = O(n \ln n)$ en el ejemplo 11, debemos ahora probar que $\ln(n!) = \Omega(n \ln n)$. En analogía con el procedimiento explicado en el ejemplo anterior, se parte del caso n impar:

$$\ln(n!) = \ln(1 \cdot 2 \cdot \ldots \cdot n)$$

$$= \ln 1 + \ln 2 + \cdots + \ln n \geq \ln \left\lceil \frac{n}{2} \right\rceil + \cdots + \ln(n-1) + \ln n$$

La función logarítmo natural es creciente razón por la cual al sustituir cada sumando por el término menor $\ln \left\lceil \frac{n}{2} \right\rceil$, se tiene:

$$\ln(n!) \geq \ln \left\lceil \frac{n}{2} \right\rceil + \cdots + \ln \left\lceil \frac{n}{2} \right\rceil + \ln \left\lceil \frac{n}{2} \right\rceil$$

Solución

Hay $\frac{n+1}{2}$ elementos en la suma, por lo tanto:

$$\ln\left\lceil\frac{n}{2}\right\rceil + \cdots + \ln\left\lceil\frac{n}{2}\right\rceil + \ln\left\lceil\frac{n}{2}\right\rceil = \frac{n+1}{2}\ln\left\lceil\frac{n}{2}\right\rceil$$

Luego:

$$\ln(n!) \geq \frac{n+1}{2}\ln\left\lceil\frac{n}{2}\right\rceil \geq \frac{n}{2}\ln\left(\frac{n}{2}\right) \text{ pues } \frac{n+1}{2} > \frac{n}{2} \text{ y } \left\lceil\frac{n}{2}\right\rceil > \frac{n}{2}$$

Solución

En $\frac{n}{2}\ln\left(\frac{n}{2}\right)$ se contiene la expresión buscada $n\ln n$. Si fuera comprobable que $\frac{n}{2}\ln\left(\frac{n}{2}\right) \geq \frac{n}{2}\frac{\ln n}{2}$, se completaría la demostración. La correctitud de esta desigualdad no es tan clara pues no exite ninguna propiedad de los logaritmos que lo garantice.

Solución

Por conveniencia, vamos a suponer que la desigualdad se satisface y estudiaremos bajo cuáles condiciones para n :

$$\frac{n}{2}\ln\left(\frac{n}{2}\right) \geq \frac{n}{2}\frac{\ln n}{2} \Rightarrow \frac{n}{2}\ln\left(\frac{n}{2}\right) \geq \frac{n\ln n}{4} \Rightarrow \frac{4n}{2n}\ln\left(\frac{n}{2}\right) \geq \ln n$$

$$\Rightarrow 2\ln\left(\frac{n}{2}\right) \geq \ln n \Rightarrow 2\left(\ln n - \ln 2\right) \geq \ln n \text{ pues } \log_a\left(\frac{x}{y}\right)$$

$$= \log_a x - \log_a y$$

$$\Rightarrow 2\ln n - 2\ln 2 \geq \ln n \Rightarrow 2\ln n - \ln n \geq 2\ln 2 \Rightarrow \ln n \geq \ln 2^2$$

$$\Rightarrow \ln n \geq \ln 4 \Rightarrow e^{\ln n} \geq e^{\ln 4} \Rightarrow n \geq 4 \text{ pues } a^{\log_a x} = x$$

Solución

Luego, se ha demostrado:

$$\ln(n!) \geq \frac{n}{2}\frac{\ln n}{2} = \frac{n \ln n}{4} \ \forall n,\ n \geq 4,\ n \in \mathbb{N} \tag{6}$$

Es decir, la demostración es válida si se asume $n \geq 4$. ¿Contradice esta restricción el concepto de la notación Ω?, la respuesta es no, dado que la definición 25 admite la posibilidad de excluir algunos números naturales (en este ejercicio el 1, 2 y 3). Quedará delegado al alumno probar la desigualdad 6 suponiendo que n es un número par. Al ser $|\ln(n!)| = \ln(n!)$, $|n \ln n| = n \ln n$ y $c = \frac{1}{4}$, en virtud de la definición 25 se concluye, $\ln(n!) = \Omega(n \ln n)$, lo cual implica, $\ln(n!) = \Theta(n \ln n)$.

Solución

Una interpretación gráfica de $\ln(n!) = \Theta(n \ln n)$ se puede obtener en *Mathematica* al emplear el comando `CDFGraficaNA` de la librería **VilCretas**.

In[ ] :=

```
CDFGraficaNA[{Log[n!], n Log[n]}, 0.01, 100, 1000]
```

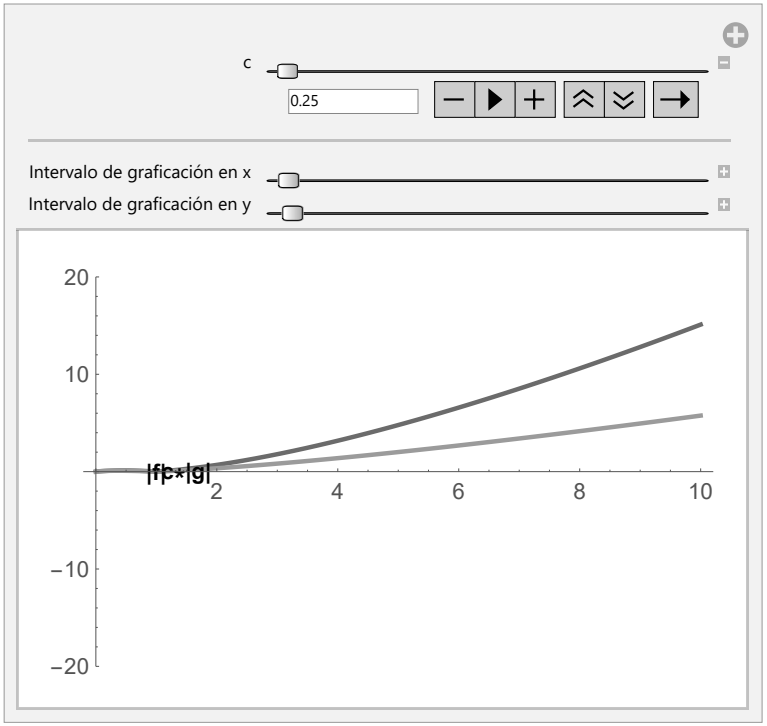
La línea de código anterior genera, como se habia señalado en la página 68, una animación donde es posible cambiar, a través de un deslizador, el valor de la constante real positiva c de la definición de O grande e inclusive de Ω. Si en ella se toma $c = \frac{1}{4} = 0{,}25$ se visualiza la notación Ω ya demostrada, pues la función $c\,|g(n)| = \frac{1}{4}\,|n \ln n|$ permanece por debajo de $|f(n)| = |\ln(n!)|$.

Solución

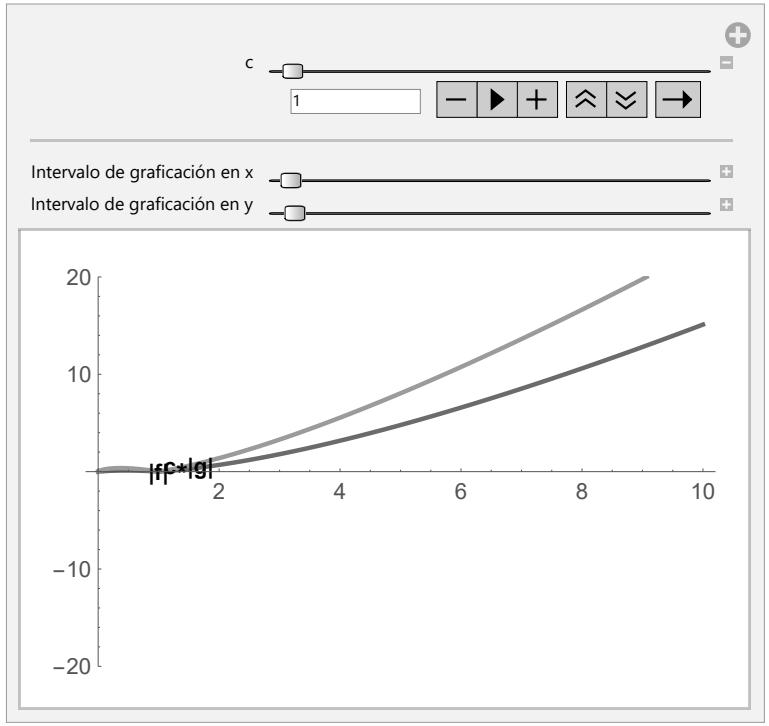
Además, si se toma $c = 1$ se verifica visualmente la notación O grande, al quedar $c\left|g\left(n\right)\right| = 1 \cdot \left|n \ln n\right| = \left|n \ln n\right|$ por encima de $\left|f\left(n\right)\right| = \left|\ln\left(n!\right)\right|$. La verificación simultánea de Ω y O grande en la gráfica, corrobora la presencia de la notación Θ.

Solución

A continuación, se muestra la salida de `CDFGraficaNA[{Log[n!], n Log[n]}, 0.01, 100, 1000]` con $c = \frac{1}{4}$ y $c = 1$:



`CDFGraficaNA` con $c = \frac{1}{4}$



`CDFGraficaNA` con $c = 1$

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-63.zip

- El siguiente teorema generaliza algunas de las propiedades compartidas en el teorema 12, además de proponer otras reglas de interés sobre las notaciones asintóticas Ω y Θ.

Theorem (3.3)

Sean f y g funciones definidas sobre el conjunto de los números naturales:

1. $c = \Theta(1)$, $c \in \mathbb{R}^+$.
2. $f(n) = \Theta(f(n))$ *(propiedad reflexiva).*
3. $f(n) = a_j n^j + a_{j-1} n^{j-1} + \cdots + a_0$ *con* $j \in \mathbb{N}$, $a_k \in \mathbb{R}\ \forall k$, $k \in \mathbb{N}$, $0 \leq k \leq j$ *y* $a_j > 0$ *entonces* $f(n) = \Theta\left(n^j\right)$.
4. $f(n) = O(g(n))$ *sí y solo sí* $g(n) = \Omega(f(n))$ *(regla de simetría transpuesta).*
5. $f(n) = \Theta(g(n))$ *sí y solo sí* $g(n) = \Theta(f(n))$ *(propiedad de simetría).*
6. $$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \begin{cases} c \in \mathbb{R}^+ \Rightarrow f(n) = \Theta(g(n)) \\ 0 \Rightarrow f(n) = O(g(n)) \\ \infty \Rightarrow f(n) = \Omega(g(n)) \end{cases}$$ *(regla del límite).*

- Las reglas de la suma y la multiplicación del teorema 12 también son válidas en Θ aunque se han omitido de las propiedades anteriores.

Comentario sobre el teorema 28

Al observar la generalización de la regla del límite en el teorema 28, cabe destacar que no contradice lo señalado en el teorema 12, donde se afirma que si el valor del límite es un número real positivo o cero, la notación presente es O grande. Esto es compatible con la propiedad 6 del teorema 28, pues al garantizarse la notación Θ siendo positivo el resultado del límite, se implica por definición de Θ, la existencia de la notación asintótica O grande.

Example (3.24)

Demuestre usando propiedades lo enunciado en el ejemplo 27.

Solución

Una solución alternativa para $\ln(n!) = \Theta(n \ln n)$ se logra plasmar empleando la regla del límite del teorema 28. En *Mathematica*:

In[ ] :=

```
Limit[Log[n!]/(n Log[n]), n -> Infinity]
```

Out[ ] =

1

Luego, $1 \in \mathbb{R}^+ \Rightarrow \ln(n!) = \Theta(n \ln n)$. Este procedimiento es tan válido y formal como el desarrollado en el ejemplo 26, claro está, con un ahorro muy significativo de pasos.

Nota

El estudiante podría verse tentado a considerar innecesaria una prueba por definición de Θ, al conjeturar que la propiedad del límite siempre es aplicable. Sin embargo, en ocasiones, la única vía de trabajo posible es la definición y no el uso del límite tal y como también se aclaró con el O grande.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-64.zip

Example (3.25)

Pruebe $\sum_{j=1}^{n} j \ln j = \Theta\left(n^2 \ln n\right)$.

Solución

Recurriendo a la propiedad del límite del teorema 28 con $f(n) = \sum_{j=1}^{n} j \ln(j)$ y $g(n) = n^2 \ln n$, se obtiene en *Wolfram*:

In[ ] :=

```
Limit[Sum[j Log[j], {j, 1, n}]/(n^2 Log[n]), n -> Infinity]
```

Out[ ] =

$1/2$

El valor del límite es $\frac{1}{2} \in \mathbb{R}^+ \Rightarrow \sum_{j=1}^{n} j \ln j = \Theta\left(n^2 \ln n\right)$.

La resolución de este tipo de ejercicios se facilita gracias al uso del software en el cálculo del límite.

- La notación Θ se utiliza, además, en el análisis de algoritmos. Veamos algunos ejemplos.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-65.zip

Example (3.26)

Encuentre una notación Θ para el siguiente algoritmo:

```
CalculaSumaAdicional[n_] := Module[{p = 0}, For[i = 1,
i <= n, For[j = 1, j <= i, p = p + 2; j++]; i++];
Return[p]]
```

Solución

En el código se presentan dos ciclos "for". El primero define el recorrido del segundo. En `i = 1` la asignación `p = p + 2` se efectúa una vez. Luego, si `i = 2`, esta asignación se ejecuta dos veces y así sucesivamente. En conclusión, una función $f(n)$ que cuenta el número de veces que se lleva a cabo la asignación `p = p + 2` es:

$$f(n) = 1 + 2 + \cdots + n$$

Por lo establecido en el ejemplo 26, con $j = 1$:

$$f(n) = 1^1 + 2^1 + \cdots + n^1 = \Theta\left(n^{1+1}\right) = \Theta\left(n^2\right)$$

Solución

Otra manera de llegar al mismo resultado, consiste en determinar una fórmula explícita para $f(n)$. El comando `Sum` de *Mathematica* nos podría ayudar en ese sentido, o bien, se aprecia que $f(n)$ es la conocida suma de *Gauss*, por lo que:

$$f(n) = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = \frac{1}{2}n^2 + \frac{1}{2}n = \Theta\left(n^2\right)$$

El último paso se justifica por la propiedad 3 del teorema 28.

Example (3.27)

Halle una notación Θ para el programa que prosigue:

```
CalculaProductoAdicional[n_] := Module[{p = 1, i = n},
While[i >= 1, p = p*3; i = i/7]; Return[p]]
```

Solución

El ejercicio consiste en encontrar una notación asintótica Θ, sin embargo, veremos que su forma de resolución procede como si se tratara de una notación O grande. En este método el parámetro `i` acumula después de "k" iteraciones el valor $\frac{n}{7^k}$, $k \in \mathbb{N}$, aspecto comprobable en el software:

In[ ] :=

```
RR[{1/7}, {n}, k, inicio -> 0]
```

Out[ ] =

7^{-k} n

Solución

La ejecución del "while" finaliza cuando $\frac{n}{7^k} < 1$, es decir, $n < 7^k$ y por ende, $k > \frac{\ln n}{\ln 7}$. El redondeo hacia arriba o el entero consecutivo de $\frac{\ln n}{\ln 7}$ nos retorna el valor de k. Luego, existe una constante a real positiva, tal que: $k = \frac{\ln n}{\ln 7} + a$. Al comparar en el límite k con $\ln n$, se tiene:

In[ ] :=

```
Limit[(Log[n]/Log[7] + a)/Log[n], n -> Infinity]
```

Out[ ] =

$1/\text{Log}[7]$

Siendo $\frac{1}{\ln 7} \in \mathbb{R}^+ \Rightarrow k = \Theta(\ln n)$. Por lo que, la generalización de la regla del límite planteada en el teorema 28, nos está permitiendo extender la demostración a la notación Θ.

Nota

El estudiante debe notar cómo en los ejemplos 20, 22, 23 y 24, k además de ser un O grande, corresponde a un Θ, pues en todos ellos, los límites calculados dieron como resultado un número real positivo.

Solución

El paquete **VilCretas** posee adicionalmente una instrucción llamada `CompLimit`, que verifica el uso de la propiedad generalizada del límite y podría ayudar al alumno en la revisión de ejercicios resueltos por cuenta propia. Si se emplea este comando con los datos encontrados en este ejemplo, se observa que:

In[ ] :=

```
CompLimit[{Log[n]/Log[7] + a, Log[n]}]
```

Out[ ] =

$1/\text{Log}[7] \to (a\ \text{Log}[7]+\text{Log}[n])/\text{Log}[7] = \Theta(\text{Log}[n])$, Notación theta

`CompLimit` recibe entre llaves las funciones a analizar en el límite y retorna como salida la interpretación asintótica sea Θ, O grande, o bien, Ω de acuerdo con el valor del límite hallado.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-66.zip

- `CompLimit` también es un interesante recurso que aborda el análisis simultáneo de las tres notaciones asintóticas estudiadas en este capítulo. El ejemplo siguiente tiene el propósito de mostrar esta funcionalidad.

Example (3.28)

Conjeture para cuáles valores enteros positivos de j se satisface las notaciones asintóticas $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$ y $f(n) = \Theta(g(n))$, donde:

$$f(n) = \frac{n^{10}}{n+1} \text{ y } g(n) = n^{j+1}$$

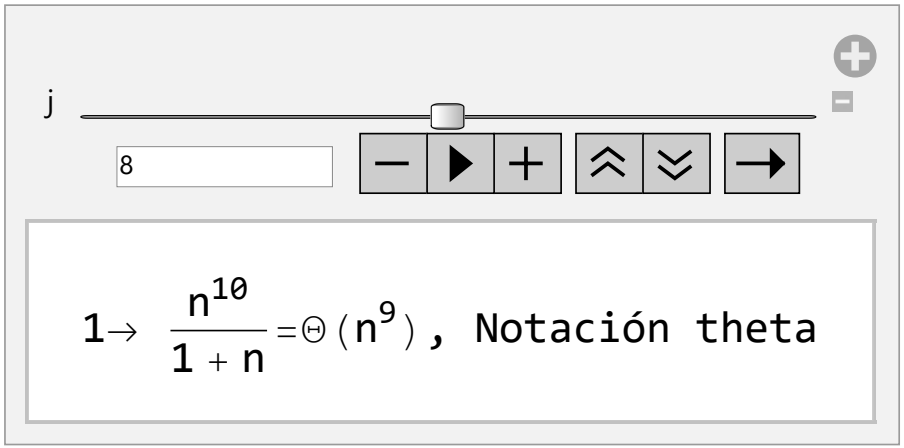
Solución

El comando `CompLimit` en funciones que dependen de n y de j, genera automáticamente una animación haciendo variar j de 1 a 1000 donde se devela el comportamiento asintótico en cada caso. `CompLimit` integra la opción `jvalor->Valor` por si se desea cambiar la variación del parámetro j a un número distinto de 1000. En este ejemplo:

In[ ] :=

```
CompLimit[{n^10/(n + 1), n^(j + 1)}, jvalor -> 15]
```

Out[ ] =



Solución

Al "jugar" con el deslizador de j se infiere que si $1 \leq j \leq 7$ se presenta $f(n) = \Omega(g(n))$, si $j = 8$ se da $f(n) = \Theta(g(n))$ y si $j \geq 9$ ocurre $f(n) = O(g(n))$, $j \in \mathbb{N}$.

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/File-67.zip

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Cuadernos/Algoritmos.pdf.rar

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/Quiz_algoritmos.rar

Abra un sitio web

https://www.symbaloo.com/mix/vilcretasalgoritmos

¡Recuerde resolver los ejercicios asignados!

Descargue un archivo

https://www.escinf.una.ac.cr/discretas/Archivos/Algoritmos/Exercises.zip

enrique.vilchez.quesada@una.cr
http://www.escinf.una.ac.cr/discretas