Ficha 27

Estrategias de Resolución de Problemas: Divide y Vencerás

1.] Aplicaciones de la recursividad: Introducción a los *gráficos fractales*.

Hemos indicado que la recursividad debe ser usada con cuidado ya que aplicada con criterio incorrecto puede llevar a situaciones de excesivo (e innecesario) consumo de memoria (como se vio para el caso del factorial), o bien a implementaciones cuyos tiempos de ejecución resultan inaceptables (como vimos para el caso de la sucesión de Fibonacci).

Pero también hemos indicado que ciertos casos generales la recursión es una excelente herramienta para planteo de soluciones a problemas que de otro modo serían extremadamente difíciles desde el punto de vista de la complejidad del código fuente. En este tipo de problemas, el consumo extra de memoria que hace la recursión está justificado y el tiempo de ejecución será aceptable.

Como ya se dijo, una de las áreas en que la recursión es recomendable es el procesamiento de *estructuras de datos no lineales* como los *árboles* o los *grafos*. Y otra de esas áreas es la *generación y tratamiento de figuras y gráficos fractales*, que es el tema sobre el que está centrado el desarrollo de esta sección. [1]

Una *figura fractal* es aquella que se compone de versiones más simples y pequeñas de la misma figura original, y curiosamente existen numerosos ejemplos de formaciones fractales en la naturaleza, como se ve en las imágenes de la figura que sigue:

Figura 1: Algunos ejemplos de formaciones *fractales naturales*.[1]



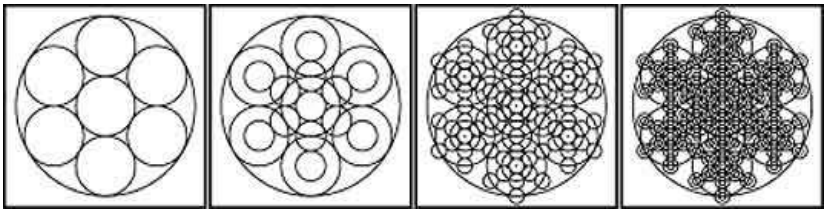
Muchas figuras fractales naturales pueden imitarse y reproducirse artificialmente mediante algoritmos recursivos. Pero también puede pensarse directamente en generar figuras y

[1] Las fuentes de la imágenes que se muestran en la *Figura 1* son las siguientes, tomadas de izquierda a derecha:

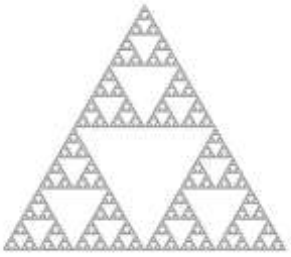
- ✓ http://ztfnews.files.wordpress.com/2010/10/helechom.gif
- ✓ http://www.oddee.com/_media/imgs/articles/a302_f5.jpg
- ✓ https://cuquialcocer.files.wordpress.com/2012/04/fractal2.jpg

curvas basadas en geometría fractal que no estén inspiradas en figuras naturales[2]. La *Figura 2* provee algunos ejemplos relativamente simples de visualizar:

Figura 2: Algunas gráficas *fractales artificiales* simples.[3]



a.) *Composición fractal con círculos*



c.) *Composición fractal con triángulos*

b.) *Composición fractal con cuadrados*

Todas la gráficas de la *Figura 2* están compuestas por la misma figura geométrica, repetida una y otra vez con diversos tamaños y posiciones siguiendo un patrón. Cada imagen remite de inmediato a una secuencia recursiva gráfica (y cada una puede entenderse como una definición recursiva intuitiva de la forma *"un círculo compuesto por círculos"* o un *"cuadrado compuesto por cuadrados"*, etc.) Y como veremos, esta clase de figuras son especialmente aptas para dibujarse mediante programas recursivos (las gráficas **b** y **c** de la figura anterior están programadas en el proyecto *[F27] Piramide* que acompaña a esta Ficha [programas *caleidoscopio.py* y *triangulo.py*])

2.] Generación de gráficos fractales.

La misma idea que se expuso en el apartado anterior vale para la atractiva pirámide coloreada que se muestra en la *Figura 3* (*página 554*). La figura es una composición fractal formada exclusivamente por cuadrados, de manera que se vayan dibujando unos sobre otros para armar los cuatro soportes de una estructura piramidal vista desde arriba: El gran

[2] Las figuras fractales naturales también son comunes en el cuerpo humano y un ejemplo es el iris del ojo en cuanto la distribución del color y las formaciones internas que pueden observarse en él. Al respecto, una extraña película del año 2014, titulada *I Origins* (conocida en español como *Orígenes*), dirigida por *Mike Cahill* y protagonizada por *Michael Pitt*, *Àstrid Bergès-Frisbey* y *Brit Marling* narra la historia de un investigador de la evolución del ojo humano que hace un descubrimiento colateral sorprendente, justamente en relación al iris… Extraña pero digna de verse, prestándole atención a los detalles.

[3] Las fuentes de los gráficos que se muestran son las siguientes:

a.) http://www.thecamino.com.ar/imagf/spiral08.jpg

b.) http://www.infinitefractal.com/movabletype/blogs/my_blog/cart/Diapositiva3.jpg

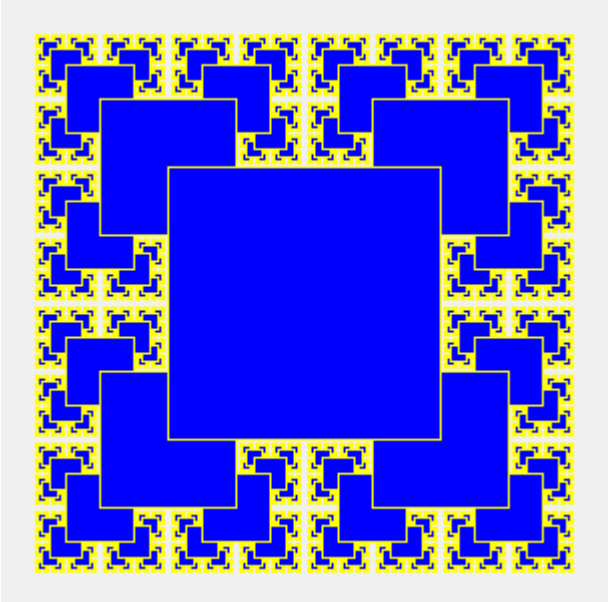
c.) http://2.bp.blogspot.com/-AD-lxWAAzg/Ughm1btCCPI/AAAAAAAAA40/1NBDGQlGhO0/s1600/fractal+6.png

cuadrado azul que se ve en primer plano constituye la cima de la pirámide, y los cuatro soportes se van estirando hacia los cuatro vértices de la gráfica. [1]

La figura no sólo es fractal por estar compuesta por cuadrados que van disminuyendo su tamaño: cada uno de los soportes es a su vez otra pirámide más pequeña, con exactamente la misma forma y proporciones que la pirámide completa.

Esto comienza a sugerir la idea de usar una función recursiva para lograr el dibujo entero. La parte superior de la pirámide muestra un gran cuadrado a modo de tapa, y otros cuatro cuadrados más pequeños (con lados la mitad de largos que el cuadrado mayor) sirviendo como apoyo en cada uno de los vértices. A su vez cada uno de esos cuatro cuadrados tiene otros cuatro más pequeños como soporte, y la idea se repite hasta la base de la pirámide.

Figura 3: ***Pirámide fractal*** **formada por una composición de cuadrados coloreados.**[4]



Python provee numerosos y potentes mecanismos para el diseño de ventanas y componentes gráficos. Si bien no forma parte de los objetivos de este curso hacer un estudio detallado respecto de esos mecanismos, el hecho es que resulta relativamente simple desarrollar un programa que genere mínimamente gráficos como el anterior. [2] [3]

Además, también es cierto que un curso de programación introductorio podría verse muy beneficiado si los estudiantes pudieran utilizar recursos visuales de alto impacto en el corto plazo: la experiencia nos ha mostrado que esos recursos incentivan la creatividad y constituyen un fuerte impulso de motivación: al fin y al cabo, resulta obvio que un programa resulta mucho más atractivo de usar (y de desarrollar...) si el mismo cuenta con ventanas y gráficos que hagan más sencillo su funcionamiento y más accesibles sus recursos. [4]

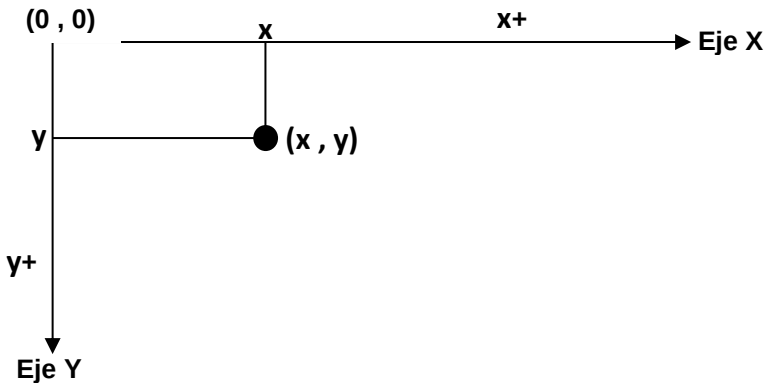
Por lo pronto, el primer elemento que debe quedar claro y dominarse rápidamente es el *esquema de coordenadas de pantalla* que Python (y todo otro lenguaje) supone para la

[4] La idea de la pirámide fractal expuesta en esta sección está inspirada en la presentación del tema en el libro *"Estructuras de Datos en Java"* de *Mark Allen Weiss* (página 179).

gestión de gráficos. El sistema de coordenadas de Python permite localizar a cada punto de luz o *pixel* de la pantalla en forma directa.

Por defecto, el punto superior izquierdo dentro del área de dibujo de una ventana o un lienzo gráfico (conocido como un *canvas*) es el *origen de coordenadas* de ese contenedor gráfico (o sea, el punto de coordenadas *(0, 0)* del contenedor). Si la distancia de un *pixel* al origen del sistema en el eje horizontal se designa como *x*, entonces *x* será la *coordenada de columna* de ese pixel. De forma similar, si la distancia del pixel al origen en el eje vertical se designa como *y*, entonces *y* será la *coordenada de fila* del pixel (ver *Figura 4*).

Figura 4: El sistema de coordenadas de pantalla.



Note que en este sistema de coordenadas, el *número de filas crece hacia abajo en la pantalla* (y no hacia arriba, como quizás el estudiante esperaría dada la costumbre de los ejes cartesianos que normalmente se usan en matemáticas). Esto quiere decir que una fila que se encuentre más cerca del borde superior de la pantalla o ventana, tendrá un número *menor* que otra fila que se encuentre más abajo. El crecimiento en el número de columnas sigue la misma idea que los gráficos cartesianos normales (es decir, los números de columna crecen hacia la derecha en la pantalla).

El programa completo para generar la pirámide de la Figura 3 es el que sigue (y viene acompañando a esta Ficha en el proyecto *[F27] Piramide*):

```
from tkinter import *

__author__ = 'Cátedra de AED'


def pyramid(canvas, x, y, r):
    if r > 0:
        # dibujar recursivamente el soporte inferior izquierdo...
        pyramid(canvas, x-r, y+r, r//2)

        # dibujar recursivamente el soporte inferior derecho...
        pyramid(canvas, x+r, y+r, r//2)

        # dibujar recursivamente el soporte superior izquierdo...
        pyramid(canvas, x-r, y-r, r//2)

        # dibujar recursivamente el soporte superior derecho...
        pyramid(canvas, x+r, y-r, r//2)

        # dibujar en (post-orden) la tapa o cima de la pirámide...
        square(canvas, x, y, r)


def square(canvas, x, y, r):
```

```
    left = x - r
    top = y - r

    right = x + r
    down = y + r

    canvas.create_rectangle((left, top, right, down), outline='yellow', fill='blue')


def render():
    # configuracion inicial de la ventana principal...
    root = Tk()
    root.title("Piramide Fractal")

    # calculo de resolucion en pixels de la pantalla...
    maxw = root.winfo_screenwidth()
    maxh = root.winfo_screenheight()

    # ajustar dimensiones y coordenadas de arranque de la ventana...
    root.geometry("%dx%d+%d+%d" % (maxw, maxh, 0, 0))

    # un lienzo de dibujo dentro de la ventana...
    canvas = Canvas(root, width=maxw, height=maxh)
    canvas.grid(column=0, row=0)

    # sea valor inicial de r igual a un duodécimo del ancho de la pantalla...
    r = maxw // 12

    # coordenadas del centro de la pantalla...
    cx = maxw // 2
    cy = maxh // 2

    # dibujar piramide centrada en (cx, cy) y el cuadrado mayor con lado = 2r.
    pyramid(canvas, cx, cy, r)

    # lanzar el ciclo principal de control de eventos de la ventana...
    root.mainloop()


if __name__ == '__main__':
    render()
```

El programa comienza con una instrucción *import* para dar acceso al contenido del módulo *tkinter*, que es el que básicamente contiene las declaraciones y funciones para manejar ventanas y gráficos:

```
from tkinter import *

__author__ = 'Cátedra de AED'
```

La función *pyramid()* es la que realiza ***el gráfico completo de la pirámide*** usando recursión, pero veremos primero la forma de dibujar un cuadrado simple.

La función *square(canvas, x, y, r)* es la encargada simplemente de dibujar un cuadrado de bordes amarillos y pintado por dentro de color azul.

```
def square(canvas, x, y, r):
    left = x - r
    top = y - r

    right = x + r
    down = y + r

    canvas.create_rectangle((left, top, right, down), outline='yellow', fill='blue')
```

El parámetro *canvas* recibido por la función, es el lienzo o área de dibujo en donde debe graficarse el cuadrado (por el momento, no es necesario saber de dónde sale ese lienzo: simplemente, la función *square()* lo toma como parámetro y dibuja en él el cuadrado pedido).

Los parámetros *x* e *y* son las coordenadas de columna y fila (respectivamente) del pixel que queremos que sea el *centro* del cuadrado dentro del *canvas* donde será dibujado. Y el parámetro *r* es el valor de la mitad de la longitud del lado del cuadrado a dibujar (en pixels), por lo que la longitud total del lado será igual a *2*r*.

Las cuatro primeras líneas de la función *square()* calculan entonces las coordenadas de pixel de las esquinas superior izquierda e inferior derecha del cuadrado (asignando esos valores en las variables (*left*, *top*) y (*right*, *down*)) (asegúrese de entender la naturaleza de estos cuatro cálculos).

Finalmente, en la última línea de la función *square()*, se invoca a la función *canvas.create_rectangle()* para dibujar el cuadrado. Un elemento u objeto de tipo *Canvas* dispone de funciones (o métodos) para dibujar distintos tipos de figuras básicas o primitivas. En este caso, el método *create_rectangle((left, top, right, down), outline='yellow', fill='blue')* recibe como primer parámetro una tupla conteniendo los valores de las coordenadas de los pixels superior izquierdo e inferior derecho del rectángulo a dibujar. El parámetro *outline* (accedido por palabra clave) indica el color con el que será dibujado el borde o perímetro del rectángulo (aquí lo asignamos en amarillo (*'yellow'*)). Y el parámetro *fill* (también accedido por palabra clave) indica el color con el que será pintado por dentro el rectángulo (y aquí lo asignamos en azul (*'blue'*)). [2]

Note que esta función ***no dibuja la pirámide***: sólo dibuja ***un*** cuadrado en las coordenadas centrales *(x, y)* que se le indiquen y con longitud de su lado igual a *2r*. Es la función *pyramid()* la que dibuja ***toda la pirámide***, aplicando recursión e invocando tantas veces como sea necesario a la función *square()*.

La *definición recursiva* del concepto de "pirámide formada por cuadrados" puede enunciarse intuitivamente así:

"Una *pirámide formada por cuadrados de lado máximo 2r* estará vacía si *r* es 0 (no se puede dibujar un lado de longitud 0), o bien contendrá un cuadrado de lado *2r* en la cima que estará apoyado en cuatro soportes. *Pero estos cuatro soportes serán a su vez pirámides formadas por cuadrados cuyos lados serán de longitud máxima r*."

Claramente esta definición es recursiva: el concepto de pirámide aparece en la propia definición. Pues bien: se trata de dibujar *una pirámide formada por cuatro pirámides menores (los soportes) y un cuadrado en la cima*. Y eso es exactamente lo que hace la función *pyramid(canvas, x, y, r)* (los parámetros tienen el mismo significado que en la función *square()*):

```
def pyramid(canvas, x, y, r):
    if r > 0:
        # dibujar recursivamente el soporte inferior izquierdo...
        pyramid(canvas, x-r, y+r, r//2)

        # dibujar recursivamente el soporte inferior derecho...
        pyramid(canvas, x+r, y+r, r//2)

        # dibujar recursivamente el soporte superior izquierdo...
        pyramid(canvas, x-r, y-r, r//2)
```

```
    # dibujar recursivamente el soporte superior derecho...
    pyramid(canvas, x+r, y-r, r//2)

    # dibujar en (post-orden) la tapa o cima de la pirámide...
    square(canvas, x, y, r)
```

La situación trivial se da cuando *r* es 0: en ese caso la función simplemente termina sin hacer nada. Pero si *r* es mayor a 0 (la longitud del lado del cuadrado es por lo menos *2*1 = 2*), entonces se procede al dibujado "desde abajo y hacia la cima", con cuatro invocaciones recursivas (una para cada soporte). Usamos recursión pues tenemos que dibujar *cuatro pirámides que sostengan a la pirámide mayor*…

Los valores *x*, *y*, *r* enviados como parámetro a cada instancia recursiva de la función *pyramid()* se calculan de forma que cada nueva pirámide se dibuje con diferentes coordenadas del centro y una longitud *2r* para el lado del cuadrado en la cima progresivamente menor, dividiendo por 2 a *r* en cada invocación. Esta sucesión de divisiones en algún momento hará que se llegue a un valor de *r* que será igual a 0, y en ese momento se detendrá la recursión.

Observe un detalle: las cuatro invocaciones recursivas se hacen ***antes*** que la invocación a la función *square()*… lo cual tiene sentido, ¡ya que primero deben construirse las paredes antes de colocar el techo! Formalmente, como el proceso de dibujar el cuadrado se hace después que terminan las invocaciones recursivas, entonces tenemos un proceso de dibujado en *post-orden()* (u *orden posterior*) [5] como se indicó en la *Ficha 26*.

La última función que queda en el programa es *render()*, y es la encargada de preparar el contexto visual y gráfico:

```
def render():
    # configuracion inicial de la ventana principal...
    root = Tk()
    root.title("Piramide Fractal")

    # calculo de resolucion en pixels de la pantalla...
    maxw = root.winfo_screenwidth()
    maxh = root.winfo_screenheight()

    # ajustar dimensiones y coordenadas de arranque de la ventana...
    root.geometry("%dx%d+%d+%d" % (maxw, maxh, 0, 0))

    # un lienzo de dibujo dentro de la ventana...
    canvas = Canvas(root, width=maxw, height=maxh)
    canvas.grid(column=0, row=0)

    # sea valor inicial de r igual a un duodécimo del ancho de la pantalla...
    r = maxw // 12

    # coordenadas del centro de la pantalla...
    cx = maxw // 2
    cy = maxh // 2

    # dibujar piramide centrada en (cx, cy) y el cuadrado mayor con lado = 2r.
    pyramid(canvas, cx, cy, r)

    # lanzar el ciclo principal de control de eventos de la ventana...
    root.mainloop()
```

Prácticamente todas las novedades técnicas para la creación y control de la ventana principal y el *canvas* están en esta función [2]. La primera instrucción crea una variable que hemos llamado *root* con la función *Tk()*. Una variable de tipo *Tk* representa una ventana

básica, con bordes definidos, barra de título y controles de minimización y cierre de la ventana. La variable *root* será entonces la que nos permitirá controlar lo que ocurre con la ventana donde se despliega el gráfico. Y la segunda instrucción simplemente asigna la cadena de caracteres que será visualizada en la barra de título de esa ventana:

```
# configuracion inicial de la ventana principal...
root = Tk()
root.title('Piramide Fractal')
```

Las siguientes dos líneas permiten básicamente averiguar la resolución (en pixels) de la pantalla en el momento de ejecutar el programa, para poder luego calcular en forma correcta las dimensiones de la ventana y, sobre todo, las coordenadas del pixel central:

```
# calculo de resolucion en pixels de la pantalla...
maxw = root.winfo_screenwidth()
maxh = root.winfo_screenheight()
```

La función *winfo_screenwidth()* retorna el ancho máximo en pixels admitido por la resolución actual de la pantalla, y la función *winfo_screehight()* hace lo mismo pero con la altura. Al momento de ejecutar este programa para preparar esta Ficha, la resolución era de 1366 pixels de ancho por 768 pixels de alto, y esos dos valores (o los que correspondan) se almacenan en las variables *maxw* y *maxh*.

La quinta línea del programa ajusta lo que se conoce como la *geometría de la ventana principal* (que como vimos, tenemos representada con la variable *root*):

```
# ajustar dimensiones y coordenadas de arranque de la ventana...
root.geometry('%dx%d+%d+%d' % (maxw, maxh, 0, 0))
```

La función *geometry()* permite ajustar el ancho y el alto de la ventana en cualquier momento, y también las coordenadas del pixel superior izquierdo desde donde debe mostrarse esa ventana. La función toma como parámetro una *cadena de caracteres* de la forma *'WxH+x+y'* en la que *W* es el ancho en pixels que queremos que tenga la ventana al mostrarse, *H* es la altura en pixels para la ventana, *x* es la coordenada de columna del pixel superior izquierdo donde queremos que aparezca, e *y* es la coordenada de fila de ese mismo pixel. A modo de ejemplo simple, si hubiésemos hecho algo como:

```
root.geometry('400x300+50+50')
```

entonces la ventana representada por la variable *root* tomaría un ancho de 400 pixels, una altura de 300 pixels, y sería visualizada con su esquina superior izquierda en el pixel de coordenadas (50, 50) de la pantalla.

El problema es que en muchos casos los valores que se quieren asignar están contenidos en variables y no vienen como constantes directas. En nuestro programa, por ejemplo, queremos que la ventana aparezca con un ancho tan grande como el que se tenga disponible en la resolución actual (y como vimos, tenemos ese valor almacenado en la variable *maxw*) y lo mismo vale para la altura (que tenemos en *maxh*). Si quisiéramos esos valores ancho y alto, y hacer aparecer la ventana desde el pixel (0, 0), entonces una invocación de la forma:

```
root.geometry('maxw x maxh + 0+0')
```

simplemente provocaría un error al ejecutarse: la cadena contiene letras donde Python esperaba encontrar dígitos…

La forma de tomar esos valores desde variables, consiste en usar *caracteres de reemplazo* (también llamados *caracteres de formato*) en la cadena. El par de caracteres *%d* se toma como si fuese un único caracter, y se interpreta como que en ese lugar debe ir el valor de una variable de tipo entero (*int*) que se informa más adelante en la propia lista de parámetros de la función. En nuestro programa, la función se está invocando así:

```
root.geometry('%dx%d+%d+%d' % (maxw, maxh, 0, 0))
```

lo cual significa que el primer caracter %d (marcado en rojo en el ejemplo) será reemplazado por el valor de la variable *maxw* que es a su vez la primera en la tupla que aparece luego del signo % (de color negro). En forma similar, el segundo %d (azul) será reemplazado por el valor de la variable *maxh*, que es la segunda en la tupla, y así hasta el final.

Las dos instrucciones siguientes asignan una variable que hemos llamado *canvas* con un elemento u objeto de tipo *Canvas* para hacer allí los gráficos que necesitamos:

```
# un lienzo de dibujo dentro de la ventana...
canvas = Canvas(root, width=maxw, height=maxh)
canvas.grid(column=0, row=0)
```

La función *Canvas()* crea un objeto que representa un lienzo de dibujo. Esa función toma varios parámetros, y en este caso el primero es la variable *root* que representa nuestra ventana principal. Al enviar la ventana como parámetro, le estamos indicando al programa que el *Canvas* que estamos creando (variable *canvas*) estará contenido en esa ventana (variable *root*) y por eso todo lo que se dibuje en *canvas*, aparecerá en la ventana *root*.

Los otros dos parámetros que estamos enviando a la función *Canvas()* se están accediendo por palabra clave, e indican el ancho y el alto en pixels que debe tener el *canvas* dentro de la ventana. En este caso, simplemente hemos asumido que el *canvas* será el único elemento contenido en la ventana *root*, y le hemos fijado un ancho y un alto máximo (los valores de nuestras variables *maxw* y *maxh*). Por supuesto, esto puede ajustarse a voluntad del programador.

Cuando una ventana es creada, todos los elementos que se incluyan dentro de ella serán distribuidos en filas y columnas, como si el interior de la ventana fuese una tabla. La invocación a la función *grid()* que sigue a la creación del canvas, le dice al programa que el *canvas* debe ser ubicado dentro de la ventana en la "casilla" indicada por los parámetros *column* y *row* (en este caso, la casilla [0, 0]).

La instrucción que sigue sólo calcula el valor inicial para *r*, la mitad de la longitud del lado del cuadrado mayor que irá en la cima de la pirámide:

```
# sea un valor inicial de r igual a un duodécimo del ancho del area de dibujo...
r = maxw // 12
```

En este caso, se está tomando un valor igual a la duodécima parte del valor del ancho máximo para la ventana. El estudiante puede cambiar el 12 por otro valor y explorar los efectos que eso produce.

Las dos instrucciones que siguen simplemente calculan las coordenadas del centro de la pantalla:

```
# coordenadas del centro de la pantalla...
cx = maxw // 2
cy = maxh // 2
```

Estas coordenadas son importantes para nuestro programa, ya que le indican en qué posición del *canvas* de la ventana deberá estar centrada la vista de la pirámide, y a partir de esa posición la pirámide se irá construyendo hacia afuera.

La anteúltima instrucción invoca a la función *pyramid()* para dibujar la pirámide completa, cuyo funcionamiento recursivo ya hemos explicado más arriba:

```
# dibujar piramide centrada en (cx, cy) y el cuadrado mayor con lado = 2r.
pyramid(canvas, cx, cy, r)
```

Sólo note que el parámetro *canvas* es el lienzo de dibujo que hemos creado dentro de la ventana, y que la función *pyramid()* lo recibe justamente para poder desarrollar en él nuestro gráfico.

Finalmente, la última instrucción es la que efectivamente pone todo a funcionar:

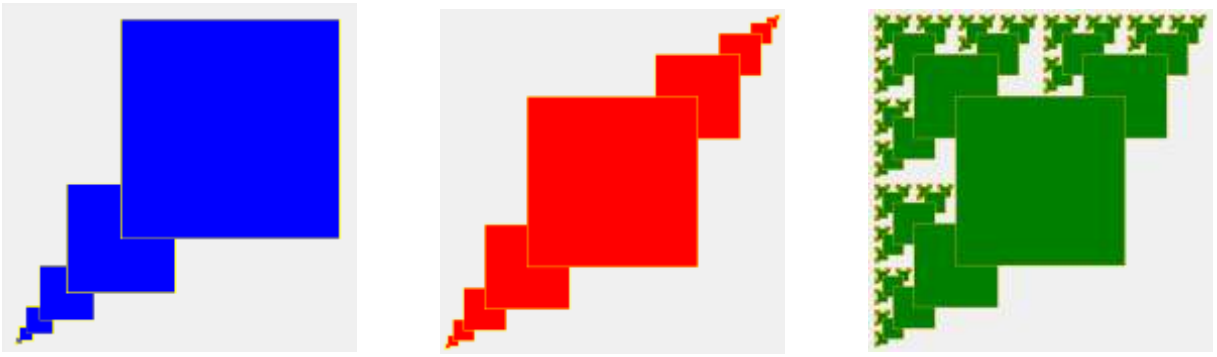
```
# lanzar el ciclo principal de control de eventos de la ventana...
root.mainloop()
```

La ventana representada por la variable *root* se usa para invocar al método *mainloop()*, el cual hace que la ventana se muestre y quede a disposición del usuario. Toda acción que ese usuario aplique sobre la ventana (pasar el mouse sobre ella, minimizarla, presionar el botón de cierre, etc.) genera en el programa lo que se conoce como un *evento*, y el método *mainloop()* se encarga de tomar todos los eventos producidos por la ventana *root* y procesarlos adecuadamente. Por caso, es el método *mainloop()* el que cierra la ventana y da por terminado el programa cuando el usuario presione el botón de cierre ubicado arriba y a la derecha de la ventana.

3.] Pruebas y aplicaciones.

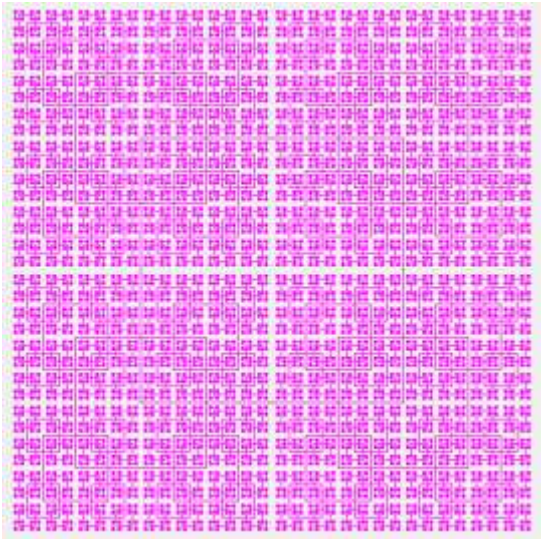
El programa que hemos mostrado en la sección anterior puede ser modificado de distintas maneras para producir diferentes (y muy interesantes) resultados gráficos. Nos permitimos mostrar más abajo algunas gráficas que hemos generado con algunas de esas variantes, y sugerimos al estudiante a que explore y aprenda en qué formas podría generar las mismas figuras efectuando distintos cambios en el código fuente original. Diviértase…

Figura 5: Gráficas producidas con modificaciones simples del programa *[F27] Piramide* [Parte 1]

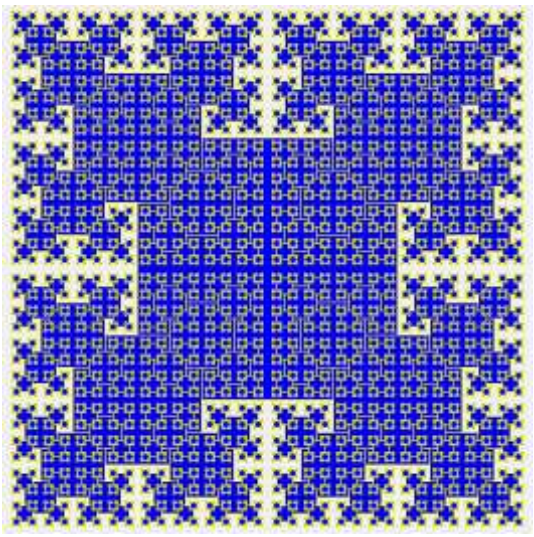


a.) **b.)** **c.)**

Figura 6: Gráficas producidas con modificaciones *no tan simples* del programa *[F27] Piramide* [Parte 2]

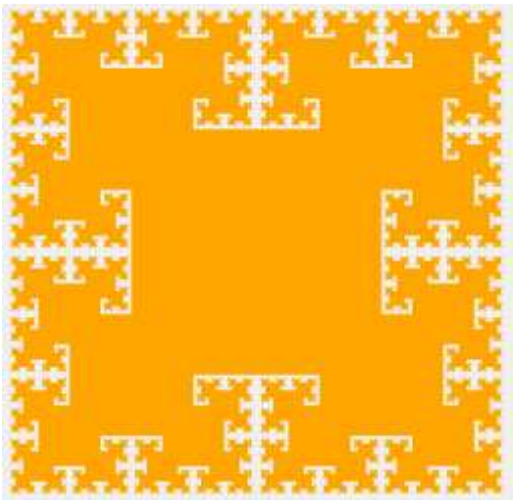


a.)

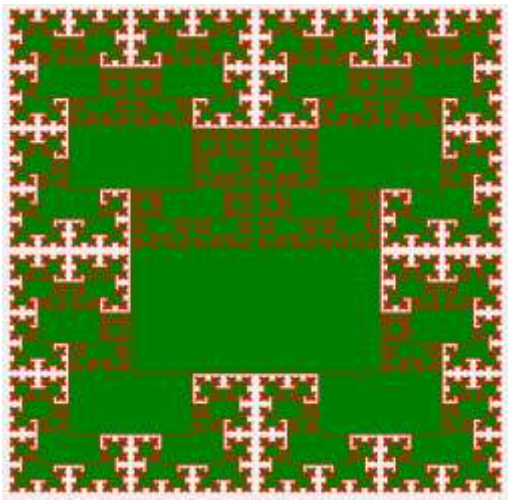


b.)

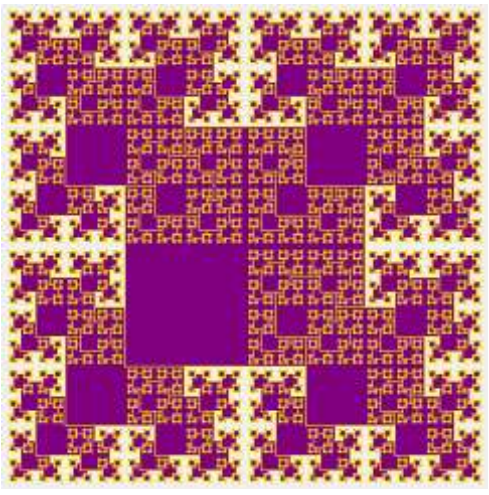
Figura 7: Gráficas producidas con modificaciones *no tan simples* del programa [F27] Piramide [Parte 3]



a.)



b.)



c.)

4.] Aplicaciones de la recursividad: El algoritmo *Quicksort*.

Si se observa el método de *Intercambio Directo* o *Bubblesort* (ver *Ficha 17*), se verá que algunos elementos (los mayores o "más pesados") tienden a viajar más rápidamente que otros (los menores o "más livianos") hacia su posición final en el arreglo. El proceso de "burbujeo" que lleva a cabo ese algoritmo directo es evidentemente *asimétrico*, pues los elementos más grandes se comparan más veces que los más chicos en la misma pasada, haciendo que en esa misma pasada cambien de lugar también más veces que los valores pequeños. Se suele llamar *liebres* y *tortugas* a estos elementos, en obvia alusión a su velocidad de traslado por el vector [5]. El resultado es que el algoritmo necesita muchas más pasadas hasta que se acomode el último de los menores que viaja desde la derecha...

En 1960, un estudiante de ciencias de la computación llamado *Charles Hoare* se haría famoso al presentar en *Communicatons of the ACM* una versión mejorada del *bubble sort*, al cual llamó simplemente *Ordenamiento Rápido* o *Quicksort*... y desde entonces ese método se ha convertido en el más estudiado de la historia de la programación, entre otras cosas por ser hasta ahora en promedio el algoritmo más rápido (aunque hay situaciones en que se comporta bastante mal, esas situaciones de peor caso son raras y poco probables y de todos modos se puede refinar el algoritmo original para que "se proteja" de esos casos y mantenga su buen rendimiento) [6] [1].

La idea es recorrer el arreglo desde los dos extremos. Se toma un elemento *pivot* (que suele ser el valor ubicado al medio del arreglo pero puede ser cualquier otro). Luego, se recorre el arreglo desde la izquierda buscando algún valor que sea *mayor que el pivot*. Al encontrarlo, se comienza una búsqueda similar pero desde la derecha, ahora buscando un *valor menor al pivot*. Cuando ambos hayan sido encontrados, se intercambian entre ellos y se sigue buscando otro par de valores en forma similar, hasta que ambas secuencias de búsqueda se crucen entre ellas. De esta forma, se favorece que tanto los valores mayores ubicados muy a la izquierda como los menores ubicados muy a la derecha, viajen rápido hacia el otro extremo... y todos se vuelven liebres.

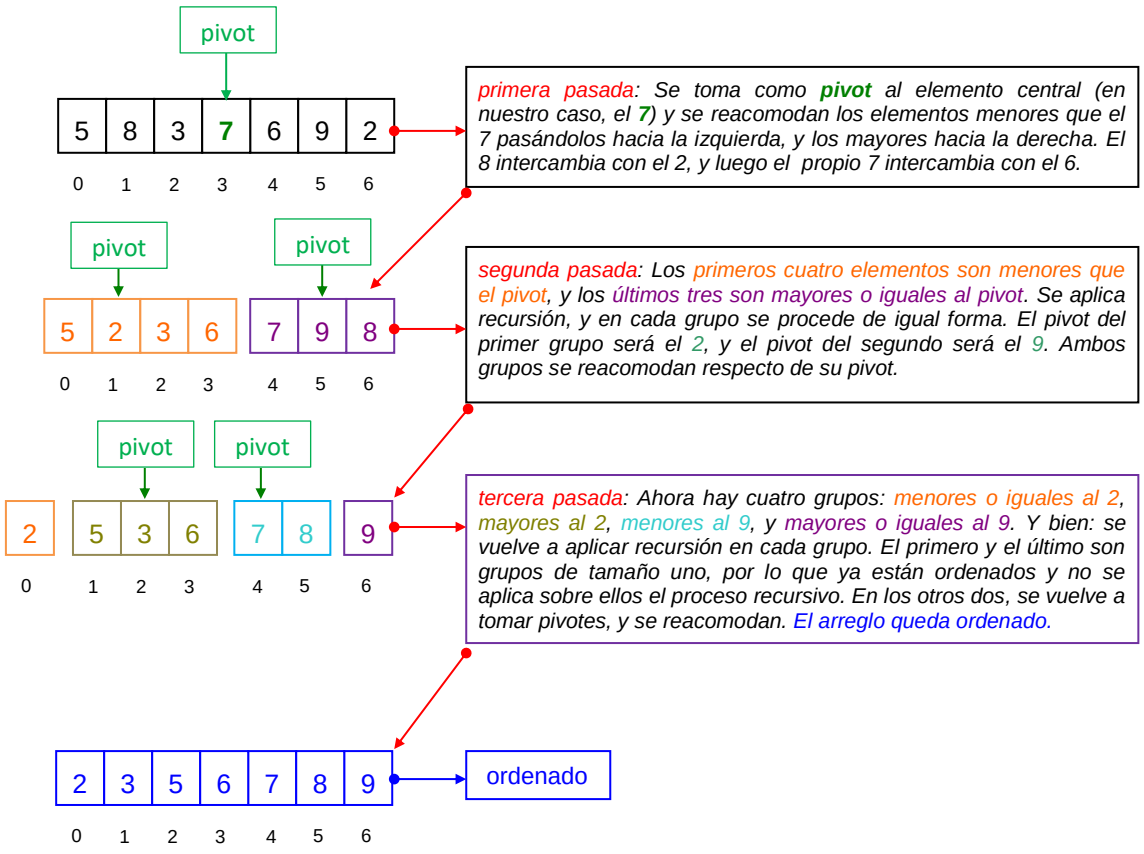
Al terminar esta pasada, se puede ver que el arreglo no queda necesariamente ordenado, pero queda claramente *dividido en dos subarreglos*: *el de la izquierda contiene elementos que son todos menores o iguales al pivot*, y *el de la derecha contiene elementos mayores o iguales al pivot*. Por lo tanto, ahora se puede aplicar exactamente el mismo proceso a cada subarreglo, usando *recursividad*. El mismo método que particionó en dos el arreglo original, se invoca a sí mismo dos veces más, para partir en otros subarreglos a los dos que se obtuvieron recién. Con esto se generan cuatro subarreglos, y con más recursión se procede igual con ellos, hasta que sólo se obtengan particiones de tamaño uno. En ese momento, el arreglo quedará ordenado. El algoritmo procede como se ve en la *Figura 8* (ver *página 564*):

En el proyecto *[F27] Divide y Venceras* que viene con esta Ficha, el módulo *ordenamiento.py* incluye la siguiente función *quicksort()* que implementa el algoritmo (de hecho, el módulo *ordenamiento.py* es básicamente el mismo que ya venía con la *Ficha 17* sobre el tema *Ordenamiento*) [5]:

```
def quick_sort(v):
    # ordenamiento Quick Sort
    quick(v, 0, len(v) - 1)


def quick(v, izq, der):
```

```
    x = get_pivot(v, izq, der)

    i, j = izq, der
    while i <= j:
        while v[i] < x and i < der:
            i += 1
        while x < v[j] and j > izq:
            j -= 1
        if i <= j:
            v[i], v[j] = v[j], v[i]
            i += 1
            j -= 1

    if izq < j:
        quick(v, izq, j)

    if i < der:
        quick(v, i, der)


def get_pivot(v, izq, der):
    # calculo del pivot: elemento central de la partición [izq, der]
    x = v[int((izq + der) / 2)]
    return x
```

Figura 8: Esquema general de funcionamiento del algoritmo *Quicksort*.



El mismo proyecto contiene el programa *main.py*, que a través de un menú de opciones permite crear y ordenar un arreglo seleccionando el algoritmo de ordenamiento a aplicar (incluyendo al *Quicksort* que acabamos de mostrar). Otra vez, el programa *main.py* es esencialmente el mismo que ya se presentó con la *Ficha 17*.

5.] La estrategia *Divide y Vencerás* para resolución de problemas.

El algoritmo *Quicksort* que hemos analizado en la sección anterior es un ejemplo de aplicación de una *estrategia de resolución de problemas* muy conocida y estudiada, que se designa como estrategia *Divide y Vencerás*, por la cual los *n* datos del problema estudiado se dividen en subconjuntos de aproximadamente el mismo tamaño (*n/2* por ejemplo), luego se procesa cada uno de esos subconjuntos en *forma recursiva* y finalmente se "unen las partes" que se acaban de procesar para lograr el resultado final. Muchas veces, y dependiendo de factores tales como el *tamaño de cada subconjunto* o la *cantidad de invocaciones recursivas* que se hagan para procesar esos subconjuntos, esta estrategia favorece el diseño de algoritmos muy eficientes en cuanto al tiempo de ejecución [6].

Como toda estrategia general de resolución de problemas, la técnica *divide y vencerás* puede aplicarse en ciertos problemas (cuya estructura interna admita la división en subconjuntos de tamaños similares) para lograr soluciones con mejor o mucho mejor tiempo de ejecución en comparación a soluciones intuitivas de *fuerza bruta*, que suelen consistir en explorar todas y cada una de las posibles combinaciones de procesamiento de los datos de entrada, pero de tal forma que la cantidad de operaciones a realizar aumenta de manera dramática a medida que crece la cantidad *n* de datos [1].

En el algoritmo *Quicksort* la estrategia *Divide y Vencerás* es clara: se trata de dividir el arreglo en dos mitades tales que una de ellas contenga elementos menores al pivot y la otra contenga elementos mayores a ese pivot. Mediante recursión cada uno de esos subarreglos se vuelve a dividir, aplicando en cada uno el mismo proceso de intercambio con respecto al pivot. Finalmente, cuando ya no sea posible volver a particionar por haber llegado a subarreglos de tamaño 1, terminar el proceso dejando los subarreglos ordenados de forma que el arreglo final quede ordenado a su vez.

La división del arreglo en dos para volver a tomar pivotes en cada subarreglo, puede hacerse recursivamente: en cada mitad se calcula un nuevo pivot, se vuelven a trasladar los menores y los mayores; y recursivamente se vuelve a dividir, una y otra vez, obteniendo particiones de tamaños *n/2*, *n/4*, *n/8*, etc., aplicando recursión sobre ellas hasta que la partición a procesar tenga un solo elemento.

¿Qué tan bueno es este algoritmo en tiempo de ejecución? Podemos ver que la función *quick()* comienza haciendo primero el cálculo del pivot y el traslado de los menores y mayores con respecto a ese pivot. Si se analiza con cuidado ese proceso, puede verse que se ejecuta en tiempo lineal (*O(n)*): los dos ciclos más internos mueven los índices *i* y *j* una vez por cada iteración, y aun cuando ambos están dentro de otro ciclo, este ciclo externo solo controla que *i* y *j* no se crucen, pero ***no fuerza*** a los ciclos internos a comenzar nuevamente desde cero (lo que provocaría un rendimiento cuadrático).

Luego de terminar el *proceso de pivoteo*, la función *quick()* hace *dos llamadas recursivas* tomando cada una el subarreglo que corresponda (el primero, delimitado por los índices *izq* y *j*, y el segundo delimitado por *i* y *der*). Para analizar el comportamiento de la función *quick()* en cuanto a su tiempo de ejecución total, veámosla en un esquema de pseudocódigo simplificado:

```
quick(subarreglo actual):
    x = pivot del subarreglo actual
    pivotear: trasladar menores y mayores a x [==> tiempo adicional: O(n)]
    quick(mitad izquierda)
    quick(mitad derecha)
```

Podemos ver que el proceso completo consta de dos llamadas recursivas (cuyo tiempo de ejecución o costo ignoramos ahora) más la ejecución del proceso *pivotear* que en este caso sabemos que es de *tiempo lineal (O(n))*. Esto significa que el tiempo total empleado por la función *quick() completa*, dependerá del tiempo que lleve la ejecución de las dos llamadas recursivas. Necesitamos entonces averiguar cuántas invocaciones recursivas serán realizadas y cuánto tiempo empleará cada una.

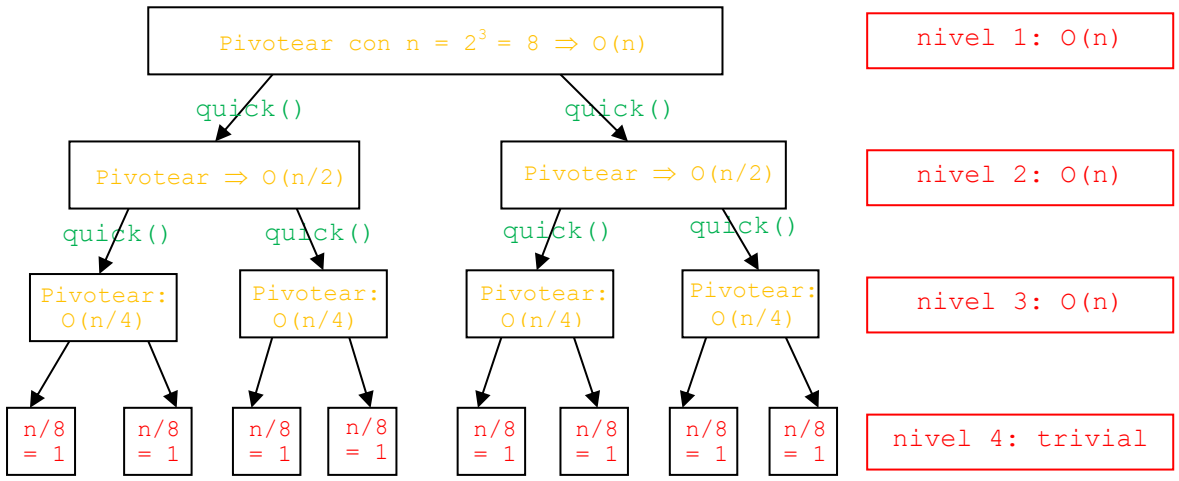
Para ello ayudará un ejemplo. Supongamos que la cantidad de elementos del arreglo es $n = 8$ (para facilitar la explicación suponemos que n es una potencia de 2, pero el análisis no cambia para cualquier otro valor). Al ser invocada por primera vez, la función *quick()* hace el primer proceso de pivoteo sobre el arreglo completo (tiempo: $O(n)$ para cumplir este paso), y luego dos llamadas recursivas (*nivel 1* del gráfico siguiente)

Cada una de las dos instancias recursivas tiene que volver a aplicar el proceso de pivoteo sobre dos subarreglos de tamaño promedio $n/2 = 4$. En cada uno demora $O(n/2)$, por lo que el tiempo conjunto de ambos lleva a $O(n/2) + O(n/2) = O(n)$ en el *nivel 2* del gráfico.

Luego en ese mismo nivel 2 se hacen cuatro llamadas recursivas a *quick()* para otros cuatro subarreglos de tamaño promedio $n/4$, por lo cual el proceso de pivoteo completo para ese nivel es $O(n/4) + O(n/4) + O(n/4) + O(n/4) = O(n)$ también en el *nivel 3*.

En cada nivel de llamadas recursivas, el tamaño promedio de cada subarreglo se divide por 2, y aunque es cierto que el proceso de pivoteo trabaja entonces cada vez menos, el hecho es que ese pivoteo se hace varias veces por nivel, con un tiempo de ejecución combinado que siempre es $O(n)$. ¿Cuántos *niveles* de llamadas a *quick()* tiene el árbol? La respuesta es: $k = 3 = \log_2(8) = \log_2(n)$ [o sea: tantos como se pueda dividir sucesivamente a n por 2 hasta llegar a un cociente de 1 (lo que finalmente es igual a $\log_2(n)$)]. En notación *Big O* se puede prescindir de la base del logaritmo, por lo que la cantidad de niveles con llamadas a *quick()* es $O(\log(n))$.

Figura 9: La estrategia *Divide y Vencerás* aplicada en el algoritmo *Quicksort*.

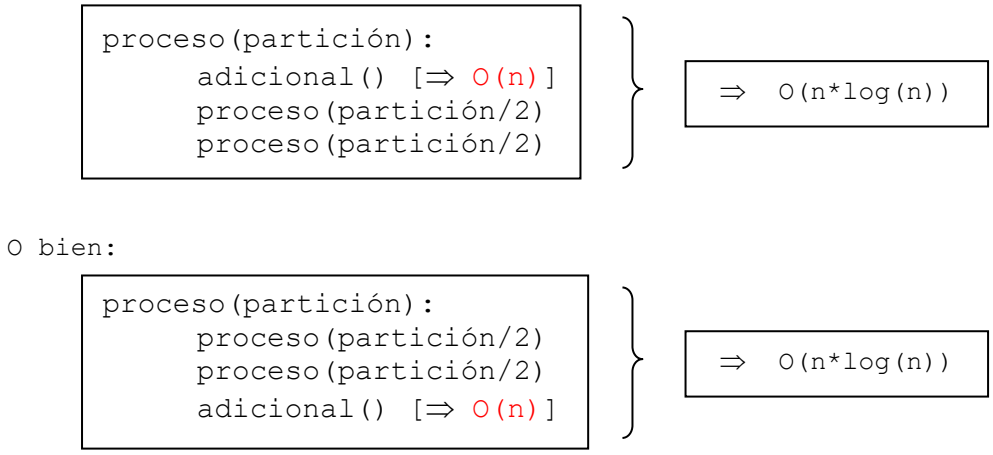


En la gráfica anterior se muestra el nivel 4 del árbol, aunque en realidad ese nivel no llega a generarse: cuando un subarreglo tiene tamaño 1, no hay invocación recursiva para él ya que teniendo un único componente está ya ordenado. Por lo tanto, si la cantidad total de niveles de invocación a *quick()* es $O(\log(n))$ y cada nivel se procesa en un tiempo de ejecución combinado $O(n)$ entonces el tiempo total de ejecución es $O(n*\log(n))$ lo cual es

subcuadrático: puede verse que $n*log(n) < n*n = n^2$ por lo que $O(n*log(n)) < O(n^2)$. Esto prueba que *en promedio*, el algoritmo *Quicksort* ejecuta en tiempo *O(n*log(n))* (decimos *en promedio*, porque el tamaño de cada subarreglo no es constantemente igual a *n/2* en cada nivel, pero si los datos vienen aleatoriamente dispuestos se puede esperar que los subarreglos tengan tamaños que *en promedio* se asemejen a *n/2*, compensando las diferencias).

¿Qué tan general es este resultado? La estrategia *divide y vencerás* se aplicó para diseñar un algoritmo de ordenamiento y al analizarlo hemos obtenido un tiempo de ejecución promedio de *O(n*log(n)* ¿Puede inferirse que siempre será así? ¿Toda vez que se aplique la estrategia *divide y vencerás* se obtendrá una solución con tiempo de ejecución *O(n*log(n))* para el caso promedio o para el peor caso? Se puede probar que la respuesta es ***sí*** *siempre y cuando el algoritmo analizado tenga la misma estructura que el algoritmo quick()*: dos llamadas recursivas aplicadas sobre lotes de datos de tamaño aproximadamente igual a la mitad del lote actual, más *un proceso adicional de tiempo de ejecución lineal (O(n))* (como nuestro proceso de *pivoteo*):

Figura 10: Esquema esencial de un algoritmo *Divide y Vencerás* con tiempo de ejecución *O(n*log(n))*.



6.] Implementación del *Quicksort* mediante selección del pivot por *Mediana de Tres*.

Hemos visto que el algoritmo *Quicksort* ejecuta con tiempo promedio de *O(n*log(n))*. Aún cuando ese comportamiento es muy bueno, existen casos de configuraciones de entrada que pueden llevar al *Quicksort* a un *peor caso* muy malo en cuanto a tiempo de ejecución, dependiendo de la forma en que se tome el pivot: puede probarse que si los elementos del arreglo están distribuidos de forma tal que cada vez que se toma el pivot su valor es el *menor* (o el *mayor*) en el subarreglo considerado, entonces *Quicksort* ejecuta en tiempo $O(n^2)$ (*cuadrático…*) [1].

Intuitivamente puede verse que esto es cierto: si recurrentemente se toma el *menor* o el *mayor* del subarreglo a pivotear, entonces se producirán dos particiones pero una de ellas tendrá un solo elemento y la otra tendrá *n-1* elementos… y al repetirse este patrón, el árbol de llamadas recursivas tendrá *n* niveles en lugar de *log(n)* niveles, por lo que surge el orden cuadrático: si cada nivel se pivotea a un costo de *O(n)*, y tenemos *O(n)* niveles, entonces el tiempo total combinado será $O(n^2)$.

Las distribuciones de datos que pueden causar que *Quicksort* se degrade a un rendimiento cuadrático dependen de la forma en que se selecciona el pivot. Algunos programadores suelen elegir como pivot al *primer elemento del subarreglo analizado* (o bien suelen elegir el *último*). Sin embargo, estas dos formas simples de selección del pivot son justamente las que *maximizan la posibilidad de caer en un rendimiento cuadrático*: si el arreglo estuviese ya ordenado, el algoritmo demoraría un tiempo de orden cuadrático en no hacer nada más que comparaciones, sin intercambio alguno. Por lo tanto, es una muy mala idea tomar como pivot a alguno de los elementos de los extremos en cada partición [6] [1].

La variante que hemos mostrado implementada en las secciones anteriores de esta Ficha, toma como pivot al *elemento central del subarreglo analizado*, y esta es una estrategia muy estable incluso si el arreglo ya estuviese ordenado (en este caso, se invierte un tiempo de *O(n*log(n))* en recorrer el arreglo, sin hacer intercambios). Sin embargo, aún puede ocurrir que aparezca alguna distribución de datos en la cual el elemento central elegido siempre sea el menor o el mayor. Hemos dicho que esa distribución es altamente improbable, pero aún así muestra que la estrategia de selección del pivot podría mejorarse todavía más.

La estrategia más usada se conoce como *selección del pivot por mediana de tres*, y permite eliminar por completo la posibilidad de caer en el peor caso, incluso para distribuciones de datos muy malas o adversas. Esta variante consiste en seleccionar como pivot al *valor mediano* entre *el primero*, *el último* y *el central* del subarreglo analizado (la *mediana* de un conjunto de *n* elementos, es el valor *x* tal que la mitad de los elementos del conjunto son menores a *x*, y la otra mitad son mayores). La forma obvia de obtener el valor mediano de un conjunto es ordenar ese conjunto y luego limitarse a tomar el valor que haya quedado en el centro [1].

En la implementación del *Quicksort por Mediana de Tres*, en cada partición se toma el valor del extremo izquierdo, el valor del extremo derecho y el valor central de la partición, se los ordena entre ellos, y se toma como pivot al que finalmente haya quedado en la casilla central. De esta forma, se garantiza que nunca se tomará como pivot al mayor o al menor, evitando el peor caso antes mencionado. También se suele aconsejar que la *mediana de tres* se tome entre tres elementos seleccionados aleatoriamente dentro de cada partición, lo cual efectivamente lleva a tiempos de ejecución ligeramente mejores.

Por otra parte, se han ensayado otras variantes tomando como pivot al *valor mediano entre cinco* o más componentes del subarreglo, pero las pruebas de rendimiento no han mostrado una mejora significativa en el algoritmo (además de ser más complicadas de programar). Por lo tanto, la estrategia de *selección de pivot por mediana de tres* se ha consolidado como una de las más usadas en la implementación del *Quicksort* (tomando o no los tres elementos en forma aleatoria).

Para finalizar este breve análisis, mostramos la implementación del algoritmo *Quicksort por Mediana de Tres*, tomando la mediana entre el primero, el central y el último de cada partición. Dejamos para los alumnos la implementación con la variante de tomar los tres elementos en forma aleatoria:

Problema 60.) *Implementar el algoritmo Quicksort, pero de forma que la selección del pivot en cada partición se realice por la técnica de la* ***Mediana de Tres****.*

Discusión y solución: Lo único que debemos definir es la forma en que puede obtenerse la mediana entre los tres elementos seleccionados (el primero, el central y el último) de cada

partición. El resto del algoritmo es exactamente el mismo que ya hemos mostrado en esta misma Ficha.

Si el arreglo a ordenar es *v*, y la partición en la que se debe tomar el pivot está delimitada por los índices *izq* y *der*, entonces la siguiente función simple reordena los tres elementos *v[izq]*, *v[der]* y *v[central]*, dejando en *v[central]* al valor mediano (que finalmente será retornado):

```
def get_pivot_m3(v, izq, der):
    # calculo del pivot: mediana de tres...
    central = (izq + der) // 2
    if v[der] < v[izq]:
        v[der], v[izq] = v[izq], v[der]
    if v[central] < v[izq]:
        v[central], v[izq] = v[izq], v[central]
    if v[central] > v[der]:
        v[central], v[der] = v[der], v[central]
    return v[central]
```

Los detalles de funcionamiento de esta función son simples y directos, por lo que dejamos su análisis para el estudiante. El algoritmo completo se implementa con las siguientes funciones (incluida la anterior) que forman parte del proyecto *[F27] Divide y Venceras* que acompaña a esta Ficha:

```
def quick_sort_m3(v):
    # ordenamiento Quick Sort
    quick_m3(v, 0, len(v) - 1)


def quick_m3(v, izq, der):
    x = get_pivot_m3(v, izq, der)

    i, j = izq, der
    while i <= j:

        while v[i] < x and i < der:
            i += 1

        while x < v[j] and j > izq:
            j -= 1

        if i <= j:
            v[i], v[j] = v[j], v[i]
            i += 1
            j -= 1

    if izq < j:
        quick(v, izq, j)

    if i < der:
        quick(v, i, der)


def get_pivot_m3(v, izq, der):
    # calculo del pivot: mediana de tres...
    central = int((izq + der) / 2)

    if v[der] < v[izq]:
        v[der], v[izq] = v[izq], v[der]
```

```
    if v[central] < v[izq]:
        v[central], v[izq] = v[izq], v[central]

    if v[central] > v[der]:
        v[central], v[der] = v[der], v[central]

    return v[central]
```

El programa *main()* del proyecto *[F27] Divide y Venceras* es esencialmente el mismo que ya se mostró en la Ficha 17: un menú que permite crear un arreglo de tamaño *n* con valores aleatorios, y numerosas opciones para ordenar el arreglo con cualquiera de los algoritmos que hemos visto. La diferencia es que ahora se incorpora una opción más para lanzar el *Quicksort por Mediana de Tres*.

Anexo: Temas Avanzados

En general, cada Ficha de Estudios podrá incluir a modo de anexo un capítulo de *Temas Avanzados*, en el cual se tratarán temas nombrados en las secciones de la Ficha, pero que requerirían mucho tiempo para ser desarrollados en el tiempo regular de clase. El capítulo de *Temas Avanzados* podría incluir profundizaciones de elementos referidos a la programación en particular o a las ciencias de la computación en general, como así también explicaciones y demostraciones de fundamentos matemáticos. En general, se espera que el alumno sea capaz de leer, estudiar, dominar y aplicar estos temas aún cuando los mismos no sean específicamente tratados en clase.

a.) La Función de Ackermann.

La *Función de Ackermann* es una función recursiva matemática que debe su nombre a quien la postuló en 1926: *Wilhelm Ackermann* [7]. La *Función de Ackermann* se basa en aplicar un operador simple (la suma) combinado con numerosas invocaciones recursivas, con la intención de lograr *un crecimiento muy rápido* en la magnitud de los valores calculados. Toda la explicación necesaria para comprender su uso y la forma de programarla, se expone en el problema que sigue:[5]

Problema 61.) *La Función de Ackermann originalmente planteada por Wilhelm Ackermann en 1926 ha sido ajustada a lo largo de los años hasta terminar siendo definida en la forma que se muestra aquí:*

Función de Ackermann

$$A(m, n) \quad (m, n \text{ enteros } m \geq 0, n \geq 0) \begin{cases} = n + 1 & (\text{si } m = 0) \\ = A(m - 1, 1) & (\text{si } m > 0 \text{ y } n == 0) \\ = A(m - 1, A(m, n - 1)) & (\text{si } m > 0 \text{ y } n > 0) \end{cases}$$

Se pide desarrollar un programa que permita calcular el valor de la Función de Ackermann para distintos valores de m y n.

Discusión y solución: La función originalmente planteada por *Ackermann* tenía tres parámetros en lugar de dos. Como dijimos, a lo largo de los años se han ido sugiriendo

[5] La fuente esencial de la explicación que sigue sobre la *Función de Ackermann*, es la *Wikipedia* (especialmente la versión en inglés): https://en.wikipedia.org/wiki/Ackermann_function. También hemos usado una referencia básica contenida en el libro "*Estructuras de Datos con C y C++*" de *Yedidyah Langsam* (y otros), así como alguna cita del libro "*Estructuras de Datos en Java*" del ya citado *Mark Allen Weiss*.

variantes y modificaciones que terminaron conformando una *familia* o *serie de funciones de Ackermann*, basadas en estructuras similares. La definición que mostramos en el enunciado es una variante que fue propuesta por *Rózsa Peter* y *Raphael Robinson*, aunque es común que se la cite como si fuese la original de *Ackermann* [8]. En algunos contextos, la variante de *Peter* y *Robinson* se conoce también como la *Función de Ackermann-Peter*.

La función está definida directamente en forma recursiva, por lo que su programación también es directa y no ofrece mayores problemas (ver el proyecto *F[27] Ackermann* que acompaña a esta Ficha):

```
def ackermann(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))
```

Técnicamente hablando, el problema está resuelto. Sin embargo, conviene tomarse un tiempo para estudiar esta función y sus propiedades. Lo primero que se observa es que sólo la primera parte de la definición es *no recursiva*: si *m = 0 entonces A(0, n) = n + 1*. Esto significa que lo que sea que termine calculando, será el resultado de aplicar ese único operador directo (la suma), para añadir 1 al valor de *n*.

El siguiente programa (incluido en el mismo proyecto *F[27] Ackermann*) calcula diversos valores para la función haciendo que *m* tome todo valor posible en el intervalo [0, 3] y *n* tome valores del intervalo [0, 6]:

```
def test():
    for m in range(4):
        for n in range(7):
            ack1 = ackermann(m, n)
            print('Ackermann(', m, ', ', n, '): ', ack1, '\t-\t', sep='', end='')
        print()

# script principal…
test()
```

Podría parecer entonces que la función entregará resultados que no serán de valor muy alto, ya que todos los cálculos se hacen en base a sumar 1 al valor actual de *n* en cada instancia recursiva. Pero la realidad es que para valores combinados de *m* y *n* relativamente bajos, la función llega a resultados asombrosamente elevados. Mientras el valor de *m* se mantenga menor a 4 y el valor de *n* se mantenga menor a 7, la función devuelve valores de magnitud aceptable, como se puede ver en la siguiente salida generada por el programa anterior:

```
A(0,0): 1   A(0,1): 2   A(0,2): 3   A(0,3): 4   A(0,4): 5    A(0,5): 6    A(0,6): 7
A(1,0): 2   A(1,1): 3   A(1,2): 4   A(1,3): 5   A(1,4): 6    A(1,5): 7    A(1,6): 8
A(2,0): 3   A(2,1): 5   A(2,2): 7   A(2,3): 9   A(2,4): 11   A(2,5): 13   A(2,6): 15
A(3,0): 5   A(3,1): 13  A(3,2): 29  A(3,3): 61  A(3,4): 125  A(3,5): 253  A(3,6): 509
```

Si el valor inicial de *m* es 0, la función directamente retorna *n + 1*, y eso produce que la primera fila de la tabla anterior simplemente contenga la progresión de los números naturales para los resultados. Las dos filas que siguen se mantienen con valores de salida pequeños. Pero la cuarta fila ya comienza a producir resultados bastante mayores. Para *A(3,5)* el resultado ya es 253 y para *A(3, 6)* se obtiene un 509.

Evidentemente, cuando los valores combinados de *m* y *n* comienzan a crecer, la función realiza una cantidad muy alta de invocaciones recursivas, y eso hace que el valor de salida sea muy alto también (aunque se obtenga sólo con la acción de sumar 1…) Si intentamos calcular *A(3,7)* el programa se interrumpe por *overflow de stack* sin llegar a mostrarnos el resultado: la cantidad de instancias recursivas es tan grande que no puede ser soportada por el *Stack Segment*.

Lo anterior lleva a que nos preguntemos si la función está efectivamente bien planteada, o si por el contrario, pudiera estar entrando en una situación de recursión infinita que sea la responsable del overflow de stack. Al fin y al cabo, no parece evidente que *A(m,n)* llegue a terminar de calcularse alguna vez para valores de *m* y *n* bastante mayores que 0, ya que la segunda parte de la definición de la función hace una llamada recursiva, pero la tercera hace algo más complejo aún: *una invocación recursiva compuesta con otra* (la primera toma como parámetro al resultado de la segunda, haciendo una *composición de funciones*).

Sin embargo, un análisis detallado del comportamiento de la función nos lleva a concluir que la función está bien planteada y no produce una regresión infinita: si *m* y *n* son diferentes de cero, ambos son restados en 1 en algún momento durante la cascada recursiva, pero se comienza restando 1 a *n*. Cuando el valor de *n* llega a ser 0, comienza desde allí a restarse 1 a *m*, lo que eventualmente llevará a *m = 0* y a la detención de la cascada recursiva. El tema es que hasta que eso ocurra, se creará una inmensa cantidad de instancias recursivas que hará que los valores retornados sean cada vez mayores, y provocando también el rebalsamiento de stack.

Para terminar de comprender la magnitud de los valores que llega a calcular la función, note que *A(4,0)* vale 13. El valor *A(4,1)* ya pasa a ser 65333 (aunque nuestra función posiblemente ya no pueda calcularlo por producirse un overflow de stack). Y el valor *A(4,2)* es tan inmenso que no puede escribirse sino en forma exponencial: ese valor es igual a $2^{65536} - 3$ (de hecho, este número resulta ser mayor que u^{200} donde *u* es la cantidad total de partículas contenidas en el universo). Y se pone mucho peor: los valores de la función para *m > 4* y *n > 1* son tan extravagantemente enormes que no podrían escribirse nunca en forma normal, ya que la secuencia de dígitos que conforma a cada uno de esos números *no cabe en el universo conocido* (y por supuesto, ni nuestro programa ni ningún otro podría siquiera comenzar a calcular esos valores, ya que mucho antes de llegar al final se produciría un overflow de stack).

Para finalizar, digamos que la *Función de Ackermann* tiene ciertos usos prácticos: por lo pronto, debido a su naturaleza profundamente recursiva, puede utilizarse para testear la habilidad de un compilador en optimizar un proceso recursivo. En general, cuando se compila un programa recursivo, el compilador intenta eliminar la recursión y producir una versión compilada iterativa, que será más eficiente en cuanto a uso de memoria y posiblemente en velocidad de ejecución. La *Función de Ackermann* puede ser entonces uno de los desafíos que se le imponen a un compilador cuando esté en etapa de prueba (o *benchmarking*).

Además, dado que la *Función de Ackermann tomada para m = n (o sea: A(n,n), que a su vez puede simplificarse como A(n))* crece de forma tan violenta, entonces su inversa (es decir, la función que a cada número *y* calculado por *A(n)* le asigna el mismo valor *n* de partida) normalmente designada con la *letra griega alfa*: $\alpha(n) = A^{-1}(n)$ aumenta de manera sorprendentemente lenta. De hecho, la función $\alpha(n)$ retorna valores menores que 5 para

prácticamente cualquier valor de n que se tome como parámetro. Esta propiedad de la *inversa de Ackermann* de crecer tan lentamente, suele usarse en el contexto del *análisis de algunos algoritmos* que se sabe que tienen un tiempo de ejecución que crece de forma muy lenta a medida que aumenta el número de datos.

Créditos

El contenido general de esta Ficha de Estudio fue desarrollado por el *Ing. Valerio Frittelli* para ser utilizada como material de consulta general en el cursado de la asignatura *Algoritmos y Estructuras de Datos* – Carrera de Ingeniería en Sistemas de Información – UTN Córdoba, en el ciclo lectivo 2018.

Actuaron como revisores (indicando posibles errores, sugerencias de agregados de contenidos y ejercicios, sugerencias de cambios de enfoque en alguna explicación, etc.) en general todos los profesores de la citada asignatura como miembros de la Cátedra, y particularmente la ingeniera *Karina Ligorria* que realizó aportes de contenidos, propuestas de ejercicios y sus soluciones, sugerencias de estilo de programación, y planteo de enunciados de problemas y actividades prácticas, entre otros elementos.

Bibliografía

[1] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.

[2] Python Software Foundation, "Python Documentation," 2018. [Online]. Available: https://docs.python.org/3/. [Accessed 24 February 2016].

[3] M. Pilgrim, "Dive Into Python - Python from novice to pro," 2004. [Online]. Available: http://www.diveintopython.net/toc/index.html.

[4] V. Frittelli, R. Teicher, M. Tartabini, J. Fernández and G. F. Bett, "Gestión de Gráficos (en Java) para Asignaturas de Programación Introductiva," in *Libro de Artículos Presentados en I Jornada de Enseñanza de la Ingeniería - JEIN 2011*, Buenos Aires, 2011.

[5] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[6] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.

[7] Wikipedia, "Ackermann function" 2015. [Online]. Available: https://en.wikipedia.org/wiki/Ackermann_function.

[8] Y. Langsam, M. Augenstein and A. Tenenbaum, Estructura de Datos con C y C++, México: Prentice Hall, 1997.