

ALGORITMOS Y ESTRUCTURAS DE DATOS

Book subtitle

Patricio Poblete, Benjamin Bustos, Ivan Sipiran

Created: jue mar 4 16:31:34 -05 2021

Contents

5

1 *Introducción* 5

41

2 *Métodos Matemáticos para el Análisis de Algoritmos* 41

51

3 *Diseño de Algoritmos Eficientes* 51

71

4 *Estructuras de datos elementales* 71

111

5 *Pilas, Colas y Colas de Prioridad* 111

135

6 *Diccionarios* 135

203

7 *Ordenación* 203

223

8 Búsqueda en Texto 223

235

9 Compresión de Datos 235

241

10 Grafos 241*Bibliography* 259

1

1 Introducción

El objetivo de este capítulo es ver los conceptos básicos de programación que luego utilizaremos a lo largo de todo el curso. Esto incluye ver tanto los datos que vamos a manejar, como las instrucciones que operan sobre ellos, así como las técnicas básicas de programación. Para la implementación de los algoritmos que vamos a ver utilizaremos el lenguaje Python, sin perjuicio de que los conceptos que veremos son independientes del lenguaje específico utilizado.

En primer lugar, veremos algunos de los tipos de datos básicos de Python. Las variables cambian de tipo dinámicamente según el tipo del valor que tengan en un instante dado.

Datos numéricos

Python distingue entre números enteros (`int`) y números de punto flotante (`float`).

```
n=5 # int
print(n, type(n))
x=3.14 # float
print(x, type(x))
```

```
5 <class 'int'>
3.14 <class 'float'>
```

Sobre estos datos se pueden ejecutar las operaciones aritméticas habituales.

```
print(n+1)
print(n/2) # noten la diferencia con Python 2.7
print(n//2) # división entera
print(x**2)
```

6
2.5
2
9.8596

Datos lógicos

Los valores de verdad son True y False. Las operaciones lógicas se indican con palabras en lugar de símbolos.

```
t=True
f=False
print(t and f)
print(t and not f)
print(n>0)
```

False
True
True

Strings

Los strings se escriben entre comillas simples o dobles, y existen muchas operaciones definidas para ellos.

```
h="Hola"
print(h)
print(len(h))
m='mundo'
print(h + " " + m)
print(h.upper())
```

Hola
4
Hola mundo
HOLA

Listas

Una lista es una secuencia de datos, posiblemente de distintos tipos, de largo variable.

```
L=[3,2,1]
print(L)
L.append(0)
```

```
print(L)
x=L.pop()
print(L)
print(x)
```

```
[3, 2, 1]
[3, 2, 1, 0]
[3, 2, 1]
0
```

Los elementos de la lista se indexan partiendo desde cero.

```
print(L[0])
print(L[2])
print(L[-1]) # contando desde el extremo derecho
```

```
3
1
1
```

Una lista se puede construir en base a iterar una fórmula:

```
C = [n**2 for n in range(1,7)]
print(C)
```

```
[1, 4, 9, 16, 25, 36]
```

Una lista se puede recorrer a través de sus subíndices:

```
for i in range(0,len(C)):
    print(C[i])
```

```
1
4
9
16
25
36
```

O bien iterando sobre los elementos de la lista:

```
for v in C:
    print(v)
```

```
1
4
```

9
16
25
36

Funciones predefinidas

Hay funciones que están disponibles para ser utilizadas:

```
import math
a=3
b=4
c=math.sqrt(a**2+b**2)
print(c)
```

5.0

Funciones

Uno puede definir sus propias funciones:

```
def hipotenusa(a,b):
    import math
    c=math.sqrt(a**2+b**2)
    return c
print(hipotenusa(3,4))
```

5.0

Expresiones condicionales

Una expresión de la forma

```
valor_si_verdadero if condicion else valor_si_falso
```

permite por ejemplo definir funciones por tramos, como por ejemplo:

$$|x| = \begin{cases} -x & \text{si } x < 0 \\ x & \text{en caso contrario} \end{cases}$$

```
def valor_absoluto(x):
    return -x if x<0 else x
print(valor_absoluto(-5), valor_absoluto(5))
```

5 5

Diccionarios

Un diccionario es similar a una lista, pero los elementos se indexan con datos no necesariamente numéricos. Por ejemplo, supongamos que queremos convertir “piedra,” “papel” y “tijera” a una representación numérica 0, 1, 2, respectivamente:

```
p_a_n = {"piedra":0, "papel":1, "tijera":2} # para convertir de palabra a número
print(p_a_n["papel"])
```

```
1
```

La conversión inversa se puede hacer simplemente con una lista:

```
n_a_p = ["piedra", "papel", "tijera"] # para convertir de número a palabra
print(n_a_p[1])
```

```
papel
```

Con esto podemos hacer una función en que el programa juegue “cachipún” contra el usuario:

```
def juega():
    from random import randint
    programa=randint(0,2)
    usuario=p_a_n[input("¿piedra, papel o tijera? ")]
    resultado="Empate" if programa==usuario else\
        "Gana Usuario" if (programa+1)%3==usuario else "Gana Programa"
    print("Usuario juega", n_a_p[usuario])
    print("Programa juega", n_a_p[programa])
    print("Resultado:", resultado)
```

```
juega()
```

```
¿piedra, papel o tijera? piedra
Usuario juega piedra
Programa juega tijera
Resultado: Gana Usuario
```

Instrucción condicional: if

La instrucción if permite elegir entre distintas alternativas de instrucciones a ejecutar.

Encontrar el máximo entre dos valores

```
def max2(a, b):
    if a>b:
        m=a
    else:
        m=b
    return m
print(max2(3,7))
```

7

Esto se puede generalizar a 3 valores, pero el resultado no es muy elegante:

Encontrar el máximo entre tres valores

```
def max3(a, b, c):
    if a>b:
        if a>c:
            m=a
        else:
            m=c
    else:
        if b>c:
            m=b
        else:
            m=c
    return m
print(max3(4,3,7))
```

7

Una mejor alternativa la podemos obtener si vamos obteniendo aproximaciones sucesivas al máximo:

Encontrar el máximo entre dos valores

```
def max2(a, b):
    m=a
    if b>m:
        m=b
    return m
print(max2(3,7))
```

7

Esto se generaliza de manera mucho más simple a tres (o más) valores:

Encontrar el máximo entre tres valores

```
def max3(a, b, c):
    m=a
    if b>m:
        m=b
    if c>m:
        m=c
    return m
print(max3(4,3,7))
```

7

Cuando hay preguntas encadenadas, se puede usar la cláusula `elif` (abreviatura de `else if`, pero que no abre un nuevo nivel de indentación):

Dice si un año dado es bisiesto

```
def es_bisiesto(a):
    if a%400==0: # Los múltiplos de 400 siempre son bisiestos
        b=True
    elif a%100==0: # Los demás múltiplos de 100 no son bisiestos
        b=False
    elif a%4==0: # De los restantes, los múltiplos de 4 son bisiestos
        b=True
    else: # Cualquier otro año no es bisiesto
        b=False
    return b
print(es_bisiesto(1900), es_bisiesto(2000), es_bisiesto(2020))
```

False True True

Instrucciones iterativas: for

La instrucción `for` itera con una variable que toma valores de una lista dada. A menudo, esa lista se especifica mediante `range`. Ilustraremos su uso generalizando los algoritmos que vimos antes para encontrar el máximo de un conjunto de dos o tres números, al caso de un conjunto con un número cualquiera de elementos:

Encuentra el máximo de una lista a

```
def maximo(a):
    m=a[0]
    # Al comenzar cada iteración, se cumple que m==max(a[0],...,a[k-1])
    for k in range(1,len(a)):
        if a[k]>m:
```

```

        m=a[k]
    return m
print(maximo([25, 42, 93, 17, 54, 28]))

```

93

El comentario que aparece junto al `for` describe lo que se llama el **invariante** del ciclo, y es muy útil para poder argumentar la correctitud del programa, así como para poder ayudar a su diseño.

El invariante una afirmación lógica que debe ser verdadera cada vez que se intenta iniciar una nueva iteración, lo cual incluye tanto la primera vez como el último intento, en que se detecta que el rango ya se ha agotado y el ciclo termina.

- La validez del invariante la primera vez la deben asegurar las instrucciones previas al ciclo, que se llaman instrucciones de *inicialización*. En este ejemplo, al comenzar el ciclo se tiene que $k = 1$, y por lo tanto trivialmente se cumple que $m = \max(a[0])$.
- Las instrucciones al interior del ciclo (el “cuerpo de ciclo”) parten de la premisa de que el invariante se cumple al inicio, y deben garantizar que se cumpla al final (para dejar todo listo para la próxima iteración). Esto se llama *preservar el invariante*. En este ejemplo, para asegurar que $m = \max(a[0], \dots, a[k])$ sabiendo que ya era cierto que $m = \max(a[0], \dots, a[k-1])$, se debe modificar m solo en el caso de que $a[k]$ sea mayor que m , o dejarlo igual si no.
- Cuando se detecta que el rango se ha agotado, el invariante igualmente se cumple, y ambas cosas juntas deben asegurar que haya logrado el objetivo final deseado. En este ejemplo, cuando k llega a ser igual a $\text{len}(a)$, el invariante implica que $m = \max(a[0], \dots, a[\text{len}(a) - 1])$, o sea, es el máximo de toda la lista.

Ejercicio 1.1

La función `maximo` hace $n - 1$ comparaciones de elementos para encontrar el máximo de un conjunto de tamaño n .

Supongamos que se desea escribir una función `minmax` que al ser llamada con una lista de números, retorne un par ordenado (tupla) (\min, \max) , con el mínimo y el máximo elemento del conjunto, respectivamente. Escriba a continuación esa función haciendo dos pasadas sobre los datos: una para encontrar el mínimo y otra para encontrar el máximo, y pruébela sobre una lista de ejemplo.

```
def minmax(a):
    # escribir la función aquí

    return(minimo,maximo)

# Probarla acá
```

La función anterior debería hacer $2n - 2$ comparaciones de elementos ($2n - 3$ si se evita comparar el elemento seleccionado en la primera pasada). ¿Será posible encontrar el mínimo y el máximo haciendo muchas menos comparaciones?

¡La respuesta es que sí! Veámoslo con un ejemplo. Para simplificar, supongamos que la lista es de largo par:

[45, 21, 34, 67, 55, 89, 44, 12]

Luego comparemos cada elemento que está en una posición par con su vecino de la derecha, e intercambiémoslos de modo que el par quede en orden ascendente (recuerde que las posiciones comienzan desde cero):

[21, 45, 34, 67, 55, 89, 12, 44]

Luego hagamos una pasada solo sobre las posiciones pares para encontrar el mínimo (12), y otra pasada solo entre las posiciones impares para encontrar el máximo (89). ¡Listo!

Programa este nuevo algoritmo, pruébelo y diga cuántas comparaciones hace en total:

```
def minmax(a):
    # escribir la función aquí

    return(minimo,maximo)

# Probarla acá
```

Instrucciones iterativas: while

La instrucción `while` ejecuta instrucciones mientras la condición especificada sea verdadera:

```

# Dice si un número dado es primo o no
def es_primo(n):
    if n==2:
        return True # 2 es primo
    if n%2==0:
        return False # ningún otro par es primo
    k=3
    while k**2<=n: # no es necesario buscar posibles factores más allá de sqrt(n)
        if n%k==0:
            return False # n es divisible por k => no es primo
        k+=2 # probamos solo los impares
    # si no se encontró ningún factor menor que raíz de n, es primo
    return True
print(es_primo(2), es_primo(7), es_primo(9), es_primo(79823492843), es_primo(79823492869))

```

True True False False True

Ejemplo de programación con invariantes: particionar un conjunto

Supongamos que se tiene un conjunto de datos en una lista $a[0], \dots, a[n-1]$ y un valor de corte p , y se desea reordenar los datos dentro de la lista, de modo que queden a la izquierda todos los que son menores que p , y a la derecha los que son mayores. Por simplicidad, supondremos que en la lista no hay ningún valor igual a p . Este es un problema cuya utilidad veremos más adelante, cuando estudiemos el algoritmo de ordenación Quicksort.

La solución clásica para este problema es la de **Hoare**, el autor de Quicksort, y se basa en ir identificando elementos menores o mayores que p , y moviéndolos hacia el extremo izquierdo o derecho de la lista, según corresponda. Esto corresponde al siguiente invariante:

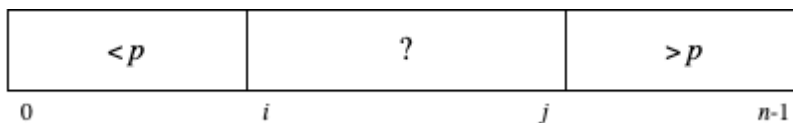


Figure 1: particio-Hoare

En este invariante, i y j son los primeros elementos desconocidos (esto es, aún no identificados como menores o mayores), viniendo desde cada extremo.

```

def particionHoare(a,p):
    # retorna el punto de corte, el número de elementos <p y la lista particionada
    n=len(a)
    (i,j)=(0,n-1) #inicialmente todos los elementos son desconocidos

```

```

while i<=j: # aún quedan elementos desconocidos
    if a[i]<p:
        i+=1
    elif a[j]>p:
        j-=1
    else:
        (a[i],a[j])=(a[j],a[i]) # intercambio
        i+=1
        j-=1
return (p,i,a)

```

Para ayudarnos a verificar que la partición se realiza correctamente, definiremos una función auxiliar:

```

def verifica_particion(t): # imprime y chequea partición
    (p,m,a)=t
    # p=punto de corte, m=número de elementos <p, a=lista completa particionada
    print(a[0:m],p,a[m:])
    print("Partición OK" if (m==0 or max(a[0:m])<p) and (m==len(a) or min(a[m:])>p)
          else "Error")

```

```
verifica_particion(particionHoare([73,21,34,98,56,37,77,65,82,15,36],50))
```

```
[36, 21, 34, 15, 37] 50 [56, 77, 65, 82, 98, 73]
Partición OK
```

```
verifica_particion(particionHoare([73,21,34,98,56,37,77,65,82,15,36],0))
```

```
[] 0 [73, 21, 34, 98, 56, 37, 77, 65, 82, 15, 36]
Partición OK
```

```
verifica_particion(particionHoare([73,21,34,98,56,37,77,65,82,15,36],100))
```

```
[73, 21, 34, 98, 56, 37, 77, 65, 82, 15, 36] 100 []
Partición OK
```

Ejercicio 1.2

Existe un algoritmo alternativo, que resulta en una codificación más sencilla. Este algoritmo, debido a **Lomuto**, se basa en el siguiente invariante:

En este algoritmo, en cada iteración, si $a[j] < p$, se intercambian $a[i]$ con $a[j]$ y se incrementa i , porque ahora hay un elemento más

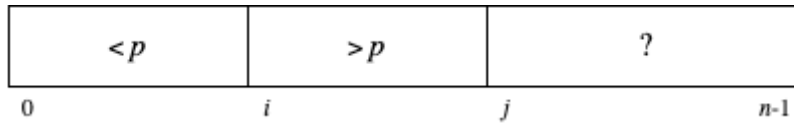


Figure 2: particion-Lomuto

en el grupo de los menores que p . Después de esto, se incrementa j , *incondicionalmente* (¿por qué es correcto hacer eso?).

Programa la partición de Lomuto en el recuadro siguiente y pruébela.

```
def particionLomuto(a,p):
    # retorna el punto de corte, el número de elementos <p y la lista particionada

    # escribir acá el algoritmo de partición de Lomuto

    return (p,i,a)
```

```
verifica_particion(particionLomuto([73,21,34,98,56,37,77,65,82,15,36],50))
```

```
verifica_particion(particionLomuto([73,21,34,98,56,37,77,65,82,15,36],0))
```

```
verifica_particion(particionLomuto([73,21,34,98,56,37,77,65,82,15,36],100))
```

Ejemplo de programación con invariantes: Calcular $y = x^n$

Supongamos que no tuviéramos una operación de elevación a potencia, y que necesitaríamos calcular x^n para n entero no negativo. El algoritmo obvio es calcular $x * x * \dots * x$ (n veces):

```
def potencia(x, n):
    y=1
    for k in range(0,n):
        y*=x
    return y
```

```
print(potencia(2,10))
```

1024

El invariante, esto es, lo que se cumple al comenzar cada nueva iteración es $y = x^k$. Así, al inicio, cuando $k = 0$, se tiene $y = 1$ (inicialización), y al término, cuando $y = n$, se tiene la condición

final buscada. La preservación del invariante consiste en multiplicar y por x , porque así se sigue cumpliendo el invariante cuando k se incrementa en 1.

Este algoritmo ejecuta n multiplicaciones para calcular x^n y, si tomamos en cuenta todo lo que hace, es evidente que demora un tiempo proporcional a n , lo cual escribiremos $O(n)$ y lo leeremos “del orden de n .” (Más adelante definiremos precisamente esta notación, y veremos que podríamos ser más precisos todavía al describir el tiempo que demora un algoritmo)

¿Será posible calcular una potencia de manera más eficiente?

Para ver cómo podríamos mejorar el algoritmo, comenzaremos por reescribirlo de modo que la variable k vaya disminuyendo en lugar de ir aumentando, usando para ello la instrucción `while`:

```
def potencia(x, n):
    y=1
    k=n
    while k>0:
        y*=x
        k-=1
    return y
```

```
print(potencia(2,10))
```

1024

El invariante en este caso sería $y = x^{n-k}$ o, lo que es lo mismo, $y * x^k = x^n$.

El reescribirlo de esta manera nos permite hacer el siguiente truco: vamos a introducir una variable z , cuyo valor inicial es x , y reformular el invariante como $y * z^k = x^n$ y preservarlo aprovechando que $y * z^k = (y * z) * z^{k-1}$:

```
def potencia(x, n):
    y=1
    k=n
    z=x
    while k>0:
        y*=z
        k-=1
    return y
```

```
print(potencia(2,10))
```

1024

Este cambio podría parecer ocioso, pero gracias a él ahora tenemos un grado adicional de libertad: en efecto, podemos modificar la variable z en la medida que eso no haga que el invariante deje de cumplirse.

En particular, una oportunidad de hacer esto aparece cuando k es par. En ese caso, como $z^n = (z^2)^{n/2}$, si elevamos z al cuadrado y al mismo tiempo dividimos k a la mitad, ambos cambios se complementan para hacer que el invariante se preserve. El algoritmo resultante se llama el *algoritmo binario*.

```
def potencia(x, n):
    y=1
    k=n
    z=x
    while k>0:
        if k%2==0: # caso k par
            z=z*z
            k//=2
        else:      # caso k impar
            y*=z
            k-=1
    return y
```

```
print(potencia(2,10))
```

1024

Este algoritmo admite todavía una pequeña optimización. Cuando k se divide por 2, no solo se preserva el invariante, sino que además k sigue siendo > 0 , y por lo tanto no es necesario en ese caso volver a preguntar por la condición del `while`. El algoritmo queda como sigue:

```
def potencia(x, n):
    y=1
    k=n
    z=x
    while k>0:
        while k%2==0: # caso k par
            z=z*z
            k//=2
        y*=z # aquí estamos seguros que k es impar
        k-=1
    return y
```

```
print(potencia(2,10))
```

1024

Este algoritmo (en cualquiera de las dos últimas versiones) se llama el *algoritmo binario*, y es mucho más eficiente que el algoritmo inicial. Cada vez que se da el caso par, k disminuye a la mitad, y eso ocurre al menos la mitad de las veces. Pero si k comienza con el valor n , la operación de dividir por 2 se puede ejecutar a lo más $\log_2 n$ veces, por lo tanto el tiempo total de ejecución es $O(\log_2 n)$, en lugar de $O(n)$.

Decimos que el algoritmo original era de tiempo lineal, y que el algoritmo binario es de tiempo logarítmico. Para n grande, la diferencia de eficiencia es muy grande en favor del algoritmo binario.

Una observación importante es que para que el algoritmo binario funcione, solo es necesario que x y z pertenezcan a un conjunto para el cual hay definida una operación multiplicativa que sea asociativa y que tenga un elemento neutro. Por lo tanto, este algoritmo no solo sirve para elevar a potencia números enteros o reales, sino que además, por ejemplo, para calcular potencias de *matrices*.

¿Por qué se llama “algoritmo binario?”

Existe una relación muy interesante entre el funcionamiento de este algoritmo y la representación binaria del número n .

Para quienes no lo recuerden (o lo estén viendo por primera vez), cuando un número se escribe en base 10, por ejemplo, el número 2019, esto significa

$$(2019)_{10} = 2 \cdot 10^3 + 0 \cdot 10^2 + 1 \cdot 10^1 + 9 \cdot 10^0$$

De manera análoga, un número escrito en base 2, por ejemplo 110101 significa

$$(110101)_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

lo cual equivale a 53 en decimal.

Más en general, un número puede estar escrito en base b :

$$(\dots d_3 d_2 d_1 d_0)_b = \sum_{k \geq 0} d_k b^k$$

donde $0 \leq d_k \leq b - 1$ para todo k .

Volviendo ahora al problema de calcular $y = x^n$, consideremos como ejemplo de $n = 53$. Si escribimos el exponente en binario, tenemos que

$$\begin{aligned}
x^{53} &= x^{(110101)_2} \\
&= x^{1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0} \\
&= x^{2^5} \cdot x^{2^4} \cdot x^{2^2} \cdot x^{2^0} \\
&= x^{32} \cdot x^{16} \cdot x^4 \cdot x^1
\end{aligned}$$

Como vemos, el valor final calculado es el producto de los x elevados a aquellas potencias de 2 que corresponden exactamente adonde hay un dígito 1 en la representación binaria del número n . Veremos que eso es exactamente lo que hace al algoritmo.

Si examinamos el algoritmo, en su primera versión, podemos ver que su estado se puede representar por una terna (k, z, y) , la cual tiene el valor inicial $(n, x, 1)$. En cada iteración tenemos dos casos:

Caso k par:

$$(k, z, y) \rightarrow \left(\frac{k}{2}, z^2, y\right)$$

Caso k impar:

$$(k, z, y) \rightarrow (k - 1, z, yz)$$

Estas reglas adquieren una forma mucho más simple si el número k se considera escrito en binario. En esta base un número es *par* si termina en 0 y es *impar* si termina en 1. Además, restarle 1 a un número impar es equivalente a sustituir el 1 de más a la derecha por un 0, y dividir por 2 un número par es equivalente a eliminar el 0 de más a la derecha.

Con esto, podemos reformular nuestras reglas, suponiendo que el número k se escribe en binario en la forma $(\alpha X)_2$, donde X es el dígito de más a la derecha y α es la secuencia de dígitos que lo preceden. Entonces

Caso k terminado en 0:

$$((\alpha 0)_2, z, y) \rightarrow ((\alpha)_2, z^2, y)$$

Caso k terminado en 1:

$$((\alpha 1)_2, z, y) \rightarrow ((\alpha 0)_2, z, yz)$$

En palabras: la variable z va tomando sucesivamente los valores $x^1, x^2, x^4, x^8, \dots$, y el valor final calculado es x elevado a la suma de aquellas potencias que corresponden exactamente adonde hay un dígito 1 en la representación binaria del número n .

La siguiente tabla muestra la ejecución del algoritmo para $n = 53$:

k	z	y
$(110101)_2$	x^1	1
$(110100)_2$	x^1	x^1

k	z	y
$(11010)_2$	x^2	x^1
$(1101)_2$	x^4	x^1
$(1100)_2$	x^4	$x^{1+4} = x^5$
$(110)_2$	x^8	$x^{1+4} = x^5$
$(11)_2$	x^{16}	$x^{1+4} = x^5$
$(10)_2$	x^{16}	$x^{1+4+16} = x^{21}$
$(1)_2$	x^{32}	$x^{1+4+16} = x^{21}$
$(0)_2$	x^{32}	$x^{1+4+16+32} = x^{53}$

Ejemplo de programación con invariantes: Evaluación de un polinomio

Supongamos que se tiene un polinomio

$$P(x) = \sum_{0 \leq k \leq n} a_k x^k$$

y se desea calcular su valor en un punto dado x .

Una solución trivial se puede obtener directamente de la fórmula anterior:

```
def evalp(a,x):
    """Evalúa en el punto x el polinomio cuyos coeficientes son a[0], a[1],...
    Retorna el valor calculado
    """
    P=0
    for k in range(0,len(a)):
        # Invariante: P=a[0]+a[1]*x+...+a[k-1]*x**(k-1)
        P += a[k]*x**k
    return P
```

Podemos probar esta función evaluando el polinomio

$$P(x) = 5 + 2x - 3x^2 + 4x^3$$

en el punto $x = 2$:

```
print(evalp([5,2,-3,4],2))
```

29

El problema es que este algoritmo puede ser ineficiente. Si el sistema calcula $x**k$ de manera simple, el tiempo total de ejecución sería del orden de n^2 , y si lo calcula usando el algoritmo binario, el tiempo sería del orden de $n \log n$. Veremos que esto se puede reducir a tiempo lineal.

Para esto, vamos a introducir una variable adicional, digamos y , que almacene el valor de x^k necesario para cada iteración. Para preservar este invariante, al final de cada vuelta del ciclo debemos dejarla multiplicada por x , para que tenga el valor correcto al iniciarse la siguiente iteración:

```
def evalp(a,x):
    """Evalúa en el punto x el polinomio cuyos coeficientes son a[0], a[1],...
    Retorna el valor calculado
    """
    P=0
    y=1
    for k in range(0,len(a)):
        # Invariante: P=a[0]+a[1]*x+...+a[k-1]*x**(k-1) and y=x**k
        P += a[k]*y
        y *= x
    return P

print(evalp([5,2,-3,4],2))
```

29

Ejercicio 1.3

Un polinomio se puede evaluar en tiempo lineal sin necesidad de una variable auxiliar si observamos que $P(x)$ se puede factorizar como:

$$P(x) = a_0 + x(a_1 + x(\cdots + x(a_{n-1} + x(a_n)) \cdots))$$

Por ejemplo,

$$\begin{aligned} P(x) &= 5 + 2x - 3x^2 + 4x^3 \\ &= 5 + x(2 + x(-3 + x(4))) \end{aligned}$$

Programa un algoritmo iterativo que evalúe el polinomio utilizando esta idea. Comience desde el paréntesis más interno y vaya avanzando hacia la izquierda. Indique cuál es el invariante que utiliza. El algoritmo resultante se llama la **Regla de Horner**.

```
def evalp(a,x):
    """Evalúa en el punto x el polinomio cuyos coeficientes son a[0], a[1],...
    utilizando la Regla de Horner
    Retorna el valor calculado
    """
```

```
# Escriba aquí su algoritmo
```

```
return P
```

```
print(evalp([5,2,-3,4],2))
```

Numpy y Arreglos

Numpy es la principal biblioteca para computación científica en Python.

Una de las características de Numpy es que provee arreglos multidimensionales de alta eficiencia. Mientras la gran flexibilidad de las listas de Python puede hacer que no sea muy eficiente el acceso a elementos específicos, los arreglos de Numpy aseguran el acceso a cada elemento en tiempo constante. Por esa razón, utilizaremos estos arreglos cuando necesitemos asegurar la eficiencia de la implementación de los algoritmos.

```
import numpy as np

a = np.array([6.5, 5.2, 4.6, 7.0, 4.3])
print(a[2])
```

4.6

```
print(len(a))
```

5

Hay varias formas de crear arreglos inicializados con ceros, unos, valores constantes o valores aleatorios.

```
b = np.ones(10)
print(b)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
c = np.zeros(7, dtype=int)
print(c)
```

```
[0 0 0 0 0 0 0]
```

En los dos ejemplos anteriores mostramos la diferencia que se produce al explicitar el tipo de datos del arreglo. En el primero, obtenemos el *default*, que es flotante, mientras en el segundo forzamos a que sea entero.

```
c = np.full(5, 2)
print(c)
```

```
[2 2 2 2 2]
```

```
d = np.random.random(6)
print(d)
```

```
[0.96086121 0.31006032 0.78951795 0.14162147 0.72680238 0.03081408]
```

También es posible crear y manejar arreglos de varias dimensiones.

```
a = np.array([[1,2,3],[4,5,6]])
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

```
print(a[0,2])
```

```
3
```

```
(m,n)=np.shape(a)
print(m,n)
```

```
2 3
```

```
b = np.zeros((3,3))
print(b)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

```
c = np.eye(3)
print(c)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```


Ejemplos de programación con invariantes: Ordenación

Ordenación por inserción

Supongamos que se tiene un arreglo a , de tamaño n , y queremos reordenar los datos almacenados en su interior de modo que queden en orden ascendente.

El método de **Ordenación por Inserción** se basa en formar en el sector izquierdo del arreglo un subconjunto ordenado, en el cual se van insertando uno por uno los elementos restantes. Para la inicialización, comenzamos con un subconjunto ordenado de tamaño 0, y el proceso termina cuando el subconjunto ordenado llega a tener tamaño n . El invariante se puede visualizar como:

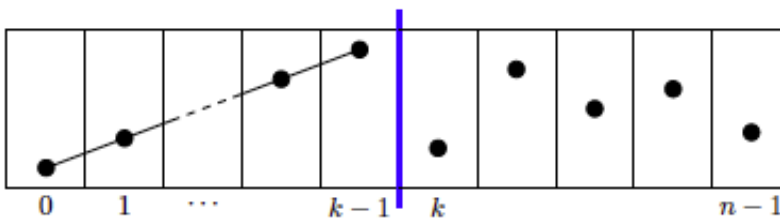


Figure 3: insercion

La variable k indica el tamaño del subconjunto ordenado. Equivalentemente, k es el subíndice del primer elemento todavía no ordenado, y que será el que se insertará en esta oportunidad.

Ordenación Por Inserción

```
def ordena_insercion(a):
    n=len(a)
    for k in range(0,n):
        insertar(a,k)
```

Este algoritmo todavía no es ejecutable, porque falta definir la función `insertar`, que se encarga de tomar $a[k]$ e insertarlo entre los anteriores. La forma más simple de hacer esto es a través de intercambios sucesivos:

Insertar $a[k]$ entre los elementos anteriores preservando el orden ascendente (versión 1)

```
def insertar(a, k):
    j=k # señala la posición del elemento que está siendo insertado
    while j>0 and a[j]<a[j-1]:
        (a[j], a[j-1]) = (a[j-1], a[j])
        j-=1
```

Para poder probar este algoritmo, generemos un arreglo con números aleatorios y ordenémoslo:

```
a = np.random.random(6)
print(a)
ordena_insercion(a)
print(a)
```

```
[0.21698872 0.31136108 0.0676283 0.6468421 0.4082596 0.78409654]
[0.0676283 0.21698872 0.31136108 0.4082596 0.6468421 0.78409654]
```

Si observamos el algoritmo de inserción, podemos ver que en los intercambios siempre uno de los dos elementos involucrados es el que se está insertando, el cual pasa por muchos lugares provisorios hasta llegar finalmente a su ubicación definitiva. Esto sugiere que podemos ahorrar trabajo si en lugar de hacer todos esos intercambios, sacamos primero el elemento a insertar hacia una variable auxiliar, luego vamos moviendo los restantes elementos hacia la derecha, y al final movemos directamente el nuevo elemento desde la variable auxiliar hasta su posición definitiva:

```
# Insertar a[k] entre los elementos anteriores preservando el orden ascendente (versión 2)
def insertar(a, k):
    b=a[k] # b almacena transitoriamente al elemento a[k]
    j=k # señala la posición del lugar "vacío"
    while j>0 and b<a[j-1]:
        a[j]=a[j-1]
        j-=1
    a[j]=b
```

```
a = np.random.random(6)
print(a)
ordena_insercion(a)
print(a)
```

```
[0.63169534 0.51338201 0.75754562 0.11005649 0.81951851 0.5797447 ]
[0.11005649 0.51338201 0.5797447 0.63169534 0.75754562 0.81951851]
```

Para analizar la eficiencia de este algoritmo, podemos considerar varios casos: * Mejor caso: Si el arreglo ya viene ordenado, el ciclo de la función insertar termina de inmediato, así que esa función demora tiempo constante, y el proceso completo demora tiempo $O(n)$, lineal en n . * Peor caso: Si el arreglo viene originalmente en orden decreciente, el ciclo de la función insertar hace el máximo de iteraciones (k), y la suma de todos esos costos da un total de $O(n^2)$, cuadrático en n . * Caso promedio: Si el arreglo viene en orden aleatorio, el número promedio de iteraciones que hace el ciclo de la función insertar es aproximadamente $k/2$, y la suma de todos

esos costos igual da un total de $O(n^2)$. Esto es, el costo promedio también es cuadrático.

Ordenación por Selección

El método de **Ordenación por Selección** se basa en extraer el máximo elemento y moverlo hacia el extremo derecho del arreglo, y repetir este proceso entre los elementos restantes hasta que todos hayan sido extraídos. El invariante se puede visualizar como:

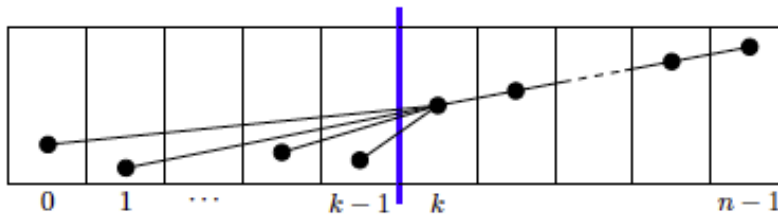


Figure 4: ord-seleccion

La variable k indica el tamaño del subconjunto que todavía falta por procesar. Equivalentemente, es el subíndice del primer elemento que ya pertenece al subconjunto ordenado.

Ordenación por Selección

```
def ordena_seleccion(a):
```

```
    n=len(a)
```

```
    for k in range(n,1,-1): # Paramos cuando todavía queda 1 elemento "desordenado" (¿por qué está bien es
```

```
        j=pos_maximo(a,k) # Encuentra posición del máximo entre a[0],...,a[k-1]
```

```
        (a[j],a[k-1])=(a[k-1],a[j])
```

Encuentra posición del máximo entre a[0],...,a[k-1]

```
def pos_maximo(a, k):
```

```
    j=0 # j señala la posición del máximo
```

```
    for i in range(1,k):
```

```
        if a[i]>a[j]: # Encontramos un nuevo máximo
```

```
            j=i
```

```
    return j
```

Nuevamente, probamos nuestro algoritmo con un arreglo aleatorio:

```
a = np.random.random(6)
```

```
print(a)
```

```
ordena_seleccion(a)
```

```
print(a)
```

```
[0.91088675 0.55781357 0.6422284 0.80222542 0.49612172 0.32394811]
[0.32394811 0.49612172 0.55781357 0.6422284 0.80222542 0.91088675]
```

En este algoritmo, siempre se recorre todo el conjunto de tamaño k para encontrar el máximo, de modo que la suma de todos estos costos de un total de $O(n^2)$, en todos los casos.

Más adelante veremos que hay maneras mucho más eficientes de calcular el máximo de un conjunto, una vez que se ha encontrado y extraído el máximo la primera vez.

Piensen por ejemplo en un típico torneo de tenis, en donde los jugadores se van eliminando por rondas, hasta que en la final queda solo un jugador invicto: el campeón. Si hay n jugadores, ese proceso requiere exactamente $n - 1$ partidos. **Pero** una vez que se ha jugado todo ese torneo, hagamos un experimento mental y pensemos que habría sucedido si el primer día el campeón no hubiera podido jugar por alguna causa. Para determinar quién habría resultado campeón en esas circunstancias (o sea, para encontrar al subcampeón), **no sería necesario repetir todo el torneo, sino solo volver a jugar los partidos en los que habría participado el campeón**. Ese número de partidos es mucho menor a n , y en realidad no es difícil ver que es logarítmico. Y eso puede repetirse para encontrar al tercero, al cuarto, etc., siempre con el mismo costo logarítmico.

Si sumamos todos esos costos, da un total de $O(n \log n)$, en el peor caso.

Lo anterior es una “demostración de factibilidad” de que existen algoritmos de ordenación de costo $O(n \log n)$, más eficientes que $O(n^2)$. Más adelante en el curso veremos algoritmos prácticos que alcanzan esta eficiencia.

Ordenación de la Burbuja

Este algoritmo se basa en ir haciendo pasadas sucesivas de izquierda a derecha sobre el arreglo, y cada vez que encuentra dos elementos adyacentes fuera de orden, los intercambia. Así, el arreglo va quedando cada vez “más ordenado,” hasta que finalmente esté totalmente ordenado.

Analizando el efecto de una pasada de izquierda a derecha, vemos que, aparte de los pequeños desórdenes que pueda ir arreglando por el camino, una vez que el algoritmo se encuentra con el máximo, los intercambios lo empiezan a trasladar paso a paso hacia la derecha, hasta que finalmente queda en el extremo derecho del arreglo. Eso significa que ya ha llegado a su posición definitiva, y no necesitamos volver a tocarlo. Por lo tanto, el algoritmo puede ignorar esos elementos al extremo derecho, los que por construcción están ordenados y son mayores que todos los de la izquierda. Esto lo podemos

visualizar como:

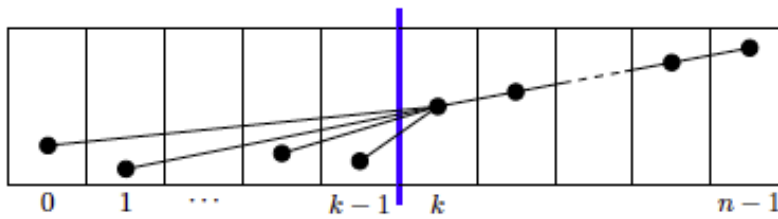


Figure 5: ord-seleccion

¡El mismo invariante que la Ordenación por Selección! Sin embargo, el algoritmo resultante es distinto.

```
# Ordenación de la Burbuja (versión 1)
def ordena_burbuja(a):
    n=len(a)
    k=n # número de elementos todavía desordenados
    while k>1:
        # Hacer una pasada sobre a[0],...,a[k-1]
        # intercambiando elementos adyacentes desordenados
        for j in range(0,k-1):
            if a[j]>a[j+1]:
                (a[j],a[j+1])=(a[j+1],a[j])
        # Disminuir k
        k-=1
```

```
a = np.random.random(6)
print(a)
ordena_burbuja(a)
print(a)
```

```
[0.67980575 0.47893026 0.29847223 0.97584905 0.07176052 0.9138015 ]
[0.07176052 0.29847223 0.47893026 0.67980575 0.9138015 0.97584905]
```

Este algoritmo demora siempre tiempo $O(n^2)$, ¡incluso si se le da para ordenar un arreglo que ya viene ordenado!

No cuesta mucho introducir una variable booleana que señale si en una pasada no se ha hecho ningún intercambio, y usar esa variable para terminar el proceso cuando eso ocurre. Pero hay una manera mejor de modificar el algoritmo para aumentar su eficiencia.

Para esto, introducimos una variable i que recuerda el punto donde se hizo el último intercambio (el cual habría sido entre $a[i-1]$ y $a[i]$). Si a partir de ese punto ya no se encontraron elementos fuera de orden, eso quiere decir que $a[i-1] < a[i]$ y luego a partir de ahí

todos los elementos están ordenados, **hasta el final del arreglo**. Por lo tanto, el invariante se preserva si hacemos $k = i$.

¿Qué pasa si no hubo ningún intercambio? Para este caso, si le damos a la variable i el valor inicial cero, al hacer $k = i$ tendríamos $k = 0$ y el proceso terminaría automáticamente. El algoritmo resultante es el siguiente:

```
# Ordenación de la Burbuja (versión 2)
def ordena_burbuja(a):
    n=len(a)
    k=n # número de elementos todavía desordenados
    while k>1:
        # Hacer una pasada sobre a[0],...,a[k-1]
        # intercambiando elementos adyacentes desordenados
        i=0
        for j in range(0,k-1):
            if a[j]>a[j+1]:
                (a[j],a[j+1])=(a[j+1],a[j])
                i=j+1 # recordamos el lugar del último intercambio
        # Disminuir k
        k=i

a = np.random.random(6)
print(a)
ordena_burbuja(a)
print(a)
```

```
[0.20877349 0.71209819 0.30996714 0.91145577 0.3479466 0.08197456]
[0.08197456 0.20877349 0.30996714 0.3479466 0.71209819 0.91145577]
```

Este algoritmo aprovecha mejor el orden previo que puede traer el arreglo, y en particular ordena arreglos ordenados en tiempo lineal. Pero tanto su peor caso como su caso promedio siguen siendo cuadráticos.

Recursividad

El poder escribir funciones que se llamen a sí mismas es una herramienta muy poderosa de programación. Veremos algunos ejemplos de aplicaciones de este concepto, y más adelante veremos cómo puede conducir al diseño de algoritmos muy eficientes.

Ejemplo: Calcular $y = x^n$

Revisemos nuevamente este problema, pero ahora desde un punto de vista recursivo. Una potencia se puede definir recursivamente de la

siguiente manera:

$$x^n = \begin{cases} x * x^{n-1} & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

lo cual se puede implementar directamente como una función recursiva:

```
def potencia(x,n):
    if n==0:
        return 1
    else:
        return x * potencia(x,n-1)
```

```
print(potencia(2,10))
```

1024

El algoritmo resultante demora tiempo $O(n)$, pero puede mejorarse si el caso n par lo tratamos aparte:

$$x^n = \begin{cases} (x^2)^{n/2} & \text{si } n > 0, \text{ par} \\ x * x^{n-1} & \text{si } n > 0, \text{ impar} \\ 1 & \text{si } n = 0 \end{cases}$$

y la función que lo implementa es:

```
def potencia(x,n):
    if n==0:
        return 1
    elif n%2==0:
        return potencia(x*x,n//2)
    else:
        return x * potencia(x,n-1)
```

```
print(potencia(2,10))
```

1024

El resultado es el algoritmo binario, que demora tiempo $O(\log n)$, en versión recursiva.

Recursividad vs. Iteración

Todo algoritmo iterativo puede escribirse recursivamente. En particular, cualquier ciclo de la forma

```
while C:
    A
```

puede implementarse como

```
def f():
    if C:
        A
        f()
f()
```

Por cierto, en la llamada recursiva se le debe entregar a la función el contexto en que habría operado en la siguiente iteración del ciclo.

Por ejemplo, si queremos imprimir uno por uno los elementos de un arreglo *a*, una forma iterativa de hacerlo sería:

```
def imprimir(a):
    k=0
    while k<len(a):
        print(a[k])
        k+=1
```

```
a=np.random.random(6)
imprimir(a)
```

```
0.7033836563496955
0.6452018723083637
0.7390309665914873
0.7347873655983367
0.7866498089752649
0.6219100789368107
```

En forma recursiva, esto queda como:

```
def imprimir(a):
    imprimir_recursivo(a,0)

def imprimir_recursivo(a,k): # imprimir desde a[k] en adelante
    if k<len(a):
        print(a[k])
        imprimir_recursivo(a,k+1)
```

```
a=np.random.random(6)
imprimir(a)
```



```
0.48983219759773877
0.32335914035360247
0.9701200719987955
0.6556528043143391
0.5783404506167958
0.7588468412552624
```

Este proceso es reversible: cuando una función recursiva lo último que hace es llamarse a sí misma, lo que se llama “recursividad a la cola” (“*tail recursion*”), eso se puede reemplazar por un `while`:

```
def imprimir(a):
    imprimir_recursivo(a,0)

def imprimir_recursivo(a,k): # imprimir desde a[k] en adelante, ahora NO recursivo
    while k<len(a): # reemplazó a "if k<len(a):"
        print(a[k])
        k+=1 # reemplazó a "imprimir_recursivo(a,k+1)"
```

```
a=np.random.random(6)
imprimir(a)
```

```
0.5879986363022844
0.22292489446199648
0.3649081383243461
0.0760636478498492
0.6282506237409337
0.9933423327883073
```

Pero ahora la función `imprimir_recursivo` es llamada desde un único lugar, con `k=0`, y por lo tanto, en ese lugar podemos sustituir la llamada por el código de la función, con lo que el resultado es:

```
def imprimir(a):
    # Esto reemplaza a "imprimir_recursivo(a,0)"
    k=0
    while k<len(a): # reemplazó a "if k<len(a):"
        print(a[k])
        k+=1 # reemplazó a "imprimir_recursivo(a,k+1)""
```

```
a=np.random.random(6)
imprimir(a)
```

```
0.2796059938546853
0.7140250823437458
```

0.27114927650914356

0.1913138660755075

0.3131581564588708

0.13509564673494912

¡Con lo cual hemos vuelto al punto de partida!

Sin embargo, esto solo funciona para eliminar la “recursividad a la cola.” Si hay llamadas recursivas que **no** son lo último que ejecuta la función, no pueden eliminarse con esta receta, y como veremos más adelante, requerirá el uso de una estructura llamada una “pila” (*stack*).

El siguiente ejemplo ilustra un caso en que esto ocurre.

Ejemplo: Torres de Hanoi



Figure 6: Torres de Hanoi

Este puzzle consiste en trasladar n discos desde la estaca 1 a la estaca 3, respetando siempre las dos reglas siguientes: * Solo se puede mover de a un disco a la vez, y * Nunca puede haber un disco más grande sobre uno más chico. Esto se puede resolver recursivamente de la siguiente manera:

Figure 7: Torres de Hanoi

Para mover n discos desde a hasta c (usando la estaca b como auxiliar): * Primero movemos (recursivamente) $n - 1$ discos desde la estaca a a la estaca b * Una vez despejado el camino, movemos 1 disco desde a hasta c * Finalmente, movemos de nuevo (recursivamente) los $n - 1$ discos, ahora desde b hasta c (usando a como auxiliar)

El caso base es $n = 0$, en cuyo caso no se hace nada.

```
def Hanoi(n, a, b, c): # Mover n discos desde "a" a "c", usando "b" como auxiliar
    if n>0:
        Hanoi(n-1, a, c, b)
        print(a, "-->", c) # Mueve 1 disco desde "a" hasta "c"
        Hanoi(n-1, b, a, c)
```

```
Hanoi(3, 1, 2, 3)
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Este algoritmo es muy claro y bastante intuitivo. Si aplicamos la regla de eliminación de “recursividad a la cola,” lo que resulta es un algoritmo equivalente, pero mucho menos trivial de entender:

```
def Hanoi(n, a, b, c): # Mover n discos desde "a" a "c", usando "b" como auxiliar
    while n>0: # reemplaza a "if n>0:"
        Hanoi(n-1, a, c, b)
        print(a, "-->", c) # Mueve 1 disco desde "a" hasta "c"
        n-=1
        (a,b)=(b,a) # reemplaza a "Hanoi(n-1, b, a, c)"
```

```
Hanoi(3, 1, 2, 3)
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

¿Puede usted dar una explicación intuitiva de por qué funciona este algoritmo?

Diagramas de Estados

Consideremos una instrucción iterativa con un invariante “I,” que se inicializa con instrucciones “B,” con condición de continuación “C,” con un cuerpo del ciclo consistente de instrucciones “A” y con

el objetivo de lograr que se cumpla una afirmación lógica "F." Esto tendría la estructura siguiente:

```
B
while C: # invariante I
    A

# al terminar se debería cumplir F
```

Anotando con un poco más de precisión, podemos identificar lo que se cumple en cada punto:

```
B
# aquí se cumple el invariante "I" por primera vez
while C:
    # aquí se cumple "I and C"
    A
    # acá se debe cumplir nuevamente "I"

# al terminar el ciclo se cumple "I and not C"
# y eso debe implicar que se cumple F
```

Esto se puede visualizar de manera más sencilla en un **diagrama de estados** como el siguiente:

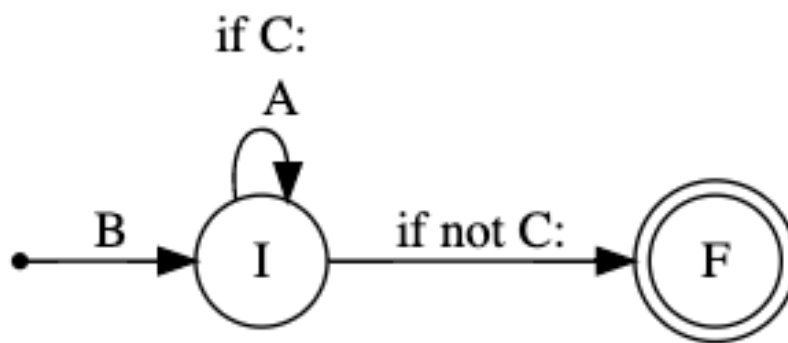


Figure 8: Diagrama de Estados 1

En este diagrama: * Los estados (círculos) representan el estado del proceso en ese momento, descrito por la afirmación lógica correspondiente. En un ciclo, esa afirmación lógica es lo que hemos llamado invariante. * Un doble círculo representa a un estado final. * Las flechas pueden llevar como rótulo un "if" con una condición, que debe cumplirse para que se siga esa flecha, y también pueden estar rotuladas con una instrucción (o un bloque de instrucciones), que se ejecuta al hacer esa transición.

Una ventaja de modelar un algoritmo en base a un diagrama de estados es que no estamos restringidos solo a los diagramas simples que corresponden a ciclos `while`, sino que podemos construir diagramas mucho más complejos, con múltiples estados y transiciones.

Piensen por ejemplo en cómo modelar el funcionamiento de un **cajero automático**:

- El cajero está originalmente en un estado inicial, que es además el estado al cual regresa cada vez que concluye la atención de un cliente.
- Cuando llega un cliente e inserta su tarjeta, el cajero debe verificar que la tarjeta es legible y si no es, debe devolverla con un mensaje de error y volver al estado inicial.
- Si la tarjeta es legible, debe pedir que ingrese el PIN y pasar a otro estado en que espera que el cliente ingrese dicha clave.
- Si la clave es incorrecta, debe pedir que la ingrese de nuevo y seguir en ese estado. Pero eso tiene un límite, porque si el usuario comete muchos errores, el cajero debe dar un mensaje de error, retener la tarjeta y volver al estado inicial.
- Si la clave es correcta, debe mostrar un menú de posibles operaciones y esperar que el cliente seleccione una.
- Si el cliente selecciona “Retirar dinero,” debe pasar a otro estado en que le pregunta el monto que quiere sacar.
- Etc., etc., etc.,

En realidad, el diagrama de estados de un cajero automático es enorme, porque en cada estado hay múltiples opciones que conducen a realizar acciones y trasladarse a otros estados. Además, hay “timeouts” que interrumpen el proceso si una operación se demora demasiado. El diagrama de estados es la herramienta que permite modelar este tipo de procesos complejos.

Ejemplo: Contar palabras

Supongamos que tenemos un string que contiene una frase y queremos contar cuántas palabras contiene. Para simplificar, supondremos que una palabra es cualquier secuencia de caracteres distintos de un espacio en blanco. Por ejemplo, el string

" Algoritmos y Estructuras de Datos "

contiene 5 palabras.

Para resolver este problema, iremos examinando uno a uno los caracteres del string, y para cada uno deberemos decidir si ese carácter es el comienzo de una nueva palabra o no. Esto depende de si estábamos FUERA de una palabra (en cuyo caso sí es el inicio y hay

que incrementar el contador de palabras) o si estábamos DENTRO de una palabra (en cuyo caso no se incrementa).

Esto sugiere tener un diagrama de estados con dos estados (FUERA y DENTRO), más un tercer estado final FIN, al que se llega cuando se agota el string.

El siguiente diagrama modela este proceso, donde el carácter que se está examinando en cada momento es $s[k]$. Para simplificar, suponemos que las transiciones hacia FIN tienen prioridad. Además, como en cada transición se examina un nuevo carácter, dejaremos implícita la inicialización $k = 0$ y el incremento $k + 1$ que hay después de cada transición.

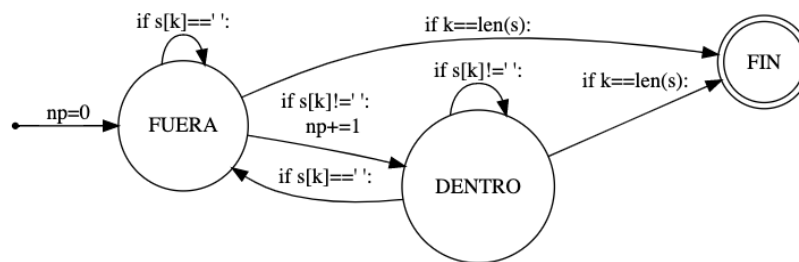


Figure 9: Diagrama de Estados 2

Una vez modelado el proceso mediante un diagrama de estados, debemos escribirlo en forma de un programa. Veremos a continuación que **hay más de una manera de hacerlo**:

Contar palabras, versión 1 con variables de estado

```

def contar_palabras(s):
    np=0
    estado="FUERA"
    for k in range(0, len(s)):
        if estado=="FUERA":
            if s[k]!=' ':
                np+=1
                estado="DENTRO"
        else: # estado=="DENTRO"
            if s[k]==' ':
                estado="FUERA"
    return np
  
```

```

s=input("Escriba frase: ")
print("Hay", contar_palabras(s), "palabras")
  
```

Contar palabras, versión 2 con variables de estado

```
def contar_palabras(s):
    np=0
    estado="FUERA"
    for k in range(0,len(s)):
        if s[k]==' ':
            estado="FUERA"
        else: # s[k]!=' ':
            if estado=="FUERA":
                np+=1
                estado="DENTRO"
    return np
```

```
s=input("Escriba frase: ")
print("Hay", contar_palabras(s), "palabras")
```

Contar palabras, versión sin variables de estado

```
def contar_palabras(s):
    np=0
    k=0
    while k<len(s):
        # Estamos en el estado FUERA
        if s[k]!=' ':
            np+=1
            k+=1
        # Ahora estamos en el estado DENTRO
        while k<len(s) and s[k]!=' ':
            k+=1
        if k==len(s):
            break
        # Por descarte, s[k]==' '
        k+=1
    return np
```

```
s=input("Escriba frase: ")
print("Hay", contar_palabras(s), "palabras")
```

Ejercicio 1.4

Se le llama "Camel Case" a la convención de escribir una frase sin espacios, pero marcando el inicio de cada palabra poniendo su primera letra en mayúscula. Por ejemplo, la frase

" Algoritmos y estructuras de datos "

transformada a Camel Case queda así:

"AlgoritmosYEstructurasDeDatos"

Escriba una función que transforme a Camel Case y pruébela:

```
def CamelCase(s):  
    """Retorna un string conteniendo la versión Camel Case del string s"""  
    # escriba aquí su algoritmo  
  
print(CamelCase("  Algoritmos y  estructuras de  datos  "))
```

2

2 Métodos Matemáticos para el Análisis de Algoritmos

Para cuantificar la eficiencia de los algoritmos, utilizamos funciones que miden, por ejemplo, cuánto tiempo demora un algoritmo en ejecutarse sobre una entrada dada, cuál es su peor caso sobre un conjunto de entradas posibles, o cuánto demora en promedio, suponiendo una cierta distribución de probabilidad de las entradas. Ocasionalmente, estudiaremos el uso de otro tipo de recursos, como por ejemplo la cantidad de memoria utilizada.

Es habitual que los resultados que obtengamos dependan de un parámetro n , que representa el tamaño del problema (por ejemplo, el número de elementos a ordenar, el número de elementos en un conjunto en el que hay que hacer una búsqueda, etc.). Por lo tanto, nuestras funciones serán normalmente *funciones discretas*, esto es funciones cuyo argumento es un número entero no negativo. Como notación, para este tipo de funciones utilizaremos indistintamente una notación de funciones $f(n)$ o de sucesiones f_n .

Tal como en Física el estudio de funciones de variable continua $f(t)$ conduce a ecuaciones diferenciales que luego hay que resolver, acá el estudio de funciones de variable discreta $f(n)$ conducirá a *ecuaciones de recurrencia*, y en este capítulo veremos algunos métodos para resolver ese tipo de ecuaciones.

Notación O

Al trabajar con funciones que pueden ser muy complicadas, en la práctica resulta útil poder ignorar los términos de orden inferior para concentrarse en el que determina la forma en que la función evoluciona cuando $n \rightarrow \infty$. También resulta útil en ese caso poder ignorar factores constantes, para concentrarse en la forma como la función depende de n .

Utilizaremos la notación

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

si existe una constante C y un número n_0 tal que

$$|f(n)| \leq C|g(n)|$$

para todo $n \geq n_0$.

Cuando la notación $O(g(n))$ aparezca en medio de una fórmula, representará a una función que cumple con la condición anterior.

Es importante notar que la notación O provee una cota superior, la cual puede o no ser cercana a la función de la izquierda. Más adelante veremos una notación más ajustada.

Ejemplos

- $3n = O(n)$
- $2 = O(1)$
- $2 = O(n)$
- $3n + 2 = O(n)$

La notación $f(n) = O(g(n))$ es utilizada por la mayoría de los autores, pero hay que usarla con cuidado, porque la igualdad que ahí aparece **no es una relación reflexiva**. En efecto, de $3n = O(n)$ y $2 = O(n)$, **no** podemos deducir que $3n = 2$. Hay que tener presente siempre que lo que aparece a la derecha del signo igual contiene menos información que lo de la izquierda.

Para evitar las posibles confusiones que podrían derivar de este uso no estándar del signo igual, algunos autores prefieren escribir

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

en donde $O(g(n))$ se interpreta como el *conjunto* de todas las funciones que acotan a $f(n)$ de la manera indicada.

Notación Ω

De manera análoga, se puede definir una notación de cota inferior.

Diremos que

$$f(n) = \Omega(g(n))$$

si existe una constante C y un número n_0 tal que

$$|f(n)| \geq C|g(n)|$$

para todo $n \geq n_0$.

Ejemplos

- $3 = \Omega(1)$

- $3n = \Omega(n)$
- $3n = \Omega(1)$
- $3n + 2 = \Omega(n)$

Notación Θ

La notación Θ nos permite especificar el orden exacto de crecimiento de una función.

Diremos que

$$f(n) = \Theta(g(n))$$

si $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$

Ejemplo

- $3n + 2 = \Theta(n)$

Esta notación enfatiza que lo más importante, cuando n crece, es el orden de magnitud de las funciones, y que podemos ignorar constantes multiplicativas y términos de orden inferior para comparar funciones a largo plazo.

Por otra parte, si por alguna razón queremos comparar funciones para n pequeño, ahí toda la información es significativa, y como muestra el siguiente gráfico, en ese rango una solución “ineficiente” puede resultar preferible a otra que solo es eficiente para n grande:

```
%pylab inline
```

Populating the interactive namespace from numpy and matplotlib

```
n=linspace(1,10)
plt.plot(n,n+6, label='$n+6$')
plt.plot(n,log(n)+8, label='$\log\{n\}+8$')
leg=plt.legend(loc='best')
```

```
n=linspace(1,10)
plt.plot(n,n*n, label='$n^2$')
plt.plot(n,2*n*log(n)+5, label='$2n\log\{n\}+5$')
leg=plt.legend(loc='best')
```

Ecuaciones de Recurrencia

Al estudiar la eficiencia de algoritmos, a menudo podremos escribir ecuaciones que relacionan el valor de la función en n con los valores de la función en $n - 1$, $n - 2$, etc. Para que estas ecuaciones tengan

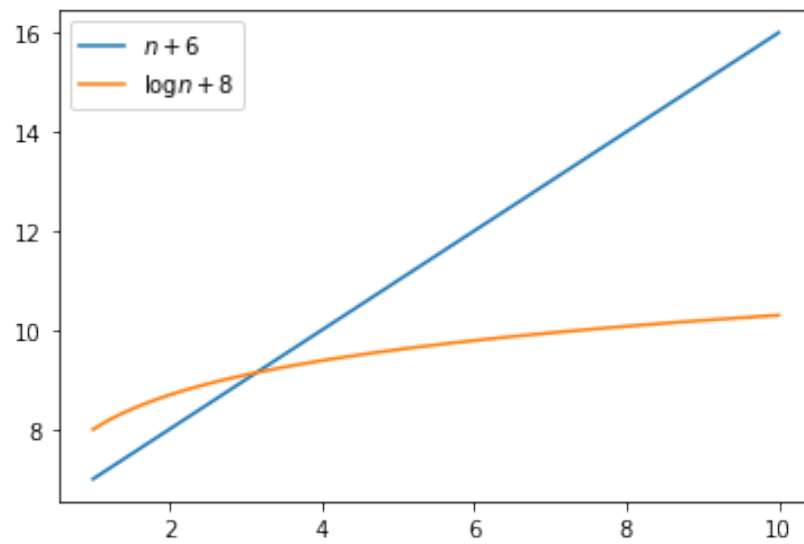


Figure 10: png

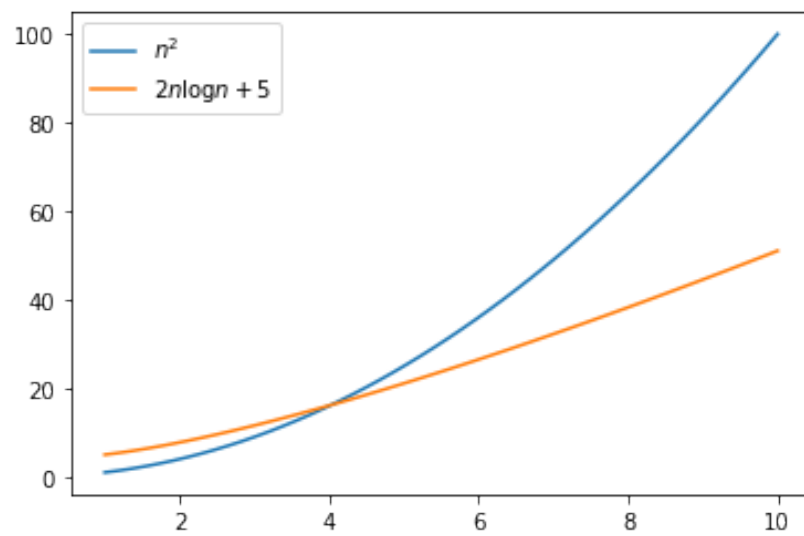


Figure 11: png

solución única, debemos especificar además las *condiciones iniciales*, también llamadas *condiciones de borde*.

Ejemplo: Número de movidas en las Torres de Hanoi

$$\begin{aligned}a_n &= 2a_{n-1} + 1 \text{ para } n \geq 1 \\ a_0 &= 0\end{aligned}$$

Ejemplo: Números de Fibonacci

$$\begin{aligned}f_n &= f_{n-1} + f_{n-2} \text{ para } n \geq 2 \\ f_0 &= 0 \\ f_1 &= 1\end{aligned}$$

Resolución de Ecuaciones Lineales de Primer Orden

Consideremos ecuaciones de la forma

$$a_n = ba_{n-1} + c_n$$

donde b es una constante distinta de cero y c_n es una función conocida.

Para ver cómo resolver este tipo de ecuaciones, a modo de “pre-calentamiento,” veamos cómo resolver esta ecuación para el caso $b = 1$:

$$a_n = a_{n-1} + c_n$$

Esto se puede reescribir como

$$a_n - a_{n-1} = c_n$$

y ahora introducimos sumatoria en ambos lados:

$$\sum_{1 \leq k \leq n} (a_k - a_{k-1}) = \sum_{1 \leq k \leq n} c_k$$

Pero la suma de la izquierda es telescópica, así que el resultado es:

$$a_n = a_0 + \sum_{1 \leq k \leq n} c_k$$

Abordemos ahora el caso general. La idea es tomar la ecuación $a_n = ba_{n-1} + c_n$ y reducirla de alguna manera al caso $a = 1$ que ya sabemos resolver.

Para esto, dividiremos ambos lados de la ecuación por el *factor sumante* b^n , obteniendo

$$\frac{a_n}{b^n} = \frac{a_{n-1}}{b^{n-1}} + \frac{c_n}{b^n}$$

Si ahora hacemos la sustitución $A_n = a_n/b^n$ y $C_n = c_n/b^n$, la ecuación queda en la forma

$$A_n = A_{n-1} + C_n$$

que ya sabemos cómo resolver:

$$A_n = A_0 + \sum_{1 \leq k \leq n} C_k$$

Haciendo ahora la sustitución inversa, obtenemos el resultado:

$$a_n = a_0 b^n + \sum_{1 \leq k \leq n} c_k b^{n-k}$$

Ejemplo: Número de Movidas en las Torres de Hanoi

La ecuación

$$a_n = 2a_{n-1} + 1 \text{ para } n \geq 1$$

$$a_0 = 0$$

tiene como solución

$$a_n = \sum_{1 \leq k \leq n} 2^{n-k} = \sum_{0 \leq k \leq n-1} 2^k$$

lo cual se simplifica a

$$a_n = 2^n - 1$$

Ejercicio 2.1

Resuelva la ecuación

$$a_n = b_n a_{n-1} + c_n$$

donde b_n y c_n son funciones conocidas y $b_n \neq 0$ for all $n \geq 0$.

Resolución de Ecuaciones Lineales Homogéneas con Coeficientes Constantes

Consideremos por ejemplo la ecuación de Fibonacci

$$f_n = f_{n-1} + f_{n-2} \text{ para } n \geq 2$$

$$f_0 = 0$$

$$f_1 = 1$$

La siguiente tabla muestra los valores que toma esta función:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
f_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

Como se ve, el crecimiento es bastante rápido. Esto es porque este tipo de ecuaciones suele tener soluciones de tipo exponencial, esto es, soluciones de la forma

$$f_n = \lambda^n$$

para alguna constante λ . Además, si encontramos más de una constante λ para la cual se satisfaga la ecuación, por ser una ecuación lineal, cualquier combinación lineal de esas soluciones también será solución.

Si logramos encontrar una solución que cumpla las condiciones iniciales bajo este supuesto, no necesitamos seguir buscando, porque la solución es única.

Sustituyendo λ^n en lugar de f_n , y dividiendo ambos lados por λ^{n-2} obtenemos la siguiente ecuación

$$\lambda^2 - \lambda - 1 = 0$$

llamada la *ecuación característica* de la ecuación de recurrencia.

Resolviendo esta ecuación de segundo grado, obtenemos las raíces

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618\dots, \quad \hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.618\dots$$

La solución general sería una combinación lineal de estas soluciones:

$$f_n = A\phi^n + B\hat{\phi}^n$$

La condición inicial $f_0 = 0$ implica que $B = -A$, y por lo tanto

$$f_n = A(\phi^n - \hat{\phi}^n)$$

La segunda condición inicial, $f_1 = 1$, implica que

$$A(\phi - \hat{\phi}) = A\sqrt{5} = 1$$

Con lo cual obtendremos la siguiente fórmula para los números de Fibonacci:

$$f_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$$

Nótese que como $|\hat{\phi}| < 1$, el aporte del segundo término tiende a cero rápidamente, y $f_n = \Theta(\phi^n)$.

```
import math
def f(n):
    r5=math.sqrt(5)
    phi=(1+r5)/2
    phihat=(1-r5)/2
    return 1/r5*(phi**n-phihat**n)
```

```
print(f(10), f(15), f(20))
```

```
55.000000000000014 610.0000000000003 6765.000000000005
```

El “ruido” que se observa en las cifras finales es producto de los errores de truncación al representar irracionales como $\sqrt{5}$ en punto flotante. Si expandiéramos las fórmulas y simplificáramos, obtendríamos los valores exactos.

Ejemplo: Número de movidas en las Torres de Hanoi

Veamos ahora una nueva manera de resolver la ecuación de Hanoi, la cual podemos describir como

$$a_n - 2a_{n-1} = 1 \text{ para } n \geq 1$$

$$a_0 = 0$$

Ésta es una ecuación lineal de coeficientes constantes *no homogénea*, de modo que el método que acabamos de ver no es directamente aplicable.

Si definimos el operador Δ como $\Delta a_n = a_{n+1} - a_n$, podemos transformar esta ecuación en una ecuación homogénea aplicando el operador Δ a ambos lados:

$$\Delta(a_n - 2a_{n-1}) = \Delta 1$$

$$(a_{n+1} - 2a_n) - (a_n - 2a_{n-1}) = 1 - 1$$

esto es

$$a_{n+1} - 3a_n + 2a_{n-1} = 0$$

De manera similar, cualquier ecuación de este tipo en que la parte no homogénea es un polinomio en n puede convertirse en una ecuación homogénea por aplicaciones reiteradas del operador Δ .

Si “pasamos en limpio” esta ecuación de modo que su término líder sea a_n , se ve que la nueva ecuación sólo es válida para $n \geq 2$, de modo que se necesita una segunda condición inicial, la cual se obtiene sin problemas desde la ecuación original.

Ejercicio 2.2

Resuelva la ecuación homogénea de las Torres de Hanoi:

$$a_n - 3a_{n-1} + 2a_{n-2} = 0 \text{ para } n \geq 2$$

$$a_0 = 0$$

$$a_1 = 1$$

El Teorema Maestro

Veremos diversos algoritmos que dan origen a recurrencias de la forma

$$T(n) = pT\left(\frac{n}{q}\right) + Cn^r$$

Para simplificar, supongamos que n es una potencia de q , digamos $n = q^k$, para que la división n/q siempre se pueda hacer en forma exacta. Entonces

$$T(q^k) = pT(q^{k-1}) + C(q^r)^k$$

Si introducimos una nueva función incógnita $a_k = T(q^k)$, podemos reescribir la ecuación como

$$a_k = pa_{k-1} + C(q^r)^k$$

la cual es de un tipo que ya sabemos resolver. Su solución es

$$\begin{aligned} a_k &= a_0 p^k + C \sum_{1 \leq j \leq k} (q^r)^j p^{k-j} \\ &= p^k \left(a_0 + C \sum_{1 \leq j \leq k} \left(\frac{q^r}{p} \right)^j \right) \end{aligned}$$

El comportamiento de esta sumatoria depende de si el cociente q^r/p es menor, igual o mayor que 1.

Antes de analizar cada caso, observemos que $n = q^k$ implica que $k = \log_q n$, y

$$p^k = p^{\log_q n} = (q^{\log_q p})^{\log_q n} = (q^{\log_q n})^{\log_q p} = n^{\log_q p}$$

con lo cual la solución puede escribirse como

$$T(n) = n^{\log_q p} \left(T(1) + C \sum_{1 \leq j \leq \log_q n} \left(\frac{q^r}{p} \right)^j \right)$$

Caso $q^r < p$:

En este caso, la sumatoria está acotada por una constante, porque la serie respectiva es convergente, de modo que

$$T(n) = \Theta(n^{\log_q p})$$

Caso $q^r = p$:

En este caso la sumatoria es logarítmica, y $\log_q p = r$, de modo que

$$T(n) = \Theta(n^r \log n)$$

Caso $q^r > p$:

En este caso la sumatoria es la suma de una progresión geométrica, que podemos escribir en forma cerrada como

$$\begin{aligned} T(n) &= n^{\log_q p} \left(T(1) + C \frac{q^r \left(\frac{q^r}{p} \right)^{\log_q n} - 1}{\frac{q^r}{p} - 1} \right) \\ &= \Theta(n^{\log_q p} + n^r) \\ &= \Theta(n^r) \text{ porque } \log_q p < r \end{aligned}$$

En conclusión, hemos demostrado lo siguiente:

Teorema Maestro

La ecuación

$$T(n) = pT\left(\frac{n}{q}\right) + Cn^r$$

tiene solución

$$T(n) = \begin{cases} \Theta(n^r) & \text{si } p < q^r \\ \Theta(n^r \log n) & \text{si } p = q^r \\ \Theta(n^{\log_q p}) & \text{si } p > q^r \end{cases}$$

Ejercicio 2.3

El método de ordenación **Stooge Sort** es un método recursivo que puede describirse de la siguiente manera:

- Si el primer elemento es mayor que el último, los intercambiamos
- Si hay 3 o más elementos en la lista, entonces:
 - Ordenar los primeros $2/3$ de la lista recursivamente
 - Ordenar los últimos $2/3$ de la lista, recursivamente, y
 - Ordenar (¡de nuevo!) los primeros $2/3$ de la lista.

Escriba una ecuación que modele el tiempo de ejecución de Stooge Sort y resuélvala usando el Teorema Maestro.

3

3 Diseño de Algoritmos Eficientes

En este capítulo veremos un conjunto de ideas que permiten diseñar algoritmos que, en muchos casos, son de los más eficientes que se conocen para sus respectivos problemas.

Dividir para Reiniciar

Este es un método de diseño de algoritmos que se basa en subdividir el problema en sub-problemas, resolverlos recursivamente, y luego combinar las soluciones de los sub-problemas para construir la solución del problema original. Es necesario que los subproblemas tengan la misma estructura que el problema original, de modo que se pueda aplicar la recursividad.

Ejemplo: Multiplicación de Polinomios

Supongamos que tenemos dos polinomios $A(x)$ y $B(x)$, cada uno de grado $n - 1$:

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$$

$$B(x) = b_0 + b_1x + b_2x^2 + \dots + b_{n-1}x^{n-1}$$

representados por sus respectivos arreglos de coeficientes $a[0], \dots, a[n - 1]$ y $b[0], \dots, b[n - 1]$.

El problema consiste en calcular los coeficientes $c[0], \dots, c[2n - 2]$ del polinomio producto $C(x) = A(x)B(x)$.

Por ejemplo,

$$A(x) = 2 + 3x - 6x^2 + x^3$$

$$B(x) = 1 - x + 3x^2 + x^3$$

$$C(x) = A(x)B(x) = 2 + x - 3x^2 + 18x^3 - 16x^4 - 3x^5 + x^6$$

La manera obvia de resolver este problema es multiplicando

cada término de $A(x)$ por cada término de $B(x)$ y acumulando los resultados que corresponden a la misma potencia de x :

```
import numpy as np
def multpol(a, b):
    n=len(a)
    assert len(b)==n
    c=np.zeros(2*n-1)
    for i in range(0,n):
        for j in range(0,n):
            c[i+j]+=a[i]*b[j]
    return c
```

```
multpol(np.array([2,3,-6,1]), np.array([1,-1,3,1]))
```

```
array([ 2.,  1., -3., 18., -16., -3.,  1.])
```

Evidentemente, este algoritmo demora tiempo $O(n^2)$. ¿Es posible hacerlo más rápido? Para esto, aplicaremos la técnica de *dividir para reinar*.

Supongamos que n es par, y dividamos los polinomios en dos partes, separando las potencias bajas de las altas. Por ejemplo, si

$$A(x) = 2 + 3x - 6x^2 + x^3$$

lo podemos reescribir como

$$A(x) = (2 + 3x) + (-6 + 3x)x^2$$

En general, podemos reescribir $A(x)$ y $B(x)$ como

$$A(x) = A'(x) + A''(x)x^{n/2}$$

$$B(x) = B'(x) + B''(x)x^{n/2}$$

y entonces (omitiendo los “ (x) ” para simplificar la notación),

$$C = A'B' + (A'B'' + A''B')x^{n/2} + A''B''x^n$$

Esto se puede implementar con 4 multiplicaciones recursivas, cada una involucrando polinomios de la mitad del tamaño. Nótese que las multiplicaciones por potencias de x son solo realineaciones de los arreglos de coeficientes, de modo que son “gratis.”

Si llamamos $T(n)$ al número total de operaciones, éste obedece la ecuación de recurrencia

$$T(n) = 4T\left(\frac{n}{2}\right) + Kn$$

para alguna constante K .

Por el Teorema Maestro, con $p = 4$, $q = 2$, $r = 1$, tenemos

$$T(n) = \Theta(n^2)$$

lo cual no es mejor que el algoritmo anterior.

Afortunadamente, hay una manera de obtener un algoritmo realmente más eficiente. Si calculamos

$$D = (A' + A'')(B' + B'')$$

$$E = A'B'$$

$$F = A''B''$$

podemos construir el polinomio C de la manera siguiente:

$$C = E + (D - E - F)x^{n/2} + Fx^n$$

¡lo cual utiliza solo 3 multiplicaciones recursivas!

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

Usando nuevamente el Teorema Maestro, esta vez con $p = 3$, tenemos que

$$T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$$

Éste se llama el *Algoritmo de Karatsuba*.

Ejercicio 3.1

Si tenemos dos números complejos

$$u = a + bi$$

$$v = c + di$$

podemos calcular su producto

$$uv = (ac - bd) + (ad + bc)i$$

haciendo 4 multiplicación de números reales.

Encuentre una forma de realizar este cálculo haciendo solo 3 multiplicaciones de números reales.

Programación Dinámica

Hay ocasiones en que la simple aplicación de la recursividad conduce a algoritmos muy ineficientes, pero es posible evitar esa ineficiencia con un uso adecuado de memoria.

Ejemplo: Cálculo de un número de Fibonacci

Recordemos la ecuación de Fibonacci

$$f_n = f_{n-1} + f_{n-2} \text{ para } n \geq 2$$

$$f_0 = 0$$

$$f_1 = 1$$

algunos de cuyos valores son:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
f_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

Queremos resolver el siguiente problema: dado un n , calcular f_n .

A partir de la ecuación de recurrencia podemos escribir de inmediato una solución recursiva:

```
def fibonacci(n):
    if n<=1:
        return n
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

```
print(fibonacci(10))
```

55

El problema es que, para n grande, este algoritmo es horriblemente ineficiente. El motivo de esto es que, a medida que se van ejecutando las llamadas recursivas, un mismo número de Fibonacci puede calcularse múltiples veces, independientemente de si ya se ha calculado antes.

Una forma de dimensionar esta ineficiencia es calcular, por ejemplo, el número de operaciones de suma que se hacen al calcular $\text{fibonacci}(n)$. Llamemos s_n a ese número de sumas. Es fácil ver que

$$s_n = s_{n-1} + s_{n-2} + 1 \text{ para } n \geq 2$$

$$s_0 = 0$$

$$s_1 = 0$$

Si definimos una nueva función incógnita $t_n = s_n + 1$, tenemos que

$$t_n = t_{n-1} + t_{n-2} \text{ para } n \geq 2$$

$$t_0 = 1$$

$$t_1 = 1$$

Esta es la misma ecuación de Fibonacci, comenzando un paso más adelante, y por lo tanto su solución es $t_n = f_{n+1}$, y el número de sumas es $s_n = f_{n+1} - 1 = \Theta(\phi^n)$. Conclusión: ¡el tiempo que demora la ejecución de `fibonacci(n)` crece exponencialmente!

Evitando la ineficiencia: Memoización

La ineficiencia de la solución recursiva se debe, como dijimos antes, a que un mismo valor de la función f se calcula y recalcula múltiples veces. Una forma de evitar esto es incorporar una memoria auxiliar, en forma de un arreglo F inicializado con ceros. La primera vez que se pide calcular un f_n dado, lo hacemos recursivamente, pero dejamos el valor anotado en $F[n]$. Las siguientes veces lo tomamos del arreglo, sin incurrir de nuevo en el costo del cálculo recursivo. Esta forma de utilizar una memoria para almacenar resultados calculados previamente, para evitar recalcularlos, se llama una *memoria caché*.

```
import numpy as np
def fibonacci(n):
    F=np.zeros(n+1,dtype=int)
    def fib_rec(k):
        if k>0 and F[k]==0: # primera vez que se calcula
            if k<=1:
                F[k]=k
            else:
                F[k]=fib_rec(k-1)+fib_rec(k-2)
        return F[k]
    return fib_rec(n)

print(fibonacci(10))
```

55

La introducción de esta memoria auxiliar tiene como efecto transformar el algoritmo original, de tiempo exponencial, en un algoritmo de tiempo lineal $\Theta(n)$.

Evitando la ineficiencia: Tabulación

La técnica de *memoización* va llenando el arreglo auxiliar F a medida que sus valores son solicitados. Este método es bastante general, pero se puede mejorar si logramos encontrar un orden para ir llenando el arreglo F que garantice que cuando se requiere el valor de un cierto casillero, éste ya está llenado.

En el caso de Fibonacci, esto se logra simplemente al ir llenando los casilleros $F[k]$ en orden creciente de k . Esta técnica se llama *tabulación*:

```
import numpy as np
def fibonacci(n):
    F=np.zeros(n+1,dtype=int)
    F[0]=0
    F[1]=1
    for k in range(2,n+1):
        F[k]=F[k-1]+F[k-2]
    return F[n]
```

```
print(fibonacci(10))
```

55

Es evidente que el tiempo que demora este algoritmo es $\Theta(n)$.

Derrotando al algoritmo lineal

Si bien parecería que para calcular f_n es necesario calcular previamente todos los f_k , para $0 \leq k < n$, en realidad esto no es cierto.

Introduzcamos una función incógnita adicional g_n en la ecuación de Fibonacci, definiéndola como $g_n = f_{n-1}$. La ecuación puede reescribirse así:

$$f_n = f_{n-1} + g_{n-1}$$

$$g_n = f_{n-1}$$

$$f_1 = 1$$

$$g_1 = 0$$

Esto se puede reescribir en forma matricial:

$$\begin{pmatrix} f_n \\ g_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ g_{n-1} \end{pmatrix} \text{ para } n \geq 2$$

con la condición inicial

$$\begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Esta ecuación es muy simple de resolver “desenrollándola,” y su solución es

$$\begin{pmatrix} f_n \\ g_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Por lo tanto, para resolver el problema del calcular f_n , basta evaluar esta fórmula matricial y luego tomar la primera componente del vector resultante.

Recordando que para la elevación a potencia podemos usar el algoritmo binario, la evaluación de la fórmula se puede hacer en tiempo logarítmico, y por lo tanto el problema de calcular f_n se puede resolver en tiempo $\Theta(\log n)$.

La siguiente es una versión del algoritmo potencia adaptada para calcular $B = A^n$ cuando A es una matriz cuadrada:

```
import numpy as np
def potencia(A, n):
    B=np.eye(len(A),dtype=int) # matriz identidad
    k=n
    C=A
    while k>0:
        while k%2==0:
            C=np.dot(C,C) # C=C**2
            k//=2
        B=np.dot(B,C) # B=C*C
        k-=1
    return B

def fibonacci(n):
    F=np.dot(potencia(np.array([[1,1],[1,0]]),n-1), np.array([[1],[0]]))
    return F[0,0]

print(fibonacci(10))
```

55

Ejercicio 3.2

Si f_n son los números de Fibonacci, demuestre que, para todo $n \geq 1$,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

Ejemplo: Encontrar la parentización óptima para multiplicación de n matrices

Hemos visto que las técnicas de *memoización* y de *tabulación* nos permiten construir algoritmos eficientes en algunos problemas en

que la recursividad aplicada directamente daría soluciones muy ineficientes.

Cuando estas técnicas se aplican a problemas de optimización, hablamos de *programación dinámica*.

Consideremos el siguiente problema: Dadas tres matrices A , B y C para las cuales se desea calcular su producto ABC , ¿qué es más eficiente, calcular $(AB)C$ o calcular $A(BC)$?

La respuesta depende de las dimensiones de las matrices involucradas. Si una matriz A es de $p \times q$ y otra matriz B es de $q \times r$, calcular su producto AB utilizando el algoritmo usual requiere hacer pqr multiplicaciones escalares, y un número similar de sumas.

Para nuestro problema de calcular ABC , supongamos por ejemplo que A es de 100×10 , B de 10×100 y C de 100×10 , tenemos que

- Calcular $(AB)C$ requiere $100 \times 10 \times 100 + 100 \times 100 \times 10 = 200.000$ multiplicaciones
- Calcular $A(BC)$ requiere $10 \times 100 \times 10 + 100 \times 10 \times 10 = 20.000$ multiplicaciones

La respuesta, por lo tanto, es que para las dimensiones dadas, la parentización óptima es $A(BC)$.

Consideremos ahora el problema general. Dadas n matrices A_1, A_2, \dots, A_n y números $p[0], p[1], \dots, p[n]$ tales que la matriz A_i es de $p[i-1] \times p[i]$, encontrar el costo (en número de multiplicaciones) de la parentización óptima para calcular el producto

$$A_1 A_2 \cdots A_n$$

Generalicemos el problema para poder abordarlo recursivamente (o inductivamente). Supongamos que el problema es encontrar el costo de la parentización óptima para calcular el producto

$$A_i \cdots A_j$$

para $1 \leq i \leq j \leq n$. Llamemos $m[i, j]$ a este costo óptimo.

En el caso $i = j$ el producto involucra a una sola matriz, así que, trivialmente, $m[i, i] = 0$. En el caso $i < j$, supongamos que parentizamos de modo que el producto se factorice como

$$(A_i \cdots A_k)(A_{k+1} \cdots A_j)$$

para algún $k \in [i..j-1]$. Suponiendo que cada producto parentizado se ha calculado en forma óptima, el costo total sería

$$m[i, k] + m[k+1, j] + p[i-1] \times p[k] \times p[j]$$

Esto no es necesariamente óptimo para el producto $A_i \cdots A_j$, porque podríamos haber elegido el valor equivocado de k . Para

asegurarnos de alcanzar el óptimo, tenemos que minimizar sobre todo k :

$$m[i, j] = \min_{i \leq k \leq j-1} \{m[i, k] + m[k+1, j] + p[i-1] \times p[k] \times p[j]\}$$

Esto lo podríamos implementar mediante una función recursiva, pero, tal como sucedía en el ejemplo de Fibonacci, ésta demoraría un tiempo exponencial en ejecutarse, porque generaría y evaluaría todas las parentizaciones posibles. Esta búsqueda exhaustiva es un método que encuentra la respuesta correcta (se le suele llamar “método fuerza bruta”), pero usualmente es demasiado ineficiente.

Afortunadamente, en este caso podemos usar tabulación, porque hay solo alrededor de $n^2/2$ casilleros que llenar en la matriz m , y los podemos ir llenando en un orden tal que al calcular el mínimo sobre todo k , los casilleros necesarios ya han sido llenados previamente.

En efecto, introduzcamos una nueva variable $d = j - i + 1$. Esto es el número de matrices involucradas en el producto $A_i \cdots A_j$. Lo que haremos será ir llenando la matriz en orden ascendente de la variable d , comenzando con el caso trivial $d = 1$, hasta terminar con el caso $d = n$, que corresponde a la solución del problema original.

```
import numpy as np
import math
def opti_multi_mat(p):
    n=len(p)-1
    m=np.zeros((n+1,n+1)) # Esto ya deja la diagonal en cero
    for d in range(2,n+1):
        for i in range(1,n-d+2):
            j=i+d-1
            m[i,j]=math.inf # +infinito
            for k in range(i,j):
                q=m[i,k]+m[k+1,j]+p[i-1]*p[k]*p[j]
                if q<m[i,j]:
                    m[i,j]=q
    return m[1,n]
```

```
p=np.array([100,10,100,10])
print(opti_multi_mat(p))
```

20000.0

Es fácil ver que este algoritmo corre en tiempo $O(n^3)$, porque debe rellenar $\Theta(n^2)$ casilleros, y cada uno puede requerir examinar n valores posibles de k en el peor caso. Esto es significativamente mejor que el algoritmo de fuerza bruta.

Esta cota cúbica podría, sin embargo, ser un poco exagerada, porque en muchos casos la variable k toma mucho menos que n valores. Un cálculo más preciso nos señala que el número total de veces que se ejecuta el cuerpo del ciclo `for k` es igual a

$$\sum_{1 \leq i \leq k < j \leq n} 1 = \frac{n(n-1)(n+1)}{6} = \Theta(n^3)$$

Por lo tanto el algoritmo en realidad demora un tiempo cúbico.

Como vemos, el resultado del proceso es el costo óptimo, pero eso no nos da ninguna información sobre cuál es la parentización óptima. Pero en realidad la información está ahí, porque el valor de k para el cual se alcanza el mínimo nos dice dónde separar la parentización en cada caso. Basta entonces con que dejemos anotado, para cada i, j cuál es el valor de k para el que se alcanza el mínimo. Llamemos $s[i, j]$ a ese valor de k .

Modifiquemos nuestra función para que construya y retorne la matriz s además del costo óptimo, y escribamos otra función que, dada esa matriz s , imprima la fórmula parentizada de la manera óptima.

```
import numpy as np
import math
def opti_multi_mat(p):
    n=len(p)-1
    m=np.zeros((n+1,n+1)) # Esto ya deja la diagonal en cero
    s=np.zeros((n+1,n+1),dtype=int)
    for d in range(2,n+1):
        for i in range(1,n-d+2):
            j=i+d-1
            m[i,j]=math.inf # +infinito
            for k in range(i,j):
                q=m[i,k]+m[k+1,j]+p[i-1]*p[k]*p[j]
                if q<m[i,j]:
                    m[i,j]=q
                    s[i,j]=k # anotamos dónde se alcanza el min
    return (m[1,n],s)
```

```
def paren_desde_hasta(s,i,j):
    if i==j:
        return "A"+str(i)
    else:
        return "("+paren_desde_hasta(s,i,s[i,j])+" "\
            +paren_desde_hasta(s,s[i,j]+1,j)+")"
```

```
def parentizacion(s):
    n=len(s)-1
    return paren_desde_hasta(s,1,n)
```

```
p=np.array([100,10,100,10])
(opt,s)=opti_multi_mat(p)
print(parentizacion(s))
print("Costo=", opt)
```

```
(A1 (A2 A3))
Costo= 20000.0
```

```
p=np.array([30,35,15,5,10,20,25])
(opt,s)=opti_multi_mat(p)
print(parentizacion(s))
print("Costo=", opt)
```

```
((A1 (A2 A3)) ((A4 A5) A6))
Costo= 15125.0
```

Para recapitular, la técnica de diseño de programación dinámica divide un problema en varios subproblemas con la misma estructura que el problema original, luego se resuelven dichos subproblemas y finalmente, a partir de éstos, se obtiene la solución al problema original. La diferencia radica en que la programación dinámica se ocupa cuando los subproblemas se repiten, como en el cálculo de los números de Fibonacci. En este caso, en vez de usar recursión para obtener las soluciones a los subproblemas éstas se van tabulando en forma bottom-up, y luego estos resultados son utilizados para resolver subproblemas más grandes. De esta forma, se evita el tener que realizar el mismo llamado recursivo varias veces.

La programación dinámica se ocupa en general para resolver problemas de optimización (maximización o minimización de alguna función objetivo). Estos problemas pueden tener una o varias soluciones óptimas, y el objetivo es encontrar alguna de ellas. Los pasos generales para utilizar programación dinámica en la resolución de un problema son los siguientes:

- Encontrar la subestructura óptima del problema, es decir, encontrar aquellos subproblemas en los que se compone el problema original, tal que si uno encuentra sus soluciones óptimas entonces es posible obtener la solución óptima al problema original.
- Definir el valor de la solución óptima en forma recursiva.

- Calcular el valor de la solución partiendo primero por los subproblemas más pequeños y tabulando las soluciones, lo que luego permite obtener la solución de subproblemas más grandes. Terminar cuando se tiene la solución al problema original.

Estos pasos permiten obtener el valor óptimo de la solución al problema. También es posible ir guardando información extra en cada paso del algoritmo, que luego permita reconstruir el camino realizado para hallar la solución óptima (por ejemplo, para obtener la instancia específica de la solución óptima, y no sólo el valor óptimo de la función objetivo).

Ejemplo: Encontrar la subsecuencia común más larga

Dadas dos secuencias de datos a y b , de largo m y n respectivamente (pueden ser listas o strings), una subsecuencia común es una subsecuencia de elementos (posiblemente saltados) de a que coincide con una subsecuencia de b . Nos interesa encontrar el largo de la subsecuencia común más larga (en inglés *LCS*, por *longest common subsequence*).

Este problema se puede resolver recursivamente. La condición de borde es que la subsecuencia común más larga entre dos secuencias en que al menos una de ellas es vacía (de largo cero) es la secuencia vacía. Si ambas secuencias son no vacías, observamos que una subsecuencia común más larga entre $a[0 : i]$ y $b[0 : j]$ se puede obtener de la siguiente manera:

- Si ambas secuencias terminan en el mismo elemento, agregamos ese elemento al final de la subsecuencia común más larga entre $a[0 : i - 1]$ y $b[0 : j - 1]$
- Si no terminan en el mismo elemento, tomamos lo más largo que resulte entre ignorar el último elemento de $a[0 : i]$ y buscar una subsecuencia común más larga entre $a[0 : i - 1]$ y $b[0 : j]$, o ignorar el último elemento de $b[0 : j]$ y buscar una subsecuencia común más larga entre $a[0 : i]$ y $b[0 : j - 1]$

Si llamamos $L_{i,j}$ al largo de la subsecuencia común más larga entre $a[0 : i]$ y $b[0 : j]$, tenemos que $L_{0,j} = L_{i,0} = 0$, y que para $i, j > 0$

$$L_{i,j} = \begin{cases} 1 + L_{i-1,j-1} & \text{si } a[i-1] = b[j-1] \\ \max\{L_{i,j-1}, L_{i-1,j}\} & \text{si } a[i-1] \neq b[j-1] \end{cases}$$

El resultado buscado es $L_{m,n}$.

Una implementación eficiente de este algoritmo se puede lograr usando tabulación:

```
import numpy as np
def LCS(a,b):
    """
    Encuentra el largo de la subsecuencia común más larga entre a y b
    """
    m=len(a)
    n=len(b)
    L=np.zeros((m+1,n+1),dtype=int)
    for i in range(1,m+1):
        for j in range(1,n+1):
            if a[i-1]==b[j-1]:
                L[i,j]=1+L[i-1,j-1]
            else:
                L[i,j]=max(L[i-1,j],L[i,j-1])
    return L[m,n]

print(LCS("abracadabra","pasapalabra"))
```

7

Ejercicio 3.3

Modifique la función LCS para que retorne una subsecuencia común más larga, en lugar de retornar su longitud.

Algoritmos Avaros (Greedy Algorithms)

Se dice que un algoritmo de optimización es *avaro* si siempre toma la decisión óptima de corto plazo. Por ejemplo, un algoritmo avaro que intenta llegar a la cima del cerro más alto, daría siempre un paso en la dirección que le permite subir más con ese paso.

En general, la estrategia avara no garantiza llegar a un óptimo global, porque es fácil quedarse atrapado en un óptimo local (en nuestro ejemplo, llegar a la cima de un cerro pequeño y no poder salir de ahí).

Sin embargo, hay problemas para los cuales la estrategia avara sí encuentra un óptimo global:

Ejemplo: Asignación de actividades

Supongamos que hay un conjunto n de actividades a_1, \dots, a_n que para poder realizarse necesitan acceso a un recurso que no puede ser compartido. Por ejemplo, n reuniones que solo se pueden realizar en una única sala.

Cada actividad tiene un tiempo de inicio t_i y un tiempo de término t_f , lo que se interpreta como que necesita ocupar el recurso en el intervalo $[t_i, t_f)$, cerrado a la izquierda, abierto a la derecha. Se dice que dos actividades son compatibles si sus respectivos intervalos no se traslapan. El problema de asignación de actividades consiste en encontrar un conjunto maximal de actividades compatibles.

Por ejemplo, para el siguiente conjunto de actividades,

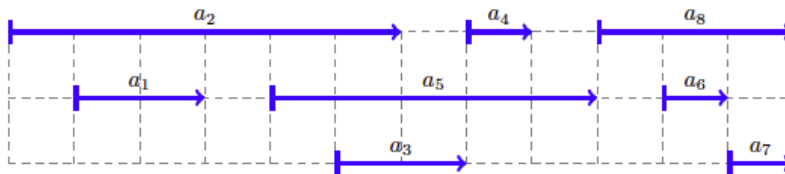


Figure 12: actividades

un conjunto compatible maximal es $\{a_1, a_3, a_4, a_6, a_7\}$.

Un problema de optimización de este tipo siempre se puede resolver por fuerza bruta, porque el número de soluciones posibles es finito, pero tratamos de evitarlo a menos que no haya alternativa, porque puede demorar un tiempo exponencial.

En este caso, podemos resolver el problema de manera mucho más eficiente (tiempo $\Theta(n)$) usando una estrategia avara.

Supongamos que las actividades están ordenadas en forma ascendente por su hora de término t_f . El algoritmo escoge la primera actividad disponible en este orden, luego elimina todas las actividades que se superponen con ella, y a continuación repite el proceso con las actividades restantes:

```
# Input es una matriz con filas de la forma [ti,tf]
# ordenada ascendente por tf
import numpy as np
A=np.array([ [0,0], # la fila 0 se ignora
             [1,3],
             [0,6],
             [5,7],
             [7,8],
             [4,9],
             [10,11],
             [11,12],
             [9,12]
            ])
1)
```



```
def asigna_avaro(A):
    n=len(A)-1
    sol=[1] # la primera tarea es siempre parte de la solución
    j=1     # j identifica a la más reciente tarea seleccionada
    for i in range(2,n+1):
        if A[i,0]>=A[j,1]: #  $t_i[i] \geq t_f[j] \implies i$  es la primera tarea
            sol.append(i) # que es compatible con j, se agrega
            j=i           # a la solución y pasa a ser la más reciente
    return sol
```

```
asigna_avaro(A)
```

```
[1, 3, 4, 6, 7]
```

El algoritmo resultante se llama “*Earliest-Finish-First*” (EFF).

¿Cómo podemos estar seguros que la solución encontrada es realmente óptima?

Demostraremos por inducción sobre n que el algoritmo avaro encuentra una solución óptima.

Trivialmente el algoritmo encuentra la solución correcta en el caso $n = 1$.

Consideremos entonces el caso $n > 1$ y supongamos, por contradicción, que no existe ninguna solución óptima que incluya a la actividad a_1 . Consideremos una solución óptima y supongamos que su actividad con menor t_f es la actividad a_k . Dado que la lista está ordenada por t_f , se tiene que $t_f[1] \leq t_f[k]$. Como la solución no puede contener ninguna otra actividad que comience antes de $t_f[k]$, entonces podemos reemplazar la actividad a_k por la actividad a_1 y obtenemos otra solución válida y que tiene el mismo número de actividades, por lo tanto también es óptima: contradicción con la hipótesis que a_1 no podía ser parte de ninguna solución óptima.

Por lo tanto, a_1 puede ser parte de una solución óptima, y en consecuencia ninguna actividad que se superponga con ella puede estar incluida. Si las eliminamos, lo que queda es un problema de optimización del mismo tipo, pero con un número estrictamente menor de actividades, el cual lo resuelve correctamente el algoritmo avaro por hipótesis de inducción.

Backtracking

Hay ocasiones en que no tenemos más alternativa que resolver un problema por prueba y error, explorando un espacio de soluciones en forma exhaustiva. Una forma de hacerlo es en forma recursiva, probando distintas vías de solución hasta que alguna funcione, o

hasta agotar las posibilidades.

El laberinto

Supongamos que nos encontramos al interior de un laberinto y queremos encontrar la salida. Si no tenemos ninguna información adicional, lo que podemos hacer es intentar salir en una dirección, si eso no funciona intentar en otras, etc.

Para fijar las ideas, supongamos que el laberinto se dibuja usando caracteres del teclado, donde los espacios en blanco son lugares por donde se puede pasar, y el signo “igual” representa la salida. Todos los demás símbolos representan murallas. Por ejemplo,

```
+--+-----+--+
|  |      |  |
|  +--+    =
|      |  |  |
+--+  |  |  |
|  |      |
|  |      |
+--+-----+--+
```

Almacenaremos el laberinto en una lista de strings (de a uno por línea):

```
L = [
"+--+-----+--+",
"|  |      |  |",
"|  +--+    =",
"|      |  |  |",
"+--+  |  |  |",
"|  |      |",
"|  |      |",
"+--+-----+--+"
]
```

El problema es determinar si existe una manera de salir si uno comienza en el casillero $[i][j]$. Solo está permitido moverse en dirección horizontal o vertical, no en diagonal.

Resolveremos este problema escribiendo una función booleana *salida*(*i*,*j*) que retorna verdadero si existe un camino hacia la salida a partir de la coordenada *i*,*j*. Esto lo hace intentando en las cuatro direcciones:

```
def salida(i,j):
    if L[i][j]=="=": # encontramos la salida
        return True
```

```

if L[i][j]!=" ": # espacio ocupado
    return False
if salida(i,j-1) \
or salida(i,j+1) \
or salida(i-1,j) \
or salida(i+1,j):
    return True
return False

```

Probemos si es posible salir desde la posición 5,2 (¡pero por si acaso estemos atentos al botón para interrumpir el proceso!)

```

L = [
"+---+-----+---+",
"|   |       |   |",
"| +--+      =",
"|   |   |   |",
"+--+ |   |   |",
"|   |       |",
"|   |   |   |",
"+---+-----+---+"
]
print(salida(5,2))

```

RecursionError Traceback (most recent call last)

```

<ipython-input-19-0ea1bd4da578> in <module>
      9 "+---+-----+---+"
     10 ]
--> 11 print(salida(5,2))

<ipython-input-18-ba8a4ad77a65> in salida(i, j)
      4     if L[i][j]!=" ": # espacio ocupado
      5         return False
--> 6     if salida(i,j-1) \
      7         or salida(i,j+1) \
      8         or salida(i-1,j) \

<ipython-input-18-ba8a4ad77a65> in salida(i, j)
      5         return False

```

```

        6     if salida(i,j-1) \
----> 7     or salida(i,j+1) \
        8     or salida(i-1,j) \
        9     or salida(i+1,j):

```

... last 2 frames repeated, from the frame below ...

```

<ipython-input-18-ba8a4ad77a65> in salida(i, j)
      4     if L[i][j]!=" ": # espacio ocupado
      5         return False
----> 6     if salida(i,j-1) \
      7     or salida(i,j+1) \
      8     or salida(i-1,j) \

```

RecursionError: maximum recursion depth exceeded in comparison

Lo más probable es que el programa se haya caído por exceso de recursividad antes de que tuviéramos que interrumpirlo a mano.

Lo que sucede es que olvidamos incluir algo que impidiera que el programa explorara de nuevo un camino que ya antes había recorrido sin éxito, y eso lo condujo a entrar en un “loop” infinito.

Para evitar esto, iremos marcando con un símbolo “x” los casilleros a medida que van siendo visitados. De esta forma, ninguno se puede visitar más de una vez:

```

def salida(i,j):
    if L[i][j]=="x": # encontramos la salida
        return True
    if L[i][j]!=" ": # espacio ocupado
        return False
    L[i]=L[i][:j]+"x"+L[i][j+1:]
    if salida(i,j-1) \
    or salida(i,j+1) \
    or salida(i-1,j) \
    or salida(i+1,j):
        return True
    return False

```

```

L = [
    "+---+---+---+",
    "|   |   |   |",
    "| +--+   =",

```

```

" |   |   |   |",
"+--+ |   |   |",
" |   |   |   |",
" |   |   |   |",
"+--+-----+--+"
]
print(salida(5,2))
for linea in L:
    print(linea)

```

False

```

+--+-----+--+
| |   |   |
| +--+   =
|   |   |
+--+ |   |
|xx|   |
|xx|   |
+--+-----+--+

```

```

L = [
"+--+-----+--+",
" |   |   |   |",
" | +--+   =",
" |   |   |   |",
"+--+ |   |   |",
" |   |   |   |",
" |   |   |   |",
"+--+-----+--+"
]
print(salida(4,10))
for linea in L:
    print(linea)

```

True

```

+--+-----+--+
|xx|xxxxx|   |
|xx+--+xxxxx=
|xxxxx|xx|xx|
+--+xx|xx|xx|
| |xxxxxxxx|
| |xxxxx|xx|
+--+-----+--+

```

Ejercicio 3.4

Modifique la función para que marque con "." los casilleros por donde hubo intentos no exitosos de salir, y con "x" los casilleros que finalmente condujeron a la salida.

4

```
!pip install aed-utilities
```

4 Estructuras de datos elementales

Los sistemas o métodos de organización de datos que permiten un almacenamiento eficiente de la información en la memoria del computador son conocidos como estructuras de datos. Estos métodos de organización constituyen las piezas básicas para la construcción de algoritmos complejos, y permiten implementarlos de manera eficiente.

En el presente capítulo se presentan las estructuras de datos básicas como son arreglos, listas enlazadas y árboles, con las cuales se implementarán posteriormente los *tipos de datos abstractos*.

Arreglos

Un arreglo es una secuencia contigua en memoria, que almacena un número fijo de elementos homogéneos. En la siguiente figura se muestra un arreglo de enteros con 10 elementos:

80	45	2	21	92	17	5	65	14	34
0	1	2	3	4	5	6	7	8	9

Figure 13: ejemplo-arreglo

Una ventaja que tienen los arreglos es que el costo de acceso a un elemento dado del arreglo es constante, es decir no hay diferencias de costo entre acceder el primer, el último o cualquier elemento del arreglo, lo cual es muy eficiente. La desventaja es que es necesario definir a priori el tamaño del arreglo, lo cual puede generar mucha

pérdida de espacio en memoria si se definen arreglos muy grandes para contener conjuntos pequeños de elementos.

Esta característica de costo de acceso constante es esencial para la eficiencia de algunos algoritmos muy importantes, como por ejemplo el siguiente:

Ejemplo: Búsqueda Binaria

Supongamos que queremos buscar un elemento x en un arreglo a de tamaño n . Si no tenemos más información sobre el orden de los elementos dentro del arreglo, lo único que podemos hacer es una *búsqueda secuencial*, la cual tiene costo $\Theta(n)$ tanto en el peor caso como en el caso promedio.

Pero si sabemos que los elementos están en orden ascendente, existe una forma mucho más eficiente, llamada *búsqueda binaria*.

La idea es comparar primero x contra el elemento del centro del arreglo. Si tenemos suerte, lo encontramos ahí, pero incluso si no tenemos suerte, podemos de inmediato descartar la mitad del arreglo. En efecto, si x es mayor que el elemento del centro, entonces basta seguir buscando en la segunda mitad. De la misma manera, si x es menor, basta seguir buscando en la primera mitad.

```
import numpy as np
a=np.array([12,25,29,34,45,53,59,67,86,92])
```

```
# Búsqueda binaria, versión recursiva
# busca x en el arreglo a, retorna subíndice o -1 si no está
def bbin(x,a):
    # Definimos una función auxiliar para
    # buscar en el subarreglo a[i],...,a[j]
    def bbin_rec(x,a,i,j):
        if i>j:
            return -1
        k=(i+j)//2
        if x==a[k]:
            return k
        if x<a[k]:
            return bbin_rec(x,a,i,k-1)
        else:
            return bbin_rec(x,a,k+1,j)

    # puntapié inicial
    n=len(a)
    return bbin_rec(x,a,0,n-1)
```



```
print(bbin(12,a), bbin(53,a), bbin(92,a), bbin(30,a))
```

```
0 5 9 -1
```

```
# Búsqueda binaria, versión iterativa
# busca x en el arreglo a, retorna subíndice o -1 si no está
def bbin(x,a):
    n=len(a)
    i=0
    j=n-1
    while i<=j:
        k=(i+j)//2
        if x==a[k]:
            return k
        if x<a[k]:
            j=k-1
        else:
            i=k+1
    return -1
```

```
print(bbin(12,a), bbin(53,a), bbin(92,a), bbin(30,a))
```

```
0 5 9 -1
```

Podemos estimar rápidamente la eficiencia de este algoritmo si vemos que para hacer una búsqueda en un conjunto de tamaño n , después de acceder el elemento del medio, en el peor caso continuamos buscando en un conjunto de tamaño aproximadamente igual a la mitad:

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

Aplicando el Teorema Maestro con $p = 1$, $q = 2$, $r = 0$, vemos que $T(n) = \Theta(\log n)$. Por lo tanto, gracias a que el arreglo está ordenado, una búsqueda binaria es mucho más eficiente que una búsqueda secuencial.

La ecuación anterior nos permite obtener una estimación rápida, pero no refleja de manera totalmente exacta lo que ocurre en el algoritmo. Si queremos modelar de manera precisa lo que ocurre en el peor caso, la ecuación correcta es

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

$$T(1) = 1$$

donde la notación “techo” $\lceil x \rceil$ denota el menor entero mayor o igual a x (y, similarmente, la notación “piso” $\lfloor x \rfloor$ denota el mayor entero menor o igual a x).

Si tabulamos el valor de la función $T(n)$ para los primeros valores de n , tenemos:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T(n)$	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5

Observando esta tabla, no es difícil adivinar la solución:

$$T(n) = \lceil \log_2(n+1) \rceil$$

Ejercicio 4.1

Demostrar que $T(n) = \lceil \log_2(n+1) \rceil$.

Una manera más eficiente de programar la búsqueda binaria

En el análisis anterior, hemos considerado que en cada iteración, el costo de acceder el elemento $a[k]$ es igual a 1, representando así el costo total de comparar primero con $=$ y luego con $<$. Si quisiéramos hacer una contabilidad más precisa, deberíamos decir que ese costo es en realidad de 2 comparaciones por cada iteración. A continuación veremos que es posible reducir eso a 1 comparación por ciclo, si utilizamos comparaciones de tipo $<=$:

```
# Búsqueda binaria, versión iterativa y con <=
# busca x en el arreglo a, retorna subíndice o -1 si no está
def bbin(x,a):
    n=len(a)
    i=0
    j=n-1
    while i<j: # conjunto tiene al menos 2 elementos
        k=(i+j)//2
        if x<=a[k]:
            j=k # x estaría en a[i],...,a[k]
        else:
            i=k+1 # x estaría en a[k+1],...,a[j]
    # al terminar, el conjunto factible se ha reducido a 0 o 1 elemento
    if i==j and x==a[i]:
        return i
    else:
        return -1
```

```
print(bbin(12,a), bbin(53,a), bbin(92,a), bbin(30,a))
```

```
0 5 9 -1
```

En esta versión logramos ahorrar una comparación de elementos por iteración, al precio de que toda las búsquedas ahora hacen el máximo de iteraciones, a diferencia del algoritmo original, en donde si teníamos suerte el algoritmo buscado se podría encontrar en las primeras iteraciones.

Este es un precio que vale la pena pagar, porque en el algoritmo original son muy pocos los casos en que la búsqueda termina tempranamente, y en la gran mayoría de los casos igual se hace un número de iteraciones muy cercano al máximo.

Estructuras enlazadas

Como hemos visto, los arreglos permiten que algunos algoritmos se puedan programar de manera muy eficiente, pero las estructuras basadas en arreglos suelen ser muy rígidas. Por ejemplo, si quisiéramos agregar un nuevo elemento al arreglo ordenado en que se hace búsqueda binaria (suponiendo que hubiera holgura suficiente), la inserción tomaría tiempo $\Theta(n)$ tanto en el peor caso como en promedio, por la necesidad de preservar el orden de los elementos.

Veremos a continuación que podemos diseñar estructuras mucho más flexibles sin hacemos uso de la capacidad de definir clases de objetos que contienen dentro de sus campos referencias (también llamadas “punteros”) a otros objetos.

Listas de enlace simple

Comenzaremos viendo la estructura más sencilla de este tipo: una secuencia de nodos, en que cada uno contiene una referencia al siguiente de la lista. Consideremos nodos compuestos de dos *campos* (o *atributos*): *info* y *sgte*. El primero almacena el elemento de la secuencia, y el segundo apunta al siguiente nodo. Por ejemplo, un nodo que almacena el valor 42 y que apunta al siguiente nodo se puede representar gráficamente así:

O, más simplemente:

Para definir el formato de estos nodos utilizaremos la siguiente definición de clase, la que incluye un constructor para inicializar sus campos al crear un objeto:

Figure 14: Nodo

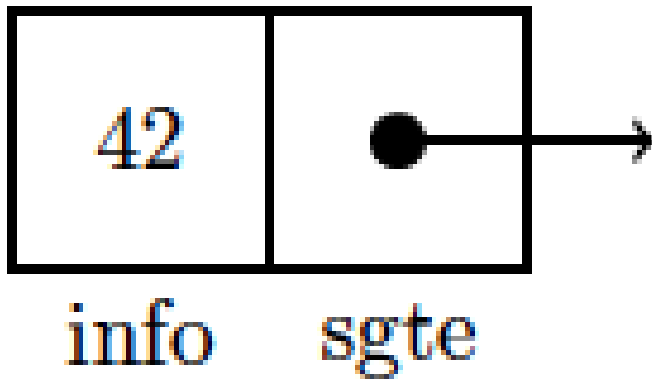
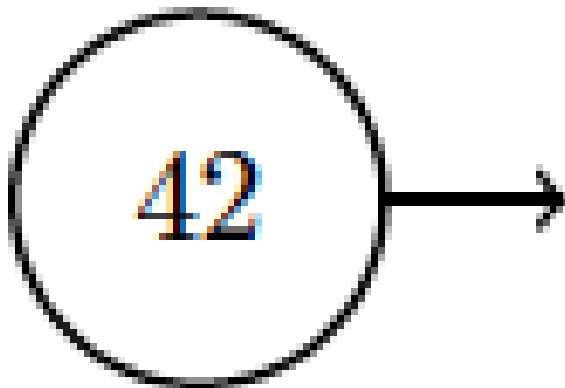


Figure 15: Nodo-circular



```
class Nodo:
    def __init__(self, info, sgte=None):
        self.info=info
        self.sgte=sgte
```

```
p=Nodo(42)
print(p.info, p.sgte)
```

42 None

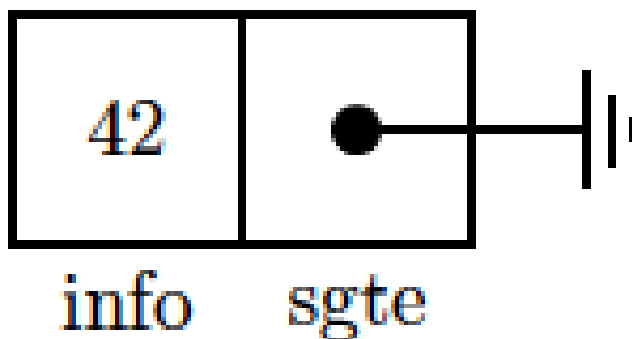
El siguiente trozo de programa muestra la construcción de una lista con 4 elementos: 42, 65, 13 y 44, y un ejemplo simple de uso:

```
primero=Nodo(42,Nodo(65,Nodo(13,Nodo(44))))
p=primero
while p is not None:
    print(p.info, end=" ")
    p=p.sgte
print()
```

42 65 13 44

Algo adicional respecto de la representación gráfica. Cuando una referencia es nula (None), es tradicional representarla como “conectada a tierra”:

Figure 16: Nodo-None



Al usar la representación con nodos circulares, la ausencia de un nodo siguiente la podemos representar simplemente por la ausencia de la flecha saliente:

Figure 17: Nodo-circular-None



O, si queremos hacer explícita la ausencia de un nodo siguiente (o, en otras palabras, que el puntero al nodo siguiente es nulo), podemos representarlo por un nodo cuadrado, que es una convención que nos resultará muy conveniente más adelante, al ver *árboles*:

Con esta última convención, la lista que construimos en el ejemplo anterior, se visualizaría así:

A continuación definiremos una clase `Lista`, que contendrá el puntero al primer nodo de la lista, así como la funcionalidad que necesitamos para operar sobre la lista:

```
import aed_utilities as aed

class Lista:
    def __init__(self):
        self.primerono=None

    def insertar_al_inicio(self,info):
        self.primerono=Nodo(info,self.primerono)

    def insertar_despues_de(self,p,info): # inserta después de nodo p
        p.sgte=Nodo(info,p.sgte)
```

Figure 18: Nodo-circular-cuadrado

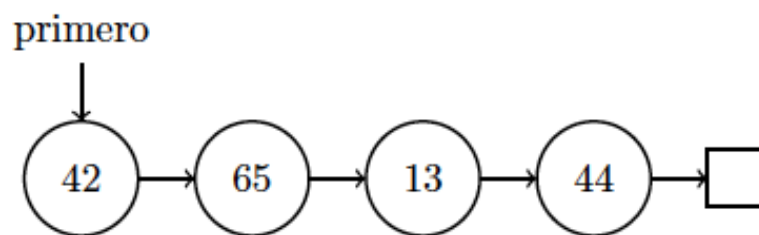
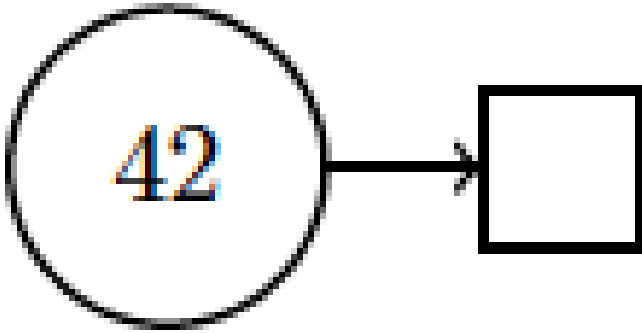


Figure 19: lista-ejemplo

```

def eliminar_al_inicio(self):
    assert self.primeros is not None
    self.primeros=self.primeros.sgte

def eliminar_sgte_de(self,p): # elimina el nodo siguiente de p
    assert p.sgte is not None
    p.sgte=p.sgte.sgte

def k_esimo(self,k): # retorna k-esimo nodo, o None si fuera de rango
    p=self.primeros
    j=1
    while p is not None:
        if j==k:
            return p
        p=p.sgte
        j+=1
    return None

def imprimir(self):
    p=self.primeros
    while p is not None:
        print(p.info, end=" ")
        p=p.sgte
    print()

#Método para dibujar una lista enlazada.
# - Se crea una instancia de la clase LinkedListDrawer. Este objeto necesita conocer cómo se llaman los
# campos de la estructura de los nodos de la lista. Para eso sirven los parámetros:
#     - fieldHeader: nombre del campo en la lista con el primer puntero de la lista
#     - fieldData: nombre del campo en el nodo que almacena la data
#     - fieldLink: nombre del campo en el nodo que almacena el puntero
#     - strHeader: cadena de texto que representa la cabecera de la lista
# - Para dibujar se llama al método "draw_linked_list" con la lista como parámetro

def dibujar(self):
    lld = aed.LinkedListDrawer(fieldHeader="primeros", fieldData="info", fieldLink="sgte", strHeader="prim
    lld.draw_linked_list(self)

```

```

L=Lista()
L.insertar_al_inicio(44)
L.insertar_al_inicio(13)
L.insertar_al_inicio(65)
L.insertar_al_inicio(42)

```



```
L.imprimir()
```

```
42 65 13 44
```

```
L.dibujar()
```

Figure 20: svg

```
L.insertar_despues_de(L.k_esimo(2),88)
```

```
L.imprimir()
```

```
L.dibujar()
```

```
42 65 88 13 44
```

Figure 21: svg

```
L.eliminar_al_inicio()
```

```
L.imprimir()
```

```
L.dibujar()
```

```
65 88 13 44
```

Figure 22: svg

```
L.eliminar_sgte_de(L.k_esimo(1))
```

```
L.imprimir()
```

```
L.dibujar()
```

```
65 13 44
```

Ejercicio 4.2

Escriba una función que pueda ser invocada como `L.reversar()`, que al ejecutarse re-enlace los nodos de la lista de modo que queden en el orden opuesto al original, en tiempo lineal en el largo de la lista. Esto debe hacerse solo modificando punteros, sin crear nuevos nodos. Escriba a continuación la definición de la clase `Lista` incluyendo la función `reversar`.

Figure 23: svg

Luego pruebe su función:

```
L=Lista()
L.insertar_al_inicio(44)
L.insertar_al_inicio(13)
L.insertar_al_inicio(65)
L.insertar_al_inicio(42)
L.imprimir()
L.reversar()
L.imprimir()
```

A continuación, escriba instrucciones para probar esta función para reversar una lista de largo 0 y una lista de largo 1.

Hay algunas cosas que no resultan muy elegantes en el diseño de listas que estamos considerando hasta el momento.

Una de ellas es que para eliminar un elemento, no se pueda indicar al elemento que se desea eliminar, sino que haya que indicar al previo. Esto es inevitable, dado el carácter unidireccional de los enlaces, y más adelante, cuando veamos *listas de doble enlace* veremos que eso puede mejorarse.

Otro punto molesto en la interfaz de uso es la necesidad de distinguir entre si se opera al comienzo de la lista, o en un punto interior. Esto es porque las operaciones afectan al elemento previo, y el primero de la lista, por definición, no tiene un elemento previo.

Esto puede subsanarse, sin embargo, introduciendo un nodo “cabecera” (“*header*”) al comienzo de la lista. Este nodo no contiene información útil y para todos los efectos es como si no existiera, excepto que sirve como el previo del primer nodo real de la lista. Para poder ubicarlo, lo identificaremos con el nodo “o-ésimo” de la lista.

A continuación reescribimos nuestra definición de la clase `Lista` y sus ejemplos de uso, bajo el supuesto de que existe un nodo cabecera.

```
class Lista_con_cabecera:
    def __init__(self):
        self.cabecera=Nodo(0,None)
```

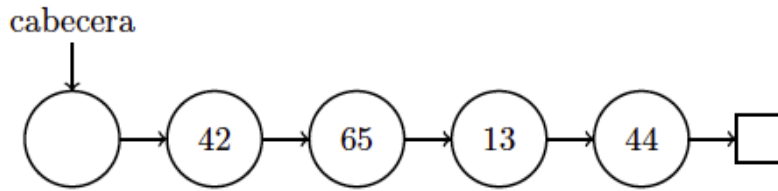


Figure 24: lista-ejemplo-con-cabecera

```

def insertar_despues_de(self,p,info): # inserta después de nodo p
    p.sgte=Nodo(info,p.sgte)

def eliminar_sgte_de(self,p): # elimina el nodo siguiente de p
    assert p.sgte is not None
    p.sgte=p.sgte.sgte

def k_esimo(self,k): # retorna k-esimo nodo, o None si fuera de rango
    p=self.cabecera
    j=0
    while True:
        if j==k:
            return p
        p=p.sgte
        if p is None:
            return None
        j+=1

def imprimir(self):
    p=self.cabecera.sgte
    while p is not None:
        print(p.info,end=" ")
        p=p.sgte
    print()

```

```

L=Lista_con_cabecera()
L.insertar_despues_de(L.k_esimo(0),42)
L.insertar_despues_de(L.k_esimo(1),65)
L.insertar_despues_de(L.k_esimo(2),13)
L.insertar_despues_de(L.k_esimo(3),44)
L.imprimir()

```

42 65 13 44

```
L.eliminar_sgte_de(L.k_esimo(0)) # eliminar el primero
L.imprimir()
```

```
65 13 44
```

```
print(L.k_esimo(7))
```

```
None
```

```
L.eliminar_sgte_de(L.k_esimo(1)) # eliminar un elemento en el medio
L.imprimir()
```

```
65 44
```

Recorriendo la lista con un iterador

A continuación veremos cómo, en lugar de imprimir la lista, podemos implementar un iterador que vaya entregando los elementos de la lista cada vez que es llamado:

```
class Lista_con_cabecera:
    def __init__(self):
        self.cabecera=Nodo(0,None)

    def insertar_despues_de(self,p,info): # inserta después de nodo p
        p.sgte=Nodo(info,p.sgte)

    def eliminar_sgte_de(self,p): # elimina el nodo siguiente de p
        assert p.sgte is not None
        p.sgte=p.sgte.sgte

    def k_esimo(self,k): # retorna k-esimo nodo, o None si fuera de rango
        p=self.cabecera
        j=0
        while p is not None:
            if j==k:
                return p
            p=p.sgte
            j+=1
        return None

    def valores(self):
        p=self.cabecera.sgte
        while p is not None:
            yield p.info
            p=p.sgte
```

```
L=Lista_con_cabecera()
L.insertar_despues_de(L.k_esimo(0),42)
L.insertar_despues_de(L.k_esimo(1),65)
L.insertar_despues_de(L.k_esimo(2),13)
L.insertar_despues_de(L.k_esimo(3),44)
```

```
for x in L.valores():
    print(x, end=" ")
print()
```

```
42 65 13 44
```

```
print([x for x in L.valores()])
```

```
[42, 65, 13, 44]
```

Listas de doble enlace

Las listas de enlace simple permiten solo procesos unidireccionales, por lo que no son muy apropiadas cuando los procesos necesitan poder recorrerlas en ambas direcciones.

Podemos mejorar esto si agregamos a los nodos una referencia al nodo *previo*, además del nodo siguiente:

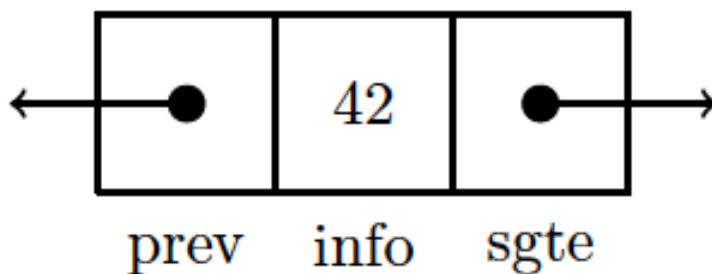


Figure 25: Nodo-doble

```
class Nodo:
    def __init__(self, prev, info, sgte):
        self.prev=prev
        self.info=info
        self.sgte=sgte
```

Con este tipo de nodos podemos formar una lista que puede ser recorrida en ambas direcciones. Por consideraciones similares a las anteriores, resulta conveniente agregar un nodo cabecera en cada extremo, pero en realidad un mismo nodo puede jugar ambos roles, con lo cual la lista adopta un aspecto físicamente circular, aunque desde un punto de vista conceptual no lo sea:

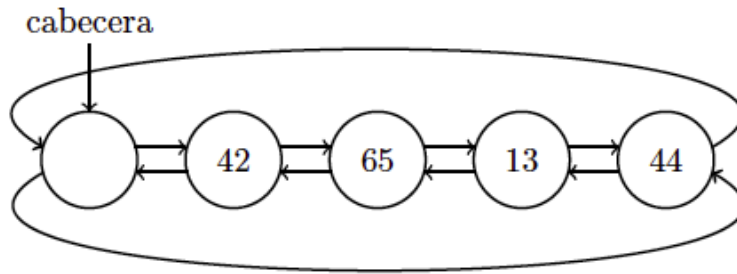


Figure 26: lista-ejemplo-doble-enlace

La siguiente es una definición de lista de doble enlace, con alguna de la funcionalidad que ella permite:

```
import aed_utilities as aed

class Lista_doble_enlace:
    def __init__(self):
        self.cabecera=Nodo(None,0,None)
        self.cabecera.prev=self.cabecera
        self.cabecera.sgte=self.cabecera

    def insertar_despues_de(self,p,info): # inserta después de nodo p
        r=p.sgte
        p.sgte=r.prev=Nodo(p,info,r)

    def eliminar(self,p): # elimina el nodo p
        assert p is not self.cabecera
        (p.prev.sgte,p.sgte.prev)=(p.sgte,p.prev)

    def k_esimo(self,k): # retorna k-esimo nodo, o None si fuera de rango
        p=self.cabecera
        j=0
        while True:
            if j==k:
                return p
            p=p.sgte
```

```

        if p is self.cabecera:
            return None
        j+=1

    def ascendente(self):
        p=self.cabecera.sgte
        while p is not self.cabecera:
            yield p.info
            p=p.sgte

    def descendente(self):
        p=self.cabecera.prev
        while p is not self.cabecera:
            yield p.info
            p=p.prev

#Para dibujar una lista doblemente enlazada, es necesario además definir el nombre del campo del puntero
# El parámetro para definir ese campo es "fieldReverseLink". Por default, este campo es None.
    def dibujar(self):
        lld=aed.LinkedListDrawer(fieldHeader="cabecera", fieldData="info", fieldLink="sgte", fieldReverseLink="prev")
        lld.draw_double_linked_list(self)

```

```

L=Lista_doble_enlace()
L.insertar_despues_de(L.k_esimo(0),42)
L.insertar_despues_de(L.k_esimo(1),65)
L.insertar_despues_de(L.k_esimo(2),13)
L.insertar_despues_de(L.k_esimo(3),44)
print([x for x in L.ascendente()])
print([x for x in L.descendente()])

L.dibujar()

```

```
[42, 65, 13, 44]
```

```
[44, 13, 65, 42]
```

Figure 27: svg

```

L.eliminar(L.k_esimo(3))
print([x for x in L.ascendente()])

L.dibujar()

```

```
[42, 65, 44]
```

Ejercicio 4.3

Suponga que por un accidente, o quizás por vandalismo, todos los punteros prev han sido destruidos. Afortunadamente, los punteros sgte están intactos. Usted debe escribir una función que pueda invocarse como `L.repara_prev()` que reconstruya todos los punteros faltantes. Escriba a continuación la definición de la clase `Lista_doble_enlace` incluyendo la nueva función:

Pruébela a continuación:

```
L=Lista_doble_enlace()
L.insertar_despues_de(L.k_esimo(0),42)
L.insertar_despues_de(L.k_esimo(1),65)
L.insertar_despues_de(L.k_esimo(2),13)
L.insertar_despues_de(L.k_esimo(3),44)
print([x for x in L.ascendente()])
L.repara_prev()
print([x for x in L.descendente()])
```

Árboles Binarios

Al usar nodos que hacen referencia a otros nodos, no es de ninguna manera obligatorio limitarse a estructuras lineales como las que hemos visto en las secciones anteriores: podemos construir estructuras enlazadas tan complejas como queramos.

Un tipo de estructura muy utilizada son los *árboles binarios*, en que cada nodo puede tener “hijos” tanto a su izquierda como a su derecha, y eso mismo se reproduce para los hijos, recursivamente.

Los nodos tienen un formato similar al de los nodos de doble enlace, pero las referencias se llaman *izq* (izquierda) y *der* (derecha).

Al dibujarlo con nodos circulares, normalmente las líneas no llevan flecha, porque se entiende que apuntan hacia abajo:

```
class Nodo:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der
```


Figure 29: Nodo-arbol-binario

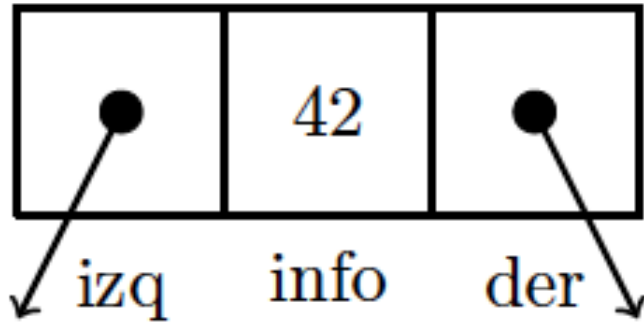
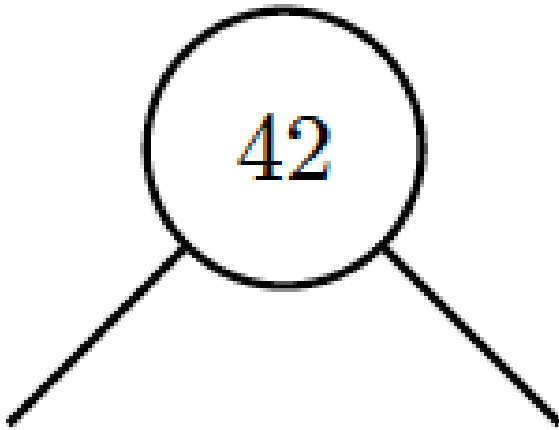


Figure 30: Nodo-arbol-binario-circular



La siguiente figura muestra un ejemplo de un árbol binario:

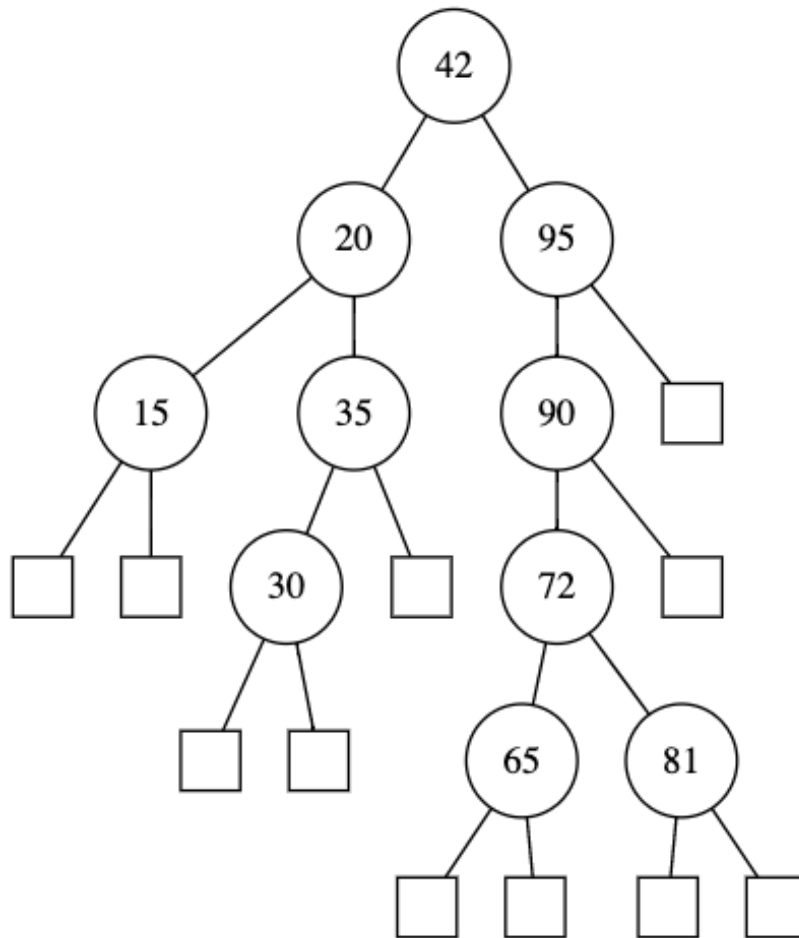


Figure 31: ejemplo-ABB

Este es un tipo especial de árbol binario, llamado *árbol de búsqueda binaria (ABB)*, que se caracteriza porque para cada nodo, sus hijos descendientes hacia la izquierda son menores que él, y los de la derecha son mayores. Más adelante estudiaremos en profundidad los ABB.

La terminología asociada a los árboles combina lo forestal con lo genealógico. El nodo de nivel superior se llama la *raíz* y los nodos que están en los niveles inferiores (los nodos cuadrados en la figura) se llaman *hojas*. Como vemos, al revés que en la naturaleza, estos árboles crecen hacia abajo.

Un nodo apunta hacia abajo a sus hijos (izquierdo y derecho), y se dice que es el *padre* de ellos. Yendo desde un nodo hacia abajo se encuentran sus *descendientes*, y hacia arriba se encuentran sus *ancestros*.

Si el nodo b es descendiente del nodo a , se dice que la distancia entre a y b es el número de pasos que hay que dar para ir de a a b . La máxima distancia entre la raíz y una hoja se llama la *altura* del árbol. En el ejemplo, la altura es 5, que se alcanza yendo desde la raíz hasta cualquiera de las hojas hijas de 65 o de 81.

Al dibujar un árbol con nodos circulares y nodos cuadrados, los circulares se llaman *nodos internos* y los cuadrados, *nodos externos*. Los nodos internos siempre tienen dos hijos (internos y/o externos) y los nodos externos no tienen hijos.

La altura máxima de un árbol binario con n nodos internos es n , y la altura mínima es $\lceil \log_2 (n + 1) \rceil$. Esto último es una consecuencia de que un árbol binario de altura h puede tener a lo más 2^h nodos externos.

A continuación definiremos una clase árbol, con un constructor que define un puntero al nodo raíz. Para poder hacer ejemplos de uso, admitiremos que el constructor reciba un puntero a la raíz de un árbol ya construido.

```
class Arbol:
    def __init__(self, raiz=None):
        self.raiz=raiz
```

Recorridos de Árboles Binarios

Un árbol binario es una estructura esencialmente recursiva, y las principales formas de recorrer un árbol se definen también recursivamente. Los tres tipos de recorridos más conocidos son:

- Preorden: Visitar la raíz, recorrer el subárbol izquierdo y recorrer el subárbol derecho
- Indorden: Recorrer el subárbol izquierdo, visitar la raíz y recorrer el subárbol derecho
- Postorden: Recorrer el subárbol izquierdo, recorrer el subárbol derecho y visitar la raíz

A continuación agregamos a la definición de la clase tres métodos que imprimen en contenido del árbol en estos recorridos:

```
import aed_utilities as aed

def pre(p):
    if p is not None:
        print(p.info, end=" ")
        pre(p.izq)
        pre(p.der)
```

```

def ino(p):
    if p is not None:
        ino(p.izq)
        print(p.info,end=" ")
        ino(p.der)

def post(p):
    if p is not None:
        post(p.izq)
        post(p.der)
        print(p.info,end=" ")

class Arbol:
    def __init__(self,raiz=None):
        self.raiz=raiz

    def preorden(self):
        print("Preorden:", end=" ")
        pre(self.raiz)
        print()

    def inorden(self):
        print("Inorden:", end=" ")
        ino(self.raiz)
        print()

    def postorden(self):
        print("Postorden:", end=" ")
        post(self.raiz)
        print()

# Para dibujar un árbol binario, necesitamos crear una instancia de la clase BinaryTreeDrawer
# Aquí también necesitamos conocer la estructura de los nodos. Esta información se envía como parámetro
# al constructor de la clase:
# - fieldData: nombre del campo del nodo que mantiene la data
# - fieldLeft: nombre del campo del nodo con el puntero izquierdo
# - fieldRight: nombre del campo del nodo con el puntero derecho
# Para dibujar se llama al método "draw_tree", enviando como parámetro el árbol binario y el nombre de

    def dibujar(self):
        btd = aed.BinaryTreeDrawer(fieldData="info", fieldLeft="izq", fieldRight="der")
        btd.draw_tree(self, "raiz")

```

```

AX=Arbol(
    Nodo(
        Nodo(
            Nodo(None, 15, None),
            20,
            Nodo(
                Nodo(None, 30, None),
                35,
                None
            )
        ),
        42,
        Nodo(
            Nodo(
                Nodo(
                    Nodo(Nodo(None, 62, None), 65, None),
                    72,
                    Nodo(None, 81, None)
                ),
                90,
                None
            ),
            95,
            None
        )
    )
)

```

Figure 32: svg

en iteradores, los cuales pueden ser utilizados desde una instrucción for:

```
class Nodo:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der
```

```
def pre(p):
    if p is not None:
        yield p
        yield from pre(p.izq)
        yield from pre(p.der)
```

```
def ino(p):
    if p is not None:
        yield from ino(p.izq)
        yield p
        yield from ino(p.der)
```

```
def post(p):
    if p is not None:
        yield from post(p.izq)
        yield from post(p.der)
        yield p
```

```
class Arbol:
    def __init__(self, raiz=None):
        self.raiz=raiz

    def preorden(self):
        yield from pre(self.raiz)

    def inorden(self):
        yield from ino(self.raiz)

    def postorden(self):
        yield from post(self.raiz)
```

```

a=Arbol(
    Nodo(
        Nodo(
            Nodo(None,15,None),
            20,
            Nodo(
                Nodo(None,30,None),
                35,
                None
            )
        ),
        42,
        Nodo(
            Nodo(
                Nodo(
                    Nodo(None,65,None),
                    72,
                    Nodo(None,81,None)
                ),
                90,
                None
            ),
            95,
            None
        )
    )
)

```

```

print ("Preorden:", [p.info for p in a.preorden()])
print ("Inorden:", [p.info for p in a.inorden()])
print ("Postorden:", [p.info for p in a.postorden()])

```

```

Preorden: [42, 20, 15, 35, 30, 95, 90, 72, 65, 81]
Inorden: [15, 20, 30, 35, 42, 65, 72, 81, 90, 95]
Postorden: [15, 30, 35, 20, 65, 81, 72, 90, 95, 42]

```

Una representación alternativa para árboles binarios

Un diseño alternativo para esta estructura se basa en darle una existencia real a los nodos externos, en lugar de que sean punteros None. Esto nos permite asociar funcionalidad a los nodos, lo cual ejemplificamos con el recorrido en inorden:

```

class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der
    def inorden(self):
        self.izq.inorden()
        print(self.info, end=" ")
        self.der.inorden()

```

```

class Nodoe:
    def __init__(self):
        pass
    def inorden(self):
        pass

```

```
import aed_utilities as aed
```

```

class Arbol:
    def __init__(self, raiz=Nodoe()):
        self.raiz=raiz

    def inorden(self):
        print("Inorden:", end=" ")
        self.raiz.inorden()
        print()

    def dibujar(self):
        btd = aed.BinaryTreeDrawer(fieldData="info", fieldLeft="izq", fieldRight="der", classNone=Nodoe)
        btd.draw_tree(self, "raiz")

```

```

a=Arbol(
    Nodoi(
        Nodoi(
            Nodoi(Nodoe(), 15, Nodoe()),
            20,
            Nodoi(
                Nodoi(Nodoe(), 30, Nodoe()),
                35,
                Nodoe()
            )
        ),
        42,
        Nodoi(

```



```

        Nodoi(
            Nodoi(
                Nodoi(Nodo(), 65, Nodo()),
                72,
                Nodoi(Nodo(), 81, Nodo())
            ),
            90,
            Nodo()
        ),
        95,
        Nodo()
    )
)

```

```
a.inorden()
```

```
Inorden: 15 20 30 35 42 65 72 81 90 95
```

```
a.dibujar()
```

Figure 33: svg

Recorrido usando iteradores

De la misma manera como lo hicimos antes, podemos usar la instrucción `yield` para entregar uno a uno los nodos que se van visitando. Ejemplificamos esto con el recorrido en inorden:

```

class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

    def inorden(self):
        yield from self.izq.inorden()
        yield self
        yield from self.der.inorden()

class Nodoe:
    def __init__(self):
        pass

```

```
def inorden(self):
    return
    yield None # no se ejecuta, permite que la función sea un generator
```

```
class Arbol:
    def __init__(self, raiz=Node()):
        self.raiz=raiz

    def inorden(self):
        yield from self.raiz.inorden()
```

```
a=Arbol(
    Nodei(
        Nodei(
            Nodei(Node(), 15, Node()),
            20,
            Nodei(
                Nodei(Node(), 30, Node()),
                35,
                Node()
            )
        ),
        42,
        Nodei(
            Nodei(
                Nodei(
                    Nodei(Node(), 65, Node()),
                    72,
                    Nodei(Node(), 81, Node())
                ),
                90,
                Node()
            ),
            95,
            Node()
        )
    )
)
```

```
print ("Inorden:", [p.info for p in a.inorden()])
```

```
Inorden: [15, 20, 30, 35, 42, 65, 72, 81, 90, 95]
```

Árboles para representar fórmulas

La estructura de una fórmula matemática, por ejemplo la fórmula

$$(a + 1) * \left(2 - \frac{1}{b}\right)$$

se puede representar mediante el árbol:

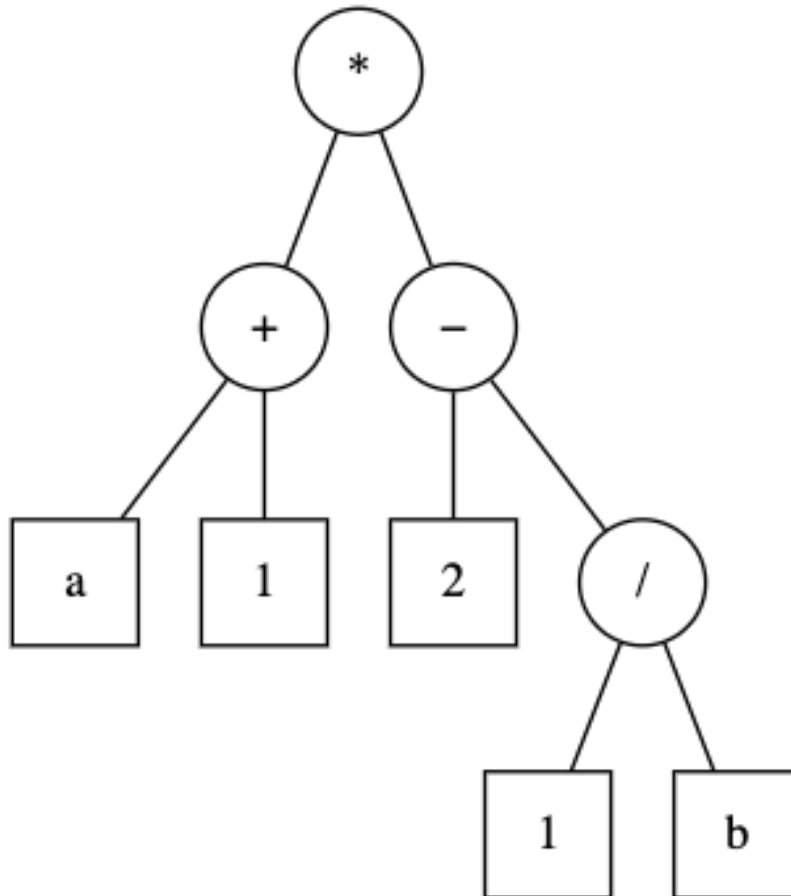


Figure 34: arbol-formula

Modifiquemos la definición de nodos externos para que puedan almacenar información en su interior y veamos el efecto de hacer un recorrido en postorden de un árbol de este tipo:

```

class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der
    def postorden(self):
  
```

```

        self.izq.postorden()
        self.der.postorden()
        print(self.info, end=" ")

class Nodoe:
    def __init__(self, info=""):
        self.info=info
    def postorden(self):
        print(self.info, end=" ")

class Arbol:
    def __init__(self, raiz=Nodoe()):
        self.raiz=raiz

    def postorden(self):
        print("Postorden:", end=" ")
        self.raiz.postorden()
        print()

    def dibujar(self):
        btd = BinaryTreeDrawer()
        btd.registerNodeNames("info", "izq", "der")
        btd.registerClassNone(Nodoe)
        btd.drawTree(self, "raiz")

```

```

formula= Arbol(
    Nodoi(
        Nodoi(Nodoe("a"), "+", Nodoe("1")),
        "*",
        Nodoi(
            Nodoe("2"),
            "-",
            Nodoi(Nodoe("1"), "/", Nodoe("b"))
        )
    )
)

formula2= Arbol(
    Nodoi(
        Nodoi(Nodoe("a"), "ss", Nodoe("1")),
        "ff",
        Nodoi(
            Nodoe("2"),

```

```

        "uu",
        Nodoi(Nodoe("1"), "ii", Nodoe("b"))
    )
)
)

```

```
formula.postorden()
```

Postorden: a 1 + 2 1 b / - *

```
formula2.dibujar()
```

```
"ss"[pos="-0.7,-0.6!" shape=circle] "ff"[pos="0.0,0.0!" shape=circle] "uu"[pos="0.7,-0.6!" shape=circle] "
```

Figure 35: svg

El resultado de este recorrido en postorden es la misma fórmula escrita en *notación polaca de postfijo* (también llamada *notación polaca reversa* o, más simplemente, *notación polaca*). Esta notación, inventada en 1924 por el lógico polaco Jan Łukasiewicz, se caracteriza porque el operador va a continuación de los operandos, mientras que en la notación usual (llamada de *infijo*) el operador va entre los operandos. Por ejemplo, la fórmula " $a + b$ " se escribe en notación polaca como " $ab+.$ "

La notación polaca tiene varias ventajas. Una de ellas es que no necesita paréntesis. Por ejemplo, si no consideramos prioridad de operadores (que es una forma implícita de parentizar), la fórmula " $a + b * c$ " sería ambigua, porque podría significar " $(a + b) * c$ " o " $a + (b * c)$." En notación polaca no habría ambigüedad, porque la primera se escribiría " $ab + c*.$ " y la segunda sería " $abc * +.$ "

Otra ventaja es que, como veremos más adelante, una fórmula en notación polaca se puede evaluar en una sola pasada de izquierda a derecha haciendo uso de una estructura llamada *pila* o *stack*.

Ejercicio 4.4

Suponga que los campos `info` de los nodos externos contienen solo números y escriba una función que pueda invocarse como `formula.evaluar()`, que al ser ejecutada entregue el valor numérico de la fórmula representada por el árbol. Escriba a continuación la definición de la clase `Arbol` que incluya la nueva función:

Pruébela a continuación:

```
formula= Arbol(
    Nodoi(
        Nodoi(Nodoe(5), "+", Nodoe(2)),
        "*",
        Nodoi(
            Nodoe(8),
            "-",
            Nodoi(Nodoe(9), "/", Nodoe(3))
        )
    )
)
print(formula.evaluar())
```

Propiedades matemáticas de los árboles binarios

Los árboles binarios tienen muchas propiedades interesantes:

Relación entre nodos internos y externos

Sea e_n el número de nodos externos de un árbol binario con n nodos internos. Entonces $e_n = n + 1$

Esta propiedad se puede demostrar de varias maneras:

Demostración 1:

Por inducción sobre el número de nodos internos. La base es un árbol vacío ($raiz == None$), para la cual $e_0 = 1$. Para el paso inductivo, consideremos un árbol con $n + 1$ nodos internos. Ese árbol debe tener al menos un nodo interno cuyos dos hijos son nodos externos (o el árbol sería infinito). Tomemos el subárbol constituido por ese nodo y sus dos hijos y reemplacémoslo por un nodo externo. El árbol resultante tiene n nodos y por lo tanto su número de nodos externos es $n + 1$, por hipótesis de inducción. Deshagamos ahora el cambio efectuado: esto agrega 1 interno, y elimina 1 nodo externo pero agrega 2, de modo que el incremento neto en el número de nodos externos es 1. Por lo tanto el número de nodos externos del árbol resultante es $e_{n+1} = n + 2$. QED

Demostración 2:

Por inducción sobre la estructura del árbol. Si el árbol es vacío, trivialmente se cumple $e_0 = 1$. Si el árbol es no vacío, digamos tiene $n + 1$ nodos, entonces consiste de una raíz, más un subárbol izquierdo y un subárbol derecho. Si el subárbol izquierdo tiene k nodos internos, entonces el derecho tiene $n - k$ nodos internos. El

número total de nodos externos es $e_{n+1} = e_k + e_{n-k}$. Usando la hipótesis de inducción, $e_{n+1} = (k+1) + (n-k+1) = n+2$. QED

Demostración 3:

Por recorrido en inorden. Supongamos que hacemos un recorrido en inorden, escribiendo un `""` cada vez que visitamos un nodo interno, y un `""` cada vez que visitamos un nodo externo.

```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der
    def inorden(self):
        self.izq.inorden()
        print("", end=" ")
        self.der.inorden()
```

```
class Nodoe:
    def __init__(self, info=""):
        self.info=info
    def inorden(self):
        print("", end=" ")
```

```
class Arbol:
    def __init__(self, raiz=Nodoe()):
        self.raiz=raiz
    def inorden(self):
        print("Inorden:", end=" ")
        self.raiz.inorden()
        print()
```

```
a=Arbol(
    Nodoi(
        Nodoi(
            Nodoi(Nodoe(), 15, Nodoe()),
            20,
            Nodoi(
                Nodoi(Nodoe(), 30, Nodoe()),
                35,
                Nodoe()
            )
        ),
        42,
        Nodoi(
```

```

        Nodoi(
            Nodoi(
                Nodoi(Nodoi(), 65, Nodoi()),
                72,
                Nodoi(Nodoi(), 81, Nodoi())
            ),
            90,
            Nodoi()
        ),
        95,
        Nodoi()
    )
)

```

```
a.inorden()
```

Inorden:

Dado que los nodos se visitan intercalando nodos externos e internos, y en ambos extremos hay nodos externos, claramente $e_n = n + 1$. QED

La última no es una demostración rigurosa (aunque se puede formalizar), pero sin duda es la que hace que la propiedad sea vea más obvia.

Relación entre largo de caminos internos y externos

Definamos el *largo de caminos internos (LCI)*, denotado I_n , como

$$I_n = \sum_{x \in \text{Nodos internos}} \text{distancia}(\text{raiz}, x)$$

De manera análoga, *largo de caminos externos (LCE)*, denotado E_n , como

$$E_n = \sum_{y \in \text{Nodos externos}} \text{distancia}(\text{raiz}, y)$$

En el árbol que hemos usado de ejemplo, tenemos $n = 10$, $I_n = 22$, $E_n = 42$.

En general, se cumple que

$$E_n = I_n + 2n$$

Demostración: Queda como ejercicio. Sugerencia: aplicar inducción sobre la estructura del árbol.

Enumeración de árboles binarios con n nodos

Si tenemos n nodos indistinguibles (no rotulados), llamemos a_n al número de árboles binarios distintos que podemos construir.

Para $n = 0$, tenemos que $a_0 = 1$ (el árbol vacío).

Para $n = 1$, tenemos $a_1 = 1$ (el árbol con un solo nodo). La siguiente figura muestra los árboles que se puede construir con $n = 1$, $n = 2$ y $n = 3$ nodos (mostrando solo los nodos internos):

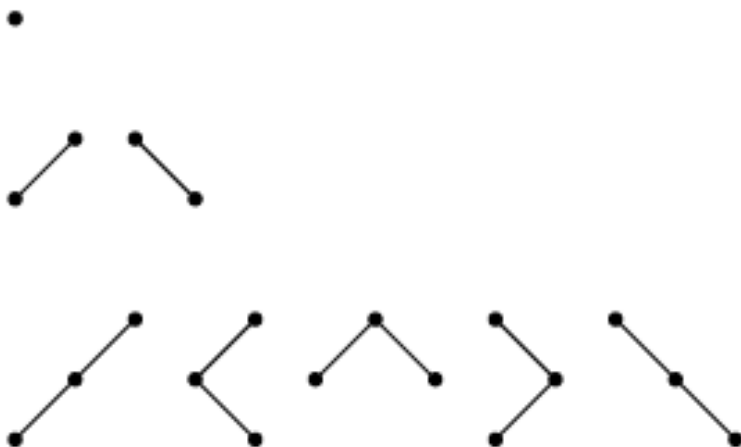


Figure 36: Enum-arboles-binarios

Por lo tanto, tenemos $a_2 = 2$ y $a_3 = 5$. Nos interesa encontrar a_n en el caso general.

Consideremos un árbol con n nodos. Uno de ellos será la raíz, y los $n - 1$ nodos restantes se deberían distribuir a la izquierda y a la derecha. Si a la izquierda quedan k nodos, ahí se puede poner cualquiera de los a_k árboles posible, el que se combinan con el de la derecha, que puede ser cualquiera de los a_{n-k-1} árboles posibles. Como ambas elecciones son independientes, el número de maneras en que se puede hacer es el producto de ambos números, y finalmente hay que sumar sobre todos los posibles valores de k . Esto conduce a la ecuación

$$a_n = \sum_{0 \leq k \leq n-1} a_k a_{n-k-1}$$

con la condición inicial $a_0 = 1$.

Esto nos permite construir una tabla de valores:

```
import numpy as np
def enum_arboles(nmax):
    a=np.zeros(nmax,dtype=int)
    a[0]=1
    for n in range(1,nmax):
```

```

    for k in range(0,n):
        a[n]+=a[k]*a[n-1-k]
    return(a)

```

```
print(enum_arboles(20))
```

```

[      1      1      2      5      14      42
  132     429    1430    4862    16796    58786
 208012   742900  2674440  9694845  35357670  129644790
477638700 1767263190]

```

Para resolver este tipo de ecuaciones necesitamos herramientas matemática más avanzadas, llamadas *funciones generatrices*, de modo que aquí simplemente diremos que la solución es

$$a_n = \frac{1}{n+1} \binom{2n}{n}$$

llamados *números de Catalan*, y podemos comprobar con el siguiente programa que esa fórmula entrega los mismos valores que calculamos recién:

```

# Calcula binomial(n,k) = n*(n-1)*...*(n-k+1)/k!
def binomial(n,k):
    numer=1
    denom=1
    for j in range(0,k):
        denom*=(j+1)
        numer*=(n-j)
    return(numer//denom)

def catalan(n):
    return binomial(2*n,n)//(n+1)

```

```

for n in range(0,12):
    print(catalan(n), end=" ")
print()

```

```
1 1 2 5 14 42 132 429 1430 4862 16796 58786
```

Los números de Catalan crecen muy rápido, y se puede demostrar que

$$a_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

Esto hace que, en particular, cualquier algoritmo de fuerza bruta que necesite recorrer todos los árboles binarios posibles no será práctico para valores grandes de n .

Árboles cardinales (o k -arios)

Los árboles binarios se pueden generalizar a **árboles k -arios**, también llamados **árboles cardinales**. Los nodos internos de un árbol k -ario tienen espacio para k punteros a sus hijos. Cada uno de estos punteros puede referirse a otro nodo interno, o bien ser nulo (visualizado como un nodo externo). La siguiente figura muestra la estructura de un ejemplo de árbol ternario (3-ario):

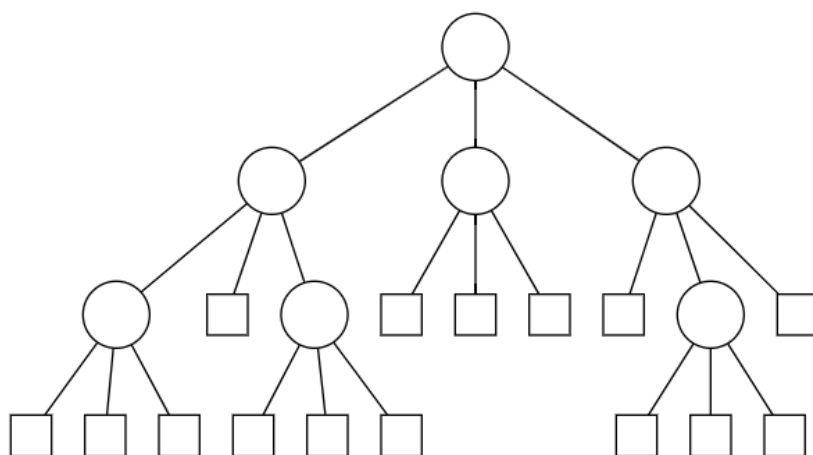


Figure 37: cardinal

Con la adición de los nodos externos, se cumple que los nodos internos siempre tienen exactamente k hijos (que pueden ser nodos internos y/o externos), y los nodos externos no tienen hijos. Tal como en un árbol binario se distingue entre un hijo izquierdo y un hijo derecho, en un árbol k -ario distinguimos entre el hijo número 1, el hijo número 2, etc.

Árboles ordinales (“multiway trees”)

A diferencia de los árboles que hemos visto hasta ahora, en un árbol ordinal cada nodo puede tener un número ilimitado de hijos. Formalmente, un árbol ordinal consiste siempre de un nodo raíz y un conjunto ordenado de cero o más árboles hijos. Por lo tanto:

- Un árbol ordinal nunca puede ser vacío, siempre tiene al menos un nodo
- No existe el concepto de un hijo “faltante,” de modo que no se utilizan nodos externos

El siguiente es un ejemplo de un árbol ordinal:

A primera vista, parecería que los nodos de un árbol ordinal necesitarían tener espacio para una cantidad variable de punteros a

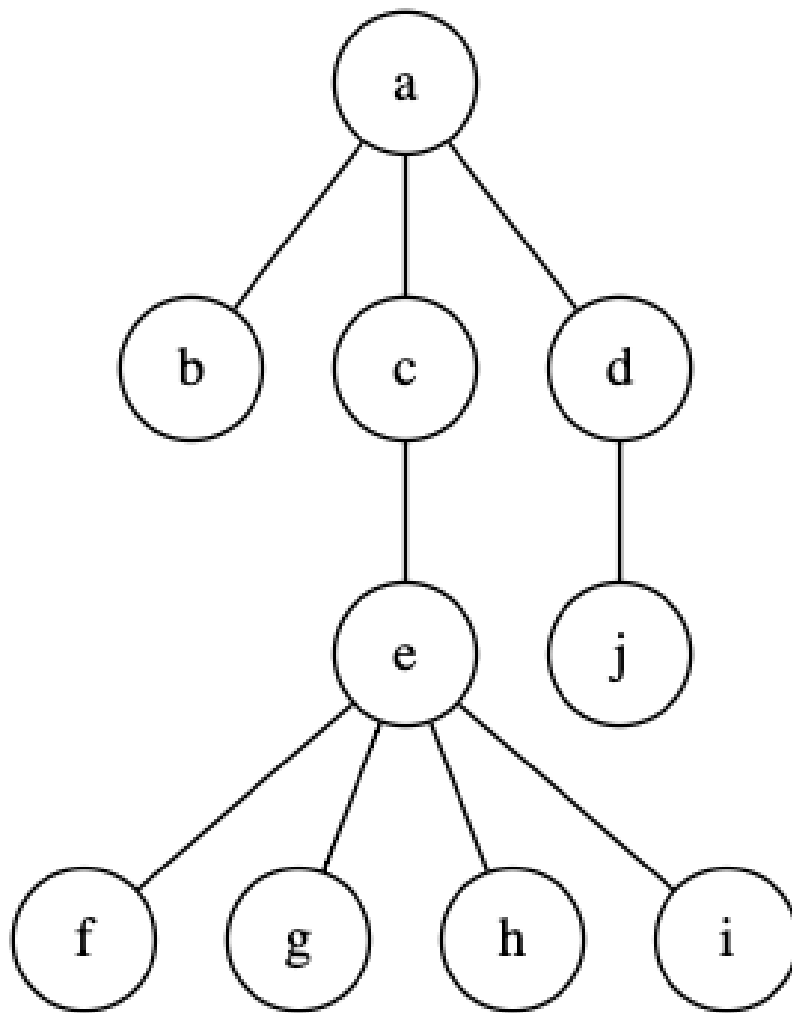


Figure 38: ordinal

sus hijos, pero existe una correspondencia con árboles binarios que permite que el árbol se pueda almacenar usando solo dos punteros en cada nodo.

La idea es la siguiente:

- Cada nodo del árbol ordinal se representa mediante un nodo interno de un árbol binario.
- El puntero izquierdo se utiliza para apuntar al **primer hijo**. Si no hay hijos, se usa un puntero nulo (nodo externo).
- El puntero derecho se utiliza para apuntar al **siguiente hermano**. Si el nodo es el último entre los hermanos, se usa un puntero nulo (nodo externo)

Usando esta técnica, el árbol del ejemplo se representaría así:

Como la raíz del árbol ordinal no tiene hermano, en la representación de árbol binario el puntero derecho de la raíz siempre es nulo. Por lo tanto, hay una biyección entre los árboles ordinales y los árboles binarios en que la raíz no tiene hijo derecho. De esta relación uno a uno podemos deducir que el número de árboles ordinales con n nodos es igual al número de árboles binarios con $n - 1$ nodos internos, esto es, el número de Catalan de orden $n - 1$:

$$\frac{1}{n} \binom{2(n-1)}{n-1}$$

Si la raíz del árbol binario pudiera tener hijo derecho, entonces ese árbol binario estaría representando a un conjunto ordenado de árboles ordinales, lo cual sería un **bosque ordinal**.

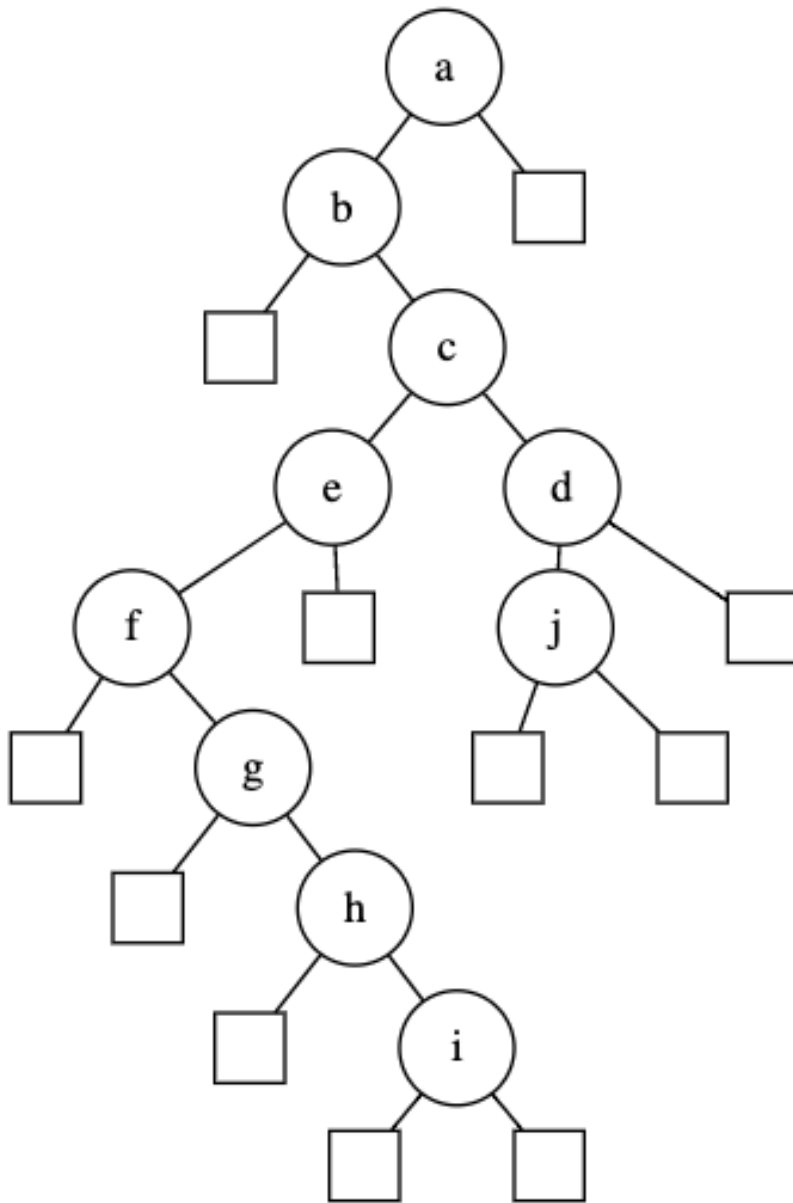


Figure 39: ordinal-binario

5

5 Pilas, Colas y Colas de Prioridad

En este capítulo veremos tres *tipos de datos abstractos* (TDAs) que son muy utilizados.

Un tipo de datos abstracto es un conjunto de datos, más operaciones asociadas, para el cual se aplica una política de “ocultamiento de información”: los usuarios del TDA saben **qué** funcionalidad éste provee, pero no saben **cómo** se implementa esta funcionalidad.

Esta separación de responsabilidad es fundamental para mantener la complejidad bajo control. Sólo los implementadores del TDA necesitan preocuparse de su implementación, y además son libres para modificarla en la medida que la interfaz de uso se mantenga intacta.

Pilas (“Stacks”)

Una **pila**, también llamada *stack* o *pushdown* en inglés, es una lista de elementos en la cual todas las operaciones se realizan solo en un extremo de la lista.

Es usual visualizar la pila creciendo verticalmente hacia arriba, y llamamos “tope” a su extremo superior:

Las dos operaciones básicas son **push** (apilar), que agrega un elemento encima de todos, y **pop** (desapilar), que extrae el elemento del tope de la pila. Más precisamente, si *s* es un objeto de tipo Pila, están disponibles las siguientes operaciones:

- *s.push(x)*: apila *x* en el tope de la pila *s*
- *x=s.pop()*: extrae y retorna el elemento del tope de la pila *s*
- *b=s.is_empty()*: retorna verdadero si la pila *s* está vacía, falso si no

Dado que los elementos salen de la pila en el orden inverso en que ingresaron, esta estructura también se conoce como “lista LIFO,” por

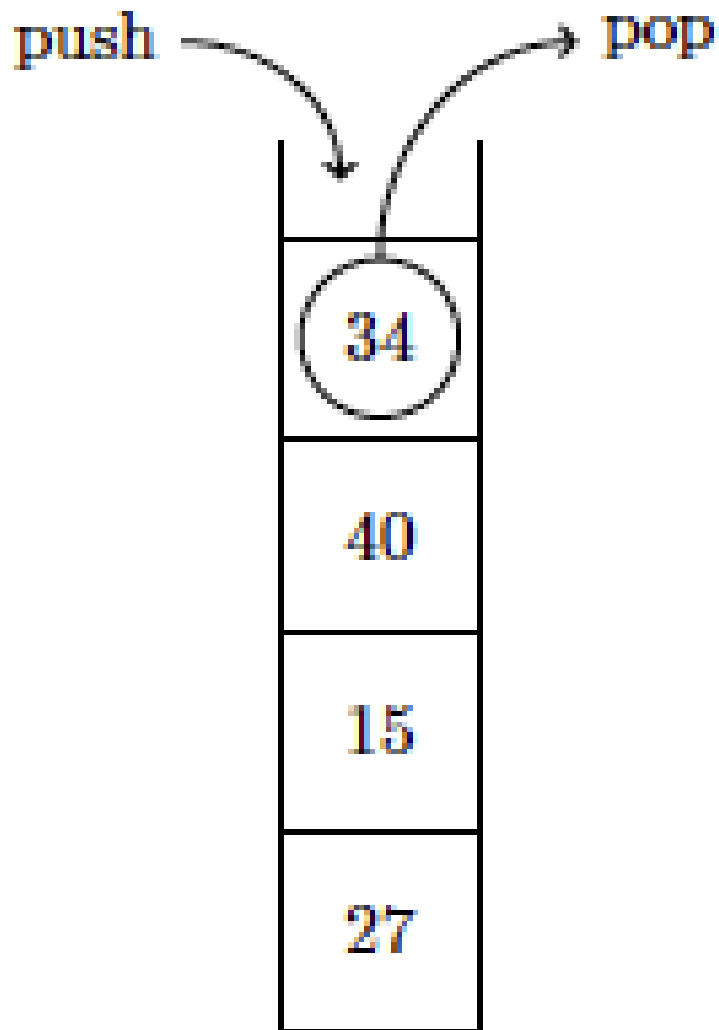


Figure 40: pila

“Last-In-First-Out.”

Implementación usando listas de Python

Es posible implementar una pila muy fácilmente usando las listas que provee el lenguaje Python:

```
class Pila:
    def __init__(self):
        self.s=[]
    def push(self,x):
        self.s.append(x)
    def pop(self):
        assert len(self.s)>0
        return self.s.pop() # pop de lista, no de Pila
    def is_empty(self):
        return len(self.s)==0
```

```
a=Pila()
a.push(10)
a.push(20)
print(a.pop())
a.push(30)
print(a.pop())
print(a.pop())
```

```
20
30
10
```

Esta implementación simple posiblemente sirve en la mayoría de los casos, pero si necesitamos poder garantizar su eficiencia, tenemos el problema que la implementación de las listas de Python está fuera de nuestro control, y no podemos garantizar, por ejemplo, que cada una de las operaciones tome tiempo constante.

Por ese motivo, es útil contar con implementaciones en que sí podamos dar ese tipo de garantía.

Implementación usando un arreglo

Utilizaremos un arreglo s , en donde los elementos de la pila se almacenarán en los casilleros $0, 1, \dots$, con el elemento del tope en el casillero ocupado de más a la derecha. Mantendremos una variable n para almacenar el número de elementos presentes en la pila, y el arreglo tendrá un tamaño máximo, el que se podrá especificar opcionalmente al momento de crear la pila.

Figure 41: pila-arreglo

```
import numpy as np
class Pila:
    def __init__(self, maxn=100):
        self.s=np.zeros(maxn)
        self.n=0
    def push(self,x):
        assert self.n<len(self.s)-1
        self.s[self.n]=x
        self.n+=1
    def pop(self):
        assert self.n>0
        self.n-=1
        return self.s[self.n]
    def is_empty(self):
        return self.n==0
```

```
a=Pila()
a.push(10)
a.push(20)
print(a.pop())
a.push(30)
print(a.pop())
print(a.pop())
```

```
20.0
30.0
10.0
```

Esta implementación es muy eficiente: no solo es evidente que cada operación toma tiempo constante, sino además esa constante es muy pequeña. Sin embargo, tiene la limitación de que es necesario darle un tamaño máximo al arreglo, el cual a la larga puede resultar insuficiente.

Existe una manera de copiar todos los elementos a un arreglo más grande y seguir operando cuando el arreglo se llena. Si el nuevo arreglo es del doble del tamaño anterior, el costo de copiar todos los elementos se puede *amortizar* a lo largo de las operaciones, de modo que en *promedio* sea constante, pero se pierde la propiedad de que las operaciones tomen tiempo constante en el peor caso.

La siguiente es otra alternativa de implementación, que no sufre de ese problema.

Implementación usando una lista enlazada

En esta implementación los elementos de la pila se almacenan en una lista de enlace simple (sin cabecera), en que el elemento del tope de la pila es el primero de la lista.

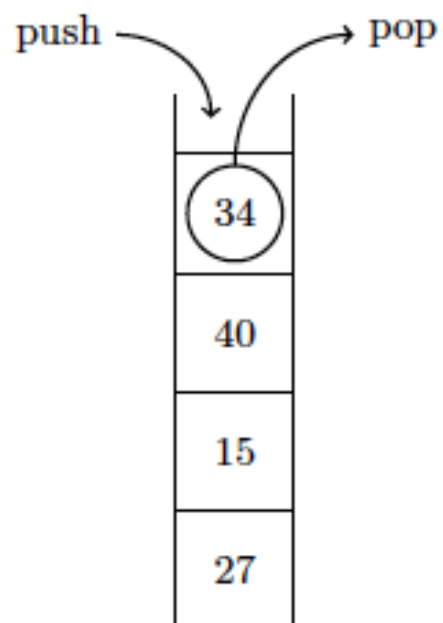


Figure 42: pila-lista



```
class NodoLista:
    def __init__(self, info, sgte=None):
        self.info=info
        self.sgte=sgte
class Pila:
    def __init__(self):
        self.tope=None
    def push(self, x):
        self.tope=NodoLista(x, self.tope)
    def pop(self):
        assert self.tope is not None
        x=self.tope.info
        self.tope=self.tope.sgte
        return x
    def is_empty(self):
        return self.tope is None
```

```
a=Pila()
a.push(10)
a.push(20)
```

```
print(a.pop())
a.push(30)
print(a.pop())
print(a.pop())
```

```
20
30
10
```

Aplicaciones de pilas

Evaluación de notación polaca Si se tiene una fórmula en notación polaca, se puede calcular su valor usando una pila, inicialmente vacía. Los símbolos de la fórmula se van leyendo de izquierda a derecha, y:

- si el símbolo es un número, se le hace push en la pila
- si el símbolo es un operador, se hacen dos pop, se efectúa la operación indicada entre los dos datos obtenidos, y el resultado se agrega de vuelta a la pila con push

Al terminar, si la fórmula estaba bien formada, debe haber solo un elemento en la pila, que es el resultado de la evaluación de la fórmula.

```
def eval_polaca(formula):
    a=Pila()
    for x in formula.split():
        if x.isnumeric():
            a.push(int(x))
        else: # tiene que ser un operador
            v=a.pop()
            u=a.pop()
            if x=="+":
                w=u+v
            elif x=="-":
                w=u-v
            elif x=="*":
                w=u*v
            elif x=="/":
                w=u/v
            else:
                print("Operador desconocido:",x)
                return 0
            a.push(w)
    return a.pop()
```

```
formula=input('Escriba la fórmula en notación polaca: ')
print("Resultado: ",eval_polaca(formula))
```

Escriba la fórmula en notación polaca: 2 3 + 9 4 2 / - *
Resultado: 35.0

Recorrido no recursivo de un árbol binario Supongamos que queremos recorrer un árbol binario en preorden. En lugar de utilizar un algoritmo recursivo, podemos imaginar que tenemos una “To DO list” en donde almacenamos la lista de nodos que debemos visitar en el futuro. Inicialmente, esta lista contiene solo la raíz. En cada iteración, extraemos un nodo de la lista, lo visitamos, y luego agregamos a la lista a sus dos hijos. Si la lista se mantiene como una pila, el orden del en que se visitan los nodos es exactamente preorden.

```
class Nodo:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

class Arbol:
    def __init__(self, raiz=None):
        self.raiz=raiz

    def preorden(self):
        print("Preorden no recursivo:", end=" ")
        s=Pila()
        s.push(self.raiz)
        while not s.is_empty():
            p=s.pop()
            if p is not None:
                print(p.info, end=" ")
                s.push(p.der)
                s.push(p.izq)
        print()
```

Es importante que las operaciones push se hagan en el orden indicado (derecho-izquierdo), para que de acuerdo a la disciplina LIFO, salga primero el izquierdo y luego el derecho.

```
a=Arbol(
    Nodo(
        Nodo(
```

```

        Nodo(None, 15, None),
        20,
        Nodo(
            Nodo(None, 30, None),
            35,
            None
        )
    ),
    42,
    Nodo(
        Nodo(
            Nodo(
                Nodo(None, 65, None),
                72,
                Nodo(None, 81, None)
            ),
            90,
            None
        ),
        95,
        None
    )
)
)
)

```

```
a.preorden()
```

Preorden no recursivo: 42 20 15 35 30 95 90 72 65 81

Hay una pequeña optimización que se puede hacer al algoritmo de recorrido no recursivo. Cuando hacemos las dos operaciones push y volvemos a ejecutar el while, sabemos que la pila no está vacía, de modo que esa pregunta es superflua. Además, al hacer el pop sabemos que lo que va a salir de la pila es lo último que se agregó, o sea, p.izq. Por lo tanto, podemos saltarnos tanto la pregunta como el pop e ir directamente al if, el cual por lo tanto se transforma en un while.

```

class Arbol:
    def __init__(self, raiz=None):
        self.raiz=raiz

    def preorden(self):
        print("Preorden no recursivo optimizado:", end=" ")

```

```

s=Pila()
s.push(self.raiz)
while not s.is_empty():
    p=s.pop()
    while p is not None:
        print(p.info, end=" ")
        s.push(p.der)
        p=p.izq
    print()

```

```

a=Arbol(
    Nodo(
        Nodo(
            Nodo(None,15,None),
            20,
            Nodo(
                Nodo(None,30,None),
                35,
                None
            )
        ),
        42,
        Nodo(
            Nodo(
                Nodo(
                    Nodo(None,65,None),
                    72,
                    Nodo(None,81,None)
                ),
                90,
                None
            ),
            95,
            None
        )
    )
)

```

```
a.preorden()
```

Preorden no recursivo optimizado: 42 20 15 35 30 95 90 72 65 81

Colas ("Queues")

Una cola es una lista en que los elementos ingresan por un extremo y salen por el otro. Debido a que los elementos van saliendo en orden de llegada, una cola también se llama "lista FIFO," por "First-In-First-Out."

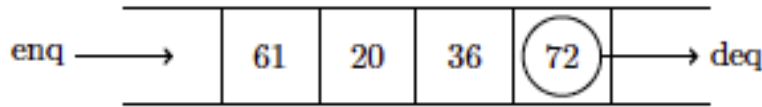


Figure 43: cola

Las dos operaciones básicas son **enq** (encolar), que agrega un elemento al final de todos, y **deq** (desencolar), que extrae el elemento que encabeza la cola. Más precisamente, si *q* es un objeto de tipo Cola, están disponibles las siguientes operaciones:

- *q*.enq(*x*): encola *x* al final de la cola *q*
- *x*=*q*.deq(): extrae y retorna el elemento a la cabeza de la cola *q*
- *b*=*q*.is_empty(): retorna verdadero si la cola *q* está vacía, falso si no

Implementación usando listas de Python

Tal como hicimos en el caso de las pilas, es muy simple implementar colas usando las listas de Python, pero no tenemos mucho control sobre la eficiencia del resultado:

```
class Cola:
    def __init__(self):
        self.q=[]
    def enq(self,x):
        self.q.insert(0,x)
    def deq(self):
        assert len(self.q)>0
        return self.q.pop()
    def is_empty(self):
        return len(self.q)==0
```

```
a=Cola()
a.enq(72)
a.enq(36)
print(a.deq())
```



```

a.enq(20)
print(a.deq())
print(a.deq())
a.enq(61)
print(a.deq())

```

```

72
36
20
61

```

Implementación usando un arreglo

De manera análoga a lo que hicimos en el caso de la pila, podemos almacenar los n elementos de la cola usando posiciones contiguas en un arreglo, por ejemplo, las n primeras posiciones. Pero hay un problema: como la cola crece por un extremo y se achica por el otro, ese grupo de posiciones contiguas se va desplazando dentro del arreglo, y después de un rato choca contra el otro extremo. La solución es ver al arreglo como *circular*, esto es, que si el arreglo tiene tamaño $maxn$, a continuación de la posición $maxn - 1$ viene la posición 0. Esto se puede hacer fácilmente usando aritmética módulo $maxn$.

Para la implementación, utilizaremos un subíndice *cabeza* que apunta al primer elemento de la cola, y una variable n que indica cuántos elementos hay en la cola. La siguiente figura muestra dos situaciones en que podría encontrarse el arreglo:

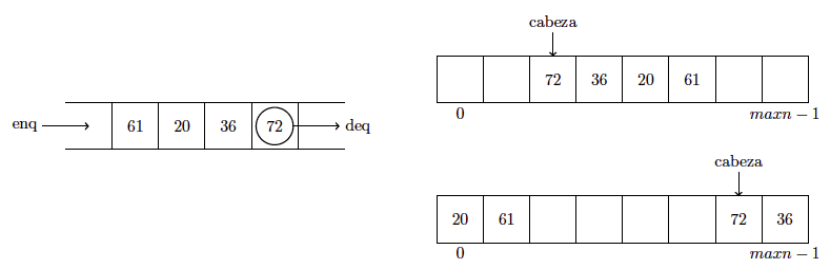


Figure 44: cola-arreglo

```

import numpy as np
class Cola:
    def __init__(self, maxn=100):
        self.q=np.zeros(maxn)
        self.n=0
        self.cabeza=0
    def enq(self, x):

```

```

    assert self.n < len(self.q) - 1
    self.q[(self.cabeza + self.n) % len(self.q)] = x
    self.n += 1
def deq(self):
    assert self.n > 0
    x = self.q[self.cabeza]
    self.cabeza = (self.cabeza + 1) % len(self.q)
    self.n -= 1
    return x
def is_empty(self):
    return self.n == 0

```

```

a = Cola(3) # para forzar circularidad
a.enq(72)
a.enq(36)
print(a.deq())
a.enq(20)
print(a.deq())
print(a.deq())
a.enq(61)
print(a.deq())

```

72.0

36.0

20.0

61.0

Implementación usando una lista enlazada

El operar en los dos extremos de la cola sugiere de inmediato el uso de una lista de doble enlace, y esa es una opción posible. Pero, como veremos, se puede implementar una cola con una lista de enlace simple:

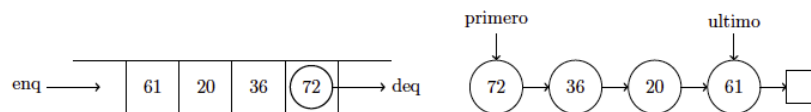


Figure 45: cola-lista

Una cosa que complica un poco la programación es que el invariante que se ve a la derecha se cumple solo si la cola es no vacía. Para una cola vacía, los dos punteros (primero y último) son nulos. Por lo tanto, un enq sobre una cola vacía, y un deq que deja una cola vacía serán casos especiales.

```

class NodoLista:
    def __init__(self, info, sgte=None):
        self.info=info
        self.sgte=sgte
class Cola:
    def __init__(self):
        self.primeros=None
        self.ultimo=None
    def enq(self,x):
        p=NodoLista(x)
        if self.ultimo is not None: # cola no vacía, agregamos al final
            self.ultimo.sgte=p
            self.ultimo=p
        else: # la cola estaba vacía
            self.primeros=p
            self.ultimo=p
    def deq(self):
        assert self.primeros is not None
        x=self.primeros.info
        if self.primeros is not self.ultimo: # hay más de 1 elemento
            self.primeros=self.primeros.sgte
        else: # hay solo 1 elemento, el deq deja la cola vacía
            self.primeros=None
            self.ultimo=None
        return x
    def is_empty(self):
        return self.primeros is None

```

```

a=Cola()
a.enq(72)
a.enq(36)
print(a.deq())
a.enq(20)
print(a.deq())
print(a.deq())
a.enq(61)
print(a.deq())

```

```

72
36
20
61

```

Aplicaciones de colas

Las colas se utilizan en los sistemas operativos siempre que hay algún recurso que no puede ser compartido. Uno de los procesos que lo requieren tiene acceso al recurso, mientras los demás deben esperar en una cola. Un ejemplo de esto son los sistemas de “spooling” para las impresoras.

También se usan mucho en sistemas de simulación, cuando se deben modelar situaciones del mundo real en que hay colas. Por ejemplo, la caja en un supermercado.

A continuación veremos una aplicación análoga a la que vimos en el caso de pilas para el recorrido de un árbol binario.

Recorrido de un árbol binario por niveles Supongamos que se desea recorrer un árbol binario, visitando sus nodos en orden de su distancia a la raíz. No hay manera de escribir esto de manera recursiva, pero el problema se puede resolver usando el mismo enfoque que utilizamos al recorrer un árbol binario en preorden de manera no recursiva, pero usando una cola en lugar de una pila.

```
class Nodo:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

class Arbol:
    def __init__(self, raiz=None):
        self.raiz=raiz

    def niveles(self):
        print("Recorrido por niveles:", end=" ")
        c=Cola()
        c.enq(self.raiz)
        while not c.is_empty():
            p=c.deq()
            if p is not None:
                print(p.info, end=" ")
                c.enq(p.izq)
                c.enq(p.der)
        print()
```

```
a=Arbol(
    Nodo(
        Nodo(
            Nodo(None, 15, None),
```

```

    20,
    Nodo(
        Nodo(None, 30, None),
        35,
        None
    )
),
42,
Nodo(
    Nodo(
        Nodo(
            Nodo(None, 65, None),
            72,
            Nodo(None, 81, None)
        ),
        90,
        None
    ),
    95,
    None
)
)
)

```

```
a.niveles()
```

Recorrido por niveles: 42 20 95 15 35 90 30 72 65 81

Colas de Prioridad

Hay muchas situaciones en que los elementos que esperan en una cola deben ir siendo atendidos no por orden de llegada, sino de acuerdo a algún criterio de *prioridad*. En esos casos, no nos sirve la cola como la hemos visto, sino que se necesita un nuevo tipo de estructura.

Una cola de prioridad es un TDA que consiste de un conjunto de datos que poseen un atributo (llamado su *prioridad*) perteneciente a algún conjunto ordenado, y en el cual se pueden ejecutar dos operaciones básicas: **insertar** un nuevo elemento con una prioridad cualquiera y **extraer** el elemento de máxima prioridad.

Más específicamente, las operaciones permitidas son:

- `q.insert(x)`: inserta un elemento de prioridad `x` en la cola de prioridad `q`

- `x=q.extract_max()`: extrae y retorna el elemento de máxima prioridad de la cola de prioridad `q`
- `b=q.is_empty()`: retorna verdadero si la cola de prioridad `q` está vacía, falso si no

Definir qué significa tener “máxima prioridad” depende de la aplicación que queramos darle a la cola de prioridad. En los ejemplos de este capítulo, supondremos que mejor prioridad corresponde a un mayor valor numérico, pero por simetría también sería válido el criterio opuesto.

Implementación usando una lista desordenada

La implementación más simple consiste en mantener el conjunto como una lista desordenada. Agregar un elemento es trivial, pero encontrar el máximo requiere tiempo $\Theta(n)$.

La lista se puede mantener ya sea en un arreglo o en una lista enlazada. Para nuestro ejemplo, utilizaremos una lista enlazada con cabecera.

```
class NodoLista:
    def __init__(self, info, sgte=None):
        self.info=info
        self.sgte=sgte
class Cola_de_prioridad:
    def __init__(self):
        self.cabecera=NodoLista(0)
    def insert(self, x):
        self.cabecera.sgte=NodoLista(x, self.cabecera.sgte)
    def extract_max(self):
        assert self.cabecera.sgte is not None
        p=self.cabecera # apunta al previo del candidato a máximo
        r=self.cabecera.sgte
        while r.sgte is not None:
            if r.sgte.info>p.sgte.info:
                p=r
                r=r.sgte
        x=p.sgte.info # anotamos el valor máximo
        p.sgte=p.sgte.sgte # eliminamos el nodo con el máximo
        return x
    def is_empty(self):
        return self.cabecera.sgte is None
```

```
a=Cola_de_prioridad()
a.insert(45)
a.insert(12)
```

```

a.insert(30)
print("max=", a.extract_max())
a.insert(20)
print("max=", a.extract_max())

```

max= 45

max= 30

Implementación usando una lista ordenada

Una implementación un poco más compleja consiste en mantener el conjunto como una lista ordenada. Agregar un elemento requiere recorrer la lista para encontrar el punto de inserción (tiempo $\Theta(n)$ en el peor caso), pero encontrar el máximo y extraerlo toma tiempo constante si la lista se ordena de mayor a menor.

```

class NodoLista:
    def __init__(self, info, sgte=None):
        self.info=info
        self.sgte=sgte
class Cola_de_prioridad:
    def __init__(self):
        self.cabecera=NodoLista(0)
    def insert(self, x):
        p=self.cabecera # al final apuntará al previo del punto de inserción
        while p.sgte is not None and x<p.sgte.info:
            p=p.sgte
        p.sgte=NodoLista(x, p.sgte)
    def extract_max(self):
        assert self.cabecera.sgte is not None
        x=self.cabecera.sgte.info # anotamos el valor máximo que está en el primer nodo
        self.cabecera.sgte=self.cabecera.sgte.sgte # eliminamos el primer nodo
        return x
    def is_empty(self):
        return self.cabecera.sgte is None

```

```

a=Cola_de_prioridad()
a.insert(45)
a.insert(12)
a.insert(30)
print("max=", a.extract_max())
a.insert(20)
print("max=", a.extract_max())

```

max= 45

max= 30

Las dos implementaciones que hemos visto están en extremos opuestos desde el punto de vista de su eficiencia. En ambas, una operación toma tiempo constante ($\Theta(1)$) mientras la otra toma tiempo lineal ($\Theta(n)$).

Veremos a continuación que es posible diseñar una estructura que equilibre de mejor forma el costo de las dos operaciones.

Implementación usando un *Heap*

Un heap es un árbol binario de una forma especial, que permite su almacenamiento sin usar punteros.

Este árbol se caracteriza porque tiene todos sus niveles llenos, excepto posiblemente el último, y en ese último nivel, los nodos están lo más a la izquierda posible.

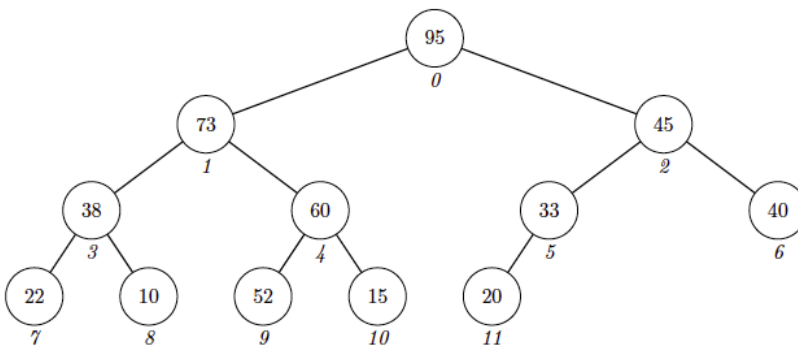


Figure 46: heap-árbol

Un árbol que cumpla esta condición diremos que tiene “forma de heap.”

Los números bajo cada nodo corresponde a una numeración por niveles, y esa numeración se utiliza para almacenar cada elemento en el casillero respectivo de un arreglo:

95	73	45	38	60	33	40	22	10	52	15	20
0	1	2	3	4	5	6	7	8	9	10	11

Figure 47: heap-arreglo

Este arreglo contiene toda la información necesaria para representar al árbol. En efecto, tenemos que la raíz está en el casillero 0, y además

$$\begin{aligned} \text{hijos del nodo } j &= \{2j + 1, 2j + 2\} \\ \text{padre del nodo } k &= \left\lfloor \frac{k - 1}{2} \right\rfloor \end{aligned}$$

Si hay n casilleros ocupados en el arreglo, cualquier subíndice que

sea mayor o igual a n corresponde a un nodo inexistente.

Un heap puede utilizarse para implementar una cola de prioridad almacenando los datos de modo que las llaves estén siempre ordenadas de arriba a abajo (a diferencia de un árbol de búsqueda binaria, que ordena sus llaves de izquierda a derecha). En otras palabras, el padre debe tener siempre mejor prioridad que sus hijos. Un árbol que cumple esta condición diremos que tiene “orden de heap,” y también se dice que es un “árbol de prioridad.”

Por lo tanto, un heap debe satisfacer un invariante consistente en dos condiciones:

- Condición estructural: el árbol debe tener “forma de heap”
- Condición de orden: el árbol debe tener “orden de heap”

Inserción La inserción se realiza agregando el nuevo elemento en la primera posición libre del heap, esto es, el próximo nodo que debería aparecer en el recorrido por niveles o, equivalentemente, un casillero que se agrega al final del arreglo.

Después de agregar este elemento, la condición estructural se cumple, pero la condición de orden no tiene por qué cumplirse. Para resolver este problema, si el nuevo elemento es mayor que su padre, se intercambia con él, y ese proceso se repite mientras sea necesario. Una forma de describir esto es diciendo que el nuevo elemento “trepa” en el árbol hasta alcanzar el nivel correcto según su prioridad.

Figure 48: heap-ins

Como la altura del árbol es $\log_2 n$ y en cada nivel se hace un trabajo constante, el tiempo que demora esta operación en el peor caso es $\Theta(\log n)$.

Extracción del máximo El máximo evidentemente está en la raíz del árbol (casillero o del arreglo). Al sacarlo de ahí, podemos imaginar que ese lugar queda vacante. Para llenarlo, tomamos al último elemento del heap y lo trasladamos al lugar vacante. En caso de que no esté bien ahí de acuerdo a su prioridad (¡que es lo más probable!), lo hacemos descender intercambiándolo siempre con el mayor de sus hijos. Decimos que este elemento “se hunde” hasta su nivel de prioridad.

Figure 49: heap-extract

Esta operación también demora un tiempo proporcional a la altura del árbol en el peor caso, esto es, $\Theta(\log n)$.

```

import numpy as np
def trepar(a,j): # El elemento a[j] trepa hasta su nivel de prioridad
    while j>=1 and a[j]>a[(j-1)//2]:
        (a[j],a[(j-1)//2])=(a[(j-1)//2],a[j]) # intercambiamos con el padre
        j=(j-1)//2 # subimos al lugar del padre

def hundir(a,j,n): # El elemento a[j] se hunde hasta su nivel de prioridad
    while 2*j+1<n: # mientras tenga al menos 1 hijo
        k=2*j+1 # el hijo izquierdo
        if k+1<n and a[k+1]>a[k]: # el hijo derecho existe y es mayor
            k+=1
        if a[j]>=a[k]: # tiene mejor prioridad que ambos hijos
            break
        (a[j],a[k])=(a[k],a[j]) # se intercambia con el mayor de los hijos
        j=k # bajamos al lugar del mayor de los hijos

class Heap:
    def __init__(self,maxn=100):
        self.a=np.zeros(maxn)
        self.n=0
    def insert(self,x):
        assert self.n<len(self.a)
        self.a[self.n]=x
        trepar(self.a,self.n)
        self.n+=1
    def extract_max(self):
        assert self.n>0
        x=self.a[0] # esta variable lleva el máximo, el casillero 0 queda vacante
        self.n-=1 # achicamos el heap
        self.a[0]=self.a[self.n] # movemos el elemento sobrante hacia el casillero vacante
        hundir(self.a,0,self.n)
        return x
    def imprimir(self):
        print(self.a[0:self.n])

```

Para poder visualizar el efecto de cada operación, agregamos una operación imprimir que muestra el contenido del arreglo.

```

a=Heap(10)
a.insert(45)
a.imprimir()
a.insert(12)
a.imprimir()
a.insert(30)

```

```

a.imprimir()
print("max=",a.extract_max())
a.imprimir()
a.insert(20)
a.imprimir()
print("max=",a.extract_max())
a.imprimir()

```

```

[45.]
[45. 12.]
[45. 12. 30.]
max= 45.0
[30. 12.]
[30. 12. 20.]
max= 30.0
[20. 12.]

```

Ejercicio 5.1

Agregar a la clase Heap un método `modificar(k,x)` que al ser invocado, cambie la prioridad del elemento del casillero k , dándole como nuevo valor x y asegurando que el heap siga cumpliendo las restricciones de orden. Esta operación debe funcionar en tiempo $O(\log n)$ en el peor caso. Escriba a continuación la definición de la clase Heap incluyendo esta nueva operación, y pruébela con las instrucciones que aparecen en el casillero siguiente.

```

a=Heap(20)
a.insert(55)
a.insert(50)
a.insert(70)
a.insert(12)
a.insert(36)
a.insert(10)
a.insert(21)
a.insert(24)
a.insert(20)
a.insert(62)
a.imprimir()
a.modificar(4,65)

```

```
a.imprimir()
a.modificar(3,15)
a.imprimir()
```

```
[70. 62. 55. 24. 50. 10. 21. 12. 20. 36.]
```

Ordenando con una cola de prioridad

Las colas de prioridad tienen múltiples aplicaciones, algunas de las cuales veremos más adelante en este curso.

Una de las aplicaciones más importantes es para resolver el problema de la ordenación. Dada cualquier implementación de una cola de prioridad, se la puede utilizar para construir un algoritmo de ordenación, de la siguiente manera:

- Crear una cola de prioridad vacía e insertar en ella todos los elementos del conjunto a ordenar
- Luego ir extrayendo máximos sucesivamente. Los elementos irán saliendo de mayor a menor.

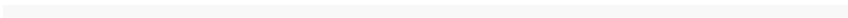
Para las dos primeras implementaciones de colas de prioridad que vimos (conjunto desordenado y conjunto ordenado) el algoritmo resultante demora tiempo $\Theta(n^2)$ en el peor caso (y corresponde a los algoritmos de ordenación por selección y ordenación por inserción, respectivamente).

En cambio, si se utiliza un heap, el algoritmo resultante demora tiempo $\Theta(n \log n)$ en el peor caso y se llama *Heapsort*. A continuación veremos una versión de Heapsort construida de acuerdo a estas ideas, y más adelante en el curso volveremos sobre el tema, porque es posible optimizar aspectos importantes del algoritmo.

```
def Heapsort(a): # Versión preliminar
    n=len(a)
    h=Heap(n)
    # Fase 1: insertamos los elementos en un heap
    for k in range(0,n):
        h.insert(a[k])
    # Fase 2: extraemos el máximo sucesivamente
    for k in range(n-1,-1,-1):
        a[k]=h.extract_max()
```

```
a = np.random.random(6)
print(a)
Heapsort(a)
print(a)
```

```
[0.84809569 0.60699999 0.8400833 0.76749145 0.18499666 0.09499075]  
[0.09499075 0.18499666 0.60699999 0.76749145 0.8400833 0.84809569]
```



6

6 Diccionarios

El TDA Diccionario es uno de los más usados en la práctica, y se conocen muchas formas distintas de implementarlo.

Un Diccionario es un conjunto de n elementos, cada uno de los cuales tiene un campo que permite identificarlo de manera única (ese campo se llama su *llave primaria*), sobre el cual están definidas las operaciones de buscar, insertar, eliminar, y ocasionalmente otras que definiremos más adelante. Más precisamente, si d es un diccionario, existirán las operaciones:

- $r=d.search(x)$: buscar el elemento de llave x , retornar un resultado que permita ubicarlo, o `None` si no está
- $d.insert(x)$: insertar un elemento de llave x , evitando crear una llave duplicada
- $d.delete(x)$: eliminar el elemento de llave x , el cual debe estar en el diccionario

Diccionarios de Python

El lenguaje Python posee un tipo `dict` que implementa la funcionalidad de diccionarios que hemos descrito (más operaciones adicionales). En un diccionario se busca por una llave y se obtiene un valor asociado.

```
distancia = {'Valparaíso':102, 'Concepción': 433, 'Arica': 1664, 'Puerto Montt': 912, 'Rancagua': 80}
```

La forma de buscar es simplemente usando la llave como sub-índice:

```
print(distancia['Arica'])
```

1664

Y la forma de agregar una nueva llave es asignándole un valor:

```
distancia['Talca']=237
```

Al buscar una llave inexistente se produce una excepción:

```
print(distancia['La Serena'])
```

```
-----
KeyError                                Traceback (most recent call last)
```

```
<ipython-input-14-0078e5a2585d> in <module>
----> 1 print(distancia['La Serena'])
```

```
KeyError: 'La Serena'
```

Pero hay una forma de buscar sin que dé un error, sino que retorne None:

```
print(distancia.get('Rancagua'), distancia.get('La Serena'))
```

```
80 None
```

Para eliminar un dato, se usa pop (lo elimina y retorna su valor):

```
distancia.pop('Rancagua')
```

```
80
```

Aparte de esto, hay muchas otras operaciones que permiten obtener la lista de todas las llaves, etc.

Dado que en Python ya existe una implementación de diccionarios, ¿por qué querríamos estudiar nosotros cómo implementarlos?

La respuesta está en que, si nosotros controlamos todos los detalles de una implementación, sabremos exactamente cuan eficiente es, y para qué tipo de aplicaciones es más apropiada. Lo último es particularmente importante, porque no hay ninguna implementación de diccionarios que sea uniformemente mejor que las otras para todas las aplicaciones.

Estudiaremos entonces cómo se puede implementar un diccionario, comenzando por las estrategias más sencillas, y avanzando hacia enfoques más sofisticados.

En nuestros ejemplos supondremos que solo almacenamos la llave, pero en la práctica siempre habrá información adicional asociada a cada llave. También por simplicidad a menudo usaremos llaves numéricas, aunque en la práctica es más frecuente que las llaves sean strings.

Búsqueda secuencial

La manera más simple de implementar un diccionario es con una lista desordenada de llaves, en la cual se hace búsqueda secuencial. La inserción es especialmente eficiente si obviamos chequear por duplicados, y la eliminación es eficiente una vez que sabemos dónde está la llave.

```
import numpy as np
```

```
class Lista_secuencial:
    def __init__(self, size=100):
        self.a=np.zeros(size,dtype=int)
        self.n=0
    def insert(self,x):
        assert self.n<len(self.a)
        self.a[self.n]=x
        self.n+=1
    def search(self,x):
        for k in range(0,self.n):
            if self.a[k]==x:
                return k
        return None
    def delete(self,x):
        k=self.search(x)
        self.a[k]=self.a[self.n-1] # movemos el último al lugar vacante
        self.n-=1
```

```
d=Lista_secuencial()
d.insert(30)
d.insert(10)
d.insert(25)
print(d.search(10))
print(d.search(80))
d.delete(30)
print(d.search(30))
```

```
1
None
None
```

La búsqueda secuencial también se puede implementar con una lista enlazada, en cuyo caso será más simple insertar al inicio.

En cualquier caso, la búsqueda demora tiempo $\Theta(n)$. Para estimar el costo promedio, suponemos que todos los elementos son

igualmente probables de ser accedidos y que el costo de buscar a un elemento que es el k -ésimo de la lista es k . Por lo tanto, el costo promedio es

$$\frac{1}{n} \sum_{1 \leq k \leq n} k = \frac{n+1}{2} = \Theta(n)$$

Por lo tanto, este tipo de implementación solo será adecuada para conjuntos muy pequeños.

Búsqueda secuencial con probabilidades de acceso no uniformes

En la práctica, es muy raro que las probabilidades de acceso a los elementos sean uniformes. Con frecuencia hay algunos elementos que son mucho más populares que otros, y empíricamente a menudo se observan distribuciones de tipo “ley de potencias,” con probabilidades de tipo

$$p_k \propto \frac{1}{k^\alpha}$$

para algún α . Para el caso $\alpha = 1$ esto se llama Ley de Zipf.

Si un conjunto de datos tiene elementos con probabilidades de acceso diferentes, entonces para la búsqueda secuencial el orden en que estén los elementos en la lista hace una diferencia.

Caso 1. Probabilidades conocidas

Si las probabilidades de acceso son conocidas, es fácil ver que el orden óptimo es en orden decreciente de probabilidad.

Más precisamente, si los elementos son X_1, X_2, \dots, X_n con probabilidades de acceso p_1, p_2, \dots, p_n respectivamente, y si están ordenados de modo que $p_1 \geq p_2 \geq p_3 \geq \dots$, entonces el costo esperado de búsqueda óptimo es

$$C_{OPT} = \sum_{1 \leq k \leq n} k p_k$$

Tomemos como ejemplo el capítulo 1 de “El Quijote” (en minúsculas y sin puntuación para simplificar su proceso), cuyo texto está en el archivo `cap1.txt`:

```
en un lugar de la mancha de cuyo nombre no quiero acordarme no ha mucho
tiempo que vivía un hidalgo de los de lanza en astillero adarga antigua
...
peregrino y significativo como todos los demás que a él y a sus cosas
había puesto
```

Acceso al archivo desde Colab Si este notebook está usándose en Google Colab, el archivo se debe almacenar en la carpeta Colab

Notebooks de Google Drive, y para que el código en Python pueda tener acceso a él se debe quitar los comentarios y ejecutar la siguiente celda:

```
#from google.colab import drive
#drive.mount("/content/gdrive")
#%cd "/content/gdrive/My Drive/Colab Notebooks/"
```

El costo óptimo para un archivo dado se puede obtener con el siguiente código en Python, en el cual hacemos uso de los diccionarios provistos por el lenguaje:

```
def calcula_costo_optimo(archivo): # lee el archivo, calcula frecuencias en orden descendente
    f=open(archivo,"r")
    texto=f.read()
    palabras=texto.split()
    frec={}
    for x in palabras:
        frec[x] = 1 if not x in frec else frec[x]+1
    lista=[]
    Copt=0
    costo=0
    for x in sorted(frec,key=frec.get,reverse=True):
        costo+=1
        Copt+=costo*frec[x]
        lista.append((frec[x],x))
    Copt/=len(palabras)
    f.close()
    return (Copt,lista)
```

Como resultado, mostramos el costo esperado de búsqueda en una lista ordenada de manera óptima (C_OPT) y las palabras más frecuentes:

```
(c,L)=calcula_costo_optimo("cap1.txt")
print('C_OPT={:6.2f}\n'.format(c))
for k in range(0,9):
    print(L[k])
```

C_OPT=157.81

```
(120, 'de')
(105, 'y')
(88, 'que')
(44, 'a')
```

```
(40, 'el')
(38, 'en')
(35, 'su')
(33, 'la')
(30, 'se')
```

Caso 2: Probabilidades desconocidas

Cuando las probabilidades son desconocidas, existen estrategias adaptativas, que van reordenando la lista dinámicamente a medida que los elementos son buscados, de modo de tratar de aproximar el orden óptimo. Hay dos técnicas que dan buenos resultados: “transpose” (TR) y “move to front” (MTF).

Transpose

Esta técnica consiste en que cada vez que un elemento es accedido, se le mueve un lugar más adelante en la lista (a menos que ya esté en el primer lugar). Esto se puede implementar ya sea en un arreglo o en una lista enlazada. En la siguiente implementación usaremos una lista enlazada con cabecera.

Si un elemento no se encuentra, simulamos como si hubiese estado al final de la lista. Para esto, mantendremos siempre disponible un nodo extra al final de la lista, en donde almacenaremos tentativamente la llave de búsqueda. Si finalmente se encuentra en ese nodo, se le incorpora a la lista y se crea un nuevo nodo extra.

Para contabilizar el costo, el método search retorna el número de comparaciones de llaves que se hizo en la búsqueda.

```
class NodoLista:
    def __init__(self, info, sgte=None):
        self.info=info
        self.sgte=sgte

class Lista_TR:
    def __init__(self):
        self.extra=NodoLista(0)
        self.cabecera=NodoLista(0, self.extra)

    def search(self, x): # busca x (si no está lo inserta al final) y lo adelanta un lugar
                        # retorna el costo de búsqueda
        self.extra.info=x # agregamos x al final, en caso que no estuviera antes
        p=self.cabecera
        q=p.sgte

        # k cuenta el número de comparaciones de llaves
        if q.info==x: # x ya está primero en la lista, no hacemos nada
            k=1
```

```

else:
    # buscamos del segundo en adelante
    r=q.sgte
    k=2
    while r.info!=x:
        (p,q,r)=(q,r,r.sgte)
        k+=1
    # r apunta al elemento buscado, lo movemos un lugar hacia adelante
    (p.sgte,q.sgte,r.sgte)=(r,r.sgte,q)
    if q.sgte is None: # se utilizó el nodo extra, agregamos uno nuevo
        self.extra=NodoLista(0)
        q.sgte=self.extra
    return k

def imprimir(self):
    p=self.cabecera.sgte
    print("[",end=" ")
    while p is not self.extra:
        print(p.info,end=" ")
        p=p.sgte
    print("]")

```

```

def test(Lista_adaptativa): # test interactivo
    a=Lista_adaptativa()
    while True:
        x=input("x=")
        if x=="fin":
            return
        print("costo=",a.search(x),end=" ")
        a.imprimir()

```

```
test(Lista_TR)
```

```

x=hola
costo= 1 [ hola ]
x=chao
costo= 2 [ chao hola ]
x=casa
costo= 3 [ chao casa hola ]
x=hola
costo= 3 [ chao hola casa ]
x=hola
costo= 2 [ hola chao casa ]
x=hola

```

```

costo= 1 [ hola chao casa ]
x=gato
costo= 4 [ hola chao gato casa ]
x=fin

```

```

def procesa(archivo,Lista_adaptativa): # lee el archivo y calcula costo promedio de búsqueda
    f=open(archivo,"r")
    texto=f.read()
    palabras=texto.split()
    npalabras=0
    costo_acum=0
    a=Lista_adaptativa()
    for x in palabras:
        costo_acum+=a.search(x)
        npalabras+=1
    print("Costo promedio de búsqueda= {:.2f}".format(costo_acum/npalabras))
    f.close()

```

```

procesa("cap1.txt",Lista_TR)

```

Costo promedio de búsqueda= 208.74

Move-To-Front

Esta técnica consiste en que cada vez que un elemento es accedido, se le mueve al primer lugar de la lista (a menos que ya esté en el primer lugar). Si un elemento no se encuentra, simulamos como si hubiese estado al final de la lista.

```

class Lista_MTF:
    def __init__(self):
        self.extra=NodoLista(0)
        self.cabecera=NodoLista(0,self.extra)

    def search(self,x): # busca x (si no está lo inserta al final) y luego lo mueve al primer lugar
        # retorna el costo de búsqueda
        self.extra.info=x # agregamos x al final, en caso que no estuviera antes
        p=self.cabecera
        q=p.sgte
        k=1 # cuenta el número de comparaciones de llaves
        while q.info!=x:
            (p,q)=(q,q.sgte)
            k+=1
        if q.sgte is None: # se utilizó el nodo extra, agregamos uno nuevo
            self.extra=NodoLista(0)

```

```

        q.sgte=self.extra
        if k>1: # x no está primero, move to front
            (self.cabecera.sgte,p.sgte,q.sgte)=(q,q.sgte,self.cabecera.sgte)
        return k

    def imprimir(self):
        p=self.cabecera.sgte
        print("[",end=" ")
        while p is not self.extra:
            print(p.info,end=" ")
            p=p.sgte
        print("]")

```

```
test(Lista_MTF)
```

```

x=hola
costo= 1 [ hola ]
x=casa
costo= 2 [ casa hola ]
x=chao
costo= 3 [ chao casa hola ]
x=hola
costo= 3 [ hola chao casa ]
x=casa
costo= 3 [ casa hola chao ]
x=casa
costo= 1 [ casa hola chao ]
x=gato
costo= 4 [ gato casa hola chao ]
x=fin

```

```
procesa("cap1.txt",Lista_MTF)
```

Costo promedio de búsqueda= 188.82

En resumen, tenemos que para este texto en particular, el costo óptimo es 157.81, el costo promedio de TR es 208.74 y el de MTF es 188.82.

Si en lugar de considerar un caso se analiza matemáticamente el caso general, suponiendo que los accesos llegan independientemente siguiendo la distribución dada y que el algoritmo corre durante un tiempo que tiende a infinito, se puede demostrar que

$$C_{OPT} \leq C_{TR} \leq C_{MTF} \leq \frac{\pi}{2} C_{OPT}$$

Búsqueda en un arreglo ordenado: Búsqueda Binaria

Ya hemos visto anteriormente que si los datos están en un arreglo ordenado, podemos hacer una búsqueda binaria, la que demora tiempo $\lceil \log_2(n+1) \rceil = \Theta(\log n)$ en el peor caso.

Esto es bastante eficiente, pero tiene el problema que agregar o eliminar datos del arreglo toma tiempo $\Theta(n)$ en el peor caso, por la necesidad de mantener el conjunto ordenado y compacto. Un objetivo que perseguiremos en el resto de este capítulo es tratar de encontrar estructuras de datos que nos permitan buscar de manera tan eficiente como la búsqueda binaria, junto con inserciones y eliminaciones igualmente eficientes.

Pero antes de avanzar en esa dirección, consideremos la pregunta de si es posible buscar más rápido que la búsqueda binaria en el peor caso.

Cota inferior para la búsqueda por comparaciones

Consideremos el problema de buscar una llave x en un conjunto de tamaño 4, digamos $\{a, b, c, d\}$, con $a < b < c < d$. La siguiente figura ilustra una manera como podría hacerse esa búsqueda:

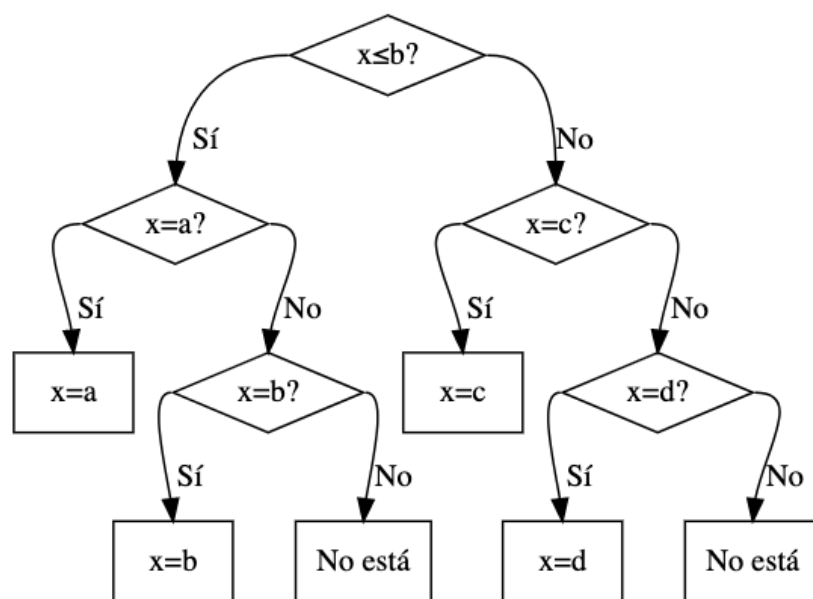


Figure 50: decision-tree

Este tipo de figura se llama un *árbol de decisión*, y en él los rombos representan preguntas y los rectángulos, las salidas (outputs) del algoritmo.

Este árbol de decisión es uno entre la infinidad de árboles que podrían resolver el problema de la búsqueda. Lo importante que hay que observar es que todo algoritmo que funcione mediante

comparaciones binarias (comparaciones con salidas “Sí/No”) se puede representar por un árbol de decisión.

En este tipo de árbol tenemos que:

- La altura representa el número de comparaciones que hace el algoritmo en el peor caso, y
- El número de hojas (cajas rectangulares) debe ser mayor o igual al número de respuestas posibles que debe ser capaz de emitir el algoritmo.

Recordemos que si N es el número de hojas y h la altura, siempre se tiene $N \leq 2^h$, de donde se deduce que $h \geq \lceil \log_2 N \rceil$ (porque la altura es un número entero), y en consecuencia, tenemos que

$$\text{Peor caso} \geq \lceil \log_2 (\text{número de respuestas distintas}) \rceil$$

Para el caso de la búsqueda binaria, tenemos que $N = n + 1$, porque el algoritmo de búsqueda debe poder identificar a cada uno de los n elementos, más la respuesta negativa cuando el elemento buscado no está. En consecuencia:

Todo algoritmo que busque en un conjunto de tamaño n mediante comparaciones binarias debe hacer al menos $\lceil \log_2 (n + 1) \rceil$ comparaciones en el peor caso.

Por lo tanto, la búsqueda binaria es óptima.

Árboles de Búsqueda Binaria (ABBs)

Un *árbol de búsqueda binaria* (ABB) es un árbol binario en que todos sus nodos internos cumplen la siguiente propiedad: Si la llave almacenada en el nodo es x , entonces todas las llaves en su subárbol izquierdo son menores que x , y las llaves en el subárbol derecho son mayores que x .

Los ABBs permiten realizar de manera eficiente (en promedio) las operaciones de inserción y búsqueda.

Búsqueda en un ABB

La búsqueda es similar a una búsqueda binaria (de ahí el nombre de estos árboles). Para buscar una llave x se comienza en la raíz. Si x se encuentra ahí, la búsqueda termina exitosamente. Si no está ahí, se continúa buscando en el subárbol izquierdo si x es menor que la raíz, o en el subárbol derecho si x es mayor que la raíz. Si se llega a una hoja (nodo externo), la búsqueda concluye infructuosamente.

Inserción en un ABB

Para insertar una llave x en un ABB, se realiza una búsqueda, que debe ser infructuosa, y la hoja en donde termina la búsqueda se reemplaza por un nodo interno conteniendo la llave x , con dos nuevas hojas como hijos.

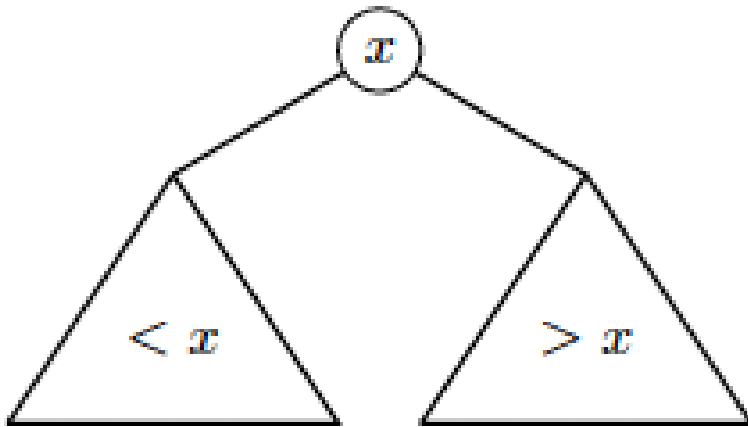


Figure 51: esquema-ABB2

Figure 52: insercioABB

Implementación recursiva

Los algoritmos de un ABB se prestan de manera natural a ser programados de manera recursiva, especialmente con la representación explícita de los nodos externos:

```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

    def search(self,x):
        if x==self.info:
            return self
        if x<self.info:
            return self.izq.search(x)
        else:
            return self.der.search(x)

    def insert(self,x):
        assert x!=self.info
        if x<self.info:
            return Nodoi(self.izq.insert(x),self.info,self.der)
        else:
            return Nodoi(self.izq,self.info,self.der.insert(x))
```

```

def __str__(self):
    return "("+self.izq.__str__()+str(self.info)+self.der.__str__()+")"

class Nodoe:
    def __init__(self):
        pass

    def search(self,x):
        return None

    def insert(self,x):
        return Nodoe(),x,Nodoe()

    def __str__(self):
        return ""

class Arbol:
    def __init__(self,raiz=Nodoe()):
        self.raiz=raiz

    def insert(self,x):
        self.raiz=self.raiz.insert(x)

    def search(self,x):
        return self.raiz.search(x)

    def __str__(self):
        return self.raiz.__str__()

```

Hemos incluido una función `__str__` para poder visualizar (de forma algo rudimentaria) el árbol construido.

```

a=Arbol()
a.insert(42)
a.insert(77)
a.insert(50)
a.insert(10)
print(a)

```

```
((10)42((50)77))
```

Para probar nuestra implementación, definiremos una función test:

```
def test(a,x):
    print(x, "está" if a.search(x) is not None else "no está")
```

```
test(a,50)
test(a,90)
```

```
50 está
90 no está
```

```
a.insert(90)
print(a)
test(a,90)
```

```
((10)42((50)77(90)))
90 está
```

Implementación no recursiva

Las operaciones de búsqueda e inserción en el árbol no necesitan programarse recursivamente, porque se pueden realizar en una sola pasada de arriba a abajo, sin necesidad de volver hacia arriba. En este caso, toda esa funcionalidad se implementa dentro de la clase Arbol:

```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

    def __str__(self):
        return "("+self.izq.__str__()+str(self.info)+self.der.__str__()+")"

class Nodoe:
    def __init__(self):
        pass

    def __str__(self):
        return ""

class Arbol:
    def __init__(self, raiz=Nodoe()):
        self.raiz=raiz

    def insert(self,x):
        if isinstance(self.raiz, Nodoe):
```

```

        self.raiz=Nodoi(Nodo(),x,Nodo())
        return
    p=self.raiz
    while True:
        assert x!=p.info
        if x<p.info:
            if isinstance(p.izq, Nodo):
                p.izq=Nodoi(Nodo(),x,Nodo())
                return
            p=p.izq
        else: # x>p.info
            if isinstance(p.der, Nodo):
                p.der=Nodoi(Nodo(),x,Nodo())
                return
            p=p.der

    def search(self,x):
        p=self.raiz
        while not isinstance(p, Nodo):
            if x==p.info:
                return p
            p=p.izq if x<p.info else p.der
        return None

    def __str__(self):
        return self.raiz.__str__()

```

```

a=Arbol()
a.insert(42)
a.insert(77)
a.insert(50)
a.insert(10)
print(a)

```

```
((10)42((50)77))
```

```

test(a,50)
test(a,90)

```

```

50 está
90 no está

```

```
a.insert(90)
print(a)
test(a,90)
```

```
((10)42((50)77(90)))
```

```
90 está
```

Costo de búsqueda en un ABB

Peor caso El peor caso para un árbol dado es la altura del árbol, y el peor árbol posible tiene altura n . Por lo tanto, el costo de búsqueda en el peor caso es $\Theta(n)$.

Lo anterior ocurre, por ejemplo, si los elementos se insertan en orden ascendente, o descendente, o en algún orden tipo “zig-zag.” Es muy improbable que esto ocurra por casualidad, pero sí puede suceder si los datos provienen de algún otro proceso en que resultan naturalmente ordenados, o bien porque la secuencia de inserciones es generada por un adversario malicioso que busca hacer que el algoritmo funcione de la manera más lenta posible.

Caso promedio Para analizar el costo esperado de búsqueda en un ABB con n nodos, supondremos que en una inserción y en una búsqueda infructuosa, los $n + 1$ nodos externos son igualmente probables como punto de destino de la búsqueda y de la inserción, y que en una búsqueda exitosa, los n nodos internos son igualmente probables como punto de destino de la búsqueda.

Recordemos que anteriormente definimos el *largo de caminos internos* (LCI) como $I_n = \sum$

(7.0)

Reemplazando $E_n = I_n + 2n$ en la fórmula para C_n , tenemos que

$$C_n = 1 + \frac{E_n - 2n}{n} = \frac{E_n}{n} - 1 = \frac{n+1}{n} C'_n - 1$$

Por lo tanto, tenemos la siguiente relación entre los costos esperados de búsqueda exitoso e infructuoso:

$$C_n = \left(1 + \frac{1}{n}\right) C'_n - 1$$

de modo que, cuando $n \rightarrow \infty$

$$C_n \approx C'_n - 1$$

Para poder completar este análisis necesitamos una segunda ecuación que vincule a estas incógnitas. Para esto, observemos que *el costo de buscar un elemento que acaba de ser insertado es exactamente 1 más que el costo de buscarlo infructuosamente antes de su inserción*. Por lo tanto, si consideramos y promediamos los costos de búsqueda de los elementos en orden de inserción, tenemos que

$$C_n = \frac{(1 + C'_0) + (1 + C'_1) + \cdots + (1 + C'_{n-1})}{n} = 1 + \frac{1}{n} \sum_{0 \leq k \leq n-1} C'_k$$

Multiplicando ambos lados por n , tenemos

$$nC_n = n + \sum_{0 \leq k \leq n-1} C'_k$$

Sustituyendo n por $n + 1$ en esta ecuación, tenemos

$$(n + 1)C_{n+1} = n + 1 + \sum_{0 \leq k \leq n} C'_k$$

Restando ambas ecuaciones, tenemos:

$$(n + 1)C_{n+1} - nC_n = 1 + C'_n$$

La relación que habíamos obtenido antes entre C_c y C'_n se puede reescribir como

$$nC_n = (n + 1)C'_n - n$$

Reemplazando en la ecuación anterior, obtenemos:

$$(n + 2)C'_{n+1} - (n + 1) - (n + 1)C'_n + n = 1 + C'_n$$

Lo que se simplifica a

$$(n + 2)(C'_{n+1} - C'_n) = 2$$

obteniéndose la ecuación de recurrencia

$$C'_{n+1} = C'_n + \frac{2}{n + 2}$$

$$C'_0 = 0$$

Esta ecuación se puede resolver “desenrollándola,” para obtener

$$C'_n = 2 \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n + 1} \right)$$

Si definimos los *números armónicos* $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$, podemos expresar la solución como

$$C'_n = 2(H_{n+1} - 1)$$

Los números armónicos son muy cercanos al logaritmo natural. Se puede demostrar que

$$H_n \leq 1 + \int_1^n \frac{1}{x} dx = 1 + \ln n$$

$$H_n \geq \int_1^{n+1} \frac{1}{x} dx = \ln(n+1)$$

de donde obtenemos que

$$\ln(n+1) \leq H_n \leq 1 + \ln n$$

Más precisamente, se puede demostrar que

$$H_n = \ln n + \gamma + O\left(\frac{1}{n}\right)$$

donde $\gamma \approx 0.577 \dots$ es la constante de Euler-Mascheroni.

```
%pylab inline
```

Populating the interactive namespace from numpy and matplotlib

```
H=np.zeros(100)
n=range(1,101)
H[0]=1;
for k in range(1,100):
    H[k]=H[k-1]+1/(k+1)
plt.plot(n,H,label='$H_n$')
plt.plot(n,log(n), label='$\ln\{n\}$')
leg=plt.legend(loc='best')
```

Por lo tanto, el costo esperado de búsqueda (exitosa o infructuosa) en un ABB es aproximadamente $2H_n \approx 2 \ln n \approx 1.386 \log_2 n$, lo cual es solo 38.6 por ciento peor que un árbol óptimo, cuyo costo de búsqueda es $\log_2 n$.

Se puede demostrar que la varianza también es logarítmica, por lo que en la práctica el tiempo de búsqueda debería ser cercano al promedio.

Sin embargo, este análisis supone que las inserciones llegan en orden aleatorio, lo cual no necesariamente se cumple en la vida real.

Eliminación en un ABB

La eliminación de una llave x es sencilla de efectuar en algunos casos, pero el caso complicado es cuando la llave tiene dos hijos:

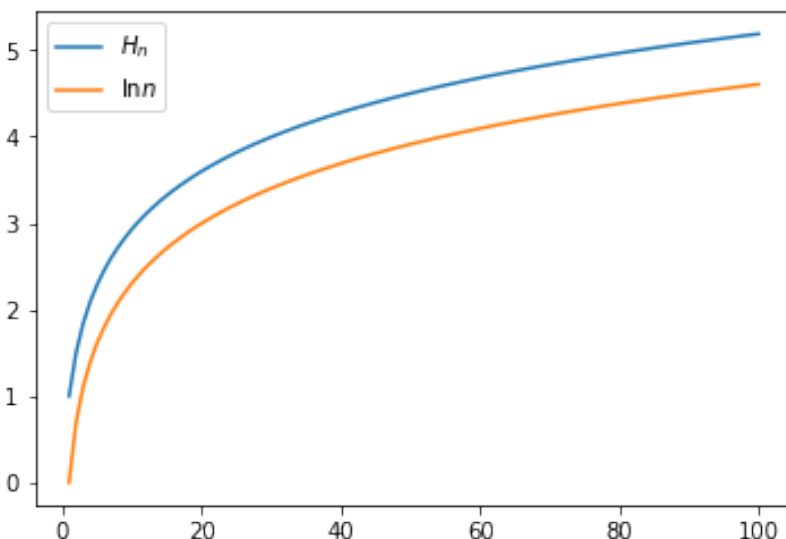


Figure 53: png

Figure 54: eliminacionABB1

Eliminación de una llave sin hijos En este caso, el nodo interno que contiene a x desaparece y en su lugar queda un nodo externo:

Eliminación de una llave con 1 hijo En este caso, el padre de la llave x pasa a apuntar al único hijo de x :

Figure 55: eliminacionABB2

Eliminación de una llave con 2 hijos Si x tiene 2 hijos, no podemos eliminarla directamente, pero sí podemos eliminar a la que la sigue en orden ascendente, digamos y . Se puede demostrar que y necesariamente es uno de los dos casos anteriores, de modo que es fácil de eliminar. Luego de concluido ese proceso, escribimos y en lugar de x en el campo info del nodo respectivo.

Por simetría, esto mismo podría haberse hecho con la llave que sigue a x en orden descendente.

El análisis del costo esperado de búsqueda que hicimos anteriormente es válido si solo hay inserciones. El análisis en el caso en que se incluyen eliminaciones es un problema matemáticamente muy complicado, y sigue siendo un problema abierto. La evidencia experimental indica que se obtienen mejores resultados si se alterna o si se aleatoriza al elegir entre sucesor o el predecesor de x en caso que

Figure 56: eliminacionABB3

haya que elegir.

Implementación recursiva de la eliminación

Por simplicidad, omitimos el código para la inserción y búsqueda, y en el caso de un nodo con 2 hijos, eliminamos siempre el sucesor. También ignoramos las eliminaciones de llaves que no están en el árbol.

```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

    def deletemin(self): # Elimina llave mínima del árbol, retorna (llave_min,raiz_arbol_restante)
        if isinstance(self.izq,Nodoe): # No hay hijo izquierdo
            return (self.info,self.der)
        # hay hijo izquierdo
        (llave_min,izq_sin_min)=self.izq.deletemin()
        return (llave_min,Nodoi(izq_sin_min,self.info,self.der))

    def delete(self,x):
        if x<self.info:
            return Nodoi(self.izq.delete(x),self.info,self.der)
        if x>self.info:
            return Nodoi(self.izq,self.info,self.der.delete(x))
        # x==self.info
        if isinstance(self.izq,Nodoe): # No hay hijo izquierdo
            return self.der
        if isinstance(self.der,Nodoe): # No hay hijo derecho
            return self.izq
        # Hay hijo izquierdo y derecho
        (y,der_sin_min)=self.der.deletemin()
        return(Nodoi(self.izq,y,der_sin_min))

    def __str__(self):
        return "("+self.izq.__str__()+str(self.info)+self.der.__str__()+")"

class Nodoe:
    def __init__(self):
        pass
```

```

def delete(self,x):
    return self

def __str__(self):
    return ""

class Arbol:
    def __init__(self,raiz=Nodo()):
        self.raiz=raiz

    def delete(self,x):
        self.raiz=self.raiz.delete(x)

    def __str__(self):
        return self.raiz.__str__()

```

Para probar este algoritmo utilizaremos el árbol que vimos en el capítulo 4:

```

a=Arbol(
    Nodoi(
        Nodoi(
            Nodoi(Nodo(),15,Nodo()),
            20,
            Nodoi(
                Nodoi(Nodo(),30,Nodo()),
                35,
                Nodo()
            )
        ),
        42,
        Nodoi(
            Nodoi(
                Nodoi(
                    Nodoi(Nodo(),65,Nodo()),
                    72,
                    Nodoi(Nodo(),81,Nodo())
                ),
                90,
                Nodo()
            ),
            95,
            Nodo()
        )
    )
)

```

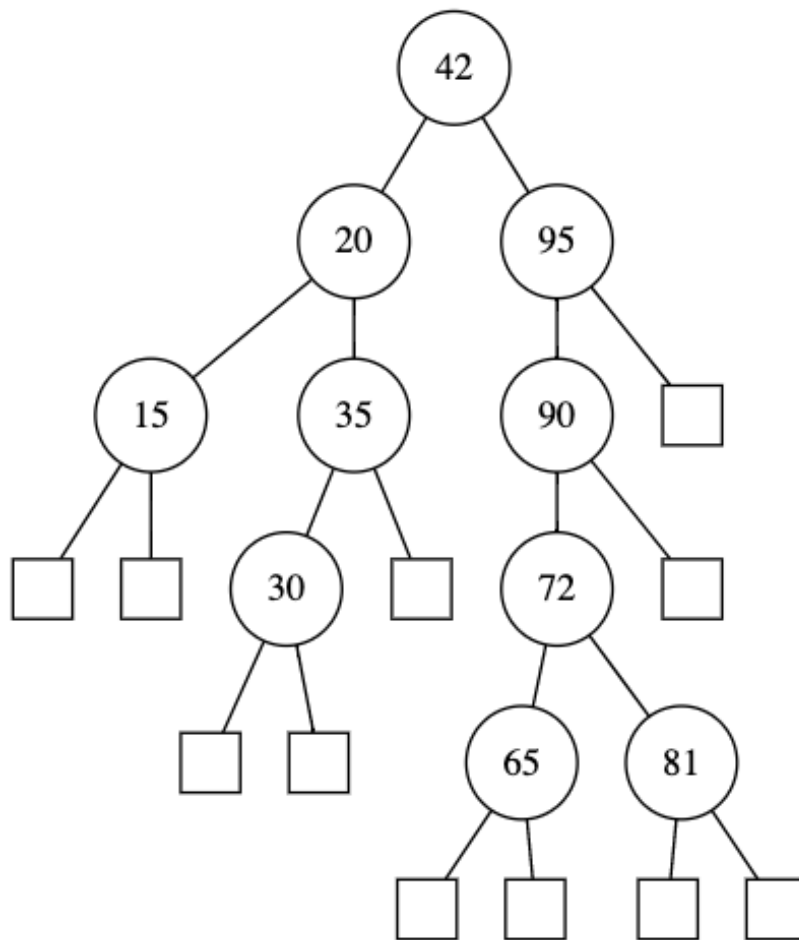


Figure 57: ejemplo-ABB

```
)
)
```

```
print(a)
```

```
(( (15)20((30)35))42(((65)72(81))90)95))
```

```
a.delete(30)
print(a)
```

```
(( (15)20(35))42(((65)72(81))90)95))
```

```
a.delete(95)
print(a)
```

```
(( (15)20(35))42(((65)72(81))90))
```

```
a.delete(42)
print(a)
```

```
(( (15)20(35))65((72(81))90))
```

```
a.delete(44) # 44 no está en el árbol
print(a)
```

```
(( (15)20(35))65((72(81))90))
```

```
a.delete(20)
print(a)
```

```
(( (15)35)65((72(81))90))
```

Implementación no recursiva de la eliminación

Al programar la eliminación de esta manera, necesitamos modificar el nodo padre de x . Para simplificar, asegurando que todo nodo tenga un padre, incluso la raíz, durante el proceso de eliminación simularemos que la raíz es hija derecha de un nodo con llave “ $-\infty$.”

```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der
```

```

def __str__(self):
    return "("+self.izq.__str__()+str(self.info)+self.der.__str__()+")"

class Nodoe:
    def __init__(self):
        pass

    def __str__(self):
        return ""

import math
class Arbol:
    def __init__(self, raiz=Nodoe()):
        self.raiz=raiz

    def delete(self, x):
        cabecera=Nodoe(None, -math.inf, self.raiz)
        p=cabecera # padre del candidato a ser eliminado
        q=cabecera.der # el candidato
        while not isinstance(q, Nodoe):
            if x<q.info:
                (p,q)=(q,q.izq)
            elif x>q.info:
                (p,q)=(q,q.der)
            else: # encontramos x
                if isinstance(q.izq, Nodoe): # no hay hijo izquierdo
                    r=q.der
                elif isinstance(q.der, Nodoe): # no hay hijo derecho
                    r=q.izq
                else: # hay 2 hijos, eliminamos el mínimo del árbol derecho y lo movemos a q
                    s=q.der
                    if isinstance(s.izq, Nodoe): # encontramos el mín de inmediato
                        q.info=s.info
                        q.der=s.der
                    else: # el mín está más abajo
                        t=s.izq # s es el padre del candidato a min, t es el candidato
                        while not isinstance(t.izq, Nodoe): # mientras no encontremos el final de la rama i
                            (s,t)=(t,t.izq)
                        q.info=t.info
                        s.izq=t.der
                    r=q
            if x<p.info:
                p.izq=r

```

```

        else:
            p.der=r
            self.raiz=cabecera.der
            return
        # si llegamos acá, x no estaba, no hacemos nada

def __str__(self):
    return self.raiz.__str__()

```

```

a=Arbol(
    Nodoi(
        Nodoi(
            Nodoi(Nodoe(),15,Nodoe()),
            20,
            Nodoi(
                Nodoi(Nodoe(),30,Nodoe()),
                35,
                Nodoe()
            )
        ),
        42,
        Nodoi(
            Nodoi(
                Nodoi(
                    Nodoi(Nodoe(),65,Nodoe()),
                    72,
                    Nodoi(Nodoe(),81,Nodoe())
                ),
                90,
                Nodoe()
            ),
            95,
            Nodoe()
        )
    )
)

```

```
print(a)
```

```
(( (15)20((30)35))42(((65)72(81))90)95))
```

```

a.delete(30)
print(a)

```

```
(( (15)20(35))42(((65)72(81))90)95))
```

```
a.delete(95)
print(a)
```

```
(( (15)20(35))42(((65)72(81))90))
```

```
a.delete(42)
print(a)
```

```
(( (15)20(35))65((72(81))90))
```

```
a.delete(44) # 44 no está en el árbol
print(a)
```

```
(( (15)20(35))65((72(81))90))
```

```
a.delete(20)
print(a)
```

```
(( (15)35)65((72(81))90))
```

Rotaciones en un ABB

Una operación que es la base de muchos algoritmos es la *rotación* (o rotación simple, para diferenciarla de la rotación doble, que veremos más adelante).

Una rotación entre los nodos b y d es la siguiente transformación:

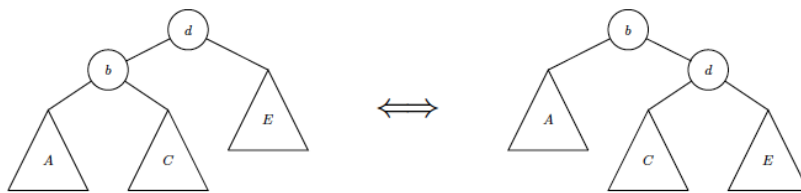


Figure 58: rotation-simple

Como lo sugiere la figura, por simetría esta operación también se puede hacer en la dirección inversa.

Esta operación tiene costo constante, porque solo requiere modificar tres punteros, y preserva el orden de izquierda a derecha que caracteriza a un ABB. Su efecto es que A y b suben un nivel, mientras que d y E bajan un nivel.

Aplicación: Inserción en la raíz

El método estándar de inserción en un ABB es inserción en las hojas. Veremos a continuación que existe un método alternativo, que deja al nuevo elemento como raíz del árbol.

El método consiste en insertar el nuevo elemento de la manera usual, y luego hacer una secuencia de rotaciones que vayan haciéndolo ascender, hasta que llegue a estar en la raíz.

```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der
    def right_rotation(self):
        return(Nodoi(self.izq.izq,
                      self.izq.info,
                      Nodoi(self.izq.der,self.info,self.der)))
    def left_rotation(self):
        return(Nodoi(Nodoi(self.izq,self.info,self.der.izq),
                      self.der.info,
                      self.der.der))
    def root_insert(self,x):
        assert x!=self.info
        if x<self.info:
            self.izq=self.izq.root_insert(x) # x queda como raíz del hijo izquierdo
            return self.right_rotation() # la rotación deja a x como raíz
        else:
            self.der=self.der.root_insert(x) # x queda como raíz del hijo derecho
            return self.left_rotation() # la rotación deja a x como raíz

    def __str__(self):
        return "("+self.izq.__str__()+str(self.info)+self.der.__str__()+")"

class Nodoe:
    def __init__(self):
        pass

    def root_insert(self,x):
        return Nodoi(Nodoe(),x,Nodoe())

    def __str__(self):
        return ""

class Arbol:
    def __init__(self,raiz=Nodoe()):
```

```

        self.raiz=raiz

    def root_insert(self,x):
        self.raiz=self.raiz.root_insert(x)

    def __str__(self):
        return self.raiz.__str__()

```

```

a=Arbol()
a.root_insert(10)
print(a)
a.root_insert(20)
print(a)
a.root_insert(30)
print(a)
a.root_insert(25)
print(a)
a.root_insert(15)
print(a)

```

```

(10)
((10)20)
(((10)20)30)
(((10)20)25(30))
((10)15((20)25(30)))

```

Aplicación: Árboles cartesianos y “Treaps”

Un *árbol cartesiano* es un árbol binario en que cada nodo interno contiene un par ordenado (x, y) , tal que:

- Si se consideran las coordenadas x , el árbol es un ABB
- Si se consideran las coordenadas y , el árbol es un árbol de prioridad

La siguiente figura muestra a un árbol cartesiano en que la máxima prioridad corresponde al valor mínimo de y :

Intuitivamente, podemos imaginar que la prioridad y corresponde a la hora en que la llave x ingresó al árbol. Esto explica que la raíz tenga el valor mínimo de y , y que estos valores vayan creciendo a medida que descendemos en el árbol.

Sin embargo, *no estamos obligados a insertar las llaves en orden cronológico*. En efecto, veremos que un nuevo par (x, y) se puede insertar en cualquier momento, aunque su y no sea mayor que los de los nodos que ya están en el árbol..

Para insertar un nuevo par ordenado en un árbol cartesiano, primero efectuamos una inserción como si fuera un ABB, con lo cual

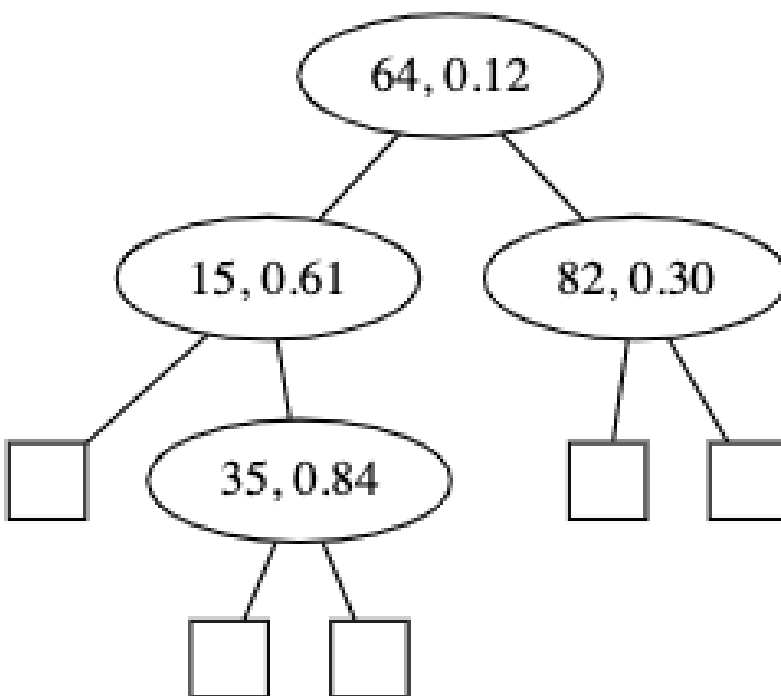


Figure 59: cartesian-tree

el nuevo nodo cumple con las restricciones “izquierda-derecha” en su coordenada x , pero posiblemente no con las restricciones de prioridad “arriba-abajo” de la coordenada y . Para corregir esto último, a continuación, hacemos una serie de rotaciones hasta conseguir que el nuevo nodo cumpla con las restricciones de prioridad.

También es posible eliminar un nodo dado de un árbol cartesiano. Como al eliminar un nodo sus dos árboles hijos quedan “huérfanos,” es necesario fusionarlos en uno solo. Para esto, elegimos a la raíz de mejor prioridad para que quede como raíz del árbol resultante. De los tres subárboles restantes, uno va a un lado de la nueva raíz, y los otros dos deben fusionarse recursivamente.

Una vez que tenemos una implementación de árboles cartesianos, podemos utilizarla para obtener un nuevo tipo de diccionario, llamado un *Treap* (mezcla de “tree” y de “heap”), que posee las mismas características de desempeño promedio que un ABB, pero que es inmune a que las llaves lleguen ordenadas, o a que un adversario le entregue la secuencia de inserciones en un orden malicioso. Para esto, dado una nueva llave x para insertar, generamos una prioridad y *al azar*, e insertamos el par (x, y) resultante. El efecto neto es equivalente a si las llaves hubieran llegado en un orden aleatorio.

Árboles AVL

Veremos a continuación que es posible implementar diccionarios en que los costos de búsqueda, inserción y eliminación son garantizada-mente $O(\log n)$ en el peor caso.

Hay muchas maneras de conseguir esto a través de modificaciones del algoritmo básico de los ABBs, y hablamos en general de *árboles balanceados* para referirnos a este tipo de diccionarios.

Comenzaremos viendo el más antiguo de estos algoritmos, inventado por Adelson-Velskii y Landis y conocido por sus iniciales como *árboles AVL*.

Si queremos garantizar un costo de búsqueda logarítmico, lo ideal sería poder construir árboles perfectamente balanceados, en que siempre los elementos se repartieran en partes iguales a ambos lados de la raíz, y en que por lo tanto las alturas a ambos lados también fueran idénticas. Pero eso no es posible por varias razones. Por una parte, porque el costo de insertar un nuevo elemento en ese tipo de árbol sería muy grande en el peor caso, pero también porque a medida que bajamos en el árbol, a menos que el número de llaves sea exactamente uno menos que una potencia de 2, necesariamente terminaremos encontrando subárboles de tamaño 2, en los cuales a un lado hay un árbol de altura 1, y al otro lado un árbol de altura 0.

Por lo tanto, es necesario relajar un poco la exigencia, para acomodar a ese tipo de subárboles.

Los árboles AVL se definen como ABBs en que todos sus nodos internos cumplen además una condición de *altura balanceada*, que consiste en que sus subárboles hijos pueden tener alturas diferentes, pero la diferencia de sus alturas no puede ser mayor que 1.

Más precisamente, si la función $h(T)$ es la altura del árbol T , para cada nodo interno sus subárboles hijos, digamos A y B , deben cumplir que $|h(A) - h(B)| \leq 1$.

El siguiente ejemplo muestra un árbol en que en cada nodo (interno o externo) se ha anotado la altura del subárbol que lo tiene como raíz. Este árbol cumple la condición AVL en todos sus nodos, excepto en el que aparece marcado con asterisco, cuyos hijos tienen diferencia de altura 2, y por lo tanto el árbol no es AVL.

En cambio, este otro árbol, muy similar, sí cumple la condición en todos sus nodos, y es por lo tanto un árbol AVL.

Chequeando si un árbol es AVL

Para determinar si un árbol es AVL, debemos calcular la altura de cada subárbol y comparar las alturas de todos los subárboles “hermanos” para ver si su diferencia excede o no 1. La siguiente implementación lo hace, pero veremos que de una manera ineficiente:

Figure 60: condicion-AVL

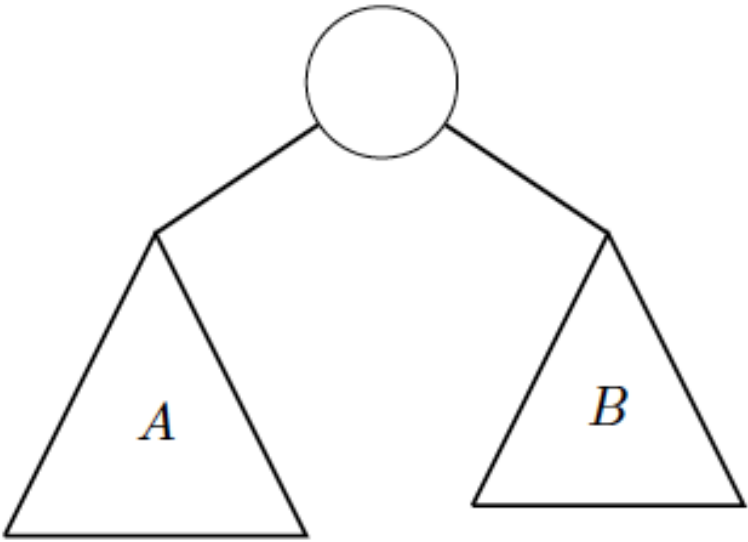


Figure 61: ejemplo-no-AVL

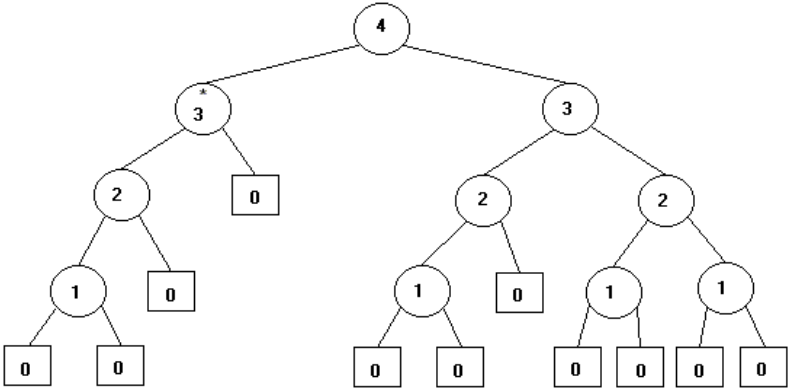
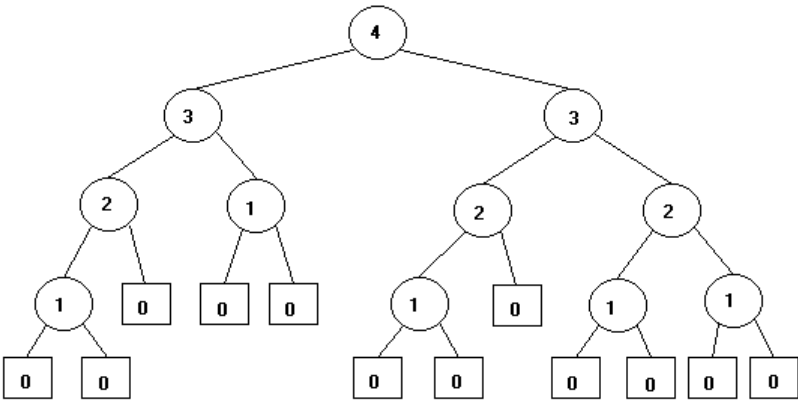


Figure 62: ejemplo-AVL



```

class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

    def altura(self):
        return 1+max(self.izq.altura(),self.der.altura())

    def es_AVL(self):
        return abs(self.izq.altura()-self.der.altura())<=1 \
            and self.izq.es_AVL() and self.der.es_AVL()

    def __str__(self):
        return "("+self.izq.__str__()+str(self.info)+self.der.__str__()+")"

class Nodoe:
    def __init__(self):
        pass

    def altura(self):
        return 0

    def es_AVL(self):
        return True

    def __str__(self):
        return ""

class Arbol:
    def __init__(self,raiz=Nodoe()):
        self.raiz=raiz

    def es_AVL(self):
        return self.raiz.es_AVL()

    def __str__(self):
        return self.raiz.__str__()

```

```

a1=Arbol(Nodoi(Nodoi(Nodoe(),1,Nodoe()),
                2,
                Nodoi(Nodoe(),3,Nodoi(Nodoe(),4,Nodoe()))))
print(a1)

```


A continuación, pruébela con los dos árboles utilizados anteriormente:

```
a1=Arbol(Nodoi(Nodoi(Nodoi(),1,Nodoi()),
                2,
                Nodoi(Nodoi(),3,Nodoi(Nodoi(),4,Nodoi()))))
print(a1)
print(a1.es_AVL())
```

```
a2=Arbol(Nodoi(Nodoi(Nodoi(),1,Nodoi()),
                2,
                Nodoi(Nodoi(),3,Nodoi(Nodoi(),4,Nodoi(Nodoi(),5,Nodoi())))))
print(a2)
print(a2.es_AVL())
```

Altura mínima y máxima de un árbol AVL

Consideremos un árbol AVL con n llaves y altura h . Sabemos que su número de hojas es $n + 1$, y también sabemos que el número de hojas es menor o igual a 2^h . Por lo tanto, $h \geq \lceil \log_2(n + 1) \rceil$.

Para encontrar una cota superior, consideremos la familia A_h de los árboles AVL de altura h con el mínimo número de hojas. Si $h = 0$, el único árbol posible es el árbol vacío (solo una hoja) y es AVL. Si $h = 1$, el único árbol posible es el que tiene un nodo interno y dos hojas como hijos, y también es AVL.

Consideremos el caso $h \geq 2$. El árbol A_h debe tener una raíz y dos subárboles hijos. Si queremos economizar al máximo el número de hojas, uno de ellos debe tener altura $h - 1$, para que la altura del árbol completo sea h , y el otro debe tener altura $h - 2$, para que se cumpla la condición AVL, y ambos deben tener el mínimo posible de hojas, es decir, deben pertenecer a la misma familia.

Por lo tanto, la familia A_h se puede construir recursivamente de la siguiente manera:

Figure 63: arboles-fibonacci

Los árboles así contruídos se llaman *árboles de Fibonacci*.

Si llamamos N_h al número de hojas del árbol A_h , tenemos que

$$N_0 = 1$$

$$N_1 = 2$$

$$N_h = N_{h-1} + N_{h-2}$$

Esta es la misma ecuación de Fibonacci, comenzando dos pasos

más adelante. Por lo tanto, si denotamos f_n al n -ésimo número de Fibonacci, tenemos que $N_h = f_{h+2} = \Theta(\phi^h)$.

Como los árboles de Fibonacci son los árboles AVL con el mínimo de hojas, cualquier otro árbol AVL tendrá más hojas que ellos, y por lo tanto, tomando logaritmos, la altura de todo árbol AVL será $O(\log_\phi(n+1))$.

Operaciones sobre un árbol AVL

Búsqueda Un árbol AVL es un ABB (con una condición adicional de balance), así que el algoritmo de búsqueda es el mismo. El que la altura de todo árbol AVL sea logarítmica garantiza que las operaciones de búsqueda tomarán tiempo logarítmico en el peor caso.

Inserción Veremos a continuación que se puede insertar una nueva llave en un árbol AVL, preservando la condición AVL, en tiempo a lo más proporcional a la altura del árbol.

Supongamos que se inserta una nueva llave x en un árbol AVL como el que muestra la siguiente figura:

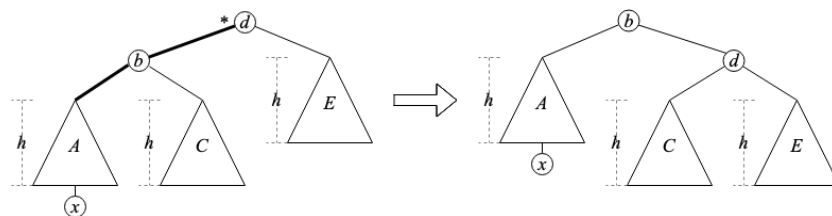


Figure 64: AVL1

Supongamos que se inserta la llave x , siguiendo una trayectoria desde la raíz hasta la hoja respectiva, y que luego se retorna hacia arriba revisando en cada nodo de esa trayectoria si la condición AVL se sigue cumpliendo. Si en todos ellos la condición se cumple, no es necesario hacer nada más. Si no, supongamos que el nodo marcado con asterisco (d) es el primero (de abajo hacia arriba) en donde la condición no se cumple.

Consideremos primero el caso en que a partir de d , los dos pasos siguientes hacia abajo fueron en la misma dirección (ambos hacia la izquierda, en la figura). Esto se llama una *inserción exterior*, y los dos pasos son de tipo “zig-zig” (o “zag-zag,” en el caso simétrico en que los dos pasos fueron hacia la derecha).

Para que la condición AVL no se cumpla en d y sí se cumpla en b , es necesario que los tres subárboles A , C y E sean todos de la misma altura, digamos h , y que la inserción de x haga que la altura de A crezca a $h+1$. Nótese que la altura del árbol, antes de la inserción, era $h+2$.

En este caso hacemos una rotación (simple) entre los nodos b y d , con el resultado que se muestra a la derecha. En el árbol resultante, los nodos b y d ahora cumplen la condición AVL (con diferencia cero) y, más aún, después de esta rotación, el árbol resultante tiene altura $h + 2$. Por lo tanto, el árbol completo no ha cambiado de altura, lo cual implica que ningún nodo más arriba puede estar desbalanceado.

Por lo tanto, en el caso de una inserción exterior, basta con a lo más una rotación para restaurar la condición AVL en todo el árbol.

Consideremos ahora el caso de una *inserción interior*, en que los dos primeros pasos fueron en direcciones opuestas ("zig-zag" o el simétrico "zag-zig"):

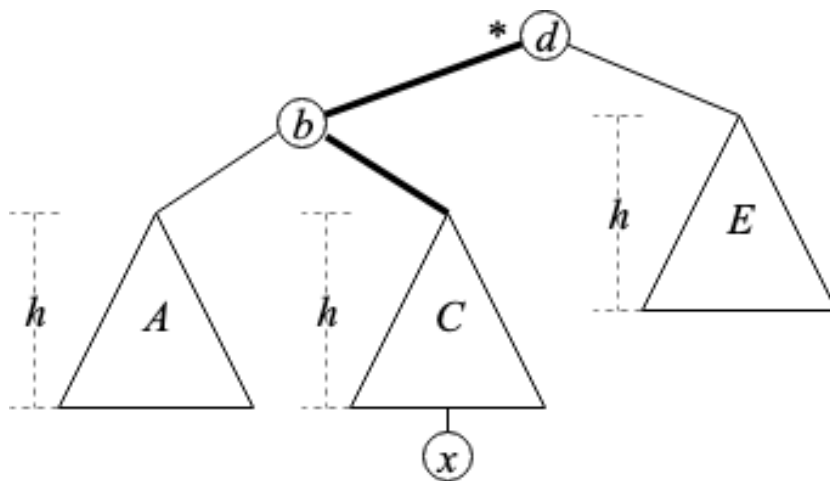


Figure 65: AVL2

En este caso, es necesario entrar a examinar la estructura interna del subárbol C:

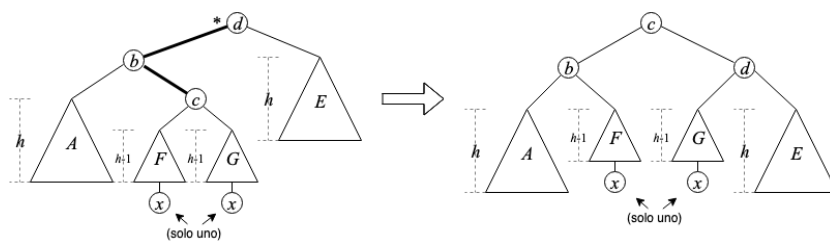


Figure 66: AVL3

En este caso, x puede ir al subárbol F o al subárbol G indistintamente (¡pero no a ambos!). Para restaurar la condición de balance AVL, necesitamos hacer *dos* rotaciones: primero entre c y b , y luego entre c y d . El resultado neto es que el zig-zag $d - b - c$ se transforma en el subárbol perfectamente balanceado $b - c - d$. El efecto combinado de estas dos rotaciones se llama una *rotación doble*.

Tal como ocurrió en el caso anterior, los tres nodos involucrados quedan cumpliendo la condición AVL, y el árbol resultante tiene altura $h + 2$, la misma que tenía antes de la inserción. Por lo tanto, no puede haber nodos desbalanceados más arriba.

Esto, sumado a lo que vimos antes, demuestra que **a lo más una rotación (simple o doble) basta para recuperar el balance AVL después de una inserción.**

```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der
        self.height=1+max(izq.height,der.height)

    def right_rotation(self):
        return(Nodoi(self.izq.izq,
                     self.izq.info,
                     Nodoi(self.izq.der,self.info,self.der)))

    def left_rotation(self):
        return(Nodoi(Nodoi(self.izq,self.info,self.der.izq),
                     self.der.info,
                     self.der.der))

    def insert(self,x):
        assert x!=self.info
        if x<self.info:
            p=Nodoi(self.izq.insert(x),self.info,self.der)
            if p.izq.height>p.der.height+1:
                if x<p.izq.info: # inserción exterior
                    p=p.right_rotation()
                else: # inserción interior
                    p=Nodoi(p.izq.left_rotation(),p.info,p.der).right_rotation()
            else: # x>self.info, simétrico del anterior
                p=Nodoi(self.izq,self.info,self.der.insert(x))
                if p.der.height>p.izq.height+1:
                    if x>p.der.info: # inserción exterior
                        p=p.left_rotation()
                    else: # inserción interior
                        p=Nodoi(p.izq,p.info,p.der.right_rotation()).left_rotation()
            return p

    def __str__(self):
```

```

        return "("+self.izq.__str__()+str(self.info)+self.der.__str__()+")"

class Nodoe:
    def __init__(self):
        self.height=0

    def insert(self,x):
        return Nodoe(x,Nodoe())

    def __str__(self):
        return ""

class ArbolAVL:
    def __init__(self,raiz=Nodoe()):
        self.raiz=raiz

    def insert(self,x):
        self.raiz=self.raiz.insert(x)

    def __str__(self):
        return self.raiz.__str__()

```

```

a=ArbolAVL()
a.insert(20)
print(a)
a.insert(40)
print(a)
a.insert(80)
print(a)
a.insert(10)
print(a)
a.insert(15)
print(a)

```

```

(20)
(20(40))
((20)40(80))
(((10)20)40(80))
(((10)15(20))40(80))

```

Eliminación La eliminación en árbol AVL se realiza de manera análoga a un ABB, pero también es necesario verificar que la condición de balance se mantenga una vez eliminado el elemento. En caso que dicha condición se pierda, será necesario realizar una rotación

simple o doble dependiendo del caso, pero es posible que se requiera más de una rotación para reestablecer el balance del árbol.

Árboles 2-3

Los árboles 2-3 son árboles cuyos nodos internos pueden ser *binarios* o *ternarios*. Los nodos binarios tienen una llave en su interior, y los nodos ternarios tienen dos llaves.

Las llaves contenidas en los nodos del árbol cumplen la siguiente relación de orden de izquierda a derecha:

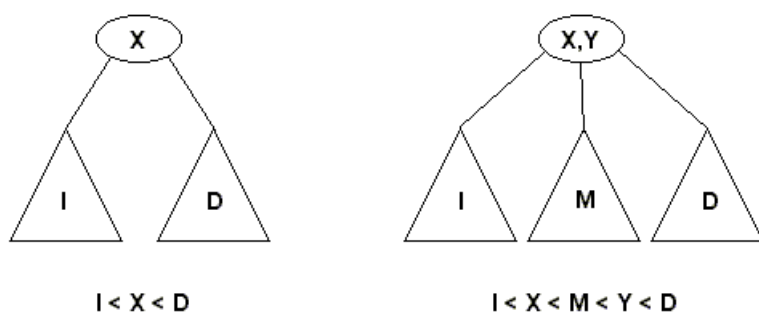


Figure 67: orden-2-3

En un nodo binario, la llave X actúa como separador entre las llaves del subárbol izquierdo y las del derecho. En un nodo ternario, la llave X separa el subárbol izquierdo del subárbol del medio, y la llave Y separa el subárbol del medio del subárbol derecho.

Gracias a este orden, se puede realizar una búsqueda utilizando una generalización del algoritmo de búsqueda en un ABB.

Además de lo anterior, un árbol 2-3 debe tener a todas sus hojas en el mismo nivel. El siguiente es un ejemplo de un árbol 2-3 que satisface ambas condiciones:

Nótese que si un árbol 2-3 tiene n llaves, ya no necesariamente se cumple que el número de hojas (nodos externos) sea uno más que el número de nodos internos, pero sí se cumple que el número de hojas es igual a $n + 1$.

Dado que la altura de un árbol 2-3 es mínima si todos sus nodos son ternarios y es máxima si todos sus nodos son binarios, es fácil ver que

$$\log_3(n+1) \leq \text{altura} \leq \log_2(n+1)$$

y por lo tanto el costo de búsqueda es logarítmico en el peor caso.

Inserción en un árbol 2-3

Para insertar una nueva llave X en un árbol 2-3, se realiza una búsqueda infructuosa y se inserta dicha llave en el último nodo

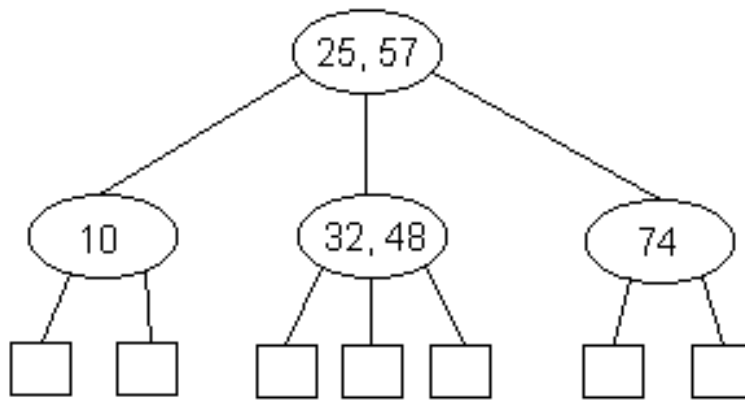


Figure 68: ejemplo-2-3

interno visitado durante la búsqueda. Esto puede conducir a dos casos distintos:

- Si el nodo tenía solo una llave (era binario), ahora queda con dos llaves y pasa a ser ternario:

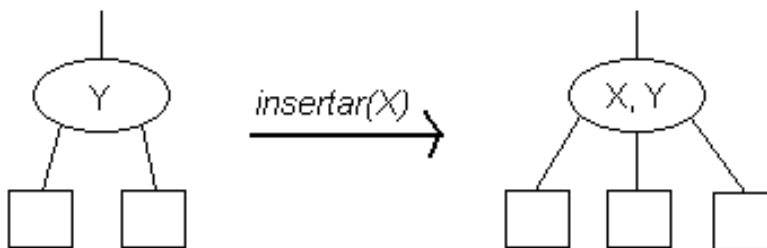


Figure 69: insercion-2-3-binario

- Si el nodo tenía 2 llaves (era ternario), queda transitoriamente con 3 llaves y 4 hijos. Se dice que ese nodo está saturado, o que tiene *overflow*, y esto se corrige dividiendo el nodo en dos nodos binarios (*split*). Los 4 hijos se reparten entre esos dos nodos. Respecto a las llaves, la menor queda en el nodo izquierdo, la mayor en el nodo derecho, y la mediana sube y se inserta en el padre, para actuar como separador entre los dos subárboles resultantes del *split*.

Si el padre era binario, la llegada de una nueva llave lo transforma en ternario y el proceso concluye ahí. Pero si ya era ternario, el padre ahora está saturado y el proceso se repite.

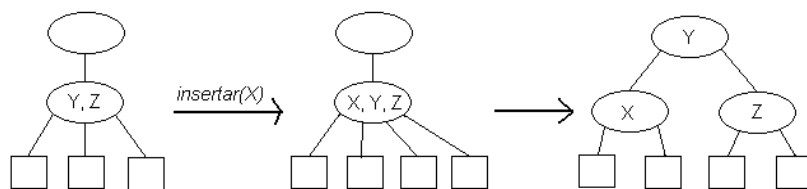


Figure 70: insercion-2-3-ternario

En el peor caso, es posible que la raíz llegue a dividirse. En ese caso, se crea una nueva raíz un nivel más arriba la altura del árbol crece en 1.

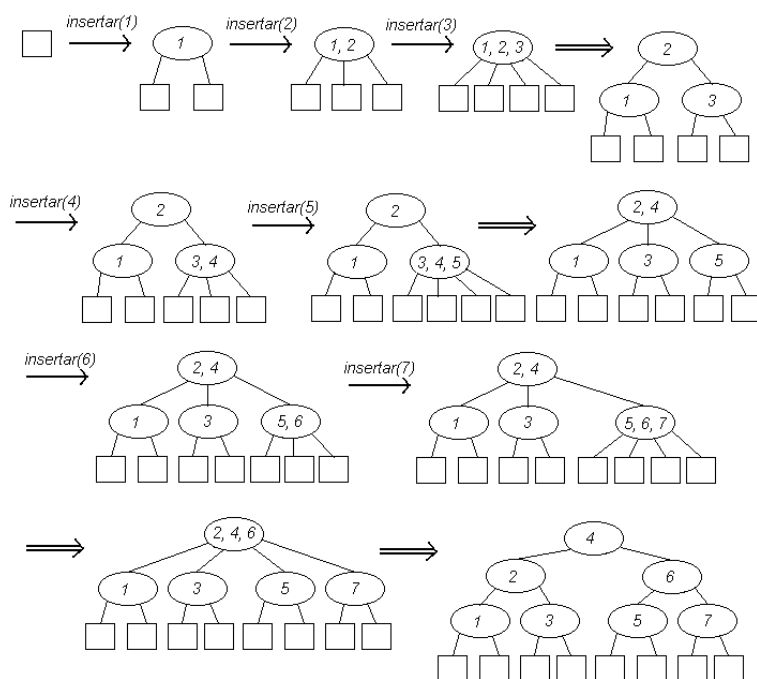


Figure 71: inserciones-2-3

Ejemplo de inserciones en un árbol 2-3

Ejercicio 6.2 (Número de operaciones *split*)

Supongamos que comenzamos con un árbol 2-3 vacío y a continuación insertamos n llaves. El objetivo de este ejercicio es calcular cuántas operaciones *split* se pueden llegar a ejecutar a lo largo de este proceso, en el peor caso.

Una manera de acotar este número de operaciones es ver que una inserción puede gatillar la ejecución de $O(\log n)$ operaciones *split*,

y como son n inserciones, el número total de operaciones *split* es $O(n \log n)$.

Lo anterior es una cota superior, pero en realidad está muy sobredimensionada.

Demuestre que el número total de operaciones *split* ejecutadas al insertar n llaves comenzando con un árbol 2-3 vacío es $O(n)$ en el peor caso. Escriba su demostración en el siguiente recuadro.

Indicación: Considere el impacto de un *split* sobre el número de nodos del árbol.

Demostración:

Eliminación en un árbol 2-3

Supongamos que queremos eliminar una llave Z . Sin perder generalidad, podemos suponer que Z está en el nivel más bajo de árbol. Si esto no es así, entonces el sucesor y el predecesor de Z se encuentran necesariamente en el nivel inferior, en cuyo caso basta con eliminar uno de ellos, y luego sobrecribir esa llave en el lugar de Z .

La eliminación presenta dos posibles casos:

- El nodo en donde se encuentra Z es ternario. En este caso, se elimina Z junto con una hoja y el nodo queda como binario.
- El nodo en donde se encuentra Z es binario. En ese caso, al eliminar Z y una hoja, el nodo queda como “unario” y sin llaves (*underflow*). Cuando esto sucede, tenemos dos casos:
 - Si el nodo hermano es ternario, le quitamos una llave y una hoja y ambos quedan como binarios. Nótese que, para preservar el orden, se debe realizar un traspaso con la llave en el padre que separa a ambos nodos:
 - Si el nodo hermano es binario, no le podemos quitar nada. En ese caso, fusionamos a ambos hermanos, resultando un nodo ternario, y la llave del padre que actuaba como separador baja a este nuevo nodo:

Cuando ocurre esto último, el padre pierde un hijo y una llave, lo cual podría dejarlo transformado en unario. En ese caso, se repite el procedimiento anterior, un nivel más arriba.

El peor caso es que finalmente la raíz quede como un nodo vacío unario. En ese caso, se elimina esa raíz y su hijo queda como raíz.

Costo de las operaciones

Dado que el árbol 2-3 tiene altura logarítmica, y que las operaciones de inserción, eliminación y búsqueda hacen una cantidad de

trabajo constante en cada nivel, el costo de dichas operaciones en el peor caso es $\Theta(\log n)$.

Árboles B (B-trees)

Los árboles 2-3 se pueden generalizar a árboles con un mayor número de hijos. La idea es que cuando un nodo excede el número máximo de hijos (y de llaves) permitidos, se divide en dos (*split*), con la mitad de los hijos cada uno.

Más precisamente, supongamos que $t \geq 2$ es un parámetro fijo, y permitimos que todo nodo tenga un mínimo de t y un máximo de $2t - 1$ hijos. Un nodo con k hijos tiene $k - 1$ llaves, que actúan como separadoras entre sus hijos.

Los árboles 2-3 son árboles B con $t = 2$. En la práctica, los árboles B se implementan con valores mucho más grandes de t , por ejemplo $t = 100$.

La motivación para utilizar valores altos de t es hacer coincidir el tamaño del nodo con la capacidad de un bloque de disco (memoria secundaria). Ésta es memoria masiva de acceso más lento que la memoria central del computador (memoria RAM), y al tener un grado alto de ramificación, el árbol tiene muy pocos niveles. Gracias a esto, el número de accesos a disco para implementar las operaciones es también muy pequeño.

Inserción en un árbol B

- La nueva llave, junto con una nueva hoja, se agregan en el nodo correspondiente en el nivel inferior del árbol.
- Si ese nodo queda saturado (*overflow*), es decir, si queda con $2t$ hijos y $2t - 1$ llaves, se divide en dos nodos con t hijos y $t - 1$ llaves cada uno. La llave sobrante, que debe ser la mediana del conjunto de llaves, sube al padre como separador de estos dos nuevos nodos.
- Si a causa de esto el padre queda saturado, se repite el procedimiento un nivel más arriba.
- El peor caso es que la raíz llegue a estar saturada. En ese caso, la raíz se divide en dos nodos y se crea un nuevo nodo binario como padre de ellos. Este nodo es la nueva raíz.

Nótese que, debido a lo anterior, la raíz es el único nodo que está exento de la obligación de tener al menos t hijos.

Eliminación en un árbol B

- Tal como hicimos en el caso de los árboles 2-3, sin perder generalidad, suponemos que la llave a eliminar está en el nivel inferior. Se elimina la llave y una hoja.

- Si a causa de esto el nodo queda con menos de t hijos, tenemos dos casos:
 - Si el hermano tiene más de t hijos, redistribuimos los hijos y las llaves entre ambos hermanos, mitad y mitad.
 - Si el hermano tiene solo t hijos, no le podemos quitar nada. En ese caso, fusionamos ambos nodos, y la llave del padre que los separaba baja al nuevo nodo. Si a causa de eso el padre queda con menos del mínimo de hijos, se repite este procedimiento un nivel más arriba.
 - Excepcionalmente, a la raíz se le permite que tenga menos de t hijos, pero si llega a quedar como un nodo unario (sin llaves), se elimina y su hijo queda como nueva raíz.

Costo de las operaciones en un árbol B

En las estructuras de datos implementadas en memoria secundaria, el costo de las operaciones se aproxima por el número de bloques leídos o escritos, y se ignora el costo del procesamiento en memoria RAM. Esta aproximación se justifica porque el acceso a memoria secundaria es mucho más lento que a memoria RAM.

En un árbol B, el número de accesos a bloques corresponde al número de niveles del árbol, el cual es logarítmico, pero ese logaritmo es en una base muy grande (entre t y $2t - 1$), de modo que en la práctica podemos decir que el costo es “casi constante.”

Variantes de los árboles B

Veremos dos variantes de la definición anterior, que son importantes en la práctica.

*Árboles B** La idea es no hacer un *split* apenas un nodo se satura, sino ver si el hermano tiene capacidad como para recibir que le enviemos llaves e hijos. El *split* solo se hace cuando *ambos* nodos están saturados, y en ese caso se crea un tercer nodo para distribuir entre los tres equitativamente el contenido.

En los árboles B, la utilización del espacio (fracción ocupada de un nodo) varía entre 50% y 100%. En un árbol B* la utilización varía entre 66,6% y 100%. Al haber una mayor utilización, el grado de ramificación de los nodos es mayor, y por la altura del árbol es menor.

Árboles B+ En los árboles B+, los datos del conjunto se almacenan en orden ascendente de llave en las hojas. Cada hoja tiene una cierta capacidad máxima (correspondiente a la capacidad del bloque de disco en donde se almacena), y cuando se excede, la hoja se divide en dos, cada una 50% llena. Adicionalmente, las hojas se enlazan en

una lista de doble enlace, que permite un recorrido secuencial de los datos en orden ascendente o descendente de llave.

La máxima llave de cada hoja (excepto la de la hoja de más a la derecha) se toma como separador y se almacena dentro de los nodos internos del árbol. Nótese que en los nodos internos solo están las llaves y no toda la información que puede acompañarlas, en cambio en las hojas está la información completa.

Debido a esto, al hacer una búsqueda, las preguntas se hacen con \leq / $>$ y se debe llegar siempre hasta el nivel de las hojas.

El nivel de las hojas se llama en "*B-file*," y el conjunto de los nodos internos del árbol se llama el "*B-index*."

Los árboles B+ son la versión más utilizada, porque permiten hacer tanto recorridos secuenciales como búsqueda eficientes en memoria secundaria.

Árboles Digitales

En esta sección exploramos una manera distinta de almacenar un diccionario usando árboles. Hasta ahora hemos supuesto que las llaves pertenecen a algún conjunto ordenado, pero no hemos hecho ninguna suposición respecto de la estructura interna de las llaves. Ahora vamos a suponer que toda llave X se representa como una secuencias de bits:

$$X = b_0b_1 \dots b_k$$

Vamos a suponer también que ninguna secuencia de bits asociada a una llave es prefijo de la secuencia de bits de otra llave. Esto se cumple trivialmente si todas las llaves son del mismo largo.

En los ejemplos vamos a considerar el siguiente conjunto de llaves:

A = 00100

B = 01000

C = 01111

D = 11000

E = 11101

Árboles digitales o "tries"

Un *trie* es un árbol binario cuyos nodos internos solo se utilizan como puntos de ramificación, para separar las llaves según los bits que la componen. Por convención, supondremos que las llaves que tienen un bit 0 en una posición dada se van hacia la izquierda, y las que tienen un 1 se van hacia la derecha. Cuando se llega a que la secuencia de bits identifica a solo una posible llave, se coloca una hoja que contiene a dicha llave en su interior. Por ejemplo:

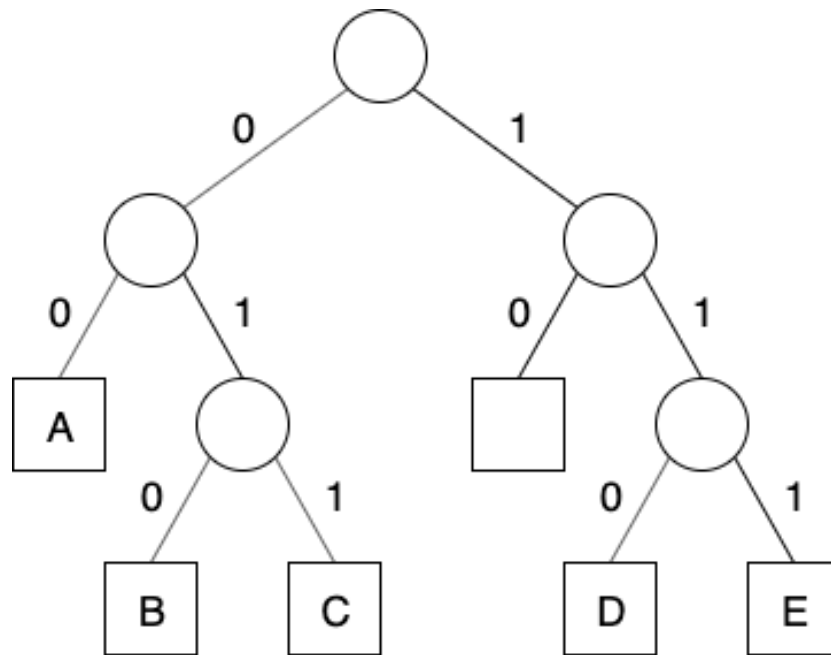


Figure 72: trie

Nótese que una hoja puede ser vacía. Esto ocurre si no hay ninguna llave que comience con ese prefijo, pero en la rama opuesta hay más de una llave.

Búsqueda en un trie Para buscar una llave X en un *trie* se procede de la siguiente manera:

- Se examinan los bits comenzando desde el bit b_0 en adelante.
- Si $b_i = 0$, se avanza por la rama izquierda y se pasa a examinar el siguiente bit, b_{i+1} .
- Si $b_i = 1$, se avanza por la rama derecha y se pasa a examinar el siguiente bit.
- El proceso termina cuando se llega a una hoja, único lugar posible en donde puede estar X . Si la llave almacenada en esa hoja es igual a X , se encontró. Si la hoja está vacía, o contiene una llave distinta de X , no está.

Insertión en un trie Para insertar una llave X en un *trie*, se realiza una búsqueda infructuosa, y se almacena transitoriamente la llave X en esa hoja.

Si la hoja estaba vacía, la llave X queda en ese lugar de manera definitiva.

Si la hoja contenía otra llave Y , se divide la hoja en dos, colgando desde un nuevo nodo interno, y las llaves se trasladan hacia la hoja

que corresponda, según si su siguiente bit es 0 o 1. Se repite este proceso en caso de ser necesario, hasta que en ninguna hoja quede más de una llave.

Es importante notar que, para un conjunto dado de llaves, el *trie* resultante es único, de modo que el orden de llegada de las llaves no cambia el resultado final. Uno podría suponer que al principio **todas** las llaves estaban en una única hoja, la cual se fue subdividiendo y subdividiendo hasta que en cada hoja quedó a lo más una llave.

Eliminación en un trie Para eliminar una llave X primero se ubica la hoja que la contiene, y se borra la llave, con lo cual la hoja queda vacía. Luego se mira si esta hoja vacía es hermana de otra hoja (la necesariamente sería no vacía) y ambas se fusionan, quedando un nivel más arriba. Se repite este procedimiento hasta que ninguna hoja con una llave quede como hermana de una hoja vacía.

Costo esperado de búsqueda exitosa en un trie Los *tries* tienden a ser mejor balanceados que los ABBs, porque si los bits son uniformemente distribuidos, la probabilidad de que una nueva llave vaya a la izquierda o a la derecha es $1/2$. En un ABB, en cambio, esa probabilidad es proporcional al tamaño del subárbol respectivo, de modo que si un árbol está desbalanceado, tiene una tendencia a desbalancearse más.

El análisis matemático del costo esperado de búsqueda es muy complicado, por lo cual solo daremos el resultado sin demostrarlo:

$$C_n = \frac{H_n}{\ln 2} + \frac{1}{2} + P(\log_2 n) \\ \approx \log_2 n + 1.3327 \dots$$

donde $P()$ es una función periódica de muy pequeña amplitud (del orden de 10^{-6}).

Árboles de Búsqueda Digital

Los árboles de búsqueda digital (ABD) son un híbrido entre los ABBs y los *tries*.

Las llaves se almacenan en los nodos internos, al igual que en un ABB, pero la ramificación izquierda-derecha se hace según si el siguiente bit es 0 o 1.

Como los elementos se van insertando en orden de llegada, este orden sí impacta en el resultado final. Por ejemplo, si el orden de llegada de las llaves fuera B, A, C, D, E , el resultado sería:

El costo esperado de búsqueda en un ABD es levemente menor al de un *trie*, y se puede demostrar que

$$C_n \approx \log_2 n - 0.7166 \dots$$

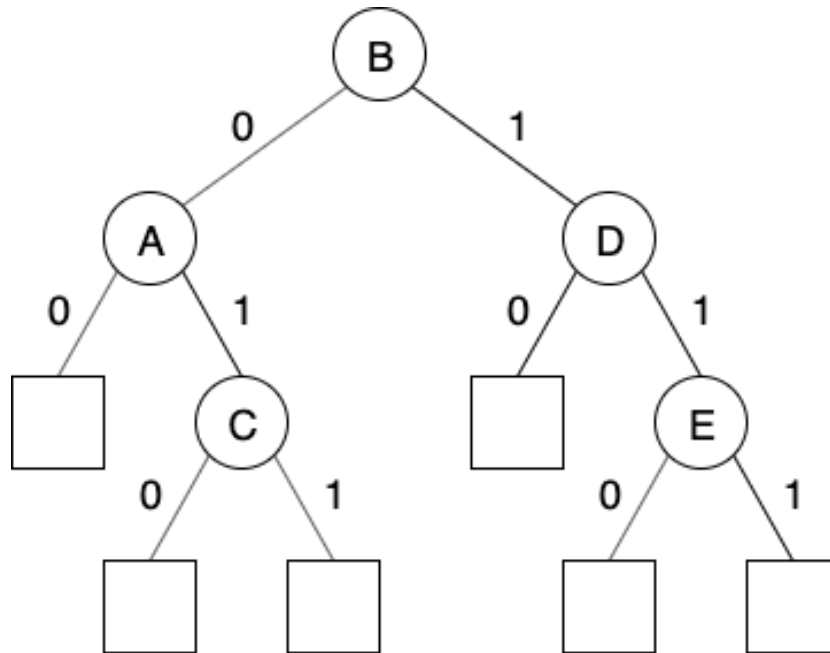
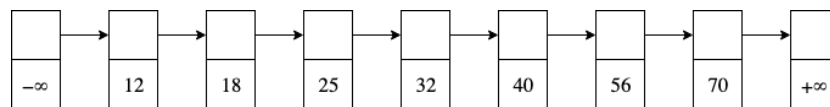


Figure 73: abd

Listas Saltadas (Skip Lists)

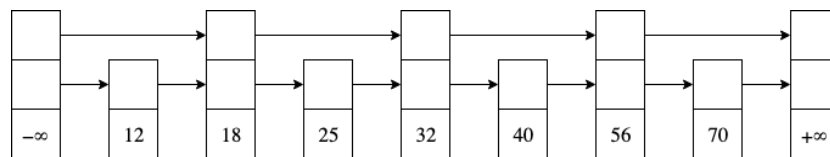
Las listas saltadas son una estructura aleatorizada basada en listas enlazadas.

Para motivar la idea, comencemos con una sencilla lista enlazada de menor a mayor, con una cabecera (que imaginamos que tiene una llave $-\infty$) y un fin de lista (que imaginamos que tiene una llave $+\infty$):

Figure 74: skip-list₁

Una lista como ésta no es muy eficiente, dado que sabemos que una búsqueda en ella demora tiempo $\Theta(n)$ tanto en promedio como en el peor caso.

Para aumentar la eficiencia de la búsqueda, supongamos que enlazamos los elementos uno por medio en una segunda lista:

Figure 75: skip-list₂

De esta manera, podemos buscar primero en la lista de más arriba para acotar en intervalo en que está la llave buscada, y luego usar la lista inferior para buscar en dicho intervalo. Con eso, el tiempo de búsqueda se reduce aproximadamente a la mitad.

Dado que esto fue una buena idea, la podemos aplicar de nuevo, las veces que sea necesaria:

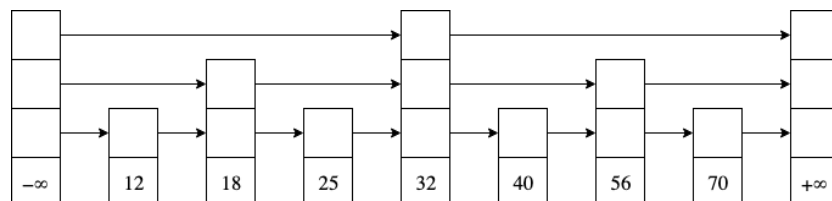


Figure 76: skip-list3

La estructura resultante permite hacer búsquedas tan eficientes como la búsqueda binaria, pero sirve solo como motivación para el diseño de una estructura verdaderamente eficiente, porque como está no admite inserciones ni eliminaciones de costo logarítmico.

La clave para obtener una estructura flexible es abandonar la idea de que la altura de cada “torre” de punteros vaya variando regularmente, y concentrarse en que esas alturas solo sigan la misma distribución. Para eso, introduciremos aleatoriedad: para determinar cuántos punteros tendrá un elemento dado, lanzaremos una moneda repetidamente hasta obtener “cara,” y el número de veces que fue necesario lanzar será el número de punteros para ese elemento.

Así, todos los elementos tienen al menos un puntero, en promedio la mitad tiene al menos 2, la cuarta parte al menos 3, etc.

El resultado podría ser, por ejemplo:

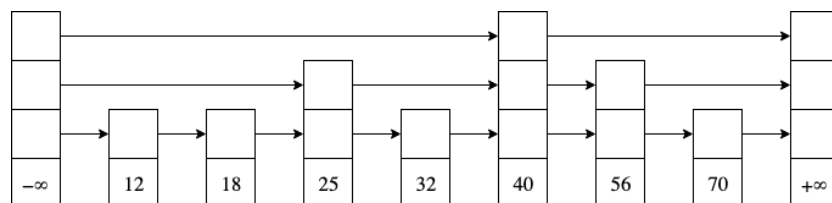


Figure 77: skip-list4

Búsqueda en una lista saltada

Supongamos que queremos buscar la llave x . Para esto, vamos a colocar una ficha inicialmente en el casillero correspondiente al tope de la torre de punteros de la cabecera (“ $-\infty$ ”), y la vamos a ir moviendo de acuerdo a las siguientes reglas.

En cada paso, llamemos y a la llave a la cual apunta el puntero donde está la ficha.

- Si $x > y$, movemos la ficha horizontalmente hacia la derecha.
- Si $x \leq y$, movemos la ficha verticalmente hacia abajo.

Lo anterior continúa hasta que al tratar de ejecutar el paso 2, no se puede porque la ficha ya está en el nivel inferior. En ese caso, hacemos una comparación de igualdad entre x e y . Si $x = y$, lo hemos encontrado, y si no, no está.

La siguiente animación ilustra este proceso:

Figure 78: skip-search

Inserción en una lista saltada

Para insertar una nueva llave, se realiza primero una búsqueda infructuosa, y se conserva en una pila la lista de todos los punteros en donde se dio un paso vertical. Luego de determinar aleatoriamente cuántos punteros debe tener el nuevo elemento, éste se inserta en el lugar correspondiente y se recorre la pila, haciendo que todos esos punteros ahora apunten al nuevo elemento, y que éste apunte hacia adonde apuntaban ellos.

Eliminación en una lista saltada

Para eliminar una llave, se realiza una búsqueda y se conserva en una pila la lista de todos los punteros en donde se dio un paso vertical. Luego se recorre la pila, haciendo que todos esos punteros apunten hacia adonde apuntaba el elemento que se está eliminando.

Análisis de las listas saltadas

El análisis matemático de las listas saltadas es muy complejo, por lo que solo daremos aquí los resultados.

Para generalizar, vamos a suponer que la moneda no es necesariamente insesgada. Suponiendo que la probabilidad de que salga “cara” es p , un primer resultado interesante es la *altura* de la lista saltada, esto es, el número de punteros que tiene la “torre” más alta, lo cual corresponde también al número de punteros que debe tener el elemento “cabecera” de las listas.

Se puede demostrar que la altura esperada de una lista saltada con n llaves es

$$\log_{1/p} n + \Theta(1)$$

Más aún, su varianza es constante, y aproximadamente igual a

$$\frac{\pi^2}{6 \ln^2 p} + \frac{1}{12}$$

con lo cual resulta muy improbable que la altura pueda exceder por mucho a su valor esperado.

Respecto al costo esperado de búsqueda, se puede demostrar que

$$\begin{aligned}
C_n &= \frac{1}{p} \log_{1/p} n + \Theta(1) \\
&= \frac{1}{p \log_2(1/p)} \log_2 n + \Theta(1)
\end{aligned}$$

Si la moneda es insesgada ($p = 1/2$), tenemos que la altura esperada es $\log_2 n + \Theta(1)$ y que el costo esperado de búsqueda es $2 \log_2 n + \Theta(1)$.

Pero el valor que minimiza la constante que multiplica al $\log_2 n$ no es $p = 1/2$, sino que es $p = 1/e = 0.3678794412 \dots$, y para este valor de p se tiene que el costo esperado de búsqueda es

$$C_n = (e \ln 2) \log_2 n + \Theta(1) \approx 1.884 \log_2 n + \Theta(1)$$

Árboles de búsqueda binaria óptimos

Si todas las llaves de un ABB son igualmente probables de ser buscadas, el árbol que minimiza el costo esperado de búsqueda es el árbol perfectamente balanceado.

Pero si las llaves tienen probabilidades no uniformes, o si las búsquedas infructuosas pueden caer en hojas con distinta probabilidad, el árbol que minimiza el costo esperado de búsqueda puede ser muy diferente.

Supongamos que tenemos n llaves $X_1 < X_2, \dots < X_n$ con probabilidades de acceso conocidas p_1, p_2, \dots, p_n respectivamente, y supongamos que las hojas, numeradas de 0 a n de izquierda a derecha, tienen probabilidades de búsqueda infructuosa q_0, q_1, \dots, q_n respectivamente.

El problema es encontrar el árbol de búsqueda binaria que minimiza el costo esperado de búsqueda

$$C(0, n) = \sum_{1 \leq i \leq n} (1 + \text{nivel}(\text{llave } i)) p_i + \sum_{0 \leq i \leq n} \text{nivel}(\text{hoja } i) q_i$$

donde los niveles del árbol se numeran de 0 en adelante a partir de la raíz.

Por ejemplo, consideremos en siguiente ABB con 6 llaves, en donde en cada nodo (interno o externo) se ha anotado la probabilidad de que la búsqueda termine en él:

El costo esperado de búsqueda es

$$C(0, 6) = (2q_0 + 2p_1 + 4q_1 + 4p_2 + 4q_2 + 3p_3 + 4q_3 + 4p_4 + 4q_4) + p_5 + (2q_5 + 2p_6 + 2q_6)$$

donde hemos parentizado la parte que corresponde al subárbol izquierdo y derecho, respectivamente.

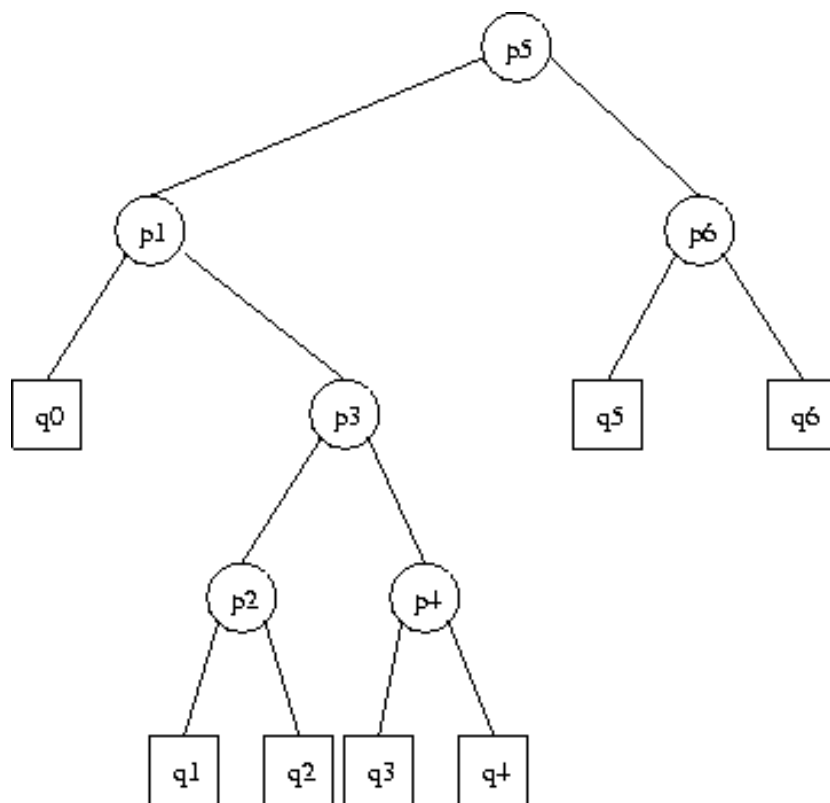


Figure 79: abbopt1

Escribamos ahora las fórmulas análogos para los subárboles, suponiendo que ellos fueran árboles independientes:

$$C(0,4) = q_0 + p_1 + 3q_1 + 3p_2 + 3q_2 + 2p_3 + 3q_3 + 3p_4 + 3q_4$$

$$C(5,6) = q_5 + p_6 + q_6$$

Si restamos esto a la fórmula para $C(0,6)$, la diferencia es la suma de todas las probabilidades, llamémosla $W(0,6)$ (W por “weight,” o peso). Por cierto, en el caso del árbol completo esta suma es igual a 1, pero no así en los subárboles.

Utilizando esta notación, tenemos que

$$C(0,6) = W(0,6) + C(0,4) + C(5,6)$$

Lo anterior es para un árbol con hojas numeradas de 0 a 6 y con la raíz ocupada por X_5 .

Más en general, si tenemos un árbol con hojas numeradas desde i a j y con X_k en la raíz, se cumple que

$$C(i,j) = W(i,j) + C(i,k-1) + C(k,j)$$

Nótese que si éste fuera el árbol óptimo, los subárboles también deberían ser óptimos, porque de lo contrario podríamos reemplazar ese subárbol por uno de costo menor, y el costo total disminuiría estrictamente, contradicción con la hipótesis de que el árbol era óptimo.

Pero al revés no necesariamente es cierto: los subárboles pueden ser óptimos, pero el árbol completo no, si es que hemos elegido mal la raíz. Para asegurarnos de que el árbol completo sea óptimo, debemos minimizar sobre todas las raíces posibles:

$$\hat{C}(i,j) = W(i,j) + \min_{i+1 \leq k \leq j} (\hat{C}(i,k-1) + \hat{C}(k,j))$$

y el costo de buscar en un árbol vacío es 0:

$$\hat{C}(i,i) = 0 \text{ para } 0 \leq i \leq n$$

Finalmente, el costo del árbol óptimo para el conjunto completo de llaves es $\hat{C}(0,n)$.

Un algoritmo de Programación Dinámica para encontrar un ABB óptimo

A partir de la recurrencia anterior se puede formular fácilmente un algoritmo recursivo para el árbol óptimo y su costo, pero eso sería equivalente a un algoritmo de fuerza bruta que examina todos los árboles posibles. Como el número de árboles es el número de

Catalan, que crece exponencialmente con n , es sería un algoritmo muy ineficiente.

En cambio, si utilizamos *tabulación*, podemos resolver el problema de manera mucho más rápida. La tabulación es posible porque hay solo $\Theta(n^2)$ valores $\hat{C}(i, j)$ distintos por calcular, los cuales podemos almacenar en una matriz, y la podemos ir llenando de manera que todos los casilleros necesarios para calcular el casillero (i, j) ya hayan sido llenados previamente.

Para implementar el algoritmo, definamos una nueva variable $d = j - i$ que calcula el número de llaves del subárbol (i, j) , y vamos a ir llenando una matriz C en orden creciente de d .

Al mismo tiempo que vamos llenando esa matriz, vamos a ir llenando una matriz paralela R que almacenará el valor del k para el cual se alcanza el óptimo en el subárbol (i, j) . Éste será el sub-índice de la raíz óptima para dicho subárbol, lo cual nos permitirá construir el árbol óptimo y también, como veremos pronto, acelerar significativamente el algoritmo.

```
import math
import numpy as np

def calculaABBOptimo(p,q):
    n=len(q)-1
    C=np.zeros((n+1,n+1)) # Esto ya deja la diagonal en cero
    W=np.zeros((n+1,n+1))
    R=np.zeros((n+1,n+1),dtype=int)
    for i in range(0,n+1):
        W[i,i]=q[i]
    for d in range(1,n+1):
        for i in range(0,n-d+1):
            j=i+d
            W[i,j]=W[i,j-1]+p[j]+q[j]
            C[i,j]=math.inf # +infinito
            for k in range(i+1,j+1):
                m=W[i,j]+C[i,k-1]+C[k,j]
                if m<C[i,j]:
                    C[i,j]=m
                    R[i,j]=k # anotamos dónde se alcanza el min
    return (C[0,n],R)

def ABBOptimo(R,i,j):
    return "" if i==j else "("+ABBOptimo(R,i,R[i,j]-1)+"X"+str(R[i,j])+ABBOptimo(R,R[i,j],j)+")"
```

```

p=[0,0.15,0.10,0.05,0.10,0.20]
q=[0.05,0.10,0.05,0.05,0.05,0.10]
(Copt,R)=calculaABBoptimo(p,q)
print("Costo óptimo=", Copt)
print("Árbol óptimo=", ABBoptimo(R,0,5))

```

Costo óptimo= 2.35

Árbol óptimo= (((X1)X2(X3))X4(X5))

Análisis del algoritmo

El algoritmo rellena todos los casilleros (i, j) de la matriz, con $0 \leq i \leq j \leq n$, y para cada casillero recorre el rango $(i + 1)..j$.

Dado que hay aproximadamente $n^2/2$ casilleros que se deben llenar, y que para cada uno de ellos el rango es de tamaño a lo más n , es evidente que el tiempo de ejecución del algoritmo es $O(n^3)$.

Sin embargo, esta cota podría ser exagerada, porque el tamaño de muchos rangos es mucho menor que n . Como el tamaño del rango que va desde $i + 1$ hasta j es $j - i$, una contabilidad más precisa sería decir que el tiempo del algoritmo es

$$\sum_{0 \leq i \leq j \leq n} (j - i) = \frac{n(n+1)(n+2)}{6} = \Theta(n^3)$$

de modo que el tiempo que demora el algoritmo en realidad es cúbico.

Un algoritmo más eficiente

Supongamos que hemos calculado que la raíz óptima para un subárbol (i, j) es $k = R[i, j]$.

Imaginemos ahora que quitamos la llave de más a la derecha y obtenemos la nueva raíz óptima $R[i, j - 1]$. Se puede demostrar que, al ser el subárbol de la derecha más "liviano," la nueva raíz no puede estar a la derecha de k : o se mantiene donde mismo, o se mueve a la izquierda. Por lo tanto,

$$R[i, j - 1] \leq R[i, j]$$

Por un razonamiento análogo, se puede demostrar que

$$R[i + 1, j] \geq R[i, j]$$

Juntando ambos resultados, tenemos que para buscar la raíz óptima para (i, j) ya no es necesario buscar en el rango $i + 1 \leq k \leq j$, sino que basta buscar en

$$R[i, j - 1] \leq k \leq R[i + 1, j]$$

con los valores de borde $R[i, i + 1] = i + 1$.

Para la implementación de este algoritmo optimizado, debemos manejar aparte el caso $d = 1$:

```
import math
import numpy as np

def calculaABBOptimo2(p,q):
    n=len(q)-1
    C=np.zeros((n+1,n+1)) # Esto ya deja la diagonal en cero
    W=np.zeros((n+1,n+1))
    R=np.zeros((n+1,n+1),dtype=int)
    for i in range(0,n+1):
        W[i,i]=q[i]
    # Caso d=1
    for i in range(0,n):
        W[i,i+1]=W[i,i]+p[i+1]+q[i+1]
        C[i,i+1]=W[i,i+1]
        R[i,i+1]=i+1
    # Casos de d=2 en adelante
    for d in range(2,n+1):
        for i in range(0,n-d+1):
            j=i+d
            W[i,j]=W[i,j-1]+p[j]+q[j]
            C[i,j]=math.inf # +infinito
            for k in range(R[i,j-1],R[i+1,j]+1):
                m=W[i,j]+C[i,k-1]+C[k,j]
                if m<C[i,j]:
                    C[i,j]=m
                    R[i,j]=k # anotamos dónde se alcanza el min
    return (C[0,n],R)
```

```
p=[0,0.15,0.10,0.05,0.10,0.20]
q=[0.05,0.10,0.05,0.05,0.05,0.10]
(Copt,R)=calculaABBOptimo2(p,q)
print("Costo óptimo=", Copt)
print("Árbol óptimo=", ABBOptimo(R,0,5))
```

Costo óptimo= 2.35

Árbol óptimo= ((X1)X2(X3))X4(X5))

Análisis del algoritmo optimizado

En este nuevo algoritmo, el tamaño del rango que hay que examinar para llenar el casillero (i, j) es $R[i+1, j] - R[i, j-1] + 1$. Al sumar todos estos tamaños, todos los elementos del interior de la matriz R se cancelan, porque aparecen en la suma una vez con signo positivo

y una vez con signo negativo. Solo sobreviven diferencias entre elementos de los bordes. Como hay $\Theta(n)$ casilleros en los bordes y cada diferencia es a lo más n , la suma total es $O(n^2)$. El término “+1” aporta un término cuadrático a la suma total, y por lo tanto el tiempo total de ejecución del algoritmo es $\Theta(n^2)$.

Por lo tanto, con esta optimización el algoritmo cúbico se transformó en un algoritmo cuadrático.

Splay Trees

Los Splay Trees son una estrategia para garantizar que cualquier secuencia de m operaciones en un árbol que llega a tener tamaño n , comenzando con un árbol vacío, toma tiempo $O(m \log n)$.

Esto no garantiza que el costo de una operación individual no pueda tener costo $O(n)$, sino que el costo acumulado dividido por el número de operaciones da un promedio de $O(\log n)$ por operación. Se dice que una estructura de este tipo es eficiente en el sentido *amortizado*.

La idea básica de un splay tree es que cuando una llave es accedida, se haga una secuencia de rotaciones para llevarla hasta la raíz. Si estas rotaciones se hacen de a un nivel a la vez, el resultado es lo que se llama “move to root,” pero esa estrategia es vulnerable a que una secuencia de m operaciones bien escogidas pueda hacerla tener un costo acumulado de $O(mn)$.

La idea del “splaying” es hacer las rotaciones de a dos niveles a la vez. Supongamos que la llave accesada es k , y que las dos llaves siguientes hacia arriba son p (padre) y a (abuelo). Hay tres casos:

Caso zig-zag

En este caso, se hace la misma rotación doble de los árboles AVL:

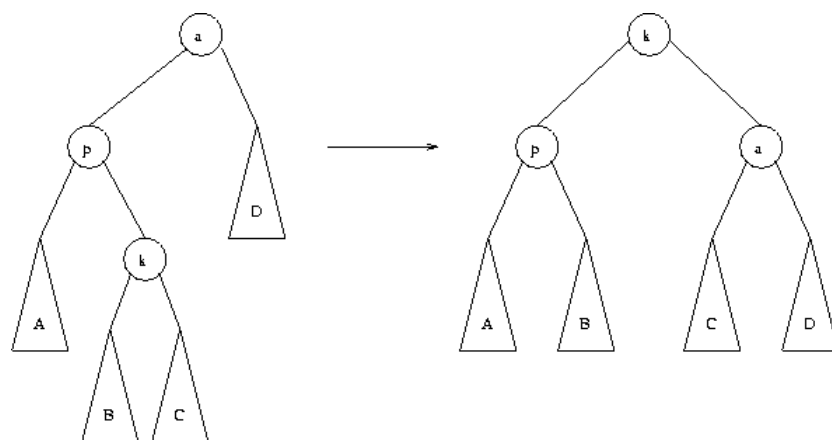


Figure 8o: splay1

Caso zig-zig

En este caso, el zig-zig se transforma en un zag-zag. Esto *no* es lo mismo que si se hubiesen hecho dos rotaciones simples:

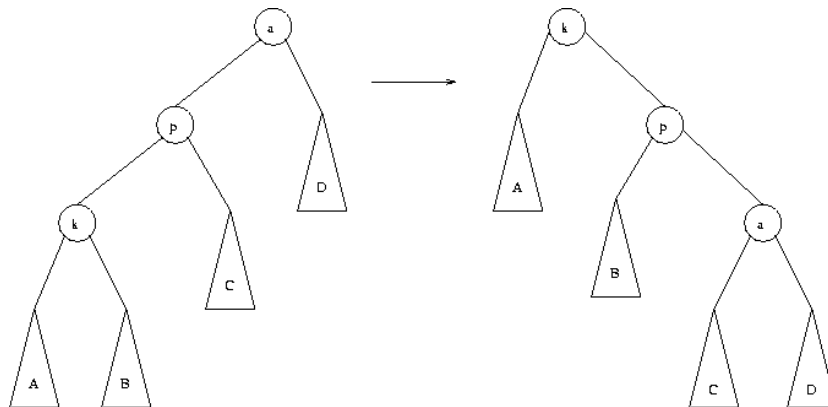


Figure 81: splay2

El análisis de los splay trees es complejo, porque debe considerar la estructura cambiante del árbol en cada acceso realizado. Por otra parte, los splay trees son más fáciles de implementar que un AVL dado que no hay que verificar una condición de balance.

Caso zig

Esto solo aplica cuando la llave accesada está a un paso de la raíz. En este caso, se hace una rotación simple para llevarla a la raíz del árbol.

Hashing

En esta sección vamos a ver un enfoque llamado *Hashing* que permite hacer búsquedas en tiempo esperado *constante*, en la medida que estemos dispuestos a no hacer uso del 100% de la memoria reservada para la estructura de datos.

Antes de ver hashing propiamente tal, veremos una estructura más sencilla y más eficiente, pero de utilidad más limitada, que nos servirá como motivación.

Bitmaps

Supongamos que queremos almacenar n números enteros no negativos, sabiendo que todos ellos están contenidos en el rango $[0..m-1]$, para algún valor de m .

Una forma muy simple de poder almacenar estos números y hacer búsquedas, inserciones y eliminaciones eficientes es utilizar un arreglo $a[0], \dots, a[m-1]$, marcando con un 1 los lugares correspondientes a los números que están presentes, y con 0 a los que no.

Por ejemplo, si $m = 8$ y el conjunto de números a almacenar es $\{1, 4, 7\}$, el arreglo contendría lo siguiente:

0	1	0	0	1	0	0	1
0	1	2	3	4	5	6	7

Figure 82: bitmap

Con esta representación, insertar la llave x es hacer $a[x] = 1$, para eliminarla hacemos $a[x] = 0$ y para buscarla preguntamos si $a[x] == 1$. Todas estas operaciones toman tiempo $\Theta(1)$.

¿Cómo es posible que se pueda violar la cota logarítmica que demostramos cuando vimos Búsqueda Binaria?

La razón es porque esa cota se demostró para cualquier algoritmo que funcione en base a comparaciones, y este algoritmo de Bitmaps no hace comparaciones de llaves, sino que las usa como subíndice, lo cual en realidad es equivalente a hacer aritmética con las llaves.

El motivo por el cual esta no es una solución muy usada es porque solo es aplicable si las llaves son enteros en un rango relativamente pequeño, porque de lo contrario el tamaño del arreglo a podría ser exageradamente grande.

El método de hashing trata de aproximar la eficiencia de los Bitmaps sin tener esa limitación sobre las llaves.

Funciones de hashing

Una función de hashing es una función que transforma una llave x en un valor $h(x) \in [0..m-1]$, para un valor dado de m , donde m es mucho menor que el tamaño del universo al cual pertenece x . La idea es utilizar la idea de un bitmap, utilizando $h(x)$ en lugar de x .

La función h debe ser de tipo *pseudoaleatorio*, esto es, debe distribuir sus valores uniformemente sobre el rango $[0..m-1]$, pero debe ser reproducible.

Considerando que toda llave x se codifica como una secuencia (posiblemente muy larga) de bits, y puede ser interpretada por lo tanto como un número entero, una familia de funciones de hashing puede obtenerse con la fórmula

$$h(x) = (cx \bmod p) \bmod m$$

donde c es una constante y p es un número primo. Distintos valores de estos parámetros producen distintas funciones de hashing.

Python tiene una función `hash()` que se vamos a utilizar en nuestros ejemplos.

```
m=int(input("m="))
x=input("x=")
print("h(x)=",hash(x)%m)
```

```

m=1000
x=Patricio Poblete
h(x)= 165

```

Dado un conjunto de n llaves, suponiendo por el momento que $n \leq m$, es posible que todas ellas sean mapeadas por la función h a valores distintos. Si fuera ese el caso, podríamos utilizar los $h(x)$ con un bitmap sin problemas. Sin embargo, en la práctica esto es muy improbable, y seguramente vamos a encontrarnos con llaves distintas x, y tales que $h(x) = h(y)$. Esto se llama una *colisión*.

El que alguna colisión sea probable que exista viene de la *paradoja de los cumpleaños*, que veremos a continuación. Por otra parte, en promedio no puede haber muchas colisiones, porque para un valor de k dado, el número promedio de llaves x que tienen $h(x) = k$ es $n/m \leq 1$, dado que suponemos que $h(x)$ distribuye las llaves uniformemente.

La Paradoja de los Cumpleaños

Supongamos que en una sala hay n personas. ¿Cuán grande debe ser n para que sea probable que existan dos o más personas con cumpleaños el mismo día? Más precisamente, ¿para qué valor de n la probabilidad de que haya alguna colisión de cumpleaños es $\geq 1/2$?

Resulta más fácil calcular la probabilidad opuesta d_n de que *no* haya colisión de cumpleaños. Esta probabilidad es igual a

$$d_n = \left(\frac{365}{365}\right) \left(\frac{364}{365}\right) \cdots \left(\frac{365 - n + 1}{365}\right)$$

```
%pylab inline
```

Populating the interactive namespace from numpy and matplotlib

```

n=range(1,101)
d=zeros(100)
d[0]=1
for k in range(1,100):
    d[k]=d[k-1]*(365-k)/365
plt.plot(n,d,label='$d_n$')

```

```
[<matplotlib.lines.Line2D at 0x117ea24e0>]
```

Para $n = 23$ se tiene $d_n = 0.4927 < 1/2$, por lo tanto para $n = 23$ la probabilidad de que exista alguna colisión es mayor que $1/2$.

Nótese que 23 es relativamente pequeño comparado con 365, lo cual indica, en general, que no es necesario que n sea muy grande en relación a m para que sea probable que exista alguna colisión. Por otra parte, el número total de colisiones no puede ser muy grande, y

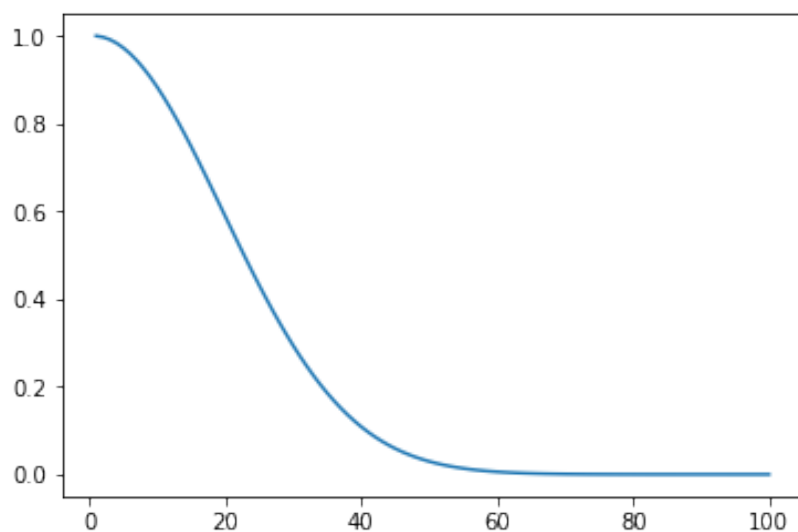


Figure 83: png

ambos hechos son los que debemos tener en consideración al aplicar la idea de hashing al diseño de diccionarios.

Métodos de resolución de colisiones

Dado que es inevitable que existan colisiones (aunque esperamos que sean pocas), debe haber alguna manera de resolver el problema que se produce cuando dos o más llaves son enviadas por la función de hashing al mismo casillero de la tabla.

Existen dos grandes familias de métodos:

- *Encadenamiento*: Utilizar punteros para enlazar los elementos que coinciden en su función de hashing.
- *Direccionamiento abierto (Open addressing)*: Utilizar una secuencia de funciones de hashing

Hashing con Encadenamiento

La idea en este método es que todos los elementos que caen en el mismo lugar de la tabla se enlazan en una lista secuencial, y en cada casillero se almacena el puntero al inicio de la lista respectiva.

Figure 84: encadenamiento

Para analizar la eficiencia de este método, recordemos que para un conjunto de n llaves hemos definido C_n y C'_n como respectivamente el número esperado de comparaciones de llaves en una búsqueda exitosa y en una búsqueda infructuosa.

Si suponemos que n llaves se lanzan al azar sobre una tabla de tamaño m , el número esperado de llaves por casillero es n/m , y eso es también el largo esperado de una lista aleatoria. En una búsqueda infructuosa hay que recorrer toda la lista, y por lo tanto tenemos que

$$C'_n = \frac{n}{m}$$

Para analizar el costo esperado de búsqueda exitosa C_n , consideremos las llaves en orden de inserción. Supongamos que hay k llaves en la tabla e insertamos una más. El largo esperado de la lista en que le toca almacenarse es k/m , y supongamos que la nueva llave se agrega al final de su lista. Si ahora hacemos una búsqueda exitosa de llave recién insertada, su costo esperado de búsqueda será $1 + k/m$. Para calcular el costo esperado de búsqueda exitosa sobre todas las llaves de la tabla, tenemos que sumar esos costos individuales y dividir por n :

$$C_n = \frac{1}{n} \sum_{0 \leq k \leq n-1} (1 + C'_k) = \frac{1}{n} \sum_{0 \leq k \leq n-1} \left(1 + \frac{k}{m}\right) = 1 + \frac{n-1}{2m}$$

Se define el *factor de carga de la tabla* α como el cuociente

$$\alpha = \frac{n}{m}$$

Se dice que una tabla está α -llena si el cuociente n/m se mantiene constante a medida que n y m crecen.

Para una tabla de hashing con encadenamiento que está α -llena, se tiene que

$$\begin{aligned} C'_n &= \alpha \\ C_n &\approx 1 + \frac{\alpha}{2} \end{aligned}$$

Esto ilustra un hecho que caracteriza a las tablas de hashing: cuando el factor de carga α se mantiene constante, los costos esperados de búsqueda son constantes, y no dependen de n y m por separado, sino de su cuociente α .

Una debilidad de muchas tablas de hashing es que su peor caso puede ser muy malo. En el caso de las tablas de hashing con encadenamiento, el peor caso se da cuando todas las llaves caen en el mismo casillero, y en ese caso los costos de búsqueda serían $\Theta(n)$, porque la estructura degenera a una simple lista lineal.

Sin embargo, una buena función de hashing hace que ese peor caso sea extremadamente improbable, de modo que en la práctica no hay problema en confiar en que el costo observado será cercano al costo esperado que predice el modelo.

La *eliminación* de una llave es sencilla, porque basta desenlazarla de la lista en que se encuentra.

Hashing con Direcccionamiento abierto

Otra forma de resolver colisiones es disponer de una *secuencia* de funciones de hashing $\{h_0(x), h_1(x), \dots\}$, donde cada valor es distinto de todos los anteriores. Para insertar una nueva llave primero se prueba en el casillero $h_0(x)$, si está ocupado se intenta en $h_1(x)$ y así sucesivamente. Para hacer una búsqueda, se sigue el mismo itinerario.

Lo anterior en realidad define una familia de métodos, los cuales dependen de cómo se defina esa secuencia de funciones de hashing. A continuación veremos los dos más importantes.

Linear Probing

Este es un método muy sencillo, que consiste en probar primero en el casillero $h(x)$, y si está ocupado, continuar buscando secuencialmente hacia la derecha, hasta encontrar un lugar libre. Si llegamos al extremo derecho de la tabla, continuamos en el extremo izquierdo, como si la tabla fuera circular.

Más formalmente, la secuencia de funciones de hashing se define como

$$\begin{aligned} h_0(x) &= h(x) \\ h_{i+1}(x) &= (h_i(x) + 1) \bmod m \end{aligned}$$

Figure 85: LinearProbing

Ejercicio 6.3 (Inserciones en una tabla de hashing con Linear Probing)

Suponga que se tiene una tabla de hashing con Linear Probing, de tamaño 10, inicialmente vacía, con la función de hashing $h(x) = x \bmod 10$ (por ejemplo, $h(64) = 4$). Muestre en la siguiente tabla el resultado de insertar (a mano) la siguiente secuencia de llaves:

34, 59, 45, 27, 14, 22, 75, 25

Utilizaremos la siguiente función para ver si su respuesta está correcta:

0	1	2	3	4	5	6	7	8	9

Figure 86: EjercicioLinearProbing

```
def chequea_tabla(lista):
    h=0
    for x in lista:
        h = (h*100+x) % 100000007
    print("OK" if h==60375958 else "Error")
```

En el siguiente recuadro reemplace la lista de ceros por la lista de los elementos resultantes en la tabla. Si un casillero queda vacío, escriba un cero.

```
chequea_tabla([0,0,0,0,0,0,0,0,0,0])
```

Error

El análisis de este método de Linear Probing es complicado, por lo que solo daremos los resultados.

Para una tabla α -llena, tenemos que

$$C'_n \approx \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right)$$

$$C_n \approx \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right)$$

En el caso de una tabla llena ($\alpha \rightarrow 1$), estas fórmulas no nos sirven, porque los denominadores tienden a cero. Se puede hacer obtener fórmulas precisas para ese caso, esto es, para el caso $n = m$ cuando la búsqueda es exitosa, o $n = m - 1$ para una búsqueda infructuosa o inserción (porque en ambos casos debe haber al menos un casillero vacío):

$$C'_{m-1} = \Theta(m)$$

$$C_m = \Theta(\sqrt{m})$$

Cuando una tabla de hashing con Linear Probing se va acercando a estar llena, el método se vuelve muy lento:

α	C_n	C'_n
0.6	1.75	3.63

α	C_n	C'_n
0.7	2.17	6.06
0.8	3.00	13.00
0.9	5.50	50.50
0.99	50.50	5000.50
0.999	500.50	500000.50

Si observamos la tabla a medida que se va llenando, veremos que empiezan a aparecer bloques (*clusters*) de casilleros ocupados.

Figure 87: clusters

Idealmente, querríamos que los *clusters* fueran pequeños, porque así una llave que viene llegando necesita recorrer un camino corto hasta encontrar un casillero vacío. Pero en realidad ocurre todo lo contrario. En efecto, si la función de hashing distribuye los elementos uniformemente dentro de la tabla, *la probabilidad que un cluster crezca es proporcional a su tamaño*. Esto implica que una mala situación se vuelve peor cada vez con mayor probabilidad. Sin embargo, este no es todo el problema. La situación empeora cuando dos clusters están separados solo por un casillero libre y ese casillero es ocupado por la llave entrante: ambos clusters se fusionan en uno mucho más grande.

Otro problema que surge con linear probing es conocido como *clustering secundario*: si al realizar la búsqueda de dos elementos en la tabla se encuentran con el mismo casillero ocupado, entonces toda la búsqueda subsiguiente es la misma para ambos elementos.

El peor caso de Linear Probing es muy malo, y también increíblemente improbable. Se da cuando todas las llaves colisionan en un solo casillero, y en ese caso la estructura degenera a una simple búsqueda secuencial, con costos de búsqueda $\Theta(n)$.

Respecto de la eliminación de elementos, es importante notar que no se puede eliminar un elemento y simplemente dejar su casillero vacío, puesto que las búsquedas terminarían en dicho casillero. Existen dos maneras para eliminar elementos de la tabla:

- Marcar el casillero como “eliminado,” pero sin liberar el espacio (excepto cuando se necesita para una inserción). Esto produce que las búsquedas puedan ser lentas incluso si el factor de carga de la tabla es pequeño.
- Eliminar el elemento, liberar el casillero y mover elementos dentro de la tabla hasta que un casillero “verdaderamente” libre sea encontrado. Implementar esta operación es complejo y costoso.

Hashing Doble

Este es un método que evita el problema del clustering secundario, reemplazándolo por clustering terciario, que afecta menos el desempeño.

La idea es generar la secuencia de funciones de hashing usando **dos** funciones de hashing independientes: $h(x)$ y $s(x)$ ("step"), tales que $h(x) \in [0..m-1]$ y $s(x) \in [1..m-1]$. Si el casillero $h(x)$ está ocupado, se intenta en $h(x) + s(x)$, $h(x) + 2s(x)$ y así sucesivamente, todo esto módulo el tamaño de la tabla.

Más precisamente,

$$h_0(x) = h(x)$$

$$h_{i+1}(x) = (h_i(x) + s(x)) \bmod m$$

Si $s(x)$ fuera mayor que 1 y divisor de m , la secuencia se repetiría después de $m/s(x)$ pasos, sin haber visitado todos los casilleros de la tabla. Para evitar esto, se debe escoger m como un número **primo**.

El análisis exacto de la eficiencia de este método es muy complicado, pero se obtienen buenos resultados usando modelos idealizados, ya sea suponiendo que la tabla se recorre usando muestreo sin reemplazo (llamado *uniform probing*) o muestreo con reemplazo (llamado *random probing*). Bajo estos modelos, se obtiene que los costos esperados de búsqueda para una tabla α -llena son:

$$C'_n \approx \frac{1}{1-\alpha}$$

$$C_n \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

Para una tabla llena, tenemos:

$$C'_{m-1} = \Theta(m)$$

$$C_m = \Theta(\log m)$$

La siguiente tabla muestra que este método se desempeña mejor que Linear Probing desde el punto de vista del número esperado de comparaciones:

α	C_n	C'_n
0.6	1.53	2.50
0.7	1.72	3.33
0.8	2.01	5.00
0.9	2.56	10.00
0.99	4.65	100.00
0.999	6.91	1000.00

El peor caso de Hashing Doble es muy malo, pero incluso más improbable que el de Linear Probing. Se da cuando todas las llaves colisionan en un solo casillero y además todas tienen el mismo valor de $s(x)$, en cuyo caso la estructura degenera a una simple búsqueda secuencial, con costos de búsqueda $\Theta(n)$.

Respecto de las eliminaciones, la única posibilidad es marcar el elemento como “eliminado,” con lo cual se considera ocupado para efecto de las búsquedas, pero libre para efectos de una inserción. Esto funciona, pero estos elementos contaminan la tabla y hacen que las búsquedas sean más lentas incluso cuando el factor de carga efectivo es bajo.

Cuando dos llaves colisionan, el método de inserción le da preferencia a la que llegó primero (a este método se le llama también *First-Come-First-Served*, o *FCFS*), pero en realidad cualquiera de las dos llaves podría ocupar el casillero en disputa, y la otra tendría que buscar en otro lugar.

A partir de esta observación, existen heurísticas para resolver el problema de las colisiones en hashing con direccionamiento abierto, como por ejemplo *Last-Come-First-Served* o *LCFS* hashing (el elemento que se mueve de casillero no es el que se inserta sino el que ya lo ocupaba) y *Robin Hood* hashing (el elemento que se queda en el casillero es aquel que se encuentre más lejos de su posición original), que si no cambian el promedio del costo de búsqueda con respecto al método original *FCFS*, sí disminuyen dramáticamente su varianza.

7

7 Ordenación

El problema de ordenar un conjunto de datos (por ejemplo, en orden ascendente) tiene gran importancia tanto teórica como práctica. En esta sección veremos principalmente algoritmos que ordenan mediante comparaciones entre llaves, para los cuales se puede demostrar una cota inferior que coincide con la cota superior provista por varios algoritmos. También veremos un algoritmo de otro tipo, que al no hacer comparaciones, no está sujeto a esa cota inferior.

Cota inferior

Supongamos que deseamos ordenar tres datos A , B y C . La siguiente figura muestra un árbol de decisión posible para resolver este problema. Los nodos internos del árbol representan comparaciones y los nodos externos representan salidas emitidas por el programa.

Figure 88: arbol-decision-ordenacion

Como se vio en el capítulo de búsqueda, todo árbol de decisión con H hojas tiene al menos altura $\log_2 H$, y la altura del árbol de decisión es igual al número de comparaciones que se efectúan en el peor caso.

En un árbol de decisión para ordenar n datos se tiene que $H = n!$, y por lo tanto se tiene que todo algoritmo que ordene n datos mediante comparaciones entre llaves debe hacer al menos $\log_2 n!$ comparaciones en el peor caso.

Usando la aproximación de Stirling, se puede demostrar que $\log_2 n! = n \log_2 n + \Theta(n)$, por lo cual la cota inferior es de $\Theta(n \log n)$.

Si suponemos que todas las posibles permutaciones resultantes son equiprobables, es posible demostrar también que el número promedio de comparaciones que cualquier algoritmo debe hacer es

también de $\Theta(n \log n)$.

Quicksort

Este método fue inventado por C.A.R. Hoare a comienzos de los '60s, y sigue siendo el método más eficiente para uso general.

Quicksort es un ejemplo clásico de la aplicación del principio de *dividir para reinar*. Su estructura es la siguiente:

- Primero se elige un elemento al azar, que se denomina el pivote.
- El arreglo a ordenar se reordena dejando a la izquierda a los elementos menores que el pivote, el pivote al medio, y a la derecha los elementos mayores que el pivote:

Figure 89: particion

- Luego cada sub-arreglo se ordena recursivamente.

La recursividad termina, en principio, cuando se llega a sub-arreglos de tamaño cero o uno, los cuales trivialmente ya están ordenados. En la práctica veremos que es preferible detener la recursividad antes de eso, para no desperdiciar tiempo ordenando recursivamente arreglos pequeños, los cuales pueden ordenarse más eficientemente usando Ordenación por Inserción, por ejemplo.

```
def quicksort(a):
    qsort(a,0,len(a)-1)

def qsort(a,i,j): # ordena a[i],...,a[j]
    if i<j: # quedan 2 o más elementos por ordenar
        k=particion(a,i,j)
        qsort(a,i,k-1)
        qsort(a,k+1,j)

def particion(a,i,j): # particiona a[i],...,a[j], retorna posición del pivote
    k=np.random.randint(i,j) # genera un número al azar k en rango i..j
    (a[i],a[k])=(a[k],a[i]) # mueve a[k] al extremo izquierdo
    # a[i] es el pivote
    s=i # invariante: a[i+1..s]<=a[i], a[s+1..t]>a[i]
    for t in range(s,j):
        if a[t+1]<=a[i]:
            (a[s+1],a[t+1])=(a[t+1],a[s+1])
            s=s+1
```

```
# mover pivote al centro
(a[i],a[s])=(a[s],a[i])
return s
```

```
def chequea_orden(a):
    print("Ordenado" if np.all(a[:-1]<=a[1:]) else "Desordenado")
```

```
import numpy as np
a = np.random.random(12)
print(a)
chequea_orden(a)
quicksort(a)
print(a)
chequea_orden(a)
```

```
[0.21939313 0.41324672 0.99870279 0.15140399 0.33302848 0.92304672
 0.11206401 0.63418302 0.64504804 0.50401626 0.2064129 0.81939607]
Desordenado
[0.11206401 0.15140399 0.2064129 0.21939313 0.33302848 0.41324672
 0.50401626 0.63418302 0.64504804 0.81939607 0.92304672 0.99870279]
Ordenado
```

Costo promedio de Quicksort

Si suponemos, como una primera aproximación, que el pivote siempre resulta ser la mediana del conjunto, entonces el costo de ordenar está dado (aproximadamente) por la ecuación de recurrencia

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

Esto tiene solución $T(n) = n \log_2 n$ y es, en realidad, el *mejor* caso de Quicksort.

Para analizar el tiempo promedio que demora la ordenación mediante Quicksort, observemos que el funcionamiento de Quicksort puede graficarse mediante un *árbol de partición*:

Figure 90: arbol-particion

Por la forma en que se construye, es fácil ver que el árbol de partición es un *árbol de búsqueda binaria*, y como el pivote es escogido al azar, entonces la raíz de cada subárbol puede ser cualquiera de los elementos del conjunto en forma equiprobable. En consecuencia, los árboles de partición y los árboles de búsqueda binaria tienen exactamente la misma distribución.

En el proceso de partición, cada elemento de los subárboles ha sido comparado contra la raíz (el pivote). Al terminar el proceso,

cada elemento ha sido comparado contra todos sus ancestros. Si sumamos todas estas comparaciones, el resultado total es igual al *largo de caminos internos*.

Usando todas estas correspondencias, tenemos que, usando los resultados ya conocidos para árboles, el número promedio de comparaciones que realiza Quicksort es de:

$$T(n) = 1.38n \log_2 n + \Theta(n)$$

Por lo tanto, Quicksort, en el caso esperado, corre en un tiempo proporcional a la cota inferior.

Peor caso de Quicksort

El peor caso de Quicksort se produce cuando el pivote resulta ser siempre el mínimo o el máximo del conjunto. En este caso la ecuación de recurrencia es

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

lo que tiene solución $T(n) = \Theta(n^2)$. Desde el punto de vista del árbol de partición, esto corresponde a un árbol en “zig-zag.”

Si bien este peor caso es extremadamente improbable si el pivote se escoge al azar, algunas implementaciones de Quicksort toman como pivote al primer elemento del arreglo (suponiendo que, al venir el arreglo al azar, entonces el primer elemento es tan aleatorio como cualquier otro). El problema es que si el conjunto viene en realidad ordenado, entonces caemos justo en el peor caso cuadrático.

Lo anterior refuerza la importancia de que el pivote se escoja al azar. Esto no aumenta significativamente el costo total, porque el número total de elecciones de pivote es $\Theta(n)$.

Mejoras a Quicksort

Quicksort puede ser optimizado de varias maneras, pero hay que ser muy cuidadoso con estas mejoras, porque es fácil que terminen empeorando el desempeño del algoritmo.

En primer lugar, es desaconsejable hacer cosas que aumenten la cantidad de trabajo que se hace dentro del “loop” de partición, porque este es el lugar en donde se concentra el costo $\Theta(n \log n)$.

Algunas de las mejoras que han dado buen resultado son las siguientes:

Quicksort con “mediana de 3” En esta variante, el pivote no se escoge como un elemento tomado al azar, sino que primero se extrae una muestra de 3 elementos, y entre ellos se escoge a la mediana de esa muestra como pivote.

Si la muestra se escoge tomando al primer elemento del arreglo, al del medio y al último, entonces lo que era el peor caso (arreglo

ordenado) se transforma de inmediato en mejor caso.

De todas formas, es aconsejable que la muestra se escoja al azar, y en ese caso el análisis muestra que el costo esperado para ordenar n elementos es

$$\frac{12}{7}n \ln n \approx 1.19n \log_2 n$$

Esta reducción en el costo se debe a que el pivote es ahora una mejor aproximación a la mediana. De hecho, si en lugar de escoger una muestra de tamaño 3, lo hiciéramos con tamaños como 7, 9, etc., se lograría una reducción aún mayor, acercándonos cada vez más al óptimo, pero con rendimientos rápidamente decrecientes.

Ejercicio 7.1 (Quicksort con mediana de 3)

Modifique el algoritmo Quicksort para que en la fase de partición utilice como pivote a la mediana de 3 elementos elegidos al azar.

Para esto, se recomienda modificar el algoritmo de partición de modo que seleccione 3 elementos al azar en el rango $i..j$ y los ordene, dejando en $a[i]$ el mínimo de los 3, en $a[i+1]$ la mediana de los 3 y en $a[j]$ el máximo de los 3. Luego, se aplica el algoritmo de partición ya conocido al segmento $a[i+2], \dots, a[j-1]$, con $a[i+1]$ como pivote. Al terminar, el pivote se mueve al centro y se retorna su posición.

Otro cambio que se debe hacer es tratar los casos de arreglos de tamaño 0, 1 y 2 como casos de borde, y aplicar qsort recursivo solo a arreglos de tamaño mayor o igual a 3.

En el siguiente recuadro escriba su algoritmo modificado y luego ejecute las instrucciones de prueba del recuadro siguiente.

```
def quicksort3(a):
    pass # reemplace esto por su programa
```

```
import numpy as np
a = np.random.random(12)
print(a)
chequea_orden(a)
quicksort3(a)
print(a)
chequea_orden(a)
```

```
[0.72391606 0.16663225 0.25313304 0.7389159  0.60516548 0.69447401
 0.56282635 0.44320546 0.81602      0.62844513 0.99977674 0.15667959]
Desordenado
[0.72391606 0.16663225 0.25313304 0.7389159  0.60516548 0.69447401
```

0.56282635 0.44320546 0.81602 0.62844513 0.99977674 0.15667959]
Desordenado

Uso de Ordenación por Inserción para ordenar sub-arreglos pequeños

Tal como se dijo antes, no es eficiente ordenar recursivamente sub-arreglos demasiado pequeños.

En lugar de esto, se puede establecer un tamaño mínimo M , de modo que los sub-arreglos de tamaño menor que esto se ordenan por inserción en lugar de por Quicksort.

Claramente debe haber un valor óptimo para M , porque si creciera indefinidamente se llegaría a un algoritmo cuadrático. Esto se puede analizar, y el óptimo es cercano a 10.

Como método de implementación, al detectarse un sub-arreglo de tamaño menor que M , se lo puede dejar simplemente sin ordenar, retornando de inmediato de la recursividad. Al final del proceso, se tiene un arreglo cuyos pivotes están en orden creciente, y encierran entre ellos a bloques de elementos desordenados, pero que ya están en el grupo correcto. Para completar la ordenación, entonces, basta con hacer una sola gran pasada de Ordenación por Inserción, la cual ahora no tiene costo $\Theta(n^2)$, sino $\Theta(nM)$, porque ningún elemento está a distancia mayor que M de su ubicación definitiva.

Ordenar recursivamente sólo el sub-arreglo más pequeño Un problema potencial con Quicksort es la profundidad que puede llegar a tener la recursividad. En el peor caso, ésta puede llegar a ser $\Theta(n)$.

Para evitar esto, se puede usar recursividad solo para la “mitad” más pequeña, y ordenar la otra “mitad” de manera iterativa. Con este enfoque, cada llamada recursiva se aplica a un sub-arreglo cuyo tamaño es a lo más la mitad del tamaño del arreglo a ordenar, de modo que si llamamos $S(n)$ a la profundidad de recursión, tenemos que

$$S(n) \leq 1 + S\left(\frac{n}{2}\right)$$

lo cual tiene solución $\log_2 n$, de modo que la profundidad de la recursión nunca es más que logarítmica.

```
def qsort(a,i,j): # ordena a[i],...,a[j]
    while i<j: # quedan 2 o más elementos por ordenar
        k=particion(a,i,j)
        if k-i<=j-k: #mitad izquierda es más pequeña
```



```

        qsort(a,i,k-1)
        i=k+1
    else:
        qsort(a,k+1,j)
        j=k-1

```

```

import numpy as np
a = np.random.random(6)
print(a)
quicksort(a)
print(a)

```

```

[0.66656624 0.76798384 0.87811887 0.74763235 0.30180187 0.82056957]
[0.30180187 0.66656624 0.74763235 0.76798384 0.82056957 0.87811887]

```

Un algoritmo de selección basado en Quicksort

Es posible modificar el algoritmo de Quicksort para seleccionar el k -ésimo elemento de un arreglo $a[0], \dots, a[n-1]$, esto es, el elemento que quedaría en la posición $a[k]$ si el arreglo se ordenara. Una solución trivial sería, precisamente, ordenar el arreglo y retornar ese elemento, pero la pregunta es si eso se puede hacer de manera más eficiente.

Una idea para resolver este problema es ejecutar Quicksort, pero en lugar de ordenar las dos mitades, hacerlo solo con aquella mitad en donde se encontraría el elemento buscado, o incluso no hacer nada si tenemos suerte y el pivote quedó justo en la posición $a[k]$.

```

def quickselect(a,k):
    assert k>=0 and k<len(a)
    qselect(a,k,0,len(a)-1)
    return a[k]

def qselect(a,k,i,j): # selecciona el elemento que quedaría en a[k]
                      # si se ordenara a[i],...,a[j] (i<=k<=j)
    if i<j:
        p=particion(a,i,j)
        if p!=k:
            if k<p:
                qselect(a,k,i,p-1)
            else:
                qselect(a,k,p+1,j)

```

```
import numpy as np
a = np.random.random(6)
print(a)
k=int(input("k="))
print(quickselect(a,k))
```

```
[0.19052461 0.578469 0.14702632 0.62380414 0.21373454 0.59978106]
k=4
0.5997810643996624
```

En realidad la recursividad de qselect es fácil de transformar en iteración, y una vez hecho eso ya no hay razón para que sea una función aparte:

```
def quickselect(a,k):
    assert k>=0 and k<len(a)
    i=0
    j=len(a)-1
    while i<j:
        p=particion(a,i,j)
        if p==k:
            break
        if k<p:
            j=p-1
        else:
            i=p+1
    return a[k]
```

```
import numpy as np
a = np.random.random(6)
print(a)
k=int(input("k="))
print(quickselect(a,k))
```

```
[0.28147833 0.06014974 0.56597127 0.42582274 0.66869136 0.11621771]
k=3
0.4258227414342961
```

El análisis del costo esperado de Quickselect es complicado, pero se puede demostrar que es $\Theta(n)$. Por otra parte, el peor caso se da cuando el pivote cae siempre en un extremo, y en ese caso el costo es $\Theta(n^2)$.

Un algoritmo de selección de tiempo lineal en el peor caso

La razón por la cual el algoritmo Quickselect puede demorar tiempo cuadrático en el peor caso es que el pivote puede resultar ser

siempre el mínimo o el máximo, o un elemento muy cercano a los extremos.

Esta debilidad se subsanaría si pudiéramos garantizar que los elementos que quedan a cada lado del pivote son siempre al menos una cierta fracción fija del total, digamos αn , para algún $\alpha \in (0, \frac{1}{2})$.

Una forma de lograrlo es la siguiente. Dividamos el conjunto completo de n elementos en grupos de tamaño 5, y encontremos la mediana de cada uno de estos pequeños conjuntos. Cada uno de ellos se puede visualizar como un orden parcial de la forma

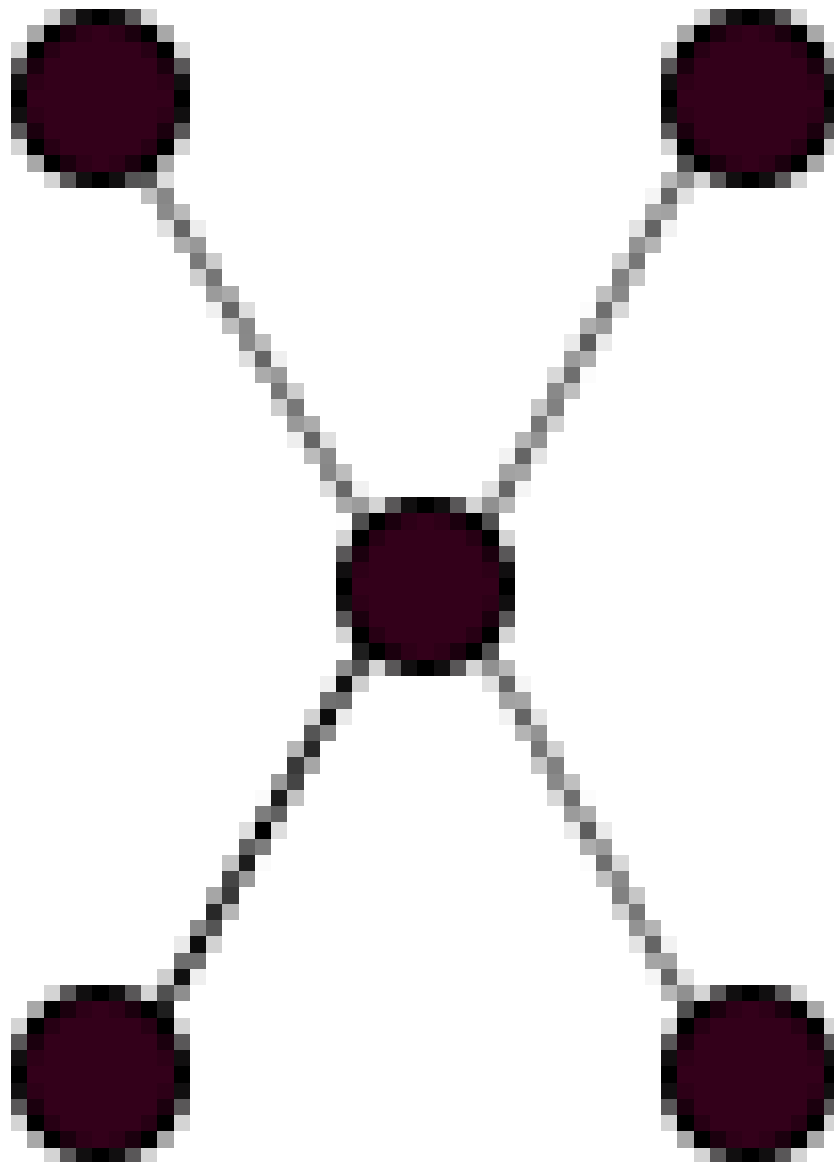


Figure 91: spider

donde los dos elementos de arriba son mayores que la mediana, y

ésta es mayor que los dos de abajo.

A continuación, aplicamos este mismo algoritmo, recursivamente, para encontrar la **mediana de las medianas**. Este elemento, que se muestra encerrado en un círculo en el diagrama siguiente, será nuestro pivote:

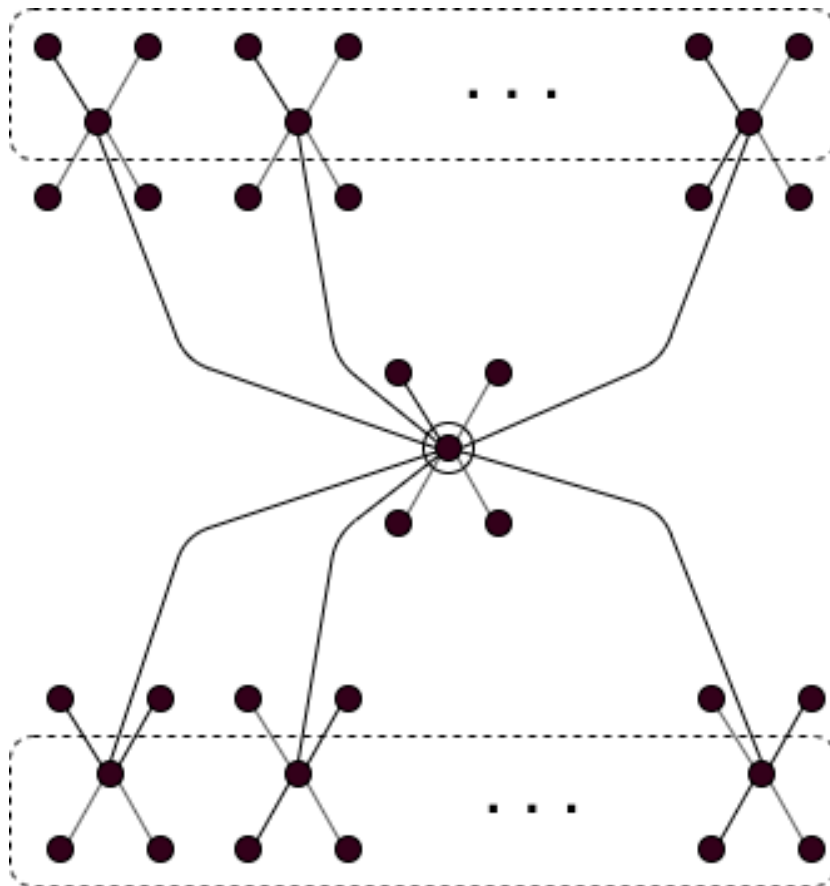


Figure 92: median-of-medians

Como se ve, todos los elementos encerrados en las líneas punteadas están garantizados de estar por sobre o por debajo del pivote, respectivamente. Por lo tanto, en el peor caso, podemos garantizar que al menos $\frac{3}{10}n$ elementos son menores que el pivote, otros $\frac{3}{10}n$ son mayores que él.

Por lo tanto, en el peor caso, la llamada recursiva que se hace para continuar el proceso se aplica a un conjunto de a lo más $\frac{7}{10}n$ elementos.

Si llamamos $T(n)$ al tiempo que demora este algoritmo en el peor caso, y si cn es el tiempo que demora encontrar las medianas de los $n/5$ subconjuntos, para alguna constante c , entonces tenemos que, en el peor caso,

$$T(n) = cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$$

Esta ecuación tiene solución lineal. En efecto, supongamos que existe una constante d tal que $T(n) = dn$. Reemplazando en la ecuación, tenemos

$$dn = cn + \frac{dn}{5} + \frac{7dn}{10}$$

de donde podemos despejar d y obtener $d = 10c$. Por lo tanto, $T(n) = 10cn$, y el algoritmo demora tiempo $\Theta(n)$ en el peor caso.

La razón por la cual la ecuación admite solución lineal es porque los coeficientes de n en las llamadas recursivas suman $\frac{1}{5} + \frac{7}{10} < 1$. Eso explica por qué no habría funcionado haber dividido el conjunto en grupos de tamaño 3. En ese caso, los coeficientes habrían sido $\frac{1}{3}$ y $\frac{2}{3}$ respectivamente, y su suma no sería menor que 1.

Heapsort

A partir de cualquier implementación de una cola de prioridad es posible obtener un algoritmo de ordenación. El esquema del algoritmo es:

- Comenzar con una cola de prioridad vacía.
- *Fase de construcción de la cola de prioridad*: Traspasar todos los elementos del conjunto que se va a ordenar a la cola de prioridad, mediante n inserciones.
- *Fase de ordenación*: Sucesivamente extraer el máximo n veces. Los elementos van apareciendo en orden decreciente y se van almacenando en el conjunto de salida.

Si aplicamos esta idea a las dos implementaciones simples de colas de prioridad, utilizando lista enlazada ordenada y lista enlazada desordenada, se obtienen los algoritmos de ordenación por Inserción y por Selección, respectivamente. Ambos son algoritmos cuadráticos, pero es posible que una mejor implementación lleve a un algoritmo más rápido. En el capítulo de *Pilas y Colas* vimos que una forma de obtener una implementación eficiente de colas de prioridad es utilizando una estructura de datos llamada *heap*.

Implementación de Heapsort

Al utilizar un heap como implementación de una cola de prioridad para construir un algoritmo de ordenación, se obtiene un algoritmo llamado *Heapsort*, para el cual resulta que tanto la fase de construcción de la cola de prioridad, como la fase de ordenación, tienen ambas costo $\Theta(n \log n)$, de modo que el algoritmo completo tiene ese mismo costo.

Por lo tanto, Heapsort tiene un orden de magnitud que coincide con la cota inferior, esto es, es óptimo incluso en el peor caso. Nótese que esto no era así para Quicksort, el cual era óptimo en promedio, pero no en el peor caso.

De acuerdo a la descripción de esta familia de algoritmos, daría la impresión de que en la fase de construcción del heap se requeriría un arreglo aparte para el heap, distinto del arreglo de entrada. De la misma manera, se requeriría un arreglo de salida aparte, distinto del heap, para recibir los elementos a medida que van siendo extraídos en la fase de ordenación.

En la práctica, esto no es necesario y basta con un sólo arreglo: todas las operaciones pueden efectuarse directamente sobre el arreglo de entrada.

En primer lugar, en cualquier momento de la ejecución del algoritmo, los elementos se encuentran particionados entre aquellos que están ya o aún formando parte del heap, y aquellos que se encuentran aún en el conjunto de entrada, o ya se encuentran en el conjunto de salida, según sea la fase. Como ningún elemento puede estar en más de un conjunto a la vez, es claro que, en todo momento, en total nunca se necesita más de n casilleros de memoria, si la implementación se realiza bien.

En el caso de Heapsort, durante la fase de construcción del heap, podemos utilizar las celdas de la izquierda del arreglo para ir “armando” el heap. Las celdas necesarias para ello se las vamos “quitando” al conjunto de entrada, el cual va perdiendo elementos a medida que van “trepar” en el heap. Al concluir esta fase, todos los elementos han sido insertados, y el arreglo completo es un solo gran heap.

Figure 93: heap-cons

En la fase de ordenación, se van extrayendo elementos del heap, con lo cual éste se contrae de tamaño y deja espacio libre al final, el cual puede ser justamente ocupado para ir almacenando los elementos a medida que van saliendo del heap (recordemos que van apareciendo en orden decreciente).

Figure 94: heap-ord

Optimización de la fase de construcción del heap

Como se ha señalado anteriormente, tanto la fase de construcción del heap como la de ordenación demoran tiempo $\Theta(n \log n)$. Esto es el mínimo posible (en orden de magnitud), de modo que no es posible mejorarlo significativamente.

Sin embargo, es posible modificar la implementación de la fase de construcción del heap para que sea mucho más eficiente.

La idea es invertir el orden de las “mitades” del arreglo, haciendo que el “input” esté a la izquierda y el “heap” a la derecha.

En realidad, si el “heap” está a la derecha, entonces no es realmente un heap, porque no es un árbol completo (le falta la parte superior), pero sólo nos interesa que en ese sector del arreglo se cumplan las relaciones de orden entre cada padre y sus hijos. En cada iteración, se toma el último elemento del “input” y se le “hunde” dentro del heap de acuerdo a su nivel de prioridad.

Figure 95: heap-cons2

Al concluir, se llega igualmente a un heap completo, pero el proceso es significativamente más rápido.

La razón es que, al ser “hundido,” un elemento paga un costo proporcional a su distancia al fondo del árbol. Dada las características de un árbol, la gran mayoría de los elementos están al fondo o muy cerca de él, por lo cual pagan un costo muy bajo. En un análisis aproximado, la mitad de los elementos pagan 0 (ya están al fondo), la cuarta parte paga 1, la octava parte paga 2, etc. Sumando todo esto, tenemos que el costo total está acotado por

$$n \sum_{i \geq 0} \frac{i}{2^{i+1}}$$

lo cual es igual a n .

En la práctica, al comenzar a construir el heap “de abajo hacia arriba,” no hace falta examinar los elementos que no tienen hijos, porque ellos cumplen automáticamente la relación de orden. Por lo tanto, basta comenzar a examinar los elementos desde el casillero $\lfloor \frac{n}{2} \rfloor - 1$ hacia atrás.

```
def hundir(a,j,n): # El elemento a[j] se hunde hasta su nivel de prioridad
    while 2*j+1<n: # mientras tenga al menos 1 hijo
        k=2*j+1 # el hijo izquierdo
        if k+1<n and a[k+1]>a[k]: # el hijo derecho existe y es mayor
            k+=1
        if a[j]>=a[k]: # tiene mejor prioridad que ambos hijos
            break
        (a[j],a[k])=(a[k],a[j]) # se intercambia con el mayor de los hijos
        j=k # bajamos al lugar del mayor de los hijos

def heapsort(a):
    n=len(a)
```

```

# Fase de construcción del heap
for i in range(n//2-1, -1, -1):
    hundir(a,i,n)
# Fase de ordenación
for i in range(n-1, -1, -1):
    (a[0],a[i])=(a[i],a[0])
    hundir(a,0,i)

```

```

import numpy as np
a = np.random.random(6)
print(a)
heapsort(a)
print(a)

```

```

[0.58123276 0.57772293 0.22690173 0.49691097 0.78430617 0.16308228]
[0.16308228 0.22690173 0.49691097 0.57772293 0.58123276 0.78430617]

```

Radix Sort

Los métodos anteriores operan mediante comparaciones de llaves, y están sujetos, por lo tanto, a la cota inferior $\Omega(n \log n)$. Veremos a continuación el método *Radix Sort* (llamado también *Bucket Sort*), que opera de una manera distinta, y logra ordenar el conjunto en tiempo lineal.

Supongamos que queremos ordenar n números, cada uno de ellos compuesto de k dígitos decimales. El siguiente es un ejemplo con $n = 10$, $k = 5$.

```

73895
93754
82149
99046
04853
94171
54963
70471
80564
66496

```

Para comenzar, veamos cómo podemos ordenar el conjunto, en una pasada, si la llave de ordenación fuera uno solo de los dígitos, por ejemplo el tercero de izquierda a derecha (lo mostramos separado para destacarlo):

```

99 0 46

```



```

82 1 49
94 1 71
70 4 71
66 4 96
80 5 64
93 7 54
73 8 95
04 8 53
54 9 63

```

La forma de hacerlo es la siguiente. Llamemos j a la posición del dígito mediante el cual se ordena. La ordenación se puede hacer utilizando una cola de entrada, que contiene al conjunto a ordenar, y un arreglo de 10 colas de salida, subindicadas de 0 a 9. Los elementos se van sacando de la cola de entrada y se van encolando en la cola de salida $Q[d_j]$, donde d_j es el j -ésimo dígito del elemento que se está transfiriendo.

Figure 96: bucket-pass

Al terminar este proceso, los elementos están separados por dígito. Para completar la ordenación, basta concatenar las k colas de salida y formar nuevamente una sola cola con todos los elementos.

Este proceso se hace en una pasada sobre los datos, y toma tiempo $\Theta(n)$.

Para ordenar el conjunto por las llaves completas, ejecutamos este proceso dígito por dígito, *de derecha a izquierda*, en cada pasada separando los elementos según el valor del dígito respectivo, luego recolectándolos para formar una sola cola, y realimentando el proceso con esos mismos datos. El conjunto completo queda finalmente ordenado (¡esto *no* es obvio!).

Como hay que realizar k pasadas y cada una de ellas toma tiempo $\Theta(n)$, el tiempo total es $\Theta(kn)$, que es el tamaño total del archivo de entrada (en bytes). Por lo tanto, la ordenación toma tiempo lineal en el tamaño de los datos.

El proceso anterior se puede generalizar para cualquier alfabeto, no sólo dígitos (por ejemplo, ASCII). Esto aumenta el número de colas de salida, pero no cambia sustancialmente el tiempo que demora el programa.

Ejercicio 7.2 (Radix Sort)

Ordene el conjunto

73895

93754
 82149
 99046
 04853
 94171
 54963
 70471
 80564
 66496

usando Radix Sort. Muestre el estado del conjunto después cada pasada (una pasada consiste en la separación en grupos de acuerdo a los dígitos presentes en la columna que se está procesando, seguida de la concatenación de los grupos resultantes). Recuerde que las columnas se procesan de derecha a izquierda.

El tiempo es lineal, pero ¿es $\Theta(n)$?

Como k es un parámetro ($k = 5$ en nuestro ejemplo), es tentador decir que $\Theta(kn) = \Theta(n)$, pero en realidad no es tan simple.

Si fijamos k , entonces el máximo número de elementos distintos que podemos generar es 10^k . Por lo tanto, $n \leq 10^k$, es decir, $k \geq \log_{10} n$.

Por lo tanto, k no puede ser constante, sino que tiene que crecer logarítmicamente con n , de modo que el algoritmo en realidad demora tiempo $\Omega(n \log n)$.

Archivos con records de largo variable

Si las líneas a ordenar no son todas del mismo largo, es posible alargarlas hasta completar el largo máximo, con lo cual el algoritmo anterior es aplicable. Pero si hay algunas pocas líneas desproporcionadamente largas y otras muy cortas, se puede perder mucha eficiencia.

Es posible, aunque no lo vamos a ver aquí, generalizar este algoritmo para ordenar líneas de largo variable sin necesidad de alargarlas. El resultado es que la ordenación se sigue realizando en tiempo proporcional al tamaño del archivo.

Mergesort

Si tenemos dos arreglos que ya están ordenados, podemos mezclarlos para formar un solo arreglo ordenado en tiempo proporcional a la suma de los tamaños de los dos arreglos.

Esto se hace leyendo el primer elemento de cada arreglo, copiando hacia un arreglo de salida al menor de los dos, y avanzando al siguiente elemento en el arreglo respectivo. Cuando uno de los dos

arreglos se termina, todos los elementos restantes del otro se copian hacia la salida. Este proceso se denomina “mezcla,” o bien “merge,” por su nombre en inglés.

Como cada elemento se copia sólo una vez, y con cada comparación se copia algún elemento, es evidente que el costo de mezclar los dos archivos es lineal.

Si bien es posible realizar el proceso de mezcla de dos arreglos contiguos *in situ*, el algoritmo es muy complicado y no resulta práctico. Por esta razón, el proceso se implementa generalmente copiando desde dos arreglos de entrada a uno de salida.

```
def merge(a,b):
    i=0
    j=0
    while i<len(a) or j<len(b):
        if j>=len(b) or (i<len(a) and a[i]<=b[j]):
            yield a[i]
            i=i+1
        else:
            yield b[j]
            j=j+1
```

```
a = [24,43,55,57,88,91]
b = [10,17,40,61,70,76]
print(a)
print(b)
c=[x for x in merge(a,b)]
print(c)
```

```
[24, 43, 55, 57, 88, 91]
[10, 17, 40, 61, 70, 76]
[10, 17, 24, 40, 43, 55, 57, 61, 70, 76, 88, 91]
```

Usando esta idea en forma reiterada, es posible ordenar un conjunto. Una forma de ver esto es recursivamente, de manera “*top-down*,” usando “dividir para reinar.”

```
def mergesort(a):
    n=len(a)
    if n>1:
        mergesort(a[0:n//2])
        mergesort(a[n//2:n])
        a[0:n]=[x for x in merge(a[0:n//2],a[n//2:n])]
```

```
import numpy as np
a = np.random.random(6)
print(a)
mergesort(a)
print(a)
```

```
[0.98208828 0.87903465 0.49512148 0.9665445 0.75927717 0.20751891]
[0.20751891 0.49512148 0.75927717 0.87903465 0.9665445 0.98208828]
```

El tiempo total está dado aproximadamente por la ecuación de recurrencia

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

la cual tiene solución $\Theta(n \log n)$, de modo que el algoritmo resulta ser óptimo.

Esto mismo se puede implementar en forma no recursiva, “*bottom-up*,” agrupando los elementos de a dos y mezclándolos para formar pares ordenados. Luego mezclamos pares para formar cuádruplas ordenadas, y así sucesivamente hasta mezclar las últimas dos mitades y formar el conjunto completo ordenado. Como cada “ronda” tiene costo lineal y se realizan $\log n$ rondas, el costo total es $\Theta(n \log n)$.

“*Runs*”

En el enfoque “*bottom-up*” no recursivo, en lugar de comenzar mezclando elementos individuales para formar pares, etc., podemos comenzar detectando “*runs*” (corridas), que son secuencias de elementos consecutivos lo más largas posibles que ya vengan ordenados. A partir de ahí, mezclamos “*runs*” consecutivas y lo seguimos haciendo hasta que quede solo una gran “*run*,” que es el arreglo completo ordenado.

Mergesort para ordenamiento externo

La idea de Mergesort es la base de la mayoría de los métodos de ordenamiento externo, esto es, métodos que ordenan conjuntos almacenados en archivos muy grandes, en donde no es posible copiar todo el contenido del archivo a memoria para aplicar alguno de los métodos estudiados anteriormente.

En las implementaciones prácticas de estos métodos, se utiliza el enfoque no recursivo, optimizado usando las siguientes ideas:

- No se comienza con elementos individuales para formar pares ordenados, sino que se generan archivos ordenados lo más grandes posibles. Para esto, el archivo de entrada se va leyendo por trozos a memoria y se ordena mediante Quicksort, por ejemplo.
- En lugar de mezclar sólo dos archivos se hace una mezcla múltiple (con k archivos de entrada. Como en cada iteración hay k candidatos a ser el siguiente elemento en salir, y siempre hay que

extraer al mínimo de ellos y sustituirlo en la lista de candidatos por su sucesor, la estructura de datos apropiada para ello es un heap. En caso que no baste con una pasada de mezcla múltiple para ordenar todo el archivo, el proceso se repite las veces que sea necesario.

Al igual que en los casos anteriores, el costo total es $\Theta(n \log n)$.

8

8 Búsqueda en Texto

La búsqueda de patrones en un texto es un problema muy importante en la práctica. Sus aplicaciones en computación son variadas, como por ejemplo la búsqueda de una palabra en un archivo de texto o problemas relacionados con biología computacional, en donde se requiere buscar patrones dentro de una secuencia de ADN, la cual puede ser modelada como una secuencia de caracteres (el problema es más complejo que lo descrito, puesto que se requiere buscar patrones en donde ocurren alteraciones con cierta probabilidad, esto es, la búsqueda no es exacta).

En este capítulo se considerará el problema de buscar la ocurrencia de un patrón dentro de un texto. Se utilizarán las siguientes convenciones:

- n denotará el largo del texto en donde se buscará el patrón, es decir, el texto es $a_0a_1 \dots a_{n-1}$
- m denotará el largo del patrón a buscar, es decir el patrón es $b_0b_1 \dots b_{m-1}$

Por ejemplo:

- Texto = “análisis de algoritmos”
- Patrón = “algo”

Algoritmo de fuerza bruta

Se alinea la primera posición del patrón con la primera posición del texto, y se comparan los caracteres uno a uno hasta que se acabe el patrón, en cuyo caso se encontró una ocurrencia del patrón en el texto, o hasta que se encuentre una discrepancia.

Si se detiene la búsqueda por una discrepancia, se desliza el patrón en una posición hacia la derecha y se intenta calzar el patrón nuevamente.

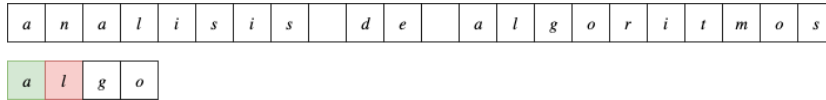


Figure 97: bruta1

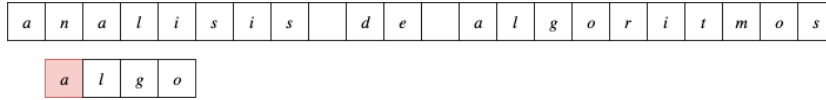


Figure 98: bruta2

El proceso se repite, siempre avanzando en una posición

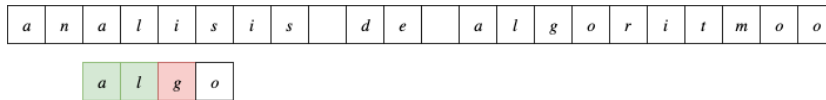


Figure 99: bruta3

hasta que finalmente se encuentre
o se agote el texto sin encontrar el patrón.

En el peor caso este algoritmo realiza $\Theta(mn)$ comparaciones de caracteres.

Si suponemos que todos los caracteres aparecen en forma equiprobable e independiente, en el caso esperado el método de fuerza bruta funciona mucho mejor. Si el alfabeto es de tamaño c , entonces en cada comparación la probabilidad de que dos caracteres sean iguales es $\frac{1}{c}$ y de que sean distintos es $1 - \frac{1}{c}$. A partir de cada posición, el número esperado de comparaciones que se efectúa hasta encontrar un descalce es

$$\frac{1}{1 - \frac{1}{c}} = \frac{c}{c - 1} = 1 + \frac{1}{c - 1}$$

Por lo tanto, en cada posición en que el patrón no está, el número esperado de comparaciones es una constante poco mayor que 1, por lo tanto el costo total de la búsqueda es $\Theta(n)$.

Algoritmo Knuth-Morris-Pratt (KMP)

Suponga que se está comparando el patrón y el texto en una posición dada, cuando se encuentra una discrepancia.

Sea x la parte del patrón que calza con el texto, e y la correspondiente parte del texto, y suponga que el largo de x es j . El algoritmo de fuerza bruta mueve el patrón una posición hacia la derecha, sin embargo, esto puede o no puede ser lo correcto en el sentido que los primeros $j - 1$ caracteres de x pueden o no pueden calzar los últimos $j - 1$ caracteres de y .

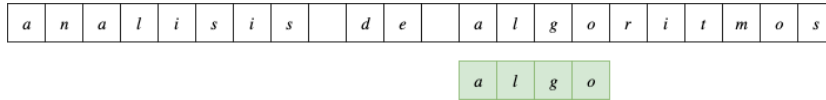


Figure 100: brutat4

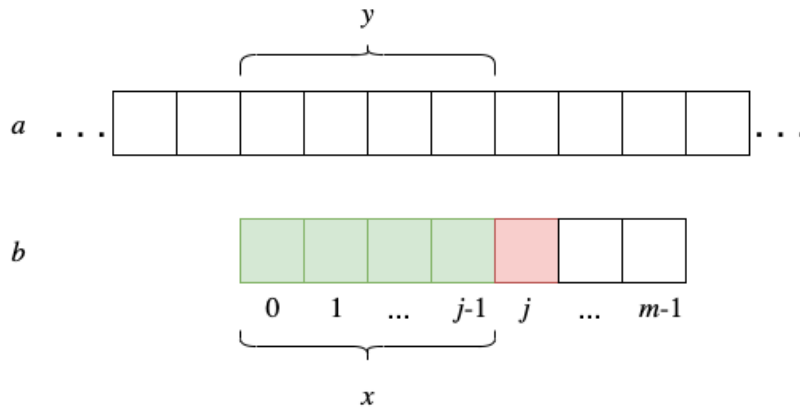


Figure 101: kmp1

La observación clave que realiza el algoritmo Knuth-Morris-Pratt (en adelante KMP) es que x es igual a y , por lo que la pregunta planteada en el párrafo anterior puede ser respondida mirando solamente el patrón de búsqueda, lo cual permite precalcular la respuesta y almacenarla en una tabla.

Por lo tanto, si deslizar el patrón en una posición no funciona, se puede intentar deslizarlo en $2, 3, \dots$, hasta j posiciones.

Se define la *función de fracaso* (*failure function*) del patrón como:

$$f(j) = \max\{i \mid 0 \leq i < j \wedge b_0 \dots b_{i-1} = b_{j-i-1} \dots b_{j-1}\}$$

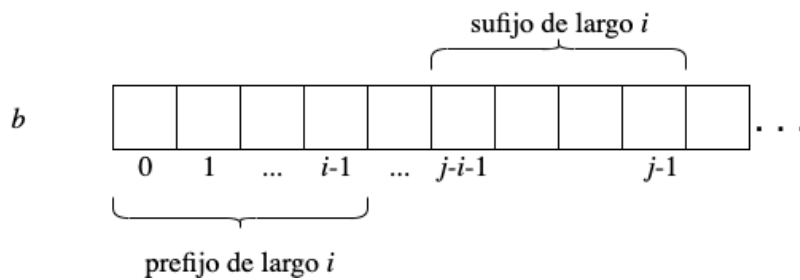


Figure 102: kmp2

Intuitivamente, $f(j)$ es el largo del mayor prefijo de x que además es sufijo del mismo x . Por ejemplo, si el patrón es "aabaaa", entonces la función de fracaso es

j	1	2	3	4	5	6
$f(j)$	0	1	0	1	2	2

Si se detecta una discrepancia entre el patrón y el texto cuando se trata de calzar b_j , se desliza el patrón de manera que $b_{f(j)-1}$ se encuentre donde b_{j-1} se encontraba, y se intenta calzar nuevamente.

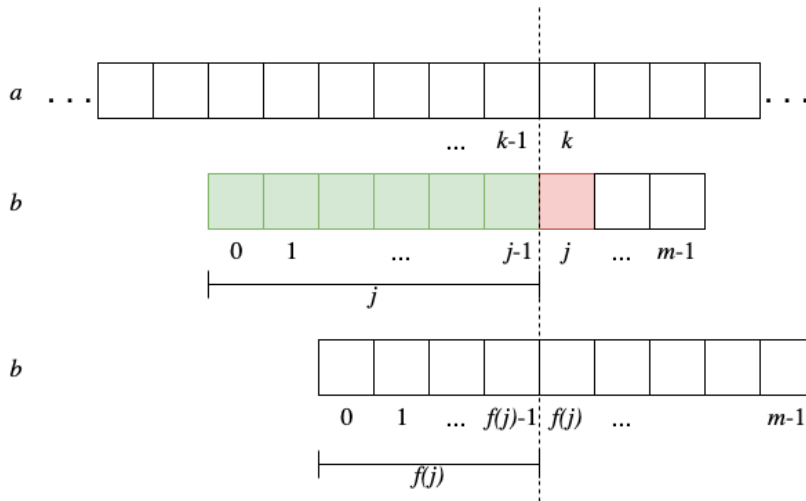


Figure 103: kmp3

En el siguiente ejemplo mostramos la ejecución de una búsqueda:

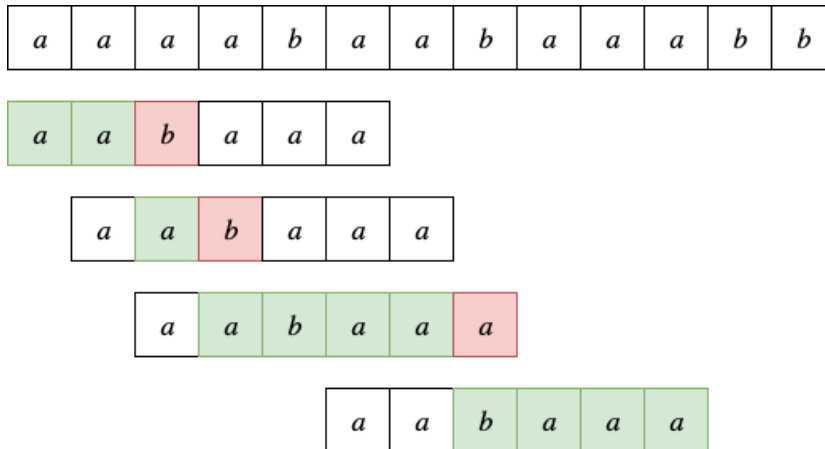


Figure 104: kmp4

Suponiendo que se tiene la función $f(j)$ precalculado, la implementación del algoritmo KMP es la siguiente:

```
def kmp(a,b,f): # busca b dentro de a, retorna None (fracaso) o posición del calce
    n=len(a)
    m=len(b)
    j=0
    for k in range(0,n):
        if j==m:
            return k-m # éxito
        while j>0 and a[k]!=b[j]:
            j=f[j]
        if a[k]==b[j]:
            j=j+1
    return None # fracaso
```

```
f=[0,0,1,0,1,2,2]
print(kmp("aaaabaabaabb", "aabaaa", f))
print(kmp("aaaabaabaabb", "abbaaa", f))
```

5

None

El tiempo de ejecución de este algoritmo no es difícil de analizar, pero es necesario ser cuidadoso al hacerlo. Dado que se tienen dos ciclos anidados, se puede acotar el tiempo de ejecución como el número de veces que se ejecuta el ciclo externo (menor o igual a n) multiplicado por el número de veces que se ejecuta el ciclo interno (menor o igual a m), lo que da una cota superior a $O(mn)$, ¡que es igual a lo que demora el algoritmo de fuerza bruta!

Sin embargo, el análisis descrito es pesimista. Observemos que el número total de veces que el ciclo interior es ejecutado es menor o igual al número de veces que se puede decrementar j , dado que $f(j) < j$. Pero j comienza desde cero y es siempre mayor o igual que cero, por lo que dicho número es menor o igual al número de veces que j es incrementado, el cual es menor que n . Por lo tanto, el tiempo total de ejecución es $\Theta(n)$ en el peor caso. Por otra parte, k nunca es decrementado, lo que implica que el algoritmo nunca se devuelve en el texto.

Cálculo de la función de fracaso

Queda por resolver el problema de definir la función de fracaso, $f(j)$. Esto se puede realizar inductivamente. Para empezar, $f(1) = 0$ por definición. Para calcular $f(j+1)$ suponga que ya se tienen almacenados los valores de $f(1), f(2), \dots, f(j)$. Se desea encontrar un i tal que el i -ésimo carácter del patrón sea igual al j -ésimo carácter del patrón.

Para esto se debe cumplir que $i = f(j)$. Si $b_i = b_j$, entonces

$f(j+1) = i+1$. En caso contrario, se reemplaza i por $f(i)$ y se verifica nuevamente la condición.

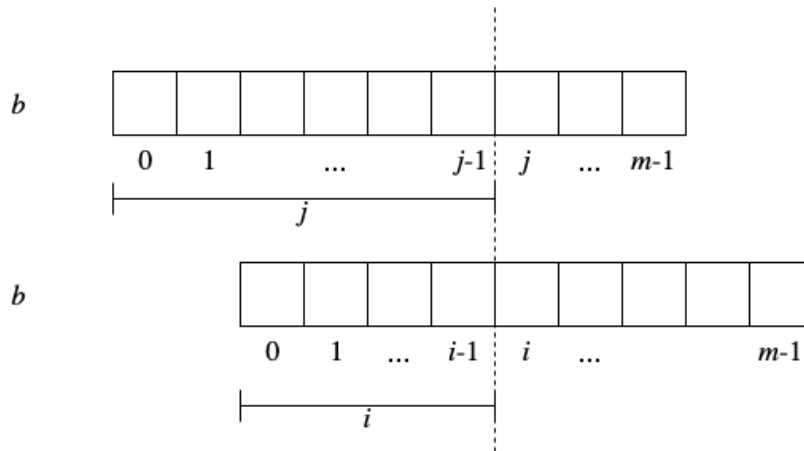


Figure 105: kmp5

El algoritmo resultante es el siguiente (note que es similar al algoritmo KMP):

```
def fracaso(b): # calcula y retorna función de fracaso para patrón b
    m=len(b)
    f=[0]*(m+1)
    for j in range(1,m):
        i=f[j]
        while i>0 and b[i]!=b[j]:
            i=f[i]
        if b[i]==b[j]:
            f[j+1]=i+1
        else:
            f[j+1]=0
    return f
```

```
print(fracaso("aabaaa"))
```

```
[0, 0, 1, 0, 1, 2, 2]
```

El tiempo de ejecución para calcular la función de fracaso puede ser acotado por los incrementos y decrementos de la variable i , y por lo tanto es $\Theta(m)$.

Por lo tanto, el tiempo total de ejecución del algoritmo en el peor caso, sumando el preprocesamiento del patrón más la búsqueda, es $\Theta(m+n)$.

Algoritmo de Boyer-Moore

Hasta el momento, los algoritmos de búsqueda en texto siempre comparan el patrón con el texto de izquierda a derecha. La idea del algoritmo de Boyer-Moore es comparar *de derecha a izquierda*: si hay una discrepancia en el último carácter del patrón y el carácter del texto no aparece en todo el patrón, entonces éste se puede deslizar m posiciones sin realizar ninguna comparación extra. En particular, no habría sido necesario comparar los primeros $m - 1$ caracteres del texto, lo cual indica que se podría realizar una búsqueda en el texto con menos de n comparaciones; sin embargo, si el carácter discrepante del texto se encuentra dentro del patrón, éste podría desplazarse en un número menor de espacios.

El método descrito es la base del algoritmo Boyer-Moore, del cual se estudiarán dos variantes: Horspool y Sunday.

Algoritmo Boyer-Moore-Horspool (BMH)

Supongamos que el último carácter del patrón de búsqueda b se encuentra alineado con el carácter a_k del texto. El algoritmo BMH compara el patrón con el texto **de derecha a izquierda**, y se detiene cuando se encuentra una discrepancia con el texto. Cuando esto sucede, se desliza el patrón de manera que la letra $c = a_k$ del texto que estaba alineada con la letra final del patrón, ahora quede alineada con un carácter anterior de b con el cual coincida, si dicho calce es posible, o si no, con b_{-1} , un carácter ficticio a la izquierda de b_0 (este es el mejor caso del algoritmo).

Para determinar el desplazamiento del patrón se define la función $g(c)$ como:

- $g(c) = i$ si i es el mayor subíndice en el rango $0 \leq i \leq m - 2$ tal que $b_i = c$. Nótese que si hay más de una ocurrencia de c , se toma la de más a la derecha.
- $g(c) = -1$ si c no aparece entre los primeros $m - 1$ caracteres del patrón.

En ambos casos se busca el carácter c solo dentro de los primeros $m - 1$ caracteres del patrón, excluyendo el último, para que el deslizamiento del patrón sea de al menos una posición.

Esta función sólo depende del patrón y se puede precalcular antes de realizar la búsqueda.

Ejemplo de Boyer-Moore-Horspool Supongamos que queremos buscar el patrón "datos" dentro de "estructuras de datos". La función $g(c)$ estaría dada por

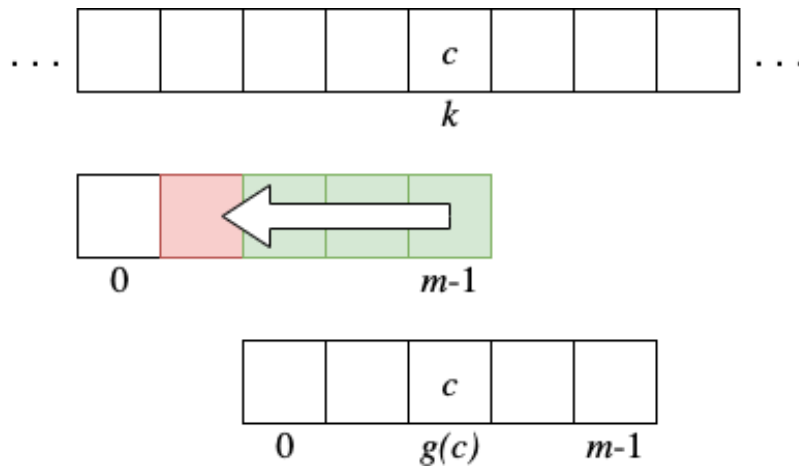


Figure 106: bmh1

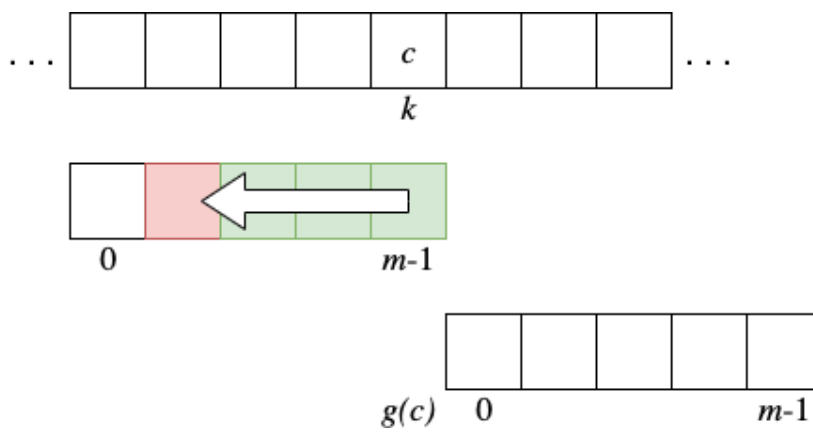


Figure 107: bmh2

c	"a"	"d"	"o"	"t"
$g(c)$	1	0	3	2

y $g(c) = -1$ para toda otra letra c .

El siguiente diagrama muestra el proceso de búsqueda:

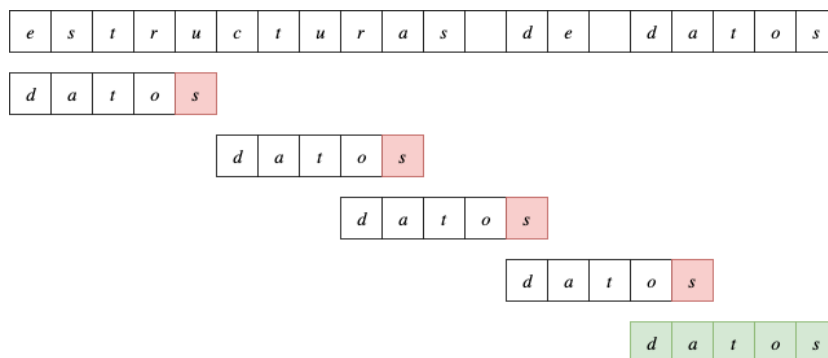


Figure 108: bmh3

Por simplicidad de programación, para implementar el algoritmo usaremos un diccionario de Python para la función g . En una implementación eficiente, esta función se implemntaría como un arreglo subindicado por la representación numérica de cada letra.

```
def construyeg(b): # construye un diccionario para representar la función g
    d={}
    for i in range(0,len(b)-1): # ignoramos el último caracter
        d[b[i]]=i
    return d

def bmh(a,b): # busca b dentro de a, retorna None (fracaso) o posición del calce
    g=construyeg(b) # Uso: g(c)=g.get(c,-1) para rellenar con -1 los valores faltantes
    n=len(a)
    m=len(b)
    k=m-1
    j=m-1
    while k<n:
        if j<0:
            return k-m+1
        if a[k-(m-1-j)]==b[j]:
            j=j-1
        else:
            k=k+(m-1-g.get(a[k],-1))
            j=m-1
    return None
```

```
print(bmh("estructuras de datos","datos"))
print(bmh("estructuras de datos","struct"))
print(bmh("estructuras de datos","gatos"))
```

15

1

None

Se puede demostrar que si el alfabeto tiene tamaño L y los caracteres aparecen con probabilidad uniforme y de manera independiente, entonces el tiempo promedio que demora BMH es

$$\Theta\left(n\left(\frac{1}{m} + \frac{1}{2L}\right)\right)$$

Para un alfabeto razonablemente grande, esto es aproximadamente $\Theta\left(\frac{n}{m}\right)$.

Algoritmo Boyer-Moore-Sunday (BMS)

El algoritmo BMH desliza el patrón basado en el símbolo del texto que corresponde a la posición del último carácter del patrón. Este siempre se desliza al menos una posición si se encuentra una discrepancia con el texto.

Es fácil ver que si se utiliza el carácter una posición más adelante en el texto como entrada de la función siguiente el algoritmo también funciona, pero en este caso es necesario considerar el patrón completo al momento de calcular los valores de la función siguiente. Esta variante del algoritmo es conocida como Boyer-Moore-Sunday (BMS).

Caso en que $c = a_{k+1}$ aparece dentro del patrón:

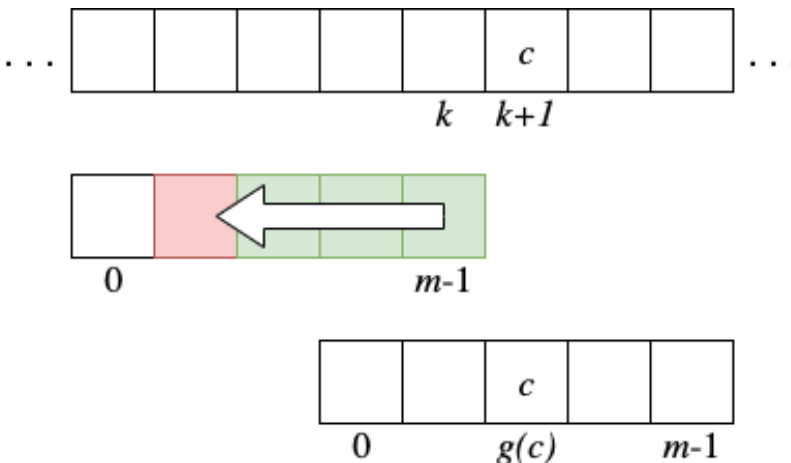


Figure 109: bms1

Caso en que $c = a_{k+1}$ no aparece dentro del patrón:

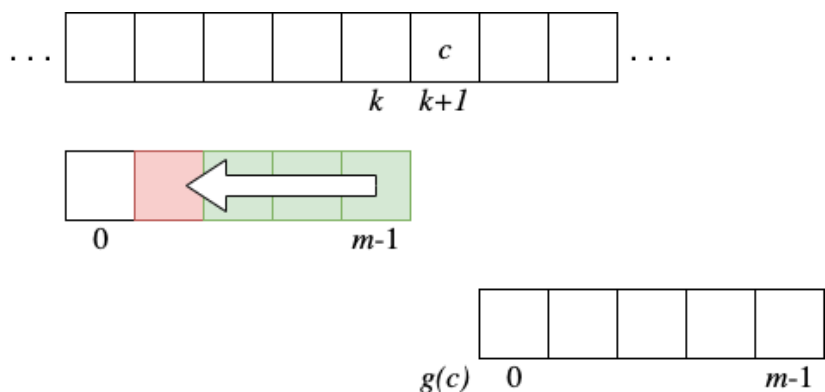


Figure 110: bms2

Ejemplo de Boyer-Moore-Sunday Supongamos que queremos buscar el patrón "datos" dentro de "estructuras de datos". La función $g(c)$ estaría dada por

c	"a"	"d"	"o"	"s"	"t"
$g(c)$	1	0	3	4	2

y $g(c) = -1$ para toda otra letra c .

El siguiente diagrama muestra el proceso de búsqueda:

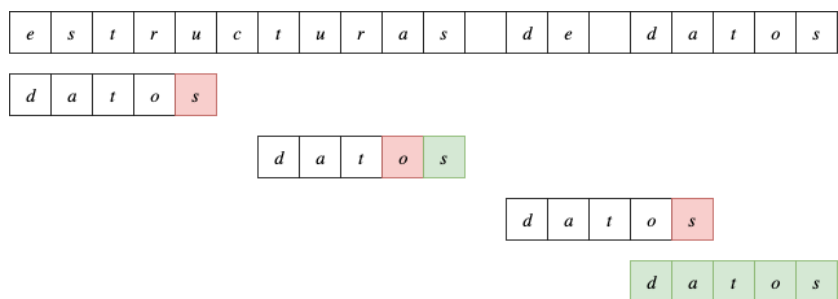


Figure 111: bms3

```
def construyeg(b): # construye un diccionario para representar la función g
    d={}
    for i in range(0,len(b)): # utilizamos hasta el último caracter
        d[b[i]]=i
    return d

def bms(a,b): # busca b dentro de a, retorna None (fracaso) o posición del calce
    g=construyeg(b) # Uso: g(c)=g.get(c,-1) para rellenar con -1 los valores faltantes
    n=len(a)
    m=len(b)
    k=m-1
    j=m-1
    while k<n:
        if j<0:
            return k-m+1
        if a[k-(m-1-j)]==b[j]:
            j=j-1
        else:
            if k>=-n-1:
                k=k-g.get(b[j],-1)
            else:
                return None
```

```
print(bms("estructuras de datos","datos"))  
print(bms("estructuras de datos","struct"))  
print(bms("estructuras de datos","gatos"))
```

15

1

None

9

9 Compresión de Datos

En esta sección veremos la aplicación de los árboles a la compresión de datos. Por compresión de datos entendemos cualquier algoritmo que reciba una cadena de datos de entrada y que sea capaz de generar una cadena de datos de salida cuya representación ocupa menos espacio de almacenamiento, y que permite -mediante un algoritmo de descompresión- recuperar total o parcialmente el mensaje recibido inicialmente. A nosotros nos interesa particularmente los algoritmos de compresión sin pérdida, es decir, aquellos algoritmos que permiten recuperar completamente la cadena de datos inicial.

Codificación de mensajes

Supongamos que estamos codificando mensajes en binario con un alfabeto de tamaño n . Para esto se necesitan $\lceil \log_2(n) \rceil$ bits por símbolo, si todos los códigos son de la misma longitud.

Ejemplo: Para el alfabeto A,...,Z de 26 letras se necesitan códigos de $\lceil \log_2(26) \rceil = 5$ bits

Problema: Es posible disminuir el número promedio de bits por símbolo?

Solución: Asignar códigos más cortos a los símbolos más frecuentes.

Un ejemplo clásico de aplicación de este principio es el código Morse:

A . -	H	O - - -	V . . . -
B - . . .	I . .	P . - - .	W . - -
C - . - .	J . - - -	Q - - . -	X - . . -
D - . .	K - . -	R . .	Y - . - -
E .	L . - . .	S . . .	Z - - . .
F . . - .	M - -	T -	
G - - .	N - .	U . . -	

Este código se puede representar mediante un árbol binario:

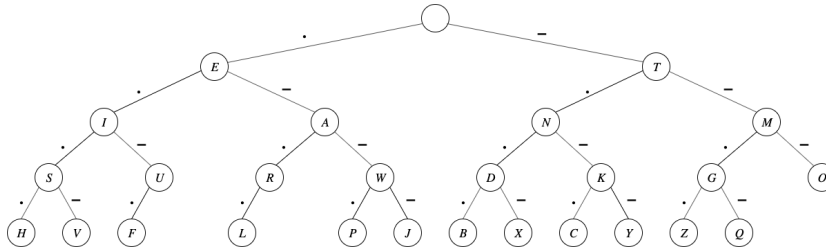


Figure 112: morse

Podemos ver en el árbol que letras de mayor probabilidad de aparición (en idioma inglés) están más cerca de la raíz, y por lo tanto tienen una codificación más corta que letras de baja frecuencia.

El código Morse tiene el problema que no es auto-delimitante. Por ejemplo, SOS y IAMS tienen la misma codificación. Eso requiere de un tercer símbolo (espacio) para separar las letras.

Se debe tener en cuenta que este problema se produce sólo cuando el código es de largo variable (como en Morse), pues en otros códigos de largo fijo (por ejemplo el código ASCII, donde cada carácter se representa por 8 bits) es directo determinar cuales elementos componen cada carácter.

La condición que debe cumplir una codificación para no presentar ambigüedades, es que la codificación de ningún carácter sea prefijo de otra. Esto se llama un *código libre de prefijos* y se puede visualizar mediante un árbol que sólo tiene información en las hojas, como por ejemplo:

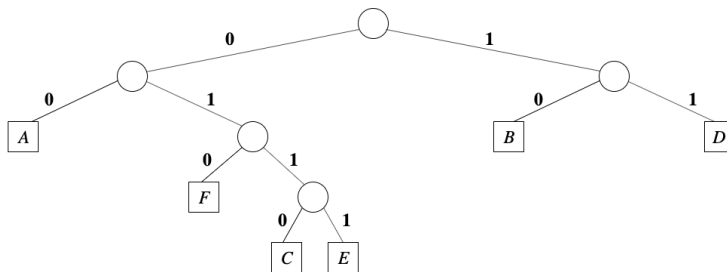


Figure 113: prefix-free

Como nuestro objetivo es obtener la secuencia codificada más corta posible, entonces tenemos que encontrar la codificación que en promedio use el menor largo promedio del código de cada letra.

Problema: Dado un alfabeto a_1, \dots, a_n tal que la probabilidad de que la letra a_i aparezca en un mensaje es p_i , encontrar un código libre de prefijos que minimice el largo promedio del código de una letra.

Supongamos que a la letra a_i se le asigna una codificación de largo t_i , entonces el largo esperado es:

$$\sum_i p_i t_i$$

es decir, el promedio ponderado de todas las letras por su probabilidad de aparición.

Ejemplo:

a_i	p_i	Código
A	0.30	00
B	0.25	10
C	0.08	0110
D	0.20	11
E	0.05	0111
F	0.12	010

$$\begin{aligned}
 \text{Costo esperado} &= 2(0.30 + 0.25 + 0.20) + 3(0.12) + 4(0.08 + 0.05) \\
 &= 2.075 + 0.36 + 0.52 \\
 &= 2.38 \text{ bits por símbolo}
 \end{aligned}$$

Entropía de Shannon

Shannon define la entropía del alfabeto como:

$$-\sum_i p_i \log_2(p_i)$$

El teorema de Shannon dice que el número promedio de bits esperable para un conjunto de letras y probabilidades dadas se aproxima a la entropía del alfabeto. Podemos comprobar esto en nuestro ejemplo anterior donde la entropía de Shannon es:

$$\begin{aligned}
 \text{Entropía} &= -(0.30 \log_2(0.30) + 0.25 \log_2(0.25) + 0.08 \log_2(0.08) \\
 &\quad + 0.2 \log_2(0.2) + 0.05 \log_2(0.05) + 0.12 \log_2(0.12)) \\
 &= 0.521 + 0.5 + 0.2915 + 0.4643 + 0.2160 + 0.3670 \\
 &\approx 2.36
 \end{aligned}$$

que es bastante cercano al costo esperado de 2.38 que calculamos anteriormente.

A continuación describiremos un algoritmo que nos permitan encontrar codificaciones que minimicen el costo total.

Algoritmo de Huffman

El algoritmo de Huffman permite construir un código libre de prefijos de costo esperado mínimo.

Inicialmente, comenzamos con n hojas desconectadas, cada una rotulada con una letra del alfabeto y con una probabilidad asociada.

Consideremos este conjunto de hojas como un bosque. El algoritmo, en pseudo código, es:

```
while N° de árboles del bosque > 1:
    Encontrar los 2 árboles de peso mínimo y
    unirlos con una nueva raíz que se crea para esto

    Arbitrariamente, rotulamos las dos
    ramas que salen como "0" y "1"

    Le damos a la nueva raíz un peso que es
    la suma de los pesos de sus subárboles
```

Ejemplo: Consideremos el siguiente conjunto de letras con sus probabilidades

a_i	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>
p_i	0.121	0.051	0.137	0.094	0.274	0.281	0.042

Entonces la construcción del árbol de Huffman es (los números en **negrita** indican los árboles con menor peso):

Figure 114: huffman

El costo esperado es de 2.53 bits por letra, mientras que una codificación de largo fijo (igual número de bits para cada símbolo) tendría un costo de 3 bits por letra.

El algoritmo de codificación de Huffman se basa en dos supuestos que le restan eficiencia:

- Supone que los caracteres son generados por una fuente aleatoria independiente, lo que en la práctica no es cierto. Por ejemplo, la probabilidad de encontrar una vocal después de una consonante es mucho mayor que la de encontrarla después de una vocal; después de una *q* es muy probable encontrar una *u*, etc.
- Debido a que la codificación se hace carácter a carácter, se pierde eficiencia al no considerar que hay grupos de caracteres más probables que otros.

Algoritmo de Lempel-Ziv

Una codificación que toma en cuenta los problemas señalados para la codificación de Huffman sería una donde no solo se consideraran caracteres uno a uno, sino que donde además se consideraran aquellas secuencias de alta probabilidad en el texto. Por ejemplo, en el texto:

aaabbaabaa

obtendríamos un mayor grado de eficiencia si además de considerar caracteres como *a* y *b*, también considerásemos la secuencia *aa* al momento de codificar.

Una generalización de esta idea es el algoritmo de Lempel-Ziv. Este algoritmo consiste en separar la secuencia de caracteres de entrada en bloques o secuencias de caracteres de distintos largos, manteniendo un diccionario de bloques ya vistos. Aplicando el algoritmo de Huffman a estos bloques y sus probabilidades, se puede sacar provecho de las secuencias que se repitan con más probabilidad en el texto. El algoritmo de codificación es el siguiente:

- 1.- Inicializar el diccionario con todos los bloques de largo 1
- 2.- Seleccionar el prefijo más largo del mensaje que calce con alguna secuencia *W* del diccionario y eliminar *W* del mensaje
- 3.- Codificar *W* con su índice en el diccionario
- 4.- Agregar *W* seguido del primer símbolo del próximo bloque al diccionario.
- 5.- Repetir desde el paso 2.

Ejemplo: Si el mensaje es

abbaabbaababbabaaabaabba

la codificación y los bloques agregados al diccionario serían (donde los bloques reconocidos son mostrados entre paréntesis y la secuencia agregada al diccionario en cada etapa es mostrada como un subíndice):

$(a)_{ab}(b)_{bb}(b)_{ba}(a)_{aa}(ab)_{abb}(ba)_{baa}(ab)_{aba}(abb)_{abba}(aa)_{aaa}(aa)_{aab}(baa)_{baab}(bb)_{bba}(a)$

Teóricamente, el diccionario puede crecer indefinidamente, pero en la práctica se opta por tener un diccionario de tamaño limitado. Cuando se llega al límite del diccionario, no se agregan más bloques.

Lempel-Ziv es una de las alternativas a Huffman. Existen varios otros algoritmos derivados de estos, como LZW (Lempel-Ziv-Welch),

que es usado en programas de compresión como el compress de UNIX.

10

10 Grafos

Un *grafo* es una estructura matemática que se utiliza para representar relaciones como las que hay entre las ciudades conectadas por caminos, los cursos con sus requisitos, las componentes conectadas en un circuito eléctrico, las páginas web vinculadas por enlaces, etc.

Definiciones básicas

Un grafo consiste de un conjunto V de *vértices* (también llamados *nodos*) y un conjunto E de *arcos*. El número de vértices se suele denotar n y el número de arcos como m (o a veces e).

Se dice que un grafo es *no dirigido* si sus arcos no tiene una dirección asociada. Por ejemplo,

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$
$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}\}$$

Un grafo es *dirigido* si sus arcos tienen una orientación. Por ejemplo:

$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{(v_1, v_2), (v_2, v_2), (v_2, v_3), (v_3, v_1), (v_3, v_4), (v_4, v_3)\}$$

Adicionalmente, los grafos pueden tener rótulos asociados a sus vértices o arcos. Estos rótulos pueden representar costos, longitudes, pesos, etc.

Representaciones de grafos en memoria

Un grafo se puede almacenar en memoria de distintas maneras, las cuales tienen distintos requerimientos de espacio, y ponen también distintas restricciones al tiempo de proceso.

Matriz de adyacencia

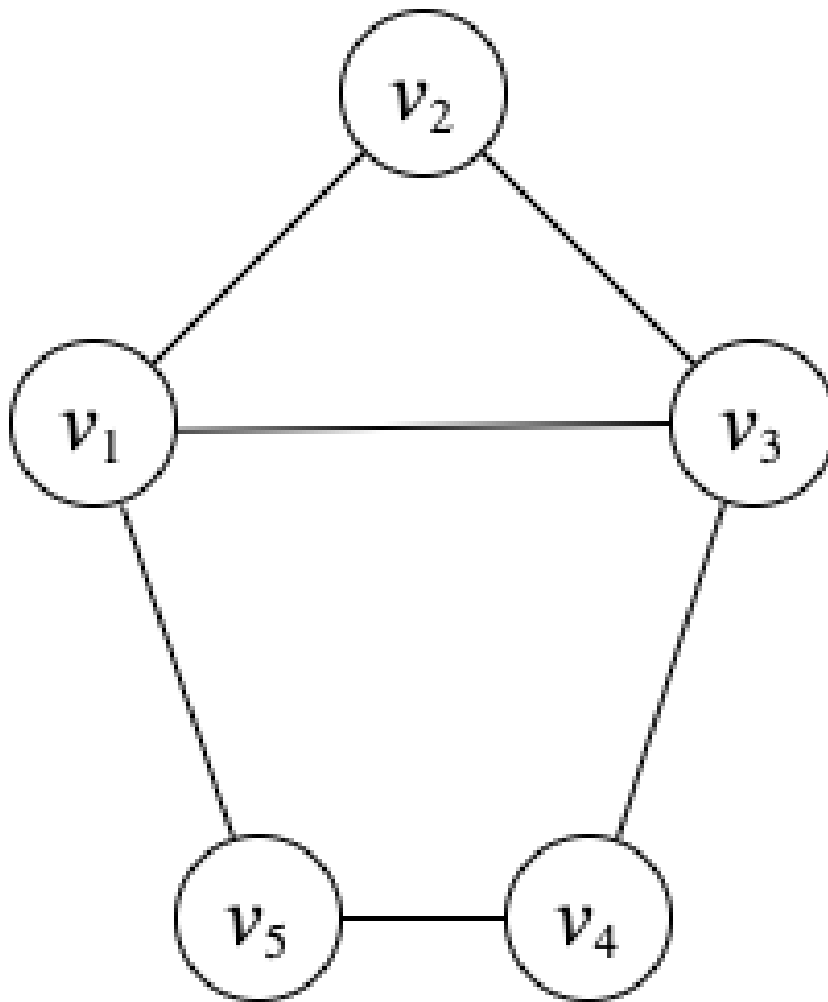
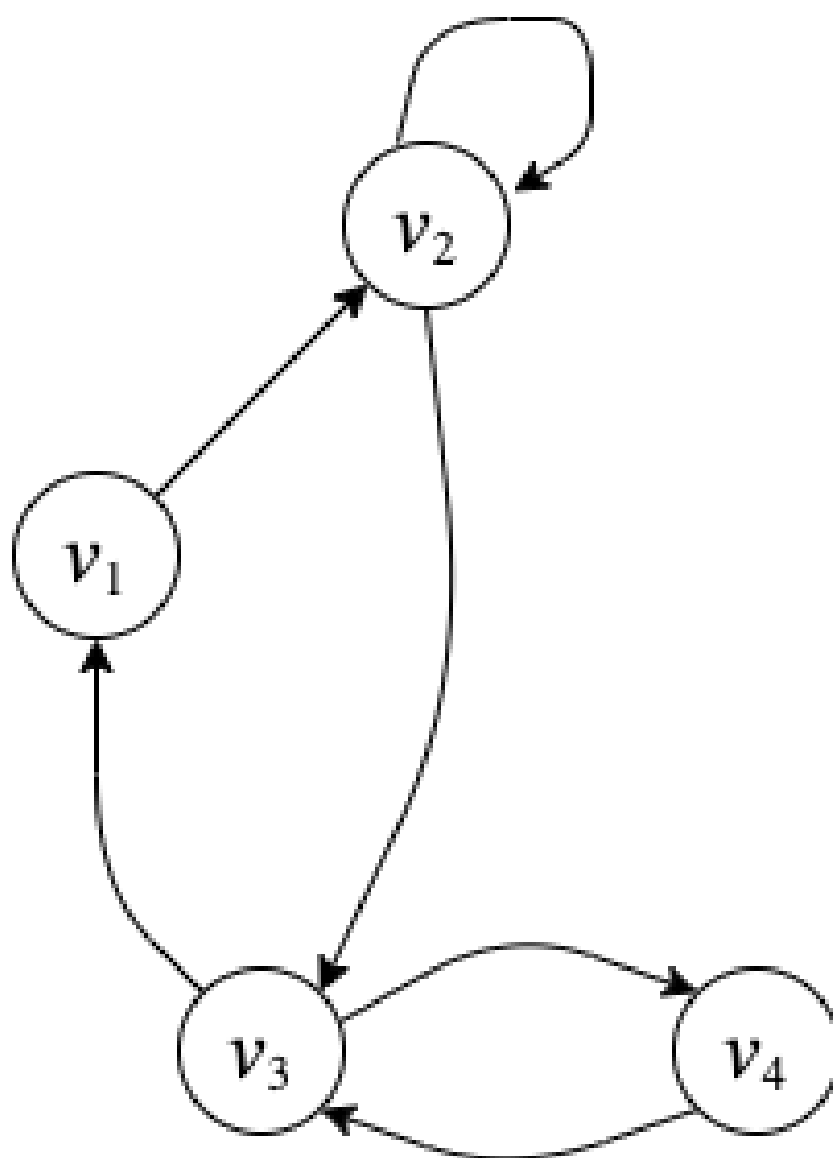


Figure 115: grafo-no-dirigido

Figure 116: grafo-dirigido



Un grafo se puede representar a través de una matriz A donde $A[i, j] = 1$ si hay un arco que conecta v_i con v_j , y 0 si no. La matriz de adyacencia de un grafo no dirigido es simétrica.

Una matriz de adyacencia permite determinar si dos vértices están conectados o no en tiempo constante, pero requieren $\Theta(n^2)$ bits de memoria. Esto puede ser demasiado para muchos grafos que aparecen en aplicaciones reales, en donde $m \ll n^2$. Otro problema es que se requiere tiempo $\Theta(n)$ para encontrar la lista de vecinos de un vértice dado.

Listas de adyacencia

Esta representación consiste en almacenar, para cada nodo, la lista de los nodos adyacentes a él (sus “vecinos”). Para el segundo ejemplo anterior, tenemos:

$$\begin{aligned}\text{vecinos}[v_1] &= [v_2] \\ \text{vecinos}[v_2] &= [v_2, v_3] \\ \text{vecinos}[v_3] &= [v_1, v_4] \\ \text{vecinos}[v_4] &= [v_3]\end{aligned}$$

Esto utiliza espacio $\Theta(m)$ y permite acceso eficiente a los vecinos, pero no permite acceso directo a los arcos.

Caminos, ciclos y árboles

Consideremos un grafo no dirigido.

Un *camino* es una secuencia de arcos en que el extremo final de cada arco coincide con el extremo inicial del siguiente en la secuencia. Por ejemplo, los arcos gruesos forman un camino desde v_2 a v_5 (o viceversa):

Un camino es *simple* si no se repiten vértices, excepto posiblemente el primero y el último.

Un *ciclo* es un camino simple y cerrado (esto es, en que el vértice inicial y final son el mismo). En el siguiente ejemplo los arcos gruesos forman un ciclo:

Un grafo que no tiene ciclos se dice que es *acíclico*.

Se dice que un grafo es *conexo* si para todo par de vértices del grafo existe un camino que los une. Si un grafo no es conexo, entonces estará compuesto por varias “islas,” cada una de las cuales se llama una *componente conexa*. Más precisamentem una componente conexa es un subgrafo conexo *maximal* (esto es, que no está estrictamente contenido dentro de un subgrafo conexo mayor).

Un *árbol* es un grafo que es *conexo* y *acíclico*. En el siguiente ejemplo, los arcos gruesos forman un árbol:

Si un árbol incluye todos los nodos de un grafo, se dice que es un

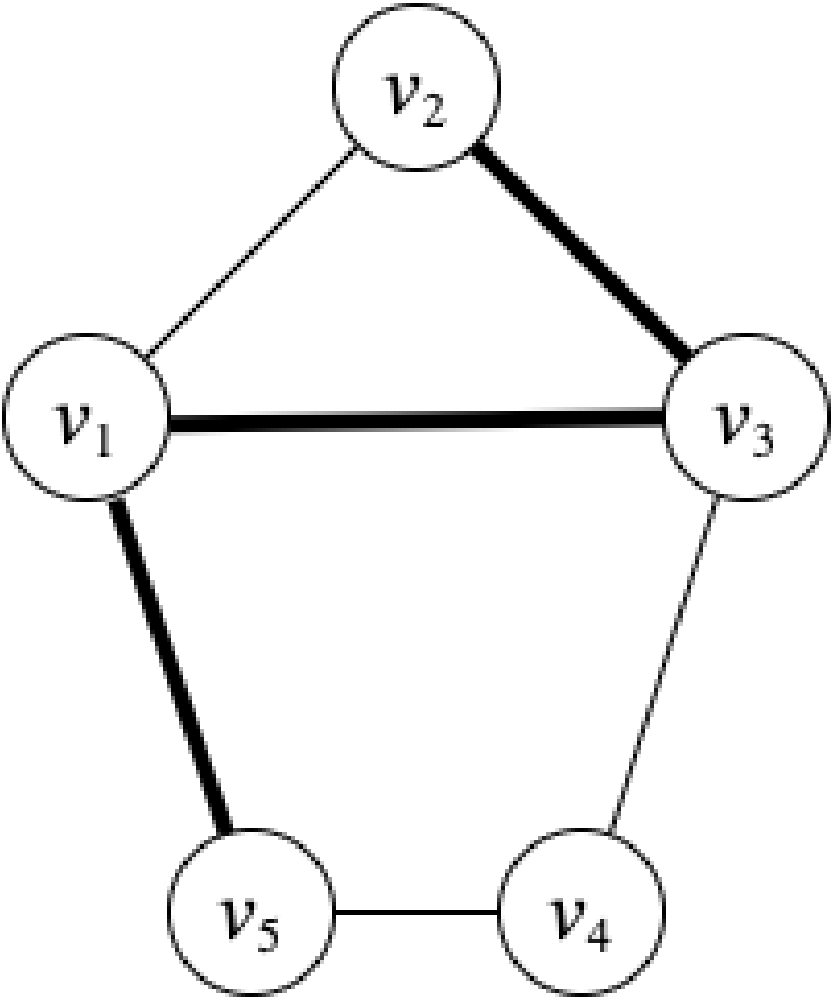


Figure 117: camino

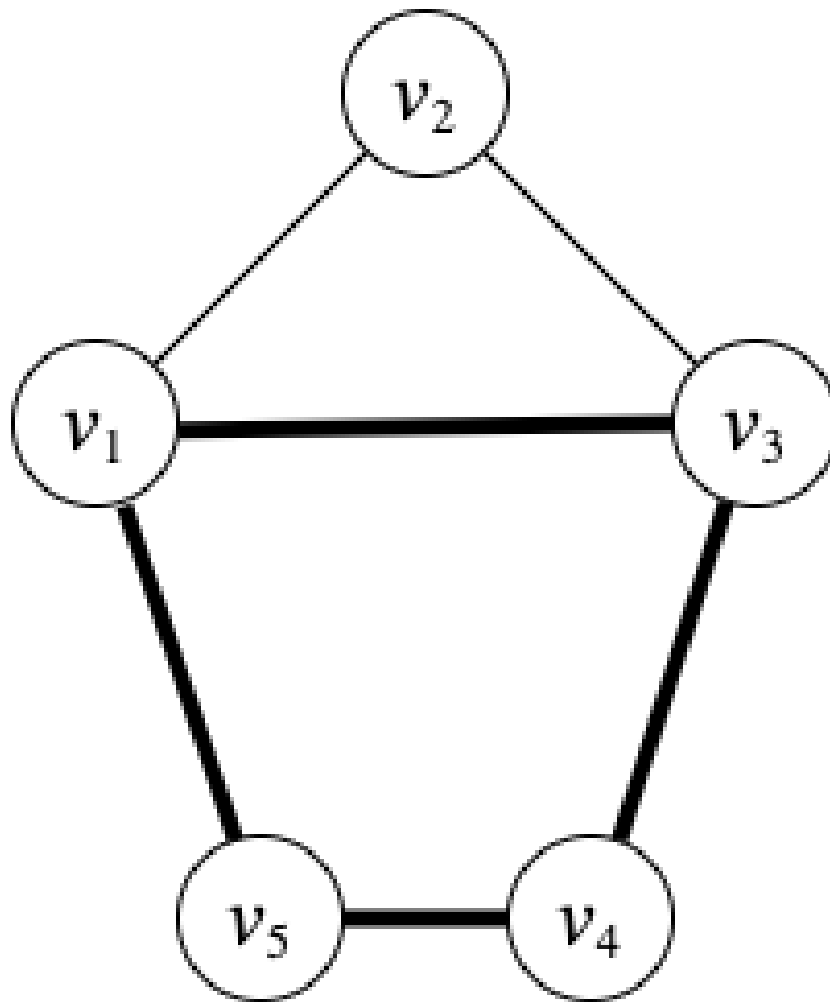


Figure 118: ciclo

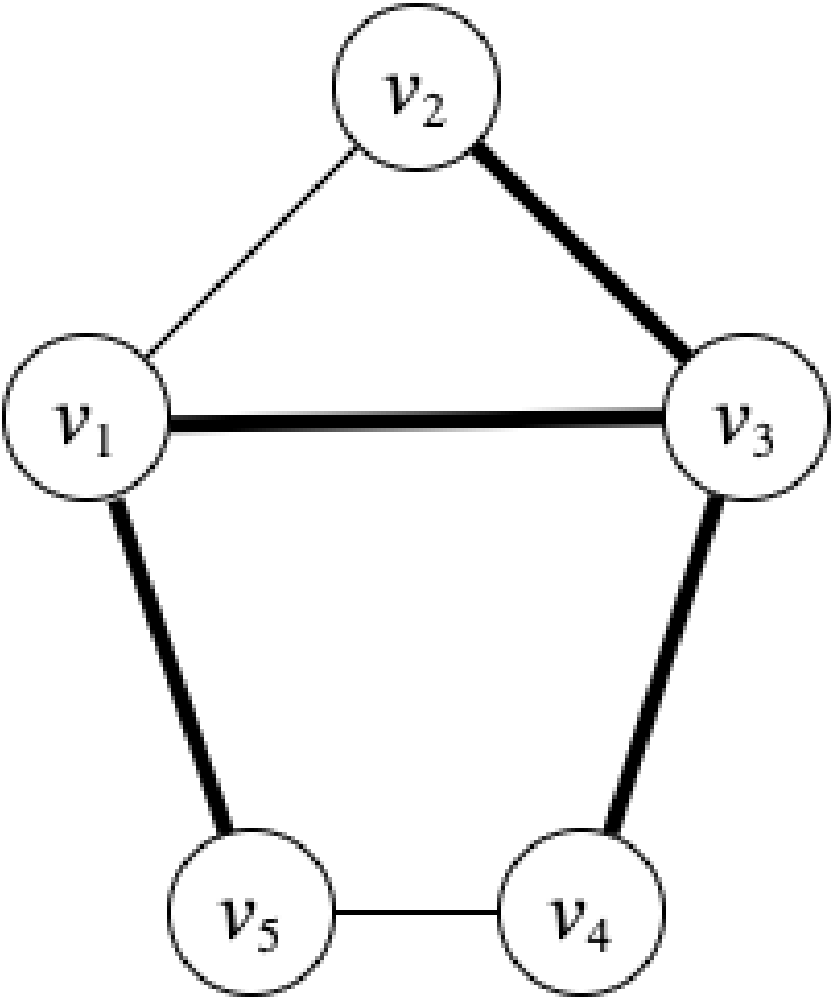


Figure 119: arbol

árbol cobertor (*spanning tree*).

Propiedades de los árboles

Es fácil demostrar las siguientes propiedades:

- Todo árbol con n nodos tiene $n - 1$ arcos.
- Si se agrega un arco a un árbol, se crea un único ciclo.

Recorridos de grafos

En muchas aplicaciones es necesario visitar todos los vértices del grafo a partir de un nodo dado. Algunas aplicaciones son:

- Encontrar ciclos
- Encontrar componentes conexas
- Encontrar árboles cobertores

Hay dos enfoque básicos:

Recorrido (o búsqueda) en profundidad (depth-first search):

La idea es alejarse lo más posible del nodo inicial (sin repetir nodos), luego devolverse un paso e intentar lo mismo por otro camino.

Recorrido (o búsqueda) en amplitud (breadth-first search):

Se visita a todos los vecinos directos del nodo inicial, luego a los vecinos de los vecinos, etc.

Recorrido en profundidad (DFS)

A medida que recorremos el grafo, iremos numerando correlativamente los nodos encontrados $1, 2, \dots$. Suponemos que todos estos números son cero inicialmente, y para ir asignando esta numeración utilizamos un contador global n , también inicializado en cero. A esta numeración asignada la llamamos el Depth-First-Number (DFN).

El siguiente algoritmo en pseudo-código muestra cómo se puede hacer este tipo de recorrido recursivamente:

```
# pseudo-código
def DFS(v): # Recorre en profundidad a partir del vértice v
    global n
    global DFN
    assert DFN[v]==0 # Supone que es primera vez que se visita el vértice v
    n=n+1
    DFN[v]=n # Numeramos el vértice v
    for w in vecinos[v]:
        if DFN[w]==0:
            DFS(w) # Visitamos w si no había sido visitado aún
```

Para dar el “puntapié inicial” al proceso, hay que hacer que todos los DFN estén en cero, inicializar en cero la variable global n e indicar el nodo de partida x :


```
# pseudo-código
def startDFS(x):
    global n
    global DFN
    n=0
    for v in V:
        DFN[v]=0
    DFS(x)
```

En el siguiente ejemplo mostraremos un posible resultado de aplicar este recorrido a un grafo dado. A la derecha se muestra el mismo grafo, con sus vértices numerados con DFN y marcando más grueso al arco que permitió llegar a por primera vez a cada vértice. A estos arcos los llamamos *arcos de árbol*. A los arcos que permiten llegar a un vértice que ya había sido visitado los llamamos *arcos de retorno* y los mostramos con línea segmentada.

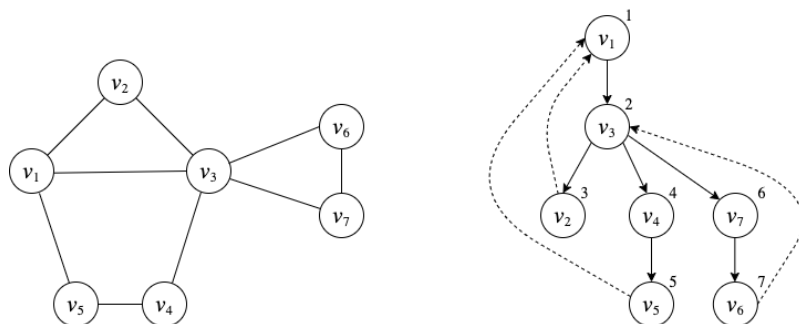


Figure 120: DFS

Si hubiera más de una componente conexa, este recorrido no llegaría a todos los nodos. Para recorrer el grafo por completo, podemos ejecutar un DFS sobre cada componente conexa.

```
# pseudo-código
def startDFS(): # recorre el grafo, retorna número de componentes conexas
    global n
    global ncc
    global DFN
    n=0
    for v in V:
        DFN[v]=0
    ncc=0 # cuenta el número de componentes conexas
    for x in V:
        if DFN[x]==0:
            ncc=ncc+1
```

```

DFS(x)
return ncc

```

Existe una gran similitud entre el DFS y el recorrido en preorden que vimos para árboles binarios. Tal como ocurrió en esa oportunidad, también es posible programar el recorrido de manera no recursiva utilizando una pila:

```

# pseudo-código
def DFSnorecursivo(x): # Recorre en profundidad a partir del vértice x
    n=0
    for v in V:
        DFN[v]=0
    s=Pila()
    s.push(x) # el vértice inicial del recorrido
    while not s.is_empty():
        v=s.pop()
        if DFN[v]==0: # primera vez que se visita este nodo
            n=n+1
            DFN[v]=n
            for w in vecinos[v]:
                s.push(w)

```

Recorrido en amplitud

Este recorrido es análogo al recorrido por niveles que vimos para árboles binarios. Su programación es similar a DFSnorecursivo, sustituyendo la pila por una cola.

Árbol Cobertor Mínimo (Minimum Spanning Tree)

Para un grafo dado pueden existir muchos árboles cobectores. Si introducimos un concepto de “peso” (o “costo”) sobre los arcos, es interesante tratar de encontrar un árbol cobector que tenga costo mínimo. El costo de un árbol es la suma de los costos de sus arcos.

En esta sección veremos dos algoritmos para encontrar un árbol cobector mínimo para un grafo no dirigido dado, conexo y con costos asociados a los arcos.

Algoritmo de Kruskal

Este es un algoritmo del tipo “avaro” (“greedy”). Comienza inicialmente con un grafo que contiene sólo los nodos del grafo original, sin arcos. Luego, en cada iteración, se agrega al grafo el arco más barato que no genere un ciclo. El proceso termina cuando el grafo está completamente conectado.

En general, la estrategia “avara” no garantiza que se encuentre un óptimo global, porque es un método “miope,” que sólo optimiza

las decisiones de corto plazo. Por otra parte, a menudo este tipo de métodos proveen buenas heurísticas, que se acercan al óptimo global. Pero en este caso, afortunadamente, se puede demostrar que el método “avaro” logra siempre encontrar el óptimo global, por lo cual un árbol cobertor encontrado por esta vía está garantizado que es un árbol cobertor mínimo.

Una forma de ver este algoritmo es diciendo que al principio cada nodo constituye su propia componente conexa, aislado de todos los demás nodos. Durante el proceso de construcción del árbol, se agrega un arco sólo si sus dos extremos se encuentran en componentes conexas distintas, y luego de agregarlo, esas dos componentes conexas se fusionan en una sola.

La siguiente animación muestra el algoritmo de Kruskal en funcionamiento. A cada paso, se intenta agregar un arco. Si se descarta, porque formaría un ciclo, se marca con una “X” y se pasa al siguiente. Si se acepta, porque une dos componentes conexas distintas, se marca como un arco sólido.

Figure 121: kruskal

Para la operatoria con componentes conexas supondremos que cada componente conexa se identifica mediante un representante canónico (el “líder” del conjunto), y que se dispone de las siguientes operaciones:

Union(a,b): Se fusionan las componentes canónicas representadas por a y b, respectivamente.

Find(x): Encuentra al representante canónico de la componente conexa a la cual pertenece x.

Con estas operaciones, el algoritmo de Kruskal se puede escribir así (en pseudo-código):

```
# pseudo-código
def kruskal(V,E):
    sort(E) # Ordenar los arcos en orden creciente de costo
    C = [[v] for v in V] # C es el conjunto de componentes conexas, inicialmente "singletons"
    T = [] # La lista de los arcos del árbol
    for e in E: # consideramos los arcos en orden creciente
        if len(C)==1: # queda solo 1 componente conexa
            break
        (v,w)=vertices(e) # los dos extremos del arco e
        if Find(v) != Find(w): # están en componentes conexas distintas
            T.append(e) # Agregar el arco e al árbol
            Union(Find(v),Find(w)) # Fusionamos las dos componentes en una sola
    return T
```

El tiempo que demora este algoritmo está dominado por lo que demora la ordenación del conjunto E de arcos, $\Theta(m \log m)$, más lo que demora realizar m operaciones Find, más n operaciones Union.

Es posible implementar Union-Find de modo que las operaciones Union demoren tiempo constante, y las operaciones Find un tiempo casi constante. Más precisamente, el costo amortizado de un Find está acotado por $\log^* n$, donde $\log^* n$ es una función definida como el número de veces que es necesario tomar el logaritmo de un número para que el resultado sea menor que 1.

Por lo tanto, el costo total es $\Theta(m \log m)$, lo cual es igual a $\Theta(m \log n)$, porque en todo grafo se tiene que $m = O(n^2)$.

La correctitud del algoritmo de Kruskal viene del siguiente lema:

Lema Sea V' subconjunto propio de V , y sea $e = \{v, w\}$ un arco de costo mínimo tal que $v \in V'$ y $w \in V - V'$. Entonces existe un árbol cobertor mínimo que incluye a e .

Este lema permite muchas estrategias distintas para escoger los arcos del árbol. Un ejemplo es el siguiente algoritmo.

Algoritmo de Prim

Comenzamos con el arco más barato, y marcamos sus dos extremos como “alcanzables.” Luego, a cada paso, intentamos extender nuestro conjunto de nodos alcanzables agregando siempre el arco más barato que tenga uno de sus extremos dentro del conjunto alcanzable y el otro fuera de él. De esta manera, el conjunto alcanzable se va extendiendo como una “mancha de aceite.”

```
# pseudo-código
def prim(V,E):
    e=arco_de_costo_minimo(E)
    (v,w)=vertices(e)
    T=[e]    # árbol
    A=[v,w]  # conjunto alcanzable
    while A!=V:
        e=arco_de_costo_minimo_que_conecta(A,V-A)
        (v,w)=vertices(e) # suponemos v en A y w en V-A
        T.append(e)
        A.append(w)
```

La siguiente animación muestra el algoritmo de Prim en funcionamiento. A cada paso, los arcos candidatos a agregarse se muestran con líneas punteadas. El arco que se acepta, por ser el de menor costo que une al conjunto alcanzable con el resto del grafo se marca como un arco sólido.

Para implementar este algoritmo eficientemente, podemos mantener una tabla donde, para cada nodo de $V - A$, almacenamos el

Figure 122: prim

costo del arco más barato que lo conecta al conjunto A . Estos costos pueden cambiar en cada iteración, así que hay que recalcularlos para todos los vecinos del nodo que se agrega al conjunto alcanzable.

Si se organiza la tabla como una cola de prioridad, el tiempo total es $\Theta(m \log n)$. Si se deja la tabla desordenada y se busca linealmente en cada iteración, el costo es $\Theta(n^2)$. Esto último es mejor que lo anterior cuando el grafo es denso.

Distancias mínimas en un grafo dirigido

Consideremos un grafo *dirigido* $G = (V, E)$, donde para cada arco $e = (u, v)$ se conoce su largo, o distancia, $d(e) = d(u, v) \geq 0$. Podemos extender la función d a todo $V \times V$ si definimos $d(u, v) = \infty$ cuando $(u, v) \notin E$.

Si se define el largo de un camino como la suma de los largos de los arcos que lo componen, es interesante encontrar los caminos más cortos que unen a nodos del grafo.

Este problema se suele estudiar en dos variantes:

- Encontrar todos los caminos más cortos desde un nodo “origen” s hasta todos los demás nodos del grafo.
- Encontrar todos los caminos más cortos entre todos los pares de nodos del grafo. Esto se puede resolver iterando n veces el problema anterior, cambiando cada vez el nodo origen, pero es posible encontrar un algoritmo más simple si se resuelve todo de una vez.

Algoritmo de Dijkstra para los caminos más cortos desde un nodo origen

La idea del algoritmo es mantener un conjunto A de nodos “alcanzables” desde el nodo origen e ir extendiendo este conjunto en cada iteración.

Los nodos alcanzables son aquellos para los cuales ya se ha encontrado su camino óptimo desde el nodo origen. Para esos nodos su distancia óptima al origen es conocida. Inicialmente $A = \{s\}$.

Para los nodos que todavía no están en A aún no se conoce su camino óptimo desde s , pero sí se puede conocer el camino óptimo *que pasa sólo por nodos de A* . Esto es, caminos en que todos los nodos intermedios son nodos de A . Llamemos a esto su camino óptimo *tentativo*.

En cada iteración, el algoritmo encuentra el nodo que no está en A cuyo camino óptimo tentativo tiene largo mínimo. Es fácil demostrar

por contradicción que ese camino tiene que ser el camino óptimo para ese nodo. Luego, ese nodo se agrega a A y su camino óptimo tentativo se convierte en su camino óptimo. Luego se actualizan los caminos óptimos tentativos para los demás nodos.

```
# pseudo-código
def Dijkstra(V,E,d,s):
    A=[s]
    for v in V:
        D[v]=d[v]
    D[s]=0
    # D[] almacena las distancias óptimas desde s para los nodos en A
    # y las distancias óptimas tentativas para los nodos en V-A
    while A!=V:
        # Encontramos el nodo v de menor distancia óptima tentativa
        v = findmin(D[u] for u in V-A)
        # Agregamos v al conjunto alcanzable
        A.append(v)
        # recalculamos las distancias óptimas tentativas
        # considerando la posibilidad de pasar por el nuevo nodo v
        for w in vecinos[v]: # los nodos w tal que (v,w) in E
            D[w] = min(D[w],D[v]+d(v,w))
    # Al terminar, D[] almacena las distancias óptimas definitivas
    return D
```

Si D se implementa como un arreglo en donde el mínimo se busca secuencialmente, encontrar el mínimo toma tiempo $\Theta(n)$, con un aporte de $\Theta(n^2)$ al tiempo total. Por otra parte, recalculer las distancias óptimas tentativas toma en total tiempo $\Theta(m)$, porque cada arco se usa a lo más una vez. Como $m = O(n^2)$, el tiempo total es $\Theta(n^2)$.

Una forma alternativa de implementar este algoritmo es usando una *cola de prioridad*, por ejemplo un heap, para almacenar los valores de D de los nodos en $V - A$. Así, encontrar y extraer el mínimo toma tiempo $\Theta(\log n)$, lo cual se ejecuta n veces, y cada recálculo toma también tiempo $\Theta(\log n)$, porque hay que cambiar la prioridad del respectivo elemento en el heap, y esto se ejecuta m veces. Por lo tanto, en esta implementación el tiempo total es $\Theta(m \log n)$.

Algoritmo de Floyd para todas las distancias más cortas

Para aplicar este algoritmo, los nodos se numeran arbitrariamente $1, 2, \dots, n$. El algoritmo va a construir una matriz D tal que, al final, $D[i, j]$ va a ser el largo del camino más corto que va desde el nodo i hasta el nodo j .

El invariante del algoritmo es que al comenzar la iteración k -ésima, $D[i, j]$ es la distancia mínima entre i y j medida a través de caminos

que pasen sólo por nodos intermedios de numeración $< k$.

En la iteración k -ésima se comparan estas distancias óptimas con las que se obtendrían si se pasara una vez por el nodo k . Si de esta manera se obtendría un camino más corto, entonces se prefiere este nuevo camino, de lo contrario nos quedamos con el camino antiguo.

Figure 123: floyd

Al terminar esta iteración, las distancias calculadas ahora incluyen la posibilidad de pasar por nodos intermedios de numeración $\leq k$, con lo cual estamos listos para ir a la iteración siguiente.

Inicialmente, hacemos $D[i, j] = d(i, j)$ para todo i, j (recordemos la convención de que $d(i, j) = \infty$ si $(i, j) \notin E$), excepto que la diagonal es $D[i, i] = 0$, porque la distancia óptima de todo nodo a sí mismo es 0.

```
# pseudo-código
def Floyd(V, E, d):
    for i in range(1, n+1):
        for j in range(1, n+1):
            D[i, j] = d(i, j)
        D[i, i] = 0

    for k in range(1, n+1):
        for i in range(1, n+1):
            for j in range(1, n+1):
                D[i, j] = min(D[i, j], D[i, k] + D[k, j])

    return D
```

El tiempo total que demora este algoritmo claramente es $\Theta(n^3)$.

Algoritmo de Warshall para cerradura transitiva y reflexiva de un grafo dirigido

Dado un grafo dirigido $G = (V, E)$, su *cerradura transitiva y reflexiva* es un grafo $G^* = (V, E^*)$, donde hay un arco $(u, v) \in E^*$ ssi existe un camino desde u hasta v en G . Si se consideran solo caminos de largo ≥ 1 , el grafo se llama la cerradura transitiva de G y se denota por G^+ . La diferencia entre G^+ y G^* es que para construir G^+ se agregan todos los arcos de transitividad, y para G^* se agregan además todos los auto-ciclos.

El siguiente ejemplo muestra un grafo dirigido G y su correspondiente cerradura transitiva y reflexiva G^* , mostrando en línea punteada los arcos que se agregan.

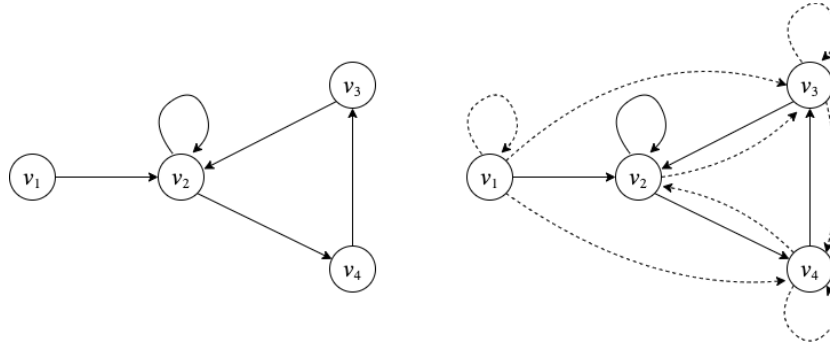


Figure 124: warshall

Supongamos que el grafo G se representa a través de su matriz de adyacencia A , con $A[i, j] = 1$ si hay un arco entre v_i y v_j , y 0 si no. Queremos calcular la matriz A^* correspondiente al grafo G^* . Para el grafo del ejemplo, tendríamos

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad A^* = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Podemos resolver este problema haciendo uso del algoritmo de Floyd, de la siguiente manera: formamos una matriz de distancias D , con

$$D[i, j] = \begin{cases} +\infty & \text{if } A[i, j] = 0 \\ 0 & \text{if } A[i, j] = 1 \end{cases}$$

Luego de aplicar el algoritmo de Floyd, tendremos que la distancia entre dos nodos es 0 ssi existe un camino entre los dos nodos, así que podemos recuperar la matriz A^* haciendo la sustitución inversa.

Yendo más allá, podemos re-examinar el funcionamiento del algoritmo de Floyd sobre esos infinitos y ceros, y ver cuál sería su interpretación si estuvieran operando sobre los ceros y unos originales, respectivamente.

El algoritmo de Floyd hace uso de dos operaciones, $+$, y \min . Las dos tablas siguientes muestran el efecto de esas operaciones:

$+$	∞	0
∞	∞	∞
0	∞	0

\min	∞	0
∞	∞	0
0	0	0

Si reemplazamos ∞ por 0 y 0 por 1, respectivamente, vemos que las tablas resultantes corresponden a las operaciones lógicas \wedge y \vee :

\wedge	0	1
----------	---	---

\vee	0	1
0	0	1
1	1	1

Si reescribimos el algoritmo de Floyd en términos de ceros y unos, con estas operaciones lógicas, obtendremos el algoritmo de Warshall:

```
# pseudo-código
def Warshall(V,E,d):
    for i in range(1,n+1):
        for j in range(1,n+1):
            A[i,j]=1 if (i,j) in E else 0
        A[i,i]=1

    for k in range(1,n+1):
        for i in range(1,n+1):
            for j in range(1,n+1):
                A[i,j]=A[i,j] or (A[i,k] and A[k,j])

    return A
```

Tal como en el caso del algoritmo de Floyd, el algoritmo de Warshall obviamente demora tiempo $\Theta(n^3)$.

Bibliography

List of Figures

1	particio-Hoare	14
2	particion-Lomuto	16
3	insercion	25
4	ord-seleccion	27
5	ord-seleccion	29
6	Torres de Hanoi	34
7	Torres de Hanoi	34
8	Diagrama de Estados 1	36
9	Diagrama de Estados 2	38
10	png	44
11	png	44
12	actividades	64
13	ejemplo-arreglo	71
14	Nodo	76
15	Nodo-circular	76
16	Nodo-None	77
17	Nodo-circular-None	78
18	Nodo-circular-cuadrado	79
19	lista-ejemplo	79
20	svg	81
21	svg	81
22	svg	81
23	svg	82
24	lista-ejemplo-con-cabecera	83
25	Nodo-doble	85
26	lista-ejemplo-doble-enlace	86
27	svg	87
28	svg	88
29	Nodo-arbol-binario	89
30	Nodo-arbol-binario-circular	89
31	ejemplo-ABB	90

32	svg	94
33	svg	97
34	arbol-formula	99
35	svg	101
36	Enum-arboles-binarios	105
37	cardinal	107
38	ordinal	108
39	ordinal-binario	110
40	pila	112
41	pila-arreglo	113
42	pila-lista	115
43	cola	120
44	cola-arreglo	121
45	cola-lista	122
46	heap-arbol	128
47	heap-arreglo	128
48	heap-ins	129
49	heap-extract	129
50	decision-tree	144
51	esquema-ABB2	146
52	insercioABB	146
53	png	153
54	eliminacionABB1	153
55	eliminacionABB2	153
56	eliminacionABB3	154
57	ejemplo-ABB	156
58	rotacion-simple	160
59	cartesian-tree	163
60	condicion-AVL	165
61	ejemplo-no-AVL	165
62	ejemplo-AVL	165
63	arboles-fibonacci	168
64	AVL1	169
65	AVL2	170
66	AVL3	170
67	orden-2-3	173
68	ejemplo-2-3	174
69	insercion-2-3-binario	174
70	insercion-2-3-ternario	175
71	inserciones-2-3	175
72	trie	180
73	abd	182

74	skip-list1	182
75	skip-list2	182
76	skip-list3	183
77	skip-list4	183
78	skip-search	184
79	abbopt1	186
80	splay1	191
81	splay2	192
82	bitmap	193
83	png	195
84	encadenamiento	195
85	LinearProbing	197
86	EjercicioLinearProbing	198
87	clusters	199
88	arbol-decision-ordenacion	203
89	particion	204
90	arbol-particion	205
91	spider	211
92	median-of-medians	212
93	heap-cons	214
94	heap-ord	214
95	heap-cons2	215
96	bucket-pass	217
97	bruta1	224
98	bruta2	224
99	bruta3	224
100	bruta4	225
101	kmp1	225
102	kmp2	225
103	kmp3	226
104	kmp4	226
105	kmp5	228
106	bmh1	230
107	bmh2	230
108	bmh3	231
109	bms1	232
110	bms2	233
111	bms3	233
112	morse	236
113	prefix-free	236
114	huffman	238

115	grafo-no-dirigido	242
116	grafo-dirigido	243
117	camino	245
118	ciclo	246
119	arbol	247
120	DFS	249
121	kruskal	251
122	prim	253
123	floyd	255
124	warshall	256

List of Tables