AYED II – PRÁCTICA 1

Preliminares y verificación de programas

EJERCICIO 1.1. Sea f_n la sucesión de Fibonacci, definida mediante $f_0 = 0$, $f_1 = 1$ y $f_n = f_{n-1} + f_{n-2}$, para $n \geq 2$. Dé los valores de las siguientes expresiones:

$$a.\ \sum_{k=1}^{0} f_k \qquad b.\ \sum_{0 \leq k^2 \leq 5} f_k \qquad c.\ \sum_{0 \leq k^2 \leq 5} f_{k^2} \qquad d.\ \sum_{0 \leq k^2 \leq 5} f_k^2$$

EJERCICIO 1.2. Considere la suma $\sum_{1 \leq i \leq j \leq k \leq 4} a_{ijk}$. Despliéguela sumando:

a) Primero en k, luego en j y luego en i.

b) Primero en i, luego en j y luego en k.

EJERCICIO 1.3. Refute o valide cada una de las siguientes igualdades:

$$\left(\sum_{k=1}^{n} a_k\right)\left(\sum_{j=1}^{n} 1/a_j\right) = \sum_{k=1}^{n}\sum_{j=1}^{n} a_k/a_j = \sum_{k=1}^{n}\sum_{k=1}^{n} a_k/a_k = \sum_{k=1}^{n} n = n^2.$$

EJERCICIO 1.4. Sea s_n la suma de los primeros n términos de la serie aritmética $a, a+b, a+2b, \ldots$. Exprese s_n utilizando el símbolo Σ y luego pruebe que s_n es exactamente $an + n(n-1)b/2$.

EJERCICIO 1.5. Sea s_n la suma de los primeros n términos de la serie geométrica $a, ar, ar^2, \ldots$. Pruebe que si $r \neq 1$ entonces s_n es exactamente $\frac{a(1-r^n)}{1-r}$.

EJERCICIO 1.6. Probar por inducción que $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

EJERCICIO 1.7. Pruebe que $\sum_{k=0}^{n} \binom{n}{k} = 2^n$.

EJERCICIO 1.8. Pruebe que $\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1}$. (Ayuda: Derive ambos miembros de $(1+x)^n = \sum_{i=0}^{n} \binom{n}{i} x^i$).

EJERCICIO 1.9. Considere n rectas no paralelas dos a dos y sin puntos de intersección múltiples. Sea L_n el número de regiones en las que queda dividido el plano. Pruebe que $L_n = 1 + n(n+1)/2$.

EJERCICIO 1.10. Considere los siguientes programas. Para cada uno de ellos, escriba formalmente la postcondición y una adecuada precondición, y luego establezca un invariante que permita verificar el ciclo.

(1) **while** $x \neq y$
  if $x > y$ **then** $x := x - y$
  if $x < y$ **then** $y := y - x$

(2) $q, r := 0, x$
  while $r \geq y$ **do** $q, r := q + 1, r - y$

EJERCICIO 1.11. Un polinomio $a_0 + a_1 t + \cdots + a_{N-1} t^{N-1}$ se puede representar con el arreglo $[a_0, a_1, \ldots, a_{N-1}]$. Hacer lo mismo que en Ejercicio 1.10 para los siguientes programas que evalúan, de dos maneras distintas, un polinomio en x.

(1) **var** i : **int**
$i, r := 0, 0$
while $i \neq N$ **do** $r, i := rx + X[N - i - 1], i + 1$

(2) **var** i, y : **int**
$i, r, y := 0, 0, 1$
while $i \neq N$ **do** $r, i, y := r + X[i]y, i + 1, yx$

EJERCICIO 1.12. Verifique la corrección de los siguientes algoritmos recursivos, especificando previamente la precondición y la postcondición.

(1) **var** X : **array**$[0..n)$ **of int**
func $suma(X$: **array**, n : **nat**$)$ **dev: int**
if $n = 0$ **then** $s := 0$
if $n > 0$ **then**
$s := suma(X, n - 1)$
$s := s + X[n - 1]$
return s

(2) **func** $dividir(a, b$: **nat**$)$ **dev: nat**, **nat**
if $a < b$ **then** $q, r := 0, a$
if $a \geq b$ **then**
$q, r := dividir(a - b, b)$
$q := q + 1$
return q, r

AYED II – PRÁCTICA 2

Análisis de algoritmos

EJERCICIO 2.1. Hallar de manera exacta y lo más simple posible el tiempo de ejecución de los siguientes programas (por supuesto, las sumatorias deben ser eliminadas).

```
t := 0;
for i := 1 to n do
    for j := 1 to n^2 do
        for k := 1 to n^3 do t := t + 1
```

```
t := 0;
for i := 1 to n do
    for j := 1 to i do
        for k := j to n do t := t + 1
```

EJERCICIO 2.2. Determinar cuáles de las siguientes afirmaciones son válidas. Justificar.

a) $\mathcal{O}(f+g) = \mathcal{O}(max(f,g))$.
b) Si $s \in \mathcal{O}(f)$ y $r \in \mathcal{O}(g)$ entonces $s + r \in \mathcal{O}(f+g)$.
c) Si $s \in \mathcal{O}(f)$ y $r \in \mathcal{O}(g)$ entonces $s - r \in \mathcal{O}(f-g)$.
d) $2^{n+1} \in \mathcal{O}(2^n)$.
e) $(m+1)! \in \mathcal{O}(m!)$.

EJERCICIO 2.3. Ordenar intercalando "=" o "$\subset$" los $\mathcal{O}$ de las siguientes funciones, donde $0 < \varepsilon < 1$:

$$n^8, \qquad nlog(n), \qquad n^{1+\varepsilon}, \qquad (1+\varepsilon)^n, \qquad n^2/log(n), \qquad (n^2-n+1)^4.$$

EJERCICIO 2.4. Pruebe o refute las relaciones $f \in \mathcal{O}(g)$, $f \in \Omega(g)$ y $f \in \Theta(g)$ para los siguientes pares:

a) $100n + log(n),\ n + (log(n))^2$.
b) $n^2/log(n),\ n(log(n))^2$.
c) $n2^n,\ 3^n$.

EJERCICIO 2.5. Pruebe que $log(n!) \in \Theta(nlog(n))$. (Ayuda: use la fórmula de Stirling, $n! = \sqrt{2\pi n}\ (n/e)^n$)

EJERCICIO 2.6. Dé soluciones exactas para las siguientes recurrencias:

(1) $t(n) = t(n-1) + n/2,\ t(1) = 1$.
(2) $t(n) = 8t(n-1) - 15t(n-2),\ t(1) = 1,\ t(2) = 4$.
(3) $t(n) = 3t(n-2) - 2t(n-3),\ t(1) = 0,\ t(2) = 0,\ t(3) = 1$.
(4) $t(n) = 8t(n-1) - 15t(n-2), t(1) = 1, t(2) = 4$.

EJERCICIO 2.7. Demuestre que:

(1) $\sum_{i=1}^{n} i^k \in \mathcal{O}(n^{k+1})$, (2) $\sum_{i=1}^{n} i^k log(i) \in \mathcal{O}(n^{k+1}log(n))$.

EJERCICIO 2.8. Determine cuales de las siguientes funciones son uniformes:

$$n, \qquad nlog(n), \qquad n^k, \qquad n^{log(n)}, \qquad 2^n, \qquad n!\,.$$

EJERCICIO 2.9. Sea $t(n) = 2t(\lfloor n/2 \rfloor) + 2nlog(n),\ t(1) = 4$. Pruebe que $t(n) \in \mathcal{O}(nlog^2(n))$.

EJERCICIO 2.10. Para cada uno de los siguientes algoritmos escriba una ecuación recursiva asintótica para $T(n)$. Dé el orden exacto de T expresándolo de la forma más simple posible.

proc $DC(n : \textbf{nat})$
  if $n \leq 1$ **then skip**
  if $n > 1$ **then**
    for $i := 1$ **to** 8 **do** $DC(n$ **div** $2)$
    for $i := 1$ **to** n^3 **do** $dummy := 0$

proc $waste(n : \textbf{nat})$
  for $i := 1$ **to** n **do**
    for $j := 1$ **to** i **do** $write\ i, j, n$
  if $n \leq 0$ **then skip**
  if $n > 0$ **for** $i := 1$ **to** 4 **do** $waste(n$ **div** $2)$

EJERCICIO 2.11. Ordenar según $\subseteq$ los $\mathcal{O}$ de las siguientes funciones. No calcular límites, utilizar las propiedades algebraicas.

(1) $n \log 2^n$
(2) $2^n \log n$
(3) $n! \log n$
(4) 2^n

EJERCICIO 2.12. Resolver la siguiente recurrencia

$$t_n = \begin{cases} 5 & n = 1 \\ 3t_{n-1} - 2^{n-1} & n > 1 \end{cases}$$

EJERCICIO 2.13. Resolver la siguiente recurrencia

$$t_n = \begin{cases} 0 & n = 0 \vee n = 1 \\ 1 & n = 2 \\ 3t_{n-1} - 4t_{n-3} & c.c. \end{cases}$$

EJERCICIO 2.14. Calcular el orden del tiempo de ejecución del siguiente algoritmo

proc $p(n : \textbf{nat})$
  for $j := 1$ **to** 6 **do**
    if $n \leq 1$ **then skip**
    if $n \geq 2$ **then**
      for $i := 1$ **to** 3 **do** $p(n$ **div** $4)$
      for $i := 1$ **to** n^4 **do write** i

¿Qué operación u operaciones está considerando como elementales? Justificar.

EJERCICIO 2.15. Calcular el orden exacto del tiempo de ejecución de cada uno de los siguientes algoritmos:

(1) $t := 0;$
  for $i := 1$ **to** n **do**

```
        for j := 1 to i do
            for k := j to j + 3 do t := t + 1
(2)  proc p(n : nat)
        if n ≥ 2 then
            for i := 1 to 16 do p(n div 4)
            for i := 1 to n do write i
            for i := 1 to n do
                for j := 1 to n do write j
```

EJERCICIO 2.16. Ordenar las siguientes funciones según $\subset$ (incluido *estricto*) e $=$ de sus 's.

(1) $n^4 + 2\log n$
(2) $\log(n^{n^4})$
(3) $2^{4\log n}$
(4) 4^n
(5) $n^3 \log n$

Justificar sin utilizar la regla del límite.

EJERCICIO 2.17. Sea p el procedimiento dado por

```
proc p(n : nat)
    if n ≤ 1 then skip
    if n ≥ 2 then
        for i := 1 to C do p(n div D)
        for i := 1 to n^4 do write i
```

Determine posibles valores de la constante C y D de manera que el procedimiento p tenga orden

(1) $\Theta(n^4 \log n)$
(2) $\Theta(n^4)$
(3) $\Theta(n^5)$

En los casos en que

(a) $D = 2$
(b) $C = 64$

EJERCICIO 2.18. Dar algoritmos cuyos tiempos de ejecución tengan los siguientes órdenes:

(1) $n^2 + 2\log n$
(2) $n^2 \log n$
(3) 3^n

No utilizar potencia, logaritmo ni multiplicación en los programas.

EJERCICIO 2.19. Dar el orden exacto de f. Justificar.

```
func f(i, k : int) dev: int
  {pre: i ≤ k }
    if i < k then
        j := (i + k) div 2
        m := max(f(i, j), f(j + 1, k))
    else m := a[i]
    return m
```

EJERCICIO 2.20. Resolver de manera exacta la siguiente recurrencia
$2t(n) = t(n-1) + t(n-2) + 12n - 16 \qquad t(0) = 1,\ t(1) = 1.$

EJERCICIO 2.21. Calcular el orden exacto del tiempo de ejecución del siguiente algoritmo:

$p := 1$
while $p < n$ **do** $p := p * 3$

EJERCICIO 2.22. Una secuencia $x_1, ..., x_n$ se dice que tiene *orden cíclico* si existe un x_i tal que la secuencia $x_i, x_{i+1}, ..., x_n, x_1, ..., x_{i-1}$ es estríctamente creciente. Supongamos que se recibe una secuencia con orden cíclico almacenada en un arreglo X definido en $[1, n]$. Utilice una idea similar a la de búsqueda binaria para diseñar e implementar un algoritmo que encuentre el menor elemento de la secuencia en tiempo $\mathcal{O}(log\ n)$. Analice en forma detallada la complejidad.

2.1. Ejercicios adicionales

EJERCICIO 2.23. Dado un arreglo de enteros $A[1, n]$ se pide "ordenar" A sin modificarlo. Para ello, el algoritmo deberá crear un arreglo $I[1, n]$, inicializarlo con los valores $I = [1, \ldots, n]$ y realizar las modificaciones necesarias en I para que al finalizar se tenga $A[I[1]] \leq A[I[2]] \leq \ldots \leq A[I[n]]$. El algoritmo deberá devolver el arreglo I. Calcular la complejidad.

EJERCICIO 2.24. Una cima de $A[0, N)$ es un entero k en el intervalo $[0, N-1]$ que satisface que $A[0, k]$ es estrictamente creciente y $A[k, N)$ es estrictamente decreciente. Dar un algoritmo lo más eficiente posible que, asumiendo que la cima existe, la encuentra.

EJERCICIO 2.25. Resuelva la siguiente recursión:

$$t(n) = n + \sum_{i=1}^{n-1} t(i), \quad t(1) = 1.$$

EJERCICIO 2.26. Pruebe que el tiempo de ejecución de las siguientes versiones de búsqueda binaria es en ambos casos $\mathcal{O}(log(n))$.

func $bs1(X$: **array**, x : **int**, i, j : **nat**$)$ **dev: nat**
if $i = j$ **then** $k := i$
if $i < j$ **then**
    var m : **nat**
    $m := \lceil (i + j)/2 \rceil$
    if $x < X[m]$ **then** $k := bs(X, x, i, m - 1)$
    if $x = X[m]$ **then** $k := bs(X, x, m, m)$
    if $x > X[m]$ **then** $k := bs(X, x, m + 1, j)$
return k

func $bs2(X$: **array**, x : **int**, i, j : **nat**$)$ **dev: nat**
while $i < j$ **do**
    $k := (i + j)$ **div** 2
    if $x < X[k]$ **to** $j := k - 1$
    if $x = X[k]$ **to** $i, j := k, k$
    if $x > X[k]$ **to** $i := k + 1$
return k

EJERCICIO 2.27. Resuelva de manera exacta las siguientes recurrencias. Exprese la respuesta usando notación Θ, de la manera más simple posible.

(1) $t(n) = a$, si $n = 0, 1$,
(2) $t(n) = t(n-1) + t(n-2) + c$, si $n \geq 2$.
(3) $t(n) = a$, si $n = 0, 1$,
(4) $t(n) = t(n-1) + t(n-2) + cn$, si $n \geq 2$.

EJERCICIO 2.28. Ordenar según $\subset$ y $=$ los de las siguientes funciones. No calcular límites, utilizar las propiedades algebraicas.

(1) $(\log_3 2)^n$
(2) $(\log_5 2)^n$
(3) $n^{\log_3 2}$
(4) $n^{\log_5 2}$
(5) n

EJERCICIO 2.29. Resolver la recurrencia $f(n) = 8f(n-1) - 15f(n-2)$, $f(1) = 1$, $f(2) = 4$.

EJERCICIO 2.30. Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}^+$ ¿Cuáles de las siguientes condiciones son equivalentes a $f(n) \in \mathcal{O}(g(n))$? Responder confeccionando dos listas: una que enumera aquéllas que sí son equivalentes y otra que enumera aquéllas que no lo son.

(a)	$4f(n) \in \mathcal{O}(\sqrt{3}g(n))$	(b)	$\mathcal{O}(g(n)) \subseteq \mathcal{O}(f(n))$	(c)	$\Omega(g(n)) \subseteq \Omega(f(n))$
(d)	$\Omega(f(n)) \subseteq \Omega(g(n))$	(e)	$f(n) \in \Omega(g(n))$	(f)	$g(n) \in \Omega(f(n))$
(g)	$\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$	(h)	$\Theta(f(n)) \subseteq \Theta(g(n))$	(i)	$\Omega(f(n)) \subseteq \mathcal{O}(g(n))$
(j)	$1/g(n) \in \Omega(1/f(n))$	(k)	$\mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$	(l)	$f(n)^2 \in \mathcal{O}(g(n)^2)$

EJERCICIO 2.31. El siguiente algoritmo obtiene el mínimo elemento de un arreglo a : **array**$[1..n]$ mediante la técnica de programación "divide y vencerás". Determinar el tiempo de ejecución de *minimo*$(1, n)$.

func *minimo*$(i, k$: **int**$)$ **dev: int**
{pre: $i \leq k$*}*
if $i = k$ **then** $m := a[i]$
else
$j := (i + k)$ **div** 2
$m := \min($*minimo*$(i, j),$ *minimo(j+1,k)*$)$
{post: m *es el mínimo de* $a[i, k]$*}*

EJERCICIO 2.32. Escriba procedimientos p que tengan órdenes

(1) $\Theta(n^2 \log n)$
(2) $\Theta(n^3)$
(3) $\Theta(n^2 \log n \log n)$

EJERCICIO 2.33. Escribir un algoritmo cuyo orden exacto es $\log^2 n$. Las únicas operaciones aritméticas permitidas son: suma, resta, multiplicación y división.

EJERCICIO 2.34. Escribir un algoritmo cuyo orden exacto es $n!$. Las únicas operaciones aritméticas permitidas son: suma, resta, multiplicación y división.

AYED II – PRÁCTICA 3

Estructuras de datos

EJERCICIO 3.1. Desarrolle de manera completa la implementación del tipo abstracto *pila* utilizando un arreglo y un entero. Todas las operaciones deben tener complejidad $\mathcal{O}(1)$. Ponga especial cuidado en los casos de excepción, por ejemplo pedir el tope de una pila vacía, etc. El comportamiento del algoritmo en estos casos debe ser acorde a la especificación abstracta.

EJERCICIO 3.2. Hacer lo mismo que en el Ejercicio 3.1 para el tipo abstracto *cola*, utilizando un arreglo y dos enteros. Todas las operaciones deben seguir siendo $\mathcal{O}(1)$.

EJERCICIO 3.3.
(1) Hacer lo mismo que en el Ejercicio 3.1 para el tipo abstracto *lista,* utilizando punteros. Estudie la complejidad de cada una de las operaciones del tipo.
(2) Repita (1) con una implementación sobre arreglos.
(3) Para cada una de las siguientes operaciones diga cuales pueden implementarse en tiempo $\mathcal{O}(1)$ sobre punteros, y cuales sobre arreglos: devolver el i-ésimo elemento, devolver el primer elemento, devolver el último elemento, insertar un elemento en el k-ésimo lugar, insertar un elemento al principio y devolver la longitud de la lista.

EJERCICIO 3.4. Considere el tipo abstracto *polinomio con coeficientes enteros.* El tipo posee las operaciones abstractas "evaluar en x" y "devolver el coeficiente de grado k". Fije constructores y dé una especificación del tipo abstracto. Obtenga luego una implementación utilizando arreglos, de manera que el lugar k aloje al coeficiente de grado k.

EJERCICIO 3.5. Considere el tipo abstracto del problema anterior. Obtenga una implementación eficiente en espacio, en la cual el espacio requerido no dependa del grado. Por ejemplo el polinomio $x^{9000}+1$ debería ser representado mediante un estructura de escaso tamaño.

EJERCICIO 3.6. Los algoritmos de ordenamiento de secuencias de palabras deben efectuar frecuentemente la operación "swap", y la pregunta ¿es la palabra del lugar i menor o igual a la palabra del lugar j en el orden lexicográfico? Implemente estas dos operaciones abstractas sobre la siguiente representación de una secuencias de k palabras. Las palabras están dispuestas de manera consecutiva en un arreglo de caracteres $C[1..M]$. Se dispone de otro arreglo $A[1..N]$ de pares de naturales que contiene en el lugar i (con $1 \leq i \leq k$) al par que indica el lugar de comienzo de la palabra i-ésima en C, y su tamaño. Por ejemplo, si $k = 4$, en los arreglos
$C = arbolcasaelefanteaseo...$
$A = (6,4)\ (1,5)\ (12,6)\ (10,2)\ ...$
se representa la secuencia $casa,\ arbol,\ efante,\ el.$

EJERCICIO 3.7. Considere el tipo "Lista enriquecida de enteros" *Elist,* que posee el género abstracto $Elist$, y las operaciones del tipo lista $\triangleright,\ [],\ tail,\ head$ a las que se agregan

$lenght$: $Elist \to nat$ longitud de la lista.
$last$: $Elist \to int$ último elemento.
$nulls$: $Elist \to nat$ cantidad de ceros que contiene la lista.

Obtenga una implementación del tipo abstracto $Elist$, que utilice el tipo lista que usted conoce, de manera que todas las operaciones sean $\mathcal{O}(1)$.

EJERCICIO 3.8. Especifique el tipo abstracto $Conjunto$, que posee las operaciones $empty$, $insert$, $delete$ y $member?$. Obtenga luego implementaciones del tipo abstracto mediante *a)* Listas ordenadas. *b)* Árboles ordenados (ABB). Estudie detalladamente la complejidad de cada operación en ambas implementaciones.

EJERCICIO 3.9. Se define el tipo nodo de la siguiente manera:

type *nodo* = **record**
letra : **char**
def : ↑*significado*
primerHijo : ↑*nodo*
hermano : ↑*nodo*

Este tipo permite definir un árbol finitario de la siguiente manera. Sean n y m de tipo ↑*nodo*.
Decimos que n es hijo de m sii $n = m\uparrow$.*primerhijo* o existe otro hijo p de m tal que $n = p\uparrow$.*hermano*.
Se utiliza esta estructura para implementar un diccionario de la siguiente manera: dada una secuencia $n_1 \ldots, n_k$ con $k > 0$ tal que n_2 es hijo de $n_1, \ldots, n_k$ es hijo de n_{k-1}, sean $l_1 = n_1 \uparrow$.*letra*, $\ldots, l_k = n_k\uparrow$.*letra*, el significado de la palabra $l_1 \ldots l_k$ es el indicado por $n_k\uparrow$.*def*.
Asumiendo que no hay dos hermanos n y m tales $n \uparrow$.*letra* $= m \uparrow$.*letra* escribir una función iterativa que dados *dicc* : ↑*nodo* y una palabra (arreglo de caracteres) devuelve un d : ↑*significado* que indica el significado de la palabra (que por convención es nil si la palabra no está en el diccionario).

EJERCICIO 3.10. Implementar el TAD cola utilizando como representación listas enlazadas circulares, es decir, una cola está dada por un puntero al último nodo y éste a su vez contiene un puntero al primer nodo, etc. Utilizar la siguiente estructura:

type *nodo* = **record**
info : *elem*
sgte : ↑*nodo*
type *cola* = ↑*nodo*

3.1. Ejercicios adicionales

EJERCICIO 3.11. (Cinta de Caracteres.) Todos los lenguajes de programación proveen una manera de manipular archivos de texto que permiten, entre otras cosas, abrir un archivo para lectura, leer caracter por caracter, preguntar si se está al final del archivo, etc. El tipo *cinta de caracteres* tiene como objetivo brindar una forma clara y precisa de introducir en nuestros programas estas operaciones, independientemente del lenguaje que se va a usar luego en la implementación.
Una cinta de caracteres C "montada" sobre una secuencia de caracteres S permite leer uno a uno los caracteres de S desde el principio al fin. El tipo abstracto posee las operaciones

- $crear(S)$ (monta la cinta C desde la secuencia de caracteres S),
- $arr(C)$ (arrancar la cinta, o sea posicionarse en el primer caracter),
- $av(C)$ (avanzar al siguiente caracter),
- $cor(C)$ (devolver el caracter corriente),

- $fin?(C)$ (respuesta a: ¿se llegó al final de la cinta?)

Por ejemplo, sea S la secuencia

$$\text{“ } cinta\text{”}$$

formada por un primer caracter blanco seguido de los caracteres de la palabra $cinta$. Mediante $crear(S)$ montamos una cinta sobre la secuencia S, y al hacer

$$C = arr(crear(S))$$

posicionamos el “cursor” de la cinta C sobre el primer caracter de S, y tenemos:

$$cor(C) = \text{ ' ' (caracter blanco)},$$
$$fin?(C) = falso$$
$$cor(av(C)) = \text{ 'c'}.$$

Para especificar el tipo abstracto es conveniente utilizar las operaciones ocultas $pi(C)$ y $pd(C)$, que devuelven las secuencias que corresponden a la parte izquierda y derecha de la cinta, respectivamente. Esta operaciones se denominan ocultas porque no son operaciones que ofrezca el tipo (en particular no pueden utilizarse en un programa), sino que son operaciones que nos sirven para describir el comportamiento de otras operaciones. Los secuencias $pi(C)$ y $pd(C)$ se encuentran delimitadas por $cor(C)$, que es el primer elemento de la parte derecha. En el ejemplo anterior, si

$$C_1 = av(av(C)),$$

entonces tenemos $pi(C_1) =$ “ c” y $pd(C_1) =$ “$inta$”. Así, desde que se monta la cinta C sobre la secuencia S se mantiene la condición invariante

$$S = pi(C) + +pd(C).$$

Vamos a especificar ahora de manera precisa el comportamiento de las operaciones del tipo a través de pre y pos condiciones. La operación $crear(S)$ no posee especificación pre-pos pues juega el rol de constructor del tipo. Las especificaciones de arr y av son:

$\{S = pi(C) + +pd(C)\}$
$\quad arr(C)$
$\{pi(C) = [\,] \ \wedge \ pd(C) = S\}$

$\{pi(C) = I \ \wedge \ pd(C) = c \triangleright D\}$
$\quad av(C)$
$\{pi(C) = I \triangleleft c \ \wedge \ pd(C) = D\}$

La especificación se completa con las propiedades:

$$fin?(C) \Leftrightarrow pd(C) = [\,]$$
$$\neg fin?(C) \Rightarrow cor(C) = head(pd(C))$$

Ambas operaciones se pueden aplicar solamente si la cinta fue arrancada. Note que de la especificación se deduce fácilmente que para ejecutar $av(C)$ debemos tener la condición $\neg fin?(C)$.

(1) Desarrolle un algoritmo iterativo que calcule la cantidad de caracteres de una secuencia S (que puede ser vacía), montando una cinta sobre S. Si se utiliza la variable de programa n para contar, entonces el invariante será

$$\text{“}n \text{ es la cantidad de caracteres de } pi(C)\text{”}.$$

(2) Desarrolle un algoritmo iterativo que determine si la cantidad de ocurrencias del caracter 'a' en una secuencia S (que puede ser vacía) es igual a la cantidad de ocurrencias de 'b'.

EJERCICIO 3.12. (Cintas de objetos abstractos.) Frecuentemente se debe leer objetos abstractos que han sido almacenados en un archivo de texto utilizando ciertos criterios (a veces arbitrarios) para disponerlos como secuencia de caracteres. Por ejemplo es usual que se disponga un texto del lenguaje castellano en un archivo separando las palabras por caracteres blancos. Cuando el objeto en cuestión es más complejo, su disposición como secuencia podría estar sujeta a decisiones más arbitrarias. Por ejemplo en el caso del tipo *Alumno*, que es una estructura compuesta por *nombre*, *DNI*, *N^o_de_alumno*, *lista_de_materias_rendidas*, etc., deberíamos escoger caracteres que separen los distintos datos de cada alumno, y a su vez que separen alumnos entre sí. Luego para respetar el criterio de modularidad en los programas y facilitar la reusabilidad, es conveniente que todas estas decisiones se resuelvan en una correcta implementación de la cinta abstracta sobre la cinta de caracteres, y que luego se manipulen los objetos abstractos a través de la cinta abstracta.

(1) Vamos a considerar como objeto abstracto a las palabras. Obtenga una especificación del tipo abstracto *cinta de palabras*. Note que en la especificación cinta de caracteres en ningún momento se hace referencia a la naturaleza de los objetos que aloja la secuencia, de manera que es posible obtener una especificación de cinta de palabras desde la especificación de la cinta de caracteres, cambiando el nombre a las funciones para que se ponga en evidencia que los objetos alojados son palabras (por ejemplo utilizar $arrCP, avCP, pal$ y $finCP?$). Ya que $pi,\ pd$ son ocultas, se puede seguir utilizando los mismos nombres para estas funciones.

(2) En este problema se debe obtener una implementación de la cinta de palabras sobre la cinta de caracteres. Vamos a ver primero que es necesario utilizar en la representación algunos datos extras, además de la cinta de caracteres. Dado que la cinta de caracteres lee un caracter por vez, debemos incorporar a la implementación una palabra que en todo momento (antes del fin de la cinta) sea la palabra corriente. Luego el "avanzar" de cinta de palabras deberá primero desechar los espacios en blanco y luego leer toda una palabra hasta llegar a un nuevo blanco. Por ejemplo para implementar la situación abstracta:

$$\left\{\begin{array}{cccc} ... & [la] & [casa] & ... \\ & \uparrow & & \\ & pal & & \end{array}\right\} \ avCP(CP) \ \left\{\begin{array}{cccc} ... & [la] & [casa] & ... \\ & & \uparrow & \\ & & pal & \end{array}\right\}$$

debemos partir de la cinta

$$\begin{array}{cccccccccccc} ... & \beta & l & a & \beta & \beta & \beta & c & a & s & a & \beta & ... \\ & & & & \uparrow & & & & & & & & \\ & & & & cor & & & & & & & & \end{array}$$

y arribar a la cinta

$$\begin{array}{cccccccccccc} ... & \beta & l & a & \beta & \beta & \beta & c & a & s & a & \beta & ... \\ & & & & & & & & & & & \uparrow & \\ & & & & & & & & & & & cor & \end{array}$$

almacenando la palabra '$casa$'. El tipo de implementación será entonces un record con un campo de tipo $Cinta\ de\ Caracteres$ y otro campo de tipo $Palabra$. Llamemos $CP.cinta$ y $CP.pal$ a estos campos. Un análisis superficial podría sugerir implementar la operación $finCP?$ mediante $fin?$. Veamos que esto tiene un inconveniente. Supongamos que tenemos la cinta anterior, pero

que la última 'a' de '$casa$' es el último caracter de la cinta:

$$\begin{array}{lcccccccccc} & ... & \beta & l & a & \beta & \beta & \beta & c & a & s & a \\ CP.cinta & & & & & \uparrow & & & & & & \\ & & & & & cor & & & & & & \end{array}$$

Después de leer una palabra tendremos la situación $fin?(CP.cinta)$, pero la situación abstracta será:

$$\begin{array}{ccccc} ... & [la] & [casa] & ... \\ & & \uparrow & \\ & & pal & \end{array}$$

que no se corresponde con el fin de la cinta abstracta pues su parte derecha es $[[casa]]$. Luego es imprescindible que incorporemos un campo booleano a la implementación, llamado $CP.fin$, que nos dirá si estamos al final de la cinta abstracta. En el caso anterior tendremos $fin?(CP.cinta)$ pero $CP.fin = false$.

El tipo de implementación será entonces:

type *Cinta_de_Plalabras* = **record**
cinta : *Cinta_de_Caracteres*
pal : *Palabra*
fin : **bool**

Complete la implementación escribiendo procedimientos y funciones que implementen todas las operaciones del tipo abstracto cinta de palabras.

EJERCICIO 3.13. Especifique el tipo abstracto $Diccionario$. Los items que contiene el diccionario poseen una clave nat, dada por una función $key : item \to nat$. Se diferencia del tipo conjunto porque en lugar de la función $member$ posee una función que dada una clave determina si hay un item con esa clave. Un diccionario no admite dos items con la misma clave. El borrado de un item recibe una clave.

EJERCICIO 3.14. El tipo abstracto *multiconjunto* (conjuntos de enteros con repeticiones) tiene operaciones

- *empty*: Multiconjunto sin elementos.
- *ins*(x, M): Inserta (una ocurrencia más de) el entero x en el multiconjunto M.
- *del*(x, M): Borra una ocurrencia del entero x en el multiconjunto M.
- *mult*(x, M): Devuelve el número de veces que ocurre el entero x en M.
- *isEmpty*(M): Devuelve *true* si y sólo si el multiconjunto M es vacío.

Por ejemplo:

$$\begin{array}{l} \textit{ins}(5, \{1, 1, -4, 5\}) = \{1, 1, -4, 5, 5\} \\ \textit{del}(1, \{1, 1, -4, 5\}) = \{1, -4, 5\} \\ \textit{mult}(1, \{1, 1, 4, 5\}) = 2 \end{array}$$

Implemente el tipo abstracto *multiconjunto* utilizando árboles binarios de búsqueda.

EJERCICIO 3.15. Se define el tipo lista de una manera diferente, con las siguientes operaciones: *vacía* que crea una lista vacía, *insertar*(l, k, e) que se aplica sólo a listas l que tienen al menos k elementos e inserta e de modo que e quede en la posición k de la lista resultante, *borrar*(l, k) que se aplica sólo a listas que tienen al menos $k + 1$ elementos y borra el que se encuentra en la posición k, *longitud*(l) que devuelve la longitud de l y *seleccionar*(l, k) que se aplica sólo a listas que tienen al menos $k + 1$ elementos y devuelve el que se encuentra en la posición k.
Implementar la función longitud y el procedimiento insertar usando:

type *nodo* $=$ **record**
    info : *elem*
    sgte : $\uparrow$*nodo*
type *lista* $=\uparrow$*nodo*

EJERCICIO 3.16. Un palíndromo es una palabra o frase que, salvo por los espacios, se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo: “dábale arroz a la zorra el abad” es un palíndromo. Escribir un algoritmo que dada una secuencia s de letras determine, con la ayuda de una pila p de letras, si s es o no un palíndromo. Asumir que se conoce la longitud inicial k de s (sin contar sus espacios). La secuencia s sólo puede manipularse a través de la función *isEmpty*(s) que dice si s es o no vacía y el procedimiento *take*(s, l) que asigna a l la primera letra de s y la elimina de s. Ambas operaciones se asumen predefinidas.

AYED II – PRÁCTICA 4

Estructuras de Datos (continuación)

EJERCICIO 4.1. Elimine la recursión del algoritmo de búsqueda de un elemento de un árbol binario de búsqueda (ABB).

EJERCICIO 4.2. Para $h = 2, 3, 4, 5$ dibuje ABB's de altura h con la mínima cantidad de nodos posible que satisfagan la siguiente propiedad: para cada nodo n las alturas de los subárboles izquierdo y derecho de n difieren a lo sumo en 1. ¿Qué cantidad de nodos posee el ABB en cada caso? Muestre ejemplos de inserción y borrado sobre estos árboles que rompan con ésta propiedad.

EJERCICIO 4.3. Especifique de manera formal la operación "altura de un árbol binario de naturales". Implemente luego tal operación suponiendo que los árboles están representados con punteros.

EJERCICIO 4.4. Podemos extender el TAD árbol binario agregando la operación p que devuelve el padre. Implemente la estructura de árbol binario de tal forma que las operaciones para obtener el valor de un nodo (*key*), el hijo izquierdo (*left*), el hijo derecho (*right*) y el padre (p) sean de tiempo constante. Estas operaciones tienen como dominio y codominio el tipo árbol. Si x es un árbol y no tiene hijo izquierdo (resp. hijo derecho, resp. padre) entonces *left*$[x]$ (resp. *right*$[x]$, resp. $p[x]$) devuelve *NIL* .

EJERCICIO 4.5. Recuerde que un árbol binario es de búsqueda (ABB) si para cada nodo x se satisface que cuando y pertenece al subárbol izquierdo de x, entonces *key*$[y] \leq$ *key*$[x]$ y si y pertenece al subárbol derecho de x, entonces *key*$[x] \leq$ *key*$[y]$. Una *recorrido inorder* en un ABB muestra todos los nodos del árbol en forma ordenada. Sea el siguiente algoritmo

proc *recorridoInorder*(x)
if $x \neq$ *NIL* **then**
    recorridoInorder(*left*$[x]$)
    write *key*$[x]$
    recorridoInorder(*right*$[x]$)

donde x es un árbol.

(1) Probar que si T es un árbol binario de búsqueda, entonces *recorridoInorder*(T) imprime los valores de los nodos en forma ordenada.
(2) Probar que el algoritmo es $\Theta(n)$.
(3) Sea A un heap con n elementos, ¿se puede imprimir en orden los valores de A en tiempo $\Theta(n)$?

EJERCICIO 4.6. Con las claves $\{1, 4, 5, 10, 16, 17, 21\}$ construir árboles binarios de búsqueda de alturas 2, 3, 4, 5 y 6.

EJERCICIO 4.7. Dé un algoritmo iterativo que imprima en orden las claves de un árbol binario de búsqueda.

EJERCICIO 4.8. Considerar el TAD **Tree** con operaciones:

$hoja$: **Int** $\rightarrow$ **Tree**
$nodo$: **Pair** $\times$ **Tree** $\times$ **Tree** $\times$ **Tree** $\rightarrow$ **Tree**

(Aquí $\mathbf{Pair} = \mathbf{Int} \times \mathbf{Int}$)

$izq, med, der : \mathbf{Tree} \to \mathbf{Tree}$

$valorNodo : \mathbf{Tree} \to \mathbf{Pair}$

$valorHoja : \mathbf{Tree} \to \mathbf{Int}$

Se consideran sólo árboles $T : Tree$ que satisfacen que, si $valor(T) = (i, j)$ entonces:
- $i < j$
- si n ocurre en una hoja de $izq(T)$, entonces $n \leq i$
- si n ocurre en una hoja de $med(T)$, entonces $i < n \leq j$
- si n ocurre en una hoja de $der(T)$, entonces $j < n$

Desarrollar un algoritmo iterativo logarítmico que determine si un entero dado n ocurre en una hoja de T.

EJERCICIO 4.9. Sean los siguientes algoritmos sobre ABB.

func *search*$(x : bintree, k : valor)$ **dev:** *bintree*
{post: devuelve un nodo cuyo valor es k, si existe. Si no devuelve NIL }
if $x =$ *NIL* $\vee\, k =$ *key*$[x]$ **then return** *x*
if $k <$ *key*$[x]$ **then return** *search*(*left*$[x], k)$
else return *search*(*right*$[x], k)$

func *min*$(x :$ *bintree*$)$ **dev:** *bintree*
{post: devuelve un nodo con clave mínima}
while *left*$[x] \neq$ *NIL* **do** $x :=$ *left*$[x]$
return x

func *max*$(x :$ *bintree*$)$ **dev:** *bintree*
{post: devuelve un nodo con clave máxima}
while *right*$[x] \neq$ *NIL* **do** $x :=$ *right*$[x]$
return x

(1) Probar que *search*, *min*, *max* tienen complejidad $\Theta(h)$, donde h es la altura del árbol.
(2) Escribir versiones recursivas de *min* y *max*.

EJERCICIO 4.10. Sea x un nodo en un ABB. El *siguiente* de x es un nodo distinto a x con un valor inmediato superior o igual al de x. Si tal nodo no existe, es indefinido. Análogamente, se define el *anterior* de x, cambiando la palabra "superior" por "inferior". Sea

func *sig*$(x :$ *bintree*$)$ **dev:** *bintree*
if *right*$[x] \neq$ *NIL* **then return** *min*(*right*$[x]$)
$y := p[x]$
while $y \neq$ *NIL* $\wedge\, x =$ *right*$[y]$ **do**
$x := y$
$y := p[y]$
return y

La notación es la del Ejercicio 4.4.

(1) Pruebe que *sig*(x) implementa siguiente.
(2) Implemente la función *ant*(x) implementa anterior.
(3) Probar que tanto *sig* como *ant* tienen complejidad $\mathcal{O}(h)$ donde h es la altura del árbol.

EJERCICIO 4.11. Suponga que tiene un ABB con etiquetas de 1 a 1000 y quiere encontrar el número 363. ¿Cuáles de las siguientes secuencias *no* puede ser una secuencia de nodos examinados?

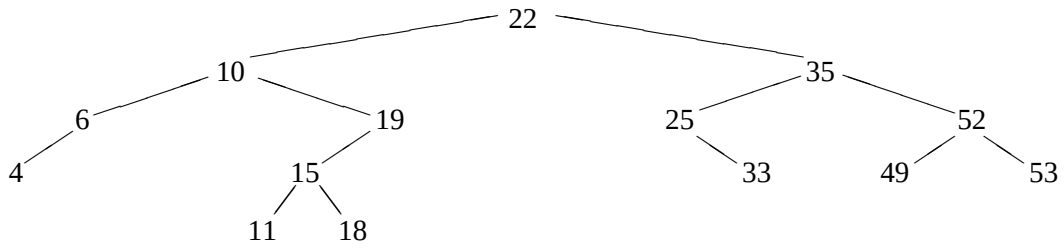
(1) 2, 252, 401, 398, 330, 344, 397, 363.
(2) 924, 220, 911, 244, 898, 258, 362, 363.
(3) 925, 202, 911, 240, 912, 245, 363.
(4) 2, 399, 387, 219, 266, 382, 381, 278, 363.
(5) 935, 278, 347, 621, 299, 392, 358, 363.

EJERCICIO 4.12. Sea T un ABB con todas las valores distintos. Sea x una hoja e y su padre. Probar que *key*$[y]$ es el mínimo valor en T con la propiedad *key*$[y] >$ *key*$[x]$ o bien es el máximo valore en T con la propiedad *key*$[y] <$ *key*$[x]$.

EJERCICIO 4.13. Recordemos que el método *insert* inserta un valor en un ABB. Podemos ordenar un conjunto de numeros construyendo un ABB con esos números (usando *insert* repetidas veces) y luego aplicando *recorridoInorder*.

(1) El peor caso de este algoritmo es cuando *insert* produce un árbol "lineal". Calcule la complejidad en este caso y de ejemplos de cuando ocurre.
(2) El mejor caso de este algoritmo es cuando *insert* produce un árbol balanceado. Calcule la complejidad en este caso y de ejemplos de cuando ocurre.

EJERCICIO 4.14. (1) Dada la secuencia de números 23, 35, 49, 51, 41, 25, 50, 43, 55, 15, 47 y 37 determinar el ABB resultante de realizar las inserciones de dichos números exactamente en ese orden a partir del ABB vacío.
(2) Dado el siguiente ABB determinar una secuencia de inserciones que pudiera haber dado lugar a dicho ABB.



EJERCICIO 4.15. ¿Cuál es el mínimo y el máximo número de elementos en un heap de altura h?

EJERCICIO 4.16. Probar que hay a lo más $n/2h + 1$ nodos de altura h en un heap de n elementos.

EJERCICIO 4.17. Probar que en cualquier subárbol de un heap, la raíz del subárbol es el valor más alto del subárbol.

EJERCICIO 4.18. Suponga que un heap tiene todos los elementos distintos. ¿Donde está ubicado el elemento mínimo del heap?

EJERCICIO 4.19. ¿Un arreglo ordenado en forma descendente es un heap?

EJERCICIO 4.20. ¿El arreglo $[23, 17, 14, 6, 13, 10, 1, 5, 7, 12]$ es un heap?

EJERCICIO 4.21. Sea $T[1..12]$ un arreglo dado por $T[i] = i$, para todo i. Muestre el estado de T después de cada una de las siguientes operaciones. Las operaciones se efectúan una después de otra, según el orden dado, y el output de una operación es el input de la siguiente.

i) Hacer un heap desde T.

ii) Se hace $T[10] := 12$, y luego se restablece la condición de heap.
iii) Idem *ii)*, pero haciendo $T[1] := 6$.
iv) Idem *ii)*, con $T[8] := 5$.

EJERCICIO 4.22. Probar que en la representación por arreglos de un heap, si el heap tiene tamaño n (es decir tiene n elementos), entonces las hojas tienen índices $n/2 + 1, n/2 + 2, \ldots, n$.

EJERCICIO 4.23. Recordar que el procedimiento *siftDown*(A, i) tiene como input un arreglo A y un entero i en el rango del arreglo. Cuando el procedimiento es llamado se supone que los árboles con raíces $A[2i]$ y $A[2i + 1]$, respectivamente, son heaps; pero que $A[i] < A[2i]$ o $A[i] < A[2i + 1]$. El procedimiento *siftDown* "hunde" al elemento $A[i]$ de tal forma que el nuevo árbol resultante con raíz en la coordenada i es ahora un heap. Hacer *siftDown*$(A, 3)$ donde $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$ y escribir como evoluciona el arreglo paso a paso.

EJERCICIO 4.24. Describir el efecto de hacer *siftDown*(A, i) cuando $A[i]$ es más grande que sus hijos.

EJERCICIO 4.25. Describir el efecto de hacer *siftDown*(A, i) cuando $i >$ *heapSize*$[A]/2$.

EJERCICIO 4.26. Recordar que el procedimiento *makeHeap*$(T[1..n])$ tiene como input un arreglo, al cual transforma (vía ciertas permutaciones) en un heap. El procedimiento es

proc *makeHeap*$(T[1..n])$
  for $i = 1$ **to** n **downto** 1 **do** *siftDown*(T, i)

Aplicar *makeHeap* al arreglo $A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$ y describir como va cambiando el arreglo antes de cada llamada de *siftDown*.

EJERCICIO 4.27. Ilustrar, usando grafos, la operación *heapSort* en el arreglo $A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$.

4.1. Ejercicios adicionales

EJERCICIO 4.28. Sea T un ABB y supongamos que k es una hoja de T. Cuando aplicamos el algoritmo de búsqueda a k, el algoritmo recorre un camino. Sea B los nodos del camino de búsqueda de k. Definimos A el conjunto de nodos de T a la izquierda de B y C el conjunto de nodos de T a la derecha de B. Demuestre la verdad o falsedad de la siguiente propiedad: dados $a \in A$, $b \in B$ y $c \in C$, entonces $a \leq b \leq c$.

EJERCICIO 4.29. Pruebe que si un nodo x de un ABB tiene dos hijos, entonces el siguiente de x no tiene hijo izquierdo y el anterior de x no tiene hijo derecho.

EJERCICIO 4.30. Sea T un ABB cuyos valores son todas distintos. Recordemos que si x es un nodo, entonces y es *ancestro* de x si $y = p^n(x)$ para $0 \leq n$ (recordemos que $p(x)$ es el padre de x). Pruebe que si el subárbol derecho de un nodo x en T es vacío e y es el siguiente de x, entonces y es el ancestro más pequeño de x cuyo hijo izquierdo es también un ancestro de x. Para la definición de siguiente ver Ejercicio 4.10.

EJERCICIO 4.31. Una recorrido inorder en un ABB puede ser implementado encontrando el mínimo elemento del árbol y después encontrando el siguiente repetidas veces. Implementar este algoritmo y demostrar que su complejidad es $\Theta(n)$.

EJERCICIO 4.32. Un ABB puede ser construido partiendo de un árbol vacío usando repetidas veces el método *insert*. Si x es un nodo, cuando insertamos x el algoritmo visita un cierto número de nodos de T, digamos k. Después que terminamos de construir el árbol, cuando hacemos *search*(x) este algoritmo visitará $k + 1$ nodos. Explique el por que.

EJERCICIO 4.33. Recordemos que el método *delete* borra un nodo de un ABB. Sea T una ABB. ¿La operación de borrado de nodos de T es conmutativa, en el sentido de que si borramos x y luego y el árbol resultante es igual al que resulta de borrar primero y y luego x? Si la respuesta es afirmativa, demuestre el resultado, de lo contrario dé un contraejemplo.

EJERCICIO 4.34. Sea T un ABB que admite nodos repetidos. Desarrolle un algoritmo que borre todas las ocurrencias de x en T. Ayuda: busque x mediante el algoritmo de búsqueda en ABB y luego elimine todas las ocurrencias de x del subárbol izquierdo. Finalmente proceda como el borrado normal.

EJERCICIO 4.35. Una implementación de *heapSort* es

proc *heapSort*(A)
makeHeap(A)
for $i =$ *length*$[A]$ **downto** 2 **do**
    swap$(A, 1, i)$
    heapSize$[A] =$ *heapSize*$[A] - 1$
    siftDown$(A, 1)$

Pruebe la corrección del algoritmo *heapSort* usando el siguiente invariante: al comienzo del ciclo **for** el subarreglo $A[1..i]$ es un heap que contiene el i-ésimo elemento más pequeño de $A[1..n]$ y el subarreglo $A[i+1..n]$ contiene los $n - i$ elementos más grandes de $A[1..n]$ y ellos están en orden.

EJERCICIO 4.36. Indique cuál es la complejidad de *heapSort* en arreglos que están ordenados en forma decreciente. Analice la complejidad cuando los arreglos están ordenados en forma creciente.

EJERCICIO 4.37. (Colas de prioridad) Una *cola de prioridad* es una estructura de datos que mantiene un conjunto y se suele implementar con un heap A. Las operaciones son:

- *insert(A,key)*: inserta el valor *key* en A.
- *maximum(A)*: devuelve el elemento de A de mayor valor.
- *extract(A)*: devuelve el mayor elemento de A y lo elimina de A (mantiene la propiedad de heap).
- *increaseKey*$(A, i,$ *key*$)$: cuando $i <$ *key*, incrementa el valor de $A[i]$ a *key*.

Una aplicación de las colas de prioridad es para darle prioridad a las tareas de un sistema operativo. Cada tarea a ser ejecutada tiene una prioridad (el valor en el heap) y cuando una tarea finaliza o es interrumpida la tarea de mayor prioridad se obtiene usando *extract(A)*. Una nueva tarea se puede agregar usando *insert(A,key)*. Se puede incrementar la prioridad de una tarea usando *increaseKey*$(A, i,$ *key*$)$.
Las siguientes son implementaciones de los procedimientos arriba descritos:

func *extract(A)* **dev:** *max*
    {pre: heapSize$[A] >= 1$*}*
    {post: devuelve el primer elemento del arreglo (el máximo) y
        lo elimina del heap. Con lo que queda rehace un heap}
        max $= A[1]$
        $A[1] = A[$*heapSize*$[A]]$
        heapSize$[A] =$ *heapSize*$[A] - 1$
        siftDown$(A, 1)$
        return *max*
proc *increaseKey*$(A, i,$ *key*$)$
    {pre: key $\geq A[i]$
    {post: incrementa el valor de $A[i]$ *a key y rehace el heap}*

$A[i] = key$
while $i > 1 \wedge A[parent(i)] < A[i]$ **do**
$swap(A, i, parent(i))$
$i := parent(i)$

proc *insert*(A, key)
{post: inserta el valor key en el heap A}
$heapSize[A] = heapSize[A] + 1$
$A[heapSize[A]] = -\infty$
$increaseKey(A, heapSize[A], key)$

(1) Calcule la complejidad de cada uno de los procedimientos.
(2) Ilustre como funciona $extract$ en el heap $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$.
(3) Ilustre como funciona $insert(A, 10)$ en el heap $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$.
(4) Pruebe la corrección del procedimiento *increaseKey* usando el siguiente invariante: al comienzo de cada iteración del ciclo **while** el arreglo $A[1..heapSize[A]]$ satisface la propiedad de heap excepto que puede ocurrir que $A[i]$ puede ser más grande que $A[parent(i)]$.

EJERCICIO 4.38. La operación *delete*(A, i) borra el nodo i en le heap A y conserva la propiedad de heap. Escriba una implementación de *delete* que tenga una complejidad $\mathcal{O}(log\, n)$ si n es el tamaño del heap.

EJERCICIO 4.39. Describa un algoritmo de complejidad $\mathcal{O}(n\, log\, k)$ que junte k listas ordenadas en forma descendente en una lista ordenada en forma descendente (Ayuda: use heaps).

EJERCICIO 4.40. Un *heap d-ario* es como un heap binario excepto (con una posible excepción) todos los nodos que no son hojas tienen d hijos en vez de 2.

(1) Represente una heap d-ario mediante un arreglo.
(2) Dado un heap d-ario de n elementos calcule la altura en términos de d y n.
(3) Implemente en forma eficiente *extract* en un heap d-ario. Analice su complejidad en términos de d y n.
(4) Implemente en forma eficiente *increaseKey*(A, i, k) en un heap d-ario, haciendo primero $A[i] =$ *max*$(A[i], k)$ y luego actualizando la estructura de heap d-ario en A. Analice complejidad del procedimiento en términos de d y n.

EJERCICIO 4.41. Un $m \times n$ *Young tableau,* o simplemente un *tableau*, es una matriz $m \times n$ tal que cada fila y cada columna está ordenada (en forma ascendente). Algunas de las entradas de un Young tableau pueden ser ∞ y veremos esas entradas como elementos inexistentes. Por lo tanto, un Young tableau puede ser usado para almacenar $r \leq nm$ números. Diremos que un Young tableau está *lleno* si tiene nm elementos.

(1) Dibuje un 4×4 Young tableau que contenga los elementos $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$
(2) Explique por qué un Young tableau Y es vacío si se cumple que $Y[1, 1] = \infty$. Explique por qué está lleno si $Y[m, n] < \infty$.
(3) Implemente el procedimiento *extractMin* que devuelve el mínimo elemento del tableau, lo elimina y rehace el tableau. Si el tableau es $m \times n$ la complejidad del algoritmo debe ser $\mathcal{O}(m + n)$. El algoritmo debería usar recursión y para resolver un problema $m \times n$ debe resolver un problema $m - 1 \times n$ o $m \times n - 1$ (ayuda: piense en *siftDown*).
(4) Muestre como insertar un nuevo elemento en un tableau que no está lleno.
(5) Sin usar otro método de ordenación como subrutina, muestre como con el uso de un $n \times n$ tableau es posible ordenar n^2 números con una complejidad $\mathcal{O}(n^3)$.

(6) Muestre un algoritmo de complejidad $\mathcal{O}(m+n)$ que determine si un determinado número está almacenado en un tableau de tamaño $m \times n$.

EJERCICIO 4.42. Considerar árboles binarios con valores booleanos en los nodos. Estos árboles pueden utilizarse para representar conjuntos de números binarios de la siguiente manera: El número binario 00101 está en el conjunto representado por el árbol T sii siguiendo el camino izquierdo (0), izquierdo (0), derecho (1), izquierdo (0), derecho (1) a partir de la raíz de T se llega a un nodo cuyo valor es true.

(1) Dar un algoritmo que inserte un número binario b en un árbol boolenano T.
(2) Dar un algoritmo que reciba dos de estos árboles T_1 y T_2 y devuelva el árbol que representa la unión de los conjuntos representados por T_1 y T_2.

AYED II – PRÁCTICA 5

Divide y vencerás

EJERCICIO 5.1. Ordene los siguientes arreglos ejecutando paso a paso los algoritmos de inserción, selección y merge.
a) [1,2,3,4,5], *b)* [5,4,3,2,1], *c)* [7,1,10,3,4,9,5].

EJERCICIO 5.2. Sea X un arreglo definido en $[1, n]$ y sea k un natural en ese intervalo. El *k-ésimo menor* elemento de X se define como el elemento que debería ocupar el lugar k en una permutación de X en la cual todos lo elementos están ordenados. Por ejemplo el 3-ésimo menor elemento de $[9, 10, 8, 8, 4, 4, 1, 3]$ es 4. Mediante $selection(X, k)$ denotamos al k-ésimo menor elemento de X. En particular, la *mediana* de X se define como $selection(X, \lceil n/2 \rceil)$. Suponga que se cuenta con un algoritmo $\mathcal{O}(n)$ que computa la mediana de X. Obtenga un algoritmo lo más efeciente posible que compute $selection(X, k)$. Efectue un análisis detallado de la complejidad.
(Nota: Un algoritmo trivial para $selection$ consiste en ordenar el arreglo, que es $\mathcal{O}(nlog(n))$. Por eficiente entendemos algo ”mejor” que ésto.)

EJERCICIO 5.3. Considere el problema de computar $selection$ (Ejercicio 5.2). Utilice la idea del pivot de quicksort para desarrollar un algoritmo que compute $selection(X, k)$. Calcule de manera detallada el orden del ”peor caso” de tal algoritmo.

EJERCICIO 5.4. Desarrolle de manera completa un algoritmo similar a *pivot* de quicksort pero que devuelve dos naturales i, j tales que todas las ocurrencias del pivot p quedan localizadas en $[i, j)$.

EJERCICIO 5.5. Se recibe un arreglo X de números reales definido en $[1, n]$ y un real x. Se quiere determinar si existen i, j en $[1, n]$ tal que $X[i] + X[j] = x$. Diseñe un algoritmo para resolver el problema cuya complejidad sea $\mathcal{O}(nlog(n))$.
(Ayuda: Existen métodos de sorting que son $\mathcal{O}(nlog(n))$, y hacer n veces una búsqueda binaria es también $\mathcal{O}(nlog(n))$.)

EJERCICIO 5.6. La *moda* de la secuencia $x_1, ..., x_n$ es el elemento que más se repite. Puede haber varios elementos que se repiten la misma cantidad (máxima) de veces, en tal caso cualquiera de ellos es considerado una moda. Se debe desarrollar un algoritmo que compute la moda de una secuencia dada en un arreglo. Utilice la idea proveniente del quicksort de separar según un pivot y luego resolver el problema para las dos subsecuencias más chicas. Note que si se parte un conjunto en dos partes disjuntas entonces es fácil recuperar la moda del conjunto en función de las modas de las partes. Calcule de manera detallada el orden del “peor caso” de tal algoritmo.

EJERCICIO 5.7. La versión de *partition* dada en la teórica no es el algoritmo original de T. Hoare (el creador de quicksort). El algoritmo original es:

func *partition*(A, p, r) **dev:** j
$x = A[p]$
$i = p - 1$

$j = r + 1$
while *true* **do**
repeat $j = j - 1$
until $A[j] \leq x$
repeat $i = i + 1$
until $A[i] \geq x$
if $i < j$ **then** *swap*(A, i, j) **else return** j

(a) Describir que hace *partition* en el arreglo $A = [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]$, mostrando los valores del arreglo y los valores auxiliares después de cada iteración del ciclo **while** .

Los tres ejercicios siguientes sirven para demostrar que el procedimiento *partition* es correcto. Pruebe los siguiente:

(b) Los índices i y j se encuentran en un rango tal que nunca acceden a un elemento fuera del subarreglo $A[p..r]$.
(c) Cuando *partition* termina devuelve un valor j tal que $p \leq j < r$.
(d) Cuando *partition* termina todo elemento de $A[p..j]$ es menor o igual que cada elemento de $A[j + 1..r]$.

Finalmente:

(e) Reescriba quicksort usando *partition*.

EJERCICIO 5.8. El algoritmo de quicksort explicado en clase tiene dos llamadas recursivas a él mismo. Después de aplicar *partition* el subarreglo de la izquierda es ordenado recursivamente. Luego se ordena recursivamente el subarreglo de la derecha. Esta segunda llamada recursiva no es realmente necesaria y puede ser evitada con el uso de iteraciones. Veamos una nueva versión de quicksort usando recursión simple:

proc *quicksortIt*(A, p, r)
$u := p$
while $u < r$ **do**
{partición y ordenamiento por izquierda}
$q =$ *partition*(A, u, r)
quicksortIt$(A, u, q - 1)$
$u = q + 1$

Pruebe la corrección del algoritmo encontrando un invariante para el ciclo **while** .

5.1. Ejercicios adicionales

EJERCICIO 5.9. Les proponemos que analicen el siguiente algoritmo de ordenación:

proc *ordenacionElegante*(A, i, j)
if $A[i] > A[j]$ **then** *swap*(A, i, j)
if $i + 1 \geq j$ **then return**
$k =$ *roundl*$((j - i + 1)/3)$ *{redondeo hacia abajo}*
ordenacionElegante$(A, i, j - k)$ *{primeros dos tercios}*
ordenacionElegante$(A, i + k, j)$ *{últimos dos tercios}*
ordenacionElegante$(A, i, j - k)$ *{de nuevo los primeros dos tercios}*

	Equipo				
Fecha	1	2	3	4	5
1	2	1	–	5	4
2	3	5	1	–	2
3	4	3	2	1	–
4	5	–	4	3	1
5	–	4	5	2	3

	Equipo					
Fecha	1	2	3	4	5	6
1	2	1	6	5	4	3
2	3	5	1	6	2	4
3	4	3	2	1	6	5
4	5	6	4	3	1	2
5	6	4	5	2	3	1

TABLA 1. Fixtures para 5 y 6 equipos

(1) De una explicación de por que si n es la longitud del arreglo A, entonces *ordenacionElegante*$(A, 1, n)$ ordena correctamente A.
(2) Analice la complejidad del algoritmo (peor caso) y compárela con las de insertion sort, merge sort, heapsort y quicksort. ¿Es un buen algoritmo?

EJERCICIO 5.10. Ordenamiento difuso de intervalos (fuzzy sort). Dados n intervalos cerrados $[a_i, b_i]$ ($1 \leq i \leq n$, $a_i \leq b_i$) el objetivo es conseguir un ordenamiento difuso de los intervalos, es decir producir una permutación $i_1, i_2, \ldots, i_n$ de los intervalos tal que existan $c_i \in [a_i, b_i]$ satisfaciendo la propiedad $c_1 \leq c_2 \leq \cdots \leq c_n$.

(1) Diseñe un algoritmo para hacer ordenamiento difuso de n intervalos. Las estructura general del algoritmo debe ser como el quicksort aplicado a los extremos izquierdos (es decir los a_i's) de los intervalos, pero debe sacar ventaja de las posibles superposiciones para mejorar el tiempo de ejecución. Observe que cuando más superposición de intervalos haya, el ordenamiento difuso de intervalos se hace cada vez más fácil. El caso extremo es en el que la intersección de todos los intervalos no es vacía, en ese caso la permutación trivial alcanza.
(2) La complejidad promedio de su algoritmo debe ser $\Theta(n \log n)$ (se calcula la complejidad promedio suponiendo que las particiones son balanceadas). Sin embargo, cuando hay mucha superposición el tiempo de ejecución debe ser menor. En particular la complejidad promedio debe ser $\Theta(n)$ cuando la intersección de todos los intervalos no es vacía (es decir cuando existe $x \in [a_i, b_i]$, para $1 \leq i \leq n$). El algoritmo no debe estar diseñado para tratar este caso en particular, sino que su eficiencia debe aumentar a medida que las superposiciones sean mayores.

EJERCICIO 5.11. Sea $T[1..n]$ un arreglo de n enteros. Un elemento de T se dice *mayoritario* si aparece más de $n/2$ veces en T. Dé un algoritmo que decida si en una arreglo $T[1..n]$ existe un elemento mayoritario y si existe el algoritmo lo encuentra e imprime. El algoritmo debe tener complejidad lineal en el pero caso.

EJERCICIO 5.12. Se está organizando un campeonato de fútbol con n equipos. Cada equipo debe jugar sólo una vez con cada uno de sus oponentes en cada una de las k fechas, excepto, quizás una fecha donde algún equipo puede tener fecha libre. Se desea diseñar un fixture, es decir una tabla donde se describe con quien tiene que jugar cada equipo y en qué fechas.

(1) Si n es una potencia de 2, dé un algoritmo que construya un fixture con $n - 1$ fechas.
(2) Para $n > 1$ entero arbitrario, dé un algoritmo que construya un fixture con $n - 1$ fechas si n es par y con n fechas si n es impar. Por ejemplo, en la Tabla 1 damos posibles fixtures para campeonatos de cinco y seis equipos.

EJERCICIO 5.13. Dados n puntos en el plano coordenado, encontrar un algoritmo capaz de encontrar el par de puntos más cercanos en tiempo $\mathcal{O}(n\,log\,n)$ en el peor caso.

EJERCICIO 5.14. Un *n-contador* es un circuito que toma n bits como input y produce $1 + [log\,n]$ bits de output (un bit puede ser 1 o 0). Lo que hace el n-contador es contar (en binario) el número de bits iguales a 1 en el input. Por ejemplo, si $n = 9$ y el input es 011001011, el output es 0101 (es decir el input tiene 5 unos). Un *(i, j)-sumador* es un circuito que recibe de input una cadena de i bits y otra de j bits, y devuelve como output una cadena de $[1 + \max(ij)]$ bits. El (i, j)-sumador suma los dos inputs en binario. Por ejemplo, si $i = 3$ y $j = 5$, y los inputs son 101 y 10111 respectivamente, el output es 011100. Es siempre posible construir un (i, j)-sumador usando exactamente $\max(i, j)$ 3-contadores. Es por ello que llamaremos, como suele hacerse, a un 3-contador un *sumador total*. Por ejemplo, para construir un $(3, 5)$-sumador harán falta 6 sumadores totales.

(1) Usando sumadores totales e (i, j)-sumadores com elementos primitivos, muestre como se construye en forma eficiente un n-contador.
(2) Dar la recurrencia y condiciones iniciales del número de sumadores totales necesarios para construir un n-contador. No olvide contar los sumadores totales necesarios para construir cada (i, j)-sumador.

EJERCICIO 5.15. (Shellsort) El algoritmo de ordenación shellsort fue creado en 1959 y es el primer algoritmo que mejoró sustancialmente, a nivel de eficiencia, la ordenación por inserción. Shellsort utiliza una secuencia $h_1 \leq h_2 \leq \cdots$ denominada *secuencia de incrementos*. Cualquier secuencia de incrementos es válida, con tal que $h_1 = 1$, pero algunas elecciones son mejores que otras. Supongamos que queremos ordenar un arreglo $a[1..n]$. Al comienzo del algoritmo se determina cuales incrementos utilizar, es decir se determina un t para utilizar luego $h_t, \cdots, h_2, h_1$ en ese orden. En la primera "pasada" de shellsort tendremos que $a[i] \leq a[i + h_t]$ para todo i en que tenga sentido la comparación. Es decir, todos los elementos separados por una distancia h_t están ordenados. La siguiente pasada se hace con h_{t-1} y así sucesivamente. En la pasada que se hace usando h_k se logra que $a[i] \leq a[i + h_k]$ para todo i en que tenga sentido la comparación. Es decir, todos los elementos separados por una distancia h_k están ordenados. Se dice entonces que el vector está *h_k-ordenado*. En cada pasada la ordenación que se realiza se hace por inserción. Claramente, después de la última pasada, es decir cuando se usa $h_1 = 1$, se obtiene un vector ordenado (estar 1-ordenado es estar ordenado).

(1) Tomando $h_1 = 1$, $h_2 = 3$ y $h_3 = 5$ y $a = [81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15]$, describir como se comporta shellsort en cada pasada.
(2) Implemente shellsort usando la secuencia $h_i = 2^i - 1$.

La complejidad del shellsort es un problema abierto y sólo se ha podido calcular para algunas secuencias de incrementos.

(3) Implemente shellsort usando la secuencia $h_i = 2^{i-1}$. Usando esta implementación, la de ordenación por inserción y la de (2), compare experimentalmente los tiempos de ejecución de: ordenación por inserción, shellsort con incrementos $h_i = 2^i - 1$ y shellsort con incrementos $h_i = 2^{i-1}$.

AYED II – PRÁCTICA 6

Algoritmos greedy

EJERCICIO 6.1. Demuestre los siguientes hechos sobre grafos:

(1) Sea G un grafo no dirigido. Pruebe que la suma de los grados (valencias) de sus vértices es exactamente el doble del número de aristas.
(2) Sea G un grafo no dirigido. Pruebe que si los grados de sus vértices son todos mayores o iguales a 2 entonces hay un ciclo.
(3) Un grafo no dirigido, conexo y de n vértices tiene al menos $n - 1$ aristas.
(4) Un *árbol* (es decir un grafo conexo y acíclico) de n vértices tiene $n - 1$ aristas.
(5) Si $G = (N, A)$ es un grafo conexo y $l : A \to$ **nat** es una función $longitud$ definida sobre las aristas entonces existe una árbol T generador de G de longitud mínima.
(6) Si G es un grafo dirigido acíclico entonces existe un vértice de G que tiene valencia entrante 0.

EJERCICIO 6.2. Un coloreo de un grafo consiste en asignarles colores a sus vértices de manera que dos adyacentes tengan siempre colores distintos. El algoritmo greedy para coloreo consiste en fijar un orden $v_1, v_2, ..., v_n$ para los vértices y luego colorear v_i con el primer color no utilizado en los adyacentes a v_i que estén en $v_1, ..., v_{i-1}$.

(1) Pruebe que el algoritmo greedy no siempre arroja la solución óptima.
(2) Pruebe que si cada vértice tiene grado menor o igual a k, entonces $k + 1$ colores son suficientes para colorearlo. ¿Serán suficientes k colores?

EJERCICIO 6.3. Ejecute paso por paso los algoritmos de Prim y Kruskal para el grafo $G = (N, A)$ y los pesos w. Aquí $N = [1, 8]$ y las aristas con sus pesos son:
$w\{1,2\} = 7,\ w\{1,7\} = 5,\ w\{1,3\} = 3,\ w\{3,8\} = 6,\ w\{8,5\} = 2,\ w\{2,4\} = 2,\ w\{4,6\} = 8,\ w\{5,6\} = 6,\ w\{6,7\} = 5,\ w\{3,6\} = 4,\ w\{2,5\} = 1,\ w\{3,4\} = 5,\ w\{2,3\} = 4,\ w\{8,7\} = 3,\ w\{5,4\} = 3,\ w\{1,6\} = 3.$

EJERCICIO 6.4. Efectúe la implementación del algoritmo de Prim utilizando (para encontrar la mínima arista con origen en T y destino fuera de T) dos arreglos en los que se guarde para cada vértice fuera de T su adyacente en T más cercano (si hay) y la longitud de tal arista. Luego haga de manera detallada el análisis de eficiencia.

EJERCICIO 6.5. ¿Cómo se ejecutan los algoritmos de Prim y Kruskal en caso de que el grafo en cuestión no sea conexo?

EJERCICIO 6.6. Efectúe el análisis detallado de la complejidad del algoritmo Kruskal.

EJERCICIO 6.7. Ejecute paso por paso el algoritmo de Dijkstra para el grafo dirigido $G = (N, A)$ y los pesos w. Aquí $N = [1, 8]$ y las aristas con sus pesos son:

$w(1,2) = 7,\ w(1,7) = 5,\ w(1,3) = 3,\ w(3,8) = 6,$
$w(8,5) = 2,\ w(2,4) = 2,\ w(4,6) = 8,\ w(5,6) = 6,$

$w(6,7) = 5,\ w(3,6) = 4,\ w(2,5) = 1,\ w(3,4) = 5,$
$w(2,3) = 4,\ w(8,7) = 3,\ w(5,4) = 3,\ w(1,6) = 3.$

Tome como origen al vértice 1. Dé en cada paso los datos $S =$ conjunto de los vértices para los cuales ya se conoce el mínimo camino, $D =$ vector de los mínimos caminos especiales.

EJERCICIO 6.8. Dado un grafo conexo no dirigido G la distancia entre dos nodos x e y es el costo del camino de menor costo que une x con y. El diámetro del grafo es la distancia máxima existente en el grafo (es decir, la distancia que separa los nodos más distantes).

(1) Escribir un algoritmo que utilice el algoritmo de Floyd para calcular el diámetro de un grafo cualquiera. Escribir todos los algoritmos involucrados.
(2) Escribir un algoritmo que resuelva el problema en orden cuadrático para un grafo conexo *acíclico*. (Ayuda: una forma de resolverlo guarda similitud con el algoritmo de Prim).

EJERCICIO 6.9. Suponga que un grafo $G = (V, E)$ es representado por una matriz de adyacencia. Dé una implementación del algoritmo de Prim con complejidad $\mathcal{O}(|V|^2)$.

EJERCICIO 6.10. Se propone el siguiente algoritmo de tipo divide y vencerás para calcular MST. Dado un grafo $G = (V, E)$, partimos el conjunto de vértices V en dos conjuntos disjuntos V_1 y V_2 tal que $|V_1|$ y $|V_2|$ difieren a lo más en uno. Sea E_1 el conjunto de aristas cuyos vértices pertenecen a V_1 y, en forma análoga, E_2 el conjunto de aristas cuyos vértices pertenecen a V_2. Luego, encontramos en forma recursiva el MST de cada uno de los subgrafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$. Finalmente, seleccionamos una arista de peso mínimo con un vértice en V_1 y el otro en V_2, y usamos esa arista par unir los dos MST en un MST de G. Si el algoritmo calcula correctamente el MST, dar una demostración. Si el algoritmo no es correcto, encuentre un contraejemplo.

EJERCICIO 6.11. Para la versión del problema de la mochila sin fragmentación de objetos demostrar que el algoritmo voraz no siempre halla la solución óptima. Para ello, primero modificar el algoritmo voraz de manera de que no permita dicha fragmentación y luego encontrar un ejemplo para el que el algoritmo obtenido no halla la solución óptima.

6.1. Ejercicios adicionales

EJERCICIO 6.12. Sea $D_S[1, n]$ (recordar el algoritmo de Dijkstra) el vector que almacena la longitud del mínimo camino desde el origen a cualquier otro vértice, cuyos nodos intermedios están en S (i.e. camino *especial)*. Sea v que minimiza $D_S[x]$, y sea $w \neq v$ un vértice fuera de S tal que $D_{S\cup\{v\}}[w] \neq \infty$. Note que por definición de D existe un camino desde el origen de longitud $D_{S\cup\{v\}}[w]$ que utiliza vértices en $S\cup\{v\}$. Probar que en realidad existe un camino con esas características que satisface una de las condiciones siguientes:

(1) no pasa por v,
(2) pasa por v sólo un paso antes de llegar a w.

EJERCICIO 6.13. Modificar el algoritmo de búsqueda del camino de costo mínimo de Dijkstra:

(1) de modo que sólo considere caminos tales que todas sus aristas sean de costo estrictamente positivo:

func *Dijkstra*$_0(L$: **array**$[1..n, 1..n]$ **of** $costo)$ **dev:** **array**$[2..n]$ **of** $costo$
$C := \{2, 3, \ldots, n\}$
{*inicialización de* D_0 }

{ *ciclo greedy*}
return D_0

Un invariante del ciclo debe ser que $D_0[i]$ mantiene el costo del camino especial de costo mínimo que une 1 con i pasando sólo por aristas de costo estrictamente positivo (o $+\infty$ si no hay un tal camino). Un camino es especial si ninguno de sus nodos intermedios pertenece a C. No puede modificarse la matriz L.

(2) de modo que sólo considere caminos tales que a lo sumo una de sus aristas sea de costo nulo:

func *Dijkstra*$_1(L$: **array**$[1..n, 1..n]$ **of** $costo)$ **dev:** **array**$[2..n]$ **of** $costo$
var D_0, D_1 : **array**$[2..n]$ **of** *costo*
$C := \{2, 3, \ldots, n\}$
{*inicialización de* D_0 *(idéntica a ejercicio anterior)*}
{*primer ciclo greedy (idéntica a ejercicio anterior)*}
$C := \{2, 3, \ldots, n\}$
{*inicialización de*D_1}
{*segundo ciclo greedy*}
for $i := 2$ **to** n **do** $D[i] := \min(D_0[i], D_1[i])$
return D

Un invariante del segundo ciclo debe ser que $D_1[i]$ mantiene el costo del camino especial de costo mínimo que une 1 con i pasando exactamente una vez por una arista de costo nulo (o $+\infty$ si no hay un tal camino). Recomendaciones: 1) en la inicialización de D_1 y en el segundo ciclo utilizar D_0, pero no modificarlo; 2) en el segundo ciclo, seleccionar v que minimice $\{D_0[v]|v \in C\} \cup \{D_1[v]|v \in C\}$, luego proceder por casos según sea $D_0[v] \leq D_1[v]$ o no.

En ambos ejercicios, se asume que todas las aristas tienen costo no negativo.

EJERCICIO 6.14. El algoritmo de Kruskal puede devolver diferentes MST para un grafo dado G. Probar que para cada T un MST de G, existe una forma de ordenar las aristas de G en el algoritmo de Kruskal de tal forma de que el algoritmo devuelva T.

EJERCICIO 6.15. Suponga que los pesos de las aristas en un grafo $G = (V, E)$ son enteros que se encuentran en el rango de 1 a $|V|$. ¿Cuán rápido puede hacer el algoritmo de Kruskal? ¿Qué puede decir si los pesos de las aristas son enteros en el rango de 1 a C, para alguna constante C?.

EJERCICIO 6.16. Suponga que los pesos de las aristas en un grafo $G = (V, E)$ son enteros que se encuentran en el rango de 1 a $|V|$. ¿Cuán rápido puede hacer el algoritmo de Prim? ¿Qué puede decir si los pesos de las aristas son enteros en el rango de 1 a C, para alguna constante C?.

EJERCICIO 6.17. (El segundo mejor MST) Sea $G = (V, E)$ un grafo conexo no dirigido con función de peso $w : E \to \mathbb{R}$ y supongamos que $|E| \geq |V|$ y que todos los pesos son distintos. El segundo mejor MST se define como sigue: Sea $\mathcal{T}$ el conjunto de árboles expandidos de G y sea S un MST de G. Entonces, el *segundo mejor MST* es un árbol de expansión T tal que $w(T) = \min_{T' \in \mathcal{T} - \{S\}}\{w(T')\}$.

(1) Probar que el MST es único, pero que el segundo mejor MST puede no serlo.
(2) Sea T un MST de G. Probar que existen aristas $(u, v) \in T$ y $(x, y) \notin T$ tal que $T - \{(u, v)\} \cup \{(x, y)\}$ es un segundo mejor MST de G.
(3) Sea T un árbol expandido de G. Si $u, v \in V$, definimos $\max[u, v]$ como la arista de mayor peso en el único camino en T de u a v. Describir un algoritmo $\mathcal{O}(|V|^2)$ que, dado T, calcule $\max[u, v]$ para todo $u, v \in V$.

(4) Dé un algoritmo eficiente que calcule el segundo mejor MST de G.

EJERCICIO 6.18. (Árboles de expansión cuello de botella) Sea G un grafo conexo no dirigido. Un *árbol de expansión cuello de botella* T es un árbol expandido de G cuya arista de peso máximo es mínima entre todos los árboles expandidos. Diremos que el *valor* de un árbol de expansión cuello de botella T es el peso de la arista de peso máximo en T.

(1) Explique por que un MST es un árbol de expansión cuello de botella.

El item (1) muestra que encontrar un árbol de expansión cuello de botella no es más difícil (costoso) que encontrar un MST. En los items siguientes veremos que existe un algoritmo de tiempo lineal para encontrar un árbol de expansión cuello de botella.

(1) Encuentre un algoritmo de tiempo lineal que dado un grafo G y un entero b, determine si el valor del árbol de expansión cuello de botella es a lo sumo b.
(2) Use su algoritmo de la parte (1) como una subrutina en un algoritmo de tiempo real para solucionar el problema del árbol de expansión cuello de botella.

EJERCICIO 6.19. En este ejercicio le daremos tres algoritmos diferentes. Cada uno de ellos tiene como input un grafo y su función de pesos y devuelve un conjunto de aristas T. Para cada algoritmo usted debe probar que o bien T es un MST, o bien no lo es. También debe calcular cual es el algoritmo más eficiente (calcule o no el MST).

proc *quizasMST-A*(G, w)
  $\{$*ordena las aristas de forma decreciente no estricta respecto a los pesos*$\}$
  $T := E$
  for *cada arista* e*, tomadas en orden* **do**
    if $(T - \{e\}$ *es un grafo conexo*$)$ **then** $T := T - \{e\}$
  return T

proc *quizasMST-B*(G, w)
  $T := \emptyset$
  for *cada arista* e*, tomadas en orden arbitrario* **do**
    if $(T \cup \{e\}$ *no tiene ciclos*$)$ **then** $T := T \cup \{e\}$
  return T

proc *quizasMST-C*(G, w)
  $T := \emptyset$
  for *cada arista* e*, tomadas en orden arbitrario* **do**
    $T := T \cup \{e\}$
    if T *tiene un ciclo* $c)$ **then**
      sea e' *una aristas de peso máximo en* c
      $T := T - \{e'\}$
  return T

AYED II – PRÁCTICA 7

Programación dinámica

EJERCICIO 7.1. Considere el problema de dar cambio. Pruebe o dé un contraejemplo: si cada moneda tiene como valor el doble (por lo menos) de la anterior, y la moneda de menor valor es 1, entonces el algoritmo greedy arroja una solución óptima.

EJERCICIO 7.2. Considere el problema de *dar cambio*. Pruebe o dé un contraejemplo: si los valores de las monedas son $1, p, p^2, ..., p^n$, con p mayor que 1, entonces el algoritmo greedy arroja una solución óptima.

EJERCICIO 7.3. Para el siguiente caso, dar la matriz que se genera usando programación dinámica en la solución de *dar cambio*. Las denominación de las monedas es 2,5,6. Se debe dar vuelto de 13. Recupere luego las soluciones (si existen) para cada caso.

EJERCICIO 7.4. Resuelva el problema *dar cambio* asumiendo que existe una cantidad fija de monedas de cada denominación. Resuelva luego el problema para el caso de tener 3 monedas con denominaciones 2,5,6. Recupere luego las soluciones (si existen) para cada caso.

EJERCICIO 7.5. Dadas n monedas y un arreglo de naturales positivos $d[1..n]$ con las denominaciones de cada una de ellas (las denominaciones pueden repetirse), escribir un algoritmo que determine si existe alguna manera de separar las n monedas en 2 partes de igual monto.

EJERCICIO 7.6. Utilice programación dinámica para resolver el siguiente problema. Se disponen de n números naturales dados en un arreglo $X[1..n]$ y un número K, y se debe determinar si existen $i_1 < \ldots < i_k$ tales que

$$X[i_1] + ... + X[i_k] = K.$$

EJERCICIO 7.7. Usted se encuentra en un extraño país donde las denominaciones de la moneda son 15, 23 y 29. Antes de regresar quiere comprar un recuerdo pero también quiere conservar el mayor número de monedas posibles. Los recuerdos cuestan 68, 74, 75, 83, 88 y 89. Asumiendo que tiene suficientes monedas para comprar cualquiera de ellos, ¿cuál de ellos elegirá? ¿qué monedas utilizará para pagarlo? Justifique claramente y mencione el método utilizado.

EJERCICIO 7.8. Se modifica el problema de las monedas de la siguiente manera: hay denominaciones que suelen ser difíciles de conseguir (por ejemplo, las monedas de 1 centavo) y otras que son fáciles de conseguir.

(1) Asumiendo que es difícil conseguir monedas de denominación d_1, ..., d_x y es fácil conseguir las de denominación d_{x+1}, ..., d_n, hallar un algoritmo que utilice programación dinámica para minimizar el número de monedas difíciles necesarias para pagar un determinado monto.
(2) Asumiendo que el costo de conseguir monedas de denominación d_i es c_i, hallar un algoritmo que utilice programación dinámica para minimizar el costo total necesario para pagar un determinado monto.

EJERCICIO 7.9. Una fábrica de automóviles tiene dos líneas de ensamblaje y cada línea tiene n estaciones de trabajo, digamos $S_{1,1}, \ldots, S_{1,n}$ para la primera y $S_{2,1}, \ldots, S_{2,n}$ para la segunda. Dos estaciones $S_{1,i}$ y

$S_{2,i}$ (para $i = 1, \ldots, n$), hacen el mismo trabajo, pero lo hacen con costos $a_{1,i}$ y $a_{2,i}$ respectivamente, que pueden ser diferentes. Para fabricar un auto debemos pasar por n estaciones de trabajo $S_{i_1,1}, S_{i_2,2} \ldots, S_{i_n,n}$ no necesariamente todas de la misma linea de montaje ($i_k = 1, 2$). Si estamos trabajando en la estación $S_{i,j}$, transferirnos a la otra linea de montaje (es decir continuar en $S_{i',j+1}$ con $i' \neq i$) cuesta $t_{i,j}$. Teniendo en cuenta estos datos, desarrolle un algoritmo para encontrar las estaciones que deben ser elegidas para fabricar un auto con costo mínimo. El algoritmo no puede tener complejidad exponencial . Ayuda: programación dinámica.

EJERCICIO 7.10. (Moviéndose en un tablero de damas) Supongamos que tenemos un tablero de damas de tamaño $n \times n$ y tenemos una ficha. Debemos mover la ficha desde el extremo inferior del tablero hasta el extremo superior del mismo. Los movimientos deben hacerse siguiendo las siguientes reglas: en cada paso la ficha puede moverse a una de las siguientes casillas

- la casilla que está inmediatamente arriba,
- la casilla que está uno arriba y uno a la izquierda (si la ficha no está en la columna extrema izquierda),
- la casilla que está uno arriba y uno a la derecha (si la ficha no está en la columna extrema derecha).

Cada vez nos movamos de la casilla x a la casilla y recibiremos la cantidad de $p(x, y)$ pesos. Se nos ha dado la tabla de todos los $p(x, y)$ para los pares (x, y) donde moverse de x a y es legal. No asumimos que $p(x, y)$ sea positivo. Dar un algoritmo que mueva la ficha desde el extremo inferior del tablero hasta el extremo superior del mismo tal que maximice la cantidad de dinero recibido. El algoritmo puede elegir para iniciar cualquier casilla del borde inferior y puede terminar en cualquier casilla del borde superior. Analice la complejidad del algoritmo.

EJERCICIO 7.11. Una institución planea hacer un almuerzo para celebrar sus primeros 50 años. La institución tiene una estructura jerárquica en forma de árbol con raíz de tal forma que el presidente es la raíz del árbol. La oficina de personal ha asignado a cada empleado un puntaje el "rating de la buena mesa" que es un número real. Con el propósito de hacer el festejo más agradable a los comensales se ha decidido que no asistan simultáneamente un empleado y su jefe inmediato. Escribir un algoritmo que imprima una lista de invitados al festejo, maximizando la suma de los ratings de la buena mesa. El árbol que describe la estructura jerárquica de la empresa está dado por la siguiente estructura (eldest child, next sibling):

type *treenode* = **record**
nombre : **string**
ranking : **double**
hijoIzquierdo : ↑*treenode*
proximoHermano : ↑*treenode*

Analice la complejidad del algoritmo.

7.1. Ejercicios adicionales

EJERCICIO 7.12. Una fábrica utiliza materia prima de dos tipos: A y B. Dispone de una cantidad MA y MB de cada una de ellas. Tiene a su vez pedidos de fabricar n productos $p_1, \ldots, p_n$ (uno de cada uno). Cada uno de ellos tiene un valor de venta $v_1, \ldots, v_n$ y requiere para su fabricación cantidades $a_1, \ldots, a_n$ de materia prima de tipo A y $b_1, \ldots, b_n$ de materia prima de tipo B. Desarrollar un algoritmo que calcula el mayor valor alcanzable con las cantidades de materia prima disponible. Explicar detalladamente el funcionamiento del algoritmo.

EJERCICIO 7.13. Se recibe un arreglo $X[0, n)$ y se debe determinar la longitud de la subsecuencia creciente más larga. Una *subsecuencia creciente* es una subsecuencia de la forma $X[i_1] \leq ... \leq X[i_k]$, donde $0 \leq i_1 < ... < i_k < n$. Utilice un argumento inductivo para obtener la solución y luego obtenga un programa utilizando programación dinámica.
(Referencia recomendada: Mamber, 6.11.1)

EJERCICIO 7.14. Dadas dos secuencias $X = x_1x_2 \dots x_n$ e $Y = y_1y_2 \dots y_m$ diremos que una secuencia Z es una *subsecuencia común máxima* (SCM) de X e Y si es subsecuencia de X e Y y tiene la mayor longitud posible. Una subsecuencia puede no ser de letras consecutivas, pero si debe conservar el orden. Por ejemplo s**pr**i**n**gtim**e** y **pi**o**ne**er tienen como subsecuencia común de longitud máxima a la palabra pine (las subsecuencias en cada palabra son las formadas por letras en negrita).

(1) De un algoritmo recursivo que encuentre la SCM. Calcule su complejidad.
(2) Diseñe un algoritmo que resuelva SCM con programación dinámica y calcule su complejidad.
(3) Diseñe un algoritmo que resuelva SCM con funciones de memoria y calcule su complejidad.

EJERCICIO 7.15. Se tiene una representación de la ciudad como un grafo dirigido donde los $n > 0$ nodos representan ciertos puntos clave y las aristas tienen pesos que indican el tiempo, medido en minutos, que necesita una ambulancia para dirigirse de un punto clave a otro adyacente. Sabiendo que los k hospitales (con $k \leq n$) se encuentran en los puntos claves $1, \dots, k$ se pide dar un algoritmo que calcula la distancia al punto clave más alejado del sistema hospitalario. Dar también un algoritmo que encuentra el punto clave más alejado. Justificar.

EJERCICIO 7.16. Dadas dos matrices A y B de dimensiones $n \times m$ y $m \times p$ respectivamente, la multiplicación usual de matrices requiere $n \times m \times p$ multiplicaciones escalares. Dadas $A_1, A_2, \dots, A_n$ matrices tal que A_i tiene dimensión $p_{i-1} \times p_i$ entonces podemos multiplicar las matrices en ese orden. Sin embargo la forma de agrupar (asociar) que usamos para multiplicar las matrices es relevante a la hora de contar la cantidad de multiplicaciones escalares que realizamos. Por ejemplo, sean A_1 de 2×3, A_2 de 3×6 y A_3 de 6×5, entonces hacer A_1A_2 requiere $2 \times 3 \times 6 = 36$ multiplicaciones y nos queda una matriz de 2×6, luego hacer $(A_1A_2)A_3$ requiere $36 + 2 \times 6 \times 5 = 96$ multiplicaciones. Haciendo un cálculo análogo vemos que hacer A_2A_3 requiere $3 \times 6 \times 5 = 90$ multiplicaciones y $A_1(A_3A_3)$ requiere $90 + 2 \times 3 \times 5 = 120$ multiplicaciones. El problema de la *multiplicación de una cadena de matrices* es encontrar la forma de asociar los productos entre matrices de tal manera que minimicemos la cantidad de operaciones escalares.

(1) Resuelva el problema de la multiplicación de una cadena de matrices usando recursión y calcule la complejidad.
(2) Resuelva el problema de la multiplicación de una cadena de matrices usando programación dinámica y calcule la complejidad.
(3) Resuelva el problema de la multiplicación de una cadena de matrices usando funciones de memoria y calcule la complejidad.

EJERCICIO 7.17. (El problema bitónico del viajante) El problema del viajante es el problema de determinar el recorrido cerrado más corto entre n puntos en el plano. El problema general es *NP*-completo y, por consiguiente, se cree que no puede ser solucionado en tiempo polinomial. El problema puede ser simplificado si requerimos que los recorridos sean bitónicos, esto es, comenzamos los recorridos en el extremo izquierdo, después siempre vamos a la derecha hasta llegar al punto extremo derecho y a partir de ahí nos movemos siempre hacia la izquierda hasta llegar de nuevo al punto de inicio (el extremo izquierdo). En este caso es posible encontrar un algoritmo de tiempo polinomial. Describa un algoritmo de complejidad $\mathcal{O}(n^2)$ que determine un recorrido bitónico óptimo. Asuma que no hay dos puntos con la misma coordenada x.

EJERCICIO 7.18. (Imprimiendo correctamente) Considere el problema de imprimir correctamente un parágrafo con una impresora. El texto es una sucesión de n palabras de longitudes $l_1, l_2, \ldots, l_n$, medidas en caracteres. Se quiere imprimir el parágrafo en líneas que tienen un máximo de M caracteres cada una. Si una linea contiene las palabras de la i a la j y dejamos exactamente un espacio entre palabra y palabra, el número de espacios extra al final de cada línea es $S = M - j + i - \sum_{k=1}^{j} l_i \geq 0$. Nuestro criterio de "corrección" es el siguiente: queremos minimizar la suma, sobre todas las líneas excepto la última, de los cubos de los números de espacios extra al final de cada línea.

(1) Usando programación dinámica dé un algoritmo que imprima correctamente un párrafo en una impresora.
(2) Analice la complejidad del algoritmo.

EJERCICIO 7.19. (Algoritmo de Viterbi) Para reconocimiento del habla (speech recognition) podemos usar programación dinámica sobre un grafo dirigido $G = (V, E)$. Cada arista $(u, v) \in E$ está etiquetada con un sonido $\sigma(u, v)$ dentro de un conjunto finito de sonidos Σ. El grafo etiquetado es un modelo formal de una persona hablando un lenguaje natural. Dado un vértice distinguido $v_0 \in V$, cada camino que comienza en v_0 corresponde a una sucesión de sonidos producidos por el modelo. Definimos la etiqueta de un camino dirigido como la concatenación de las etiquetas de las aristas del camino.

(1) Describir un algoritmo eficiente que dada una sucesión $s = \sigma_1\sigma_2 \ldots \sigma_k$ de caracteres de Σ devuelva un camino en G que comience en v_0 y tenga a s como etiqueta, si tal camino existe. Si el camino no existe el algoritmo debe devolver un mensaje significativo (por ejemplo, "No existe el camino"). Analice la complejidad del algoritmo.

Suponga ahora que cada arista $(u, v) \in E$ tiene asociada una probabilidad $0 \leq p(u, v) \leq 1$. Este número indica la probabilidad de que a partir del vértice u vayamos a v, y por consiguiente se produzca el sonido $\sigma(u, v)$. La suma de las probabilidades de las aristas que parten de un vértice dado es 1. La probabilidad de un camino se define como el producto de las probabilidades de sus aristas. Podemos ver a la probabilidad de un camino que comienza en v_0 como la probabilidad de que una "caminata al azar" que comienza en v_0 siga el camino especificado.

(2) Extienda el resultado de (1) de tal forma que el camino devuelto sea el camino con máxima probabilidad que comienza en v_0 y tiene etiqueta s.

EJERCICIO 7.20. (Distancia de edición) El objetivo es "transformar" una cadena $X[1..m]$ en la cadena $Y[1..n]$ usando un conjunto de operaciones que definimos más adelante. Usaremos un arreglo Z (de longitud conveniente) para almacenar los resultados intermedios. Inicialmente Z está vacío y al finalizar $Z[j] = Y[j]$ para $j = 1, 2, \ldots, n$. Debemos mantener en memoria posiciones i en X y j en Z. Sólo está permitido que las operaciones modifiquen Z y esos índices. Inicialmente $i = j = 1$. Se requiere que se debe examinar todos los caracteres de X durante la transformación, es decir que al terminar el programa i deberá ser igual a $m + 1$ y deberá haber tomado todos los valores entre 1 y m. Hay seis operaciones posibles:

- **Copia** un carácter de X a Z haciendo $Z[j] := X[i]$ e incrementando en uno i y j. Esta operación examina $X[i]$.
- **Reemplaza** un carácter de X por otro carácter c haciendo $Z[j] := c$ e incrementando en uno i y j. Esta operación examina $X[i]$.
- **Borra** un carácter de X incrementando en uno i y dejando j igual. Esta operación examina $X[i]$.
- **Inserta** el carácter c en Z haciendo $Z[j] := c$ incrementando en uno j y dejando i igual. Esta operación no examina ningún carácter de X.

- **Intercambia** las dos caracteres siguientes de X haciendo una copia de ellos a Z pero invirtiendo el orden. Esto se hace mediante $Z[j] := X[i+1]$ y $Z[j+1] := X[i]$ y entonces poniendo $i := i+2$ y $j := j+2$. Esta operación examina $X[i]$ y $X[i+1]$.
- **Elimina** lo que resta de X haciendo $i := m+1$. Esta operación examina todos los caracteres de X que aún no habían sido examinados. Cuando se usa esta operación el programa termina.

A continuación, para clarificar, daremos un ejemplo. Convertiremos la secuencia *algorithm* a la secuencia *altruistic* donde los caracteres subrayados son $X[i]$ y $Z[j]$ después de la operación:

Operación	X	Z
	algorithm	_
copia	algorithm	a_
copia	algorithm	al_
reemplaza por t	algorithm	alt_
borra	algorithm	alt_
copia	algorithm	altr_
inserta u	algorithm	altru_
inserta i	algorithm	altrui_
inserta s	algorithm	altruis_
intercambia	algorithm	altruisti_
inserta c	algorithm	altruistic_
elimina	algorithm_	altruistic_

Observe que puede haber otras secuencias de transformaciones que transformen algorithm en altruistic. Cada una de las operaciones tiene un costo asociado que asumimos es una constante conocida por nosotros. Asumimos también que los costos individuales de copia y reemplazo son menores que los costos combinados de las operaciones borrar e insertar. De otra forma las operaciones copia y reemplaza no serían utilizadas. El costo de una secuencia de operaciones es la suma de los costos de las operaciones individuales en la secuencia. Por ejemplo, la transformación de algorithm a altruistic realizada más arriba tiene un costo

$$3\,costo(\text{copia}) + costo(\text{reemplaza}) + costo(\text{borra}) + 4\,costo(\text{inserta}) + costo(\text{intercambia}) + costo(\text{elimina}).$$

(1) Dadas dos secuencias $X[1..m]$ e $Y[1..n]$ y un conjunto de operaciones y costos, la *distancia de edición* de X a Y es el costo de la secuencia menos costosa de operaciones que transforma X en Y. Escriba un algoritmo con la técnica de programación dinámica que encuentre la distancia de edición entre X e Y e imprima la secuencia óptima. Analice la complejidad del algoritmo y los requerimientos de espacio que él tiene.

El problema de la distancia de edición es la generalización del problema de alinear dos secuencias de ADN. Hay varios métodos para medir la similitud de dos secuencias de ADN haciendo alineaciones entre ellas. Uno de los métodos es el siguiente: dadas dos secuencias de ADN, digamos X e Y, se insertan espacios en blanco en ubicaciones arbitrarias en ambas secuencias (incluso al final) de tal manera que las secuencias resultantes X' e Y' tengan la misma longitud y además se cumpla la propiedad de que no tienen un espacio en la misma posición (es decir, si $X[i]$ es espacio, $Y[i]$ no lo es, y viceversa). Se asigna entonces una "puntuación" a cada posición. La posición j recibe un puntaje de la siguiente manera:

- $+1$ si $X'[j] = Y'[j]$,
- -1 si $X'[j] \neq Y'[j]$ y ninguno es espacio, y
- -2 si $X'[j]$ o bien $Y'[j]$ es espacio.

El puntaje para el alineamiento es la suma de los puntajes de las posiciones individuales. Por ejemplo, dadas las secuencias $X = GATCGGCAT$ y $Y = CAATGTGAATC$, un alineamiento es

$$\begin{array}{cccccccccccc} G & & A & T & C & G & & G & C & A & T & \\ C & A & A & T & & G & T & G & A & A & T & C \\ - & * & + & + & * & + & * & + & - & + & + & * \end{array}$$

Un $+$ bajo la posición indica un puntaje $+1$, un $-$ indica -1 y $*$ indica -2, entonces este alineamiento tiene un puntaje total $6 \times 1 - 2 \times 1 - 4 \times 2 = -4$.

(2) Encuentre un algoritmo para encontrar el alineamiento de mínimo costo entre dos secuencias de ADN. El algoritmo es una adaptación directa del algoritmo de distancia mínima de edición usando un subconjunto de las operaciones usadas en aquel algoritmo.

AYED II – PRÁCTICA 8

Algoritmos sobre grafos

EJERCICIO 8.1. Utilice el método *backtracking* para encontrar *todas* las soluciones para el Ejercicio 7.6 en el caso $K = 6$,

$$X = [1, 4, 7, 2, 5, 1, 6].$$

Obtenga luego un programa que efectúe este trabajo.

EJERCICIO 8.2. Resuelva el Ejercicio 7.6 utilizando backtracking.

EJERCICIO 8.3. Se dispone de un tablero $n \times n$ cuyos casilleros están pintados de color blanco o negro (sin ningún criterio definido). Se debe dar todas las maneras (si hay) de llegar del casillero $(1, 1)$ al casillero (n, n) pisando sólo casilleros blancos. Desde un casillero (i, j) es posible moverse sólo en las direcciones N, S, E, O. No se permiten movimientos en diagonal. Utilice el método de backtracking para resolver el problema.

EJERCICIO 8.4. Se recibe un arreglo $X[0, n)$ y se deben listar todas las subsecuencias crecientes. Una *subsecuencia creciente* es una subsecuencia de la forma $X[i_1] \leq ... \leq X[i_k]$, donde $0 \leq i_1 < ... < i_k < n$. Utilice backtracking para obtener la solución.

EJERCICIO 8.5.
(1) Desarrolle una algoritmo recursivo que determine si un elemento x está un árbol binario T. No se asume ninguna propiedad de orden para el árbol.
(2) Elimine la recursión del algoritmo anterior.

EJERCICIO 8.6.
(1) Desarrolle una algoritmo recursivo que devuelva la lista de hojas de un árbol binario T. Elimine luego la recursión de este algoritmo.
(2) Lo mismo que en i pero ahora se deben listar las hojas con su profundidad. Por ejemplo para el árbol

$$\langle\ \langle\langle\ \rangle, a, \langle\ \rangle\rangle\ , b, \langle\ \rangle\rangle\ ,\ c\ ,\ \langle\ \langle\ \rangle, d, \langle\ \rangle\rangle\ \rangle$$

se debe devolver $[(a, 2), (d, 1)]$.

EJERCICIO 8.7. Considere el grafo $G = (N, A)$ con vértices $N = [1, 8]$ y aristas $\{1, 2\}$, $\{1, 3\}$, $\{1, 5\}$, $\{3, 4\}$, $\{5, 3\}$, $\{5, 4\}$, $\{6, 1\}$, $\{7, 6\}$, $\{7, 8\}$, $\{6, 7\}$. Describa como se efectúan las llamadas recursivas al procedimiento *dfs* para este caso, y dé el orden resultante. Luego de paso por paso como evoluciona la pila de la versión iterativa.

EJERCICIO 8.8. Repita el Ejercicio 8.7 para *bfs*, cambiando pila por cola.

EJERCICIO 8.9. Repita los dos problemas anteriores para el grafo dirigido que se obtiene reemplazando $\{i, j\}$ por (i, j) en Ejercicio 8.7.

EJERCICIO 8.10. Sea $G = (V, A)$ un grafo dirigido. Una *caminata Euleriana* es una caminata que pasa por todas las aristas exactamente una vez. Utilice backtracking para encontrar todas las caminatas Eulerianas (si existen) que parten de un nodo v.

EJERCICIO 8.11. Se pide utilizar backtracking para resolver el problema de la moneda.

(1) a modo de ejemplo, dibujar el árbol implícito de búsqueda para monedas de denominaciones 3, 5 y 7 y monto a pagar 19.
(2) escribir un algoritmo que resuelva el problema de la moneda en general utilizando la técnica de backtracking.

EJERCICIO 8.12. Se pide utilizar backtracking para resolver el problema de la mochila. Se tienen n objetos con pesos $p[1], \ldots, p[n]$ y valores $v[1], \ldots, v[n]$; cada uno de ellos puede utilizarse una sola vez.

(1) Para una mochila con peso límite P.
(2) Para dos mochilas con peso límite P_1 y P_2 respectivamente. Se debe maximizar el valor total de los objetos cargados en las dos mochilas.

EJERCICIO 8.13. Dado un tablero de n por n, cuyas casillas pueden ser de color blanco o negro, se debe llegar del casillero $(1, 1)$ al (n, n) (ambos blancos) pisando solamente casilleros blancos. Los movimientos posibles son *left*, *right*, *up*, *down* que consiste en avanzar un casillero en la dirección indicada. Se pide desarrollar un algoritmo que liste todos los caminos posibles utilizando la técnica de backtracking. El coloreado de los casilleros esta dado por una matriz $C[1..n, 1..n]$.

EJERCICIO 8.14. Dado un grafo y m colores, dar un algoritmo que utiliza backtracking para encontrar el número de maneras diferentes de colorear los nodos del grafo de modo de que todo par de nodos adyacentes reciban colores diferentes.

8.1. Ejercicios adicionales

EJERCICIO 8.15. Dada una grilla de $N \times M$ celdas, se pide encontrar un algoritmo que utilice backtracking para encontrar todos los caminos válidos de la celda $(0, 0)$ a la celda $(N - 1, M - 1)$. Un camino es válido si está compuesto exclusivamente de movimientos válidos. Los movimientos válidos están determinados por un arreglo de listas de pares V tal que $(n, m) \in V[i, j]$ sii el movimiento de (i, j) a (n, m) es válido.

EJERCICIO 8.16.
(1) Sea $G = (V, A)$ un grafo dirigido de n vértices y m aristas. Desarrolle un algoritmo iterativo de orden $\mathcal{O}(m)$ que devuelva un array en el que se muestre la valencia entrante de cada vértice. El grafo viene representado mediante un array que en el lugar k posee la lista de vértices i tales que existe una arista de la forma (k, i).
(2) Sea F una función entera, $F : [0, n) \to [0, n)$. Desarrolle un algoritmo iterativo de orden $\mathcal{O}(n)$ que devuelva un array en el que se muestre la *incidencia* de cada vértice. La *incidencia* de i es la cantidad de elementos $k \in [0, n)$ tales que $F(k) = i$. Note las similitudes de este problema con el del inciso (1).

EJERCICIO 8.17.
(1) Pruebe que un grafo dirigido acíclico tiene siempre un vértice con valencia entrante 0.
(2) Sea $G = (V, A)$ un grafo dirigido acíclico, con $|V| = n$. Un *orden topológico para* G es un orden para sus vértices tal que si existe un camino desde v hasta w, entonces v aparece antes que w en ese orden. Pruebe por inducción constructiva que bajo las hipótesis dadas siempre existe un orden topológico. (El argumento inductivo debe arrojar el orden.) Ayuda: si v tiene valencia entrante 0, entonces claramente puede ocupar el primer lugar en el orden.
(3) Implemente de manera eficiente el algoritmo dado en (2). Para esto deberá en cada paso disponerse de un vértice con valencia entrante 0. Para esto se llevará:

(a) Un array que dé para cada vértice su valencia entrante (Ejercicio 8.16 (1)). Cuando se selecciona un vértice de valencia entrante 0, se le "quita" del grafo y habrá que actualizar el arreglo.
(b) Una pila o cola que mantenga los vértices con valencia entrante 0 del grafo actual.

(Referencia recomendada: Mamber, 7.4)

EJERCICIO 8.18. Dado una función entera $F : [0, n) \to [0, n)$, se debe encontrar un subconjunto D contenido en $[0, n)$ con la mayor cantidad de elementos posibles, tal que F restringida a D sea 1-a-1, con su imagen en D. La función está dada por un arreglo definido en el intervalo $[0, n)$.

(1) Utilice un argumento inductivo constructivo para resolver el problema. Ayuda: Si D está contenido en $F(D)$, entonces D es la solución (¿por qué?). Por otro lado, si k pertenece a D y k no está en $F(D)$, entonces k no pertenecerá a la solución, de manera que puede ser extraído, disminuyendo así el tamaño de D.
(2) Implemente el algoritmo. El algoritmo debe ser $\mathcal{O}(n)$. Se deberá mantener una estructura de datos que permita acceder en cada paso a un vértice que no está en $F(D)$. Note que esta situación es similar a la de mantener los "vértices con valencia entrante 0", del Ejercicio 8.17. Para esto el algoritmo desarrollado en el Ejercicio 8.16 (2) será útil

(Referencia recomendada: Mamber, 5.4)

EJERCICIO 8.19. Considere el algoritmo recursivo:

proc *fs*(v)
mark$[v] :=$ *true*
for w *adyacente a* v **do**
  if $mark[v] \neq$ *true* **then** *dfs*(w)
write (v)

Pruebe que si G es un grafo dirigido entonces el output es el reverso de un orden topológico para G.

(Referencia recomendada: Brassart y Bratley, 9.4.1)

EJERCICIO 8.20. ¿Qué sucede en cada uno de los algoritmos que devuelven un orden topológico (Ejercicios 8.17 y 8.19) si el grafo no es acíclico?

AYED II – PRÁCTICA 9

Otros

EJERCICIO 9.1. Sean $A[1, n]$ y $B[1, m]$ dos cadenas de caracteres con $n \leq m$.

(1) Obtener un algoritmo que encuentra la posición en que comienza la primer ocurrencia de A en B (si existe).
(2) Obtener un algoritmo que cuenta el número de ocurrencias de A en B.

EJERCICIO 9.2. Se recibe un arreglo de números naturales $a[0, N)$ y se pide escribir un algoritmo iterativo que devuelva naturales distintos $i, j \in [0, N)$ tales que $a[i] = a[j] = 0$ y la suma del intervalo $a[i, j)$ sea mínima. Se puede asumir que a tiene al menos 2 ocurrencias del número 0.

EJERCICIO 9.3. Obtenga algoritmos iterativos para los recorridos *preorder* e *inorder* de un árbol binario.

EJERCICIO 9.4. Considere la versión iterativa de quicksort. Para el caso $X = [4, 1, 8, 2, 4, 9]$ dé paso por paso la evolución de la pila y la permutación de X.

Soluciones

EJERCICIO 2.12. Resolver la siguiente recurrencia

$$t_n = \begin{cases} 5 & n = 1 \\ 3t_{n-1} - 2^{n-1} & n > 1 \end{cases}$$

SOLUCIÓN EJERCICIO 2.12. $t_n = 3t_{n-1} - 2^{n-1}$ es equivalente a $t_n - 3t_{n-1} = -2^{n-1}$. Luego la recurrencia es de la forma $t_n - 3t_{n-1} = b^n g(n)$ con $b = 2$, $g(n) = -1/2$ y $d = 0$, donde d indica el grado de g. Por lo tanto el polinomio característico asociado a la recurrencia es $(x-3)(x-2)^1$. Este polinomio tiene raíces 3 y 2, luego

$$t_n = c_1 3^n + c_2 2^n.$$

Por otro lado $t_1 = 5$ y $t_2 = 3t_1 - 2^1 = 3.5 - 2 = 13$. Por lo tanto

(1) $$5 = 2c_1 + 3c_2$$

(2) $$13 = 4c_1 + 9c_2.$$

A la ecuación (1) se la multiplica por 2 y se obtiene $10 = 4c_1 + 6c_2$ y restando esto a la ecuación (2) se obtiene $3 = 3c_2$. Por lo tanto $c_2 = 1$ y como $c_1 = (5 - 3c_2)/2$ obtenemos que $c_1 = 1$.
Es decir que $t_n = 3^n + 2^n$. Como comprobación final veamos que esta última fórmula cumple con la recursión:

$$\begin{aligned} t_1 &= 5 = 3^1 + 2^1 \\ t_{n+1} &= 3t_n - 2^n \\ &= 3(3^n + 2^n) - 2^n \\ &= 3^{n+1} + 3.2^n - 2^n \\ &= 3^{n+1} + 2.2^n \\ &= 3^{n+1} + 2^{n+1}. \end{aligned}$$

□

EJERCICIO 2.15. Calcular el orden exacto del tiempo de ejecución de cada uno de los siguientes algoritmos:

(1)
```
t := 0;
for i := 1 to n do
    for j := 1 to i do
        for k := j to j + 3 do t := t + 1
```

(2)
```
proc p(n : nat)
    if n ≥ 2 then
        for i := 1 to 16 do p(n div 4)
```

for $i := 1$ **to** n **do write** i
for $i := 1$ **to** n **do**
    for $j := 1$ **to** n **do write** j

SOLUCIÓN EJERCICIO 2.15.
(1) La única operación representativa es $t := t + 1$. Por lo tanto debemos contar la cantidad de veces que se hace esta operación. Recordemos que si tenemos la estructura de repetición

for $i := 1$ **to** n **do** $S(i)$

donde $S(i)$ es un conjunto de instrucciones a ejecutarse en la i-ésima iteración, entonces el orden de de este **for** es

$$\sum_{i=1}^{n} s(i),$$

donde $s(i)$ es la cantidad de operaciones que requiere hacer $S(i)$. Aplicando este resultado tres veces obtenemos que la cantidad de operaciones necesarias para ejecutar el programa es

$$\sum_{i=1}^{n} s(i) = \sum_{i=1}^{n}(\sum_{j=1}^{i} t(i)) = \sum_{i=1}^{n}(\sum_{j=1}^{i}(\sum_{k=j}^{j+3} 1)) = \sum_{i=1}^{n}(\sum_{j=1}^{i} 4) = \sum_{i=1}^{n} 4i = \frac{4n(n+1)}{2} = 2n(n+1).$$

Como $\Theta(2n(n+1)) = \Theta(n^2)$, el orden exacto del programa es $\Theta(n^2)$.
(2) Si $t(n)$ es el tiempo de ejecución del programa (con input n), entonces se cumple la siguiente recursión:

$$t(n) = 16t(\lfloor n/4 \rfloor) + n + n^2$$

para $n > 1$. El primer término de la suma de la ecuación anterior se obtiene del primer **for** , el segundo y tercer término se obtienen del segundo y tercer **for** respectivamente. Luego, para n que es potencia de 4 el tiempo de ejecución está dado por una recurrencia del tipo "divide y vencerás" de la forma $t(n) = at(n/b) + g(n)$, con $a = 16$, $b = 4$ y $g(n) = n^2 + n \in \Theta(n^2)$ (es decir $k = 2$ en la fórmula de "divide y vencerás"). Si probamos que la función es eventualmente no decreciente, la fórmula de "divide y vencerás" implica que $t(n) \in \Theta(n^2 \log n)$. Veamos ahora que $t(n)$ es eventualmente no decreciente, (de hecho, es creciente). Lo hacemos por inducción:

$$\begin{aligned} t(1) &= 16t(\lfloor 1/4 \rfloor) + 2 + 4 = 16t(0) + 6 > t(0). \\ t(n+1) &= 16t(\lfloor \frac{n+1}{4} \rfloor) + (n+1) + (n+1)^2 \\ &\geq 16t(\lfloor \frac{n}{4} \rfloor) + (n+1) + (n+1)^2 \\ &> 16t(\lfloor \frac{n}{4} \rfloor) + n + n^2. \end{aligned}$$

En vista de lo anterior, deducimos que el orden exacto del algoritmo es $\Theta(n^2 \log n)$. □

EJERCICIO 2.16. Ordenar las siguientes funciones según $\subset$ (incluido *estricto*) e $=$ de sus 's.

(1) $n^4 + 2\log n$
(2) $\log(n^{n^4})$
(3) $2^{4\log n}$
(4) 4^n
(5) $n^3 \log n$

Justificar sin utilizar la regla del límite.

SOLUCIÓN EJERCICIO 2.16. El orden correcto es

$$(n^3 \log n) \subset (2^{4 \log n}) = (n^4 + 2 \log n) \subset \log(n^{n^4}) \subset 4^n.$$

Justifiquemos el resultado.

- $(n^3 \log n) \subset (2^{4 \log n})$. Observemos que $2^{4 \log n} = (2^{\log n})^4 = n^4$. Sabemos que $(\log n) \subset (n)$, es decir que existe $c \in \mathbb{R}^+$ tal que $\forall^\infty n \in \mathbb{N}$, $\log n < n$. Multiplicando por n^3 las expresiones obtenemos que existe $c \in \mathbb{R}^+$ tal que $\forall^\infty n \in \mathbb{N}$, $n^3 \log n < n^4$, lo cual implica el resultado.
- $(2^{4 \log n}) = (n^4 + 2 \log n)$. Sabemos que $(\log n) = (2 \log n)$ y entonces $(2 \log n) = (\log n) \subset (n) \subset (n^4)$. Por lo tanto $2 \log n \in (n^4)$. Sabemos que si $f(n) \in (g(n))$, entonces $(f(n)+g(n)) = (g(n))$. Tomando $f(n) = \log n$ y $g(n) = n^4$ obtenemos que $(n^4 + 2 \log n) = (n^4) = (2^{4 \log n})$.
- $(n^4 + 2 \log n) \subset \log(n^{n^4})$. Observemos que $\log(n^{n^4}) = n^4 \log n$. Sabemos que $(1) \subset (\log n)$ y con el mismo argumento usado en el primer item podemos inferir que $(n^4) \subset (n^4 \log n)$.
- $\log(n^{n^4}) \subset 4^n$. Tenemos que $(\log(n^{n^4})) = (n^4 \log n) \subset (n^5)$. Además, sabemos que dado $x \in \mathbb{N}$ y $c > 1$, se cumple que $(n^x) \subset (c^n)$. Por lo tanto $(\log(n^{n^4})) \subset (n^5) \subset (4^n)$. □

EJERCICIO 2.18. Dar algoritmos cuyos tiempos de ejecución tengan los siguientes órdenes:

(1) $n^2 + 2 \log n$
(2) $n^2 \log n$
(3) 3^n

No utilizar potencia, logaritmo ni multiplicación en los programas.

SOLUCIÓN EJERCICIO 2.18.
(1) Observar que como $\log n \in (n^2)$, tenemos que $\Theta(n^2 + 2 \log n) = \Theta(n^2)$. Por lo tanto, el ejercicio es equivalente a encontrar un algoritmo con orden exacto n^2:

```
proc p(n : nat)
    for i := 1 to n do
        for j := 1 to n do write j
```

(2)

```
proc q(n : nat)
    if n > 1 then
        for i := 1 to n do
            for j := 1 to n do write j
        for k := 1 to 4 do
            q(n div 2)
```

Esta es una recurrencia del tipo “divide y vencerás” con la fórmula

$$t(n) = 4t(\frac{n}{2}) + n^2.$$

Es decir $t(n) = at(\frac{n}{b}) + n^k$ con $a = 2$, $b = 4$ y $k = 2$. Se puede ver fácilmente que $t(n)$ es no decreciente y como $a = b^k$, por la fórmula de “divide y vencerás” obtenemos que el tiempo de ejecución del algoritmo es $n^2 \log n$.
(3)

```
proc ep(n : nat)
    if n > 0 then
```

for $i := 1$ **to** 3 **do** $ep(n-1)$
else
write (n)

Para $n = 0$, $t(n) = 1$. Para $n > 0$, $t(n) = 3t(n-1)$. Usando inducción podemos probar que $t(n) = 3^n$: tenemos que $t(0) = 1 = 3^0$ y $t(n+1) = 3t(n) \stackrel{HI}{=} 3.3^n = 3^{n+1}$. □

EJERCICIO 3.6. Los algoritmos de ordenamiento de secuencias de palabras deben efectuar frecuentemente la operación "swap", y la pregunta ¿es la palabra del lugar i menor o igual a la palabra del lugar j en el orden lexicográfico? Implemente estas dos operaciones abstractas sobre la siguiente representación de una secuencias de k palabras. Las palabras están dispuestas de manera consecutiva en un arreglo de caracteres $C[1..M]$. Se dispone de otro arreglo $A[1..N]$ de pares de naturales que contiene en el lugar i (con $1 \leq i \leq k$) al par que indica el lugar de comienzo de la palabra i-ésima en C, y su tamaño. Por ejemplo, si $k = 4$, en los arreglos
$C = arbolcasaelefanteaseo...$
$A = (6,4)\ (1,5)\ (12,6)\ (10,2)\ ...$
se representa la secuencia $casa,\ arbol,\ efante,\ el.$

SOLUCIÓN EJERCICIO 3.6. En esta implementación de diccionario estamos usando dos arreglos $C[1..M]$ que es un arreglo de caracteres, y $A[1..N]$ que es un arreglo de pares ordenados de enteros. Las declaraciones iniciales podrían ser:

var C : **array**$[1..M]$ **of char**
var A : **array**$[1..N]$ **of Par**

Donde **Par** indica un par de enteros. Si x es de tipo **Par** , entonces $x.pri$ es el primer entero y $x.seg$ es el segundo entero.
Si $C = arbolcasaelefanteaseo...$ y $A = (6,4)\ (1,5)\ (12,6)\ (10,2)\ ...$, entonces, por ejemplo, $C[6] = c$, $A[3].pri = 12$ y $A[3].seg = 6$. La *palabra i-ésima* es la palabra que comienza en $A[i].pri$ y tiene longitud $A[i].seg$, es decir la palabra

$$C[\,A[i].pri\,]\ C[\,A[i].pri + 1\,]\ \ldots\ C[\,A[i].pri + A[i]seg - 1\,]$$

Con estas convenciones implementaremos $swap(i,j)$ y $menor(i,j)$. El primer procedimiento intercambia la palabra i por la j. El segundo procedimiento (en realidad una función) devuelve *true* si la palabra i-ésima es menor que la palabra j-ésima en el orden lexicográfico. En caso contrario devuelve *false* .
El procedimiento $swap$ es bastante simple, sólo debemos permutar $A[i]$ con $A[j]$:

proc $swap(i$: **nat**, j : **nat**$)$
var x : **Par**
$x := A[j]$
$A[j] := A[i]$
$A[i] := x$

Para el procedimiento $menor$ debemos usar la siguiente propiedad: si el caracter x es anterior al caracter y en el orden alfabético, entonces $x < y$ es *true* , de lo contrario es *false* . Muchos lenguajes de programación implementan esta propiedad (C, Pascal, etc.). Recordemos que el orden lexicográfico se define de la siguiente manera: si a y b dos arreglos de caracteres, entonces $a < b$ si y sólo si existe k tal que $a[1] = b[1]$, $a[2] = b[2]$, ..., $a[k-1] = b[k-1]$ y o bien $a[k] < b[k]$, o bien $a.size < k$ y $b.size \geq k$. Implementemos

primero el orden lexicográfico definido así. No es una función que usaremos luego, pero nos servirá de guía para hacer la función $menor$.

func $ordLex(a, b : \textbf{array of char})$ **dev: bool**
  var k : **int**
  $k := 1$
  while $(a[k] = b[k] \wedge k \leq a.size \wedge k \leq b.size)$ **do** $k := k + 1$
  return $(a.size \geq k \wedge b.size \geq k \wedge a[k] < b[k]) \vee (a.size < k \wedge b.size \geq k)$

En base a la función anterior es claro como debemos hacer menor

func $menor(i : \textbf{nat}, j : \textbf{nat})$ **dev: bool**
  var k : **int**
  $k := 1$
  while $(C[A[i].pri + k - 1] = C[A[j].pri + k - 1] \wedge k \leq A[i].seg] \wedge k \leq A[j].seg)$ **do**
    $k := k + 1$
  return $(A[i].seg \geq k \wedge A[j].seg \geq k \wedge C[A[i].pri + k - 1] < C[A[j].pri + k - 1]) \vee$
    $\vee (A[i].seg < k \wedge A[j].seg \geq k)$

□