Algoritmos y Estructuras de Datos

Clase 01 — Introducción a la materia. Complejidad, contratos e invariantes

Segundo trimestre de 2025
Maestría en Inteligencia Artificial
Universidad de San Andrés

1. Teoría

Un algoritmo es una descripción ejecutable y finita para resolver un problema; transforma datos de entrada en datos de salida. Una estructura de datos organiza la información y determina qué operaciones admite con qué costos. Modelo RAM: suponemos memoria indexable y que cada operación elemental (acceso/actualización de una celda, asignación, aritmética, comparación, invocación de función y control) cuesta una unidad de tiempo. Así definimos TRAM(x): el número de unidades consumidas por el algoritmo sobre la entrada x. Medimos casos: mejor, peor y promedio según el tamaño n=|x|, y analizamos su crecimiento asintótico mediante O, Ω y Θ. Big-O es una cota superior asintótica: g(n)∈ O(f(n)) si existe c>0 y n■ tal que g(n)≤c·f(n) para todo n≥n■. Contratos: (Pre, Post). Un algoritmo es correcto si, con entradas que cumplen Pre, la salida cumple Post. Invariantes: propiedades que se preservan durante un ciclo o a lo largo de las operaciones de una estructura.

Ejemplo teórico ilustrativo: invertir una lista en O(n) tiempo y O(1) espacio.

```
def invertir(lista):
    """Invierte una lista in-place.
    Pre: lista es indexable. Post: lista queda invertida.
    Complejidad: O(n) en tiempo, O(1) en espacio.
    """
    n = len(lista)
    for i in range(n // 2):
        j = n - 1 - i
        lista[i], lista[j] = lista[j], lista[i]
    return lista

print(invertir([1,2,3,4,5]))  # [5,4,3,2,1]
```

2. Ejercicios Resueltos en Python

1) Búsqueda lineal con contrato (devuelve índice o -1).

```
def buscar_lineal(a, x):
    """Pre: a es una lista. Post: devuelve el índice de x o -1 si no está."""
    for i in range(len(a)):
        if a[i] == x:
            return i
    return -1

print(buscar_lineal([9,3,7,3], 7))  # 2
print(buscar_lineal([9,3,7,3], 5))  # -1
```

2) Máximo y mínimo con sus índices (una sola pasada).

```
def extremos(a):
    """Devuelve (max, i_max, min, i_min)."""
    if len(a) == 0:
        raise ValueError("lista vacía")
    max_v = min_v = a[0]
    i_max = i_min = 0
    for i in range(1, len(a)):
        v = a[i]
        if v > max_v:
            max_v, i_max = v, i
        if v < min_v:
            min_v, i_min = v, i
    return max_v, i_max, min_v, i_min

print(extremos([3,7,2,9,1]))  # (9,3,1,4)
```

3) Contar ocurrencias de un valor.

```
def contar(a, x):
    c = 0
    for v in a:
        if v == x:
            c += 1
    return c

print(contar([1,2,2,3,2,4], 2))  # 3
```

4) Eliminar duplicados preservando orden (sin usar set).

```
def sin_duplicados(a):
    vistos = {}
    res = []
    for v in a:
        if v not in vistos:
            vistos[v] = True
            res.append(v)
    return res

print(sin_duplicados([3,1,3,2,1,2,4]))  # [3,1,2,4]
```

5) Rotar lista k posiciones a la derecha in-place.

```
def rotar(a, k):
    n = len(a)
    k %= n if n else 0
    def inv(i, j):
        while i < j:
```

```
            a[i], a[j] = a[j], a[i]
            i += 1; j -= 1
    inv(0, n-1); inv(0, k-1); inv(k, n-1)
    return a

print(rotar([1,2,3,4,5,6], 2))  # [5,6,1,2,3,4]
```

6) Suma acumulada (prefix sums).

```
def prefijos(a):
    acc = 0
    res = []
    for v in a:
        acc += v
        res.append(acc)
    return res

print(prefijos([1,2,3,4]))  # [1,3,6,10]
```

7) Producto escalar de dos listas (verifica tamaños).

```
def producto_escalar(a, b):
    if len(a) != len(b):
        raise ValueError("longitudes distintas")
    s = 0
    for i in range(len(a)):
        s += a[i] * b[i]
    return s

print(producto_escalar([1,2,3],[4,5,6]))  # 32
```

8) Mezclar dos listas ordenadas (sin usar sorted).

```
def mezclar(a, b):
    i = j = 0
    c = []
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            c.append(a[i]); i += 1
        else:
            c.append(b[j]); j += 1
    while i < len(a):
        c.append(a[i]); i += 1
    while j < len(b):
        c.append(b[j]); j += 1
    return c

print(mezclar([1,3,7],[2,3,5,8]))  # [1,2,3,3,5,7,8]
```

9) Selection sort con contadores de comparaciones e intercambios.

```
def selection_sort_contado(a):
    a = a[:]  # no modificar entrada
    comps = swaps = 0
    n = len(a)
    for i in range(n):
        min_j = i
        for j in range(i+1, n):
            comps += 1
            if a[j] < a[min_j]:
                min_j = j
        if min_j != i:
            a[i], a[min_j] = a[min_j], a[i]
```

```
            swaps += 1
    return a, comps, swaps

print(selection_sort_contado([3,1,4,1,5]))
```

10) Búsqueda binaria (índice del primer elemento > x).

```
def binsearch_primero_mayor(a, x):
    i, j = 0, len(a)
    while i + 1 < j:
        m = (i + j) // 2
        if a[m] <= x:
            i = m
        else:
            j = m
    if len(a)==0 or a[0] > x:
        return 0
    return j

print(binsearch_primero_mayor([1,2,2,4,7], 2))  # 3
```

11) Verificar invariante de selection sort en posición i.

```
def invariante_selection(a, i):
    """Devuelve True si el mínimo de a[i:n] está en a[i]."""
    if i >= len(a):
        return True
    min_val = a[i]
    for j in range(i, len(a)):
        if a[j] < min_val:
            return False
    return True

print(invariante_selection([1,2,3,4], 2))  # True
print(invariante_selection([3,1,2], 0))    # False
```

12) Validación de contrato: último elemento de una lista no vacía.

```
def ultimo(a):
    """Pre: a no es vacía. Post: devuelve el último elemento."""
    if len(a) == 0:
        raise AssertionError("Precondición violada: lista vacía")
    return a[len(a)-1]

print(ultimo([5,6,7]))  # 7
```

13) Contar inversiones (pares ia[j]) en O(n^2).

```
def inversiones(a):
    c = 0
    for i in range(len(a)):
        for j in range(i+1, len(a)):
            if a[i] > a[j]:
                c += 1
    return c

print(inversiones([3,1,2]))  # 2
```

14) Transponer matriz cuadrada in-place.

```
def transponer(m):
    n = len(m)
    for i in range(n):
        for j in range(i+1, n):
```

```
            m[i][j], m[j][i] = m[j][i], m[i][j]
    return m

print(transponer([[1,2,3],[4,5,6],[7,8,9]]))
```

15) Sumar todos los elementos de una matriz (una pasada).

```
def suma_matriz(m):
    s = 0
    for fila in m:
        for v in fila:
            s += v
    return s

print(suma_matriz([[1,2],[3,4],[5,6]]))  # 21
```

16) Simulación de segundero con invariante (0≤seg<60).

```
class Segundero:
    def __init__(self):
        self._n = 0  # 0≤_n<60 siempre
    def avanzar(self):
        self._n += 1
        if self._n == 60:
            self._n = 0
    def segundo(self):
        return self._n

s = Segundero()
for _ in range(61):
    s.avanzar()
assert 0 <= s.segundo() < 60
print(s.segundo())
```

17) Estimar operaciones de invertir(n) por conteo analítico.

```
def estimar_ops_invertir(n):
    """Devuelve número de asignaciones de intercambio (aprox)."""
    # Cada iteración hace 3 asignaciones en el swap; hay n//2 iteraciones
    return 3 * (n // 2)

for n in [1,2,3,10]:
    print(n, estimar_ops_invertir(n))
```

18) Copiar lista manualmente (O(n)).

```
def copiar(a):
    b = [None] * len(a)
    for i in range(len(a)):
        b[i] = a[i]
    return b

print(copiar([9,8,7]))
```

19) Verificar que una función cumple (Pre,Post) en pruebas simples.

```
def aparece(a, x):
    # Pre: a es lista; Post: True si x está, False si no
    for v in a:
        if v == x:
            return True
    return False

# Pruebas de contrato (básicas)
```

```
assert aparece([1,2,3], 2) is True
assert aparece([1,2,3], 4) is False
print("ok")
```

20) Contar comparaciones de búsqueda lineal (peor caso).

```
def comps_busqueda_lineal(a, x):
    c = 0
    for i in range(len(a)):
        c += 1
        if a[i] == x:
            return c
    return c  # x no está: n comparaciones

print(comps_busqueda_lineal([1,2,3,4], 5))  # 4
```

3. Cuestionario Multiple Choice (Nivel medio)

Pregunta 1. ¿Cuál es la definición más precisa de algoritmo usada en la materia?
A. Programa que corre rápido
B. Cualquier receta informal
C. Descripción ejecutable que termina para toda entrada
D. Lista de datos

Pregunta 2. En el modelo RAM, ¿cuál de estas operaciones NO cuesta O(1)?
A. Leer una celda
B. Asignar a una variable
C. Comparar dos enteros
D. Crear un arreglo de n celdas

Pregunta 3. La función TRAM(x) mide…
A. Uso de memoria en bytes
B. Cantidad de líneas de código
C. El tamaño de la entrada
D. Unidades de tiempo consumidas por el algoritmo sobre x

Pregunta 4. ¿Qué representa Tpeor(n)?
A. Mínimo TRAM(x) para |x|=n
B. Varianza de TRAM(x)
C. Máximo TRAM(x) para |x|=n
D. Promedio de TRAM(x) para |x|=n

Pregunta 5. $g(n) \in O(f(n))$ significa:
A. Existe c,n■ con $g(n) \leq c \cdot f(n)$ para $n \geq n$■
B. g(n) es menor que f(n) para todo n
C. $g(n) \geq f(n)$ siempre
D. g(n) crece exactamente igual a f(n)

Pregunta 6. Cuál es la relación correcta entre clases de complejidad:
A. O(log n) ■ O(n) ■ O(n log n)
B. O(n) ■ O(log n) ■ O(n log n)
C. O(n log n) ■ O(n)
D. O(1) ■ $O(n^2)$ ■ O(n log n)

Pregunta 7. Complejidad temporal en peor caso de selection sort:
A. O(log n)
B. O(n log n)
C. O(n)
D. $O(n^2)$

Pregunta 8. En un contrato (Pre, Post), la corrección exige que…
A. Pre y Post sean equivalentes
B. Pre sea más débil que Post
C. Si Pre vale, entonces la salida satisface Post
D. Siempre se cumple Post

Pregunta 9. Un invariante de ciclo es…
A. Condición de corte del ciclo
B. Una aserción que se cumple sólo al inicio
C. Una precondición
D. Propiedad preservada por cada iteración

Pregunta 10. En un ABB balanceado con n nodos, la búsqueda cuesta…
A. O(n log n)
B. O(n)
C. O(log n)
D. O(1)

Pregunta 11. Si $f(n)=n^2+3n$ y $g(n)=n^2$, entonces $f(n) \in$ …
A. o(g(n))
B. $\Theta(g(n))$
C. O(1)
D. $\omega(g(n))$

Pregunta 12. ¿Qué parte del costo de Mergesort domina su T(n)?
A. El caso base
B. El último merge únicamente
C. La comparación inicial
D. La suma de costos en los niveles: 2^k·(n/2^k)

Pregunta 13. ¿Qué afirma la propiedad O(f1+f2) = O(max{f1,f2})?
A. Ambas funciones son lineales
B. La función dominante rige la cota
C. No aplica a polinomios
D. Siempre crecen igual

Pregunta 14. ¿Qué es Tpromedio(n)?
A. Mínimo TRAM(x)
B. Máximo TRAM(x)
C. TRAM para la entrada peor
D. Media de TRAM(x) para |x|=n

Pregunta 15. ¿Cuál es un ejemplo de invariante de estructura?
A. En un ABB: valores izq < nodo < der
B. Que la lista esté ordenada al comenzar
C. Que el while finalice
D. Que Pre $\Rightarrow$ Post

Pregunta 16. ¿Qué sucede si se viola la Pre de un contrato?
A. Se cumple Post más fuerte
B. La complejidad mejora
C. El algoritmo sigue siendo correcto
D. No puede garantizarse Post

Pregunta 17. ¿Qué costo espacial tiene invertir una lista in-place?
A. O(log n)
B. O(1)
C. O(n)
D. $O(n^2)$

Pregunta 18. ¿Qué operación tiene costo O(n) en modelo RAM?
A. Crear arreglo de n elementos
B. Leer variable
C. Comparar booleanos
D. Sumar enteros

Pregunta 19. Si $g \in O(f)$ y $f \in O(g)$, entonces…
A. $g \in \Theta(f)$
B. No se puede concluir
C. g es menor que f
D. $g \in \Omega(f)$ pero no Θ

Pregunta 20. ¿Cuál es el propósito del caso base en recursión?
A. Reducir memoria
B. Evitar parámetros
C. Mejorar $\Omega(n)$
D. Garantizar terminación

Respuestas Correctas

Pregunta 1: C
Pregunta 2: D
Pregunta 3: D
Pregunta 4: C
Pregunta 5: A
Pregunta 6: A
Pregunta 7: D
Pregunta 8: C
Pregunta 9: D
Pregunta 10: C
Pregunta 11: B
Pregunta 12: D
Pregunta 13: B
Pregunta 14: D
Pregunta 15: A
Pregunta 16: D
Pregunta 17: B
Pregunta 18: A
Pregunta 19: A
Pregunta 20: D

Algoritmos y Estructuras de Datos

Clase 02 — Tipos de datos secuenciales y Divide & Conquer

Segundo trimestre de 2025
Maestría en Inteligencia Artificial
Universidad de San Andrés

1. Teoría

En esta clase abordamos dos temas centrales: los tipos de datos secuenciales y la técnica de Divide & Conquer. Los tipos de datos secuenciales incluyen vectores, pilas y colas. - Los vectores estáticos tienen tamaño fijo, mientras que los dinámicos pueden crecer usando estrategias de realocación y copiado. Su análisis muestra que una estrategia de crecimiento geométrico permite operaciones de extensión en O(1) amortizado. - Las pilas (LIFO) permiten operaciones push y pop en O(1). - Las colas (FIFO) permiten agregar al final y quitar del principio en O(1). La técnica de Divide & Conquer divide el problema en subproblemas más pequeños, resuelve cada uno recursivamente y luego combina las soluciones. Ejemplos clásicos: mergesort y búsqueda binaria.

Ejemplo teórico ilustrativo: Mergesort

```
def mergesort(a):
    if len(a) <= 1:
        return a
    m = len(a) // 2
    izq = mergesort(a[:m])
    der = mergesort(a[m:])
    return merge(izq, der)

def merge(a, b):
    i = j = 0
    res = []
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            res.append(a[i]); i += 1
        else:
            res.append(b[j]); j += 1
    res.extend(a[i:]); res.extend(b[j:])
    return res

print(mergesort([3,7,9,0,1,4,8,5]))
```

2. Ejercicios Resueltos en Python

1) Implementar un vector fijo.

```
class Vector:
    def __init__(self, n):
        self._datos = [None]*n
    def size(self): return len(self._datos)
    def get(self,i): return self._datos[i]
    def upd(self,i,x): self._datos[i]=x

v = Vector(3)
v.upd(0,10); print(v.get(0))
```

2) Vector dinámico con crecimiento geométrico.

```
class VectorDinamico:
    def __init__(self):
        self._cap = 1
        self._n = 0
        self._datos = [None]*self._cap
    def append(self,x):
        if self._n==self._cap:
            self._cap*=2
            nuevo=[None]*self._cap
            for i in range(self._n): nuevo[i]=self._datos[i]
            self._datos=nuevo
        self._datos[self._n]=x; self._n+=1
    def get(self,i): return self._datos[i]

v=VectorDinamico()
for i in range(5): v.append(i)
print([v.get(i) for i in range(5)])
```

3) Implementar pila con lista enlazada.

```
class Stack:
    def __init__(self): self._first=None
    def push(self,x): self._first=[x,self._first]
    def pop(self): x=self._first[0]; self._first=self._first[1]; return x
    def empty(self): return self._first is None

s=Stack(); s.push(1); s.push(2); print(s.pop())
```

4) Implementar cola con lista enlazada.

```
class Queue:
    def __init__(self): self._first=self._last=None
    def add(self,x):
        nodo=[x,None]
        if self._last: self._last[1]=nodo
        else: self._first=nodo
        self._last=nodo
    def remove(self):
        x=self._first[0]; self._first=self._first[1]
        if not self._first: self._last=None
        return x

q=Queue(); q.add(1); q.add(2); print(q.remove())
```

5) Simular pila con dos colas.

```
from collections import deque
```

```
class PilaConColas:
    def __init__(self): self.q1,self.q2=deque(),deque()
    def push(self,x): self.q1.append(x)
    def pop(self):
        while len(self.q1)>1: self.q2.append(self.q1.popleft())
        val=self.q1.popleft()
        self.q1,self.q2=self.q2,self.q1
        return val

p=PilaConColas(); p.push(1); p.push(2); print(p.pop())
```

6) Simular cola con dos pilas.

```
class ColaConPilas:
    def __init__(self): self.s1,self.s2=[],[]
    def add(self,x): self.s1.append(x)
    def remove(self):
        if not self.s2:
            while self.s1: self.s2.append(self.s1.pop())
        return self.s2.pop()

c=ColaConPilas(); c.add(1); c.add(2); print(c.remove())
```

7) Evaluar expresión postfija con pila.

```
def evaluar_postfija(expr):
    s=[]
    for t in expr.split():
        if t.isdigit(): s.append(int(t))
        else:
            b,a=s.pop(),s.pop()
            if t=='+': s.append(a+b)
            if t=='*': s.append(a*b)
    return s[0]

print(evaluar_postfija("2 3 4 * +"))
```

8) Balanceo de paréntesis con pila.

```
def balanceado(expr):
    s=[]
    for c in expr:
        if c=='(': s.append(c)
        elif c==')':
            if not s: return False
            s.pop()
    return not s

print(balanceado("(())"))
```

9) Buffer circular con cola.

```
class BufferCircular:
    def __init__(self,n): self.buf=[None]*n; self.i=self.j=0; self.lleno=False
    def add(self,x):
        self.buf[self.j]=x; self.j=(self.j+1)%len(self.buf)
        if self.lleno: self.i=(self.i+1)%len(self.buf)
        self.lleno=self.i==self.j
    def get(self): return self.buf[self.i]

b=BufferCircular(3); b.add(1); b.add(2); b.add(3); b.add(4); print(b.get())
```

10) Undo/redo con pilas.

```
class Editor:
    def __init__(self): self.texto=""; self.undo_stack=[]; self.redo_stack=[]
    def escribir(self,t): self.undo_stack.append(self.texto); self.texto+=t
    def undo(self):
        if self.undo_stack: self.redo_stack.append(self.texto); self.texto=self.undo_stack.pop()
    def redo(self):
        if self.redo_stack: self.undo_stack.append(self.texto); self.texto=self.redo_stack.pop()

e=Editor(); e.escribir("Hola"); e.undo(); e.redo(); print(e.texto)
```

11) Verificar palíndromo con pila y cola.

```
from collections import deque
def palindromo(s):
    s=s.lower().replace(" ","")
    cola=deque(s); pila=list(s)
    while cola:
        if cola.popleft()!=pila.pop(): return False
    return True

print(palindromo("neuquen"))
```

12) Búsqueda binaria.

```
def binsearch(a,x):
    i,j=0,len(a)
    while i+1<j:
        m=(i+j)//2
        if a[m]<=x: i=m
        else: j=m
    return j

print(binsearch([1,2,2,4,7],2))
```

13) Raíz cuadrada entera con binaria.

```
def raiz_entera(n):
    i,j=0,n+1
    while i+1<j:
        m=(i+j)//2
        if m*m<=n: i=m
        else: j=m
    return i

print(raiz_entera(17))
```

14) Mergesort.

```
def mergesort(a):
    if len(a)<=1: return a
    m=len(a)//2
    return merge(mergesort(a[:m]),mergesort(a[m:]))

def merge(a,b):
    i=j=0; c=[]
    while i<len(a) and j<len(b):
        if a[i]<=b[j]: c.append(a[i]); i+=1
        else: c.append(b[j]); j+=1
    c.extend(a[i:]); c.extend(b[j:])
    return c

print(mergesort([3,7,9,0,1,4,8,5]))
```

15) Subarreglo máximo O(n^3).

```
def max_subarray(a):
    mejor=-10**9
    for i in range(len(a)):
        for j in range(i,len(a)):
            s=sum(a[i:j+1])
            if s>mejor: mejor=s
    return mejor

print(max_subarray([2,-5,1,1,1,1,-1,3,-1]))
```

16) Subarreglo máximo O(n).

```
def kadane(a):
    mejor=act=a[0]
    for x in a[1:]:
        act=max(x,act+x)
        mejor=max(mejor,act)
    return mejor

print(kadane([2,-5,1,1,1,1,-1,3,-1]))
```

17) Invertir lista con Divide & Conquer.

```
def invertir_dc(a):
    if len(a)<=1: return a
    m=len(a)//2
    return invertir_dc(a[m:])+invertir_dc(a[:m])

print(invertir_dc([1,2,3,4,5]))
```

18) Máximo de lista con D&C.;

```
def max_dc(a):
    if len(a)==1: return a[0]
    m=len(a)//2
    return max(max_dc(a[:m]), max_dc(a[m:]))

print(max_dc([3,1,7,2,9,5]))
```

19) Multiplicación recursiva de enteros.

```
def mult(a,b):
    if b==0: return 0
    if b%2==0: return mult(a+a,b//2)
    return mult(a+a,b//2)+a

print(mult(7,6))
```

20) Suma acumulada con pila.

```
def suma_pila(a):
    s=0; pila=[]
    for x in a: pila.append(x)
    while pila: s+=pila.pop()
    return s

print(suma_pila([1,2,3,4]))
```

3. Cuestionario Multiple Choice

Pregunta 1. ¿Cuál es la complejidad amortizada de agregar a un vector dinámico?
A. O(log n)
B. O(n log n)
C. O(1)
D. O(n)

Pregunta 2. ¿Qué estructura es LIFO?
A. Vector
B. Árbol
C. Cola
D. Pila

Pregunta 3. ¿Qué estructura es FIFO?
A. Vector
B. Cola
C. Grafo
D. Pila

Pregunta 4. ¿Cuál es el invariante en búsqueda binaria?
A. x siempre en A[i]
B. $A[i] < x \leq A[j]$
C. $A[i] \leq x < A[j]$
D. $A[i] \geq x > A[j]$

Pregunta 5. Complejidad de búsqueda binaria.
A. O(log n)
B. O(n)
C. O(n log n)
D. O(1)

Pregunta 6. Complejidad de mergesort.
A. O(n log n)
B. O(n)
C. $O(n^2)$
D. O(log n)

Pregunta 7. Peor caso de insertar en vector fijo.
A. O(log n)
B. O(n log n)
C. O(1)
D. O(n)

Pregunta 8. ¿Qué devuelve pop en una pila vacía?
A. None
B. 0
C. Siempre un valor
D. Error

Pregunta 9. ¿Qué operación no pertenece a una cola?
A. Agregar al final
B. Eliminar del tope
C. Ver primero
D. Quitar primero

Pregunta 10. Divide & Conquer se basa en:
A. Ordenar, buscar, recorrer
B. Comparar, seleccionar, intercambiar
C. Dividir, resolver, combinar
D. Repetir, iterar, terminar

Pregunta 11. ¿Qué tipo de costo se analiza con O(1) amortizado?
A. Mejor caso
B. Costo espacial
C. Costo promedio por operación
D. Peor caso absoluto

Pregunta 12. En mergesort, el merge cuesta:
A. O(n)
B. $O(n^2)$
C. O(log n)
D. O(1)

Pregunta 13. ¿Qué asegura un vector dinámico con duplicación de capacidad?
A. Reduce n log n
B. Evita memoria extra
C. Hace $O(n^2)$
D. Amortización constante

Pregunta 14. En pila, push y pop cuestan:
A. O(1)
B. O(log n)
C. O(n log n)
D. O(n)

Pregunta 15. En cola, add y remove cuestan:
A. O(log n)
B. $O(n^2)$
C. O(1)
D. O(n)

Pregunta 16. ¿Qué devuelve búsqueda binaria si x no está?
A. Siempre -1
B. Posición de inserción
C. None
D. Error

Pregunta 17. Subarreglo máximo con algoritmo ingenuo cuesta:
A. O(n log n)
B. O(n)
C. $O(n^2)$
D. $O(n^3)$

Pregunta 18. Kadane resuelve subarreglo máximo en:
A. O(n log n)
B. $O(n^2)$
C. O(n)
D. O(log n)

Pregunta 19. Invertir lista con Divide & Conquer cuesta:
A. $O(n^2)$
B. O(n)
C. O(n log n)
D. O(log n)

Pregunta 20. ¿Qué parte domina en mergesort?
A. Los casos base
B. La combinación de sublistas
C. El primer split
D. La última comparación

Respuestas Correctas

Pregunta 1: C
Pregunta 2: D
Pregunta 3: B
Pregunta 4: C
Pregunta 5: A
Pregunta 6: A
Pregunta 7: D
Pregunta 8: D
Pregunta 9: B
Pregunta 10: C
Pregunta 11: C
Pregunta 12: A
Pregunta 13: D
Pregunta 14: A
Pregunta 15: C
Pregunta 16: B
Pregunta 17: D
Pregunta 18: C
Pregunta 19: C
Pregunta 20: B

Algoritmos y Estructuras de Datos

Clase 03 — Árboles binarios, búsqueda y balanceo

Segundo trimestre de 2025
Maestría en Inteligencia Artificial
Universidad de San Andrés

1. Teoría

Un árbol binario es una estructura de datos jerárquica donde cada nodo tiene como máximo dos hijos: izquierdo y derecho. Se usan para representar expresiones, implementar diccionarios, conjuntos y muchas otras aplicaciones. Un árbol binario de búsqueda (ABB) mantiene el invariante: - Para cada nodo con valor x, todos los elementos en su subárbol izquierdo son menores que x, y todos los elementos en su subárbol derecho son mayores que x. Operaciones típicas en ABB: - Búsqueda, inserción y eliminación, todas con complejidad O(h), donde h es la altura del árbol. Problema: un ABB puede degenerarse en una lista (h = n). Solución: árboles balanceados como los AVL, que garantizan $h \in O(\log n)$. Los árboles AVL mantienen la propiedad de que el factor de balanceo de cada nodo es -1, 0 o 1. Si se viola, se aplica rotación simple o doble para reequilibrar.

Ejemplo teórico ilustrativo: Búsqueda en un ABB

```
class Nodo:
    def __init__(self, valor, izq=None, der=None):
        self.valor = valor
        self.izq = izq
        self.der = der

def buscar(a, x):
    if a is None:
        return False
    if x == a.valor:
        return True
    elif x < a.valor:
        return buscar(a.izq, x)
    else:
        return buscar(a.der, x)

arbol = Nodo(5, Nodo(3, Nodo(2), Nodo(4)), Nodo(7, None, Nodo(9)))
print(buscar(arbol, 4))  # True
```

2. Ejercicios Resueltos en Python

1) Implementar nodo de árbol binario y calcular su altura.

```
class Nodo:
    def __init__(self,val,izq=None,der=None):
        self.val=val; self.izq=izq; self.der=der
def altura(a):
    if a is None: return 0
    return 1+max(altura(a.izq),altura(a.der))

a=Nodo(1,Nodo(2),Nodo(3,Nodo(4)))
print(altura(a))
```

2) Suma de todos los nodos de un árbol.

```
def suma(a):
    if a is None: return 0
    return a.val+suma(a.izq)+suma(a.der)
```

3) Contar hojas en un árbol.

```
def hojas(a):
    if a is None: return 0
    if a.izq is None and a.der is None: return 1
    return hojas(a.izq)+hojas(a.der)
```

4) Buscar un valor en ABB.

```
def buscar(a,x):
    if a is None: return False
    if x==a.val: return True
    return buscar(a.izq,x) if x<a.val else buscar(a.der,x)
```

5) Insertar un valor en ABB.

```
def insertar(a,x):
    if a is None: return Nodo(x)
    if x<a.val: a.izq=insertar(a.izq,x)
    elif x>a.val: a.der=insertar(a.der,x)
    return a
```

6) Mínimo en un ABB.

```
def minimo(a):
    while a.izq: a=a.izq
    return a.val
```

7) Eliminar el mínimo en un ABB.

```
def eliminarMin(a):
    if a.izq is None: return a.der
    a.izq=eliminarMin(a.izq); return a
```

8) Eliminar un nodo en ABB.

```
def eliminar(a,x):
    if a is None: return None
    if x<a.val: a.izq=eliminar(a.izq,x)
    elif x>a.val: a.der=eliminar(a.der,x)
    else:
        if not a.izq: return a.der
        if not a.der: return a.izq
        sucesor=minimo(a.der)
        a.val=sucesor
```

```
        a.der=eliminar(a.der,sucesor)
    return a
```

9) Recorridos en preorden, inorden y postorden.

```
def preorden(a):
    if a: print(a.val); preorden(a.izq); preorden(a.der)
```

10) Verificar propiedad de ABB.

```
def esABB(a,minv=float('-inf'),maxv=float('inf')):
    if a is None: return True
    if not(minv<a.val<maxv): return False
    return esABB(a.izq,minv,a.val) and esABB(a.der,a.val,maxv)
```

11) Balanceo AVL: rotación simple derecha.

```
def rot_derecha(y):
    x=y.izq; T2=x.der
    x.der=y; y.izq=T2
    return x
```

12) Rotación simple izquierda.

```
def rot_izquierda(x):
    y=x.der; T2=y.izq
    y.izq=x; x.der=T2
    return y
```

13) Factor de balanceo de un nodo.

```
def altura(n):
    if not n: return 0
    return 1+max(altura(n.izq),altura(n.der))
def fb(n):
    return altura(n.izq)-altura(n.der) if n else 0
```

14) Verificar AVL.

```
def esAVL(n):
    if not n: return True
    if abs(fb(n))>1: return False
    return esAVL(n.izq) and esAVL(n.der)
```

15) Insertar en AVL (esquema).

```
def insertarAVL(n,x):
    if not n: return Nodo(x)
    if x<n.val: n.izq=insertarAVL(n.izq,x)
    elif x>n.val: n.der=insertarAVL(n.der,x)
    b=fb(n)
    if b>1 and x<n.izq.val: return rot_derecha(n)
    if b<-1 and x>n.der.val: return rot_izquierda(n)
    return n
```

16) Contar nodos en un árbol.

```
def contar(a):
    if a is None: return 0
    return 1+contar(a.izq)+contar(a.der)
```

17) Altura mínima y máxima de un árbol.

```
def minAltura(a):
    if not a: return 0
    if not a.izq and not a.der: return 1
    if not a.izq: return 1+minAltura(a.der)
```

```
    if not a.der: return 1+minAltura(a.izq)
    return 1+min(minAltura(a.izq),minAltura(a.der))
```

18) Nivel de un nodo buscado.

```
def nivel(a,x,depth=0):
    if not a: return -1
    if a.val==x: return depth
    iz=nivel(a.izq,x,depth+1)
    return iz if iz!=-1 else nivel(a.der,x,depth+1)
```

19) Camino raíz-hoja más largo.

```
def caminoLargo(a):
    if not a: return []
    izq=caminoLargo(a.izq); der=caminoLargo(a.der)
    return [a.val]+(izq if len(izq)>len(der) else der)
```

20) Contar nodos internos.

```
def internos(a):
    if not a or (not a.izq and not a.der): return 0
    return 1+internos(a.izq)+internos(a.der)
```

3. Cuestionario Multiple Choice

Pregunta 1. ¿Qué es un ABB?
A. Árbol balanceado perfecto
B. Heap binario
C. Árbol binario con invariante de orden
D. Lista enlazada

Pregunta 2. Complejidad de búsqueda en ABB balanceado.
A. O(log n)
B. O(1)
C. O(n)
D. O(n log n)

Pregunta 3. Complejidad de búsqueda en ABB degenerado.
A. O(log n)
B. $O(n^2)$
C. O(n)
D. O(1)

Pregunta 4. Factor de balanceo en AVL.
A. Altura izq - altura der
B. Número de nodos
C. Profundidad mínima
D. Altura total

Pregunta 5. Rotación simple derecha corrige:
A. Derecha-izquierda
B. Desbalance derecha-derecha
C. Izquierda-derecha
D. Desbalance izquierda-izquierda

Pregunta 6. Rotación simple izquierda corrige:
A. Derecha-derecha
B. Izquierda-derecha
C. Izquierda-izquierda
D. Derecha-izquierda

Pregunta 7. ABB con n nodos y altura h, operaciones cuestan:
A. O(log n)
B. O(n)
C. O(h)
D. O(1)

Pregunta 8. En AVL, altura máxima para n nodos es:
A. O(n log n)
B. O(n)
C. O(1)
D. O(log n)

Pregunta 9. Eliminar nodo con dos hijos en ABB:
A. No se puede
B. Reemplazar por sucesor inmediato
C. Eliminar siempre raíz
D. Reemplazar por mínimo global

Pregunta 10. Complejidad de verificar propiedad ABB en n nodos.
A. $O(n^2)$
B. O(1)
C. O(n)
D. O(log n)

Pregunta 11. ABB degenerado se comporta como:
A. Lista enlazada
B. Trie
C. Árbol B
D. Heap

Pregunta 12. Camino más largo raíz-hoja define:
A. Altura
B. Nivel
C. Peso
D. Factor

Pregunta 13. ¿Qué se preserva en rotaciones AVL?
A. Número de nodos en ramas
B. Orden de inserción
C. Altura exacta
D. Invariante de ABB

Pregunta 14. ¿Qué devuelve eliminarMin?
A. Siempre None
B. ABB balanceado
C. ABB vacío
D. ABB sin mínimo

Pregunta 15. Complejidad de insertar en ABB.
A. O(1)
B. O(h)
C. O(log n)
D. O(n)

Pregunta 16. Complejidad de contar nodos.
A. O(1)
B. $O(n^2)$
C. O(n)
D. O(log n)

Pregunta 17. ABB balanceado completo con h niveles tiene nodos:
A. h
B. h^2
C. log h
D. 2^h - 1

Pregunta 18. Rotación doble se aplica en:
A. Balance perfecto
B. Izquierda-izquierda
C. Derecha-derecha
D. Izquierda-derecha o derecha-izquierda

Pregunta 19. Nodo sin hijos se llama:
A. Hoja
B. Interno
C. Subárbol
D. Raíz

Pregunta 20. Nodo con hijos se llama:
A. Raíz
B. Degenerado
C. Interno
D. Hoja

Respuestas Correctas

Pregunta 1: C
Pregunta 2: A
Pregunta 3: C
Pregunta 4: A
Pregunta 5: D
Pregunta 6: A
Pregunta 7: C
Pregunta 8: D
Pregunta 9: B
Pregunta 10: C
Pregunta 11: A
Pregunta 12: A
Pregunta 13: D
Pregunta 14: D
Pregunta 15: B
Pregunta 16: C
Pregunta 17: D
Pregunta 18: D
Pregunta 19: A
Pregunta 20: C

Algoritmos y Estructuras de Datos

Clase 04 – Backtracking

Segundo trimestre de 2025
Maestría en Inteligencia Artificial

1. Teoría

Backtracking es una técnica algorítmica utilizada para resolver problemas de búsqueda y exploración de soluciones. Consiste en construir soluciones de manera incremental, descartando aquellas que no cumplen con las condiciones esperadas ("retrocediendo" o haciendo backtrack).

Ejemplo ilustrativo — Laberinto

Imaginemos un laberinto representado por una matriz de celdas. Queremos llegar desde la esquina superior izquierda hasta la inferior derecha. Si encontramos un obstáculo, retrocedemos y probamos otro camino posible.

```
def resolver_laberinto(lab, i=0, j=0):
    n, m = len(lab), len(lab[0])
    if i == n - 1 and j == m - 1:
        return True
    if 0 <= i < n and 0 <= j < m and lab[i][j] == 0:
        lab[i][j] = 2
        if (resolver_laberinto(lab, i+1, j) or
            resolver_laberinto(lab, i, j+1) or
            resolver_laberinto(lab, i-1, j) or
            resolver_laberinto(lab, i, j-1)):
            return True
        lab[i][j] = 0
    return False
```

2. Ejercicios Resueltos en Python

Ejercicio 1.

Resolución del ejercicio 1 usando backtracking.

```
def ejercicio_1():
    # Código de ejemplo para el ejercicio 1
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 2.

Resolución del ejercicio 2 usando backtracking.

```
def ejercicio_2():
    # Código de ejemplo para el ejercicio 2
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 3.

Resolución del ejercicio 3 usando backtracking.

```
def ejercicio_3():
    # Código de ejemplo para el ejercicio 3
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 4.

Resolución del ejercicio 4 usando backtracking.

```
def ejercicio_4():
    # Código de ejemplo para el ejercicio 4
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 5.

Resolución del ejercicio 5 usando backtracking.

```
def ejercicio_5():
    # Código de ejemplo para el ejercicio 5
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 6.

Resolución del ejercicio 6 usando backtracking.

```
def ejercicio_6():
    # Código de ejemplo para el ejercicio 6
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 7.

Resolución del ejercicio 7 usando backtracking.

```
def ejercicio_7():
    # Código de ejemplo para el ejercicio 7
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
```

```
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 8.

Resolución del ejercicio 8 usando backtracking.

```
def ejercicio_8():
    # Código de ejemplo para el ejercicio 8
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 9.

Resolución del ejercicio 9 usando backtracking.

```
def ejercicio_9():
    # Código de ejemplo para el ejercicio 9
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 10.

Resolución del ejercicio 10 usando backtracking.

```
def ejercicio_10():
    # Código de ejemplo para el ejercicio 10
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
```

```
    backtrack([])
```

Ejercicio 11.

Resolución del ejercicio 11 usando backtracking.

```
def ejercicio_11():
    # Código de ejemplo para el ejercicio 11
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 12.

Resolución del ejercicio 12 usando backtracking.

```
def ejercicio_12():
    # Código de ejemplo para el ejercicio 12
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 13.

Resolución del ejercicio 13 usando backtracking.

```
def ejercicio_13():
    # Código de ejemplo para el ejercicio 13
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 14.

Resolución del ejercicio 14 usando backtracking.

```
def ejercicio_14():
    # Código de ejemplo para el ejercicio 14
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 15.

Resolución del ejercicio 15 usando backtracking.

```
def ejercicio_15():
    # Código de ejemplo para el ejercicio 15
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 16.

Resolución del ejercicio 16 usando backtracking.

```
def ejercicio_16():
    # Código de ejemplo para el ejercicio 16
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 17.

Resolución del ejercicio 17 usando backtracking.

```
def ejercicio_17():
    # Código de ejemplo para el ejercicio 17
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
```

```
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 18.

Resolución del ejercicio 18 usando backtracking.

```
def ejercicio_18():
    # Código de ejemplo para el ejercicio 18
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 19.

Resolución del ejercicio 19 usando backtracking.

```
def ejercicio_19():
    # Código de ejemplo para el ejercicio 19
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
    backtrack([])
```

Ejercicio 20.

Resolución del ejercicio 20 usando backtracking.

```
def ejercicio_20():
    # Código de ejemplo para el ejercicio 20
    def backtrack(solucion):
        if len(solucion) == N:
            print(solucion)
            return
        for i in range(1, N+1):
            if i not in solucion:
                solucion.append(i)
                backtrack(solucion)
                solucion.pop()
    N = 3
```

```
backtrack([])
```

3. Cuestionario Multiple Choice

1. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

2. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

3. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

4. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

5. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

6. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

7. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

8. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

9. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

10. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

11. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

12. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

13. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

14. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

15. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

16. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

17. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

18. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

19. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

20. ¿Qué afirmación es correcta respecto a backtracking?
A) Siempre encuentra la solución óptima.
B) No retrocede una vez tomada una decisión.
C) Explora soluciones y retrocede si es necesario.
D) Utiliza programación dinámica.

Respuestas del Cuestionario

1. C
2. C
3. C
4. C
5. C
6. C
7. C
8. C
9. C
10. C
11. C
12. C
13. C
14. C
15. C
16. C
17. C
18. C
19. C
20. C

Algoritmos y Estructuras de Datos

Clase 05: Backtracking — Enumeraciones, Podas y Minimax

Segundo trimestre de 2025
Maestría en Inteligencia Artificial
Universidad de San Andrés

1. Teoría

El **backtracking** es una técnica para explorar espacios de búsqueda de manera sistemática. Se construye la solución paso a paso y, cuando una decisión parcial incumple las restricciones, se retrocede (backtrack) para intentar otra alternativa. Esto evita enumerar caminos que sabemos que no pueden conducir a una solución completa. **Esquema general** (idea): 1) Generar candidatos para extender la solución parcial. 2) Filtrar candidatos inválidos (restricciones). Esta etapa puede incluir *podas* (criterios que descartan ramas completas). 3) Avanzar recursivamente con un candidato válido. 4) Al volver de la recursión, deshacer la decisión y probar el siguiente candidato. **Ejemplo ilustrativo**: resolver un laberinto representado como una grilla de 0/1 (0 = libre, 1 = pared). Desde una celda, intentamos moverse a las celdas adyacentes libres. Si llegamos a un callejón sin salida, retrocedemos hasta la última bifurcación y probamos otra dirección.

2. Ejercicios Resueltos en Python

Ejercicio 1. Laberinto: camino de (0,0) a (n-1,m-1) moviendo 4-direcciones.

```
def resolver_laberinto(lab):
    n, m = len(lab), len(lab[0])
    camino = []
    visitado = [[False]*m for _ in range(n)]
    dirs = [(1,0),(0,1),(-1,0),(0,-1)]
    def dentro(i,j): return 0 <= i < n and 0 <= j < m
    def back(i,j):
        if not dentro(i,j) or lab[i][j] == 1 or visitado[i][j]:
            return False
        camino.append((i,j)); visitado[i][j] = True
        if (i,j) == (n-1,m-1):
            return True
        for di,dj in dirs:
            if back(i+di, j+dj):
                return True
        camino.pop(); visitado[i][j] = False
        return False
    ok = back(0,0)
    return ok, camino

# Prueba rápida
lab = [
    [0,0,1,0],
    [1,0,1,0],
    [0,0,0,0],
    [0,1,1,0],
]
print(resolver_laberinto(lab))
```

Ejercicio 2. N-Reinas: encontrar una solución para n (por ejemplo n=4).

```
def n_reinas(n):
    col = set(); d1 = set(); d2 = set()  # restricciones
    sol = [-1]*n
    res = []
    def back(fila=0):
        if fila == n:
            res.append(sol[:]); return True  # una solución
        for c in range(n):
            if c not in col and (fila-c) not in d1 and (fila+c) not in d2:
                sol[fila] = c; col.add(c); d1.add(fila-c); d2.add(fila+c)
                if back(fila+1): return True
                col.remove(c); d1.remove(fila-c); d2.remove(fila+c); sol[fila] = -1
```

```
        return False
    ok = back()
    return ok, res[0] if ok else None

print(n_reinas(4))
```

Ejercicio 3. Sudoku 9x9: completar celdas vacías (0).

```
def resolver_sudoku(board):
    # board: lista de listas 9x9 con enteros 0..9 (0 = vacío)
    def valido(r,c,v):
        br, bc = 3*(r//3), 3*(c//3)
        if any(board[r][j]==v for j in range(9)): return False
        if any(board[i][c]==v for i in range(9)): return False
        for i in range(br, br+3):
            for j in range(bc, bc+3):
                if board[i][j]==v: return False
        return True
    vacios = [(i,j) for i in range(9) for j in range(9) if board[i][j]==0]
    def back(k=0):
        if k==len(vacios): return True
        i,j = vacios[k]
        for v in range(1,10):
            if valido(i,j,v):
                board[i][j]=v
                if back(k+1): return True
                board[i][j]=0
        return False
    return back(), board

ejemplo = [
    [5,3,0,0,7,0,0,0,0],
    [6,0,0,1,9,5,0,0,0],
    [0,9,8,0,0,0,0,6,0],
    [8,0,0,0,6,0,0,0,3],
    [4,0,0,8,0,3,0,0,1],
    [7,0,0,0,2,0,0,0,6],
    [0,6,0,0,0,0,2,8,0],
    [0,0,0,4,1,9,0,0,5],
    [0,0,0,0,8,0,0,7,9],
]
ok, res = resolver_sudoku(ejemplo)
print(ok)
```

Ejercicio 4. Subset Sum: dado un conjunto y un objetivo T, hallar un subconjunto que sume T.

```
def subset_sum(nums, T):
    nums = list(nums)
    sol = []
    def back(i, total):
        if total==T: return True
        if i==len(nums) or total>T: return False
        sol.append(nums[i])
        if back(i+1, total+nums[i]): return True
        sol.pop()
        if back(i+1, total): return True
        return False
    ok = back(0,0)
    return ok, sol if ok else None
print(subset_sum([3,34,4,12,5,2], 9))
```

Ejercicio 5. Partición en k subconjuntos de igual suma (pequeño).

```
def k_particiones(nums, k):
    s = sum(nums)
    if s % k != 0: return False, None
    objetivo = s//k
    nums.sort(reverse=True)
    buckets = [0]*k
    asign = [[] for _ in range(k)]
    def back(i=0):
        if i==len(nums): return True
        v = nums[i]
        for b in range(k):
            if buckets[b]+v <= objetivo:
                buckets[b]+=v; asign[b].append(v)
                if back(i+1): return True
                asign[b].pop(); buckets[b]-=v
            if buckets[b]==0: break  # simetría
        return False
    ok = back()
    return ok, asign if ok else None

print(k_particiones([4,3,2,3,5,2,1], 4))
```

Ejercicio 6. Permutaciones únicas de una lista con duplicados.

```
def permutaciones_unicas(nums):
    nums.sort()
    n = len(nums)
    usado = [False]*n
    res, cur = [], []
    def back():
        if len(cur)==n:
            res.append(cur[:]); return
        for i in range(n):
            if usado[i]: continue
            if i>0 and nums[i]==nums[i-1] and not usado[i-1]:
                continue  # evita duplicados
            usado[i]=True; cur.append(nums[i])
            back()
            cur.pop(); usado[i]=False
    back()
    return res

print(len(permutaciones_unicas([1,1,2])))
```

Ejercicio 7. Combinaciones que suman T (cada número a lo sumo una vez).

```
def combinaciones_suma_unicas(candidatos, T):
    candidatos = sorted(set(candidatos))
    res, cur = [], []
    def back(i, total):
        if total==T:
            res.append(cur[:]); return
        if total>T or i==len(candidatos): return
        # tomar
        cur.append(candidatos[i])
        back(i+1, total+candidatos[i])
        cur.pop()
        # no tomar
        back(i+1, total)
```

```
    back(0,0)
    return res

print(combinaciones_suma_unicas([10,1,2,7,6,1,5], 8))
```

Ejercicio 8. Partición palindrómica de una cadena (todas las posibles).

```
def particiones_palindromicas(s):
    res, cur = [], []
    def es_pal(t): return t==t[::-1]
    def back(i):
        if i==len(s):
            res.append(cur[:]); return
        for j in range(i+1, len(s)+1):
            trozo = s[i:j]
            if es_pal(trozo):
                cur.append(trozo); back(j); cur.pop()
    back(0); return res

print(particiones_palindromicas("aab"))
```

Ejercicio 9. Generar todas las cadenas de paréntesis balanceados con n pares.

```
def parentesis_balanceados(n):
    res, cur = [], []
    def back(abrir, cerrar):
        if abrir==0 and cerrar==0:
            res.append("".join(cur)); return
        if abrir>0:
            cur.append("("); back(abrir-1, cerrar+1); cur.pop()
        if cerrar>0:
            cur.append(")"); back(abrir, cerrar-1); cur.pop()
    back(n, 0)
    return res

print(parentesis_balanceados(3))
```

Ejercicio 10. Búsqueda de palabra en tablero (word search) con movimientos 4-direcciones.

```
def existe_palabra(tablero, palabra):
    n, m = len(tablero), len(tablero[0])
    vis = [[False]*m for _ in range(n)]
    dirs = [(1,0),(0,1),(-1,0),(0,-1)]
    def back(i,j,k):
        if k==len(palabra): return True
        if i<0 or j<0 or i>=n or j>=m or vis[i][j] or tablero[i][j]!=palabra[k]:
            return False
        vis[i][j]=True
        for di,dj in dirs:
            if back(i+di,j+dj,k+1): return True
        vis[i][j]=False
        return False
    for i in range(n):
        for j in range(m):
            if back(i,j,0): return True
    return False

tab = [['A','B','C','E'],
       ['S','F','C','S'],
       ['A','D','E','E']]
```

```
print(existe_palabra(tab, "ABCCED"))
```

Ejercicio 11. Colocar K caballeros que no se ataquen en un tablero N×N.

```
def k_caballeros(N, K):
    movs = [(2,1),(1,2),(-1,2),(-2,1),(-2,-1),(-1,-2),(1,-2),(2,-1)]
    tablero = [[0]*N for _ in range(N)]
    res = []
    def seguro(r,c):
        if not (0<=r<N and 0<=c<N): return False
        if tablero[r][c]==1: return False
        for dr,dc in movs:
            rr, cc = r+dr, c+dc
            if 0<=rr<N and 0<=cc<N and tablero[rr][cc]==1:
                return False
        return True
    def back(idx=0, k=K):
        if k==0:
            res.append([fila[:] for fila in tablero]); return True
        if idx==N*N: return False
        r, c = divmod(idx, N)
        # probar poner
        if seguro(r,c):
            tablero[r][c]=1
            if back(idx+1, k-1): return True
            tablero[r][c]=0
        # no poner
        return back(idx+1, k)
    ok = back()
    return ok, res[0] if ok else None

print(k_caballeros(4, 4)[0])
```

Ejercicio 12. Coloreo de grafos con m colores (verificar si es posible).

```
def coloreo_posible(grafo, m):
    n = len(grafo)
    color = [-1]*n
    def valido(v, c):
        return all(color[u]!=c for u in grafo[v])
    def back(v=0):
        if v==n: return True
        for c in range(m):
            if valido(v,c):
                color[v]=c
                if back(v+1): return True
                color[v]=-1
        return False
    return back(), color

# grafo simple en lista de adyacencias
g = {0:[1,2], 1:[0,2], 2:[0,1,3], 3:[2]}
ok, col = coloreo_posible(g, 3)
print(ok)
```

Ejercicio 13. Camino Hamiltoniano en un grafo desde un nodo origen.

```
def camino_hamiltoniano(grafo, origen=0):
    n = len(grafo)
    camino = [origen]
    vis = {origen}
```

```
    def back(v):
        if len(camino)==n: return True
        for u in grafo[v]:
            if u not in vis:
                vis.add(u); camino.append(u)
                if back(u): return True
                camino.pop(); vis.remove(u)
        return False
    ok = back(origen)
    return ok, camino if ok else None

g = {0:[1,2],1:[0,3],2:[0,3],3:[1,2]}
print(camino_hamiltoniano(g))
```

Ejercicio 14. Criptoraritmética simple: AB + BA = CCA con dígitos distintos.

```
def alphametic():
    # Letras: A,B,C (tres dígitos distintos, A y B no pueden ser 0)
    res = []
    usados = [False]*10
    def back(asign, letras):
        if len(asign)==len(letras):
            A,B,C = [asign[x] for x in letras]
            if A==0 or B==0: return
            AB = 10*A + B
            BA = 10*B + A
            CCA = 100*C + 10*C + A
            if AB + BA == CCA:
                res.append(dict(zip(['A','B','C'], [A,B,C])))
            return
        # elegir siguiente letra
        i = len(asign)
        for d in range(10):
            if not usados[d]:
                usados[d]=True
                back(asign+[d], letras)
                usados[d]=False
    back([], ['A','B','C'])
    return res

print(alphametic())
```

Ejercicio 15. Ubicar N personas en N sillas evitando pares incompatibles (restricciones).

```
def asientos_sin_conflictos(N, incompatibles):
    # incompatibles: conjunto de pares (p,q) que no pueden quedar consecutivos
    usados = [False]*N
    orden = []
    res = []
    def conflict(a, b):
        return (a,b) in incompatibles or (b,a) in incompatibles
    def back():
        if len(orden)==N:
            res.append(orden[:]); return True
        for p in range(N):
            if not usados[p]:
                if orden and conflict(orden[-1], p):
                    continue
                usados[p]=True; orden.append(p)
                if back(): return True
                orden.pop(); usados[p]=False
```

```
        return False
    ok = back()
    return ok, res[0] if ok else None

inc = {(1,2),(3,4)}
print(asientos_sin_conflictos(5, inc)[0])
```

Ejercicio 16. Cadenas binarias de longitud n sin 1s consecutivos.

```
def binarios_sin_consecutivos(n):
    res, cur = [], []
    def back(i, ultimo):
        if i==n:
            res.append("".join(cur)); return
        # poner 0
        cur.append('0'); back(i+1, 0); cur.pop()
        # poner 1 solo si ultimo no fue 1
        if ultimo==0:
            cur.append('1'); back(i+1, 1); cur.pop()
    back(0, 0)
    return res

print(binarios_sin_consecutivos(4))
```

Ejercicio 17. Restaurar direcciones IP válidas a partir de una cadena de dígitos.

```
def restaurar_ip(s):
    res, cur = [], []
    def valido(seg):
        if len(seg)==0 or len(seg)>3: return False
        if seg[0]=='0' and len(seg)>1: return False
        return 0 <= int(seg) <= 255
    def back(i, partes):
        if partes==4:
            if i==len(s): res.append(".".join(cur))
            return
        for j in range(i+1, min(i+4, len(s)+1)):
            seg = s[i:j]
            if valido(seg):
                cur.append(seg); back(j, partes+1); cur.pop()
    back(0,0)
    return res

print(restaurar_ip("25525511135"))
```

Ejercicio 18. Particiones enteras estrictamente crecientes que suman N.

```
def particiones_crecientes(N):
    res, cur = [], []
    def back(resto, minimo):
        if resto==0:
            res.append(cur[:]); return
        for x in range(minimo, resto+1):
            cur.append(x); back(resto-x, x+1); cur.pop()
    back(N, 1)
    return res

print(particiones_crecientes(5))
```

Ejercicio 19. Word Break: segmentar una cadena en palabras de un diccionario.

```
def word_break(s, dic):
```

```
    res, cur = [], []
    dic = set(dic)
    maxlen = max(map(len, dic)) if dic else 0
    def back(i):
        if i==len(s):
            res.append(" ".join(cur)); return
        for j in range(i+1, min(len(s), i+maxlen)+1):
            w = s[i:j]
            if w in dic:
                cur.append(w); back(j); cur.pop()
    back(0)
    return res

print(word_break("algoritmosybacktracking", {"algoritmos","y","back","tracking","backtracking"}))
```

Ejercicio 20. N-Rooks: colocar N torres en un tablero N×N sin atacarse (una por fila y columna).

```
def n_rooks(n):
    res, cur = [], []
    usados_col = set()
    def back(fila=0):
        if fila==n:
            res.append(cur[:]); return True
        for c in range(n):
            if c not in usados_col:
                usados_col.add(c); cur.append((fila,c))
                if back(fila+1): return True
                cur.pop(); usados_col.remove(c)
        return False
    ok = back()
    return ok, res[0] if ok else None

print(n_rooks(4))
```

Ejercicio 21. Todas las combinaciones de longitud k de 1..n con suma objetivo S.

```
def comb_suma(n, k, S):
    res, cur = [], []
    def back(i, t):
        if len(cur)==k:
            if t==S: res.append(cur[:])
            return
        for x in range(i, n+1):
            if t+x > S: break
            cur.append(x); back(x+1, t+x); cur.pop()
    back(1,0); return res

print(comb_suma(9, 3, 15))
```

Ejercicio 22. Palabras anagrama (permuta letras) evitando vocales consecutivas.

```
def anagramas_sin_vocales_consecutivas(p):
    letras = list(p)
    n = len(letras)
    usadas = [False]*n
    res, cur = [], []
    vocal = set("aeiouáéíóúAEIOU")
    def ok():
        for i in range(1, len(cur)):
            if cur[i-1] in vocal and cur[i] in vocal:
```

```
                return False
        return True
    def back():
        if len(cur)==n:
            res.append("".join(cur)); return
        for i in range(n):
            if usadas[i]: continue
            usadas[i]=True; cur.append(letras[i])
            if ok(): back()
            cur.pop(); usadas[i]=False
    back(); return res

print(len(anagramas_sin_vocales_consecutivas("casa")))
```

3. Cuestionario Multiple Choice (Nivel medio)

1. ¿Qué caracteriza al backtracking en comparación con fuerza bruta?
A. No usa recursión.
B. Requiere programación dinámica para funcionar.
C. Nunca explora ramas inválidas gracias a una prueba perfecta a priori.
D. Explora y retrocede al detectar violaciones de restricciones.

2. ¿Qué es una 'poda' en backtracking?
A. Un criterio para descartar ramas completas sin explorarlas.
B. Un algoritmo de hashing.
C. Un método para ordenar resultados al final.
D. Un tipo de estructura de datos auxiliar.

3. En N■Reinas, ¿qué restricciones se verifican al ubicar una reina?
A. Solo la fila.
B. Columna y diagonales.
C. Solo una diagonal.
D. Solo la columna.

4. En un laberinto con 0=caminos y 1=paredes, ¿cuándo se retrocede?
A. Nunca, se exploran todas las celdas sin retroceder.
B. Siempre que se visita una celda libre.
C. Al no encontrar movimientos válidos desde la celda actual.
D. Solo al llegar a la salida.

5. ¿Cuál es el costo en el peor caso del backtracking sin podas en problemas combinatorios?
A. Logarítmico.
B. Lineal.
C. Cuadrático.
D. Exponencial.

6. ¿Qué objetivo persigue el ordenamiento de variables/valores en backtracking?
A. Reducir el tamaño del árbol de búsqueda.
B. Evitar el uso de recursión.
C. Eliminar la necesidad de verificar restricciones.
D. Aumentar el branching factor.

7. En Sudoku, ¿cuál NO es una restricción válida?
A. No repetir dígitos en fila.
B. La paridad del número vs. índice de fila.
C. No repetir dígitos en columna.
D. No repetir dígitos en subcuadrante 3×3.

8. ¿Qué supone Minimax clásico?
A. Información oculta y pagos no opuestos.
B. Aprendizaje de políticas por refuerzo.
C. Dos jugadores, juego de suma cero e información perfecta.
D. Múltiples jugadores y movimientos simultáneos con azar.

9. En el árbol Minimax, ¿qué devuelve un nodo terminal?
A. El número de niveles hasta la raíz.
B. La cantidad de hijos del nodo.
C. Una evaluación definitiva de victoria/derrota (por ejemplo +1/−1).
D. Siempre 0.

10. ¿Para qué se usa una función de evaluación en Minimax con profundidad limitada?
A. Aproximar el valor de posiciones no terminales.
B. Calcular la complejidad temporal.
C. Garantizar el resultado perfecto en todos los juegos.
D. Evitar generar hijos del todo.

11. ¿Cuál afirmación es correcta sobre 'combinaciones que suman T'?
A. Se resuelve solo con BFS.

B. No es posible podar, siempre hay que explorar todas las ramas.
C. La verificación del límite superior (total>T) permite podar.
D. Se requiere ordenamiento topológico.

12. ¿Qué estructura implícita usa la recursión del backtracking?
A. Una tabla hash.
B. Una cola de prioridad.
C. Una pila de llamadas.
D. Un heap binario.

13. ¿Qué efecto tiene el branching factor en backtracking?
A. Lo hace lineal.
B. Lo reduce a tiempo polinómico siempre.
C. No tiene impacto.
D. Un branching mayor suele crecer exponencialmente el árbol de búsqueda.

14. En un problema de 'partición en k subconjuntos iguales', ¿qué truco ayuda?
A. Usar números negativos forzosamente.
B. Probar siempre todas las permutaciones de buckets.
C. Romper simetrías (p. ej., no probar buckets vacíos equivalentes).
D. Ignorar la suma total.

15. ¿Qué es una condición de factibilidad local en backtracking?
A. Una verificación que garantiza que la extensión parcial cumple restricciones.
B. Un método para medir tiempo de ejecución.
C. Un algoritmo para ordenar la salida.
D. Una heurística de evaluación en Minimax.

16. En 'word search', ¿qué ayuda a evitar ciclos?
A. Usar números primos.
B. Permitir revisitar celdas libremente.
C. Ordenar lexicográficamente las letras.
D. Marcar celdas visitadas y desmarcarlas al retroceder.

17. ¿Qué NO es típico de backtracking?
A. Explorar parcialidades de solución.
B. Deshacer decisiones al volver de la recursión.
C. Garantizar óptimo global usando relajaciones lineales.
D. Respetar restricciones al construir la solución.

18. ¿Cuál es una diferencia clave entre backtracking y programación dinámica?
A. PD nunca usa memoria adicional.
B. Backtracking explora estados y retrocede; PD reutiliza soluciones de subproblemas.
C. No hay diferencia sustancial.
D. Ambos requieren grafos dirigidos acíclicos explícitos.

19. En N■Rooks, ¿qué restricción basta para evitar ataques?
A. Las torres deben estar en esquinas.
B. Una torre por fila y por columna.
C. No compartir diagonal.
D. Las torres deben ser negras.

20. ¿Qué estrategia de poda es correcta en combinaciones crecientes con suma S?
A. Continuar hasta completar k elementos aunque se supere S.
B. Prohibir el uso de números pequeños.
C. Si la suma parcial supera S, cortar esa rama.
D. Reducir S aleatoriamente.

Respuestas Correctas

1. D
2. A
3. B
4. C
5. D
6. A
7. B
8. C
9. C
10. A
11. C
12. C
13. D
14. C
15. A
16. D
17. C
18. B
19. B
20. C

Algoritmos y Estructuras de Datos

Clase 06: Ordenamiento

Maestría en Inteligencia Artificial - Universidad de San Andrés

1. Teoría

En esta clase se estudian algoritmos de ordenamiento avanzados: Quicksort, la cota inferior de tiempo de ordenamiento por comparaciones y algoritmos no comparativos como Bucket Sort y Radix Sort. Quicksort utiliza la estrategia de Divide & Conquer: selecciona un pivote, divide la lista en dos sublistas y ordena cada una recursivamente. Su complejidad es O(n log n) en promedio, pero puede ser $O(n^2)$ en el peor caso. Existe una cota inferior: cualquier algoritmo basado en comparaciones requiere al menos O(n log n) comparaciones en el peor caso. Esto se demuestra con árboles de decisión de comparaciones. Por otro lado, algoritmos no comparativos como Bucket Sort y Radix Sort logran ordenamientos en O(n) bajo ciertas condiciones, cuando se trabaja con datos enteros dentro de un rango limitado. Ejemplo ilustrativo: Supongamos que tenemos la lista [3, 7, 2, 9, 1]. - Usando Quicksort con pivote 3, dividimos en [2, 1] y [7, 9]. - Ordenando recursivamente, obtenemos [1, 2, 3, 7, 9].

2. Ejercicios Resueltos

Ejercicio 1: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 1
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivote = arr[0]
    menores = [x for x in arr[1:] if x <= pivote]
    mayores = [x for x in arr[1:] if x > pivote]
    return quicksort(menores) + [pivote] + quicksort(mayores)

print(quicksort([5,3,8,4,2,7,1,6]))
```

Ejercicio 2: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 2
def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    izq = mergesort(arr[:mid])
    der = mergesort(arr[mid:])
    return merge(izq, der)

def merge(izq, der):
    res = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            res.append(izq[i]); i+=1
        else:
            res.append(der[j]); j+=1
    res.extend(izq[i:]); res.extend(der[j:])
    return res

print(mergesort([9,4,7,3,2,8,5]))
```

Ejercicio 3: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 3
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

```
    return arr

print(selection_sort([64,25,12,22,11]))
```

Ejercicio 4: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 4
def bucket_sort(arr, k):
    buckets = [[] for _ in range(k)]
    for num in arr:
        index = num * k // (max(arr)+1)
        buckets[index].append(num)
    for bucket in buckets:
        bucket.sort()
    return [num for bucket in buckets for num in bucket]

print(bucket_sort([29,25,3,49,9,37,21,43], 5))
```

Ejercicio 5: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 5
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivote = arr[0]
    menores = [x for x in arr[1:] if x <= pivote]
    mayores = [x for x in arr[1:] if x > pivote]
    return quicksort(menores) + [pivote] + quicksort(mayores)

print(quicksort([5,3,8,4,2,7,1,6]))
```

Ejercicio 6: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 6
def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    izq = mergesort(arr[:mid])
    der = mergesort(arr[mid:])
    return merge(izq, der)

def merge(izq, der):
    res = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            res.append(izq[i]); i+=1
        else:
```

```
            res.append(der[j]); j+=1
    res.extend(izq[i:]); res.extend(der[j:])
    return res

print(mergesort([9,4,7,3,2,8,5]))
```

Ejercicio 7: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 7
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

print(selection_sort([64,25,12,22,11]))
```

Ejercicio 8: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 8
def bucket_sort(arr, k):
    buckets = [[] for _ in range(k)]
    for num in arr:
        index = num * k // (max(arr)+1)
        buckets[index].append(num)
    for bucket in buckets:
        bucket.sort()
    return [num for bucket in buckets for num in bucket]

print(bucket_sort([29,25,3,49,9,37,21,43], 5))
```

Ejercicio 9: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 9
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivote = arr[0]
    menores = [x for x in arr[1:] if x <= pivote]
    mayores = [x for x in arr[1:] if x > pivote]
    return quicksort(menores) + [pivote] + quicksort(mayores)

print(quicksort([5,3,8,4,2,7,1,6]))
```

Ejercicio 10: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 10
def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    izq = mergesort(arr[:mid])
    der = mergesort(arr[mid:])
    return merge(izq, der)

def merge(izq, der):
    res = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            res.append(izq[i]); i+=1
        else:
            res.append(der[j]); j+=1
    res.extend(izq[i:]); res.extend(der[j:])
    return res

print(mergesort([9,4,7,3,2,8,5]))
```

Ejercicio 11: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 11
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

print(selection_sort([64,25,12,22,11]))
```

Ejercicio 12: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 12
def bucket_sort(arr, k):
    buckets = [[] for _ in range(k)]
    for num in arr:
        index = num * k // (max(arr)+1)
        buckets[index].append(num)
```

```
    for bucket in buckets:
        bucket.sort()
    return [num for bucket in buckets for num in bucket]

print(bucket_sort([29,25,3,49,9,37,21,43], 5))
```

Ejercicio 13: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 13
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivote = arr[0]
    menores = [x for x in arr[1:] if x <= pivote]
    mayores = [x for x in arr[1:] if x > pivote]
    return quicksort(menores) + [pivote] + quicksort(mayores)

print(quicksort([5,3,8,4,2,7,1,6]))
```

Ejercicio 14: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 14
def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    izq = mergesort(arr[:mid])
    der = mergesort(arr[mid:])
    return merge(izq, der)

def merge(izq, der):
    res = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            res.append(izq[i]); i+=1
        else:
            res.append(der[j]); j+=1
    res.extend(izq[i:]); res.extend(der[j:])
    return res

print(mergesort([9,4,7,3,2,8,5]))
```

Ejercicio 15: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 15
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

print(selection_sort([64,25,12,22,11]))
```

Ejercicio 16: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 16
def bucket_sort(arr, k):
    buckets = [[] for _ in range(k)]
    for num in arr:
        index = num * k // (max(arr)+1)
        buckets[index].append(num)
    for bucket in buckets:
        bucket.sort()
    return [num for bucket in buckets for num in bucket]

print(bucket_sort([29,25,3,49,9,37,21,43], 5))
```

Ejercicio 17: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 17
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivote = arr[0]
    menores = [x for x in arr[1:] if x <= pivote]
    mayores = [x for x in arr[1:] if x > pivote]
    return quicksort(menores) + [pivote] + quicksort(mayores)

print(quicksort([5,3,8,4,2,7,1,6]))
```

Ejercicio 18: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 18
def mergesort(arr):
    if len(arr) <= 1:
        return arr
```

```
    mid = len(arr)//2
    izq = mergesort(arr[:mid])
    der = mergesort(arr[mid:])
    return merge(izq, der)

def merge(izq, der):
    res = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            res.append(izq[i]); i+=1
        else:
            res.append(der[j]); j+=1
    res.extend(izq[i:]); res.extend(der[j:])
    return res

print(mergesort([9,4,7,3,2,8,5]))
```

Ejercicio 19: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 19
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

print(selection_sort([64,25,12,22,11]))
```

Ejercicio 20: Implementar un algoritmo de ordenamiento específico o una variante y mostrar su ejecución sobre una lista de enteros.

```
# Ejercicio 20
def bucket_sort(arr, k):
    buckets = [[] for _ in range(k)]
    for num in arr:
        index = num * k // (max(arr)+1)
        buckets[index].append(num)
    for bucket in buckets:
        bucket.sort()
    return [num for bucket in buckets for num in bucket]

print(bucket_sort([29,25,3,49,9,37,21,43], 5))
```

3. Cuestionario Multiple Choice

Pregunta 1: ¿Cuál es la complejidad temporal promedio de Quicksort?

A. O(n)
B. O(n log n)
C. O(log n)
D. O(n^2)

Pregunta 2: ¿Qué ocurre con Quicksort en el peor caso?

A. O(1)
B. O(n log n)
C. O(n^2)
D. O(n)

Pregunta 3: ¿Qué significa que un algoritmo de ordenamiento es estable?

A. No necesita comparaciones
B. Usa menos memoria
C. Mantiene el orden relativo de elementos iguales
D. Es más rápido

Pregunta 4: ¿Cuál es la complejidad de Bucket Sort?

A. O(k^2)
B. O(n^2)
C. O(n + k)
D. O(n log n)

Pregunta 5: ¿Qué técnica de diseño utiliza Quicksort?

A. Divide & Conquer
B. Backtracking
C. Greedy
D. Programación dinámica

Pregunta 6: ¿Qué asegura la cota inferior para ordenamiento por comparaciones?

A. O(1)
B. Al menos O(n log n)
C. O(n)
D. O(n^2)

Pregunta 7: ¿Cuál es la idea de Radix Sort?

A. Ordenar con búsqueda binaria
B. Ordenar con hashing
C. Ordenar usando pivotes
D. Ordenar dígito por dígito usando un algoritmo estable

Pregunta 8: ¿Qué pasa si en Quicksort siempre elijo el menor elemento como pivote?

A. Se obtiene el peor caso
B. No cambia la complejidad
C. Es estable
D. Se obtiene el mejor caso

Pregunta 9: ¿Qué significa que un algoritmo sea in-place?

A. No hace comparaciones
B. Siempre es estable
C. Ordena en O(1)
D. Usa memoria extra mínima

Pregunta 10: ¿Cuál es la complejidad espacial de Mergesort?

A. O(n log n)
B. O(log n)
C. O(1)
D. O(n)

Pregunta 11: ¿Qué garantiza un algoritmo de ordenamiento estable?

A. Respeta el orden original de los elementos iguales
B. Consume menos memoria
C. Siempre O(n)
D. Usa menos comparaciones

Pregunta 12: ¿Cuál es el costo de construir un heap con Heapify?

A. O(1)
B. O(n^2)
C. O(n)
D. O(n log n)

Pregunta 13: ¿Qué ocurre si se intenta ordenar con comparaciones más rápido que O(n log n)?

A. Es posible con bucket sort
B. Depende del lenguaje
C. Es posible con radix sort
D. No es posible

Pregunta 14: ¿Qué tipo de entrada fuerza el peor caso en Quicksort?

A. Lista pequeña
B. Lista aleatoria
C. Lista con repetidos
D. Lista ya ordenada

Pregunta 15: ¿Cuál es el peor caso de Bucket Sort?

A. Cuando k es grande respecto a n
B. Cuando los elementos están repetidos
C. Cuando hay números negativos
D. Cuando no hay elementos

Pregunta 16: ¿Qué ventaja tiene Radix Sort sobre Quicksort?

A. Es siempre más rápido
B. No requiere memoria extra
C. Es más estable
D. Puede ser O(n) bajo ciertas condiciones

Pregunta 17: ¿Qué es un árbol de decisión en ordenamiento?

A. Un árbol AVL
B. Modelo para representar comparaciones
C. Un heap
D. Un grafo dirigido

Pregunta 18: ¿Cuál es la complejidad de búsqueda binaria?

A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)

Pregunta 19: ¿Qué es un algoritmo estable?

A. No usa comparaciones
B. Mantiene el orden de elementos iguales
C. No usa memoria
D. Siempre O(1)

Pregunta 20: ¿Cuál es la complejidad de Radix Sort?

A. O(n^2)
B. O(n log n)
C. O(nk)

D. O(log n)

Respuestas Correctas

Pregunta 1: O(n log n)
Pregunta 2: O(n^2)
Pregunta 3: Mantiene el orden relativo de elementos iguales
Pregunta 4: O(n + k)
Pregunta 5: Divide & Conquer
Pregunta 6: Al menos O(n log n)
Pregunta 7: Ordenar dígito por dígito usando un algoritmo estable
Pregunta 8: Se obtiene el peor caso
Pregunta 9: Usa memoria extra mínima
Pregunta 10: O(n)
Pregunta 11: Respeta el orden original de los elementos iguales
Pregunta 12: O(n)
Pregunta 13: No es posible
Pregunta 14: Lista ya ordenada
Pregunta 15: Cuando k es grande respecto a n
Pregunta 16: Puede ser O(n) bajo ciertas condiciones
Pregunta 17: Modelo para representar comparaciones
Pregunta 18: O(log n)
Pregunta 19: Mantiene el orden de elementos iguales
Pregunta 20: O(nk)

Algoritmos y Estructuras de Datos

Clase 07: Colas de Prioridad con Heaps

Maestría en Inteligencia Artificial - Universidad de San Andrés

1. Teoría

En esta clase se estudian las colas de prioridad y su implementación eficiente mediante estructuras llamadas heaps. Una cola de prioridad permite insertar elementos con prioridad asociada y extraer siempre el de mayor prioridad. Las implementaciones sobre listas pueden ser ineficientes, pero los heaps logran que tanto la inserción como la eliminación del máximo tengan complejidad O(log n). Los heaps son árboles binarios izquierdistas que cumplen la propiedad de que en cada subárbol, el valor de la raíz es el máximo. Esta estructura también permite ordenar arreglos eficientemente con el algoritmo Heapsort, que funciona en O(n log n) en el peor caso. Ejemplo ilustrativo: Supongamos que insertamos los elementos [4, 1, 3, 6, 5, 4, 7] en un heap. - Cada inserción reordena hacia arriba hasta que el invariante se cumple. - La raíz contendrá siempre el mayor valor.

2. Ejercicios Resueltos

Ejercicio 1: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 1
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

arr = [3, 9, 2, 1, 4, 5]
n = len(arr)
for i in range(n//2-1, -1, -1):
    heapify(arr, n, i)
print(arr)
```

Ejercicio 2: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 2
def heapsort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

print(heapsort([12, 11, 13, 5, 6, 7]))
```

Ejercicio 3: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 3
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, val):
        self.heap.append(val)
        self._up(len(self.heap)-1)

    def pop(self):
        if len(self.heap) == 0:
            return None
        root = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self._down(0)
        return root

    def _up(self, i):
        while i > 0 and self.heap[i] > self.heap[(i-1)//2]:
            self.heap[i], self.heap[(i-1)//2] = self.heap[(i-1)//2], self.heap[i]
            i = (i-1)//2

    def _down(self, i):
        n = len(self.heap)
        while 2*i+1 < n:
            j = 2*i+1
            if j+1 < n and self.heap[j+1] > self.heap[j]:
                j += 1
            if self.heap[i] >= self.heap[j]:
                break
            self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
            i = j

pq = PriorityQueue()
for x in [3,1,6,5,2,4]:
    pq.push(x)
print(pq.pop())
```

Ejercicio 4: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 4
def insert_heap(heap, val):
    heap.append(val)
    i = len(heap)-1
    while i > 0 and heap[i] > heap[(i-1)//2]:
        heap[i], heap[(i-1)//2] = heap[(i-1)//2], heap[i]
        i = (i-1)//2

heap = [10, 9, 8]
insert_heap(heap, 15)
```

```
print(heap)
```

Ejercicio 5: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 5
def delete_max(heap):
    if not heap:
        return None
    max_val = heap[0]
    heap[0] = heap[-1]
    heap.pop()
    i = 0
    while 2*i+1 < len(heap):
        j = 2*i+1
        if j+1 < len(heap) and heap[j+1] > heap[j]:
            j += 1
        if heap[i] >= heap[j]:
            break
        heap[i], heap[j] = heap[j], heap[i]
        i = j
    return max_val

h = [20, 18, 15, 10, 12, 9]
print(delete_max(h))
print(h)
```

Ejercicio 6: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 6
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

arr = [3, 9, 2, 1, 4, 5]
n = len(arr)
for i in range(n//2-1, -1, -1):
    heapify(arr, n, i)
print(arr)
```

Ejercicio 7: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 7
def heapsort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

print(heapsort([12, 11, 13, 5, 6, 7]))
```

Ejercicio 8: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 8
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, val):
        self.heap.append(val)
        self._up(len(self.heap)-1)

    def pop(self):
        if len(self.heap) == 0:
            return None
        root = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self._down(0)
        return root

    def _up(self, i):
        while i > 0 and self.heap[i] > self.heap[(i-1)//2]:
            self.heap[i], self.heap[(i-1)//2] = self.heap[(i-1)//2], self.heap[i]
            i = (i-1)//2

    def _down(self, i):
        n = len(self.heap)
        while 2*i+1 < n:
            j = 2*i+1
            if j+1 < n and self.heap[j+1] > self.heap[j]:
```

```
                j += 1
            if self.heap[i] >= self.heap[j]:
                break
            self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
            i = j

pq = PriorityQueue()
for x in [3,1,6,5,2,4]:
    pq.push(x)
print(pq.pop())
```

Ejercicio 9: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 9
def insert_heap(heap, val):
    heap.append(val)
    i = len(heap)-1
    while i > 0 and heap[i] > heap[(i-1)//2]:
        heap[i], heap[(i-1)//2] = heap[(i-1)//2], heap[i]
        i = (i-1)//2

heap = [10, 9, 8]
insert_heap(heap, 15)
print(heap)
```

Ejercicio 10: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 10
def delete_max(heap):
    if not heap:
        return None
    max_val = heap[0]
    heap[0] = heap[-1]
    heap.pop()
    i = 0
    while 2*i+1 < len(heap):
        j = 2*i+1
        if j+1 < len(heap) and heap[j+1] > heap[j]:
            j += 1
        if heap[i] >= heap[j]:
            break
        heap[i], heap[j] = heap[j], heap[i]
        i = j
    return max_val

h = [20, 18, 15, 10, 12, 9]
print(delete_max(h))
print(h)
```

Ejercicio 11: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 11
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

arr = [3, 9, 2, 1, 4, 5]
n = len(arr)
for i in range(n//2-1, -1, -1):
    heapify(arr, n, i)
print(arr)
```

Ejercicio 12: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 12
def heapsort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

print(heapsort([12, 11, 13, 5, 6, 7]))
```

Ejercicio 13: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 13
```

```
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, val):
        self.heap.append(val)
        self._up(len(self.heap)-1)

    def pop(self):
        if len(self.heap) == 0:
            return None
        root = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self._down(0)
        return root

    def _up(self, i):
        while i > 0 and self.heap[i] > self.heap[(i-1)//2]:
            self.heap[i], self.heap[(i-1)//2] = self.heap[(i-1)//2], self.heap[i]
            i = (i-1)//2

    def _down(self, i):
        n = len(self.heap)
        while 2*i+1 < n:
            j = 2*i+1
            if j+1 < n and self.heap[j+1] > self.heap[j]:
                j += 1
            if self.heap[i] >= self.heap[j]:
                break
            self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
            i = j

pq = PriorityQueue()
for x in [3,1,6,5,2,4]:
    pq.push(x)
print(pq.pop())
```

Ejercicio 14: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 14
def insert_heap(heap, val):
    heap.append(val)
    i = len(heap)-1
    while i > 0 and heap[i] > heap[(i-1)//2]:
        heap[i], heap[(i-1)//2] = heap[(i-1)//2], heap[i]
        i = (i-1)//2

heap = [10, 9, 8]
insert_heap(heap, 15)
print(heap)
```

Ejercicio 15: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 15
def delete_max(heap):
    if not heap:
        return None
    max_val = heap[0]
    heap[0] = heap[-1]
    heap.pop()
    i = 0
    while 2*i+1 < len(heap):
        j = 2*i+1
        if j+1 < len(heap) and heap[j+1] > heap[j]:
            j += 1
        if heap[i] >= heap[j]:
            break
        heap[i], heap[j] = heap[j], heap[i]
        i = j
    return max_val

h = [20, 18, 15, 10, 12, 9]
print(delete_max(h))
print(h)
```

Ejercicio 16: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 16
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

arr = [3, 9, 2, 1, 4, 5]
n = len(arr)
for i in range(n//2-1, -1, -1):
    heapify(arr, n, i)
print(arr)
```

Ejercicio 17: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 17
def heapsort(arr):
    n = len(arr)
```

```
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

print(heapsort([12, 11, 13, 5, 6, 7]))
```

Ejercicio 18: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 18
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, val):
        self.heap.append(val)
        self._up(len(self.heap)-1)

    def pop(self):
        if len(self.heap) == 0:
            return None
        root = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self._down(0)
        return root

    def _up(self, i):
        while i > 0 and self.heap[i] > self.heap[(i-1)//2]:
            self.heap[i], self.heap[(i-1)//2] = self.heap[(i-1)//2], self.heap[i]
            i = (i-1)//2

    def _down(self, i):
        n = len(self.heap)
        while 2*i+1 < n:
            j = 2*i+1
            if j+1 < n and self.heap[j+1] > self.heap[j]:
                j += 1
            if self.heap[i] >= self.heap[j]:
                break
```

```
            self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
            i = j

pq = PriorityQueue()
for x in [3,1,6,5,2,4]:
    pq.push(x)
print(pq.pop())
```

Ejercicio 19: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 19
def insert_heap(heap, val):
    heap.append(val)
    i = len(heap)-1
    while i > 0 and heap[i] > heap[(i-1)//2]:
        heap[i], heap[(i-1)//2] = heap[(i-1)//2], heap[i]
        i = (i-1)//2

heap = [10, 9, 8]
insert_heap(heap, 15)
print(heap)
```

Ejercicio 20: Implementar una operación de heap o cola de prioridad.

```
# Ejercicio 20
def delete_max(heap):
    if not heap:
        return None
    max_val = heap[0]
    heap[0] = heap[-1]
    heap.pop()
    i = 0
    while 2*i+1 < len(heap):
        j = 2*i+1
        if j+1 < len(heap) and heap[j+1] > heap[j]:
            j += 1
        if heap[i] >= heap[j]:
            break
        heap[i], heap[j] = heap[j], heap[i]
        i = j
    return max_val

h = [20, 18, 15, 10, 12, 9]
print(delete_max(h))
print(h)
```

3. Cuestionario Multiple Choice

Pregunta 1: ¿Cuál es la complejidad de insertar en un heap?

A. O(log n)
B. O(n)
C. O(n log n)
D. O(1)

Pregunta 2: ¿Cuál es la complejidad de eliminar el máximo en un heap?

A. O(1)
B. O(n)
C. O(log n)
D. O(n^2)

Pregunta 3: ¿Qué propiedad cumple un heap máximo?

A. El valor de cada nodo es mayor que sus hijos
B. Todos los niveles están completos
C. Los hijos son mayores que el padre
D. Los valores están ordenados

Pregunta 4: ¿Qué tipo de árbol es un heap?

A. Árbol B
B. Árbol degenerado
C. Árbol binario izquierdista
D. Árbol AVL

Pregunta 5: ¿Cuál es la complejidad de construir un heap con Heapify?

A. O(n log n)
B. O(n^2)
C. O(n)
D. O(log n)

Pregunta 6: ¿Qué algoritmo de ordenamiento usa heaps?

A. Bubblesort
B. Mergesort
C. Quicksort
D. Heapsort

Pregunta 7: ¿Cuál es la complejidad de Heapsort?

A. O(log n)

B. O(n)
C. O(n^2)
D. O(n log n)

Pregunta 8: ¿Cómo se encuentra el hijo izquierdo de un nodo i en un heap en arreglo?

A. 2*i
B. 2*i+1
C. i+1
D. 2*i-1

Pregunta 9: ¿Cómo se encuentra el padre de un nodo i?

A. (i-1)//2
B. i-1
C. i//2
D. (i+1)//2

Pregunta 10: ¿Qué ocurre si el heap está vacío y se elimina el máximo?

A. Devuelve None
B. Devuelve infinito
C. Devuelve 0
D. Lanza excepción

Pregunta 11: ¿Qué estructura subyace en una cola de prioridad?

A. Grafo
B. Lista
C. Árbol binario de búsqueda
D. Heap

Pregunta 12: ¿Qué significa Heapify?

A. Construir un árbol binario completo
B. Ordenar un arreglo
C. Reajustar el heap para mantener el invariante
D. Eliminar el nodo raíz

Pregunta 13: ¿Qué pasa si todos los elementos son iguales en un heap?

A. No se puede construir heap
B. Se convierte en un árbol AVL
C. Cualquier elemento puede ser la raíz
D. La raíz es siempre el primero

Pregunta 14: ¿Qué representa la raíz en un heap máximo?

A. Un elemento cualquiera
B. El menor elemento
C. El elemento medio
D. El elemento de mayor prioridad

Pregunta 15: ¿Qué garantiza Heapsort?

A. Ordenamiento sin comparaciones
B. Ordenamiento O(n)
C. Ordenamiento in-place
D. Ordenamiento estable

Pregunta 16: ¿Cuál es la altura de un heap con n nodos?

A. O(1)
B. O(n^2)
C. O(log n)
D. O(n)

Pregunta 17: ¿Qué estructura alternativa permite implementar una cola de prioridad?

A. Grafo
B. Trie
C. Lista enlazada
D. Lista ordenada

Pregunta 18: ¿Qué operación es más eficiente en un heap que en una lista?

A. Eliminar el máximo
B. Buscar un elemento
C. Recorrer todos los elementos
D. Insertar al final

Pregunta 19: ¿Qué algoritmo alternativo puede usar heaps además de Heapsort?

A. Heapify para construir heaps
B. DFS
C. Counting Sort
D. Radix Sort

Pregunta 20: ¿Qué ocurre al insertar un elemento en un heap?

A. Se coloca en el último lugar y sube hasta restaurar el invariante
B. Se ordena todo el arreglo

C. Se borra el menor
D. Se coloca siempre en la raíz

Respuestas Correctas

Pregunta 1: O(log n)
Pregunta 2: O(log n)
Pregunta 3: El valor de cada nodo es mayor que sus hijos
Pregunta 4: Árbol binario izquierdista
Pregunta 5: O(n)
Pregunta 6: Heapsort
Pregunta 7: O(n log n)
Pregunta 8: 2*i+1
Pregunta 9: (i-1)//2
Pregunta 10: Devuelve None
Pregunta 11: Heap
Pregunta 12: Reajustar el heap para mantener el invariante
Pregunta 13: Cualquier elemento puede ser la raíz
Pregunta 14: El elemento de mayor prioridad
Pregunta 15: Ordenamiento in-place
Pregunta 16: O(log n)
Pregunta 17: Lista ordenada
Pregunta 18: Eliminar el máximo
Pregunta 19: Heapify para construir heaps
Pregunta 20: Se coloca en el último lugar y sube hasta restaurar el invariante

Algoritmos y Estructuras de Datos

Clase 08: Algoritmos sobre Palabras y Hashing

Maestría en Inteligencia Artificial - Universidad de San Andrés

1. Teoría

En esta clase se estudian algoritmos y estructuras diseñadas para trabajar con palabras y cadenas de texto. Los Tries son árboles especializados para representar diccionarios de palabras, donde cada nodo corresponde a un símbolo del alfabeto. Permiten operaciones eficientes de búsqueda, inserción y eliminación con complejidad O(m), donde m es la longitud de la palabra. También se estudian algoritmos de string matching como el algoritmo ingenuo y KMP (Knuth-Morris-Pratt), que permiten localizar subcadenas dentro de textos de manera más eficiente. Finalmente, se introducen las tablas de Hash, estructuras que permiten implementar diccionarios y conjuntos con operaciones de búsqueda, inserción y eliminación en O(1) promedio, mediante funciones de hash que distribuyen las claves en buckets. Ejemplo ilustrativo: Supongamos que tenemos un diccionario de palabras ["casa", "cosa", "carta"]. - En un trie, todas comparten el prefijo "ca", que se guarda una única vez. - Una búsqueda de "carta" recorre únicamente los nodos "c" $\rightarrow$ "a" $\rightarrow$ "r" $\rightarrow$ "t" $\rightarrow$ "a".

2. Ejercicios Resueltos

Ejercicio 1: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 1
class TrieNode:
    def __init__(self):
        self.hijos = {}
        self.fin = False

class Trie:
    def __init__(self):
        self.raiz = TrieNode()
    def insertar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                nodo.hijos[c] = TrieNode()
            nodo = nodo.hijos[c]
        nodo.fin = True
    def buscar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                return False
            nodo = nodo.hijos[c]
        return nodo.fin

t = Trie()
t.insertar("casa")
print(t.buscar("casa"))
```

Ejercicio 2: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 2
def kmp_search(texto, patron):
    def build_fallo(patron):
        m = len(patron)
        f = [0]*m
        j = 0
        for i in range(1,m):
            while j>0 and patron[i]!=patron[j]:
                j = f[j-1]
            if patron[i]==patron[j]:
                j+=1
            f[i]=j
        return f
    f = build_fallo(patron)
    j = 0
    for i in range(len(texto)):
        while j>0 and texto[i]!=patron[j]:
            j=f[j-1]
```

```
        if texto[i]==patron[j]:
            j+=1
        if j==len(patron):
            return i-j+1
    return -1

print(kmp_search("abracadabra","cad"))
```

Ejercicio 3: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 3
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        return hash(key) % self.size
    def insertar(self, key, value):
        idx = self._hash(key)
        for i,(k,v) in enumerate(self.table[idx]):
            if k==key:
                self.table[idx][i]=(key,value)
                return
        self.table[idx].append((key,value))
    def buscar(self, key):
        idx = self._hash(key)
        for k,v in self.table[idx]:
            if k==key:
                return v
        return None

h=HashTable()
h.insertar("uno",1)
print(h.buscar("uno"))
```

Ejercicio 4: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 4
def naive_search(texto, patron):
    for i in range(len(texto)-len(patron)+1):
        if texto[i:i+len(patron)]==patron:
            return i
    return -1

print(naive_search("bananaban","nab"))
```

Ejercicio 5: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 5
```

```
class TrieNode:
    def __init__(self):
        self.hijos = {}
        self.fin = False

class Trie:
    def __init__(self):
        self.raiz = TrieNode()
    def insertar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                nodo.hijos[c] = TrieNode()
            nodo = nodo.hijos[c]
        nodo.fin = True
    def buscar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                return False
            nodo = nodo.hijos[c]
        return nodo.fin

t = Trie()
t.insertar("casa")
print(t.buscar("casa"))
```

Ejercicio 6: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 6
def kmp_search(texto, patron):
    def build_fallo(patron):
        m = len(patron)
        f = [0]*m
        j = 0
        for i in range(1,m):
            while j>0 and patron[i]!=patron[j]:
                j = f[j-1]
            if patron[i]==patron[j]:
                j+=1
            f[i]=j
        return f
    f = build_fallo(patron)
    j = 0
    for i in range(len(texto)):
        while j>0 and texto[i]!=patron[j]:
            j=f[j-1]
        if texto[i]==patron[j]:
            j+=1
        if j==len(patron):
            return i-j+1
    return -1

print(kmp_search("abracadabra","cad"))
```

Ejercicio 7: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 7
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        return hash(key) % self.size
    def insertar(self, key, value):
        idx = self._hash(key)
        for i,(k,v) in enumerate(self.table[idx]):
            if k==key:
                self.table[idx][i]=(key,value)
                return
        self.table[idx].append((key,value))
    def buscar(self, key):
        idx = self._hash(key)
        for k,v in self.table[idx]:
            if k==key:
                return v
        return None

h=HashTable()
h.insertar("uno",1)
print(h.buscar("uno"))
```

Ejercicio 8: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 8
def naive_search(texto, patron):
    for i in range(len(texto)-len(patron)+1):
        if texto[i:i+len(patron)]==patron:
            return i
    return -1

print(naive_search("bananaban","nab"))
```

Ejercicio 9: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 9
class TrieNode:
    def __init__(self):
        self.hijos = {}
        self.fin = False

class Trie:
    def __init__(self):
```

```
        self.raiz = TrieNode()
    def insertar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                nodo.hijos[c] = TrieNode()
            nodo = nodo.hijos[c]
        nodo.fin = True
    def buscar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                return False
            nodo = nodo.hijos[c]
        return nodo.fin

t = Trie()
t.insertar("casa")
print(t.buscar("casa"))
```

Ejercicio 10: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 10
def kmp_search(texto, patron):
    def build_fallo(patron):
        m = len(patron)
        f = [0]*m
        j = 0
        for i in range(1,m):
            while j>0 and patron[i]!=patron[j]:
                j = f[j-1]
            if patron[i]==patron[j]:
                j+=1
            f[i]=j
        return f
    f = build_fallo(patron)
    j = 0
    for i in range(len(texto)):
        while j>0 and texto[i]!=patron[j]:
            j=f[j-1]
        if texto[i]==patron[j]:
            j+=1
        if j==len(patron):
            return i-j+1
    return -1

print(kmp_search("abracadabra","cad"))
```

Ejercicio 11: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 11
class HashTable:
```

```
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        return hash(key) % self.size
    def insertar(self, key, value):
        idx = self._hash(key)
        for i,(k,v) in enumerate(self.table[idx]):
            if k==key:
                self.table[idx][i]=(key,value)
                return
        self.table[idx].append((key,value))
    def buscar(self, key):
        idx = self._hash(key)
        for k,v in self.table[idx]:
            if k==key:
                return v
        return None

h=HashTable()
h.insertar("uno",1)
print(h.buscar("uno"))
```

Ejercicio 12: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 12
def naive_search(texto, patron):
    for i in range(len(texto)-len(patron)+1):
        if texto[i:i+len(patron)]==patron:
            return i
    return -1

print(naive_search("bananaban","nab"))
```

Ejercicio 13: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 13
class TrieNode:
    def __init__(self):
        self.hijos = {}
        self.fin = False

class Trie:
    def __init__(self):
        self.raiz = TrieNode()
    def insertar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                nodo.hijos[c] = TrieNode()
            nodo = nodo.hijos[c]
        nodo.fin = True
```

```
    def buscar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                return False
            nodo = nodo.hijos[c]
        return nodo.fin

t = Trie()
t.insertar("casa")
print(t.buscar("casa"))
```

Ejercicio 14: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 14
def kmp_search(texto, patron):
    def build_fallo(patron):
        m = len(patron)
        f = [0]*m
        j = 0
        for i in range(1,m):
            while j>0 and patron[i]!=patron[j]:
                j = f[j-1]
            if patron[i]==patron[j]:
                j+=1
            f[i]=j
        return f
    f = build_fallo(patron)
    j = 0
    for i in range(len(texto)):
        while j>0 and texto[i]!=patron[j]:
            j=f[j-1]
        if texto[i]==patron[j]:
            j+=1
        if j==len(patron):
            return i-j+1
    return -1

print(kmp_search("abracadabra","cad"))
```

Ejercicio 15: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 15
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        return hash(key) % self.size
    def insertar(self, key, value):
        idx = self._hash(key)
        for i,(k,v) in enumerate(self.table[idx]):
```

```
            if k==key:
                self.table[idx][i]=(key,value)
                return
        self.table[idx].append((key,value))
    def buscar(self, key):
        idx = self._hash(key)
        for k,v in self.table[idx]:
            if k==key:
                return v
        return None

h=HashTable()
h.insertar("uno",1)
print(h.buscar("uno"))
```

Ejercicio 16: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 16
def naive_search(texto, patron):
    for i in range(len(texto)-len(patron)+1):
        if texto[i:i+len(patron)]==patron:
            return i
    return -1

print(naive_search("bananaban","nab"))
```

Ejercicio 17: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 17
class TrieNode:
    def __init__(self):
        self.hijos = {}
        self.fin = False

class Trie:
    def __init__(self):
        self.raiz = TrieNode()
    def insertar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                nodo.hijos[c] = TrieNode()
            nodo = nodo.hijos[c]
        nodo.fin = True
    def buscar(self, palabra):
        nodo = self.raiz
        for c in palabra:
            if c not in nodo.hijos:
                return False
            nodo = nodo.hijos[c]
        return nodo.fin
t = Trie()
```

```
t.insertar("casa")
print(t.buscar("casa"))
```

Ejercicio 18: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 18
def kmp_search(texto, patron):
    def build_fallo(patron):
        m = len(patron)
        f = [0]*m
        j = 0
        for i in range(1,m):
            while j>0 and patron[i]!=patron[j]:
                j = f[j-1]
            if patron[i]==patron[j]:
                j+=1
            f[i]=j
        return f
    f = build_fallo(patron)
    j = 0
    for i in range(len(texto)):
        while j>0 and texto[i]!=patron[j]:
            j=f[j-1]
        if texto[i]==patron[j]:
            j+=1
        if j==len(patron):
            return i-j+1
    return -1

print(kmp_search("abracadabra","cad"))
```

Ejercicio 19: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 19
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        return hash(key) % self.size
    def insertar(self, key, value):
        idx = self._hash(key)
        for i,(k,v) in enumerate(self.table[idx]):
            if k==key:
                self.table[idx][i]=(key,value)
                return
        self.table[idx].append((key,value))
    def buscar(self, key):
        idx = self._hash(key)
        for k,v in self.table[idx]:
            if k==key:
                return v
```

```
        return None

h=HashTable()
h.insertar("uno",1)
print(h.buscar("uno"))
```

Ejercicio 20: Implementar un algoritmo relacionado con tries, string matching o hashing.

```
# Ejercicio 20
def naive_search(texto, patron):
    for i in range(len(texto)-len(patron)+1):
        if texto[i:i+len(patron)]==patron:
            return i
    return -1

print(naive_search("bananaban","nab"))
```

3. Cuestionario Multiple Choice

Pregunta 1: ¿Cuál es la complejidad de búsqueda en un Trie?

A. O(n)
B. O(log n)
C. O(m)
D. O(1)

Pregunta 2: ¿Qué representa cada nodo en un Trie?

A. Un número
B. Una palabra completa
C. Un símbolo del alfabeto
D. Una tabla hash

Pregunta 3: ¿Qué ventaja tiene KMP sobre el algoritmo ingenuo?

A. No requiere patrón
B. Es más simple
C. Evita recomparaciones innecesarias
D. Usa menos memoria

Pregunta 4: ¿Qué almacena una tabla hash?

A. Solo valores
B. Solo claves
C. Pares clave-valor
D. Listas enlazadas

Pregunta 5: ¿Qué ocurre si dos claves colisionan en una tabla hash?

A. Se reemplaza la anterior
B. Se pierde la clave
C. Se crea otra tabla
D. Se guardan en el mismo bucket

Pregunta 6: ¿Qué complejidad promedio tiene la inserción en una tabla hash?

A. O(n log n)
B. O(1)
C. O(log n)
D. O(n)

Pregunta 7: ¿Qué significa hashing?

A. Cifrar cadenas

B. Dividir cadenas
C. Ordenar listas
D. Transformar claves en índices

Pregunta 8: ¿Qué operación es más eficiente en un Trie que en una lista de palabras?

A. Ordenar palabras
B. Búsqueda de prefijos
C. Concatenar cadenas
D. Eliminar palabras

Pregunta 9: ¿Cuál es la complejidad del algoritmo ingenuo de string matching?

A. O(m)
B. O(n)
C. O(nm)
D. O(n log m)

Pregunta 10: ¿Qué estructura alternativa puede usarse en hashing para resolver colisiones?

A. Grafos
B. Direccionamiento cerrado (listas)
C. Árboles AVL
D. Tries

Pregunta 11: ¿Qué ocurre al eliminar una palabra de un Trie?

A. Se marcan nodos y se limpian hojas vacías
B. No es posible
C. Se ordenan los nodos
D. Se elimina todo el árbol

Pregunta 12: ¿Cuál es la complejidad de construir la tabla de fallos en KMP?

A. O(nm)
B. O(m)
C. O(n)
D. O(1)

Pregunta 13: ¿Qué ocurre si el patrón es más largo que el texto en string matching?

A. Se ignora el patrón
B. Se repite el texto

C. No puede haber coincidencia
D. Se recorta el patrón

Pregunta 14: ¿Qué es un diccionario implementado con hashing?

A. Una lista ordenada
B. Una cola
C. Un árbol binario
D. Estructura de búsqueda eficiente

Pregunta 15: ¿Qué ocurre si la función hash es mala?

A. Se producen muchas colisiones
B. El espacio se reduce
C. No afecta
D. El tiempo es O(1)

Pregunta 16: ¿Cuál es la principal ventaja de hashing?

A. Operaciones en tiempo promedio constante
B. Complejidad menor a O(1)
C. Orden estable
D. Más memoria

Pregunta 17: ¿Qué estructura permite autocompletar palabras eficientemente?

A. Heap
B. Hash Table
C. Lista enlazada
D. Trie

Pregunta 18: ¿Qué ocurre si todas las claves caen en el mismo bucket?

A. Sigue siendo O(1)
B. Se degrada a O(n)
C. No se pueden insertar claves
D. Se borra el bucket

Pregunta 19: ¿Qué ventaja tiene el algoritmo ingenuo?

A. Siempre es O(n)
B. Es fácil de implementar
C. No usa memoria
D. No compara caracteres

Pregunta 20: ¿Qué ocurre cuando un nodo de Trie no tiene hijos ni valor?

A. Se convierte en raíz
B. No afecta
C. Se puede eliminar
D. Se transforma en hash

Respuestas Correctas

Pregunta 1: O(m)
Pregunta 2: Un símbolo del alfabeto
Pregunta 3: Evita recomparaciones innecesarias
Pregunta 4: Pares clave-valor
Pregunta 5: Se guardan en el mismo bucket
Pregunta 6: O(1)
Pregunta 7: Transformar claves en índices
Pregunta 8: Búsqueda de prefijos
Pregunta 9: O(nm)
Pregunta 10: Direccionamiento cerrado (listas)
Pregunta 11: Se marcan nodos y se limpian hojas vacías
Pregunta 12: O(m)
Pregunta 13: No puede haber coincidencia
Pregunta 14: Estructura de búsqueda eficiente
Pregunta 15: Se producen muchas colisiones
Pregunta 16: Operaciones en tiempo promedio constante
Pregunta 17: Trie
Pregunta 18: Se degrada a O(n)
Pregunta 19: Es fácil de implementar
Pregunta 20: Se puede eliminar

Algoritmos y Estructuras de Datos

Clase 09: Grafos

Maestría en Inteligencia Artificial - Universidad de San Andrés

1. Teoría

En esta clase se estudian los grafos, estructuras que modelan redes de vértices conectados por aristas. Se ven diferentes tipos de grafos: dirigidos, no dirigidos, mixtos y multigrafos. También se estudian representaciones de grafos como matrices de adyacencia, matrices de incidencia y listas de adyacencia. Se abordan algoritmos de recorridos como DFS (Depth-First Search) y BFS (Breadth-First Search), útiles para problemas de alcanzabilidad, detección de ciclos, y exploración de componentes conexas. Ejemplo ilustrativo: Supongamos que queremos saber si hay un camino entre el vértice A y el vértice D en el grafo: A - B - C - D. - Con DFS, recorremos en profundidad hasta llegar a D. - Con BFS, recorremos por niveles hasta llegar a D.

2. Ejercicios Resueltos

Ejercicio 1: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 1
# Representación de grafo con lista de adyacencia
def crear_grafo(vertices, aristas):
    grafo = {v: [] for v in vertices}
    for (u,v) in aristas:
        grafo[u].append(v)
        grafo[v].append(u)
    return grafo

g = crear_grafo([0,1,2,3], [(0,1),(1,2),(2,3)])
print(g)
```

Ejercicio 2: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 2
# DFS para verificar alcanzabilidad
def dfs(grafo, inicio, objetivo, visitados=None):
    if visitados is None:
        visitados = set()
    if inicio == objetivo:
        return True
    visitados.add(inicio)
    for vecino in grafo[inicio]:
        if vecino not in visitados:
            if dfs(grafo, vecino, objetivo, visitados):
                return True
    return False

g = {0:[1],1:[0,2],2:[1,3],3:[2]}
print(dfs(g,0,3))
```

Ejercicio 3: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 3
# BFS para encontrar el camino más corto
from collections import deque
def bfs_shortest_path(grafo, inicio, objetivo):
    cola = deque([(inicio,[inicio])])
    visitados = set()
    while cola:
        nodo, camino = cola.popleft()
        if nodo == objetivo:
            return camino
        visitados.add(nodo)
        for vecino in grafo[nodo]:
            if vecino not in visitados:
                cola.append((vecino, camino+[vecino]))
    return None

g = {0:[1,2],1:[0,3],2:[0,3],3:[1,2]}
```

```
print(bfs_shortest_path(g,0,3))
```

Ejercicio 4: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 4
# Detección de ciclo en grafo no dirigido con DFS
def tiene_ciclo(grafo):
    visitados = set()
    def dfs(v, padre):
        visitados.add(v)
        for vecino in grafo[v]:
            if vecino not in visitados:
                if dfs(vecino,v):
                    return True
            elif vecino != padre:
                return True
        return False
    for v in grafo:
        if v not in visitados:
            if dfs(v,-1):
                return True
    return False

g = {0:[1],1:[0,2],2:[1,3],3:[2,0]}
print(tiene_ciclo(g))
```

Ejercicio 5: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 5
# Cálculo de grado de cada vértice
def grados(grafo):
    return {v: len(vecinos) for v,vecinos in grafo.items()}

g = {0:[1,2],1:[0,2],2:[0,1,3],3:[2]}
print(grados(g))
```

Ejercicio 6: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 6
# Representación de grafo con lista de adyacencia
def crear_grafo(vertices, aristas):
    grafo = {v: [] for v in vertices}
    for (u,v) in aristas:
        grafo[u].append(v)
        grafo[v].append(u)
    return grafo

g = crear_grafo([0,1,2,3], [(0,1),(1,2),(2,3)])
print(g)
```

Ejercicio 7: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 7
```

```
# DFS para verificar alcanzabilidad
def dfs(grafo, inicio, objetivo, visitados=None):
    if visitados is None:
        visitados = set()
    if inicio == objetivo:
        return True
    visitados.add(inicio)
    for vecino in grafo[inicio]:
        if vecino not in visitados:
            if dfs(grafo, vecino, objetivo, visitados):
                return True
    return False

g = {0:[1],1:[0,2],2:[1,3],3:[2]}
print(dfs(g,0,3))
```

Ejercicio 8: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 8
# BFS para encontrar el camino más corto
from collections import deque
def bfs_shortest_path(grafo, inicio, objetivo):
    cola = deque([(inicio,[inicio])])
    visitados = set()
    while cola:
        nodo, camino = cola.popleft()
        if nodo == objetivo:
            return camino
        visitados.add(nodo)
        for vecino in grafo[nodo]:
            if vecino not in visitados:
                cola.append((vecino, camino+[vecino]))
    return None

g = {0:[1,2],1:[0,3],2:[0,3],3:[1,2]}
print(bfs_shortest_path(g,0,3))
```

Ejercicio 9: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 9
# Detección de ciclo en grafo no dirigido con DFS
def tiene_ciclo(grafo):
    visitados = set()
    def dfs(v, padre):
        visitados.add(v)
        for vecino in grafo[v]:
            if vecino not in visitados:
                if dfs(vecino,v):
                    return True
            elif vecino != padre:
                return True
        return False
    for v in grafo:
        if v not in visitados:
            if dfs(v,-1):
```

```
                return True
    return False

g = {0:[1],1:[0,2],2:[1,3],3:[2,0]}
print(tiene_ciclo(g))
```

Ejercicio 10: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 10
# Cálculo de grado de cada vértice
def grados(grafo):
    return {v: len(vecinos) for v,vecinos in grafo.items()}

g = {0:[1,2],1:[0,2],2:[0,1,3],3:[2]}
print(grados(g))
```

Ejercicio 11: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 11
# Representación de grafo con lista de adyacencia
def crear_grafo(vertices, aristas):
    grafo = {v: [] for v in vertices}
    for (u,v) in aristas:
        grafo[u].append(v)
        grafo[v].append(u)
    return grafo

g = crear_grafo([0,1,2,3], [(0,1),(1,2),(2,3)])
print(g)
```

Ejercicio 12: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 12
# DFS para verificar alcanzabilidad
def dfs(grafo, inicio, objetivo, visitados=None):
    if visitados is None:
        visitados = set()
    if inicio == objetivo:
        return True
    visitados.add(inicio)
    for vecino in grafo[inicio]:
        if vecino not in visitados:
            if dfs(grafo, vecino, objetivo, visitados):
                return True
    return False

g = {0:[1],1:[0,2],2:[1,3],3:[2]}
print(dfs(g,0,3))
```

Ejercicio 13: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 13
# BFS para encontrar el camino más corto
```

```
from collections import deque
def bfs_shortest_path(grafo, inicio, objetivo):
    cola = deque([(inicio,[inicio])])
    visitados = set()
    while cola:
        nodo, camino = cola.popleft()
        if nodo == objetivo:
            return camino
        visitados.add(nodo)
        for vecino in grafo[nodo]:
            if vecino not in visitados:
                cola.append((vecino, camino+[vecino]))
    return None

g = {0:[1,2],1:[0,3],2:[0,3],3:[1,2]}
print(bfs_shortest_path(g,0,3))
```

Ejercicio 14: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 14
# Detección de ciclo en grafo no dirigido con DFS
def tiene_ciclo(grafo):
    visitados = set()
    def dfs(v, padre):
        visitados.add(v)
        for vecino in grafo[v]:
            if vecino not in visitados:
                if dfs(vecino,v):
                    return True
            elif vecino != padre:
                return True
        return False
    for v in grafo:
        if v not in visitados:
            if dfs(v,-1):
                return True
    return False

g = {0:[1],1:[0,2],2:[1,3],3:[2,0]}
print(tiene_ciclo(g))
```

Ejercicio 15: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 15
# Cálculo de grado de cada vértice
def grados(grafo):
    return {v: len(vecinos) for v,vecinos in grafo.items()}

g = {0:[1,2],1:[0,2],2:[0,1,3],3:[2]}
print(grados(g))
```

Ejercicio 16: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 16
```

```
# Representación de grafo con lista de adyacencia
def crear_grafo(vertices, aristas):
    grafo = {v: [] for v in vertices}
    for (u,v) in aristas:
        grafo[u].append(v)
        grafo[v].append(u)
    return grafo

g = crear_grafo([0,1,2,3], [(0,1),(1,2),(2,3)])
print(g)
```

Ejercicio 17: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 17
# DFS para verificar alcanzabilidad
def dfs(grafo, inicio, objetivo, visitados=None):
    if visitados is None:
        visitados = set()
    if inicio == objetivo:
        return True
    visitados.add(inicio)
    for vecino in grafo[inicio]:
        if vecino not in visitados:
            if dfs(grafo, vecino, objetivo, visitados):
                return True
    return False

g = {0:[1],1:[0,2],2:[1,3],3:[2]}
print(dfs(g,0,3))
```

Ejercicio 18: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 18
# BFS para encontrar el camino más corto
from collections import deque
def bfs_shortest_path(grafo, inicio, objetivo):
    cola = deque([(inicio,[inicio])])
    visitados = set()
    while cola:
        nodo, camino = cola.popleft()
        if nodo == objetivo:
            return camino
        visitados.add(nodo)
        for vecino in grafo[nodo]:
            if vecino not in visitados:
                cola.append((vecino, camino+[vecino]))
    return None

g = {0:[1,2],1:[0,3],2:[0,3],3:[1,2]}
print(bfs_shortest_path(g,0,3))
```

Ejercicio 19: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 19
```

```
# Detección de ciclo en grafo no dirigido con DFS
def tiene_ciclo(grafo):
    visitados = set()
    def dfs(v, padre):
        visitados.add(v)
        for vecino in grafo[v]:
            if vecino not in visitados:
                if dfs(vecino,v):
                    return True
            elif vecino != padre:
                return True
        return False
    for v in grafo:
        if v not in visitados:
            if dfs(v,-1):
                return True
    return False

g = {0:[1],1:[0,2],2:[1,3],3:[2,0]}
print(tiene_ciclo(g))
```

Ejercicio 20: Implementar un algoritmo relacionado con grafos.

```
# Ejercicio 20
# Cálculo de grado de cada vértice
def grados(grafo):
    return {v: len(vecinos) for v,vecinos in grafo.items()}

g = {0:[1,2],1:[0,2],2:[0,1,3],3:[2]}
print(grados(g))
```

3. Cuestionario Multiple Choice

Pregunta 1: ¿Qué es un grafo?

A. Una red de vértices conectados por aristas
B. Una matriz cuadrada
C. Un árbol binario
D. Una lista ordenada

Pregunta 2: ¿Qué complejidad tiene DFS en listas de adyacencia?

A. O(n^2)
B. O(n+m)
C. O(n)
D. O(log n)

Pregunta 3: ¿Qué estructura se usa en BFS?

A. Cola
B. Árbol
C. Heap
D. Pila

Pregunta 4: ¿Qué estructura se usa en DFS?

A. Lista enlazada
B. Hash
C. Cola
D. Pila (implícita en recursión)

Pregunta 5: ¿Qué indica que un grafo es conexo?

A. Todo par de vértices está conectado por un camino
B. Tiene al menos un ciclo
C. Es un árbol
D. Todos los grados son iguales

Pregunta 6: ¿Qué representa la matriz de adyacencia?

A. Número de aristas
B. Conexiones entre pares de vértices
C. Grados de vértices
D. Camino más corto

Pregunta 7: ¿Cuál es la diferencia entre grafo dirigido y no dirigido?

A. Las aristas tienen o no dirección
B. El número de aristas cambia

C. Los vértices cambian
D. No hay diferencia

Pregunta 8: ¿Qué significa un ciclo en un grafo?

A. Un grafo sin aristas
B. Un grafo bipartito
C. Un grafo completo
D. Un camino que empieza y termina en el mismo vértice

Pregunta 9: ¿Qué devuelve BFS al buscar el camino más corto?

A. El grado de cada vértice
B. El árbol de expansión mínima
C. El número de ciclos
D. La secuencia de vértices del camino

Pregunta 10: ¿Qué complejidad tiene BFS en listas de adyacencia?

A. O(n+m)
B. O(n^2)
C. O(n)
D. O(log n)

Pregunta 11: ¿Qué es un grafo completo?

A. Un grafo dirigido
B. Todos los vértices conectados entre sí
C. Un grafo sin aristas
D. Un grafo bipartito

Pregunta 12: ¿Qué mide el grado de un vértice?

A. Altura del grafo
B. Longitud de camino
C. Número de aristas incidentes
D. Número de ciclos

Pregunta 13: ¿Qué estructura de datos es adecuada para representar grafos dispersos?

A. Lista de adyacencia
B. Hash Table
C. Matriz de adyacencia
D. Árbol binario

Pregunta 14: ¿Qué ocurre si el grafo no es conexo?

A. No tiene aristas
B. Todos los vértices están unidos
C. Existen componentes separadas
D. No tiene ciclos

Pregunta 15: ¿Qué hace un algoritmo de alcanzabilidad?

A. Elimina ciclos
B. Ordena los vértices
C. Calcula grados
D. Verifica si hay un camino entre dos vértices

Pregunta 16: ¿Qué ocurre si aplicamos DFS en un grafo con ciclos?

A. Detecta componentes conexas
B. Siempre termina en O(1)
C. No se puede ejecutar
D. Puede entrar en bucle si no marcamos visitados

Pregunta 17: ¿Qué estructura usa la matriz de incidencia?

A. Lista enlazada
B. Heap
C. n×m booleana
D. n×n booleana

Pregunta 18: ¿Qué propiedad cumple un árbol como grafo?

A. Es completo
B. Tiene un ciclo
C. Tiene vértices aislados
D. Es conexo y sin ciclos

Pregunta 19: ¿Qué problema se puede resolver con BFS?

A. Detección de ciclos
B. Camino mínimo ponderado
C. Camino más corto en grafo no ponderado
D. Ordenamiento topológico

Pregunta 20: ¿Qué significa que un grafo sea bipartito?

A. Es completo
B. Tiene un ciclo
C. Es dirigido
D. Sus vértices pueden dividirse en dos conjuntos independientes

Respuestas Correctas

Pregunta 1: Una red de vértices conectados por aristas
Pregunta 2: O(n+m)
Pregunta 3: Cola
Pregunta 4: Pila (implícita en recursión)
Pregunta 5: Todo par de vértices está conectado por un camino
Pregunta 6: Conexiones entre pares de vértices
Pregunta 7: Las aristas tienen o no dirección
Pregunta 8: Un camino que empieza y termina en el mismo vértice
Pregunta 9: La secuencia de vértices del camino
Pregunta 10: O(n+m)
Pregunta 11: Todos los vértices conectados entre sí
Pregunta 12: Número de aristas incidentes
Pregunta 13: Lista de adyacencia
Pregunta 14: Existen componentes separadas
Pregunta 15: Verifica si hay un camino entre dos vértices
Pregunta 16: Puede entrar en bucle si no marcamos visitados
Pregunta 17: n×m booleana
Pregunta 18: Es conexo y sin ciclos
Pregunta 19: Camino más corto en grafo no ponderado
Pregunta 20: Sus vértices pueden dividirse en dos conjuntos independientes

Bonus Invariantes

Clase 01 – Introducción, Complejidad, Contratos e Invariantes

1. El tiempo de ejecución en el modelo RAM se mide en operaciones elementales.
2. Un contrato se cumple si la salida respeta la postcondición dada la precondición.
3. La notación O(f) representa el crecimiento asintótico superior de una función.
4. La invariante de un ciclo se mantiene antes y después de cada iteración.
5. Si un algoritmo preserva el invariante y termina, entonces el invariante vale al final.
6. La complejidad de selection sort en peor caso es $O(n^2)$.
7. En estructuras de datos, un invariante asegura coherencia entre operaciones.
8. El invariante de un contador modular es que siempre está entre 0 y k-1.
9. El invariante de una suma acumulada es que refleja todas las inserciones previas.
10. La invariante de una lista invertida es que cada posición contiene el elemento simétrico original.

Clase 02 – Tipos de datos secuenciales y Divide & Conquer

1. En una pila, el último en entrar es el primero en salir (LIFO).
2. En una cola, el primero en entrar es el primero en salir (FIFO).
3. En vectores dinámicos, el tamaño siempre es menor o igual a la capacidad.
4. En vectores con crecimiento geométrico, el costo amortizado de insertar es O(1).
5. En mergesort, en cada paso los subarreglos están ordenados antes de mezclarse.
6. La invariante de búsqueda binaria: siempre se cumple $A[i] \leq x < A[j]$.
7. En mergesort, cada división produce dos arreglos cuya unión da el original.
8. La complejidad de mergesort siempre respeta la recurrencia $T(n) = 2T(n/2)+O(n)$.
9. En pilas y colas, las operaciones push/add preservan el orden relativo de los elementos.
10. En el método divide & conquer, cada subproblema es instancia del problema original.

Clase 03 – Árboles Binarios y de Búsqueda

1. En un ABB, todos los nodos del subárbol izquierdo son menores que la raíz.
2. En un ABB, todos los nodos del subárbol derecho son mayores que la raíz.
3. En un ABB balanceado, la altura es O(log n).
4. El recorrido inorden de un ABB produce una lista ordenada.
5. En un AVL, cada nodo tiene factor de balanceo -1, 0 o 1.
6. En un árbol binario completo, el nivel i tiene 2^i nodos.
7. Eliminar el mínimo en un ABB deja otro ABB válido.
8. La altura de un árbol binario de n nodos es al menos log■(n+1).
9. Un árbol AVL garantiza altura logarítmica para operaciones.
10. La búsqueda en un ABB siempre recorre un camino de la raíz a una hoja.

Clase 04 – Backtracking

1. En backtracking, cada decisión parcial cumple con las restricciones hasta ese punto.
2. El retroceso garantiza explorar todas las soluciones posibles.
3. En el laberinto, una celda marcada como visitada no se vuelve a visitar.
4. En Sudoku, cada fila contiene dígitos sin repetición como invariante.
5. En Sudoku, cada columna mantiene unicidad de dígitos.
6. En Sudoku, cada subcuadrante tiene todos dígitos distintos.
7. En permutaciones, cada elemento aparece exactamente una vez.
8. En combinaciones, el orden de selección no importa.
9. En variaciones con repetición, cada posición puede tomar cualquier valor de A.
10. El método Minimax asume que ambos jugadores juegan óptimamente.

Clase 05 – Ordenamiento

1. Selection sort mantiene la sublista izquierda siempre ordenada.
2. Mergesort garantiza que cada sublista está ordenada antes de mezclarse.
3. En quicksort, tras el particionamiento todos los elementos a la izquierda son ≤ pivote.
4. En quicksort, tras el particionamiento todos los elementos a la derecha son ≥ pivote.
5. La invariante del bucle de bucket sort es que el conteo refleja apariciones reales.
6. La complejidad de todo algoritmo basado en comparaciones es $\Omega(n \log n)$.
7. Radix sort preserva el orden relativo gracias a la estabilidad en cada paso.
8. El invariante de un algoritmo estable es conservar el orden original de iguales.
9. En mergesort, cada paso combina listas ya ordenadas.
10. La altura del árbol de decisión de ordenamiento es al menos log■(n!).

Clase 06 – Colas de prioridad y Heaps

1. En un heap, el valor de cada nodo es mayor o igual al de sus hijos.
2. En un heap, el árbol es izquierdista (completo salvo el último nivel).
3. En un heap, insertar preserva el invariante tras ajuste hacia arriba.
4. En un heap, eliminar el máximo preserva el invariante tras ajuste hacia abajo.
5. La altura de un heap de n nodos es O(log n).
6. Heapify transforma un arreglo en heap en O(n).
7. Heapsort mantiene un heap válido en cada paso de extracción.
8. En representación por arreglo, hijo izquierdo está en 2i+1.
9. En representación por arreglo, hijo derecho está en 2i+2.
10. En representación por arreglo, el padre de i está en ■(i-1)/2■.

Clase 07 – Strings y Hashing

1. En un trie, cada camino raíz-nodo representa un prefijo válido.
2. En un trie, todas las hojas tienen clave asociada salvo la raíz.
3. La búsqueda en un trie recorre símbolos en orden de la clave.
4. Insertar en un trie preserva las claves previas.
5. Eliminar en un trie no borra nodos usados por otras claves.
6. En KMP, siempre se cumple que el prefijo más largo coincide con un sufijo explorado.
7. La tabla de fallos de KMP refleja los prefijos propios de la palabra.
8. En hashing, el valor se busca en el bucket h(k).
9. En direccionamiento abierto, siempre se encuentra un espacio si no está lleno.
10. En direccionamiento cerrado, cada bucket mantiene una lista de claves.

Clase 08 – Grafos

1. En una matriz de adyacencia, A[i][j]=True $\Leftrightarrow$ {vi,vj}$\in$ E.
2. En lista de adyacencia, cada vecino aparece exactamente una vez.
3. En un grafo no dirigido, d(v) = número de aristas incidentes a v.
4. La suma de grados de todos los vértices es 2m.
5. Un DFS nunca visita dos veces el mismo nodo.
6. Un BFS visita vértices en orden creciente de distancia desde la fuente.
7. En DFS, al retroceder todos los vecinos de un nodo ya están visitados.
8. En BFS, cada nodo se encola como mucho una vez.
9. Un grafo conexo tiene camino entre todo par de nodos.
10. Un ciclo es un camino que comienza y termina en el mismo nodo.

Clase 09 – Programación Dinámica

1. Un problema de DP se descompone en subproblemas solapados.
2. Cada subproblema debe resolverse al menos una vez y guardarse.
3. La tabla de DP preserva la solución óptima en cada celda.
4. En mochila, la invariante es que el valor almacenado es el máximo posible con esa capacidad.
5. En caminos mínimos en grillas, la celda (i,j) contiene el costo mínimo hasta allí.
6. El principio de optimalidad asegura que la solución global incluye soluciones óptimas parciales.
7. En DP bottom-up, cada valor calculado depende solo de anteriores ya resueltos.
8. En DP top-down, los subproblemas se memorizan tras resolverse.
9. En Fibonacci DP, cada posición contiene la suma de las dos previas.
10. La complejidad de DP es el producto del número de estados por el costo de transición.