Estructuras de Datos y Algoritmos II

Rodrigo Moreno

Facultad de Ingeniería – UNAM

Índice general

Parte I

Ordenamiento

Introducción a los algoritmos de ordenamiento

El problema de ordenamiento es uno de los más fundamentales en ciencias de la computación. Dados n elementos con una relación de orden definida (por ejemplo, números enteros, reales o cadenas de texto), el objetivo es reorganizarlos en una secuencia no decreciente:

$$a_1 \leq a_2 \leq \cdots \leq a_n$$

o en orden decreciente, según se requiera.

Importancia del ordenamiento

Ordenar no es solo una operación básica, sino una necesidad en múltiples contextos:

Problema de Ordenamiento

Dados n elementos con una relación de orden definida (por ejemplo, números enteros, reales o cadenas de texto), reorganizar los elementos en una secuencia no decreciente:

$$a_1 \leq a_2 \leq \cdots \leq a_n$$

o en orden decreciente, según se requiera.
Entrada: una secuencia de n elementos $\{a_1, a_2, \ldots, a_n\}$.
Salida: una permutación ordenada de dicha secuencia.
Restricciones: los elementos pueden o no ser repetidos; pueden ser comparables o tener claves.

Importancia del ordenamiento

Ordenar no es solo una operación básica, sino una necesidad en múltiples contextos:

- Optimización de búsquedas: muchos algoritmos de búsqueda asumen datos ordenados.
- Agrupamiento y clasificación eficiente de datos.
- Preprocesamiento para otros algoritmos (e.g. Kruskal, radix join en bases de datos).
- Comprensión de técnicas algorítmicas como recursión, divide y vencerás, estructuras de datos, etc.

Tipos de algoritmos de ordenamiento

Existen muchas estrategias para ordenar datos. En este curso estudiaremos seis algoritmos representativos, que nos permitirán contrastar estilos, paradigmas y análisis:

- Algoritmos **comparativos** como Bubble Sort, QuickSort, Merge Sort y Heapsort, que se basan exclusivamente en comparaciones binarias entre elementos.
- Algoritmos **no comparativos** como Counting Sort y Radix Sort, que aprovechan propiedades específicas del dominio de los datos.

Criterios de evaluación de un algoritmo de ordenamiento

A lo largo del curso, cada algoritmo será evaluado con base en los siguientes aspectos:

- **Correctitud:** produce una permutación ordenada de la entrada.
- **Complejidad temporal:** en el peor, mejor y caso promedio.
- **Complejidad espacial:** si requiere memoria adicional o es in-place.
- **Estabilidad:** conserva el orden relativo de elementos iguales.
- **Adaptabilidad:** mejora si la entrada está parcialmente ordenada.

Preludio al análisis

Más adelante se demostrará que cualquier algoritmo de ordenamiento basado únicamente en comparaciones requiere al menos $\Omega(n \log n)$ comparaciones en el peor caso. Esta cota inferior nos permite entender los límites fundamentales del problema y motiva la búsqueda de métodos alternativos cuando se conocen propiedades adicionales de los datos.

1. Bubble Sort (Ordenamiento Burbuja)

Problema de Ordenamiento

Dada una secuencia de n elementos $\{a_1, a_2, \ldots, a_n\}$ con una relación de orden definida, reordenarla de forma no decreciente, es decir, tal que:

$$a_1 \leq a_2 \leq \cdots \leq a_n$$

Entrada: Secuencia de n elementos.
Salida: Permutación ordenada de la secuencia.
Restricciones: Se permite comparar dos elementos entre sí y, si es necesario, intercambiarlos.

Descripción del algoritmo

Bubble Sort compara pares consecutivos de elementos y los intercambia si están en el orden incorrecto. Este proceso se repite *n* veces, ”burbujeando” los elementos más grandes hacia el final de la lista.

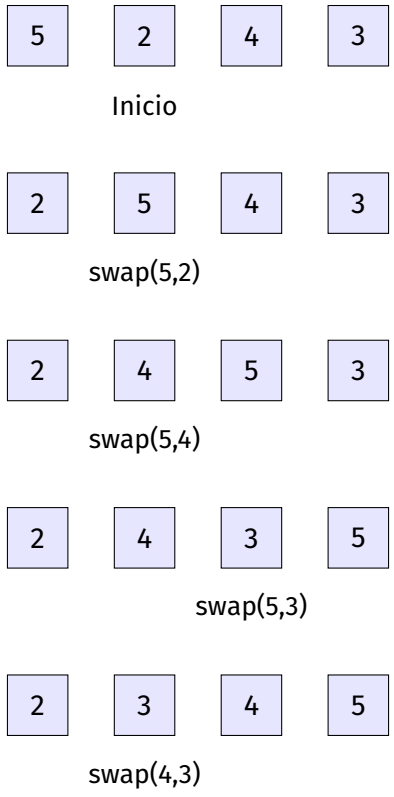
Pseudocódigo

Bubble Sort

```
BUBBLE-SORT(A)                          // Ordena el arreglo A in-place
n ← longitud(A)                         // Obtener el tamaño del arreglo
para i de 0 hasta n - 2 hacer           // Pasadas desde el inicio
  para j de 0 hasta n - i - 2 hacer     // Comparar pares adyacentes
    si A[j] > A[j + 1] entonces         // Si están en orden incorrecto
      intercambiar A[j] y A[j + 1]      // Swap para ordenarlos
```

Ejemplo visual

Aplicamos Bubble Sort al arreglo inicial: [5, 2, 4, 3]. En cada paso, resaltamos las comparaciones y los intercambios realizados:



Observación: En este ejemplo se realizan 4 pasos con intercambios visibles. El algoritmo termina cuando no se requieren más swaps en una pasada completa.

Nota: La mejora típica consiste en interrumpir si no hubo intercambios.

Análisis de complejidad

- **Peor caso:** $O(n^2)$ Esto ocurre cuando el arreglo está completamente en orden inverso. En ese caso, en cada pasada se hace la **máxima cantidad de intercambios** posibles. La primera pasada hace $(n - 1)$ comparaciones, la segunda $(n - 2)$, y así sucesivamente, hasta 1. Por tanto, el número total de comparaciones es:

$$(n - 1) + (n - 2) + \cdots + 1 = \frac{n(n - 1)}{2} = O(n^2)$$

- **Mejor caso:** $O(n)$ Cuando el arreglo ya está ordenado, no se hace ningún intercambio. Si se implementa la versión optimizada con una bandera (‘flag‘) para detectar que no hubo swaps, entonces el algoritmo termina después de una sola pasada con $n-1$ comparaciones.
- **Caso promedio:** $O(n^2)$ En general, el arreglo estará parcialmente desordenado. Aunque no tan malo como el peor caso, se siguen haciendo muchas comparaciones y swaps. El número esperado de intercambios también es proporcional a n^2 porque en promedio cada elemento podría estar ”desplazado” múltiples posiciones hacia la izquierda.
- **Espacio adicional:** $O(1)$ Bubble Sort trabaja directamente sobre el arreglo original, sin requerir espacio extra significativo. Solo usa variables temporales para intercambiar elementos.
- **Estabilidad:** Sí Bubble Sort **es estable**, porque cuando compara elementos iguales, no los intercambia. Esto preserva el orden relativo de elementos con claves duplicadas.
- **Adaptabilidad:** Parcialmente adaptable Aunque el algoritmo básico no es adaptable, si se le agrega la optimización del ”flag de intercambio”, se vuelve **más eficiente para entradas parcialmente ordenadas**, evitando pasadas innecesarias.

Comentarios pedagógicos

Bubble Sort no es eficiente para listas largas, pero es muy útil como herramienta didáctica para introducir:

- Análisis de complejidad.
- Estabilidad de algoritmos.
- Optimización por casos especiales.

Se recomienda enseñar este algoritmo solo como base para contrastar con algoritmos más eficientes.

2. Heapsort

Problema de Ordenamiento

Dada una secuencia de n elementos $\{a_1, a_2, \dots, a_n\}$ con una relación de orden total, reorganizarla en orden no decreciente.

$$a_1 \leq a_2 \leq \cdots \leq a_n$$

Entrada: Secuencia de n elementos.
Salida: Permutación ordenada de la secuencia.
Restricciones: Se permite comparar e intercambiar elementos.

Descripción del algoritmo

Heapsort se basa en construir una estructura de datos llamada **heap máximo**, que garantiza que el elemento más grande esté en la raíz. Luego se extrae la raíz y se restablece la propiedad de heap. Este proceso se repite n veces para obtener el arreglo ordenado.

Pseudocódigo

Heapsort

```
HEAPSORT(A)                              // Ordena el arreglo A
BUILD-MAX-HEAP(A)                        // Construye el heap máximo
n ← longitud(A)
para i de n - 1 hasta 1 hacer
  intercambiar A[0] y A[i]              // Mueve el máximo al final
  n ← n - 1                              // Reduce el tamaño del heap
  MAX-HEAPIFY(A, 0)                      // Restablece propiedad de heap
```

BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
n ← longitud(A)
para i de n/2 hasta 0 hacer
  MAX-HEAPIFY(A, i)
```

MAX-HEAPIFY

```
MAX-HEAPIFY(A, i)
l ← hijo izquierdo(i)
r ← hijo derecho(i)
```

```
mayor ← i
si l < n y A[l] > A[mayor] entonces
  mayor ← l
si r < n y A[r] > A[mayor] entonces
  mayor ← r
si mayor ≠ i entonces
  intercambiar A[i] y A[mayor]
  MAX-HEAPIFY(A, mayor)
```

Estructura de datos: Max-Heap

Un **heap** es un árbol binario completo almacenado como arreglo, donde se cumple que:

$$A[\text{padre}(i)] \geq A[i]$$

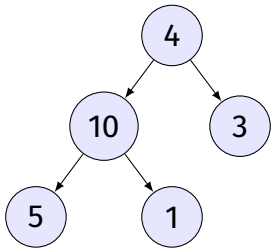
Se accede así:

- Padre de i: $\left\lfloor \frac{i-1}{2} \right\rfloor$
- Hijo izquierdo de i: $2i + 1$
- Hijo derecho de i: $2i + 2$

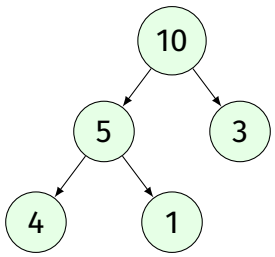
Ejemplo visual

Heapsort sobre el arreglo inicial $[4, 10, 3, 5, 1]$.

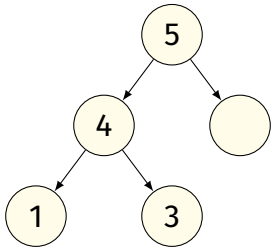
1. Representación inicial como árbol binario completo



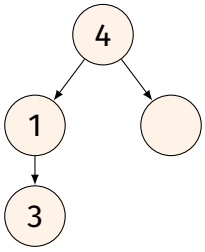
2. Después de BUILD-MAX-HEAP



3. Extracción 1: swap(10, 1), heapify



4. Extracción 2: swap(5, 3), heapify



Nota: Se repite el proceso hasta que el heap queda vacío y el arreglo completamente ordenado.

Análisis de complejidad

(por qué? revisa el apéndice A :))

- *Tiempo peor caso:* $O(n \log n)$ Cada extracción cuesta $O(\log n)$, y se hacen n extracciones.
- *Tiempo mejor caso:* $O(n \log n)$ La estructura de heap siempre requiere $O(\log n)$ por operación.
- *Tiempo promedio:* $O(n \log n)$ Independientemente del arreglo de entrada, el tiempo se mantiene consistente.
- *Espacio adicional:* $O(1)$ Todo se hace sobre el mismo arreglo (in-place).
- *Estabilidad:* No Intercambios pueden reordenar elementos iguales.

Comentarios pedagógicos

Heapsort es un algoritmo eficiente y confiable. Aunque no es estable ni adaptable, tiene ventajas como no requerir espacio adicional y garantizar $O(n \log n)$ siempre. Es excelente para contrastar con QuickSort, que en promedio es más rápido pero no garantiza lo mismo.

3. Quick Sort

Idea

Quick Sort es un algoritmo *divide y vencerás*: elige un pivote, *particiona* el arreglo en elementos ≤ pivote y > pivote, y luego ordena recursivamente ambas partes. Su fortaleza radica en una excelente locality de memoria y en que, en promedio, realiza $O(n \log n)$ comparaciones.

Pseudocódigo (Lomuto Partition)

Pseudocódigo

```
QUICKSORT(A, lo, hi):
    if lo < hi:
        p <- PARTITION(A, lo, hi)      # usa A[hi] como pivote
        QUICKSORT(A, lo, p-1)
        QUICKSORT(A, p+1, hi)

PARTITION(A, lo, hi):
    pivot <- A[hi]
    i <- lo - 1
    for j <- lo to hi - 1:
        if A[j] <= pivot:
            i <- i + 1
            swap A[i], A[j]
    swap A[i+1], A[hi]
    return i + 1
```

Corrección (bosquejo)

Sea p el índice retornado por PARTITION. Por construcción:

- Para todo $k \in [\mathtt{lo}, p-1]$, $A[k] \le A[p]$.
- Para todo $k \in [p+1, \mathtt{hi}]$, $A[k] > A[p]$.

La recursión ordena ambos subarreglos disjuntos y la concatenación preserva el orden total.

Complejidad y propiedades

Complejidad

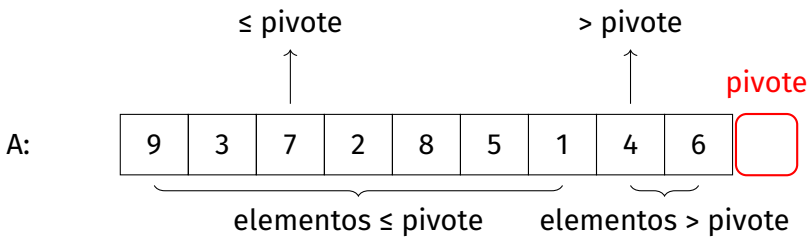
$$T(n) = T(i) + T(n - i - 1) + \Theta(n)$$

donde i es el tamaño del subarreglo izquierdo tras la partición.

- **Peor caso:** $O(n^2)$ (pivote sistemáticamente el mínimo o máximo).
- **Caso promedio / mejor:** $O(n \log n)$.
- **Espacio:** $O(\log n)$ esperado por la pila (con particiones balanceadas).
- **Estabilidad:** *No estable.*
- **In-place:** Sí (ignorando la pila de recursión).

Buenas prácticas: escoger pivote aleatorio o *median-of-three* y conmutar a Insertion Sort para subarreglos pequeños.

Ejemplo visual de partición (TikZ)



Implementación en Python

```python
import random

def partition_lomuto(a, lo, hi):
    pivot = a[hi]
    i = lo - 1
    for j in range(lo, hi):
        if a[j] <= pivot:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[i+1], a[hi] = a[hi], a[i+1]
```

```
    return i + 1

def quicksort(a, lo=None, hi=None, random_pivot=True, cutoff=10):
    if lo is None or hi is None:
        lo, hi = 0, len(a) - 1
    while lo < hi:
        # cutoff a insertion sort
        if hi - lo + 1 <= cutoff:
            insertion_sort(a, lo, hi)
            return
        if random_pivot:
            p = random.randint(lo, hi)
            a[p], a[hi] = a[hi], a[p]
        p = partition_lomuto(a, lo, hi)
        # Tail recursion elimination: resolver primero el subarreglo menor
        if p - lo < hi - p:
            quicksort(a, lo, p - 1, random_pivot, cutoff)
            lo = p + 1
        else:
            quicksort(a, p + 1, hi, random_pivot, cutoff)
            hi = p - 1

def insertion_sort(a, lo, hi):
    for i in range(lo+1, hi+1):
        key = a[i]
        j = i - 1
        while j >= lo and a[j] > key:
            a[j+1] = a[j]
            j -= 1
        a[j+1] = key
```

Listing 1: Quick Sort (Lomuto) con pivote aleatorio opcional

Pruebas unitarias (pytest)

```
import random
from math import isclose

def is_sorted(a):
    return all(a[i] <= a[i+1] for i in range(len(a)-1))

def test_quicksort_basic():
    arr = [3,1,4,1,5,9,2,6,5,3,5]
    quicksort(arr)
    assert is_sorted(arr)
```

```python

def test_quicksort_duplicates():
    arr = [5] * 100
    quicksort(arr)
    assert is_sorted(arr) and len(arr) == 100

def test_quicksort_random_large():
    arr = [random.randint(-10**6, 10**6) for _ in range(10_000)]
    quicksort(arr)
    assert is_sorted(arr)

def test_quicksort_cutoff_effect():
    arr = [7,6,5,4,3,2,1]
    quicksort(arr, cutoff=3)
    assert is_sorted(arr)
```

Listing 2: Tests para Quick Sort

Experimentos sugeridos

- Comparar tiempos de ejecución con y sin `random_pivot`, y distintos `cutoff` hacia Insertion Sort.
- Medir profundidad de recursión máxima vs. tamaño de entrada aleatoria.

4. Counting Sort

Idea

Counting Sort ordena claves enteras en un rango acotado $\{0, \dots, k\}$ contando ocurrencias. Es *lineal* en $n + k$ y puede ser *estable* si se usa acumulación de conteos y escritura desde el final. No es por comparación y requiere memoria adicional proporcional a k.

Pseudocódigo (estable)

Pseudocódigo

```
COUNTING-SORT(A, k):
    # A contiene enteros en [0..k]
    C[0..k] <- 0
    for x in A:
        C[x] <- C[x] + 1
    # acumulados
    for i <- 1 to k:
        C[i] <- C[i] + C[i-1]
    B[1..n]
    for j <- n downto 1:
        x <- A[j]
        B[C[x]] <- x
        C[x] <- C[x] - 1
    return B
```

Corrección (bosquejo)

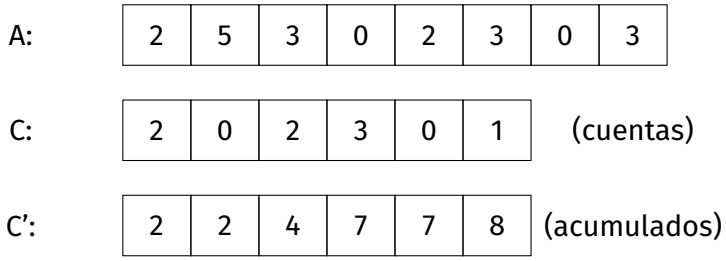
Tras el primer bucle, $C[i]$ almacena el número de claves iguales a i. Luego, los acumulados hacen que $C[i]$ sea la *posición final* del último i en B. Al recorrer A en orden inverso, se rellena establemente B.

Complejidad y propiedades

Complejidad

- **Tiempo:** $O(n + k)$.
- **Espacio:** $O(n + k)$ (arreglo de salida B y conteos C).
- **Estabilidad:** Sí (con pasada inversa).
- **Requiere:** claves enteras en rango pequeño/moderado; para claves fuera de rango, mapear/normalizar.

Ilustración (TikZ): conteo y acumulados



Implementación en Python (estable)

```python
def counting_sort(a, k):
    n = len(a)
    c = [0]*(k+1)
    for x in a:
        if x < 0 or x > k:
            raise ValueError("Clave fuera de rango [0..k]")
        c[x] += 1
    for i in range(1, k+1):
        c[i] += c[i-1]
    b = [None]*n
    for j in range(n-1, -1, -1):
        x = a[j]
        b[c[x]-1] = x
        c[x] -= 1
    return b
```

Listing 3:)]Counting Sort estable (claves en [0..k])

Pruebas unitarias (pytest)

```python
import random

def test_counting_sort_basic():
    arr = [2,5,3,0,2,3,0,3]
    out = counting_sort(arr, k=5)
    assert out == sorted(arr)

def test_counting_sort_stable_buckets():
    # pares (clave, id) -> validamos estabilidad emulando IDs por bucket
    keys = [1,0,1,2,1,0]
    out = counting_sort(keys, k=2)
    assert out == sorted(keys)
```

```python

def test_counting_sort_large_random():
    k = 100
    arr = [random.randint(0,k) for _ in range(10000)]
    out = counting_sort(arr, k=k)
    assert out == sorted(arr)

def test_counting_sort_out_of_range():
    try:
        counting_sort([0,1,2,100], k=50)
        assert False, "Debió lanzar excepción"
    except ValueError:
        assert True
```

Listing 4: Tests para Counting Sort

Experimentos sugeridos

- Medir tiempo al variar n con k fijo (p.ej., $k = 10^3$) vs. variar k con n fijo.
- Comparar con Quick Sort en entradas con rango pequeño (Counting) vs. rango grande (Quick).

5. Radix Sort (LSD)

Idea

Radix Sort (LSD) ordena enteros *no negativos* procesando los dígitos de **menor** a **mayor** peso (Least Significant Digit primero). En cada pasada aplica un *algoritmo estable* por dígito (típicamente **Counting Sort**). Si los dígitos están en una base b y el máximo tiene d dígitos, el costo es $O(d\,(n + b))$.

Supuestos y notas

- Las claves son **enteros no negativos**. Para negativos, mapear/normalizar primero.
- Elegir la base b (p. ej., 10 o 2^r). Mayores b reducen d pero aumentan memoria C.
- Es **no comparativo**; requiere un **estabilizador** por dígito (Counting Sort).

Pseudocódigo (LSD con Counting Sort estable por dígito)

Pseudocódigo

```
RADIX-SORT-LSD(A, base):
    # A: arreglo de enteros >= 0
    if A está vacío: return A
    d <- número de dígitos del máximo en base 'base'
    for pos <- 0 to d-1:
        A <- COUNTING-SORT-BY-DIGIT(A, base, pos)  # estable
    return A

COUNTING-SORT-BY-DIGIT(A, base, pos):
    # pos = 0 (unidades), 1 (decenas), ...
    C[0..base-1] <- 0
    for x in A:
        d <- DIGITO(x, base, pos)         # (x // base^pos) % base
        C[d] <- C[d] + 1
    for i <- 1 to base-1:                  # acumulados
        C[i] <- C[i] + C[i-1]
    B[0..n-1]
    for j <- n-1 downto 0:                # pasada inversa = estable
        d <- DIGITO(A[j], base, pos)
        C[d] <- C[d] - 1
        B[C[d]] <- A[j]
    return B
```

Corrección (bosquejo)

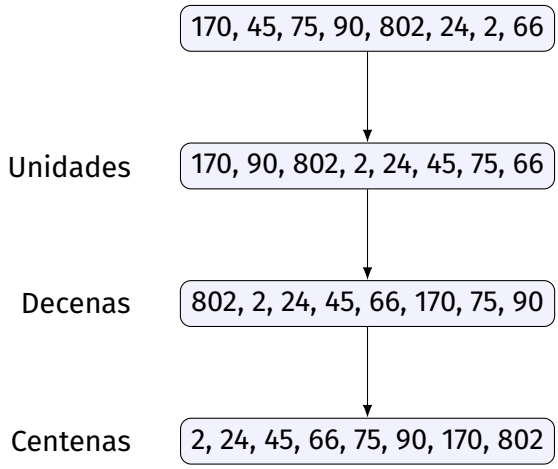
Cada pasada por dígito reordena *establemente* según ese dígito. La estabilidad garantiza que el orden relativo inducido por posiciones menos significativas se preserva cuando consideramos dígitos más significativos. Tras d pasadas, los elementos quedan ordenados.

Complejidad y propiedades

Complejidad

- **Tiempo:** $O(d\,(n+b))$ donde d = # dígitos del máximo y b = base.
- **Espacio:** $O(n+b)$ (arreglo auxiliar B y conteos C).
- **Estabilidad:** Sí (al usar Counting Sort estable por dígito).
- **Elección de base:** $b = 10$ es simple; $b = 2^r$ (p. ej. 256) suele ser eficiente.

Ilustración (TikZ): pipeline de dígitos y buckets



Implementación en Python (LSD, base configurable)

```python
def _digit(x, base, pos):
    # extrae el dígito en 'pos' (0=unidades) en base 'base'
    return (x // (base ** pos)) % base

def counting_sort_by_digit(a, base, pos):
    n = len(a)
    c = [0] * base
    for x in a:
        if x < 0:
            raise ValueError("Radix LSD aquí asume enteros no negativos")
        c[_digit(x, base, pos)] += 1
    # acumulados
    for i in range(1, base):
        c[i] += c[i-1]
    # salida estable
    b = [None] * n
```

```
    for j in range(n-1, -1, -1):
        d = _digit(a[j], base, pos)
        c[d] -= 1
        b[c[d]] = a[j]
    return b

def radix_sort_lsd(a, base=10, max_digits=None):
    if not a:
        return []
    if base < 2:
        raise ValueError("base debe ser >= 2")
    maxv = max(a)
    if maxv < 0:
        raise ValueError("Radix LSD asume enteros no negativos")
    d = max_digits if max_digits is not None else (0 if maxv == 0 else len(
       str(int(maxv),)))
    # cálculo robusto de dígitos si base != 10
    if base != 10 and max_digits is None:
        # computa d = floor(log_base(maxv)) + 1
        p = 0
        t = 1
        while t <= maxv:
            t *= base
            p += 1
        d = max(1, p)
    out = list(a)
    for pos in range(d):
        out = counting_sort_by_digit(out, base, pos)
    return out
```

Listing 5: Radix Sort (LSD) con Counting Sort estable por dígito

Pruebas unitarias (pytest)

```
import random

def test_radix_lsd_basic():
    arr = [170, 45, 75, 90, 802, 24, 2, 66]
    out = radix_sort_lsd(arr, base=10)
    assert out == sorted(arr)

def test_radix_lsd_many_random_base10():
    arr = [random.randint(0, 10**6) for _ in range(5000)]
    out = radix_sort_lsd(arr, base=10)
    assert out == sorted(arr)
```

```python

def test_radix_lsd_power_of_two_base():
    # base 256 (2^8) suele ser eficiente
    arr = [random.randint(0, 10**6) for _ in range(5000)]
    out = radix_sort_lsd(arr, base=256)
    assert out == sorted(arr)

def test_radix_lsd_zero_and_single_digit():
    assert radix_sort_lsd([0], base=10) == [0]
    assert radix_sort_lsd([9,0,5,1], base=10) == [0,1,5,9]

def test_radix_lsd_stability_property():
    # Para estabilidad por dígito, probamos que empatar por dígitos menos
    # significativos no rompe el orden inducido en pasadas previas.
    arr = [21, 11, 31, 12, 22, 32]  # mismo dígito unidades por grupos
    out = radix_sort_lsd(arr, base=10)
    assert out == sorted(arr)

def test_radix_lsd_reject_negative():
    try:
        radix_sort_lsd([1, -2, 3], base=10)
        assert False, "Debió lanzar excepción para negativos"
    except ValueError:
        assert True
```

Listing 6: Tests para Radix Sort (LSD)

Experimentos sugeridos

- Variar $b \in \{10, 2^4, 2^8\}$ y medir tiempos vs. n y el número de dígitos d.
- Comparar Radix (LSD) con Quicksort cuando los datos son de *rango muy amplio*: Radix gana si d es relativamente pequeño; Quicksort puede rendir mejor cuando d es grande.

6. Merge Sort

Problema de Ordenamiento

Dada una secuencia de n elementos $\{a_1, a_2, \dots, a_n\}$ con una relación de orden definida, reordenarla de forma no decreciente, es decir, tal que:

$$a_1 \leq a_2 \leq \cdots \leq a_n$$

Entrada: Secuencia de n elementos.
Salida: Permutación ordenada de la secuencia.
Restricciones: Se permite dividir el arreglo en subarreglos y luego fusionarlos de manera ordenada.

Descripción del algoritmo

Merge Sort es un algoritmo basado en la técnica **Divide y Vencerás**:

1. Dividir el arreglo en dos mitades aproximadamente iguales.
2. Ordenar recursivamente cada mitad.
3. Combinar (merge) las dos mitades ordenadas en un único arreglo ordenado.

Observación: La fusión se realiza en tiempo lineal respecto a la suma de tamaños de los subarreglos.

Pseudocódigo

Merge Sort

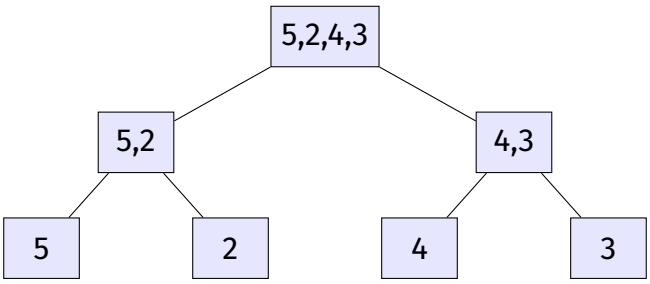
```
MERGE-SORT(A, l, r)                  // Ordena A[l..r]
  si l < r entonces
    m ← [(l + r)/2]_                 // índice medio
    MERGE-SORT(A, l, m)               // ordenar mitad izquierda
    MERGE-SORT(A, m+1, r)             // ordenar mitad derecha
    MERGE(A, l, m, r)                 // fusionar resultados

MERGE(A, l, m, r)                     // fusiona dos mitades ordenadas
  n1 ← m - l + 1
  n2 ← r - m
  crear arrays L[1..n1], R[1..n2]
  copiar A[l..m] en L y A[m+1..r] en R
  i ← 1, j ← 1, k ← l
  mientras i ≤ n1 y j ≤ n2 hacer
```

```
    si L[i] ≤ R[j] entonces
      A[k] ← L[i]; i ← i + 1
    sino
      A[k] ← R[j]; j ← j + 1
    k ← k + 1
  copiar elementos restantes de L y R en A
```

Ejemplo visual

Aplicamos Merge Sort al arreglo inicial: $[5, 2, 4, 3]$.



Explicación:

- El arreglo se divide en mitades hasta llegar a subarreglos de un solo elemento.
- Luego se realiza la fusión de abajo hacia arriba: $\{5\}$ y $\{2\} \to \{2, 5\}$, $\{4\}$ y $\{3\} \to \{3, 4\}$, finalmente $\{2, 5\}$ y $\{3, 4\} \to \{2, 3, 4, 5\}$.

Tipo de recorrido (traversing)

El árbol de recursión de Merge Sort es un **árbol binario completo** de profundidad $\log n$.

- El algoritmo sigue un **DFS** (Depth First Search).
- Específicamente, un recorrido **preorden en la fase de división** (primero procesa el nodo actual dividiendo, luego va a la izquierda y después a la derecha).
- La **fusión** ocurre en el regreso de la recursión, lo cual puede verse como un recorrido **postorden** (procesar hijos primero y después el nodo).

Análisis de complejidad

- **División:** $O(\log n)$ niveles en el árbol de recursión.
- **Fusión:** Cada nivel procesa n elementos en total.

- **Complejidad total:** $O(n \log n)$ en peor, mejor y promedio caso.
- **Espacio adicional:** $O(n)$ (arrays auxiliares para la fusión).
- **Estabilidad:** Sí, ya que no altera el orden relativo de elementos iguales.

Comentarios pedagógicos

Merge Sort es un excelente ejemplo de **Divide y Vencerás**, útil para explicar:

- Árbol de recursión y sus recorridos.
- Diferencia entre dividir y combinar en algoritmos recursivos.
- Comparación de eficiencia contra algoritmos $O(n^2)$ como Bubble Sort.

Apéndices

Apéndice A: Altura de un heap binario y su complejidad

Objetivo: Justificar por qué la operación `MAX-HEAPIFY` tiene complejidad $O(\log n)$ en un heap binario con n elementos.

1. Estructura de un heap binario

Un **heap binario** es un árbol binario completo donde cada nivel está completamente lleno, excepto posiblemente el último.

Cada nodo cumple la propiedad de heap máximo:

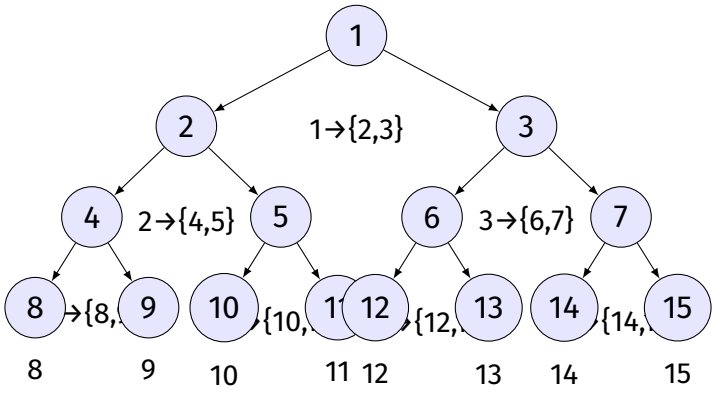
$$\text{Para todo nodo } i: \quad A[i] \geq A[\text{hijo izquierdo}], \quad A[i] \geq A[\text{hijo derecho}]$$

El heap se representa comúnmente como un arreglo donde: - i: posición actual. - $2i + 1$: hijo izquierdo. - $2i + 2$: hijo derecho.

2. Altura de un árbol binario completo

Definición: La *altura* de un árbol es el número de aristas en el camino más largo desde la raíz hasta una hoja.

Árbol binario completo con 15 nodos y sus hijos indicados



Número total de nodos:

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \quad \Rightarrow \quad h = \lfloor \log_2(n+1) \rfloor - 1$$

3. Relación formal entre n y la altura h

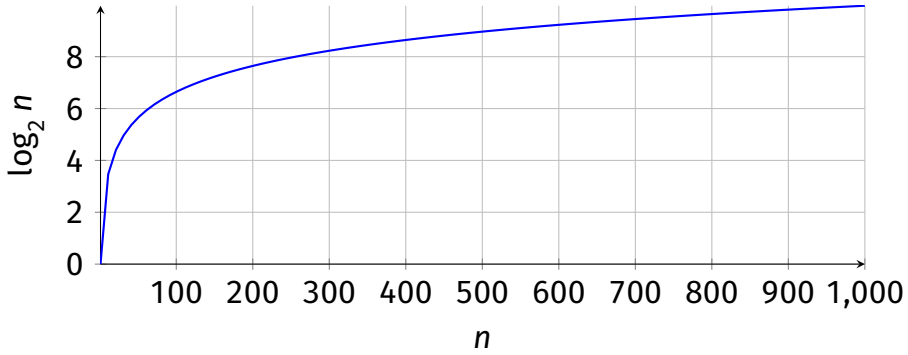
Sea h la altura del heap. Sabemos que:

$$2^h \leq n \leq 2^{h+1} - 1$$

$$\Rightarrow h \leq \log_2 n \leq h + 1 \quad \Rightarrow \quad h = \lfloor \log_2 n \rfloor$$

Por lo tanto, la altura de un heap binario con n elementos crece en:

$$O(\log n)$$



Crecimiento logarítmico de la altura del heap conforme aumenta el número de elementos.

4. ¿Qué hace `MAX-HEAPIFY`?

La operación `MAX-HEAPIFY(A, i)` realiza:

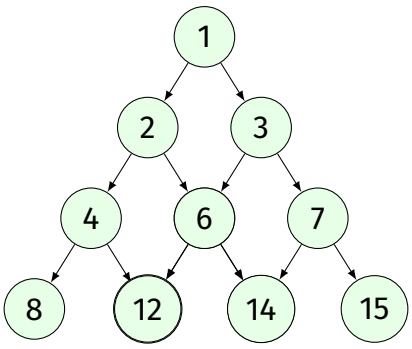
1. Comparación del nodo i con sus hijos.
2. Intercambio si uno de los hijos es mayor.
3. Llamada recursiva en la subrama afectada.

Peor caso: recorrer desde la raíz hasta una hoja — eso implica h pasos.

Ejemplo:

$$A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$$

Ejemplo: Árbol completo de altura 3 con 15 nodos



Número total de nodos:

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \quad \Rightarrow \quad h = \lfloor \log_2(n+1) \rfloor - 1$$

3. Relación formal entre n y la altura h

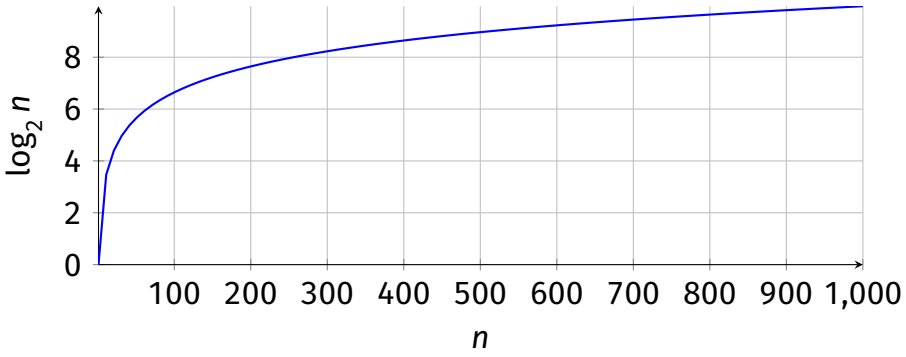
Sea h la altura del heap. Sabemos que:

$$2^h \leq n \leq 2^{h+1} - 1$$

$$\Rightarrow h \leq \log_2 n \leq h + 1 \quad \Rightarrow \quad h = \lfloor \log_2 n \rfloor$$

Por lo tanto, la altura de un heap binario con n elementos crece en:

$$O(\log n)$$



Crecimiento logarítmico de la altura del heap conforme aumenta el número de elementos.

4. ¿Qué hace `MAX-HEAPIFY`?

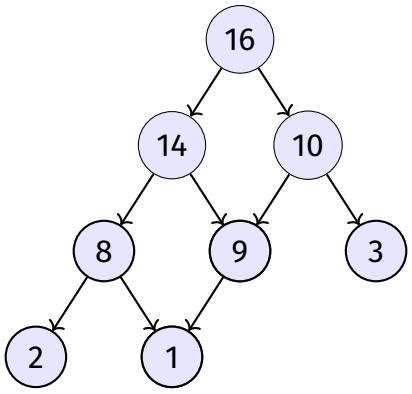
La operación `MAX-HEAPIFY(A, i)` realiza:

1. Comparación del nodo *i* con sus hijos.
2. Intercambio si uno de los hijos es mayor.
3. Llamada recursiva en la subrama afectada.

Peor caso: recorrer desde la raíz hasta una hoja — eso implica *h* pasos.

Ejemplo:

$A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$



Este es un *heap binario máximo* con $n = 10$ elementos.

La altura del heap es:

$$\lfloor \log_2 10 \rfloor = 3$$

Por lo tanto, una operación como `MAX-HEAPIFY` puede realizar hasta **3 intercambios recursivos** en el peor caso (cuando el nodo se propaga hasta una hoja).

—

5. Conclusión

Como la altura de un heap binario es $\lfloor \log_2 n \rfloor$, cualquier operación que recorra desde la raíz hacia una hoja —como `heapify`, `insert` o `extract-max`— tiene una complejidad temporal de:

$$\boxed{O(\log n)}$$

Este comportamiento garantiza que el algoritmo **Heapsort** mantenga una eficiencia de:

$$\boxed{O(n \log n)}$$

en el *mejor*, *peor* y *caso promedio*, ya que aplica `MAX-HEAPIFY` $O(n)$ veces.

A. Los básicos de GitHub

Panorama del curso y resultados de aprendizaje

El objetivo de este curso es darte una introducción breve a GitHub. También te daremos materiales para seguir aprendiendo y algunas ideas para empezar a usar la plataforma.

Git y GitHub

Git es un Sistema de Control de Versiones (VCS) distribuido: sirve para rastrear cambios, colaborar y compartir. Con Git puedes llevar un historial de lo que has hecho y volver a una versión anterior cuando lo necesites. Facilita el trabajo en equipo: varias personas pueden trabajar en el mismo proyecto y fusionar sus cambios en una fuente final.

GitHub lleva el poder de Git a la web con una interfaz sencilla. Se usa en toda la industria de software y más allá para colaborar y mantener el historial de los proyectos. Este curso comienza con lo básico de GitHub; más adelante profundizaremos.

Entendiendo el *GitHub Flow*

El *GitHub flow* es un flujo de trabajo ligero que te permite experimentar y colaborar sin riesgo de perder tu trabajo previo.

Repositorios Un repositorio es donde vive tu proyecto (piénsalo como una carpeta del proyecto). Contiene los archivos y el historial de revisiones. Puedes trabajar solo o invitar colaboradores.

Clonación (*Cloning*) Cuando creas un repositorio en GitHub, se guarda de forma remota en la nube. Al *clonar* obtienes una copia local en tu computadora y luego usas Git para sincronizar ambos lados. Esto facilita corregir problemas, agregar o eliminar archivos y hacer *commits* más grandes usando tu editor favorito. Al clonar, también traes todas las versiones previas de cada archivo y carpeta.

Commit* y *Push Los *commits* son puntos de control con un mensaje que explica qué cambiaste (por ejemplo: “Agrega README con información del proyecto”). Cuando estás listo para compartir tus cambios con el repositorio remoto en GitHub, ejecutas `git push`.

Términos clave de GitHub

Repositorios y README: Un archivo `README.md` explica por qué tu proyecto es útil, cómo usarlo y cómo contribuir.

Ramas (*branches*): Permiten trabajar en paralelo sin afectar la rama principal `main`.

***Forks*:** Copia de un repositorio ajeno para contribuir sin afectar el original.

***Pull Requests* (PR):** Propuesta de cambios para ser revisados antes de fusionar.

Issues: Seguimiento de tareas, errores o mejoras.

Perfil de usuario: Muestra tu historial y contribuciones.

Markdown: Lenguaje de marcado para dar formato en GitHub.

Comunidad: *Starring* y seguir usuarios para recibir actualizaciones.

Siguientes pasos (opcionales)

- Abre un PR y notifica que terminaste.
- Crea un archivo Markdown con lo que aprendiste y dudas pendientes.
- Crea tu Profile README.
- Crea un repositorio nuevo y explora sus funciones.
- Danos retroalimentación sobre este contenido.

Recursos

- Video corto: ¿Qué es GitHub?
- Documentación oficial de Git y GitHub.
- GitHub Learning Lab y foros de la comunidad.

B. Ejemplo guiado: “Hola Mundo” versionado (Asignación de Classroom)

Contexto

En GitHub Classroom, aceptarás una asignación que crea un repositorio privado para ti a partir de un *starter repo*. El objetivo es versionar un “Hola Mundo” siguiendo el *GitHub flow*.

Estructura inicial

```
README.md     # instrucciones
src/hello.py  # o main.java / index.js / Main.cpp, etc.
```

Pasos del estudiantado

1. Clonar tu repo privado:

```
git clone https://github.com/org-clase/assignments-holamundo-usuario.git
cd assignments-holamundo-usuario
```

2. Crear y cambiar a tu rama:

```
git checkout -b feature/hola-mundo-usuario
```

3. Implementar y probar:

```
echo 'print("Hola, mundo desde Git!")' > src/hello.py
python src/hello.py
```

4. Añadir, confirmar y enviar:

```
git add src/hello.py
git commit -m "feat: hola mundo inicial"
git push -u origin feature/hola-mundo-usuario
```

5. Abrir un Pull Request en GitHub.

6. Atender comentarios y hacer *merge*.

7. Actualizar `main` local:

```
git checkout main
git pull origin main
```

Reglas sugeridas

- Proteger `main` para exigir PR y revisión.
- Mensajes de commit con prefijos como `feat`, `fix`, `docs`, `chore`.
- Ramas con formato: `feature/nombre-usuario`.

C. Diagramas TikZ: Integración Git (local) & GitHub (remoto)

Arquitectura: áreas de Git y remoto en GitHub

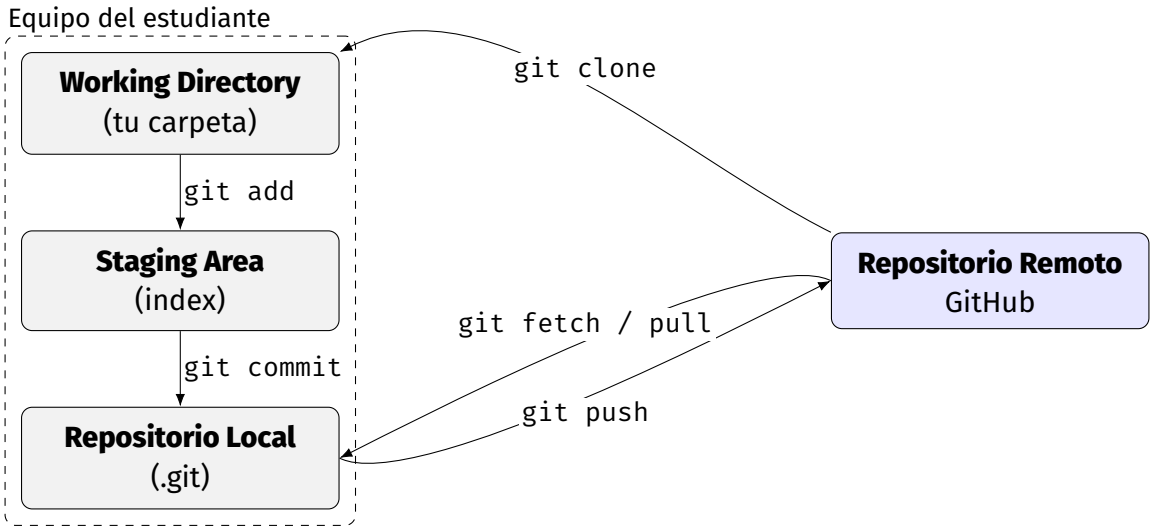


Figura 1: Flujo básico entre las áreas de Git y el repositorio remoto en GitHub.

Creación de una clave SSH en macOS

1. **Verificar si ya existe una clave SSH en el sistema**:

   ```
   ls -al ~/.ssh
   ```

   Buscar archivos como `id_ed25519` y `id_ed25519.pub` o `id_rsa` y `id_rsa.pub`.

2. **Generar un nuevo par de claves SSH** (recomendado Ed25519):

   ```
   ssh-keygen -t ed25519 -C "tu_email@example.com"
   ```

En caso de necesitar RSA por compatibilidad:

```
ssh-keygen -t rsa -b 4096 -C "tu_email@example.com"
```

3. **Elegir ubicación y passphrase**:
   - Presionar `Enter` para usar la ruta por defecto (`/Users/usuario/.ssh/id_ed25519`).
   - Ingresar una passphrase opcional (vacío para no usar passphrase).
4. **Iniciar el agente SSH**:

```
eval "$(ssh-agent -s)"
```

5. **Agregar la clave al agente y al Keychain de macOS**:

```
ssh-add --apple-use-keychain ~/.ssh/id_ed25519
```

6. **Visualizar la clave pública**:

```
cat ~/.ssh/id_ed25519.pub
```

Copiar el contenido y pegarlo en el servicio o servidor donde se vaya a usar (por ejemplo, en GitHub en `Settings → SSH and GPG keys`).

7. **(Opcional) Configurar el archivo `/.ssh/config`** para cargar la clave automáticamente al iniciar sesión:

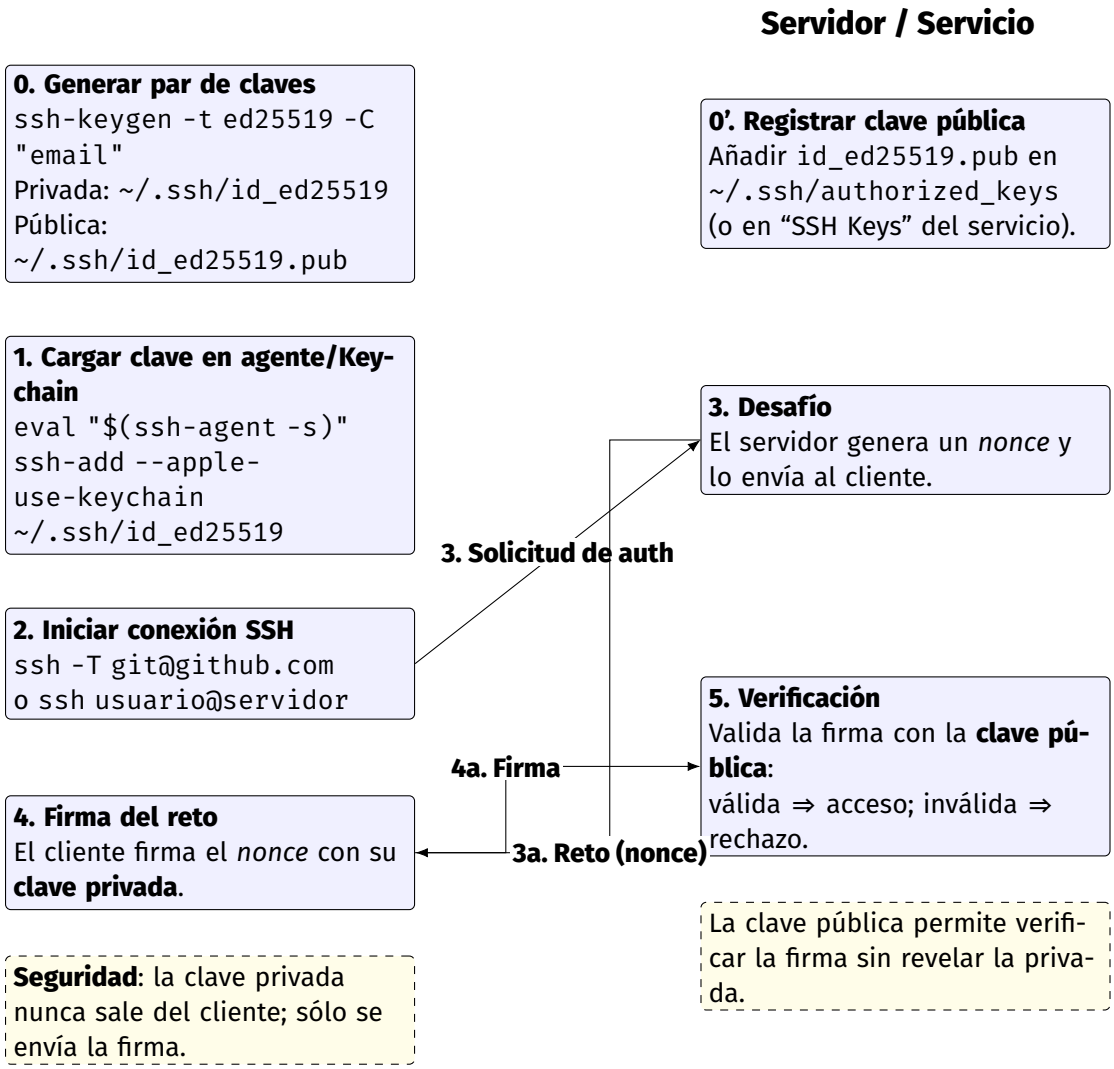
```
Host *
  AddKeysToAgent yes
  UseKeychain yes
  IdentityFile ~/.ssh/id_ed25519
```

8. **Probar la conexión**:

```
ssh -T git@github.com
```

o para un servidor:

```
ssh usuario@servidor.com
```


Servidor / Servicio
0. Generar par de claves
ssh-keygen -t ed25519 -C "email"
Privada: ~/.ssh/id_ed25519
Pública: ~/.ssh/id_ed25519.pub
0'. Registrar clave pública
Añadir id_ed25519.pub en ~/.ssh/authorized_keys (o en "SSH Keys" del servicio).
1. Cargar clave en agente/Keychain
eval "$(ssh-agent -s)"
ssh-add --apple-use-keychain ~/.ssh/id_ed25519
3. Desafío
El servidor genera un nonce y lo envía al cliente.
3. Solicitud de auth
2. Iniciar conexión SSH
ssh -T git@github.com
o ssh usuario@servidor
5. Verificación
Valida la firma con la clave pública:
válida ⇒ acceso; inválida ⇒ rechazo.
4a. Firma
4. Firma del reto
El cliente firma el nonce con su clave privada.
3a. Reto (nonce)
La clave pública permite verificar la firma sin revelar la privada.
Seguridad: la clave privada nunca sale del cliente; sólo se envía la firma.