

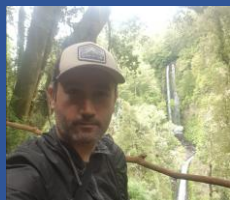


UNIVERSIDAD  
AUTÓNOMA  
DE CHILE

MÁS UNIVERSIDAD

# ANÁLISIS DE ALGORITMOS

Mg. Marcos Contreras F.



**Marcos Contreras F.**

<https://www.linkedin.com/in/marcos-c-b7162610a/>



[marcos.contreras@cloud.uautonoma.cl](mailto:marcos.contreras@cloud.uautonoma.cl)



Expectativas de los  
estudiantes

## SER PUNTUAL



“ES VALORAR TU TIEMPO Y EL MÍO”



ATENCION Y CONCENTRACION





**¿Qué es un Algoritmo?**

**¿Cual es su objetivo?**



**Un algoritmo es una secuencia finita de pasos bien definidos para resolver un problema computacional.**

**Objetivo: Evaluar la eficiencia de los algoritmos en términos de tiempo y espacio.**

# Modelamiento de Problemas

El modelamiento de problemas es el proceso de traducir un problema del mundo real en una representación matemática o computacional que pueda ser resuelta mediante algoritmos.





# Pasos del Modelamiento de Problemas

1. **Definición del Problema:** Identificación clara de los datos de entrada y salida esperados.
2. **Abstracción:** Ignorar detalles irrelevantes y centrarse en la estructura esencial del problema.
3. **Elección del Modelo:** Representación del problema mediante estructuras de datos adecuadas (grafos, matrices, listas, etc.).
4. **Identificación de Restricciones:** Reglas que limitan las soluciones posibles.
5. **Determinación del Criterio de Evaluación:** ¿Qué hace que una solución sea óptima o válida?
6. **Selección del Algoritmo Apropriado:** Elegir la mejor estrategia basada en complejidad y eficiencia.



## Ejemplo: Modelamiento de un Problema de Rutas

- **Entrada:** Un mapa con ciudades y carreteras.
- **Salida:** La ruta más corta entre dos ciudades.
- **Modelo:** Representación con un grafo donde los nodos son ciudades y las aristas representan carreteras con pesos.
- **Restricciones:** Algunas carreteras pueden estar cerradas o tener costos variables.
- **Criterio de Evaluación:** Minimización de la distancia o el tiempo de viaje.
- **Algoritmo Seleccionado:** Algoritmo de Dijkstra o A\*.



# Evaluación de Decisiones Heurísticas

Las heurísticas son estrategias aproximadas para encontrar soluciones eficientes cuando no es factible calcular la solución óptima en un tiempo razonable.

## Conceptos Claves en Heurísticas

- **Solución Aproximada:** No garantiza el resultado óptimo, pero es útil cuando el costo de calcular la mejor solución es prohibitivo.
- **Balance entre Precisión y Eficiencia:** Se sacrifica precisión a cambio de una mayor rapidez en la solución.
- **Exploración vs. Explotación:** Decidir entre probar nuevas soluciones o mejorar las ya existentes.



## Evaluación de Heurísticas

- **Exactitud:** ¿Cuán cerca está la solución de la óptima?
- **Tiempo de Ejecución:** ¿Es más rápido que un enfoque exacto?
- **Robustez:** ¿Funciona bien en diferentes tipos de instancias del problema?
- **Simplicidad:** ¿Es fácil de implementar y entender?

```
# Algoritmo Merge Sort
def merge_sort(arr):
    if len(arr) > 1:
        medio = len(arr) // 2 # Encuentra el punto medio del arreglo
        izquierda = arr[:medio] # Divide la lista en dos mitades
        derecha = arr[medio:]
        merge_sort(izquierda) # Ordena la mitad izquierda
        merge_sort(derecha) # Ordena la mitad derecha
        i = j = k = 0 # Índices para recorrer las listas
```

### Paso 1: División del arreglo

- Calculamos el punto medio del arreglo.
- Dividimos el arreglo en dos mitades: izquierda y derecha.

### Paso 2: Llamada recursiva para ordenar las mitades

- Aplicamos merge\_sort de manera recursiva a ambas mitades hasta que cada lista contenga un solo elemento.

```
# Mezclar las sublistas ordenadas
```

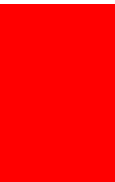
```
while i < len(izquierda) and j < len(derecha):  
    if izquierda[i] < derecha[j]:  
        arr[k] = izquierda[i]  
        i += 1  
    else:  
        arr[k] = derecha[j]  
        j += 1  
    k += 1  
while i < len(izquierda):  
    arr[k] = izquierda[i]  
    i += 1  
    k += 1  
while j < len(derecha):  
    arr[k] = derecha[j]  
    j += 1  
    k += 1
```

Paso 3: Mezcla de las mitades ordenadas

- Se comparan los elementos de las dos mitades (izquierda y derecha).
- Se insertan en el arreglo original arr en orden ascendente.
- Se avanza en la mitad correspondiente (i o j).

Paso 4: Agregar elementos restantes

- Si quedaron elementos sin procesar en izquierda, los añadimos al final del arreglo.
- Si quedaron elementos en derecha, también los agregamos.



```
# Ejemplo de uso
arr = [38, 27, 43, 3, 9, 82, 10]
merge_sort(arr)
print("Lista ordenada:", arr)
```

```
⇨ Lista ordenada: [3, 9, 10, 27, 38, 43, 82]
```



## **Paso a Paso:**

### **1.División del Problema:**

- Se divide el arreglo en dos mitades recursivamente hasta que cada sublista contenga un solo elemento.

### **2.Conquista (Ordenación de Sublistas):**

- Se ordenan las mitades de forma recursiva.

### **3.Fusión de Resultados:**

- Se combinan las sublistas ordenadas en un solo arreglo ordenado.

### **4.Salida:**

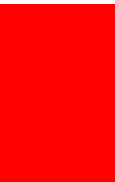
- Se obtiene una lista completamente ordenada.

```
import heapq
def dijkstra(grafo, inicio):
    # 1. Inicialización de distancias con infinito, excepto el nodo de inicio
    distancias = {nodo: float('inf') for nodo in grafo}
    distancias[inicio] = 0
```

## Inicialización de estructuras

- Se crea un diccionario distancias con valores  $\infty$  (infinito) para representar que al inicio no conocemos la distancia a ningún nodo.
- Se establece la distancia al nodo de inicio como 0 porque ya estamos ahí.





```
# 2. Min-Heap para seleccionar el nodo con la menor distancia  
cola_prioridad = [(0, inicio)] # (distancia, nodo)
```

- Se usa un min-heap (cola de prioridad) para procesar primero los nodos con la menor distancia conocida.
- heapq maneja la cola ordenando los elementos automáticamente para que siempre podamos extraer el nodo con la menor distancia en  $O(\log n)$ .

```
while cola_prioridad:
    # 3. Extraemos el nodo con la menor distancia
    distancia_actual, nodo_actual = heapq.heappop(cola_prioridad)
```

## Selección del nodo con la menor distancia

- Se extrae el nodo con la menor distancia de la cola de prioridad.

```
# 4. Ignoramos si encontramos una distancia mayor (ya fue procesado)
if distancia_actual > distancias[nodo_actual]:
    continue
```

Si el nodo extraído tiene una distancia mayor que la almacenada en distancias, se ignora (porque ya fue procesado con una mejor distancia antes).

## Relajación de aristas

```
# 5. Relajación de las aristas
```

```
    for vecino, peso in grafo[nodo_actual].items():  
        nueva_distancia = distancia_actual + peso
```

- Se recorren los nodos vecinos del nodo\_actual y se calcula una nueva distancia posible sumando el peso de la arista.

```
# Si encontramos una ruta más corta, actualizamos la distancia
```

```
    if nueva_distancia < distancias[vecino]:  
        distancias[vecino] = nueva_distancia  
        heapq.heappush(cola_prioridad, (nueva_distancia, vecino))
```

- Si encontramos un camino más corto hacia el vecino, actualizamos su distancia en distancias.
- Se inserta el nodo actualizado en la cola de prioridad para que se procese en el futuro.

# ¿Para que sirve el análisis de algoritmos?

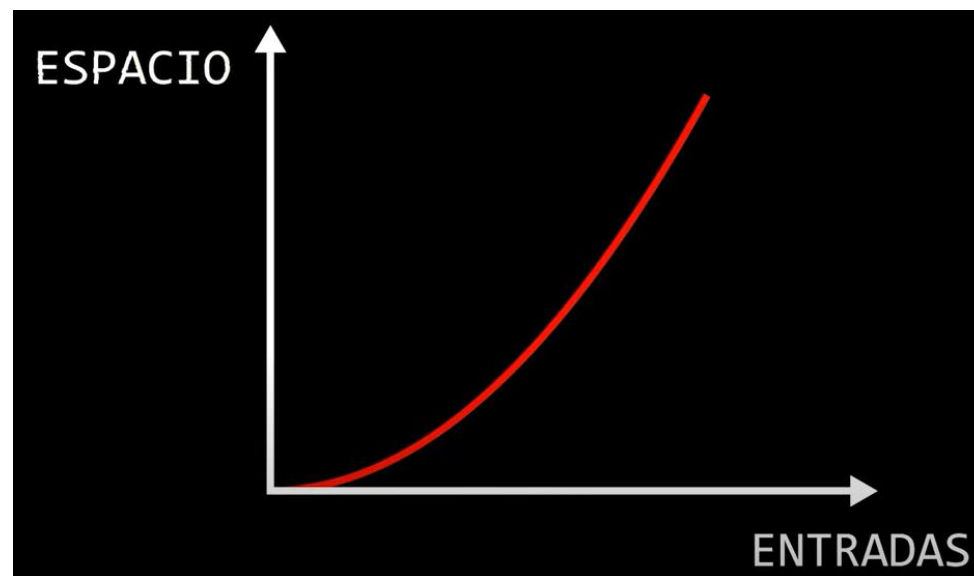
El análisis de algoritmos es una técnica que permite evaluar la eficiencia de los algoritmos, es decir, cuánto tiempo y espacio se necesitan para realizar operaciones.

El análisis de algoritmos sirve para:

- Decidir la estructura de datos más adecuada para una tarea específica
- Mejorar la velocidad y el rendimiento de un algoritmo y del software en general
- Evaluar la idoneidad de un algoritmo para diversas aplicaciones
- Comparar algoritmos para la misma aplicación
- Estimaciones teóricas de los recursos que necesita un algoritmo

La complejidad de un algoritmo es la función que mide cuántos recursos va a necesitar nuestro algoritmo con respecto al tamaño de la entrada, normalmente, en tiempo o en espacio.

La complejidad en espacio es la cantidad de memoria que va a necesitar nuestro algoritmo en la ejecución.



Imagina que tienes que reordenar una lista de números aleatoriamente. Tu algoritmo puede consistir en generar una lista con los índices y otra lista con el mismo número de elementos con cualquier valor, por ejemplo, ceros.

Empezamos recorriendo la lista de números, elegimos uno de los índices aleatoriamente y copiamos ese número en la lista de ceros, en la posición del índice elegido, borramos el índice y empezamos de nuevo con el siguiente elemento, elegimos aleatoriamente otro índice de los restantes, copiamos el elemento, borramos el índice y pasamos al siguiente elemento, así hasta recorrer todos los elementos.

```
[1, 2, 3, 4, 5]
[0, 1, 2, 3, 4]
[0, 0, 0, 0, 0]
```

```
  ▼
[①, 2, 3, 4, 5]
[0, 1, 2, 3, 4]
[0, 0, 1, 0, 0]
```

```
  ▼
[1, 2, 3, 4, 5]
  [0, 1, 3, 4]
[0, 0, 1, 0, 2]
```

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

Hemos usado dos listas auxiliares, una para los índices y otra para los elementos que hemos copiado, con 5 elementos. Si el tamaño de la entrada es 5, este algoritmo necesita 2 veces más elementos en memoria para poder usarse, 10. Es decir, si el número de elementos de entrada es  $n$ , necesita  $2n$ .

```
[ 5 elementos ] 5  
[ 5 elementos ]  
[ 5 elementos ] 10
```

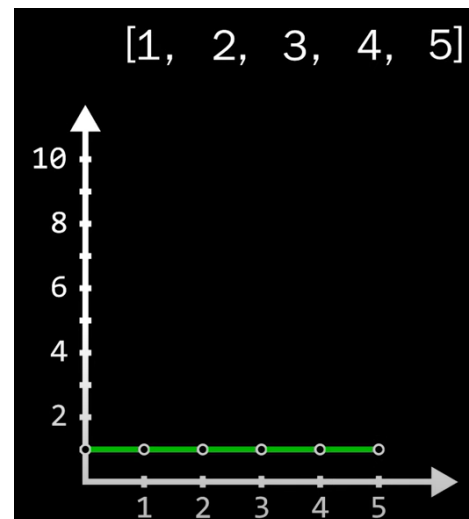
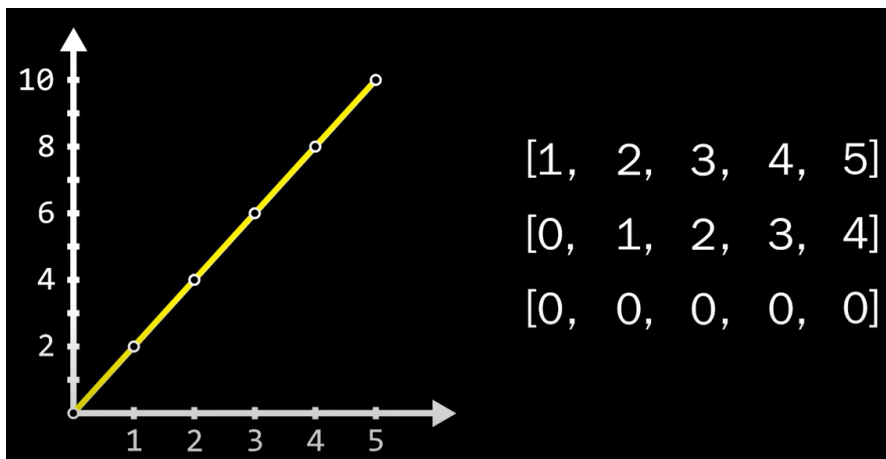
```
[ n elementos ] n  
[ n elementos ]  
[ n elementos ] 2n
```



Imaginemos que tenemos otro algoritmo donde recorremos la lista [1,2,3,4,5] y elegimos una posición de la lista aleatoriamente, almacenamos una copia del elemento, luego intercambiamos y pasamos al siguiente elemento, luego elegimos otra posición aleatoria, almacenamos y así hasta terminar de recorrer la lista.

Para este algoritmo no necesitamos una lista auxiliar, solo una copia del elemento para poder intercambiarlos, solo se necesita un elemento en memoria en toda la ejecución.

El primer algoritmo a medida que la lista es mayor va a necesitar mas memoria para ejecutarse. En cambio, el segundo, no necesita memoria extra para ejecutar, solo almacena un elemento durante toda la ejecución.



## Mejor Caso (Best Case) – $O(B(n))$

Tenemos un arreglo y queremos buscar el número 5 usando búsqueda lineal:

```
def busqueda_lineal(arr, x):  
    for i in range(len(arr)):  
        if arr[i] == x:  
            return i # Elemento encontrado, se detiene  
    return -1 # Elemento no encontrado  
  
arr = [5, 8, 10, 20, 50]  
print(busqueda_lineal(arr, 5)) # Retorna 0 en el mejor caso
```

### Qué significa:

- Es la cantidad **mínima** de operaciones que ejecutará el algoritmo.
- Representa la ejecución más rápida posible.
- Se da cuando los datos están en la **mejor configuración posible** para el algoritmo.

## Peor Caso (Worst Case) – $O(W(n))$

¿Qué significa?

- Es la cantidad máxima de operaciones que ejecutará el algoritmo.
- Representa la ejecución más lenta posible.
- Se da cuando los datos están en la peor configuración posible para el algoritmo.

```
def busqueda_lineal(arr, x):  
    for i in range(len(arr)):  
        if arr[i] == x:  
            return i # Elemento encontrado, se detiene  
    return -1 # Elemento no encontrado  
  
arr = [5, 8, 10, 20, 50]  
print(busqueda_lineal(arr, 50)) # Retorna 4 en el peor caso
```



## ¿Que buscamos con el análisis de Algoritmos ?

El análisis de algoritmos busca determinar la eficiencia de un algoritmo midiendo dos recursos principales:

- **Tiempo de Ejecución (Complejidad Temporal):** Cuánto tiempo tarda el algoritmo en ejecutarse en función del tamaño de la entrada.
- **Uso de Memoria (Complejidad Espacial):** Cuánta memoria consume el algoritmo mientras se ejecuta.

### Tipos de Análisis

- **Mejor Caso (Best Case):** Situación más favorable para el algoritmo.
- **Peor Caso (Worst Case):** Situación más desfavorable, cuando el algoritmo toma más tiempo.
- **Caso Promedio (Average Case):** El tiempo esperado considerando todas las posibles entradas.

# Notación Asintótica

La notación asintótica se usa para describir el comportamiento de un algoritmo cuando el tamaño de la entrada tiende a infinito.

Nos ayuda a entender la eficiencia de un algoritmo sin depender de detalles específicos de la implementación o del hardware.

Se utilizan para describir la complejidad del algoritmo cuando la entrada tiende a infinito.

- $O$  (Big O): Cota superior. Describe el crecimiento máximo del algoritmo.
- $\Omega$  (Omega): Cota inferior. Describe el crecimiento mínimo.
- $\Theta$  (Theta): Cota ajustada. Acota por arriba y por abajo el comportamiento del algoritmo.

Ejemplo:

Un algoritmo que recorre una lista de  $n$  elementos tiene complejidad  $O(n)$ .

# Diseño de Algoritmos

El diseño de algoritmos implica planificar y construir algoritmos eficientes y correctos. Existen varios **paradigmas de diseño**, cada uno útil para diferentes tipos de problemas:

## Principales Paradigmas:

### 1. Divide y Vencerás (Divide and Conquer):

1. Divide el problema en subproblemas más pequeños, resuelve cada uno y combina sus resultados.
2. Ejemplo: Merge Sort, Quick Sort.

### 2. Programación Dinámica:

1. Resuelve problemas dividiéndolos en subproblemas más pequeños y almacenando sus soluciones para evitar recalcularlas.
2. Ejemplo: Fibonacci, Problema de la Mochila.



## **Algoritmos Voraces (Greedy):**

Toma la mejor decisión en cada paso sin considerar consecuencias futuras. Ejemplo: Dijkstra.

## **Backtracking:**

Explora todas las posibles soluciones, retrocediendo cuando una no cumple las condiciones. Ejemplo: Sudoku, N-Reinas.

## **Branch and Bound:**

Similar al Backtracking, pero con una estrategia que descarta soluciones no prometedoras. Ejemplo: Problemas de optimización combinatoria.



# Encontrar el mayor valor de una lista arbitraria

Considere la implementación defectuosa de Python, que intenta encontrar el mayor valor de una lista arbitraria que contenga al menos un valor, comparando cada valor de  $A$  con  $my\_max$ , actualizando  $my\_max$  según sea necesario cuando se encuentren valores mayores.

## Implementación defectuosa para localizar el valor más grande de la lista

```
def flawed(A):  
    my_max = 0  
    for v in A:  
        if my_max < v:  
            my_max = v  
    return my_max
```

1.  $my\_max$  es una variable que contiene el valor máximo; aquí  $my\_max$  se inicializa a 0.
2. El bucle for define una variable  $v$  que itera sobre cada elemento de  $A$ . La sentencia if se ejecuta una vez por cada valor  $v$ .
3. Actualiza  $my\_max$  si  $v$  es mayor.

Un elemento central de esta solución es el operador menor que (<), que compara dos números para determinar si un valor es menor que otro.

En la Figura 1-2, a medida que **v** toma valores sucesivos de **A**, puedes ver que **my\_max** se actualiza tres veces para determinar el mayor valor de **A**. **flawed()** determina el mayor valor de **A**, invocando menos que seis veces, una para cada uno de sus valores.

```
def flawed(A):  
    my_max = 0  
    for v in A:  
        if my_max < v:  
            my_max = v  
    return my_max
```

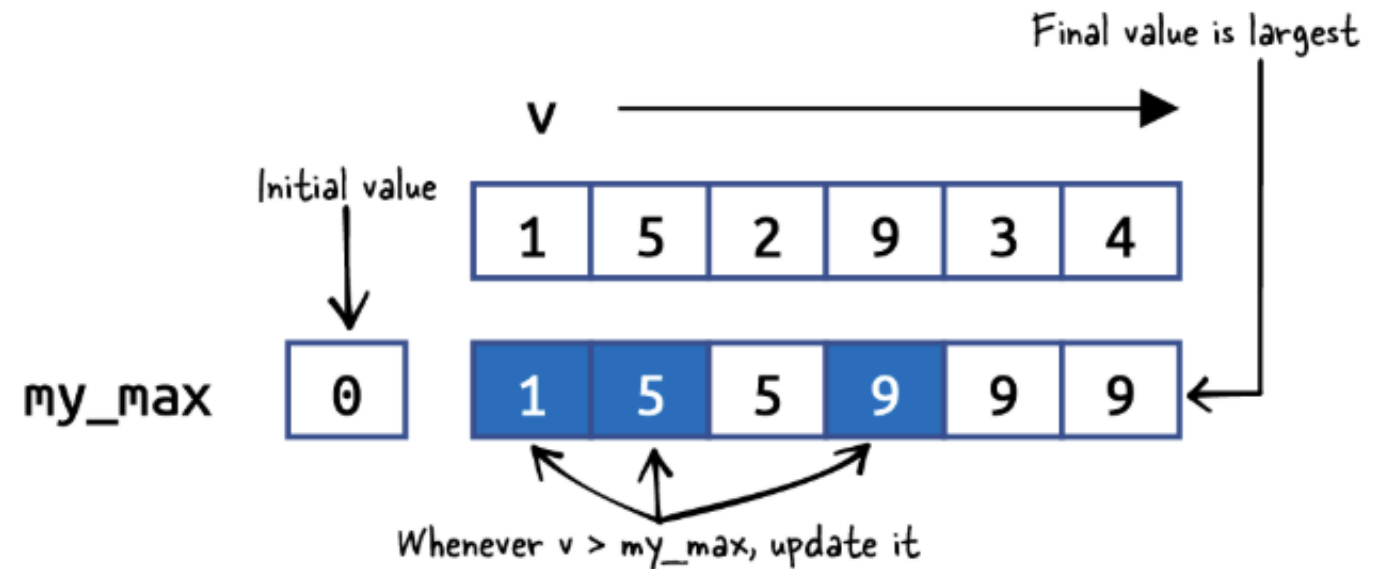


Figura 1-2. Visualización de la ejecución de `flawed()`



Esta implementación es defectuosa porque supone que al menos un valor de A es mayor que 0.

Al calcular `flawed([-5,-3,-11])` se obtiene 0, lo cual es incorrecto. Una solución habitual es inicializar `my_max` con el valor más pequeño posible, como `my_max = float('-inf')`.

Este enfoque sigue siendo defectuoso, ya que devolvería este valor si A fuera la lista vacía `[]`.

# Operaciones clave de recuento

Como el valor mayor debe estar contenido en A, la función `largest()` correcta del Listado 1-2 selecciona el primer valor de A como `my_max`, comprobando los demás valores para ver si alguno es mayor.

```
def largest(A):  
    my_max = A[0] ❶  
    for idx in range(1, len(A)):  
        if my_max < A[idx]: ❷  
            my_max = A[idx] ❸  
    return my_max
```

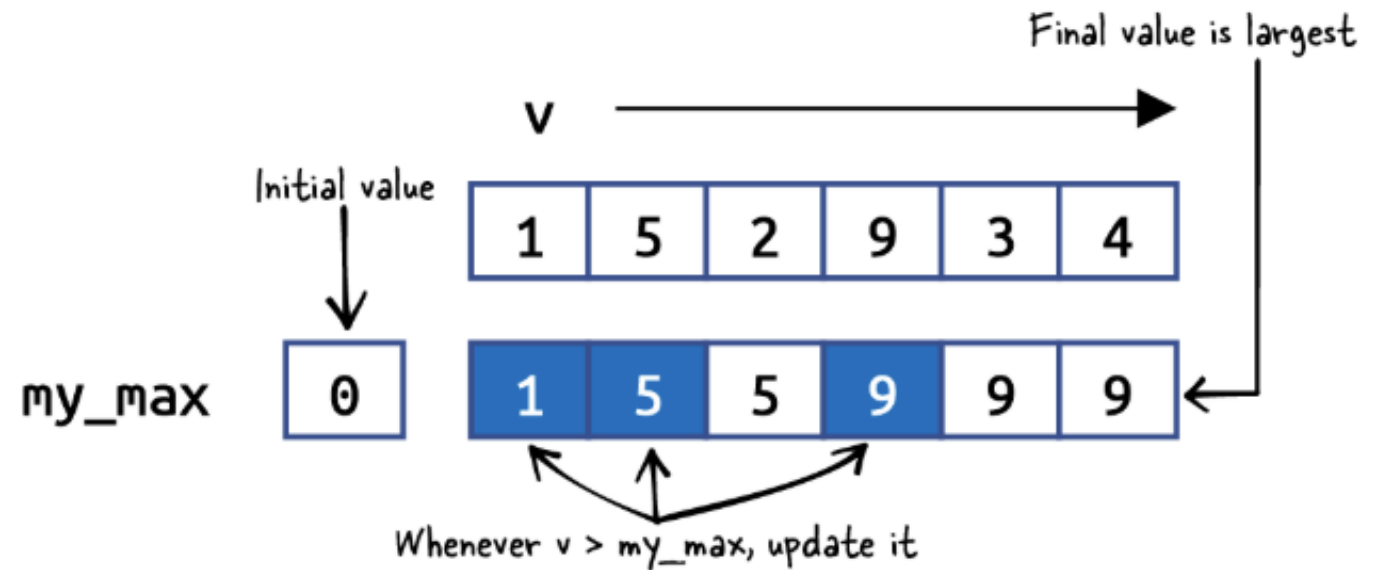


Figura 1-2. Visualización de la ejecución de `largest()`

1. Establece `my_max` en el primer valor de `A`, que se encuentra en la posición 0 del índice.
2. `Idx` toma valores enteros desde 1 hasta, pero sin incluir, `len( A)`.
3. Actualiza `my_max` si el valor de `A` en la posición `idx` es mayor.

```
def largest(A):  
    my_max = A[0] ❶  
    for idx in range(1, len(A)):  
        if my_max < A[idx]: ❷  
            my_max = A[idx] ❸  
    return my_max
```

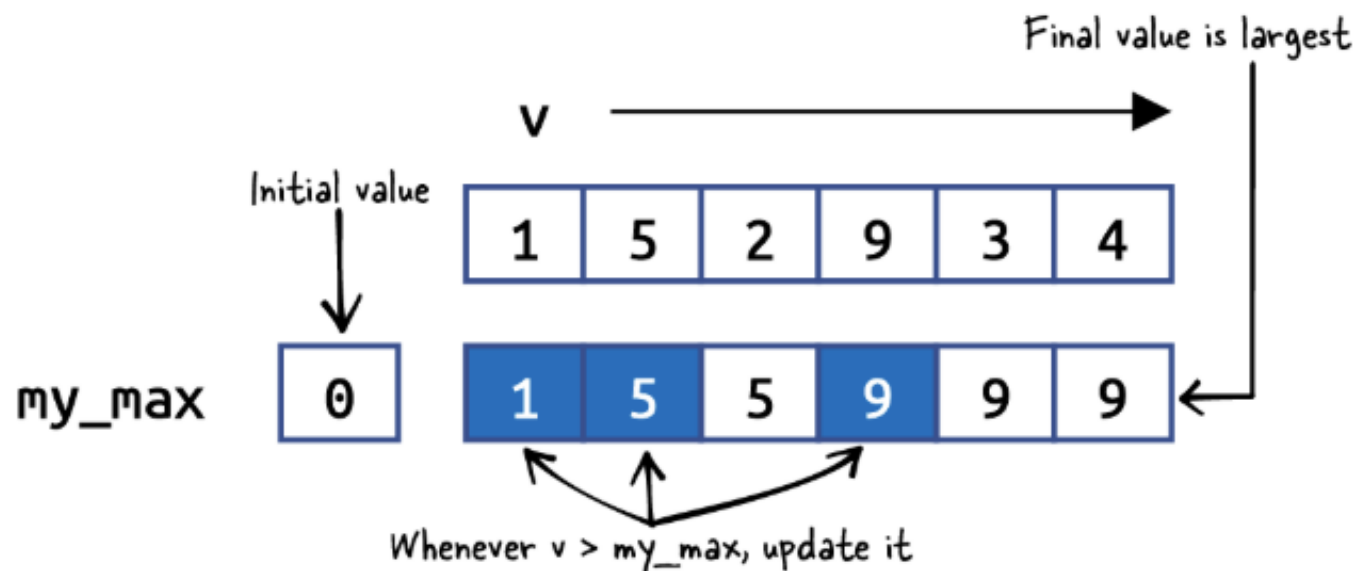


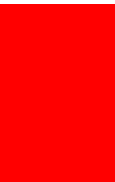
Figura 1-2. Visualización de la ejecución de `largest()`



**Los modelos pueden predecir el rendimiento de los algoritmos.**

¿Y si alguien te muestra un algoritmo diferente para este mismo problema?

¿Cómo determinarías cuál utilizar?



Considere el algoritmo `alternate()` del Listado 1-3 que comprueba repetidamente cada valor de A para ver si es mayor o igual que todos los demás valores de la misma lista.

¿Dará este algoritmo el resultado correcto?

¿Cuántas veces invoca a menos que en un problema de tamaño N?



## Un enfoque diferente para localizar el valor más grande en A

```
def alternate(A):  
    for v in A:  
        v_is_largest = True ❶  
        for x in A:  
            if v < x:  
                v_is_largest = False ❷  
                break  
        if v_is_largest:  
            return v ❸  
    return None ❹
```

1. Al iterar sobre A, asume que cada valor v, puede ser el mayor.
2. Si v es menor que otro valor, x distensión y nota que v no es mayor.
3. Si v\_is\_largestes true, devuelve v ya que es el valor máximo en A.
4. Si A es una lista vacía, devuelve None.

alternate() Intenta encontrar un valor, **v** en **A** tal que ningún otro valor, **x** en **A** sea mayor.

La implementación utiliza dos bucles anidados for. Esta vez no es tan sencillo calcular cuantas veces se invoca a menos que, porque el bucle interno for sobre x se detiene en cuanto se encuentra un x que sea mayor que v.

Asimismo, el bucle externo for sobre v se detiene una vez que se encuentra el valor máximo. La Figura 1-3 muestra la ejecución de alternate() en nuestro ejemplo de lista.

```
def alternate(A):  
    for v in A:  
        v_is_largest = True ❶  
        for x in A:  
            if v < x:  
                v_is_largest = False ❷  
                break  
        if v_is_largest:  
            return v ❸  
    return None ❹
```

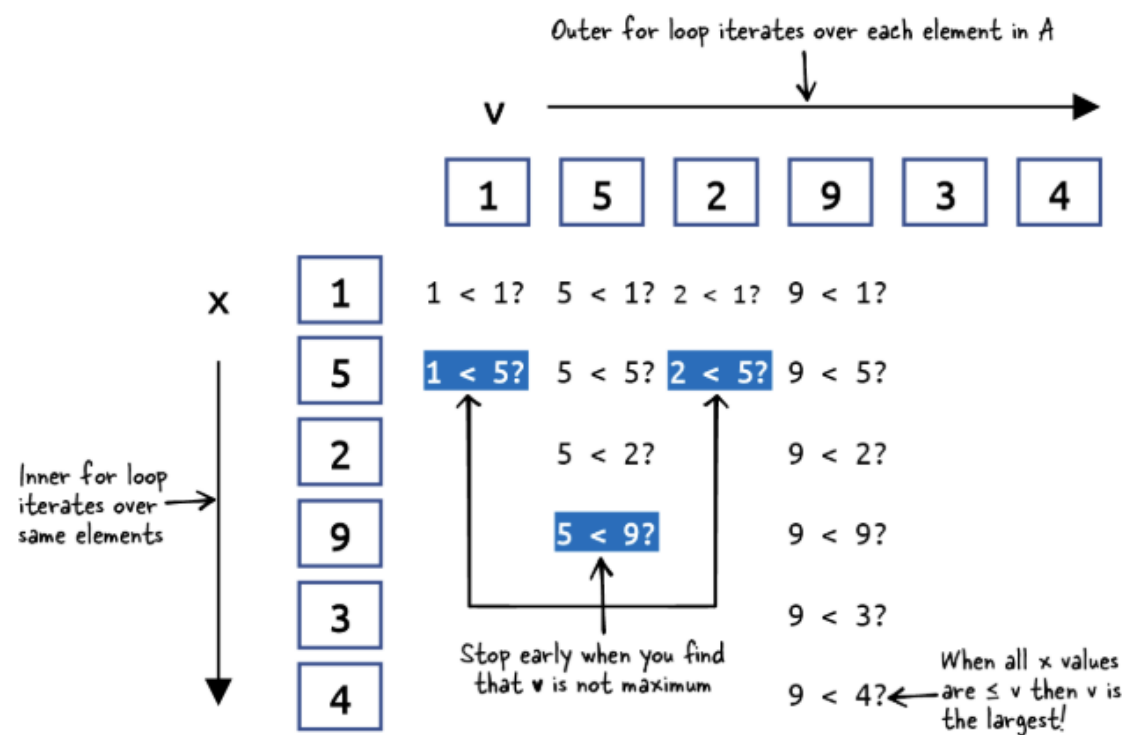
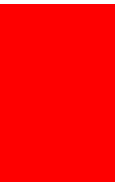


Figura 1-3. Visualización de la ejecución de `alternate()`



En el caso de este problema, se invoca a menos que 14 veces. Pero puedes ver que este recuento total depende de los valores concretos de la lista A.

¿Y si los valores estaban en otro orden?

¿Se te ocurre una disposición de los valores que requieren el menor número de invocaciones de menos que?

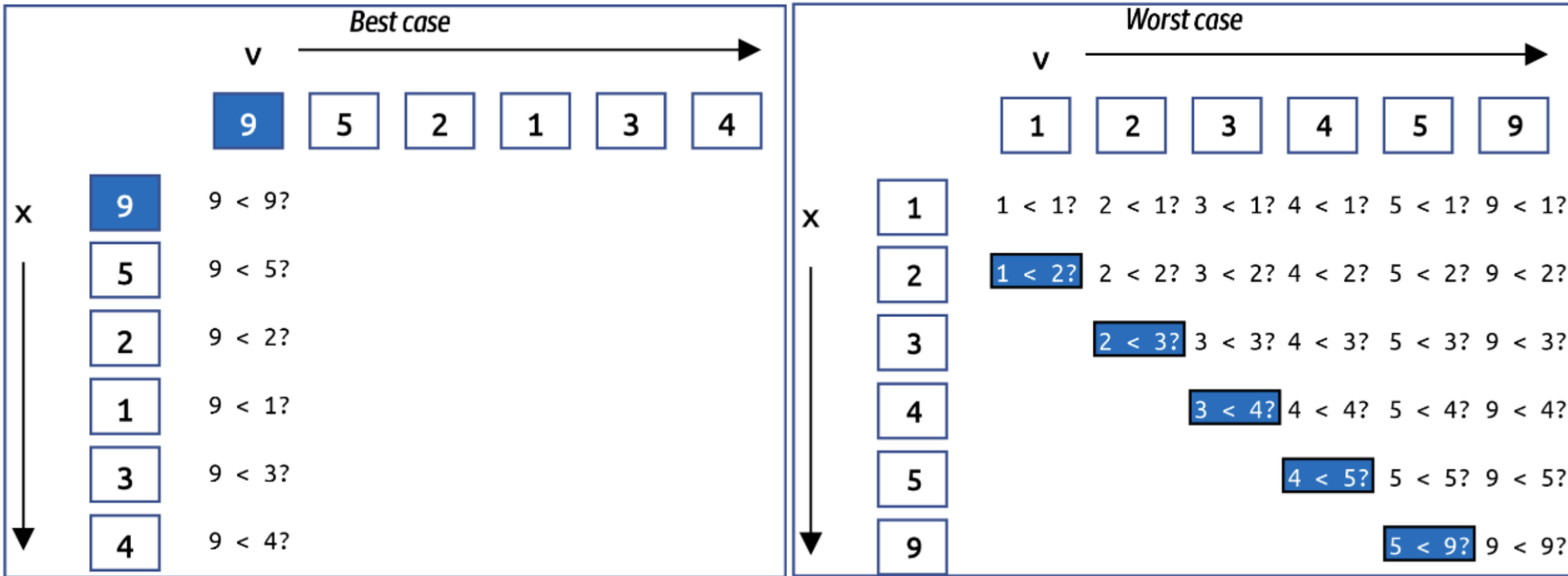
Un problema de este tipo se consideraría el mejor caso para `alternate()`. Por ejemplo, si el primer valor de **A** es el mayor de todos los N valores, entonces el número total de llamadas a menos que es siempre N.

En el mejor de los casos, Una instancia de problema de tamaño N que requiere la menor cantidad de trabajo realizado por un algoritmo.

En el peor de los casos, Una instancia de problema de tamaño N que exige la mayor cantidad de trabajo

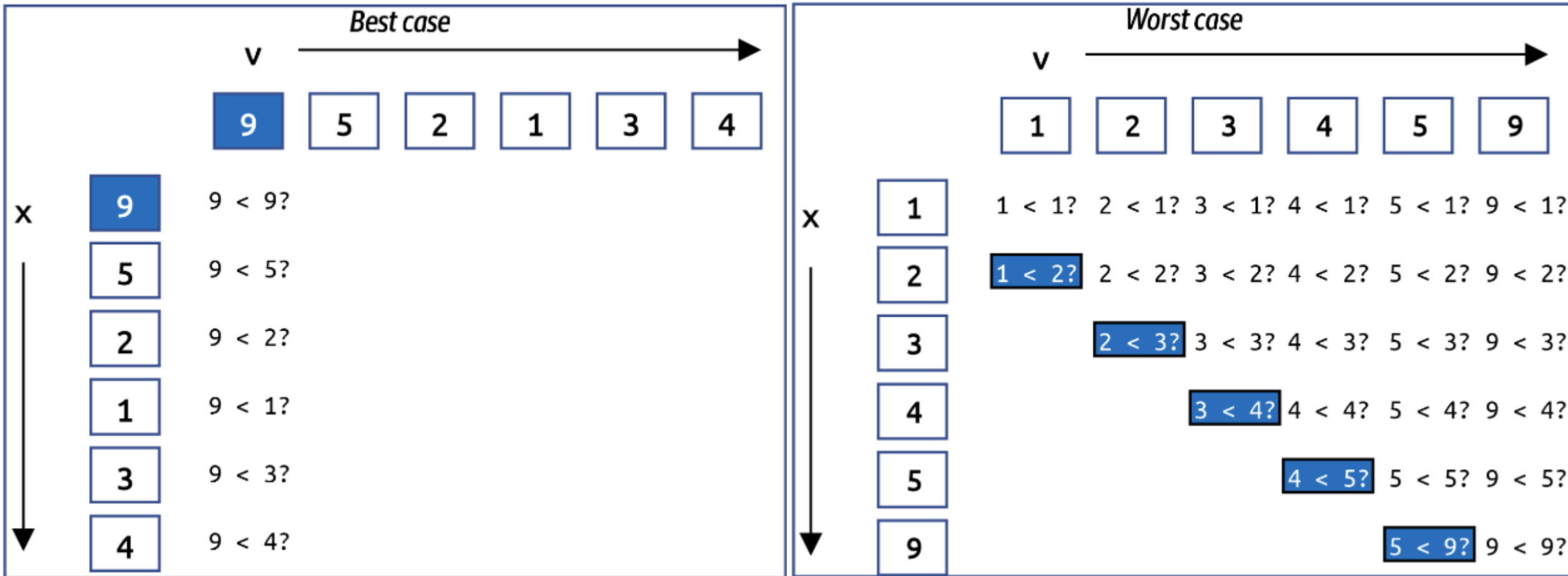
Intentemos identificar el peor caso para `alternate()` que requiera el mayor número de llamadas a menos que. Más que asegurarnos de que el valor mayor es el último de A, en el peor de los casos para `alternate()`, los valores de A deben aparecer en orden ascendente.

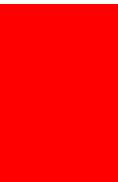
visualiza el mejor caso en la parte superior donde  $p = [9, 5, 2, 1, 3, 4]$  y el peor caso en la parte inferior donde  $p = [1, 2, 3, 4, 5, 9]$ .



En el *mejor de los casos*, hay 6 llamadas a menos que; si hubiera N valores en el *mejor de los casos*, el número total de invocaciones a menos-que sería N.

Es un poco más complicado para el *peor de los casos*. En [la Figura 1-4](#) puedes ver que hay un total de 26 llamadas a menos que cuando la lista de N valores está en orden ascendente.





Pruebas empíricas sobre largest() y alternate()  
en el peor de los casos de problemas de tamaño  
N.

norte	La más grande	Alternativa	La más grande	Alternativa
	(#menos-que)	(#menos-que)	(tiempo en ms)	(tiempo en ms)
8	7	43	0.001	0.001
16	15	151	0.001	0.003
32	31	559	0.002	0.011
64	63	2,143	0.003	0.040
128	127	8,383	0.006	0.153
256	255	33,151	0.012	0.599
512	511	131,839	0.026	2.381
1.024	1,023	525,823	0.053	9.512
2.048	2,047	2,100,223	0.108	38.161

## Algoritmo del Torneo

Un torneo de eliminación simple consiste en un número de equipos que compiten por ser el campeón. Lo ideal es que el número de equipos sea una potencia de 2, como 16 o 64.

El torneo se construye a partir de una secuencia de rondas en las que todos los equipos que quedan en el torneo se emparejan para jugar un partido; los perdedores del partido son eliminados, mientras que los ganadores pasan a la siguiente ronda. El último equipo que queda es el campeón del torneo.

Considere la instancia del problema  $p = [3,1,4,1,5,9,2,6]$  con  $N = 8$  valores. La Figura 1-6 muestra el torneo de eliminación simple cuya ronda inicial compara ocho valores vecinos utilizando menos que; los valores mayores avanzan en el torneo.

En la ronda Elite Eight, se eliminan cuatro valores, quedando los valores  $[3,4,9,6]$ . En la ronda de los Cuatro Finalistas, los valores  $[4,9]$  avanzan, y finalmente 9 es declarado campeón.

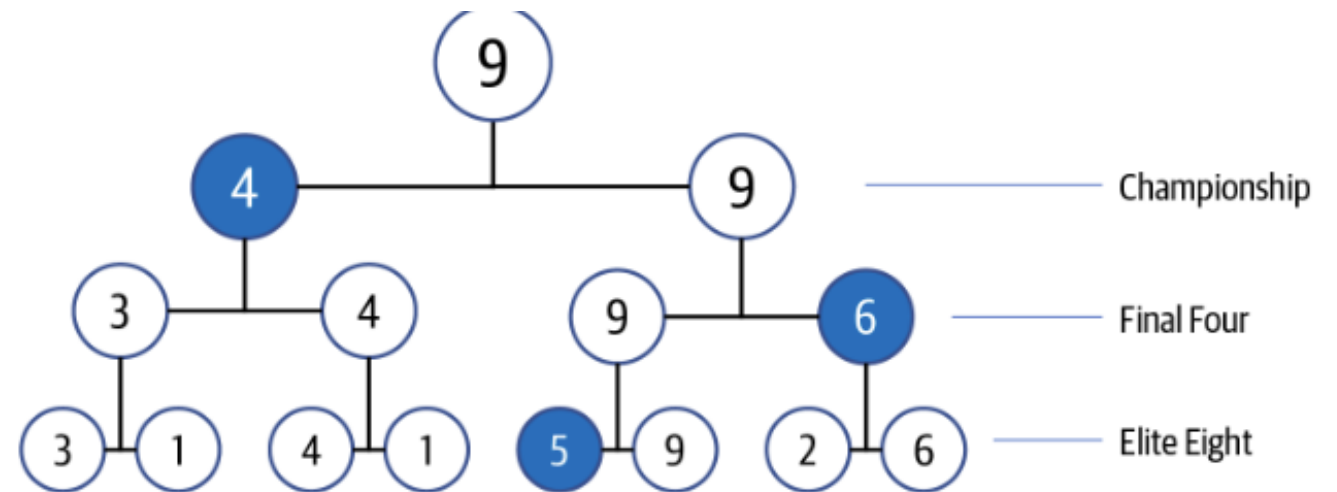



Figura 1-6. Un torneo con ocho valores iniciales





En este torneo, hay siete invocaciones de menos-que (es decir, una por cada coincidencia), lo que debería ser tranquilizador, ya que esto significa que el valor más grande se encuentra con  $N - 1$  usos de menos-que, como habíamos comentado antes.

Si almacenas cada una de estas  $N - 1$  coincidencias, podrás localizar rápidamente el segundo valor más grande, como te muestro a continuación.

¿Dónde se puede "esconder" el segundo valor más grande una vez que el 9 haya sido declarado campeón? Empieza con el 4 como candidato a segundo valor más grande, ya que fue el valor que perdió en la ronda del Campeonato.

Pero el valor más grande, 9, tuvo dos enfrentamientos anteriores, por lo que debes comprobar los otros dos valores perdedores: el valor 6 en la ronda de los Cuatro Finalistas y el valor 5 en la ronda de los Ocho Elite. Así pues, el segundo valor más grande es 6.

Para ocho valores iniciales, sólo necesitas 2 invocaciones menos-que adicionales (¿es  $4 < 6$ ?) y (¿es  $6 < 5$ ?) para determinar que 6 es el segundo valor más grande.

Un algoritmo construye un torneo con 3 rondas. La Figura 1-7 visualiza un torneo de 5 rondas formado por 32 valores.

Para duplicar el número de valores del torneo, sólo necesitas una ronda adicional; En otras palabras, con cada nueva ronda  $K$ , puedes agregar  $2^K$  valores más. ¿Quieres encontrar el mayor de 64 valores? Sólo necesitas 6 rondas, ya que  $2^6 = 64$ .

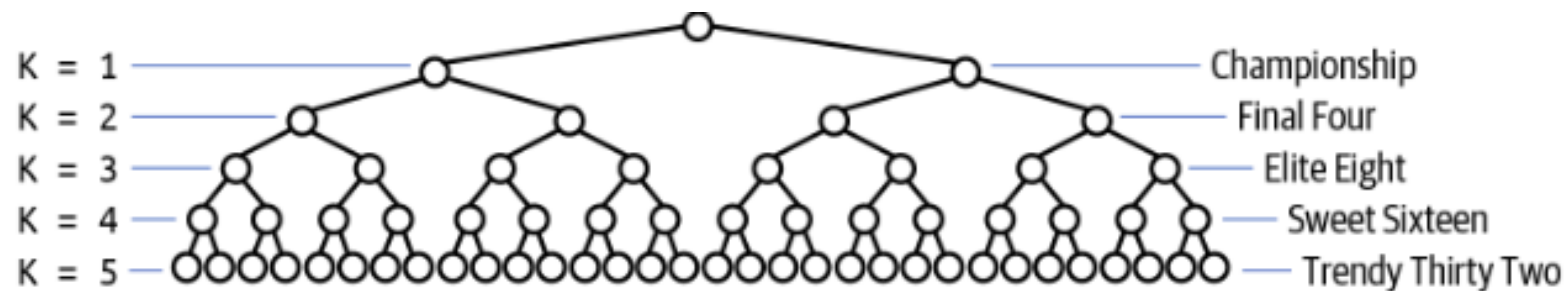


Figura 1-7. Un torneo con 32 valores iniciales.



## Algoritmo para encontrar los dos valores más grandes A en el torneo

```
def tournament_two(A):
    N = len(A)
    winner = [None] * (N-1)           #1
    loser = [None] * (N-1)           #2
    prior = [-1] * (N-1)

    idx = 0
    for i in range(0, N, 2):          #3
        if A[i] < A[i+1]:
            winner[idx] = A[i+1]
            loser[idx] = A[i]
        else:
            winner[idx] = A[i]
            loser[idx] = A[i+1]
        idx += 1

    m = 0                             #4
    while idx < N-1:
        if winner[m] < winner[m+1]:   #5
            winner[idx] = winner[m+1]
            loser[idx] = winner[m]
            prior[idx] = m+1
        else:
            winner[idx] = winner[m]
            loser[idx] = winner[m+1]
            prior[idx] = m
        m += 2                         #6
        idx += 1

    largest = winner[m]
    second = loser[m]                 #7
    m = prior[m]
    while m >= 0:                     #8
        if second < loser[m]:
            second = loser[m]
        m = prior[m]

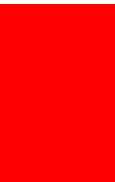
    return (largest, second)
```



```
def tournament_two(A):  
    N = len(A) #Longitud del arreglo  
    winner = [None] * (N-1) #Arreglo para almacenar ganadores  
    loser = [None] * (N-1) #Arreglo para almacenar perdedores  
    prior = [-1] * (N-1) #Relaciona cada ganador con su subtorneo previo (importante para el segundo  
    mayor)
```

Se crean tres listas auxiliares:

- winner: guarda el ganador de cada ronda.
- loser: guarda el perdedor correspondiente.
- prior: para rastrear contra quién compitió el ganador.

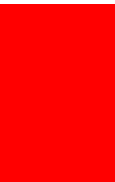


```
idx = 0
for i in range(0, N, 2): # Primera ronda: comparar elementos por pares
    if A[i] < A[i+1]:
        winner[idx] = A[i+1]
        loser[idx] = A[i]
    else:
        winner[idx] = A[i]
        loser[idx] = A[i+1]
    idx += 1
```

- Compara los elementos de dos en dos.
- Guarda el mayor en winner y el menor en loser.
- Incrementa idx para avanzar en el arreglo auxiliar.

```
m = 0
while idx < N-1:                                # Rondas siguientes: comparar ganadores previos
    if winner[m] < winner[m+1]:
        winner[idx] = winner[m+1]
        loser[idx] = winner[m]
        prior[idx] = m+1                        # Guarda el índice del subtorneo previo
    else:
        winner[idx] = winner[m]
        loser[idx] = winner[m+1]
        prior[idx] = m
    m += 2
    idx += 1
```

- A partir de los ganadores previos, se hace un torneo de los ganadores hasta tener el mayor absoluto.
- También se registra contra quién compitió para poder rastrear el segundo mayor.



```
largest = winner[m]           # El ganador final es el máximo
second = loser[m]             # Su oponente es un candidato para segundo mayor
m = prior[m]                  # Revisamos los subtorneos
while m >= 0:
    if second < loser[m]:     # Si algún perdedor fue mayor, actualizamos
        second = loser[m]
    m = prior[m]              # Seguimos hacia atrás en los subtorneos
```

- Una vez encontrado el mayor, buscamos entre todos los elementos que perdió contra él.
- Se compara para encontrar el segundo mayor.

```
return (largest, second) # Devuelve el mayor y el segundo mayor
```

3	1	4	1	5	9	2	6
---	---	---	---	---	---	---	---

Array A

	0	1	2	3	4	5	6	7
winner	3	4	9	6				
loser	1	1	5	2				

↑ m                      ↑ idx

Initialize step

winner	3	4	9	6	4		
loser	1	1	5	2	3		

↑ m                      ↑ idx

Advance step 1

winner	3	4	9	6	4	9	
loser	1	1	5	2	3	6	

↑ m                      ↑ idx

Advance step 2

winner	3	4	9	6	4	9	9
loser	1	1	5	2	3	6	4

↑ m                      ↑ idx

Advance step 3

Figura 1-8. Ejecución paso a paso del algoritmo del torneo



# ¿Qué es un algoritmo de ordenamiento?

Los algoritmos de ordenamiento son técnicas utilizadas para organizar un conjunto de elementos, que pueden ser números o letras, según un orden determinado.

Este orden puede solicitarse en:

- Orden alfabético (A, B, C... o Z, Y, X...).
- Orden numérico ascendente (1, 2, 3...) o descendente (9, 8, 7...).

Estos algoritmos permiten manipular y reorganizar datos de manera efectiva, siendo una habilidad fundamental en la programación y ciencias de la computación.

## Ejemplos de algoritmos de ordenamiento:

- Bubble Sort
- Quick Sort
- Merge Sort
- Insertion Sort

## ¿Qué es el análisis asintótico?

"El análisis asintótico nos permite comparar algoritmos sin necesidad de implementarlos ni de depender de detalles de hardware."

— *Cormen et al., CLRS*

Se enfoca en cómo **crece el tiempo de ejecución o el uso de memoria** a medida que aumenta el **tamaño de la entrada ( $n$ )**, ignorando constantes o factores menores.

## Ejemplo 1: Suma de elementos en una lista

```
def suma(lista):  
    total = 0  
    for x in lista:  
        total += x  
    return total
```

Paso a paso:

1. La función hace una sola operación por cada elemento (n veces).
2. No hay bucles anidados ni llamadas recursivas.
3. Complejidad:
  - Mejor caso:  $\Omega(n)$
  - Peor caso:  $O(n)$
  - Promedio:  $\Theta(n)$



## ¿Qué es el modelamiento de problemas?

“Modelar un problema consiste en **representarlo formalmente** con estructuras de datos y algoritmos adecuados para poder resolverlo computacionalmente.”

— *Cormen et al., CLRS*

Dicho de otra forma: transformar un enunciado, descripción o necesidad del mundo real en **un problema que la computadora pueda entender y resolver**.



## Etapas del modelamiento de problemas

### 1. Entender el problema (contexto)

- ¿Qué se pide?
- ¿Qué datos tengo?
- ¿Cuáles son las restricciones?

### 2. Identificar entradas y salidas

- Entrada: ¿Qué datos necesita mi algoritmo?
- Salida: ¿Qué debe producir?

### 3. Elegir estructuras de datos adecuadas

- Listas, pilas, colas, árboles, grafos, diccionarios, matrices...

### 4. Definir el enfoque algorítmico

- ¿Qué tipo de algoritmo se adapta mejor? (Greedy, fuerza bruta, backtracking, programación dinámica, etc.)

### 5. Analizar la complejidad

- ¿Es eficiente en tiempo y espacio para datos grandes?

## Problema del texto palíndromo con signos y mayúsculas

Elemento	Detalle
Descripción	Validar si una cadena es palíndroma
Entrada	Cadena de texto con signos, espacios, etc.
Salida	<code>True</code> o <code>False</code>
Estructura de datos	String → Lista (opcional)
Algoritmo	Limpiar texto + invertir y comparar
Complejidad	$O(n)$

1. ¿Qué tipo de datos tengo?
2. ¿Cuál es el objetivo?
3. ¿Qué estructuras representan mejor el problema?
4. ¿Cuál es la mejor estrategia de resolución?
5. ¿Es escalable?



## ¿Qué es una heurística?

“Una heurística es una estrategia o regla informal para tomar decisiones o encontrar soluciones que son ‘buenas lo suficiente’, especialmente cuando la solución óptima es costosa o desconocida.”

— *Russell & Norvig, Artificial Intelligence: A Modern Approach*

Ejemplos de heurísticas:

- En ajedrez: “Controlar el centro del tablero”.
- En logística: “Siempre elige el almacén más cercano”.
- En programación: “Elegir primero los casos extremos”.

## Ejemplo: Problema del viajante (TSP)

**Visitar una serie de ciudades una sola vez y volver al origen, minimizando la distancia total.**

Solución exacta: muy costosa (factorial:  $O(n!)$ )

Soluciones heurísticas:

- Greedy: siempre ir a la ciudad más cercana.
- 2-opt: intercambiar pares para mejorar rutas.
- Simulated Annealing: permitir “malas decisiones” para escapar de mínimos locales.



Heurística	Distancia Total	Tiempo de Cálculo
Greedy	1500 km	10 ms
2-opt	1200 km	50 ms
Óptima (brute force)	1100 km	2 horas

Resultado: **2-opt es una muy buena solución**, muy cercana a la óptima, y rápida.

## Greedy Algorithm (Algoritmo voraz)

“Toma siempre la **mejor decisión local inmediata**, con la esperanza de que lleve a una solución global óptima.”

Ejemplo: Cambio de monedas

```
def cambio_greedy(cantidad, monedas):  
    monedas.sort(reverse=True)  
    resultado = []  
    for m in monedas:  
        while cantidad >= m:  
            cantidad -= m  
            resultado.append(m)  
    return resultado  
  
print(cambio_greedy(87, [50, 20, 10, 5, 1])) # Ej: [50, 20, 10, 5, 1, 1]
```

## 2-opt

“Toma una solución inicial (por ejemplo, un recorrido de ciudades) y mejora la ruta **intercambiando pares de aristas (edges)** que se crucen.”

1. Empieza con una ruta inicial.
2. Recorre todos los pares de ciudades  $(i, j)$ .
3. Intercambia el orden de ese subtrayecto (2-opt swap).
4. Si la nueva ruta es más corta, guárdala.
5. Repite hasta que ya no haya mejoras.

```
def distancia(ruta, distancias):
    return sum(distancias[ruta[i-1]][ruta[i]] for i in range(len(ruta)))

def two_opt(ruta, distancias):
    mejor = ruta
    mejor_dist = distancia(mejor, distancias)
    cambio = True
    while cambio:
        cambio = False
        for i in range(1, len(ruta) - 2):
            for j in range(i + 1, len(ruta)):
                if j - i == 1: continue
                nueva = mejor[:i] + mejor[i:j][::-1] + mejor[j:]
                nueva_dist = distancia(nueva, distancias)
                if nueva_dist < mejor_dist:
                    mejor = nueva
                    mejor_dist = nueva_dist
                    cambio = True
    return mejor
```

# Simulated Annealing (Enfriamiento simulado)

“Inspirado en el enfriamiento de metales. Permite aceptar soluciones **peores temporalmente** para escapar de mínimos locales.”

## ¿Cómo funciona?

1. Empieza con una solución inicial y una **temperatura alta**.
2. En cada paso:
  1. Propones una pequeña modificación.
  2. Si mejora, la aceptas.
  3. Si empeora, **puede** que la aceptes con cierta probabilidad (controlada por la temperatura).
3. La temperatura va bajando (enfriamiento), y la probabilidad de aceptar soluciones malas también.

```
import random, math

def simulated_annealing(ruta, distancias, temp_inicial, enfriamiento):
    actual = ruta
    mejor = ruta
    temp = temp_inicial

    def distancia_ruta(r):
        return sum(distancias[r[i-1]][r[i]] for i in range(len(r)))

    while temp > 1:
        i, j = sorted(random.sample(range(1, len(ruta)), 2))
        nueva = actual[:i] + actual[i:j][::-1] + actual[j:]
        delta = distancia_ruta(nueva) - distancia_ruta(actual)

        if delta < 0 or random.random() < math.exp(-delta / temp):
            actual = nueva
            if distancia_ruta(actual) < distancia_ruta(mejor):
                mejor = actual
            temp *= enfriamiento

    return mejor
```



**THANK YOU**

GRACIAS  
ARIGATO  
SHUKURIA  
JUSPAXAR  
DANKSCHEEN  
TASHAKKUR ATU  
SUKSAMA  
EKHMET  
BIYAN  
SHUKRIA  
TINGKI  
YAQHANYELAY  
MAARKE  
MEHRBANI  
PALDIES  
BOLZIN  
MERCIE  
GOZAIMASHITA  
EFCHARISTO  
KOMAPSUNIDA  
MAKEM



**¡¡¡Gracias por la asistencia... Éxito!!!**