

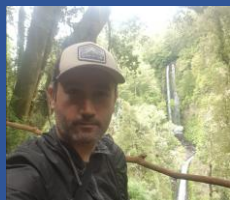


UNIVERSIDAD
AUTÓNOMA
DE CHILE

MÁS UNIVERSIDAD

AUDITORÍA TRADICIONAL

Mg. Marcos Contreras F.



Marcos Contreras F.

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



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
```

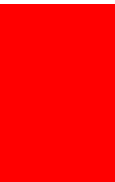
```
    k += 1
```

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:

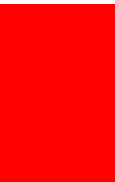
- Se obtiene una lista completamente ordenada.



```
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 ∞ (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



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.

Notación O (Big-O)

- Representa un **límite superior asintótico** (Que se acerca indefinidamente a una recta o a otra curva sin llegar nunca a encontrarla).
- Describe el peor caso o un caso límite de crecimiento del tiempo de ejecución.
- Ignora constantes y términos de menor grado.

Ejemplos de Notación O (Big-O)

- **$O(1)$ - Tiempo Constante**
 1. La cantidad de operaciones no depende del tamaño de la entrada.
 2. **Ejemplo:** Acceder a un elemento de un array por su índice.

```
def obtener_primer(lista):  
    return lista[0] # Siempre toma el mismo tiempo
```

- Este código define una función llamada **obtener_primer(lista)**, que recibe una lista como parámetro y devuelve el primer elemento de esa lista.
- Se define una función llamada **obtener_primer** que recibe un parámetro lista.
- La función retorna el elemento en la posición 0 de la lista, es decir, el primer elemento.

La operación `lista[0]` accede directamente al primer elemento de la lista, lo que en términos de complejidad es $O(1)$ (tiempo constante).

Esto significa que, sin importar el tamaño de la lista, el tiempo de ejecución siempre será el mismo.

O(log n) - Tiempo Logarítmico

- Se reduce la cantidad de elementos procesados en cada iteración.
- **Ejemplo:** Búsqueda binaria en un array ordenado.

✓
0 s



```
def busqueda_binaria(arr, objetivo):  
    izquierda, derecha = 0, len(arr) - 1  
    while izquierda <= derecha:  
        medio = (izquierda + derecha) // 2  
        if arr[medio] == objetivo:  
            return medio  
        elif arr[medio] < objetivo:  
            izquierda = medio + 1  
        else:  
            derecha = medio - 1  
    return -1
```

Se define la función `busqueda_binaria` que recibe:

- **arr**: Una lista ordenada donde se buscará el objetivo.
- **objetivo**: El valor que queremos encontrar en **arr**.

Inicialización de los límites:

- **izquierda**: Indica el inicio del rango de búsqueda (posición 0).
- **derecha**: Indica el final del rango de búsqueda (última posición de la lista).

Bucle de búsqueda:

- Mientras el rango de búsqueda sea válido (es decir, izquierda no sobrepase derecha), se ejecutará la búsqueda.

Cálculo del punto medio: Se encuentra el índice medio de la lista.

Comparación con el objetivo: Si `arr[medio]` es igual al objetivo, se retorna la posición medio.

Ajuste del rango de búsqueda: Si `arr[medio]` es menor que el objetivo, significa que el número buscado está en la mitad derecha, por lo que ajustamos izquierda a `medio + 1`.

Si `arr[medio]` es mayor que objetivo, significa que el número buscado está en la mitad izquierda, por lo que ajustamos derecha a `medio - 1`.

Si el bucle termina sin encontrar el objetivo, se retorna -1, indicando que el valor no está en la lista.

```
def busqueda_binaria(arr, objetivo):
    izquierda, derecha = 0, len(arr) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if arr[medio] == objetivo:
            return medio
        elif arr[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
```

```
arr = [1, 3, 5, 7, 9, 11, 13]
objetivo = 7

resultado = busqueda_binaria(arr, objetivo)
print(resultado)
```

O(n) - Tiempo Lineal

- La cantidad de operaciones crece proporcionalmente al tamaño de la entrada.
- **Ejemplo:** Recorrer una lista de n elementos.

✓
0 s

```
def imprimir_lista(lista):  
    for elemento in lista:  
        print(elemento)
```

Definición de la función:

def imprimir_lista(lista): → Define una función llamada imprimir_lista que recibe una lista como argumento.

Iteración sobre la lista:

for elemento in lista: → Recorre cada elemento de la lista.

Impresión de los elementos:

print(elemento) → Muestra cada elemento en la consola.

O(n) - Tiempo Lineal significa que el tiempo de ejecución de un algoritmo crece proporcionalmente al tamaño de la entrada n.

Explicación Intuitiva: Si el algoritmo procesa 10 elementos en x tiempo, al procesar 100 elementos tomará aproximadamente 10x tiempo. Es decir, si n crece, el tiempo de ejecución crece en la misma proporción.

$O(n \log n)$ - Tiempo Cuasilineal

$O(n \log n)$ - Tiempo Cuasilineal significa que el tiempo de ejecución crece **casi linealmente**, pero con un pequeño factor adicional de **$\log(n)$** .

Se encuentra en algoritmos más eficientes que **$O(n^2)$** pero un poco más costosos que **$O(n)$** . Es común en algoritmos de ordenamiento eficientes.

✓
0 s



```
def merge(arr1, arr2):
    resultado = []
    i, j = 0, 0

    # Combina los dos arreglos de forma ordenada
    while i < len(arr1) and j < len(arr2):
        if arr1[i] < arr2[j]:
            resultado.append(arr1[i])
            i += 1
        else:
            resultado.append(arr2[j])
            j += 1

    # Añade los elementos restantes de arr1 o arr2
    resultado.extend(arr1[i:])
    resultado.extend(arr2[j:])

    return resultado

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    medio = len(arr) // 2
    izquierda = merge_sort(arr[:medio])
    derecha = merge_sort(arr[medio:])
    return merge(izquierda, derecha)
```

Esta es la función principal que implementa el algoritmo **Merge Sort**, un algoritmo de ordenamiento de tipo *divide y vencerás*.

Su tarea es dividir el arreglo en subarreglos más pequeños, ordenarlos de forma recursiva y luego combinarlos (fusionarlos) de nuevo en una secuencia ordenada.

Función merge_sort:

Esta es la función principal que implementa el algoritmo Merge Sort, un algoritmo de ordenamiento de tipo divide y vencerás. Su tarea es dividir el arreglo en subarreglos más pequeños, ordenarlos de forma recursiva y luego combinarlos (fusionarlos) de nuevo en una secuencia ordenada.

Pasos de merge_sort:

Base de la recursión: Si el tamaño del arreglo es 1 o menor ($\text{len}(\text{arr}) \leq 1$), significa que no es necesario ordenar nada, por lo que simplemente se devuelve el arreglo tal como está.

División del arreglo: Si el arreglo tiene más de un elemento, se divide en dos subarreglos:

- `medio = len(arr) // 2`: Aquí se calcula el punto medio del arreglo, que se usa para dividirlo en dos mitades.
- `izquierda = merge_sort(arr[:medio])`: Se ordena recursivamente la primera mitad del arreglo.
- `derecha = merge_sort(arr[medio:])`: Se ordena recursivamente la segunda mitad del arreglo.

Fusión: Después de ordenar las dos mitades, se invoca la función merge para combinarlas y obtener un arreglo ordenado. La función merge es clave porque toma dos listas ordenadas y las fusiona en una sola lista ordenada.



Función merge:

La función merge toma dos listas ordenadas (arr1 y arr2) y las fusiona en una lista ordenada. Este es un paso crucial en el algoritmo de Merge Sort, ya que asegura que las sublistas ordenadas se combinen correctamente.

Pasos de merge:

Inicialización de índices: Se crean dos índices, i y j , que se usarán para recorrer los dos arreglos arr1 y arr2, respectivamente.

Comparación de elementos: Mientras haya elementos en ambos arreglos por recorrer (es decir, mientras $i < \text{len}(\text{arr1})$ y $j < \text{len}(\text{arr2})$), se compara el elemento actual de arr1[i] con el de arr2[j]. El menor de los dos se agrega a la lista resultado, y el índice correspondiente (i o j) se incrementa.

Agregar elementos restantes: Una vez que uno de los arreglos ha sido completamente recorrido, los elementos restantes del otro arreglo (que ya están ordenados) se agregan directamente a la lista resultado usando extend.

Retorno: La función devuelve la lista resultado, que es la combinación de las dos listas ordenadas.

Ejemplo:

Supongamos que tienes el arreglo [4, 3, 2, 1]:

1. merge_sort divide este arreglo en [4, 3] y [2, 1].
2. Recursivamente, divide cada mitad en dos: [4], [3] y [2], [1].
3. Luego, fusiona las mitades ordenadas:
 - [4] y [3] se fusionan en [3, 4].
 - [2] y [1] se fusionan en [1, 2].

Finalmente, fusiona las dos listas ordenadas [3, 4] y [1, 2] para obtener [1, 2, 3, 4].

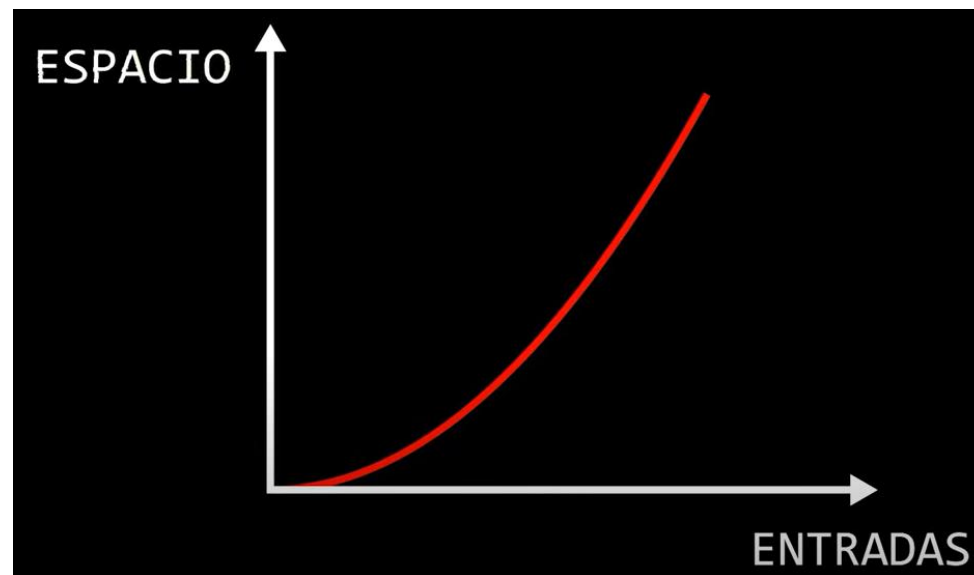
Complejidad:

Tiempo: La complejidad de tiempo de Merge Sort es $O(n \log n)$, donde n es el número de elementos en el arreglo. Esto se debe a que el arreglo se divide en $\log(n)$ pasos y cada fusión de subarreglos lleva un tiempo de $O(n)$.

Espacio: La complejidad espacial es $O(n)$, ya que se requieren arreglos adicionales para realizar las fusiones.

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 $2n$.

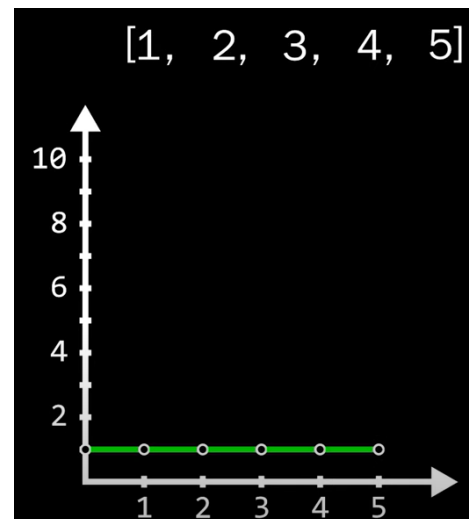
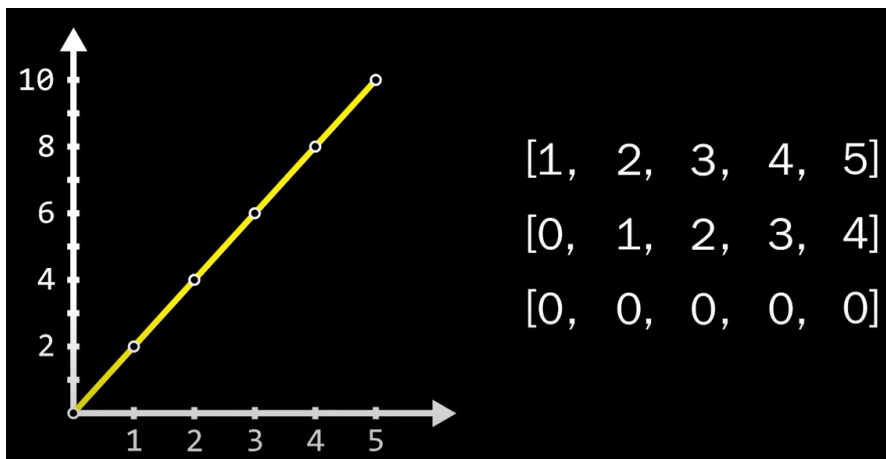
```
[ 5 elementos ] 5  
[ 5 elementos ]  
[ 5 elementos ] 10
```

```
[ 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
```


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
```

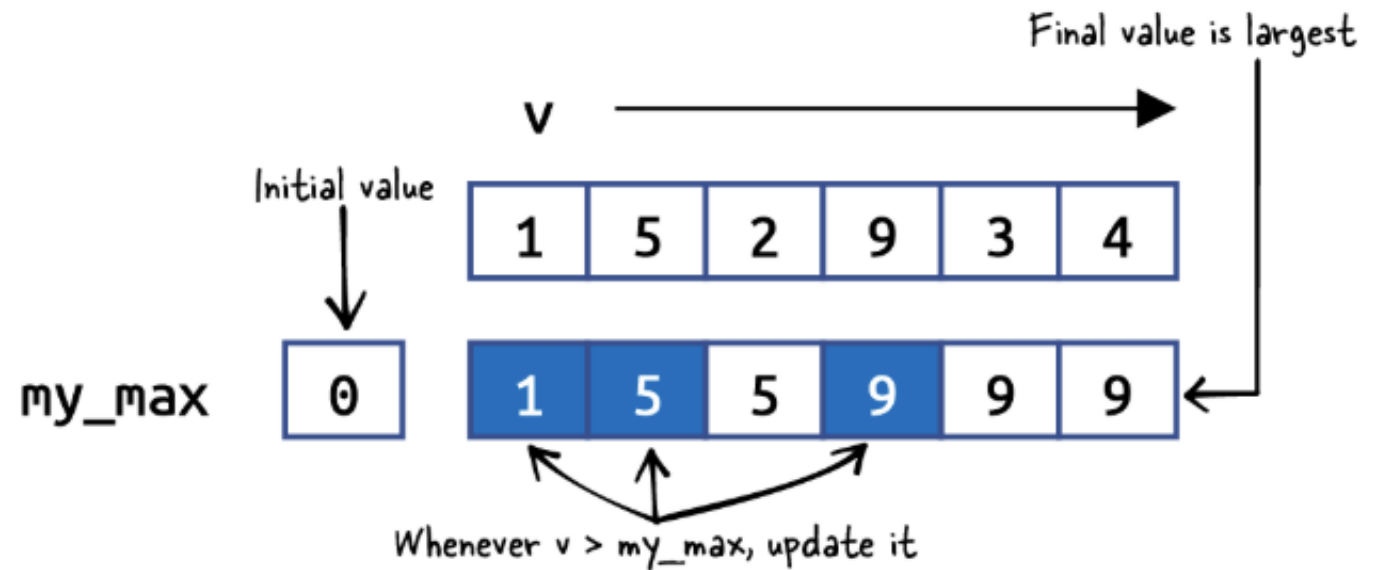


Figura 1-2. Visualización de la ejecución de `flawed()`



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
```

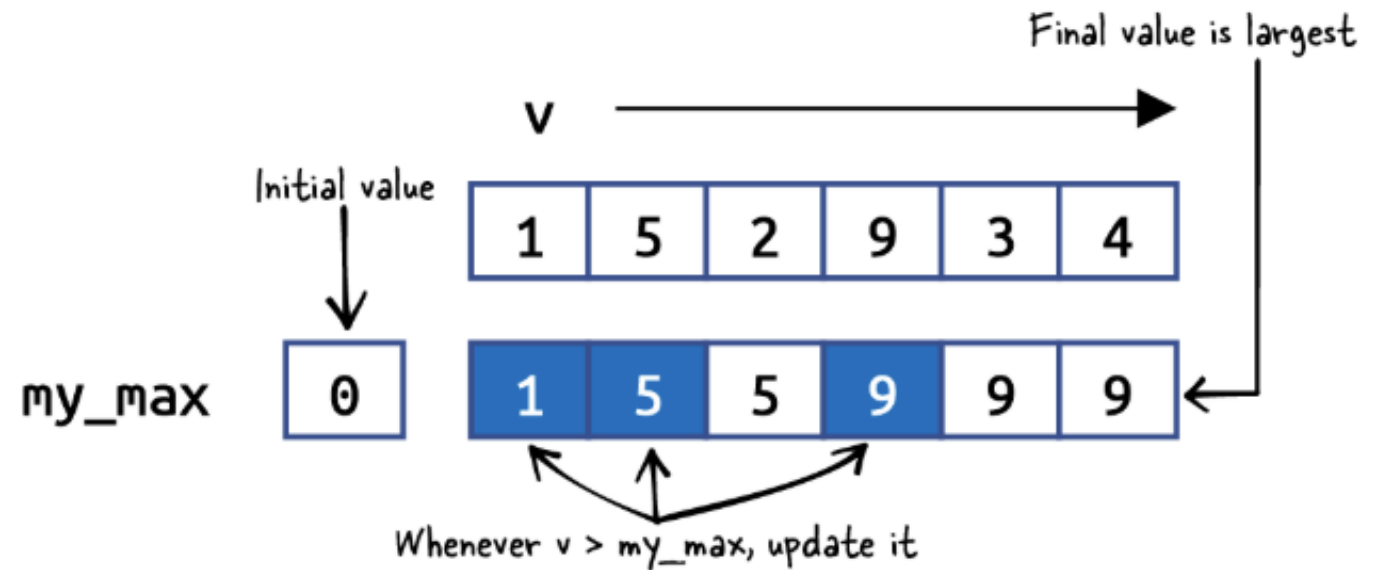


Figura 1-2. Visualización de la ejecución de `largest()`

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
```

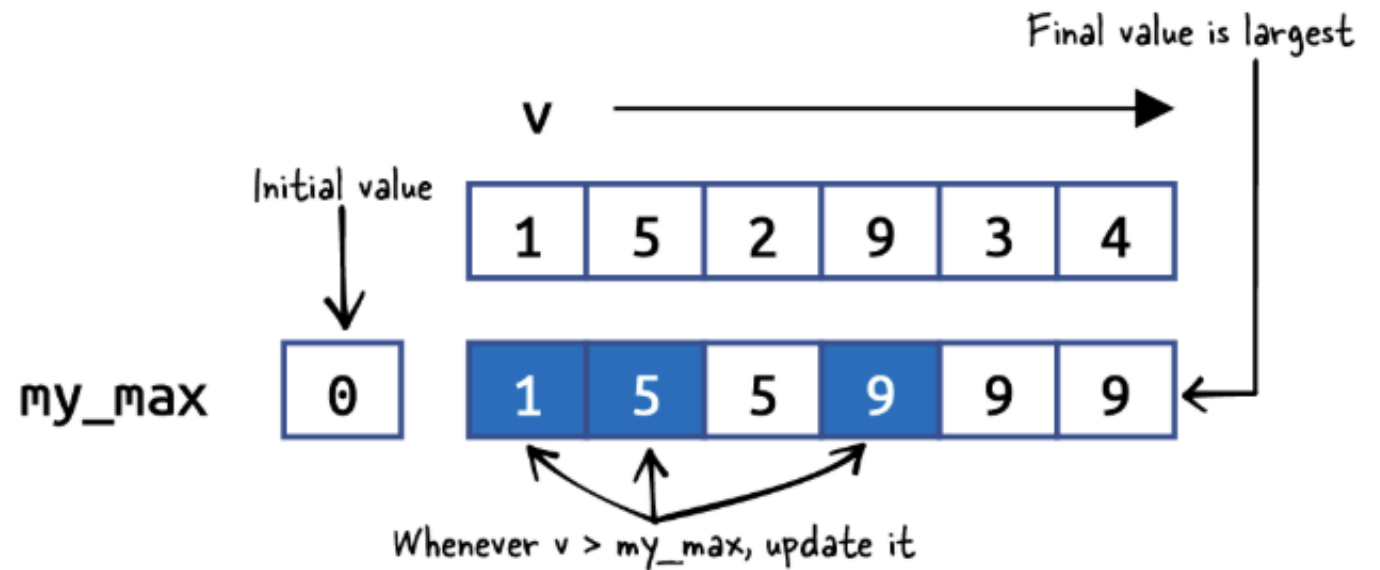


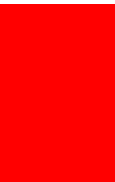
Figura 1-2. Visualización de la ejecución de `largest()`



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
```

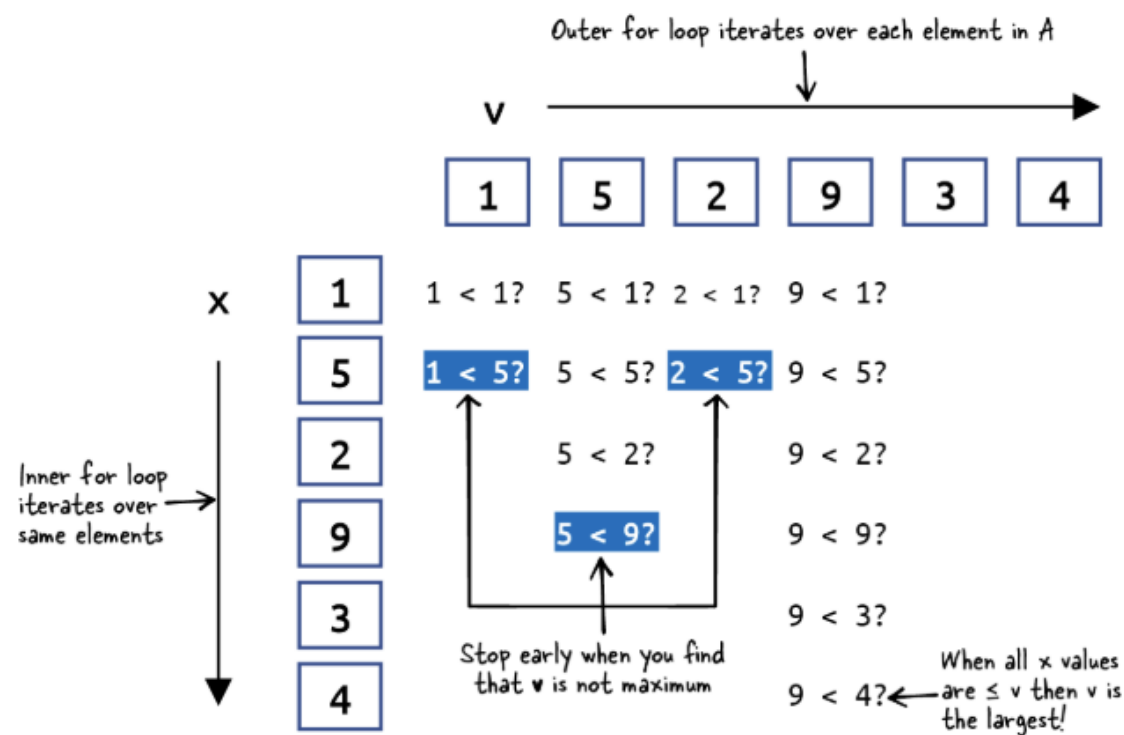
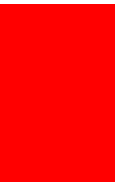


Figura 1-3. Visualización de la ejecución de `alternate()`



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?

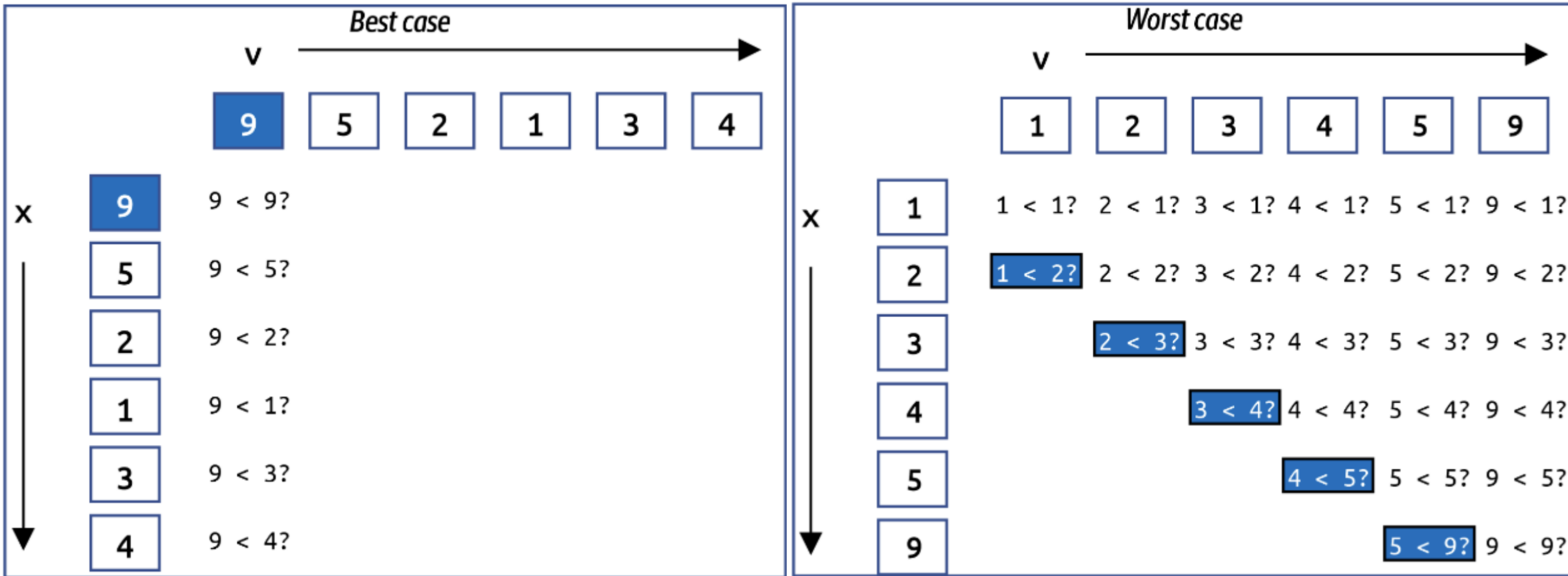
Un problema de este tipo se consideraría el mejor caso para `alternate()`. Por ejemplo, si el primer valor de **A** es el mayor de todos los N valores, entonces el número total de llamadas a menos que es siempre N.

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

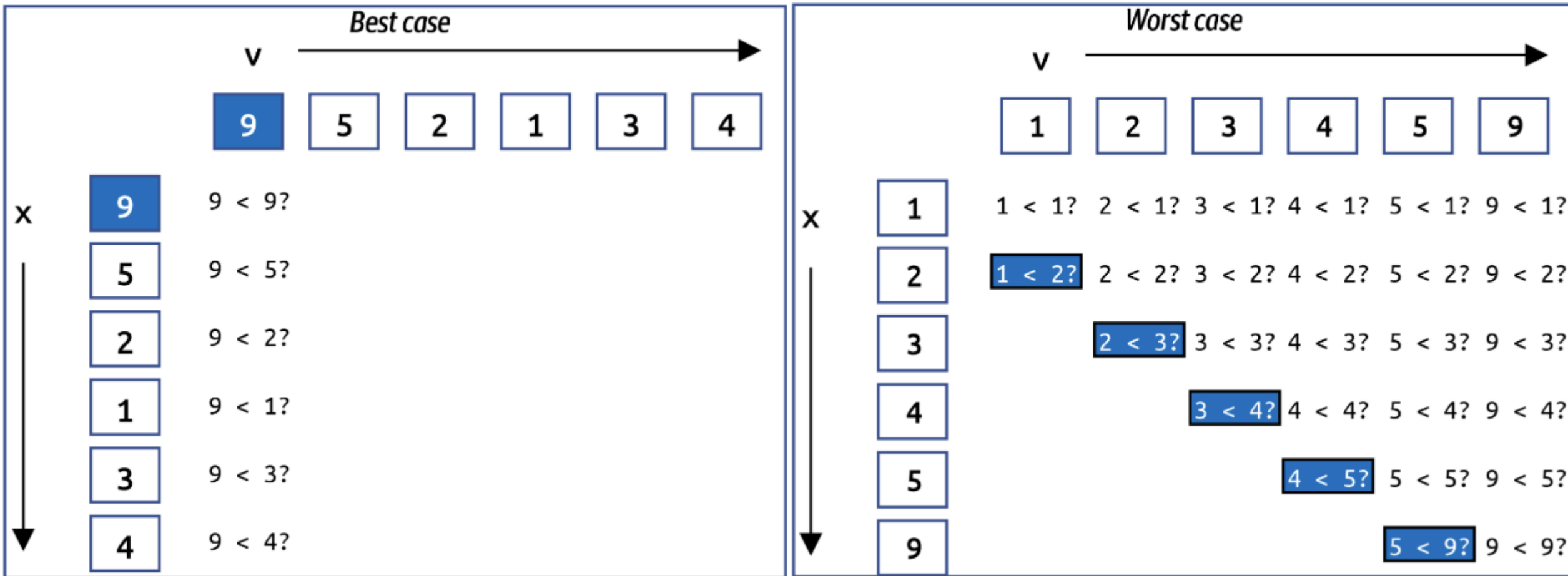
Intentemos identificar el peor caso para `alternate()` que requiera el mayor número de llamadas a menos que. Más que asegurarnos de que el valor mayor es el último de A, en el peor de los casos para `alternate()`, los valores de A deben aparecer en orden ascendente.

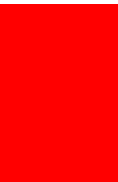
visualiza el mejor caso en la parte superior donde $p = [9, 5, 2, 1, 3, 4]$ y el peor caso en la parte inferior donde $p = [1, 2, 3, 4, 5, 9]$.



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



1

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