Introducción a la Programación

Prof. Agustín Gravano

Primer semestre de 2022

Clase teórica 18: Algoritmos de búsqueda

Problema de búsqueda

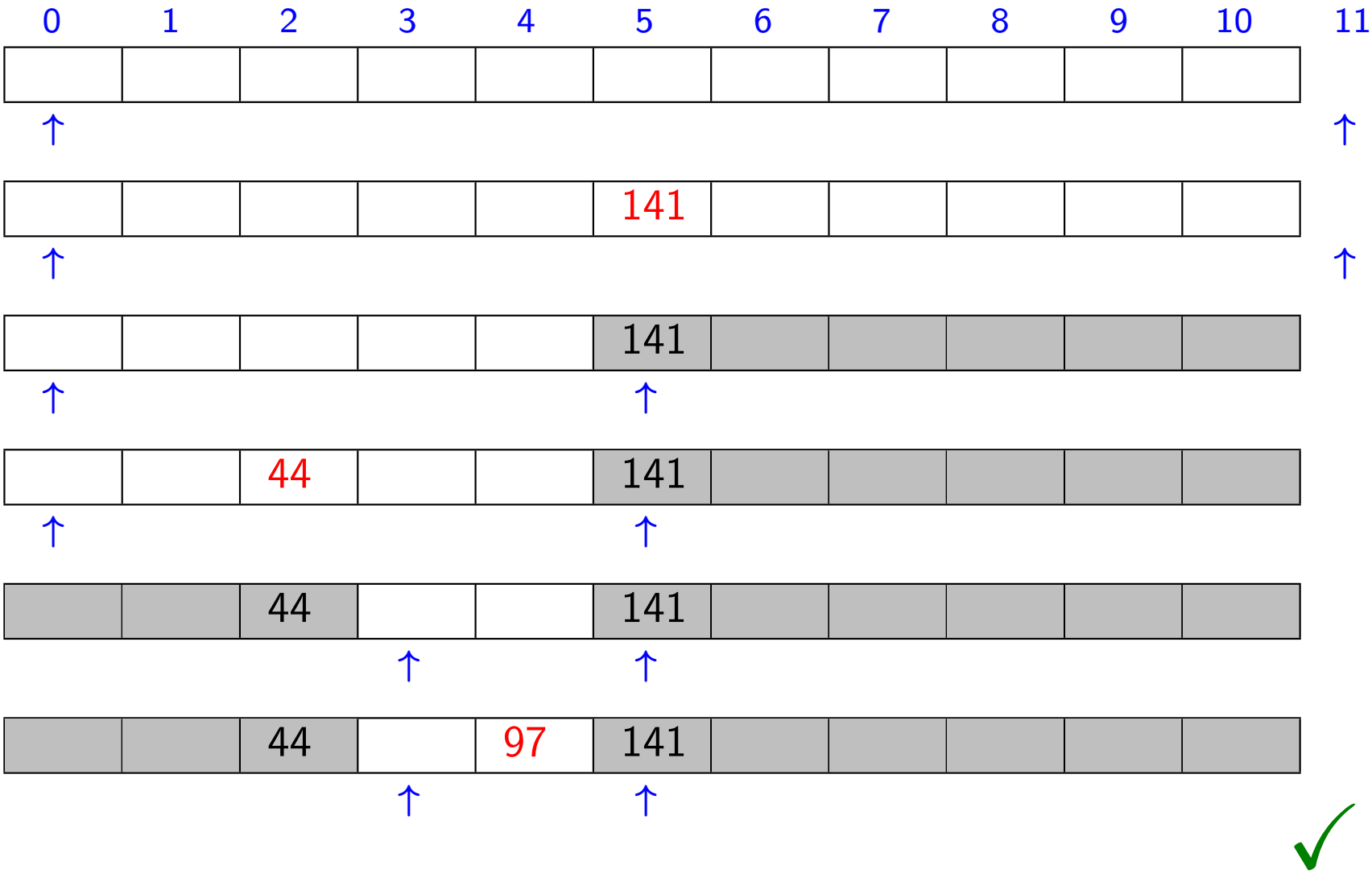
Problema: Dados una lista de enteros `A` y un entero `x`, determinar si `x` aparece al menos una vez en `A`.

```
1  def está(x:int, A:List[int]) -> bool:
2      ''' Requiere: nada.
3          Devuelve: True si x está al menos una vez en la lista A;
4                    False en caso contrario.
5      '''
6      for elem in A:
7          if elem == x:
8              return True
9      return False
```

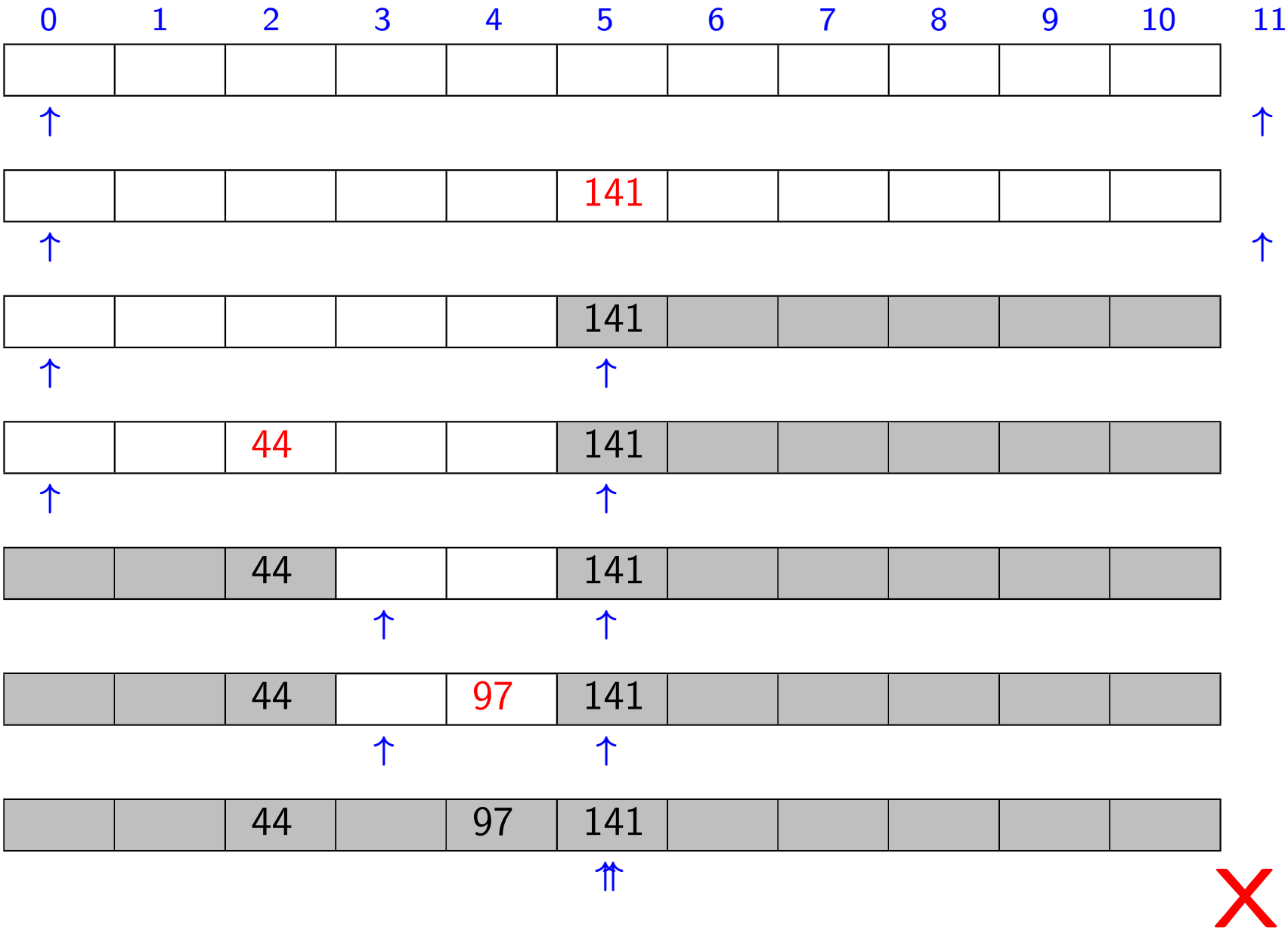
¿Y si la lista `A` está ordenada?

Este algoritmo resuelve el problema, pero sería bueno sacar provecho de que la lista está ordenada.

Busquemos el número **97** en una lista ordenada



Busquemos el número **100** en una lista ordenada



Problema de búsqueda en una lista ordenada

Problema: Dados una lista de enteros `A` ordenada de menor a mayor, y un entero `x`, determinar si `x` aparece al menos una vez en `A`.

Algoritmo de búsqueda binaria:

Empezar considerando la lista `A` completa, y repetir:

a) Buscar el punto medio (aprox.) de la (sub)lista actual.
b) Si justo en el punto medio está `x`, devolver `Verdadero`.
c) Partir la lista en el punto medio, en dos mitades (aprox.).
d) Continuar la búsqueda en la única mitad que puede contener a `x`.
   (Ejercicio: pensar por qué sólo una mitad puede contener a `x`.)

Indefectiblemente, llega un momento en que la lista ya no puede ser dividida, porque nos quedamos sin elementos. Devolver `Falso`.
(Ejercicio: pensar por qué no podríamos habernos pasado por alto a `x`.)

¿Qué complejidad tiene este algoritmo? La cantidad de pasos es proporcional al logaritmo de `len(A)`. ($log_2\, n$ = cuántas veces se puede dividir n por 2.)

Algoritmo de búsqueda binaria

Problema: Dados una lista ordenada de enteros `A` y un entero `x`, determinar si `x` está en `A`.

```
1  def está(x:int, A:List[int]) -> bool:
2      ''' Requiere: la lista A está ordenada en forma creciente.
3          Devuelve: True si x está al menos una vez en la lista A;
4                    False en caso contrario.
5      '''
6      izq:int = 0                                  O(1)
7      der:int = len(A)                             O(1)
8      while izq < der:          Condición: O(1)    ¿Cuántas iteraciones?
9          med:int = (izq+der) // 2                 O(1)
10         if A[med] == x:                          O(1)
11             return True                          O(1)
12         elif A[med] < x:                         O(1)
13             izq = med + 1                        O(1)
14         else:
15             der = med                            O(1)
16     return False                                 O(1)
```

Algoritmo de búsqueda binaria

¿Cuántas iteraciones se ejecutan (en el peor caso)?

El peor caso de este algoritmo es que `x` no está en la lista `A`.

Como vimos, la distancia entre `izq` y `der` se reduce aproximadamente a la mitad en cada iteración.

Esta distancia empieza siendo `len(A)`; después `len(A)//2`, después `len(A)//4`, después `len(A)//8`, después `len(A)//16`, ... así hasta llegar a 0, cuando se niega la condición del while.

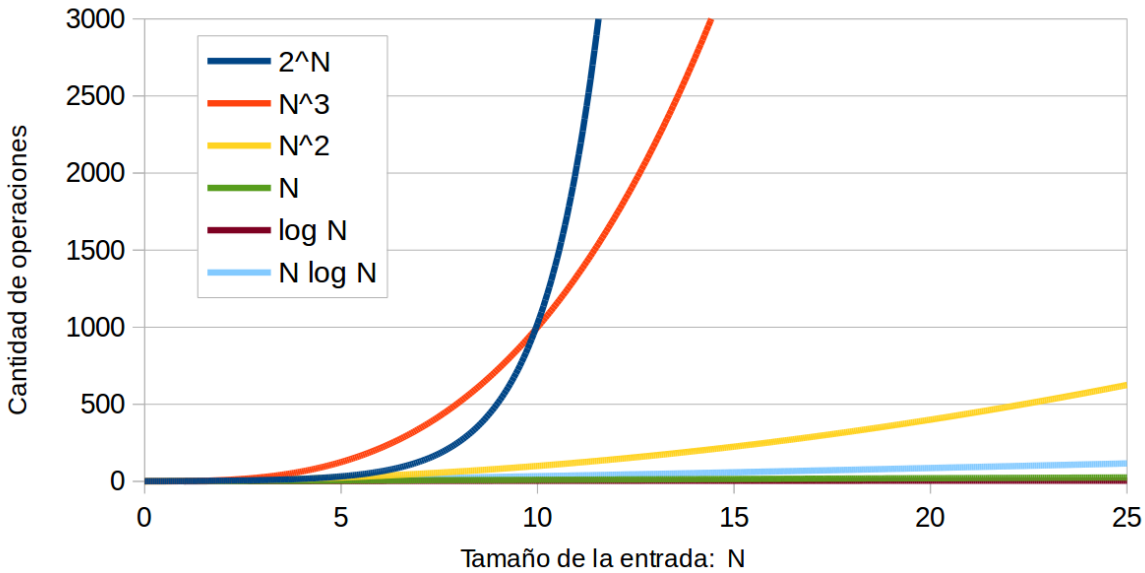
Esto lo podemos hacer a lo sumo $\lfloor \log_2(\texttt{len(A)}) \rfloor + 1$ veces.

Entonces, el ciclo se ejecuta $O(\log_2(\texttt{len(A)}))$ veces.

Por eso, decimos que el algoritmo de búsqueda binaria tiene complejidad logarítmica.

Consideraciones finales

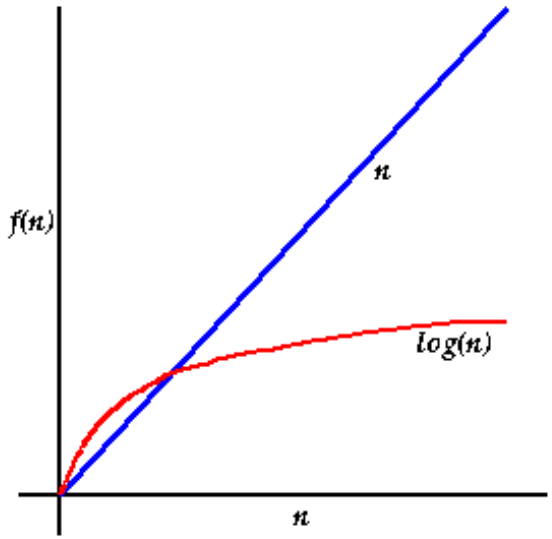
Primera conclusión: debemos prestar atención a la complejidad algorítmica de los algoritmos que escribimos.



Pero no es lo único que tenemos que considerar.

¡La dificultad del código también es un factor importante!

Simple y Lento vs. Complejo y Rápido



¿Cuán importante es la diferencia entre dos órdenes de complejidad?

¿Se justifica el esfuerzo (y el riesgo) de implementar un algoritmo más elaborado para ahorrar tiempo de cómputo?

Depende de nuestro contexto...

- ¿Cuál será el tamaño de la entrada? (PyME vs. Facebook)
- ¿Cuántas veces vamos a ejecutar el programa? (1-2 veces vs. millones de veces por día)
- A veces alcanza con implementar un algoritmo simple y lento.
- Otras veces conviene invertir tiempo y trabajo en implementar un algoritmo más complejo y más rápido.

Repaso de la clase de hoy

- Complejidad algorítmica.
- Búsqueda lineal y búsqueda binaria.

Con lo visto, ya pueden resolver la Guía de Ejercicios 7 completa.

Bibliografía: capítulo 2 de "The Algorithm Design Manual", S. Skiena, 2da edición, Springer (disponible en el campus).