Fundamentos de Algoritmos

Profs. Agustín Gravano y Hernán Czemerinski

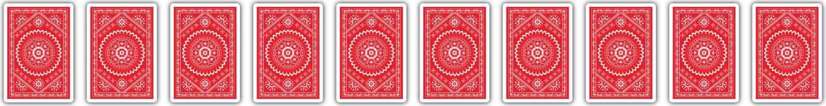
Primer Semestre de 2020

Clase 5: Complejidad de algoritmos

Problema 1: Contar ases

¿Cuántos pasos demandará contar los ases en estas 5 cartas?

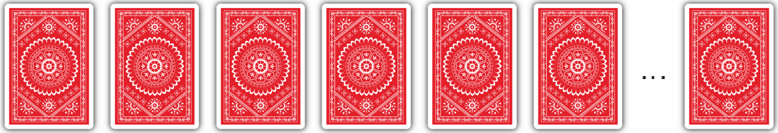
¿Y si tenemos que contar los ases en 10 cartas?



¿Y en 50 cartas? (supongamos que provienen de varios mazos)

Problema 1: Contar ases

En general, ¿cuántos pasos demanda contar ases en N cartas?



La cantidad de pasos necesarios crece en forma lineal respecto del tamaño de la entrada del problema (la cantidad de cartas).

Problema 1: Contar ases - Posible algoritmo

```
1 Contar ases en N cartas:
2     cant = 0
3     i = 0
4     mientras i < N:
5         dar vuelta la carta i
6         si la carta es un as:
7             cant = cant + 1
8         i = i + 1
9     devolver cant
```

Las líneas 5-8 son lo que veníamos llamando un “paso”.

La ejecución individual de cada “paso” no depende de N.

El ciclo repite esas operaciones N veces (i va de 0 a $\mathrm{N}-1$).

Decimos que el algoritmo tiene $\mathrm{O}(\mathrm{N})$, o bien “orden lineal”.

Contar las apariciones de un elemento en una lista

```
1 def contar(elem, lista):
2   cant = 0                  O(1)
3   i = 0                     O(1)
4   while i < len(lista):     len(lista) iteraciones
5      if lista[i]==elem:     O(1)
6         cant = cant + 1     O(1)
7      i = i + 1              O(1)
8   return cant               O(1)
```

Importante: Cada operación de las líneas 5-7 va a ejecutarse en cierto tiempo fijo, que no depende de `len(lista)`. Son operaciones simples: expresiones de tipos básicos, accesos a variables, etc. Decimos que tienen O(1), u "orden constante".

El `while` repite `len(lista)` veces ese bloque de operaciones constantes. Entonces, el algoritmo tiene O(`len(lista)`).

Contar las apariciones de un elemento en una lista

El mismo razonamiento vale para este código parecido:

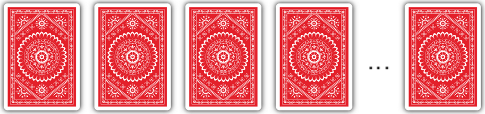
```
1 def contar(elem, lista):
2   cant = 0                      O(1)
3   for x in lista:               len(lista) iteraciones
4     if x==elem:                 O(1)
5       cant = cant + 1           O(1)
6   return cant                   O(1)
```

El `for` ejecuta `len(lista)` veces las líneas 4-5, que consisten en operaciones simples (todas O(1)).

Por lo tanto, este algoritmo también tiene O(`len(lista)`).

Ejercicio

¿Cuántos pasos demandará determinar, en el peor caso, si el as de corazones (A♡) está presente entre N cartas?



1. Escribir el algoritmo en pseudocódigo, y decir cuál es su orden de complejidad (en el peor caso).

2. Escribir en Python una función `buscar(elem, lista)`, que devuelva `True` si `elem` está en `lista`, y `False` en caso contrario. Decir cuál es su orden de complejidad.

Problema 2: Determinar el palo

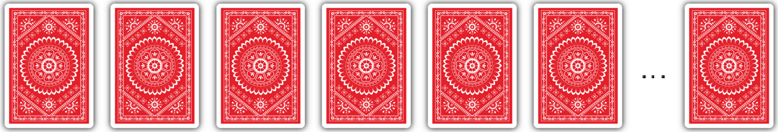
Nos dicen que estas 5 cartas tienen el mismo palo (♡, ♣, ♢ o ♠).
¿Cuántos pasos demandará averiguar de qué palo se trata?

¿Y lo mismo, pero con 10 cartas?

¿Y con 50 cartas?

Problema 2: Determinar el palo

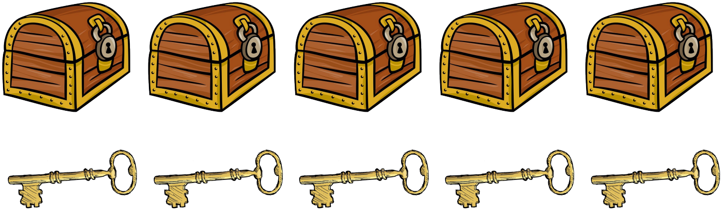
En general, si nos dicen que las N cartas tienen el mismo palo, ¿cuántos pasos demandará averiguar de qué palo se trata?



Siempre alcanza con un solo paso: mirar el palo de la primera carta (o de cualquier otra). Esto es independiente de la cantidad de cartas.

Por eso, decimos que la cantidad de pasos es constante respecto del tamaño de la entrada del problema: O(1).

Problema 3: Cofres y llaves



Tenemos N cofres cerrados con llave y N llaves necesarias para abrirlos, pero no sabemos cuál llave abre cuál cofre.

¿Cuántos pasos llevará (en el peor caso) abrir todos los cofres?

Problema 3: Cofres y llaves - Posible algoritmo

```
1 Abrir los N cofres:
2     i = 0                                      O(1)
3     mientras i < N:                            N iteraciones
4         j = buscar la llave para el cofre i    O(?)
5         abrir el cofre i con la llave j        O(1)
6         i = i + 1                              O(1)
```

Las líneas 2, 5 y 6 se ejecutan en tiempo constante.

El ciclo se repite N veces (la cantidad de cofres).

¿Pero qué pasa con la línea 4? Buscar la llave correcta para el cofre i depende de la cantidad de llaves (también N). Por eso, esa línea no tiene O(1).

Para analizar la complejidad de la línea 4, veamos un posible algoritmo que la implemente.

Problema 3: Cofres y llaves - Posible algoritmo

```
1 Buscar la llave que abre el cofre c:
2     llave_correcta = ?                          O(1)
3     j = 0                                       O(1)
4     mientras j < N:                     N iteraciones
5         si la llave j abre el cofre c:          O(1)
6             llave_correcta = j                  O(1)
7         j = j + 1                               O(1)
8     devolver llave_correcta                     O(1)
```

Las líneas 2, 3, 5, 6, 7 y 8 se ejecutan en tiempo constante.

El ciclo se repite N veces (la cantidad de llaves).

Entonces, buscar la llave correcta para un cofre tiene O(N).
(Notar que es similar a buscar el A♡ en un mazo de cartas.)

Problema 3: Cofres y llaves - Posible algoritmo

```
Abrir los N cofres:
    i = 0                                         O(1)
    mientras i < N:                      N iteraciones
        j = buscar la llave para el cofre i       O(N)
        abrir el cofre i con la llave j           O(1)
        i = i + 1                                 O(1)
```

Volviendo al algoritmo principal, tenemos que el ciclo ejecuta N veces una operación que tiene O(N).

¿Cuánto tiempo demanda repetir 10 veces una tarea de 5 minutos? Claramente, $10 \times 5 = 50$ minutos.

Por lo tanto, el algoritmo principal tiene $\mathrm{O}(\mathrm{N}\times\mathrm{N}) = \mathrm{O}(\mathrm{N}^2)$.

Para abrir todos los cofres, la cantidad de operaciones crece cuadráticamente con la cantidad de llaves y cofres.

Ejercicio: Determinar la complejidad de cada función

```
def sumarHasta(A, i):
    suma = 0
    j = 0
    while j < i:
        suma = suma + A[j]
        j = j + 1
    return suma

def cantSumasAnteriores(A):
    cant = 0
    i = 0
    while i < len(A):
        sumaAnt = sumarHasta(A, i)
        if sumaAnt == A[i]:
            cant = cant + 1
        i = i + 1
    return cant
```

Problema de ordenamiento

59	7	388	41	2	280	50	123

Selection sort

Para cada `i` entre `0` y `len(A)`−`1` (inclusive):
  Buscar el menor elemento en `A[i:]`.
  Intercambiarlo con `A[i]`.

0	1	2	3	4	5	6	7
59	7	388	41	2	280	50	123
2	7	388	41	59	280	50	123
2	7	388	41	59	280	50	123
2	7	41	388	59	280	50	123
2	7	41	50	59	280	388	123
2	7	41	50	59	280	388	123
2	7	41	50	59	123	388	280
2	7	41	50	59	123	280	388
2	7	41	50	59	123	280	388

Propiedad invariante: `A[0:i]` está ordenada.

```
def pos_minimo(L):
  min = 0                          O(1)
  for j in range(0, len(L)):       len(L) iteraciones
    if L[j] < L[min]:              O(1)
      min = j                      O(1)
  return min                       O(1)
```

Esto ejecuta $len(L)$ veces operaciones de orden constante.
Entonces, `pos_minimo(L)` es lineal respecto de `len(L)`.

```
def selection_sort(A):
  for i in range(0, len(A)):          len(A) iteraciones
    min = pos_minimo(A[i:]) + i       O(len(A))
    aux = A[i]                        O(1)
    A[i] = A[min]                     O(1)
    A[min] = aux                      O(1)
```

Esto ejecuta $len(A)$ veces operaciones que tienen $O(len(A))$.
Entonces, `selection_sort(A)` es cuadrático respecto de `len(A)`.

Insertion sort

Para cada `i` entre `0` y `len(A)`$-$`1` (inclusive):

Sacar el elemento `A[i]`.

Insertarlo en la posición correcta en `A[0:i]`.

0	1	2	3	4	5	6	7
59	7	388	41	2	280	50	123
59	7	388	41	2	280	50	123
7	59	388	41	2	280	50	123
7	59	388	41	2	280	50	123
7	41	59	388	2	280	50	123
2	7	41	59	388	280	50	123
2	7	41	59	280	388	50	123
2	7	41	50	59	280	388	123
2	7	41	50	59	123	280	388

Propiedad invariante: `A[0:i]` está ordenada.

```
def pos_primer_mayor(x, L):
  for j in range(0, len(L)):   len(L) iteraciones
    if L[j] > x:               O(1)
      return j                 O(1)
  return len(L)                O(1)
```

`pos_primer_mayor(x, L)` es lineal respecto de `len(L)`.

```
def insertion_sort(A):
  for i in range(0, len(A)):      len(A) iteraciones
    x = A.pop(i)                         O(len(A))
    j = pos_primer_mayor(x, A[0:i])      O(len(A))
    A.insert(j, x)                       O(len(A))
```

`insertion_sort(A)` es cuadrático respecto de `len(A)`.

(¡Cuidado con `A.pop(i)`, `A[0:i]` y `A.insert(j,x)`, que son lineales!)

Complejidad y Ordenamiento

Algoritmos:

- $O(n^2)$: selection, insertion, bubble (ver la guía).
- $O(n \log n)$: merge sort, heapsort.
- Límite teórico para algoritmos basados en comparaciones: $O(n \log n)$.

Bibliografía:

- Aho, Hopcroft & Ullman, "Estructuras de Datos y Algoritmos", Addison-Wesley, 1988.
- Balcazar, "Programación metódica", McGraw-Hill, 1993.

Demos y otras yerbas:

- http://www.sorting-algorithms.com/
- http://www.youtube.com/watch?v=MtcrEhrt_K0
- http://www.youtube.com/watch?v=INHF_5RIxTE

Problema de búsqueda en una lista ordenada

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

Ahora buscamos el número **97** en una lista ordenada:

0	1	2	3	4	5	6	7	8	9	10

					141						
		44			141						
		44		97	141						✓

¿Cuánto va a demorar este algoritmo en relación a `len(A)`?

La cantidad de pasos es proporcional al logaritmo de `len(A)`.

(Obs.: $log_2\, n =$ cuántas veces se puede dividir n por 2.)

Algoritmo de búsqueda binaria

Si la lista está ordenada, en cada paso puedo partir la lista en:

a) la mitad que puede contener el elemento; y
b) la mitad que no puede contenerlo.

Indefectiblemente, se llega a un punto en que la lista ya no puede ser dividida (tiene un solo elemento) y, o bien el elemento es el buscado o no.

Este algoritmo se conoce como búsqueda binaria.

Algoritmo de búsqueda binaria

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

```
def está(x, A):
  izq = 0                       O(1)
  der = len(A)                  O(1)
  while izq < der:              ¿Cuántas iteraciones?
     med = (izq+der) // 2       O(1)
     if A[med] == x:            O(1)
        return True             O(1)
     elif A[med] < x:           O(1)
        izq = med + 1           O(1)
     else:                      O(1)
        der = med               O(1)
  return False                  O(1)
```

Algoritmo de búsqueda binaria

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

Notemos que la distancia entre *izq* y *der* se reduce aproximadamente a la mitad en cada iteración:

Sea $dist = der - izq$ al comienzo de una iteración.

Al final de la iteración, pueden ocurrir dos cosas:

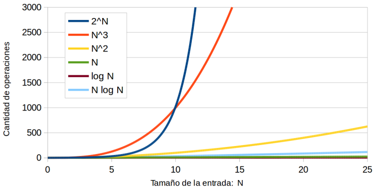
- $dist' = der - \lfloor \frac{izq+der}{2} \rfloor - 1 \approx \lfloor \frac{dist}{2} \rfloor$
- $dist' = \lfloor \frac{izq+der}{2} \rfloor - izq \approx \lfloor \frac{dist}{2} \rfloor$

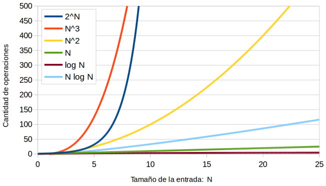
En ambos casos, *dist* termina valiendo aproximadamente la mitad que al principio de la iteración.

Como el ciclo termina cuando $dist \leq 0$, el cuerpo del ciclo se ejecuta aproximadamente $\log_2(len(A))$ veces.

Consideraciones finales

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

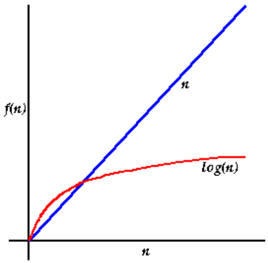




Pero no es lo único que tenemos que considerar.

¡La dificultad del código también es un factor importante!

Búsqueda lineal vs. binaria



¿Cuán importante es la diferencia entre $O(n)$ y $O(\log n)$?

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

Depende de nuestro contexto...

- ¿Cuál es el tamaño del listado en el cual haremos la búsqueda? (PyME vs. ANSES)
- ¿Cuántas veces vamos a necesitar hacer esta búsqueda? (una vez por semana vs. millones de veces por día)

A veces alcanza con implementar un algoritmo fácil y lento; otras veces conviene tomarse el trabajo de implementar un algoritmo más complejo y más rápido.

Repaso de la clase de hoy

- Complejidad temporal de algoritmos.
- Búsqueda lineal y búsqueda binaria.
- Algoritmos de ordenamiento: selection sort, insertion sort.

  Con lo visto, ya pueden resolver la sección 6 de la guía de ejercicios.