Algoritmos Aleatorizados

José Ortiz Bejar

Facultad de Ingeniería Eléctrica
Universidad Michoacana de San Nicolás de Hidalgo

Junio 1, 2022

Contenido

- Ejemplos
- Algoritmo las Vegas y Monte Carlo
- Mas Ejemplos

¿Recuerda el método de Quick-Sort?

¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1   if begin < end
                         2       q = Partition(A, begin, end)
                         3       QuickSort(A, begin, q − 1)
                         4       QuickSort(A, q + 1, end)
```

¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que* v
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que* v

¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que v*
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que v*

2	36	5	21	1	13	11	20	5	4	8

¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que v*
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que v*

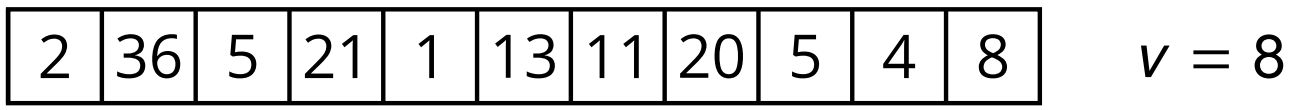
2	36	5	21	1	13	11	20	5	4	8

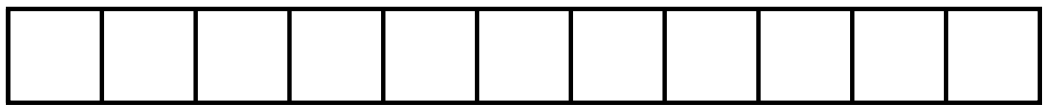
$v = 8$

¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que v*
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que v*





¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1   if begin < end
                         2       q = Partition(A, begin, end)
                         3       QuickSort(A, begin, q − 1)
                         4       QuickSort(A, q + 1, end)
```

- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que v*
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que v*

2	36	5	21	1	13	11	20	5	4	8

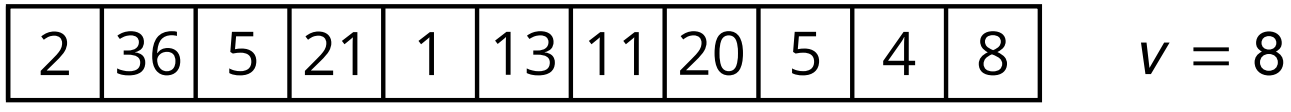
$v = 8$

2	4	1	5	5						

¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que v*
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que v*



¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que v*
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que v*

2	36	5	21	1	13	11	20	5	4	8

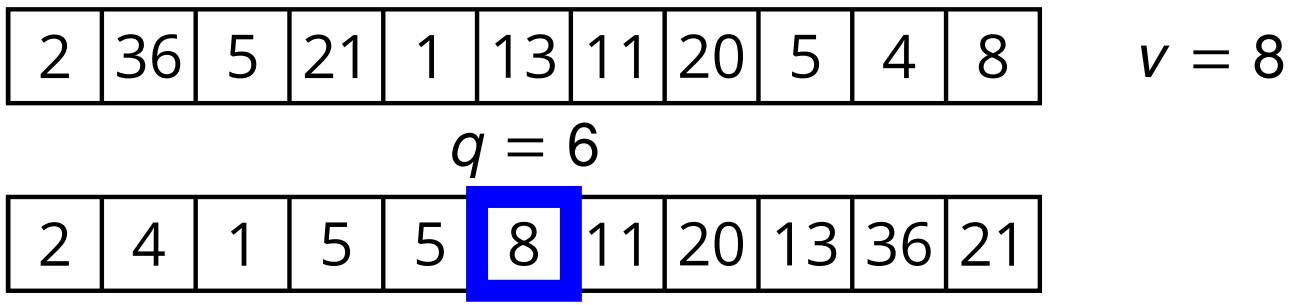
$v = 8$

2	4	1	5	5	8	11	20	13	36	21

¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

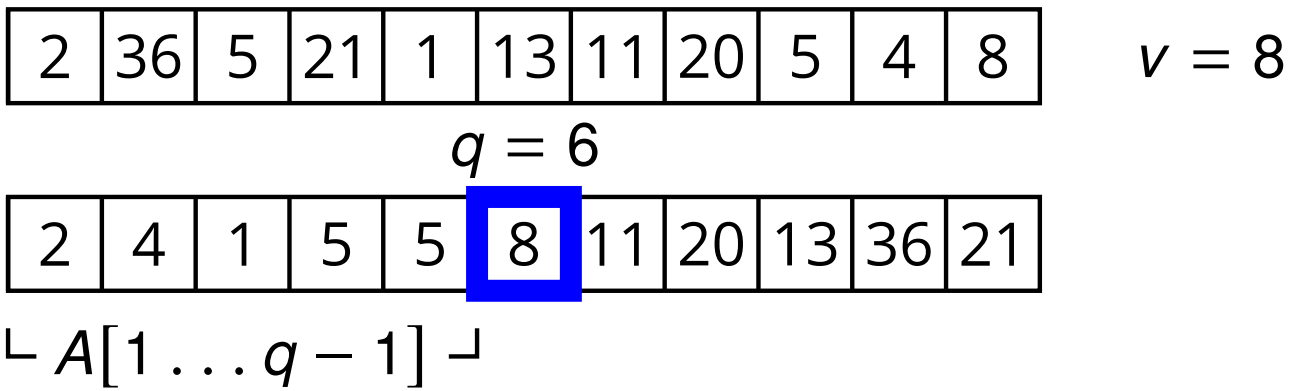
- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que v*
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que v*



¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

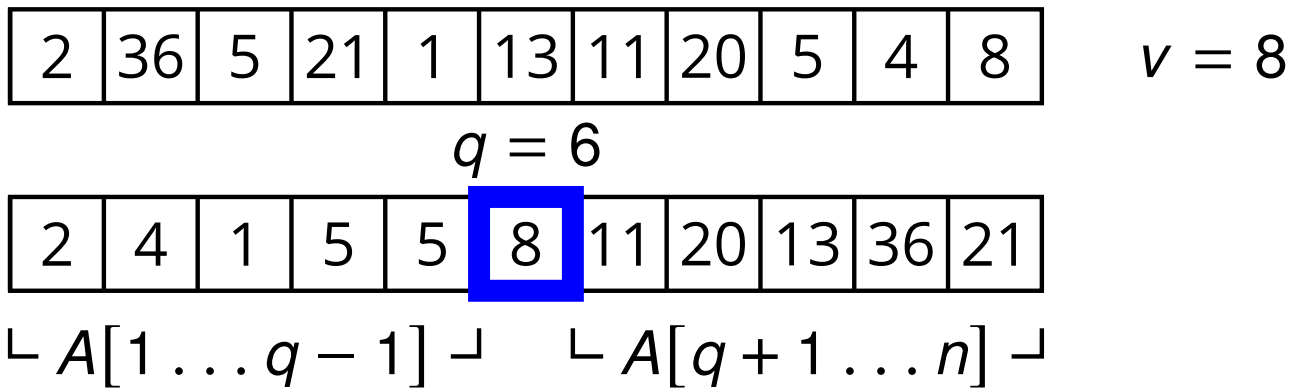
- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que v*
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que v*



¿Recuerda el método de Quick-Sort?

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- Idea: *partir* la secuencia $A[1 \ldots n]$ en tres partes
  - $A[1 \ldots q-1]$ contiene los elementos que son *igual o menores que v*
  - $A[q] = v$ es el "pivote" ($v \in A$)
  - $A[q+1 \ldots n]$ contiene los elementos que son *mayores o iguales que v*



Un clásico del Divide y vencerás

- ***Dividir:*** partir A rn $A[1 \ldots q-1]$ y $A[q+1 \ldots n]$ tal que ...

Un clásico del Divide y vencerás

- ***Dividir:*** partir A rn $A[1 \ldots q-1]$ y $A[q+1 \ldots n]$ tal que ...
- ***Vencer:*** ordenar $A[1 \ldots q-1]$ y $A[q+1 \ldots n]$

Un clásico del Divide y vencerás

- ***Dividir:*** partir A rn $A[1 \ldots q-1]$ y $A[q+1 \ldots n]$ tal que ...
- ***Vencer:*** ordenar $A[1 \ldots q-1]$ y $A[q+1 \ldots n]$
- ***Combinar:*** no hay combinación
  - el algoritmo de *particionado* realiza la tarea

Un clásico del Divide y vencerás

- ***Dividir:*** partir A rn $A[1 \ldots q-1]$ y $A[q+1 \ldots n]$ tal que ...
- ***Vencer:*** ordenar $A[1 \ldots q-1]$ y $A[q+1 \ldots n]$
- ***Combinar:*** no hay combinación
  - el algoritmo de *particionado* realiza la tarea

```
Partition(A, begin, end)
1  q = begin
2  v = A[end]  // elección determinista de v
3  for i = begin to end
4       if A[i] ≤ v
5            swap(A[i], A[q])
6            q = q + 1
7  return q − 1
```

Complejidad de **QuickSort**

```
QuickSort(A, begin, end) 1   if begin < end
                         2       q = Partition(A, begin, end)
                         3       QuickSort(A, begin, q − 1)
                         4       QuickSort(A, q + 1, end)
```

Complejidad de QuickSort

```
QuickSort(A, begin, end) 1   if begin < end
                         2       q = Partition(A, begin, end)
                         3       QuickSort(A, begin, q − 1)
                         4       QuickSort(A, q + 1, end)
```

- *Peor caso:* $q = 1$ or $q = n$

Complejidad de QuickSort

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- *Peor caso:* $q = 1$ or $q = n$
  - al particionar se tranforma un problema P de tamaño n en P de tamaño $n-1$, entonces $T(n) = T(n-1) + \Theta(n)$

Complejidad de QuickSort

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- *Peor caso:* $q = 1$ or $q = n$
  - al particionar se tranforma un problema P de tamaño n en P de tamaño $n-1$, entonces $T(n) = T(n-1) + \Theta(n)$

$$T(n) = \Theta(n^2)$$

Complejidad de QuickSort

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- *Peor caso:* $q = 1$ or $q = n$
  - al particionar se tranforma un problema P de tamaño n en P de tamaño $n-1$, entonces $T(n) = T(n-1) + \Theta(n)$

$$T(n) = \Theta(n^2)$$

- *Mejor caso:* $q = \lceil n/2 \rceil$

Complejidad de QuickSort

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- *Peor caso:* $q = 1$ or $q = n$
  - al particionar se tranforma un problema P de tamaño n en P de tamaño $n-1$, entonces $T(n) = T(n-1) + \Theta(n)$

$$T(n) = \Theta(n^2)$$

- *Mejor caso:* $q = \lceil n/2 \rceil$
  - al particionar transformamos el problema P de tamaño n en *dos* problemas P de tamaño $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil - 1$, respectivamente; entonces tenemos $T(n) = 2T(n/2) + \Theta(n)$

Complejidad de QuickSort

```
QuickSort(A, begin, end) 1  if begin < end
                         2      q = Partition(A, begin, end)
                         3      QuickSort(A, begin, q − 1)
                         4      QuickSort(A, q + 1, end)
```

- *Peor caso:* $q = 1$ or $q = n$
  - al particionar se tranforma un problema P de tamaño n en P de tamaño $n-1$, entonces $T(n) = T(n-1) + \Theta(n)$

$$T(n) = \Theta(n^2)$$

- *Mejor caso:* $q = \lceil n/2 \rceil$
  - al particionar transformamos el problema P de tamaño n en *dos* problemas P de tamaño $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil - 1$, respectivamente; entonces tenemos $T(n) = 2T(n/2) + \Theta(n)$

$$T(n) = \Theta(n \log n)$$

Complejidad de QuickSort

```
QuickSort(A, begin, end) 1   if begin < end
                         2       q = Partition(A, begin, end)
                         3       QuickSort(A, begin, q − 1)
                         4       QuickSort(A, q + 1, end)
```

- *Peor caso:* $q = 1$ or $q = n$ ← *no es tan improbable*
  - al particionar se tranforma un problema P de tamaño n en P de tamaño $n-1$, entonces $T(n) = T(n-1) + \Theta(n)$

$$T(n) = \Theta(n^2)$$

- *Mejor caso:* $q = \lceil n/2 \rceil$
  - al particionar transformamos el problema P de tamaño n en *dos* problemas P de tamaño $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil - 1$, respectivamente; entonces tenemos $T(n) = 2T(n/2) + \Theta(n)$

$$T(n) = \Theta(n \log n)$$

Solución aleatoriza

Solución aleatoriza

- Simple

```
Randomized-Partition(A, begin, end)
1  i = Random(begin, end)
2  swap(A[i], A[end])
3  return Partition(A, begin, end)
```

Solución aleatoriza

- Simple

```
Randomized-Partition(A, begin, end)
1  i = Random(begin, end)
2  swap(A[i], A[end])
3  return Partition(A, begin, end)
```

```
QuickSort(A, begin, end)
1  if begin < end
2      q = Randomized-Partition(A, begin, end)
3      QuickSort(A, begin, q − 1)
4      QuickSort(A, q + 1, end)
```

Otros Ejemplos

Otros Ejemplos

```
Tree-Randomized-Insert(t, z)
 1  if t = nil
 2       return z
 3  r = valor aleatorio de {1, ..., size(t) + 1}
 4  if r = 1                          // Pr[r = 1] = 1/(size(t) + 1)
 5       size(z) = size(t) + 1
 6       return Tree-Root-Insert(t, z)
 7  if key(z) < key(t)
 8       left(t) = Tree-Randomized-Insert(left(t), z)
 9  else right(t) = Tree-Randomized-Insert(right(t), z)
10  size(t) = size(t) + 1
11  return t
```

Idea General

Idea General

- Desarrolla *variantes aleatorizadas* de algoritmos

Idea General

- Desarrolla *variantes aleatorizadas* de algoritmos
- La idea es hace que un algoritmo dado se ejecute con *la mejor complejidad* ***con probabilidad alta*** *sobre cualquier entrada*

Idea General

- Desarrolla *variantes aleatorizadas* de algoritmos
- La idea es hace que un algoritmo dado se ejecute con *la mejor complejidad* ***con probabilidad alta*** *sobre cualquier entrada*
- El algoritmo no debería modificarse significativamente,

Idea General

- Desarrolla *variantes aleatorizadas* de algoritmos
- La idea es hace que un algoritmo dado se ejecute con *la mejor complejidad* ***con probabilidad alta*** *sobre cualquier entrada*
- El algoritmo no debería modificarse significativamente,

- Sin embargo, es posible hacer mucho más mediante el uso de algoritmos aleatorizados...

Ejemplo simple

- Problema: *encontrar un bit cero*
  - *Entrada:* un arreglo A, de n bits, que contiene aproximadamente el mismo número de more de 1s y 0s (*peso de hamming* es estrictamente $n/2$)
  - *Salida:* i tal que $A[i] = 0$, o NIL si no hay ceros

Ejemplo simple

- Problema: *encontrar un bit cero*
  - *Entrada:* un arreglo A, de n bits, que contiene aproximadamente el mismo número de more de 1s y 0s (*peso de hamming* es estrictamente $n/2$)
  - *Salida:* i tal que $A[i] = 0$, o NIL si no hay ceros

- Solución trivial

```
Find-A-Zero-Bit(A)  1  for i = 1 to |A|
                    2      if A[i] == 0
                    3          return i
                    4  return nil
```

Ejemplo simple

- Problema: *encontrar un bit cero*
  - *Entrada:* un arreglo A, de n bits, que contiene aproximadamente el mismo número de more de 1s y 0s (*peso de hamming* es estrictamente $n/2$)
  - *Salida:* i tal que $A[i] = 0$, o NIL si no hay ceros

- Solución trivial

```
Find-A-Zero-Bit(A) 1  for i = 1 to |A|
                   2      if A[i] == 0
                   3          return i
                   4  return nil
```

- Inconvenientes
  - Si A está ordenado de forma descendente (primero los bits 1, seguidos de los bits 0)

Ejemplo simple

- Problema: *encontrar un bit cero*
  - *Entrada:* un arreglo A, de n bits, que contiene aproximadamente el mismo número de more de 1s y 0s (*peso de hamming* es estrictamente $n/2$)
  - *Salida:* i tal que $A[i] = 0$, o NIL si no hay ceros

- Solución trivial

```
Find-A-Zero-Bit(A) 1  for i = 1 to |A|
                   2      if A[i] == 0
                   3          return i
                   4  return nil
```

- Inconvenientes
  - Si A está ordenado de forma descendente (primero los bits 1, seguidos de los bits 0)
  - cualquier algoritmo de búsqueda *determinista* es vulnerable

Un algoritmo aleatorizado

Un algoritmo aleatorizado

- Toma uno

```
Randomized-Find-A-Zero-Bit1(A)
1  repeat
2      i = Random(1, |A|)
3  until A[i] == 0
4  return i
```

Un algoritmo aleatorizado

- Toma uno

```
Randomized-Find-A-Zero-Bit1(A)
1  repeat
2      i = Random(1, |A|)
3  until A[i] == 0
4  return i
```

  - Iteraciones esperadas: 2

Un algoritmo aleatorizado

- Toma uno

```
Randomized-Find-A-Zero-Bit1(A)
1  repeat
2      i = Random(1, |A|)
3  until A[i] == 0
4  return i
```

  - Iteraciones esperadas: 2
  - peor caso:

Un algoritmo aleatorizado

- Toma uno

```
Randomized-Find-A-Zero-Bit1(A)
1  repeat
2      i = Random(1, |A|)
3  until A[i] == 0
4  return i
```

  - Iteraciones esperadas: 2
  - peor caso: no acotado

Un algoritmo aleatorizado

- Toma uno

```
Randomized-Find-A-Zero-Bit1(A)
1  repeat
2      i = Random(1, |A|)
3  until A[i] == 0
4  return i
```

  - Iteraciones esperadas: 2
  - peor caso: no acotado

Un algoritmo aleatorizado

- Toma uno

```
Randomized-Find-A-Zero-Bit1(A)
1  repeat
2      i = Random(1, |A|)
3  until A[i] == 0
4  return i
```

  - Iteraciones esperadas: 2
  - peor caso: no acotado

Un algoritmo aleatorizado

- Toma uno

```
Randomized-Find-A-Zero-Bit1(A)
1  repeat
2      i = Random(1, |A|)
3  until A[i] == 0
4  return i
```

  - Iteraciones esperadas: 2
  - peor caso: no acotado

Un algoritmo aleatorizado

- Toma uno

```
Randomized-Find-A-Zero-Bit1(A)
1  repeat
2      i = Random(1, |A|)
3  until A[i] == 0
4  return i
```

  - Iteraciones esperadas: 2
  - peor caso: no acotado
  - ***Las Vegas***

Un algoritmo aleatorizado

- Toma 2

```
Randomized-Find-A-Zero-Bit2(A)
1  for j = 1 to k
2      i = Random(1, |A|)
3      if A[i] == 0
4          return i
5  return nil
```

Un algoritmo aleatorizado

- Toma 2

```
Randomized-Find-A-Zero-Bit2(A)
1   for j = 1 to k
2       i = Random(1, |A|)
3       if A[i] == 0
4           return i
5   return nil
```

  - numero de iteraciones en el peor caso i: k

Un algoritmo aleatorizado

- Toma 2

```
Randomized-Find-A-Zero-Bit2(A)
1   for j = 1 to k
2       i = Random(1, |A|)
3       if A[i] == 0
4           return i
5   return nil
```

  - numero de iteraciones en el peor caso i: k
  - peor-caso: resultado incorrecto!

Un algoritmo aleatorizado

- Toma 2

```
Randomized-Find-A-Zero-Bit2(A)
1   for j = 1 to k
2       i = Random(1, |A|)
3       if A[i] == 0
4           return i
5   return nil
```

  - numero de iteraciones en el peor caso i: k
  - peor-caso: resultado incorrecto!

Un algoritmo aleatorizado

- Toma 2

```
Randomized-Find-A-Zero-Bit2(A)
1   for j = 1 to k
2       i = Random(1, |A|)
3       if A[i] == 0
4           return i
5   return nil
```

  - numero de iteraciones en el peor caso i: k
  - peor-caso: resultado incorrecto!

Un algoritmo aleatorizado

- Toma 2

```
Randomized-Find-A-Zero-Bit2(A)
1   for j = 1 to k
2       i = Random(1, |A|)
3       if A[i] == 0
4           return i
5   return nil
```

  - numero de iteraciones en el peor caso i: k
  - peor-caso: resultado incorrecto!

Un algoritmo aleatorizado

- Toma 2

```
Randomized-Find-A-Zero-Bit2(A)
1   for j = 1 to k
2       i = Random(1, |A|)
3       if A[i] == 0
4           return i
5   return nil
```

  - numero de iteraciones en el peor caso i: k
  - peor-caso: resultado incorrecto!

Un algoritmo aleatorizado

- Toma 2

```
Randomized-Find-A-Zero-Bit2(A)
1   for j = 1 to k
2       i = Random(1, |A|)
3       if A[i] == 0
4           return i
5   return nil
```

  - numero de iteraciones en el peor caso i: k
  - peor-caso: resultado incorrecto!

Un algoritmo aleatorizado

- Toma 2

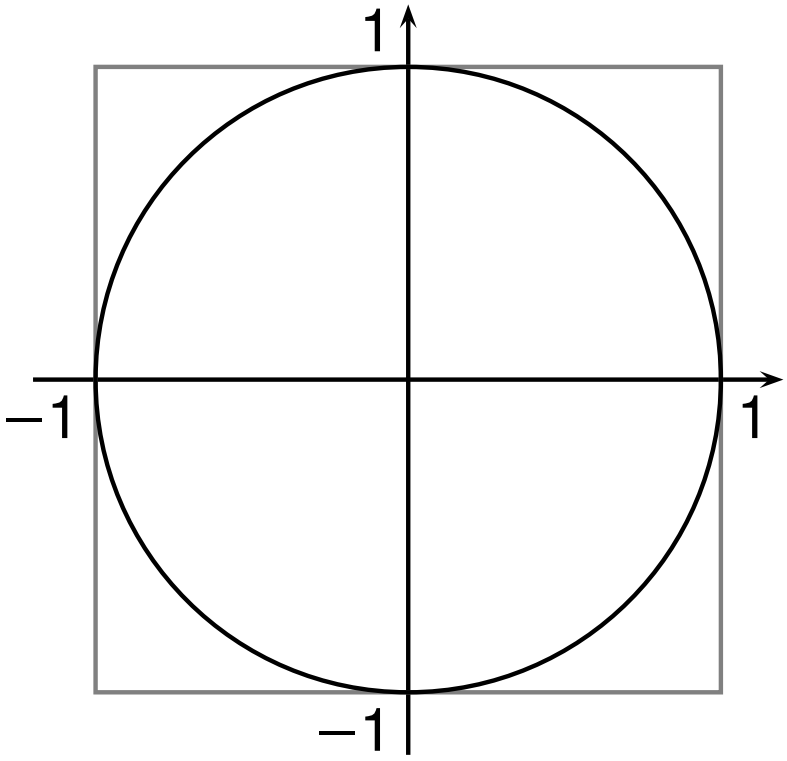
```
Randomized-Find-A-Zero-Bit2(A)
1  for j = 1 to k
2      i = Random(1, |A|)
3      if A[i] == 0
4          return i
5  return nil
```

  - numero de iteraciones en el peor caso i: k
  - peor-caso: resultado incorrecto!
  - ***Monte Carlo***

Algoritmo de Monte Carlo simple

- Problema: *compute la superficie de un disco unitario*
  - suponmga que conociera el valor de π— y no conociera que $S = \pi r^2$, pero sabe que

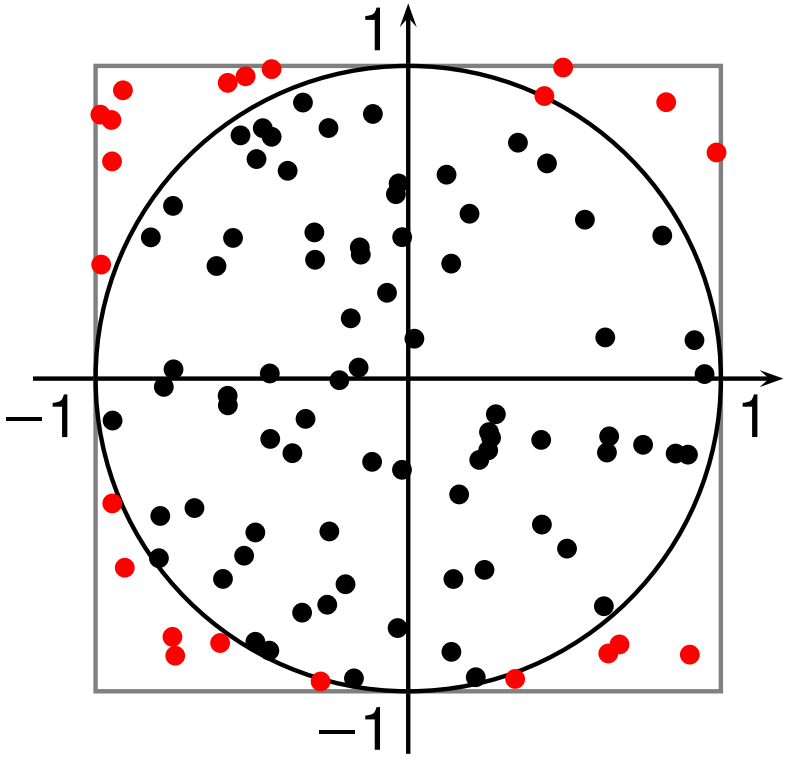
$$(x, y) \quad \text{is} \quad \begin{cases} \textit{fuera} \text{ del circulo si} & x^2 + y^2 > 1 \\ \textit{dentro} \text{ del circulo si} & x^2 + y^2 \leq 1 \end{cases}$$



Algoritmo de Monte Carlo simple

- Problema: *compute la superficie de un disco unitario*
  - suponmga que conociera el valor de π— y no conociera que $S = \pi r^2$, pero sabe que

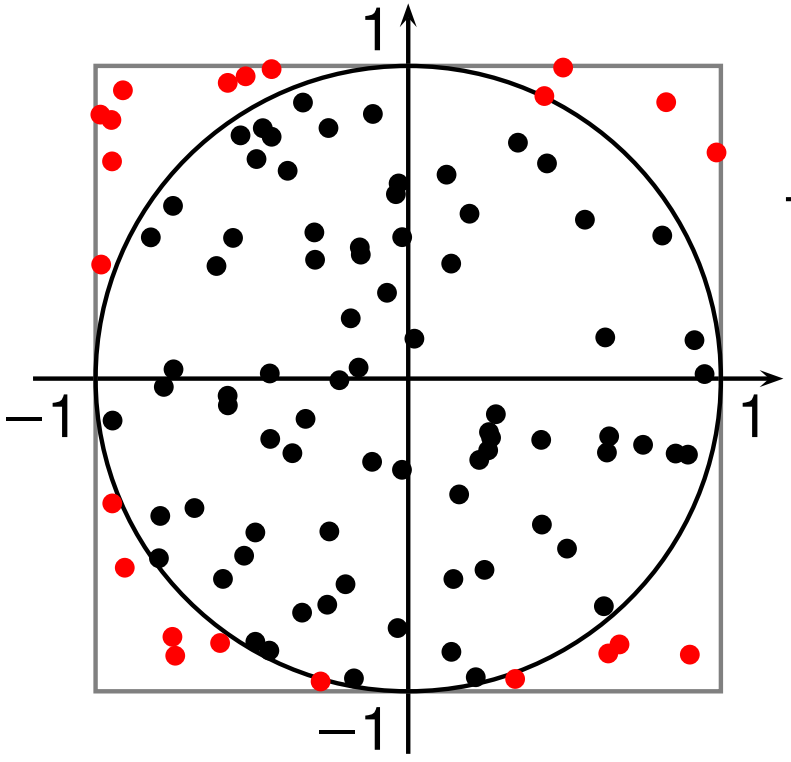
$$(x, y) \quad \text{is} \quad \begin{cases} \textit{fuera} \text{ del circulo si} & x^2 + y^2 > 1 \\ \textit{dentro} \text{ del circulo si} & x^2 + y^2 \leq 1 \end{cases}$$



Algoritmo de Monte Carlo simple

- Problema: *compute la superficie de un disco unitario*
  - suponmga que conociera el valor de π— y no conociera que $S = \pi r^2$, pero sabe que

$$(x, y) \quad \text{is} \quad \begin{cases} \textit{fuera} \text{ del circulo si} & x^2 + y^2 > 1 \\ \textit{dentro} \text{ del circulo si} & x^2 + y^2 \leq 1 \end{cases}$$

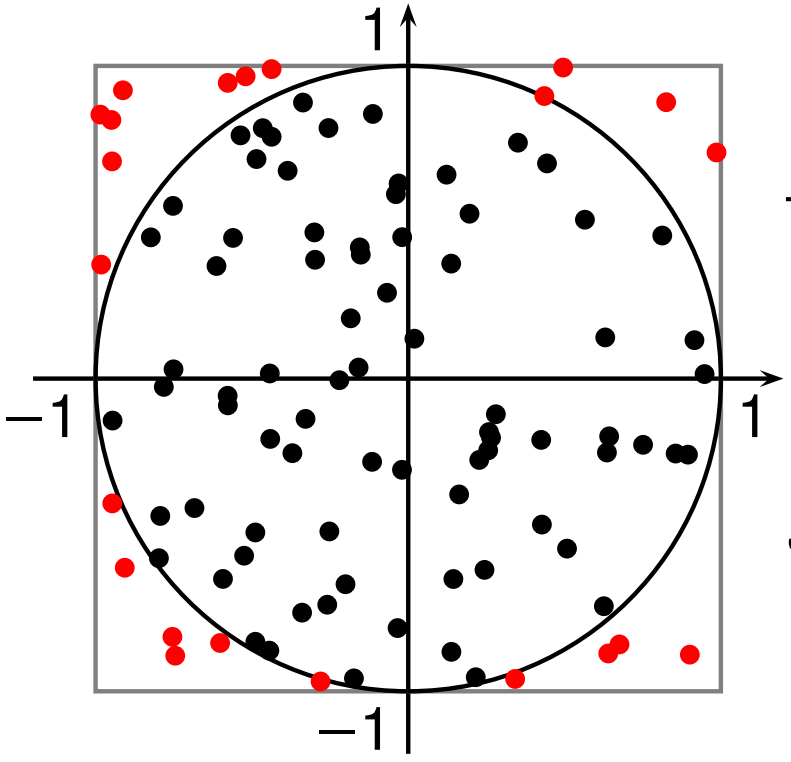


78 de los 100 son puntos *interiores*

Algoritmo de Monte Carlo simple

- Problema: *compute la superficie de un disco unitario*
  - suponmga que conociera el valor de π— y no conociera que $S = \pi r^2$, pero sabe que

$$(x, y) \quad \text{is} \quad \begin{cases} \textit{fuera} \text{ del circulo si} & x^2 + y^2 > 1 \\ \textit{dentro} \text{ del circulo si} & x^2 + y^2 \leq 1 \end{cases}$$



78 de los 100 son puntos *interiores*

$$S = 4 \times 78/100 = 3.12$$