Randomized Algorithms

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

May 8, 2012

Outline

- Examples
- Las Vegas and Monte Carlo algorithms
- More examples

Remember Quick-Sort?

Remember Quick-Sort?

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

Remember Quick-Sort?

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

- Idea: *partition* the sequence $A[1 \dots n]$ in three parts
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the “pivot” value ($v \in A$)
  - $A[q+1 \dots n]$ contain elements that are *greater than* v

Remember Quick-Sort?

```
QUICKSORT(A, begin, end)
if begin < end
        q = PARTITION(A, begin, end)
        QUICKSORT(A, begin, q − 1)
        QUICKSORT(A, q + 1, end)
```

- Idea: *partition* the sequence $A[1 \dots n]$ in three parts
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the “pivot” value ($v \in A$)
  - $A[q+1 \dots n]$ contain elements that are *greater than* v

2	36	5	21	1	13	11	20	5	4	8

Remember Quick-Sort?

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

- Idea: *partition* the sequence $A[1 \dots n]$ in three parts
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the "pivot" value ($v \in A$)
  - $A[q+1 \dots n]$ contain elements that are *greater than* v

2	36	5	21	1	13	11	20	5	4	8

$v = 8$

Remember Quick-Sort?

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

- Idea: *partition* the sequence $A[1 \ldots n]$ in three parts
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the “pivot” value ($v \in A$)
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v

2	36	5	21	1	13	11	20	5	4	8

$v = 8$

Remember Quick-Sort?

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

- Idea: *partition* the sequence $A[1 \dots n]$ in three parts
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the “pivot” value ($v \in A$)
  - $A[q+1 \dots n]$ contain elements that are *greater than* v

2	36	5	21	1	13	11	20	5	4	8

$v = 8$

2	4	1	5	5						

Remember Quick-Sort?

```
QUICKSORT(A, begin, end)
  if begin < end
      q = PARTITION(A, begin, end)
      QUICKSORT(A, begin, q − 1)
      QUICKSORT(A, q + 1, end)
```

- Idea: *partition* the sequence $A[1 \ldots n]$ in three parts
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the "pivot" value ($v \in A$)
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v

2	36	5	21	1	13	11	20	5	4	8

$v = 8$

2	4	1	5	5	8					

Remember Quick-Sort?

QuickSort(A, *begin*, *end*)

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

- Idea: *partition* the sequence $A[1 \ldots n]$ in three parts
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the "pivot" value ($v \in A$)
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v

2	36	5	21	1	13	11	20	5	4	8

$v = 8$

2	4	1	5	5	$\boxed{8}$	11	20	13	36	21

Remember Quick-Sort?

```
QUICKSORT(A, begin, end)
  if begin < end
      q = PARTITION(A, begin, end)
      QUICKSORT(A, begin, q − 1)
      QUICKSORT(A, q + 1, end)
```

- Idea: *partition* the sequence $A[1 \ldots n]$ in three parts
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the "pivot" value ($v \in A$)
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v

2	36	5	21	1	13	11	20	5	4	8

$v = 8$

$q = 6$

2	4	1	5	5	$\boxed{8}$	11	20	13	36	21

Remember Quick-Sort?

```
QUICKSORT(A, begin, end)
    if begin < end
        q = PARTITION(A, begin, end)
        QUICKSORT(A, begin, q − 1)
        QUICKSORT(A, q + 1, end)
```

- Idea: *partition* the sequence $A[1 \dots n]$ in three parts
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the “pivot” value ($v \in A$)
  - $A[q+1 \dots n]$ contain elements that are *greater than* v

2	36	5	21	1	13	11	20	5	4	8

$v = 8$

$q = 6$

2	4	1	5	5	$\boxed{8}$	11	20	13	36	21

$\underbrace{\qquad\qquad}_{A[1 \dots q-1]}$

Remember Quick-Sort?

```
QUICKSORT(A, begin, end)
if begin < end
        q = PARTITION(A, begin, end)
        QUICKSORT(A, begin, q − 1)
        QUICKSORT(A, q + 1, end)
```

- Idea: *partition* the sequence $A[1 \dots n]$ in three parts
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$ is the "pivot" value ($v \in A$)
  - $A[q+1 \dots n]$ contain elements that are *greater than* v

2	36	5	21	1	13	11	20	5	4	8

$v = 8$

$q = 6$

2	4	1	5	5	$\boxed{8}$	11	20	13	36	21

$A[1 \dots q-1]$ $A[q+1 \dots n]$

A Classic Divide-and-Conquer

- ***Divide:*** partition A in $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$ such that…

A Classic Divide-and-Conquer

- ***Divide:*** partition A in $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$ such that…
- ***Conquer:*** sort $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$

A Classic Divide-and-Conquer

- ***Divide:*** partition A in $A[1 \dots q-1]$ and $A[q+1 \dots n]$ such that…
- ***Conquer:*** sort $A[1 \dots q-1]$ and $A[q+1 \dots n]$
- ***Combine:*** nothing to do here
  - it is all in the *partition* algorithm

A Classic Divide-and-Conquer

- ***Divide:*** partition A in $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$ such that...
- ***Conquer:*** sort $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$
- ***Combine:*** nothing to do here
  - it is all in the *partition* algorithm

```
PARTITION(A, begin, end)
q = begin
v = A[end]  // we deterministically choose v
for i = begin to end
    if A[i] ≤ v
        swap(A[i], A[q])
        q = q + 1
return q − 1
```

Complexity of QUICKSORT

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

- *Worst case:* $q = 1$ or $q = n$

Complexity of QUICKSORT

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

- *Worst case:* $q = 1$ or $q = n$
  - the partition transforms P of size n in P of size $n - 1$, so $T(n) = T(n-1) + \Theta(n)$

Complexity of QuickSort

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

- *Worst case:* $q = 1$ or $q = n$
  - the partition transforms P of size n in P of size $n - 1$, so $T(n) = T(n-1) + \Theta(n)$

$$\boxed{T(n) = \Theta(n^2)}$$

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

- *Worst case:* $q = 1$ or $q = n$
  - the partition transforms P of size n in P of size $n - 1$, so $T(n) = T(n-1) + \Theta(n)$

$$\boxed{T(n) = \Theta(n^2)}$$

- *Best case:* $q = \lceil n/2 \rceil$

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

- *Worst case:* $q = 1$ or $q = n$
  - the partition transforms P of size n in P of size $n - 1$, so $T(n) = T(n-1) + \Theta(n)$

$$\boxed{T(n) = \Theta(n^2)}$$

- *Best case:* $q = \lceil n/2 \rceil$
  - the partition transforms P of size n into *two* problems P of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$, respectively; so $T(n) = 2T(n/2) + \Theta(n)$

Complexity of QuickSort

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

- *Worst case:* $q = 1$ or $q = n$
  - the partition transforms P of size n in P of size $n - 1$, so $T(n) = T(n-1) + \Theta(n)$

$$\boxed{T(n) = \Theta(n^2)}$$

- *Best case:* $q = \lceil n/2 \rceil$
  - the partition transforms P of size n into *two* problems P of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$, respectively; so $T(n) = 2T(n/2) + \Theta(n)$

$$\boxed{T(n) = \Theta(n \log n)}$$

Complexity of QuickSort

QuickSort$(A, begin, end)$

if $begin < end$
  $q = $ **Partition**$(A, begin, end)$
  QuickSort$(A, begin, q-1)$
  QuickSort$(A, q+1, end)$

- *Worst case:* $q = 1$ or $q = n$ *not so improbable!*
  - the partition transforms P of size n in P of size $n-1$, so $T(n) = T(n-1) + \Theta(n)$

$$\boxed{T(n) = \Theta(n^2)}$$

- *Best case:* $q = \lceil n/2 \rceil$
  - the partition transforms P of size n into *two* problems P of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$, respectively; so $T(n) = 2T(n/2) + \Theta(n)$

$$\boxed{T(n) = \Theta(n \log n)}$$

A Randomized Solution

A Randomized Solution

- Simple

```
RANDOMIZED-PARTITION(A, begin, end)
    i = RANDOM(begin, end)
    swap(A[i], A[end])
    return PARTITION(A, begin, end)
```

A Randomized Solution

- Simple

```
RANDOMIZED-PARTITION(A, begin, end)
    i = RANDOM(begin, end)
    swap(A[i], A[end])
    return PARTITION(A, begin, end)
```

```
QUICKSORT(A, begin, end)
    if begin < end
        q = RANDOMIZED-PARTITION(A, begin, end)
        QUICKSORT(A, begin, q − 1)
        QUICKSORT(A, q + 1, end)
```

Other Examples

Other Examples

```
TREE-RANDOMIZED-INSERT(t, z)
if t = NIL
        return z
r = uniformly random value from {1, . . . , size(t) + 1}
if r = 1                              // Pr[r = 1] = 1/(size(t) + 1)
        size(z) = size(t) + 1
        return TREE-ROOT-INSERT(t, z)
if key(z) < key(t)
        left(t) = TREE-RANDOMIZED-INSERT(left(t), z)
else right(t) = TREE-RANDOMIZED-INSERT(right(t), z)
size(t) = size(t) + 1
return t
```

General Idea

General Idea

- We develop *randomized variants* of algorithms

General Idea

- We develop *randomized variants* of algorithms

- The idea is to make the algorithm run with their *good complexity* ***with high probability*** *on every input*

General Idea

- We develop *randomized variants* of algorithms

- The idea is to make the algorithm run with their *good complexity* ***with high probability*** *on every input*

- The algorithm does not change that much, though

General Idea

- We develop *randomized variants* of algorithms
- The idea is to make the algorithm run with their *good complexity* ***with high probability*** *on every input*
- The algorithm does not change that much, though

- However, we can do a lot more with randomized algorithms…

Simple Example

- Problem: *find a zero bit*
  - *Input:* an array A, of n bits, containing more or less the same number of 1s and 0s (*hamming weight* is roughly $n/2$)
  - *Output:* i such that $A[i] = 0$, or NIL if none exists

- Problem: *find a zero bit*
  - *Input:* an array A, of n bits, containing more or less the same number of 1s and 0s (*hamming weight* is roughly $n/2$)
  - *Output:* i such that $A[i] = 0$, or NIL if none exists

- Obvious solution

```
FIND-A-ZERO-BIT(A)
for i = 1 to |A|
        if A[i] == 0
                return i
return NIL
```

Simple Example

- Problem: *find a zero bit*
  - *Input:* an array A, of n bits, containing more or less the same number of 1s and 0s (*hamming weight* is roughly $n/2$)
  - *Output:* i such that $A[i] = 0$, or NIL if none exists

- Obvious solution

```
FIND-A-ZERO-BIT(A)
for i = 1 to |A|
    if A[i] == 0
        return i
return NIL
```

- Problems?
  - what if A is sorted in reverse order? (all 1-bits before the 0-bits)

Simple Example

- Problem: *find a zero bit*
  - *Input:* an array A, of n bits, containing more or less the same number of 1s and 0s (*hamming weight* is roughly $n/2$)
  - *Output:* i such that $A[i] = 0$, or NIL if none exists

- Obvious solution

```
FIND-A-ZERO-BIT(A)
for i = 1 to |A|
    if A[i] == 0
        return i
return NIL
```

- Problems?
  - what if A is sorted in reverse order? (all 1-bits before the 0-bits)
  - any *deterministic* search strategy is vulnerable

A Randomized Algorithm

A Randomized Algorithm

- Take one

```
RANDOMIZED-FIND-A-ZERO-BIT1(A)
repeat
    i = RANDOM(1, |A|)
until A[i] == 0
return i
```

A Randomized Algorithm

- Take one

```
RANDOMIZED-FIND-A-ZERO-BIT1(A)
repeat
    i = RANDOM(1, |A|)
until A[i] == 0
return i
```

- expected iterations: 2

A Randomized Algorithm

- Take one

```
RANDOMIZED-FIND-A-ZERO-BIT1(A)
repeat
    i = RANDOM(1, |A|)
until A[i] == 0
return i
```

- expected iterations: 2
- worst-case:

A Randomized Algorithm

- Take one

```
RANDOMIZED-FIND-A-ZERO-BIT1(A)
repeat
    i = RANDOM(1, |A|)
until A[i] == 0
return i
```

- expected iterations: 2
- worst-case: unbounded!

A Randomized Algorithm

- Take one

```
RANDOMIZED-FIND-A-ZERO-BIT1(A)
repeat
    i = RANDOM(1, |A|)
until A[i] == 0
return i
```

- expected iterations: 2
- worst-case: unbounded!

- Take two

```
RANDOMIZED-FIND-A-ZERO-BIT2(A)
for j = 1 to k
    i = RANDOM(1, |A|)
    if A[i] == 0
        return i
return NIL
```

A Randomized Algorithm

- Take one

```
RANDOMIZED-FIND-A-ZERO-BIT1(A)
repeat
    i = RANDOM(1, |A|)
until A[i] == 0
return i
```

- expected iterations: 2
- worst-case: unbounded!

- Take two

```
RANDOMIZED-FIND-A-ZERO-BIT2(A)
for j = 1 to k
    i = RANDOM(1, |A|)
    if A[i] == 0
        return i
return NIL
```

- worst-case iterations: k

A Randomized Algorithm

- Take one

```
RANDOMIZED-FIND-A-ZERO-BIT1(A)
repeat
    i = RANDOM(1, |A|)
until A[i] == 0
return i
```

- expected iterations: 2
- worst-case: unbounded!

- Take two

```
RANDOMIZED-FIND-A-ZERO-BIT2(A)
for j = 1 to k
    i = RANDOM(1, |A|)
    if A[i] == 0
        return i
return NIL
```

- worst-case iterations: k
- worst-case: wrong result!

A Randomized Algorithm

- Take one

```
RANDOMIZED-FIND-A-ZERO-BIT1(A)
repeat
    i = RANDOM(1, |A|)
until A[i] == 0
return i
```

- ▶ expected iterations: 2
- ▶ worst-case: unbounded!
- ▶ ***Las Vegas***

- Take two

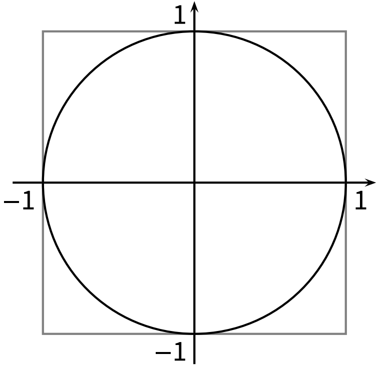
```
RANDOMIZED-FIND-A-ZERO-BIT2(A)
for j = 1 to k
    i = RANDOM(1, |A|)
    if A[i] == 0
        return i
return NIL
```

- ▶ worst-case iterations: k
- ▶ worst-case: wrong result!
- ▶ ***Monte Carlo***

Simple Monte Carlo Algorithm

- Problem: *compute the surface of the unit disc*
  - you don't know the value of π—you may not even know that $S = \pi r^2$, but you know that a point

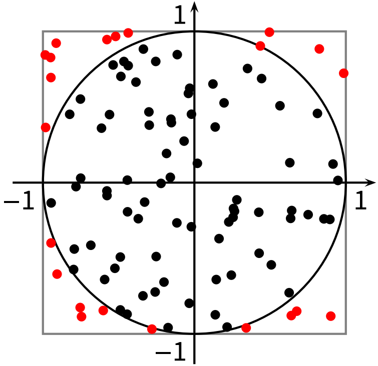
$$(x, y) \quad \text{is} \begin{cases} \textit{outside} \text{ the unit disc if} & x^2 + y^2 > 1 \\ \textit{inside} \text{ the unit disc if} & x^2 + y^2 \leq 1 \end{cases}$$



Simple Monte Carlo Algorithm

- Problem: *compute the surface of the unit disc*
  - you don't know the value of π—you may not even know that $S = \pi r^2$, but you know that a point

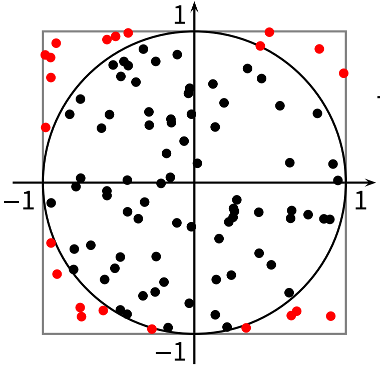
$$(x, y) \quad \text{is} \begin{cases} \textit{outside} \text{ the unit disc if} & x^2 + y^2 > 1 \\ \textit{inside} \text{ the unit disc if} & x^2 + y^2 \leq 1 \end{cases}$$



Simple Monte Carlo Algorithm

- Problem: *compute the surface of the unit disc*
  - you don't know the value of π—you may not even know that $S = \pi r^2$, but you know that a point

$$(x, y) \quad \text{is} \begin{cases} \textit{outside} \text{ the unit disc if} & x^2 + y^2 > 1 \\ \textit{inside} \text{ the unit disc if} & x^2 + y^2 \leq 1 \end{cases}$$

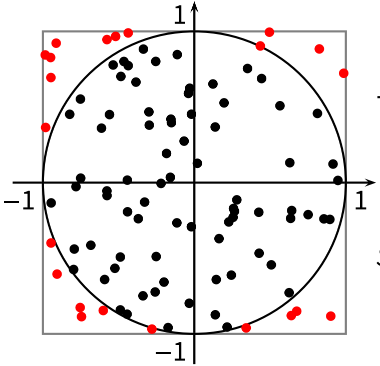


78 out of 100 points are *inside*

Simple Monte Carlo Algorithm

- Problem: *compute the surface of the unit disc*
  - you don't know the value of π—you may not even know that $S = \pi r^2$, but you know that a point

$$(x, y) \quad \text{is} \begin{cases} \textit{outside} \text{ the unit disc if} & x^2 + y^2 > 1 \\ \textit{inside} \text{ the unit disc if} & x^2 + y^2 \leq 1 \end{cases}$$



78 out of 100 points are *inside*

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

Monte Carlo π

```
MONTE-CARLO-PI(n)
i = 0
for j = 1 to n
        x = RANDOM(−1, 1)
        y = RANDOM(−1, 1)
        if x^2 + y^2 ≤ 1
                i = i + 1
return 4i/n
```

```
MONTE-CARLO-PI(n)
i = 0
for j = 1 to n
        x = RANDOM(−1, 1)
        y = RANDOM(−1, 1)
        if x^2 + y^2 ≤ 1
                i = i + 1
return 4i/n
```

- The precision grows with n
  - more specifically, the *expected* precision grows with n

```
MONTE-CARLO-PI(n)
i = 0
for j = 1 to n
        x = RANDOM(−1, 1)
        y = RANDOM(−1, 1)
        if x^2 + y^2 ≤ 1
                i = i + 1
return 4i/n
```

- The precision grows with *n*
  - more specifically, the *expected* precision grows with *n*

- It is also easy to think about a better (adaptive) *stopping condition* other than going through *n* loops