SORTING

Juan Mendivelso

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

CONTENTS

1. PROBABILISTIC ANALYSIS & RANDOMIZED ALGORITHMS

Indicator Random Variables

- Useful to analyze algorithms.
- Convenient method for converting between probabilities and expectations.
- Given a sample space S and an event A, the **indicator random variable** I{A} associated with event A is:

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs ,} \\ 0 & \text{if } A \text{ does not occur .} \end{cases}$$

Example of Indicator Random Variables

- Let us determine the expected number of heads that we obtain when flipping a fair coin.
- Sample Space S={H,T}.
- Event H: the coin comes up heads; event T: the coin comes up tails.
- Pr{H} = P{T} = ½.
- Indicator variable X_H: number of heads obtained in this flip.

$$
\begin{aligned}
X_H &= \mathrm{I}\{H\} \\
&= \begin{cases} 1 & \text{if } H \text{ occurs} , \\ 0 & \text{if } T \text{ occurs} . \end{cases}
\end{aligned}
$$

Example of Indicator Random Variables

$$X_H = \mathrm{I}\{H\} = \begin{cases} 1 & \text{if } H \text{ occurs}, \\ 0 & \text{if } T \text{ occurs}. \end{cases}$$

- Expected number of heads obtained in one flip:

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{H\}] \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2. \end{aligned}$$

Expectation of an Indicator Random Variable

Lemma 5.1

Given a sample space S and an event A in the sample space S, let $X_A = \mathrm{I}\{A\}$. Then $\mathrm{E}[X_A] = \Pr\{A\}$.

Proof By the definition of an indicator random variable from equation (5.1) and the definition of expected value, we have

$$\begin{aligned}
\mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\
&= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \\
&= \Pr\{A\} ,
\end{aligned}$$

where $\overline{A}$ denotes $S - A$, the complement of A. ■

Repeated Random Trials

- Indicator random variables are useful for analyzing situations in which we perform repeated random trials.
- Let X_i be the indicator random variable associated with the event in which the i-th flip comes up heads.
- Let X be the random variable denoting the total number of heads in n coin flips.

$$X = \sum_{i=1}^{n} X_i \qquad \mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right]$$

Repeated Random Trials

- We can compute this easily because of the linearity of expectation:

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/2 \\
&= n/2 .
\end{aligned}
$$

Randomized Algorithms

- In order to use probabilistic analysis, we need to know something about the distribution of the inputs.
- Yet we often can use probability and randomness as a tool for algorithm design and analysis, by making the behavior of part of the algorithm random.
- Instead of assuming a distribution of inputs, we impose a distribution.

Probabilistic Analysis VS Randomized Algorithms

- **Probabilistic Analysis**
  - The algorithm is deterministic.
  - We make assumptions about the distribution of the input.
  - The execution depends merely on the input.
  - Different executions on the same input take the same time.
  - Some input produces the worst-case.
  - Its efficiently depend on the input.
- **Randomized Algorithm**
  - It starts by randomizing the input.
  - It does not make assumptions about the input of the algorithm.
  - The execution depends on the random choices made.
  - Different executions on the same input can vary.
  - No particular input elicits the worst-case.
  - It performs badly only if the random generator produces an unlucky permutation.

Random-Number Generators

- Its behavior is determined not only by its input but also by values produced by a **random-number generator**.
- Random(a,b) returns a number between a and b, inclusive, with each integer being equally likely.
- Probability of picking a number: 1/(b-a+1).
- Each integer is generated by Random(a,b) is independent of the integers generated in previous calls.
- Programming language provide **pseudorandom-number generators:** deterministic algorithms that produce numbers that look statistically random.

Analyzing the Running Time of an Algorithm

- **Expected Running Time of a Randomized Algorithm:** Expectation of the running time over the distribution values returned by the random-number generator.
- **Average-Case Running Time:** The probability distribution is over the inputs of the algorithm.

2. THE SORTING PROBLEM

The Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Why sorting?

- Applications inherently need to sort information.
- Algorithms of different areas use sorting as a key subroutine.
- This problem has a rich set of algorithmic techniques.
- It has a historical interest.
- We have non-trivial lower bounds for sorting, so we can evaluate optimality.
- Many engineering issues can be address from an algorithmic perspective.

Satellite Data

- We consider the elements to be sorted as a sequence of **keys**.
- But, in fact, they can be stored along with **satellite data**.
- Then, we sort **registers** with keys and satellite data.
- Notwithstanding, we can have an array with **pointers** to each register. When we sort the elements, we don’t move the whole data, just the pointers in the array.

Comparison-Based Sorting Algorithms

- It is proven that they take $\Omega(n \lg n)$.

- Insertion Sort
  - $O(n^2)$.
  - In place.
- Merge Sort
  - $\Theta(n \lg n)$.
  - Not in place.
- Heapsort
  - $O(n \lg n)$.
  - In place.
- Quicksort
  - $O(n^2)$.
  - In place.
  - Average case: $\Theta(n \lg n)$.

Linear Time Sorting Algorithms

- There are some algorithms that run in linear time.
- They make assumptions on the distribution or the range of the data.
- Some of them are:
  - Counting sort
  - Radix sort
  - Bucket sort

SORTING ALGORITHMS

- The most important algorithms for sorting are:

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

3. QUICKSORT

Quicksort

- Divide & Conquer algorithm to sort A[p..r].
- Take x=A[r] as a pivot.
- Find the correct position q of x in A.
- **Divide** the array into A[p..q-1] and A[q+1..r] such that
  - A[p..q-1] contains the elements ≤ x.
  - A[q+1..r] contains the elements > x.
- Place x at A[q].
- **Conquer** by recursively sorting A[p..q-1] and A[q+1..r].
- **Combine** nothing. Everything is already sorted.

Quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

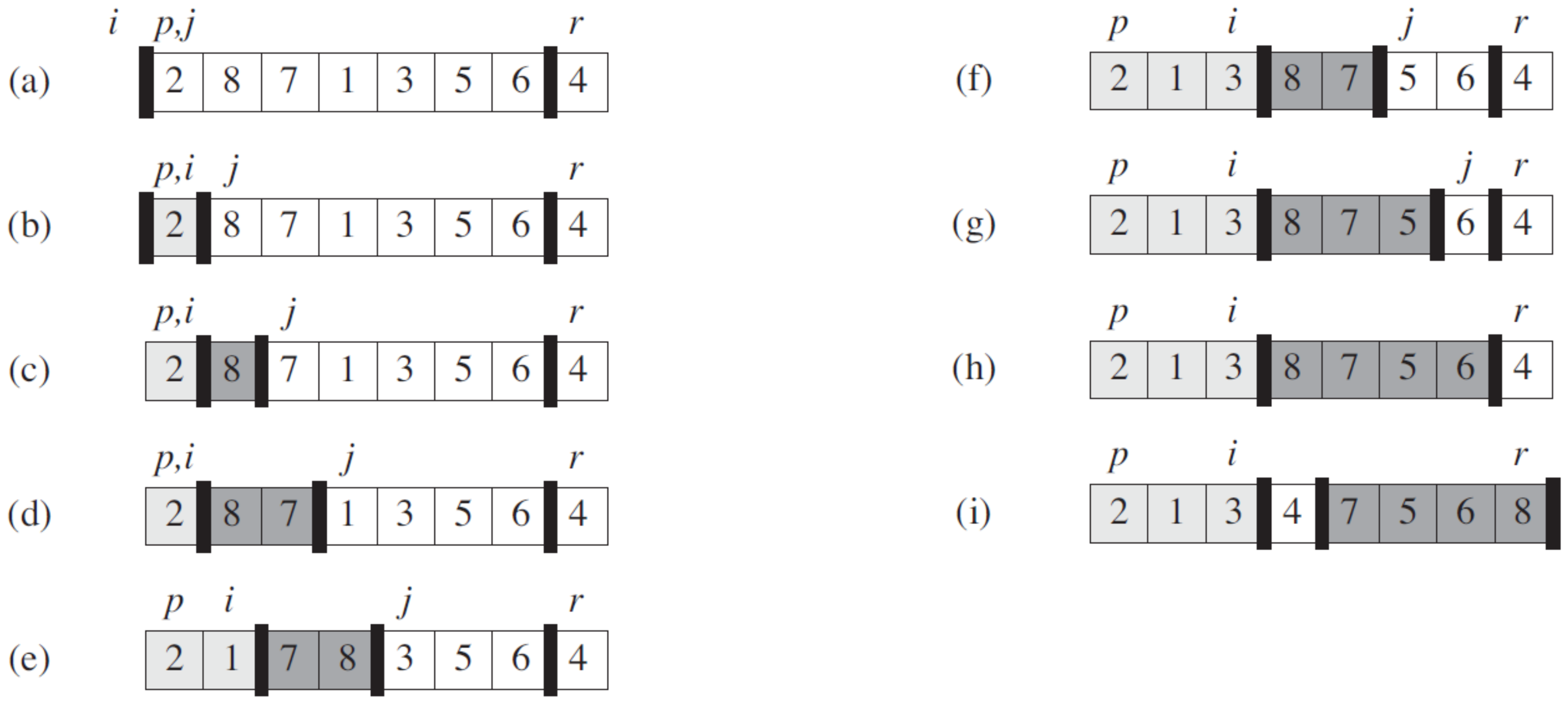
- Divide & Conquer algorithm to sort A[p..r].
- Take x=A[r] as a pivot.
- Find the correct position q of x in A.
- **Divide** the array into A[p..q-1] and A[q+1..r] such that
  - A[p..q-1] contains the elements ≤ x.
  - A[q+1..r] contains the elements > x.
- Place x at A[q].
- **Conquer** by recursively sorting A[p..q-1] and A[q+1..r].
- **Combine** nothing. Everything is already sorted.

Partition

- We can traverse A from left to right to establish which elements are ≤ x and which are > x.
- At the same time, we can relocate some of them so that the elements of the same set are contiguous.
- We can locate the elements ≤ x in A[1..i].
- The elements > x are placed in A[i+1..j-1].
- The elements we haven’t revised are in A[j..r-1].

Partition

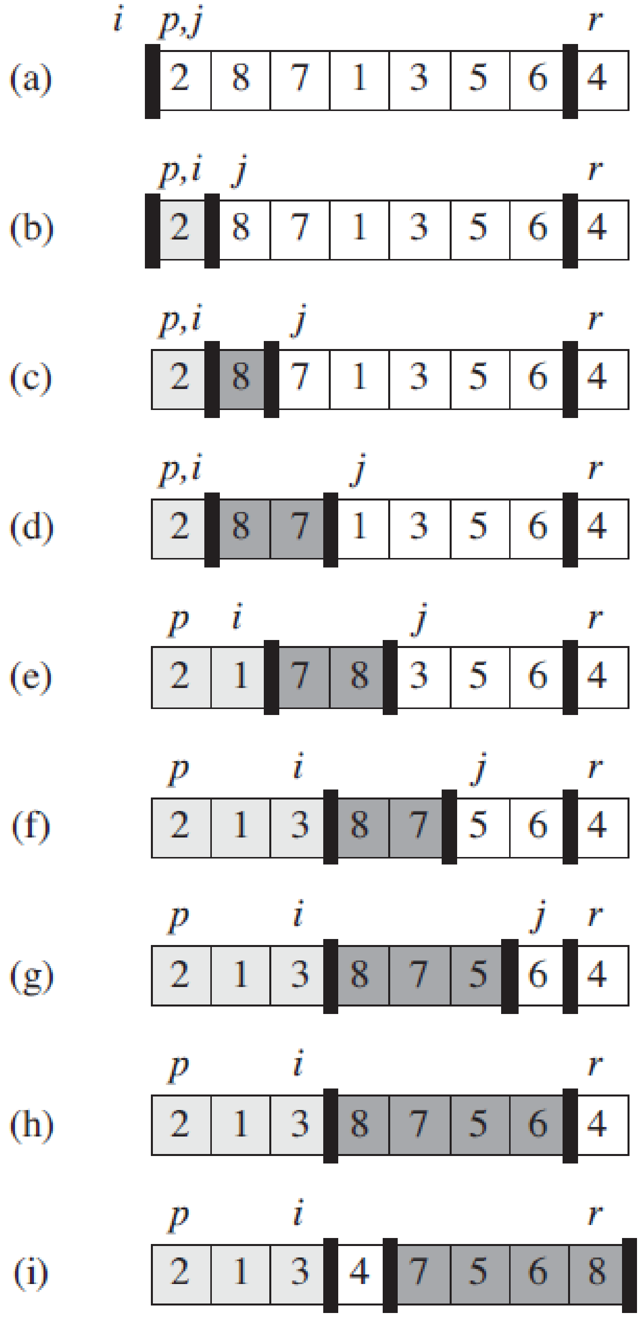
- For example, let's sort 2,8,7,1,3,5,6,4. The pivot is x=A[8] = 4.



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Partition

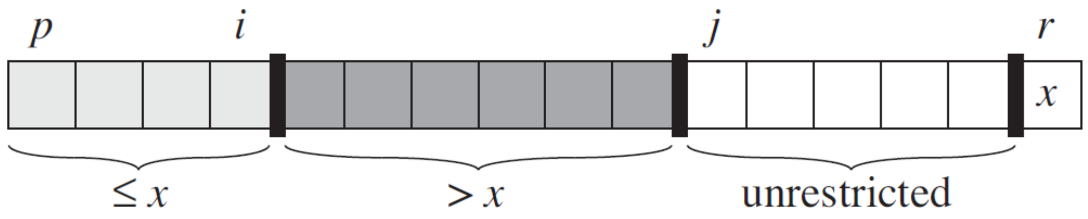
```
PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

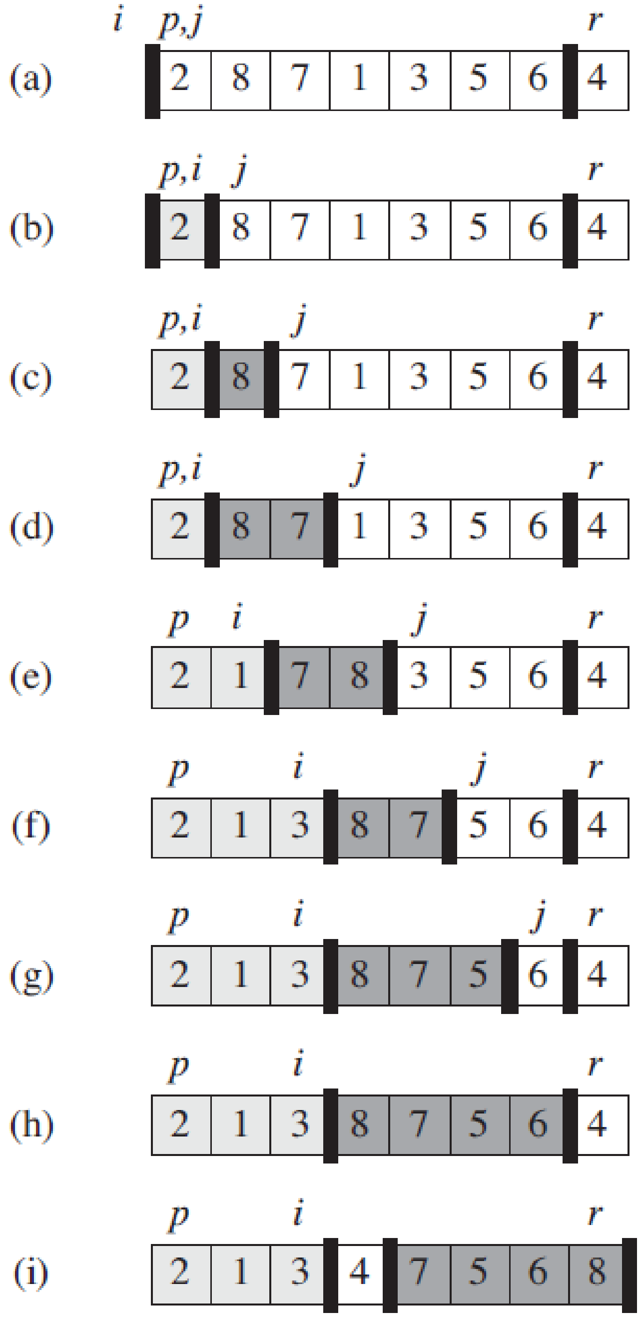


Loop Invariant

PARTITION(A, p, r)

```
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

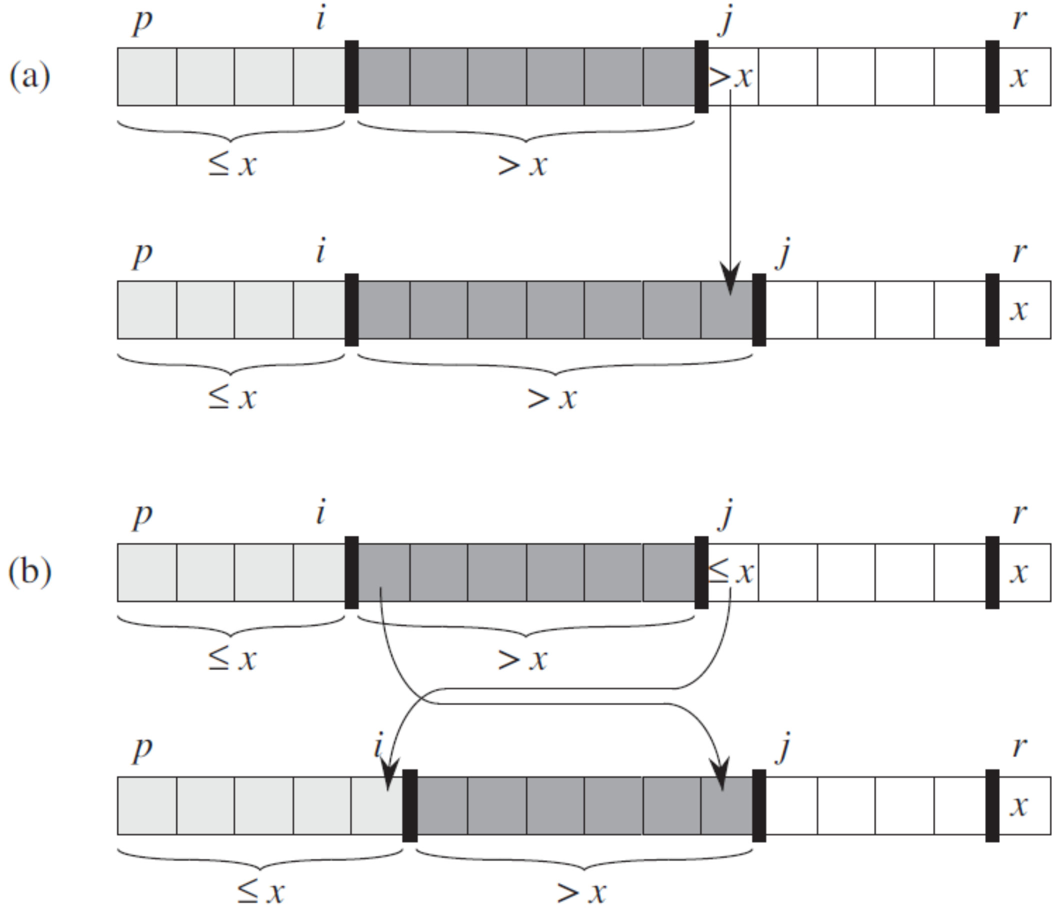




At the beginning of each iteration of the loop of lines 3–6, for any array index k,

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

Loop Invariant



PARTITION(A, p, r)

1 $x = A[r]$
2 $i = p - 1$
3 **for** $j = p$ **to** $r - 1$
4   **if** $A[j] \leq x$
5         $i = i + 1$
6   exchange $A[i]$ with $A[j]$
7 exchange $A[i + 1]$ with $A[r]$
8 **return** $i + 1$

At the beginning of each iteration of the loop of lines 3–6, for any array index k,

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

Complexity

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Time Complexity?
- Space Complexity?

Complexity of Partition

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Time Complexity: θ(n), n= r-p+1.
- Space Complexity: θ(1).

Complexity of Quicksort

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

- Recurrence?

Complexity of Quicksort

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

- Recurrence: T(n) = T(?)+ T(?) + θ(n)
- It depends on the case!

Complexity of Quicksort – Worst Case

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

- What is the worst case for this algorithm?

Complexity of Quicksort – Worst Case

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

- Every partition produces n-1 elements ≤ x and zero elements > x, or vice versa.

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \,. \end{aligned}$$

- When does this happen?

- What is the solution to this recursion?

Complexity of Quicksort – Worst Case

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

- Every partition produces n-1 elements ≤ x and zero elements > x, or vice versa.

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

- When does this happen?
  - Input already sorted!
- What is the solution to this recursion?

Complexity of Quicksort – Worst Case

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

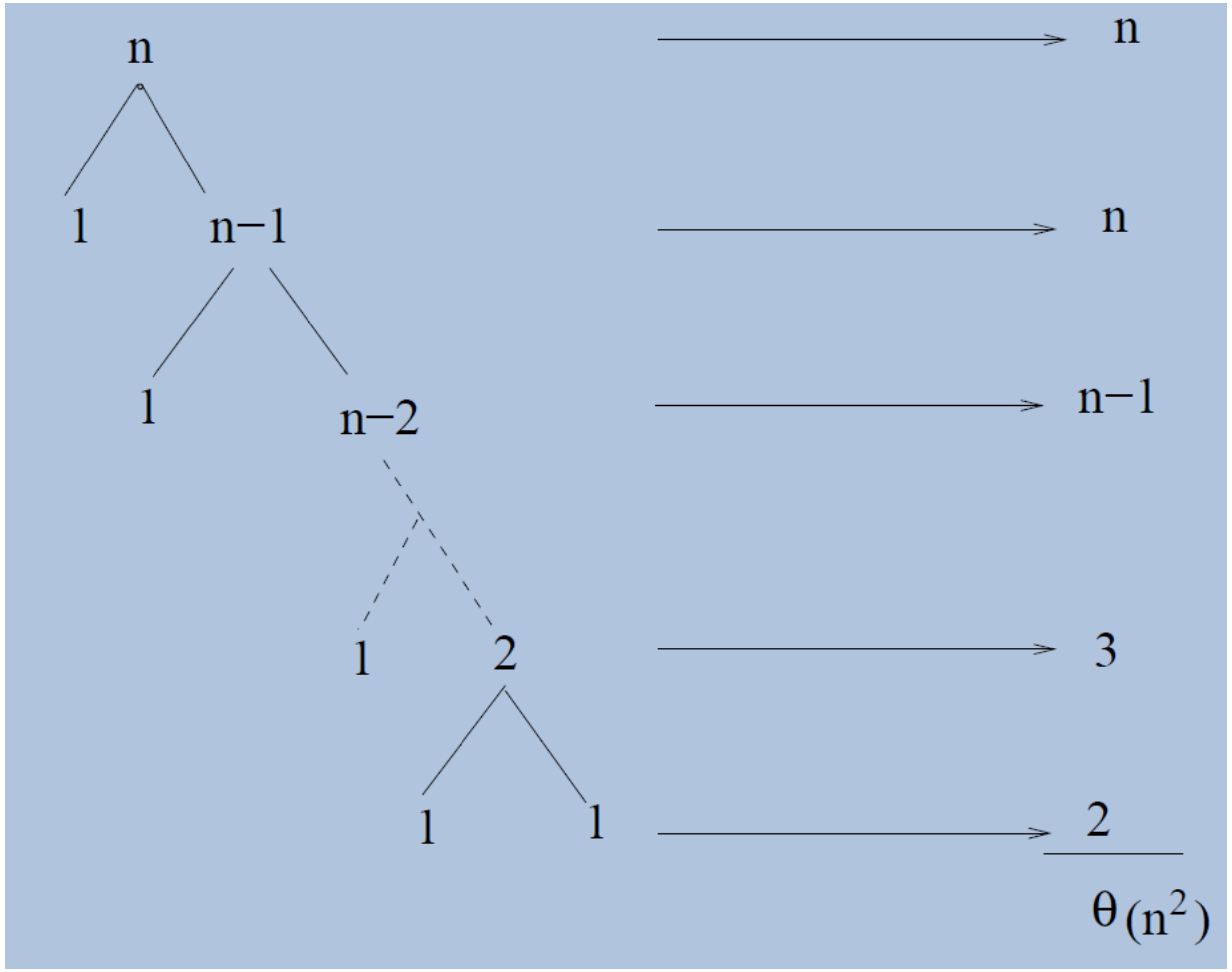
```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

$$
\begin{aligned}
T(n) &= \underbrace{T(n-1)}_{\text{una particion}} + \underbrace{T(0)}_{\text{otra particion}} + \underbrace{\theta(n)}_{\text{Particion}} \\
&= T(n-1) + \theta(n) \\
&= T(n-2) + \theta(n-1) + \theta(n) \\
&\vdots \\
&= \sum_{i=0}^{n-1} \theta(n-i) = \sum_{i=1}^{n} \theta(i) \\
&= \theta\left(\sum_{i=1}^{n} i\right) = \theta(n^2)
\end{aligned}
$$

Complexity of Quicksort – Worst Case

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```



Complexity of Quicksort – Worst Case

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

- Every partition produces n-1 elements ≤ x and zero elements > x, or vice versa.

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(n) \\
&= T(n-1) + \Theta(n) .
\end{aligned}
$$

- What is the solution to this recursion?
  - $\Theta(n^2)$.
  - It is not better than Insertion Sort.
  - Actually, Insertion Sort is linear when the input is sorted.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Analysis of Quicksort – Worst Case

- We formally prove this with the substitution method.
- Let T(n) denote the worst-case running time Quicksort.
- There are two subproblems: one of size q and the other of size n-(q-1)

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

- Prove that $T(n) = \theta(n^2)$, i.e. $T(n) \le cn^2$.
- Inductive hypotheses:
  - $T(q) \le cq^2$
  - $T(n-q-1) \le c(n-q-1)^2$

Analysis of Quicksort – Worst Case

- We formally prove this with the substitution method.
- Let T(n) denote the worst-case running time Quicksort.
- There are two subproblems: one of size q and the other of size n-(q-1)

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

- Prove that $T(n) = \theta(n^2)$, i.e. $T(n) \le cn^2$.
- Inductive hypotheses:
  - $T(q) \le cq^2$
  - $T(n-q-1) \le c(n-q-1)^2$

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

Analysis of Quicksort – Worst Case

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

- The expression $q^2+(n-q-1)^2$ achieves its maximum on either endpoint.

$$\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$$

- Thus, $\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$

- Then, we can pick a constant c sufficiently large so that –c(2n-1) dominates θ(n).

Complexity of Quicksort – Best Case

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
      if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

- Every partition produces two subproblems of size no more than n/2:
  - $\lfloor n/2 \rfloor$
  - $\lceil n/2 \rceil -1$
- Recurrence: $$T(n) = 2T(n/2) + \Theta(n) ,$$
- What is the solution to this recursion?

Complexity of Quicksort – Best Case

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

- Every partition produces two subproblems of size no more than n/2:
  - $\lfloor n/2 \rfloor$
  - $\lceil n/2 \rceil - 1$
- Recurrence: $T(n) = 2T(n/2) + \Theta(n)$,
- What is the solution to this recursion?
  - Θ(n lg n).

Complexity of Quicksort – Other Bad Partition

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

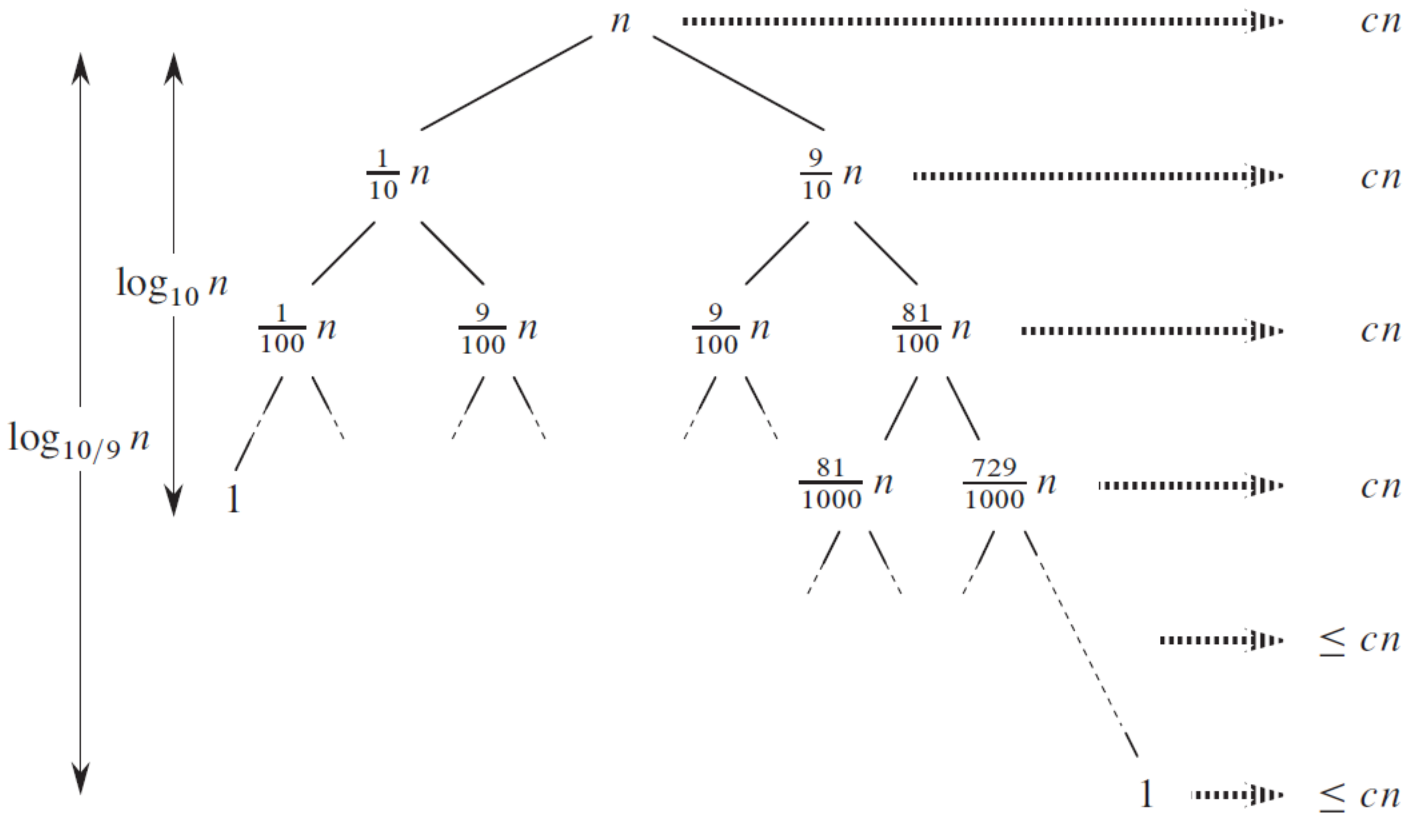
- Every partition produces two subproblems such as:

$$T(n) = \theta(n) + T(\frac{a-1}{a}n) + T(\frac{1}{a}n), \quad a > 2,$$

Complexity of Quicksort – Other Bad Partition

```
PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

```
QUICKSORT(A, p, r)
    if p < r
        q = PARTITION(A, p, r
        QUICKSORT(A, p, q - 1
        QUICKSORT(A, q + 1, r,
```



$$T(n) = T(9n/10) + T(n/10) + cn$$

$O(n \lg n)$

Complexity of Quicksort – Other Bad Partition

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

- In the second worst partition, the solution to the recurrence is also

  $\theta(n \lg n)$.

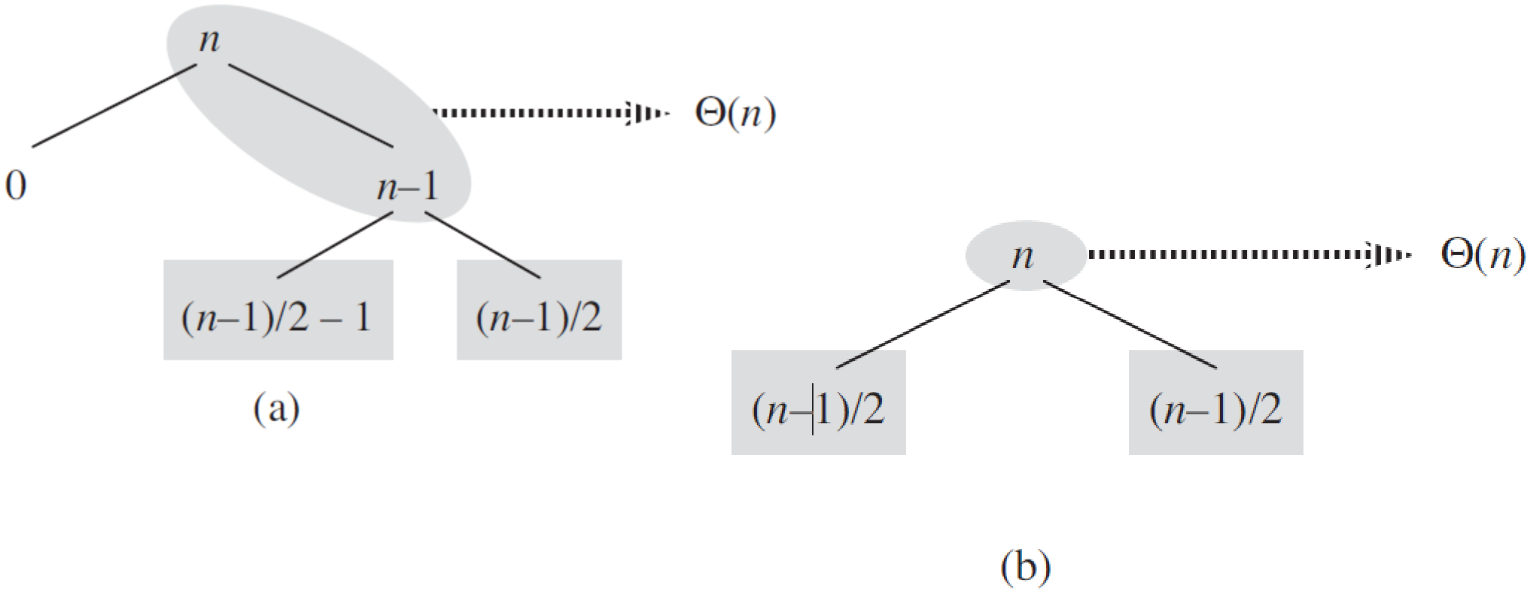
$$T(n) = T(9n/10) + T(n/10) + cn$$

Complexity of Quicksort – Average Case

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

- Assume in an execution of Quicksort half partitions are good and half partitions are bad, alternating.
- Good partitions absorb bad partitions:

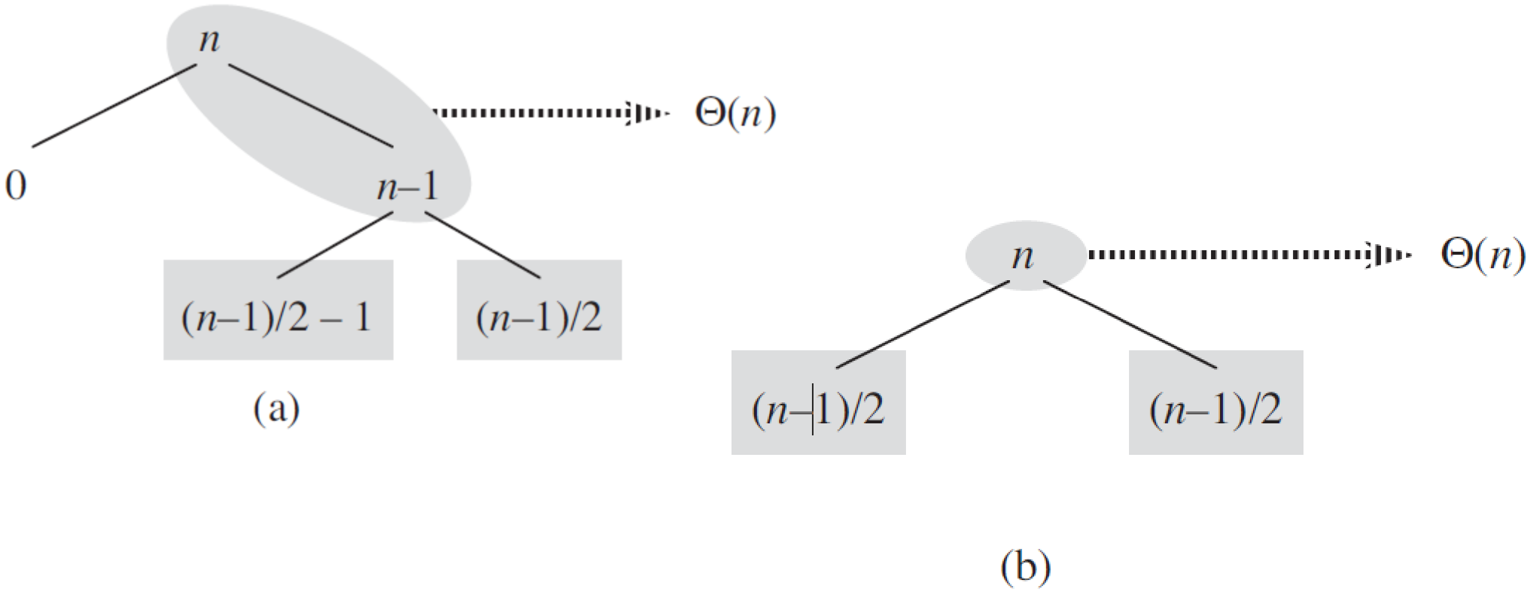


Complexity of Quicksort – Average Case

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

- Thus, the running in the average case is θ(n lg n).



(a) (b)

Randomized Versions of Quicksort

We can no longer have an input associated to the worst case. We can do so by…

1. Permuting the input.
2. Random Sampling: Instead of always using A[r] as the pivot, we select a randomly chosen element from A[p..r]. Such chosen element is exchanged with A[r].
   - Notice that now the pivot can be now any of the r-p+1 elements of the array with equal probability.
   - We expect the input array to be balanced on average.

Randomized-Quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```


Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Expected Running Time of Randomized-Quicksort

- Each call to Randomized-Partition takes O(1).
- There are at most n calls to Randomized-Partition since each element can be choses as pivot at most once.
- Each call of Partition takes time proportional to the number of iterations of the for loop (lines 3-6).

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Expected Running Time of Randomized-Quicksort

- At least the algorithm takes n steps due to the first execution of Partition.
- Complexity: O(n+X)
  - Where X is the number of times that line 4 in Partition is executed overall.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Expected Running Time of Randomized-Quicksort

- Complexity: O(n+X)
  - Where X is a random variable that indicates the number of comparisons of elements in the array.
- Let z_i be the i-th smallest element of array A.
- Each pair z_i and z_j are compared at most once.
- Thus, the sorted array is $z_1, z_{2,\dots,} z_n$.
- Also, let's define $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$.
- We define the following indicator random variable:

$$X_{ij} = \mathrm{I}\{z_i \text{ is compared to } z_j\}$$

Expected Running Time of Randomized-Quicksort

- Once a pivot x is chosen with $z_i < x < z_j$, we know that z_i and z_j are not compared at any subsequent time.
- If z_i is the first pivot chosen from Z_{ij}, z_i and z_j are compared.
- If z_j is the first pivot chosen from Z_{ij}, z_i and z_j are compared.
- Thus, z_i and z_j are compared iff the first pivot to be chosen from Z_{ij} is either z_i or z_j .

Expected Running Time of Randomized-Quicksort

$$
\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} .
\end{aligned}
$$

Expected Running Time of Randomized-Quicksort

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \qquad (k = j - i) \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) .
\end{aligned}$$

BIBLIOGRAPHY

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press. 2009.
- Some images were extracted from Julio Cesar Lopez’s material.