Quicksort

References:

Introduction to Algorithms

by Cormen, Leiserson, Rivest & Stein [Chapter 7]

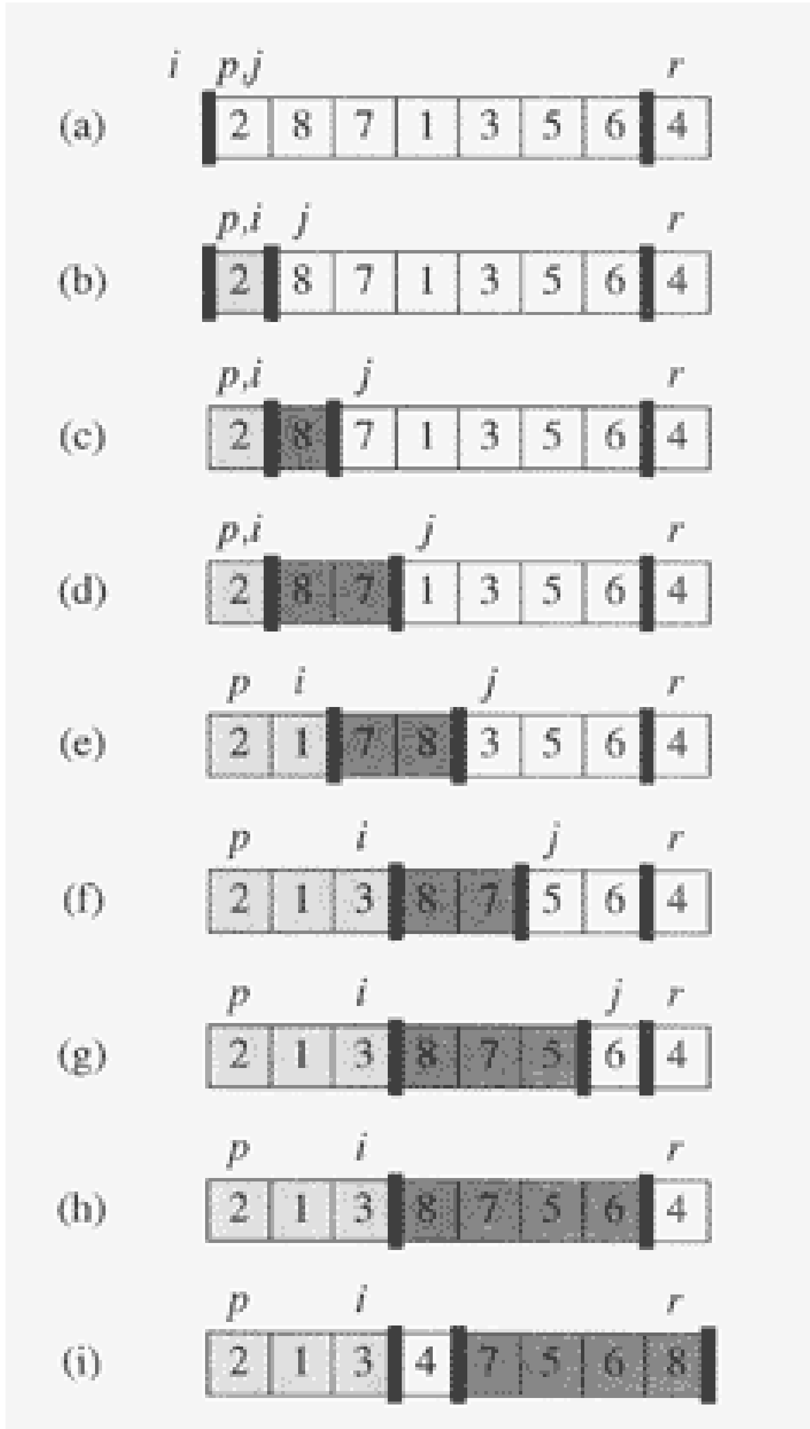
Fundamentals of Algorithmics

by Gilles Brassard and Paul Bratley [Chapter 10]

A: array to be *in-place* sorted.
p and *r* are the start and the end indices.
q is the final index of the pivot.

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

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
        do if A[j] ≤ x
              then i ← i + 1
                   exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```



(a) i p,j r
2 8 7 1 3 5 6 4
(b) p,i j r
2 8 7 1 3 5 6 4
(c) p,i j r
2 8 7 1 3 5 6 4
(d) p,i j r
2 8 7 1 3 5 6 4
(e) p i j r
2 1 7 8 3 5 6 4
(f) p i j r
2 1 3 8 7 5 6 4
(g) p i j r
2 1 3 8 7 5 6 4
(h) p i r
2 1 3 8 7 5 6 4
(i) p i r
2 1 3 4 7 5 6 8

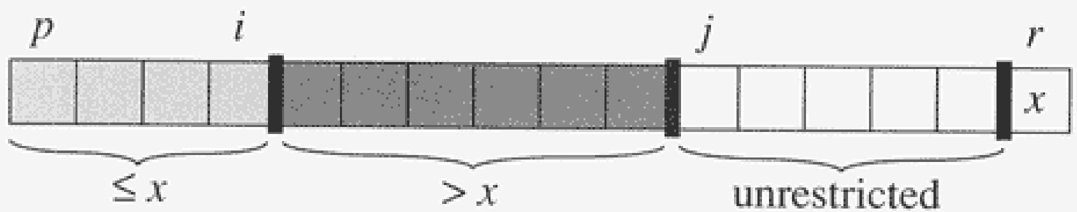


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p \, . . \, r]$. The values in $A[p \, . . \, i]$ are all less than or equal to x, the values in $A[i+1 \, . . \, j-1]$ are all greater than x, and $A[r] = x$. The values in $A[j \, . . \, r-1]$ can take on any values.

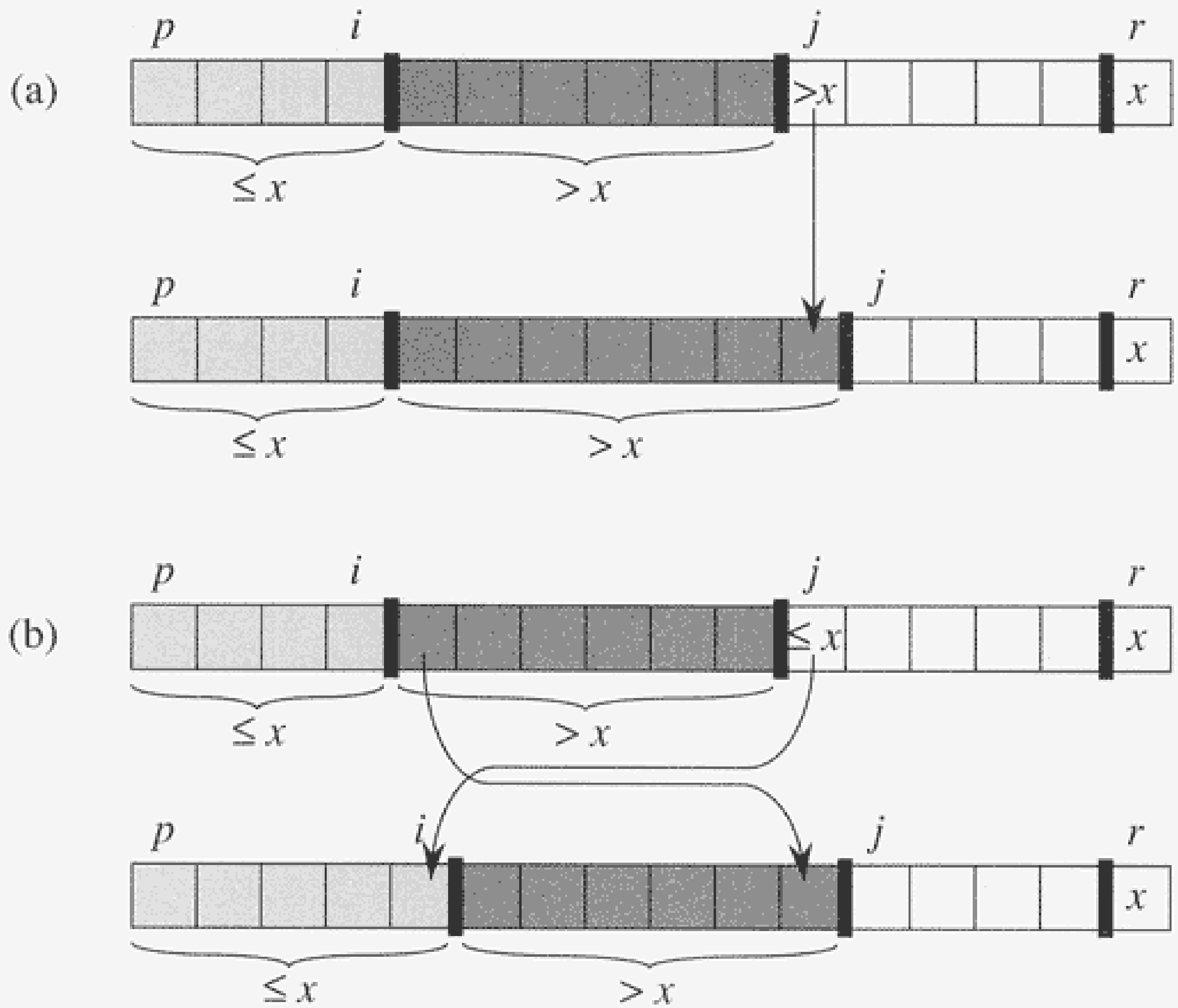
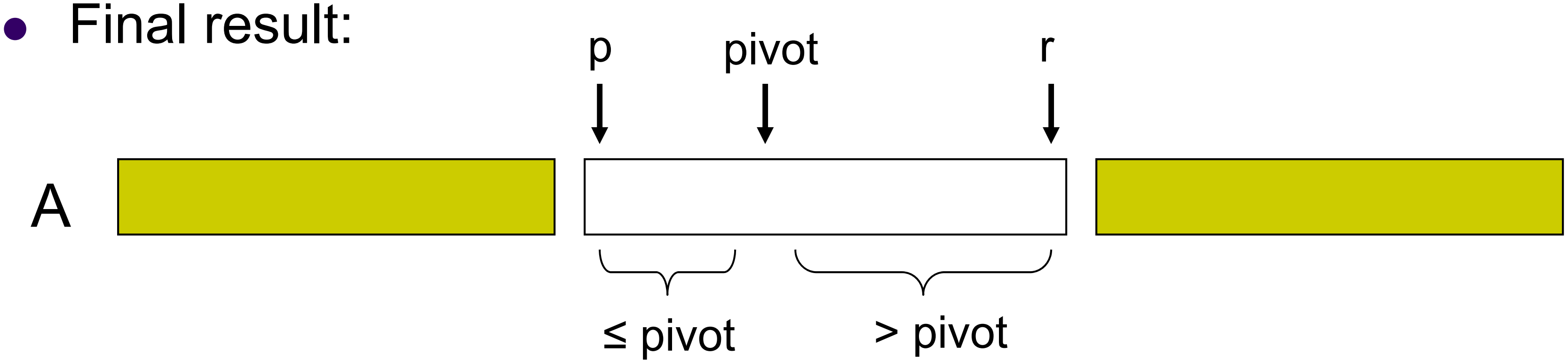


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Trace of Partition

```
PARTITION(A, p, r)
  i ← p − 1
  for j ← p to r − 1
          if A[j] ≤ A[r]
                  i ← i + 1
                  swap A[i] and A[j]
  swap A[i + 1] and A[r]
  return i + 1
```

- A[r] is called the ***pivot***
- Partitions the elements A[p…r-1] into two sets, those ≤ pivot and those > pivot
- Operates in place
- Final result:



… 5 7 1 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i

… 5 7 1 2 8 4 3 6 …

p r

$\text{Partition}(A, p, r)$

```
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

PARTITION(A, p, r)

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$

    if $A[j] \leq A[r]$

        $\boxed{i \leftarrow i + 1}$

        swap $A[i]$ and $A[j]$

swap $A[i+1]$ and $A[r]$

return $i + 1$

i j

… 5 7 1 2 8 4 3 6 …

p **r**

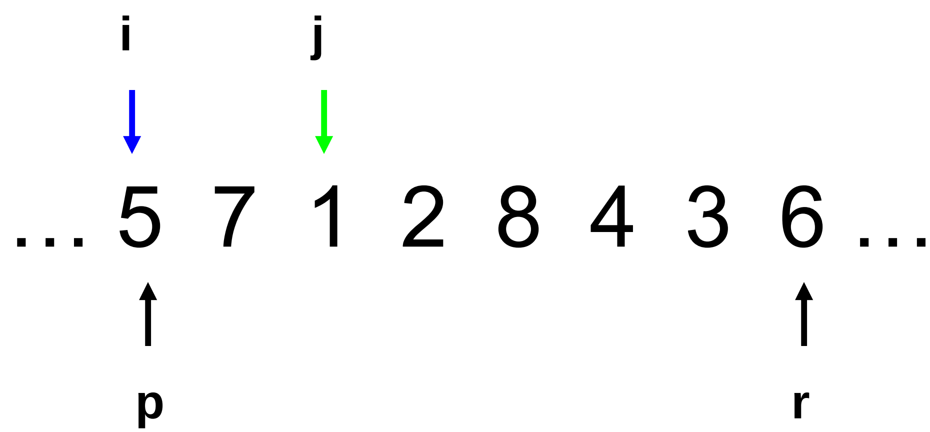
```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i **j**

… 5 7 1 2 8 4 3 6 …

p **r**

```
Partition(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```




```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 7 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 7 2 8 4 3 6 …

p r

PARTITION(A, p, r)

```
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 7 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

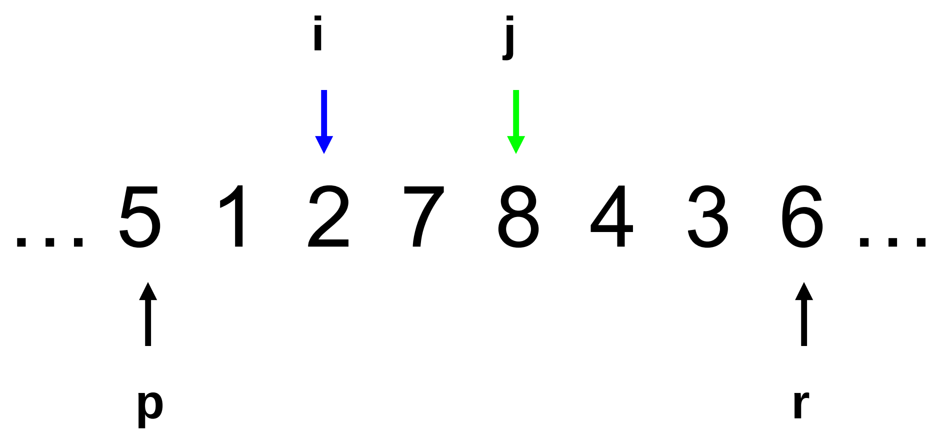
i j

… 5 1 2 7 8 4 3 6 …

p r

PARTITION(A, p, r)

```
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```




```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 2 7 8 4 3 6 …

p r

What’s happening?

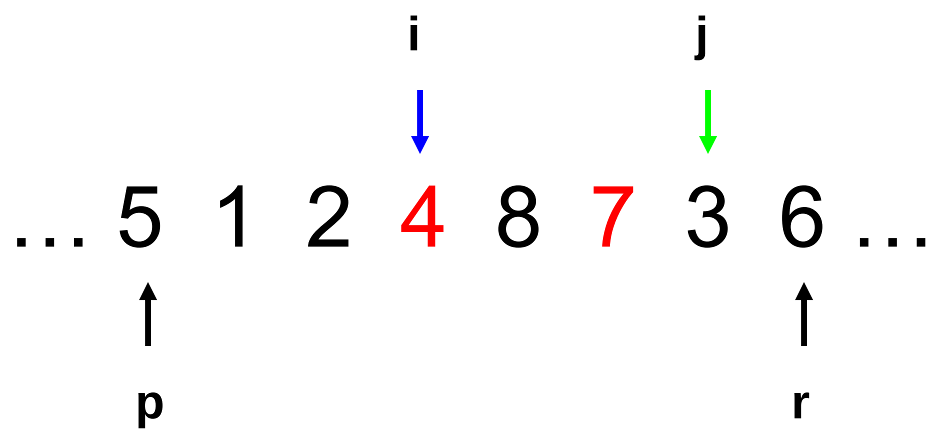
$$\dots\ \underbrace{\underset{p}{5}\ \ 1\ \ \overset{i}{2}}_{\le\ \text{pivot}}\ \ \underbrace{7\ \ \overset{j}{8}}_{>\ \text{pivot}}\ \ \underbrace{4\ \ 3}_{\text{unprocessed}}\ \ \underset{r}{6}\ \dots$$

i j

… 5 1 2 7 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```



PARTITION(A, p, r)

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$

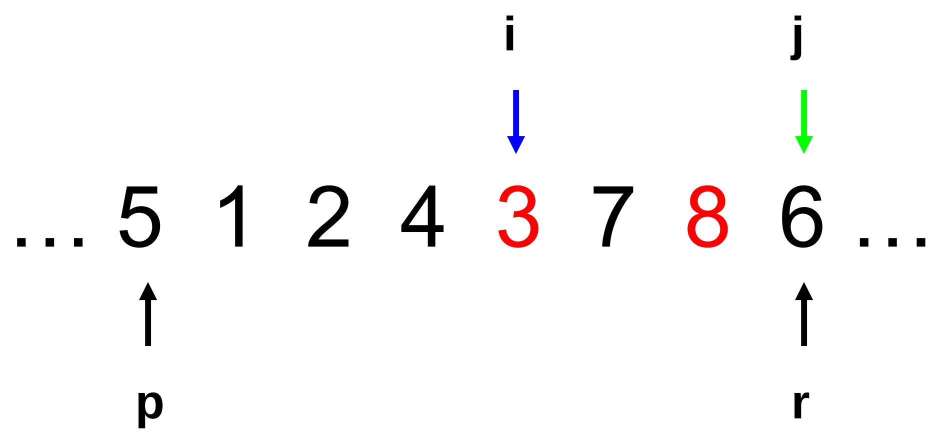
  $\boxed{\textbf{if } A[j] \leq A[r]}$

    $i \leftarrow i + 1$

    swap $A[i]$ and $A[j]$

swap $A[i+1]$ and $A[r]$

return $i + 1$




```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

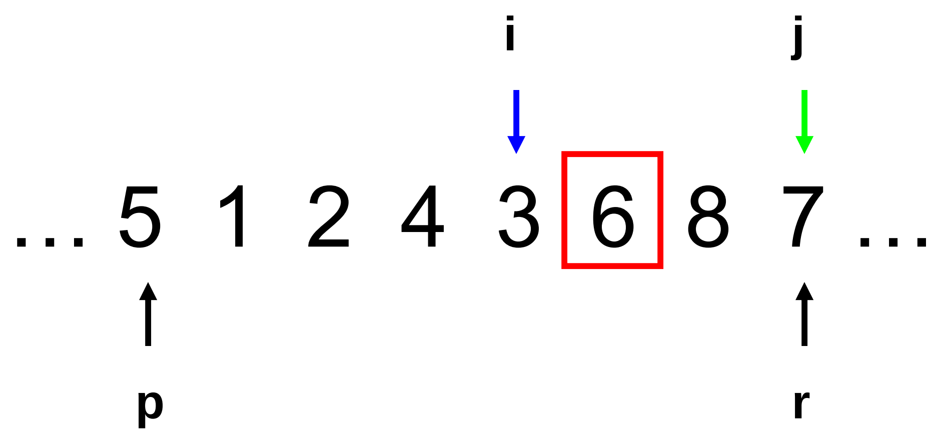
i j

… 5 1 2 4 3 6 8 7 …

p r

PARTITION(A, p, r)

```
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```




```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Trace of Quicksort

Quicksort

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

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
    if A[j] ≤ A[r]
        i ← i + 1
        swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

8 5 1 3 6 2 7 4

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

8	5	1	3	6	2	7	$\boxed{4}$

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

$$1 \quad 3 \quad 2 \quad \boxed{4} \quad 6 \quad 8 \quad 7 \quad 5$$

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

$$\boxed{1 \quad 3 \quad 2} \; 4 \; \boxed{6 \quad 8 \quad 7 \quad 5}$$

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

1 3 $\boxed{2}$ 4 6 8 7 5

QUICKSORT(A, p, r)

```
1  if p < r
2      q ← PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

$[\,1 \quad \boxed{2} \quad 3\,] \quad 4 \quad [\,6 \quad 8 \quad 7 \quad 5\,]$

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

$\boxed{1}$ 2 $\boxed{3}$ 4 $\boxed{6 \quad 8 \quad 7 \quad 5}$

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

1 2 3 4 6 8 7 5

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

1 2 3 4 6 8 7 5

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

1 2 3 4 6 8 7 5

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

1 2 3 4 5 8 7 6

What happens here?

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

1 2 3 4 5 $\boxed{8\ 7\ 6}$

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

1 2 3 4 5 8 7 6

$\text{QUICKSORT}(A, p, r)$

if $p < r$

$q \leftarrow \text{PARTITION}(A, p, r)$

$\text{QUICKSORT}(A, p, q - 1)$

$\text{QUICKSORT}(A, q + 1, r)$

1 2 3 4 5 6 7 8

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

1 2 3 4 5 6 $\boxed{7 \quad 8}$

QUICKSORT(A, p, r)

1 **if** $p < r$

2 $q \leftarrow$ PARTITION(A, p, r)

3 QUICKSORT$(A, p, q-1)$

4 QUICKSORT$(A, q+1, r)$

Quicksort Analysis

(1) “Bad” split on sorted lists

(2) “Bad” split vs Uneven split

(3) Average case analysis

Sorted List : Partition (Beginning and ending of each iteration)

i	p j			r
	1	2	3	4

	p i j			r
	1	2	3	4

	p i	j		r
	1	2	3	4

	p	ij		r
	1	**2**	3	4

	p	i	j	r
	1	2	3	4

	p		i j	r
	1	2	**3**	4

	p		i j	r
	1	2	3	**4**

Red: O(n) comparisons and O(n) (trivial) swaps for each call of Partition;
$T(n) = T(n-1) + O(n)$;
$O(n^2)$ swaps for quicksort

Reverse Sorted List : Partition (Beginning and ending of each iteration)

i	p j			r
	4	3	2	1

	p i j			r
	4	3	2	1

	p i	j		r
	4	3	2	1

	p i	j		r
	4	3	2	1

	p i		j	r
	4	3	2	1

	p i		j	r
	4	3	**2**	1

	p i		j	r
	1	3	2	**4**

Red: O(n) comparisons and
one swap for each call of Partition;
$T(n) = T(n-1) + O(n)$
O(n) swaps for quicksort
$O(n^2)$ comparisons for quicksort

Quicksort Average case?

- How close to "even" split do we need to maintain an O(n log n) running time?
  - Say the Partition procedure always splits the array into some constant ratio b-to-a, e.g., 9-to-1
  - What is the recurrence?

$$T(n) \leq T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$

$$E.g., T(n) \leq T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + cn$$

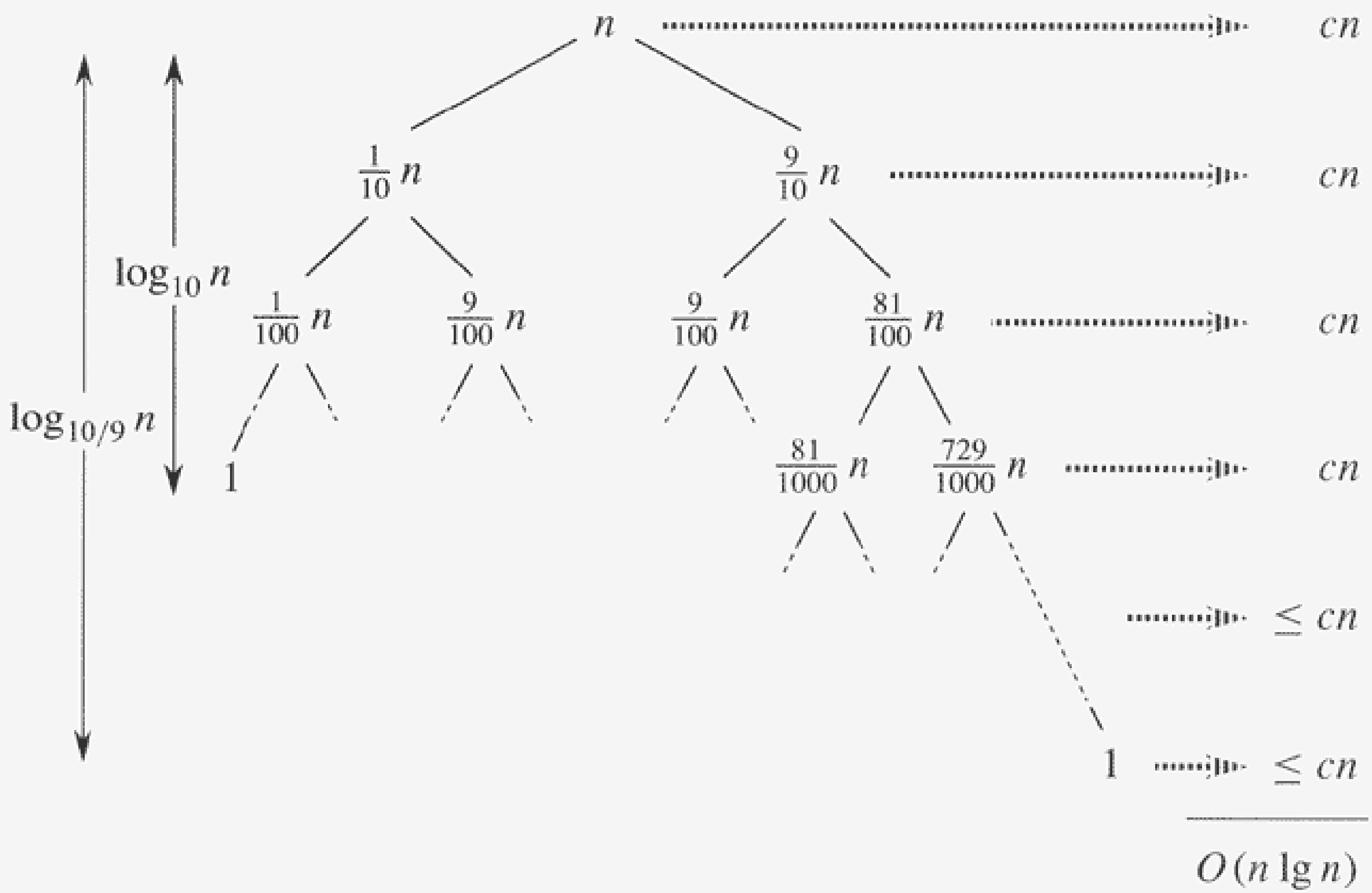


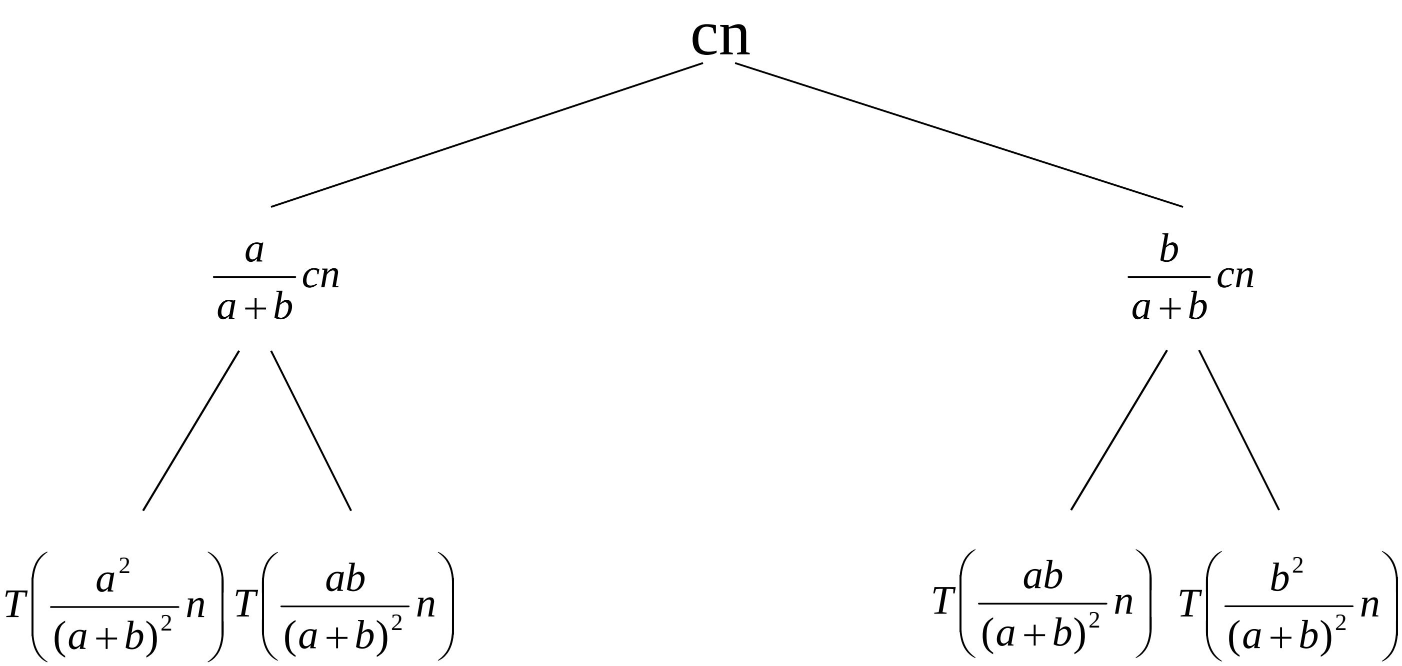
Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$

cn

$T(\frac{a}{a+b}n)$ $T(\frac{b}{a+b}n)$

$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$



$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$

$$\text{cn}$$

$$\frac{a}{a+b}cn \qquad \frac{b}{a+b}cn$$

$$\frac{a^2}{(a+b)^2}cn \qquad \frac{ab}{(a+b)^2}cn \qquad \frac{ab}{(a+b)^2}n \qquad \frac{b^2}{(a+b)^2}n$$

$$T(\frac{a^3}{(a+b)^2}n) \quad T(\frac{a^2b}{(a+b)^2}n) \quad T(\frac{a^2b}{(a+b)^2}n) \quad T(\frac{ab^2}{(a+b)^2}n) \quad T(\frac{a^2b}{(a+b)^2}n) \quad T(\frac{ab^2}{(a+b)^2}n) \quad T(\frac{ab^2}{(a+b)^2}n) \quad T(\frac{b^3}{(a+b)^2}n)$$

$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$

Level 0: cn

Level 1: $= cn\left(\frac{a}{a+b}\right) + cn\left(\frac{b}{a+b}\right) = cn$

Level 2: $= cn\left(\frac{a^2}{(a+b)^2}\right) + cn\left(\frac{ab}{(a+b)^2}\right) + cn\left(\frac{ab}{(a+b)^2}\right) + cn\left(\frac{b^2}{(a+b)^2}\right)$

$$= cn\left(\frac{a^2+2ab+b^2}{(a+b)^2}\right) = cn\left(\frac{(a+b)^2}{(a+b)^2}\right) = cn$$

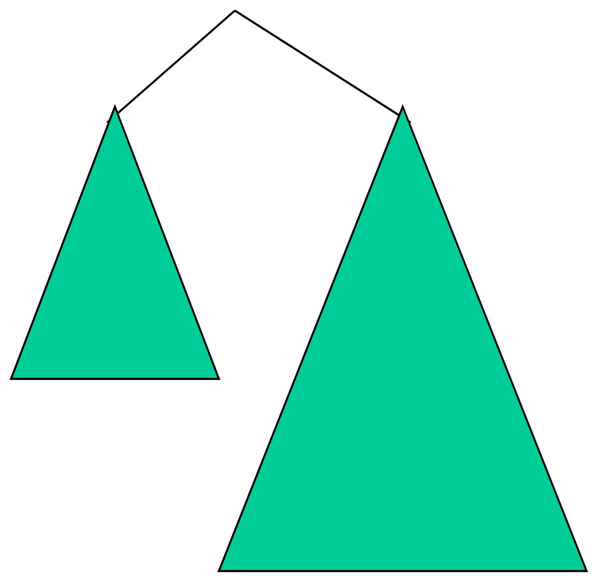
Level 3: $= cn\left(\frac{(a+b)^2 a + (a+b)^2 b}{(a+b)^3}\right)$

$$= cn\left(\frac{(a+b)(a+b)^2}{(a+b)^3}\right) = cn$$

Level d: $= cn\left(\frac{(a+b)^d}{(a+b)^d}\right) = cn$

What is the depth of the tree?

- Leaves will have different heights
- Want to pick the deepest leaf
- Assume $a < b$



What is the depth of the tree?

- Assume a < b

$$\left(\frac{b}{a+b}\right)^d n = 1$$

$$\dots$$

$$d = \log_{\frac{a+b}{b}} n$$

Mixing "Good" and "Bad" Splits

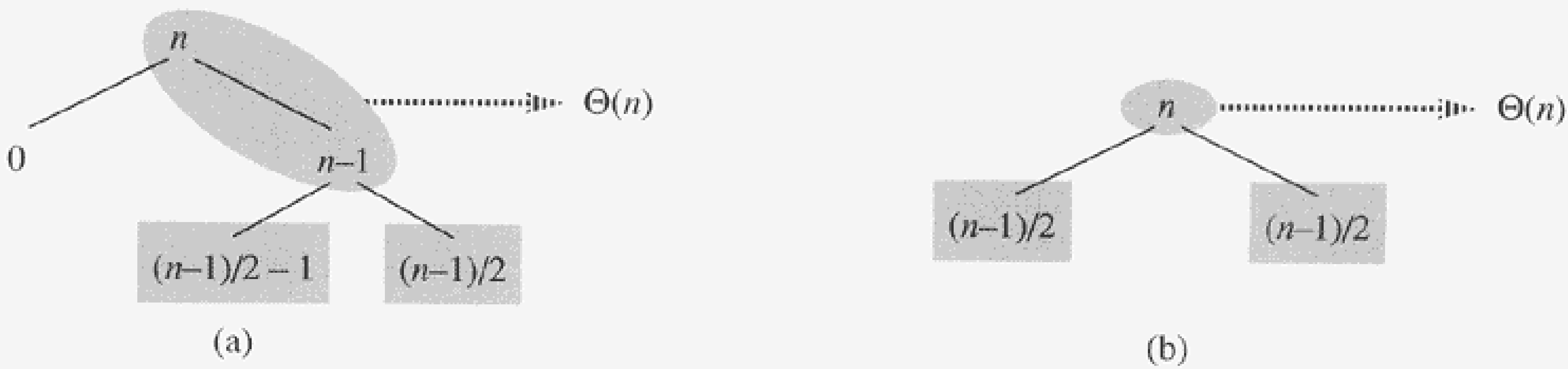
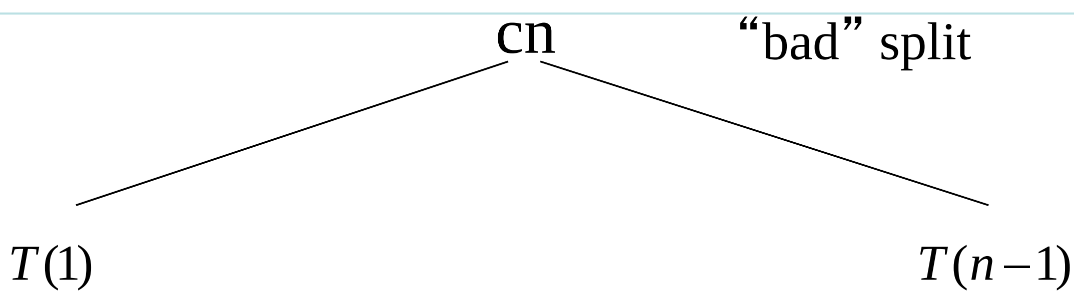


Figure 7.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n-1$. The partitioning of the subarray of size $n-1$ costs $n-1$ and produces a "good" split: subarrays of size $(n-1)/2-1$ and $(n-1)/2$. **(b)** A single level of a recursion tree that is very well balanced. In both parts, the partitioning cost for the subproblems shown with elliptical shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with square shading, are no larger than the corresponding subproblems remaining to be solved in (b).

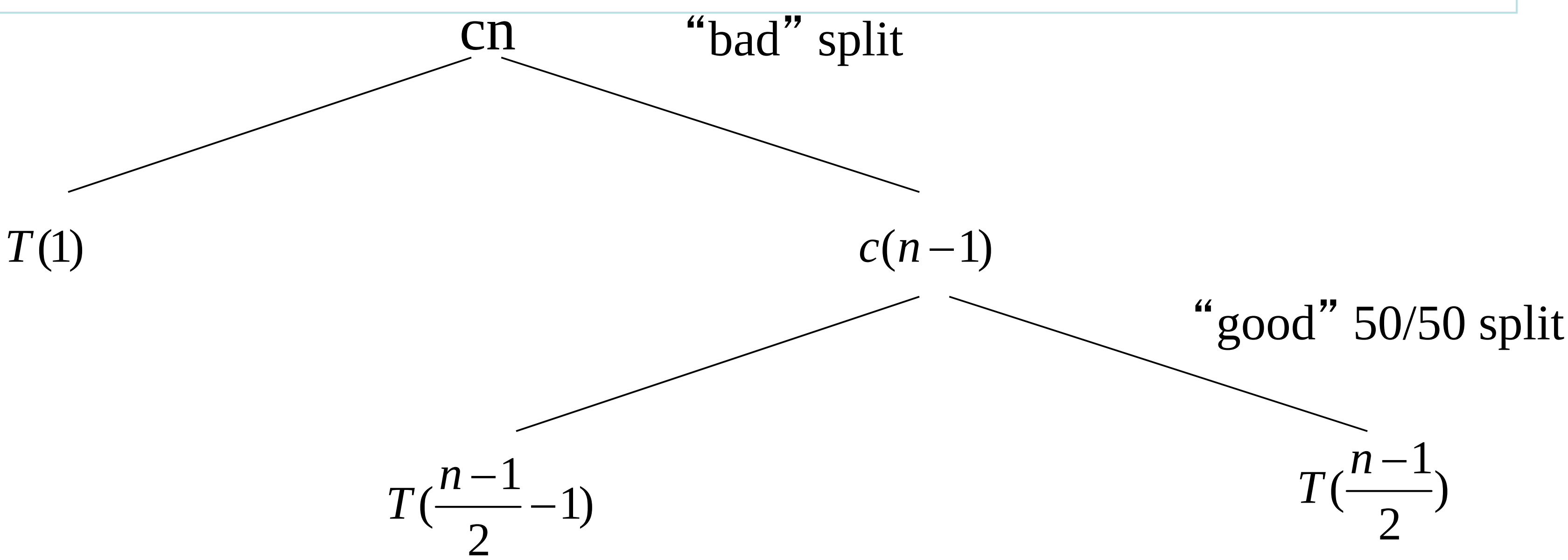
Quicksort average case



Limited number of “bad splits” followed by a “good split” is a slower “good split”!!

Quicksort average case :

“bad split” + “good split” => “good split”



$$T(n) = \overbrace{T(1) + T(\frac{n-1}{2}-1) + T(\frac{n-1}{2})}^{\text{recursion cost}} + \overbrace{\Theta(n) + \Theta(n-1)}^{\text{partition cost}}$$

Average Case Behavior: Probabilistic Analysis vs. Average Case Design: Randomized Algorithms

- Probabilistic analysis of a deterministic algorithm:
  - assume probability distribution on the inputs
- Randomized algorithm design:
  - use random choices in the algorithm
    - E.g., generate random permutation of the input
    - E.g., pick a pivot randomly at each step

Randomized Algorithms

- Randomized algorithms make use of a randomizer (such as a random number generator).
- Some of the decisions of the algorithm depend on the output of the randomizer.
  - The output may vary from run to run.
  - The execution time may vary from run to run.

Why Randomness?

Making good decisions could be expensive.

A randomized algorithm is **faster**.

- Minimum spanning trees

  A linear time randomized algorithm,

  but no known linear time deterministic algorithm.

- Primality testing

  A randomized polynomial time algorithm,

  but it takes thirty years to find a deterministic one.

- Volume estimation of a convex body

  A randomized polynomial time **approximation** algorithm,

  but no known deterministic polynomial time approximation algorithm.

Why Randomness?

In many practical problems,
we need to deal with HUGE input,
and don't even have time to read it once.
But can we still do something useful?

Sublinear algorithm: randomness is essential.

- **Fingerprinting**: verifying equality of strings, pattern matching.
- **The power of two choices**: load balancing, hashing.
- **Random walk**: check connectivity in log-space.

Two Major Classes of Randomized Algorithms

- Las Vegas algorithms:
  - Always produce the same (correct) output for the same input.
  - Execution time depends on the randomizer.
  - Goal is to minimize the odds of encountering worst case performance.
  - *Example*: Randomized quick sort

Two Major Classes of Randomized Algorithms

- Monte Carlo algorithms:
  - Give an incorrect answer with very low probability.
  - Typically does not display variation in execution time for a particular input.
  - Goal is to provide an answer that has a high likelihood of being correct in a reasonable time.
  - *Example*: Determining primality of an integer with several hundred decimal digits.

Analysis of Probabilistic/Randomized Algorithms

- Analysis is often complex and typically requires concepts from
  - Probability
  - Statistics
  - Number theory
- Uses (pseudo-random generator) *Random(a,b)* which returns an integer between a and b, inclusive with each integer being equally likely.
- Assume cost of producing a single random value is constant.

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  exchange A[r] ↔ A[i]
3  return PARTITION(A, p, r)
```

Pick pivot randomly

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

```
HOARE-PARTITION(A, p, r)
x ← A[p]
i ← p − 1
j ← r + 1
while TRUE
      do repeat j ← j − 1
            until A[j] ≤ x
         repeat i ← i + 1
            until A[i] ≥ x
         if i < j
            then exchange A[i] ↔ A[j]
            else return j
```

```
QUICKSORT'(A, p, r)
while p < r
      do ▷ Partition and sort left subarray.
         q ← PARTITION(A, p, r)
         QUICKSORT'(A, p, q − 1)
         p ← q + 1
```