Quicksort

Description

- Divide and conquer based algorithm, the steps can be summarized as:
- *Divide*: given an array A[p, …, r], partition it into two (possibly empty) sub-arrays A[p, …, q-1] and A[q+1, …, r] such that A[p…q-1] contains all the elements less than or equal to A[q] and A[q+1, …, r] contains all the elements greater than or equal to A[q]
- *Conquer*: Sort the two sub-arrays A[p, …, q-1] and A[q+1, …, r] by recursive calls to the quicksort procedure
- *Combine*: as the resulting sub-arrays are already sorted, nothing is to be done in the combine step

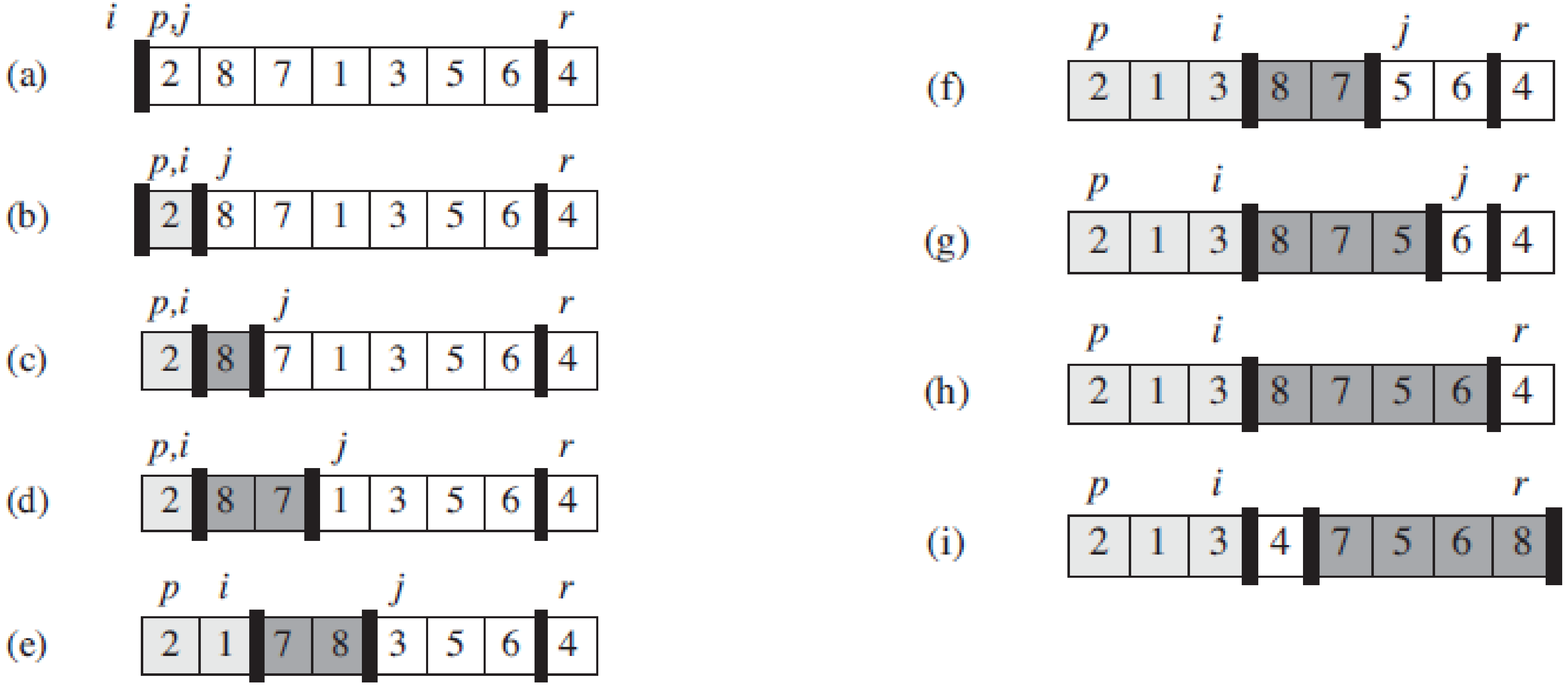
Algorithm

QUICKSORT(A, p, r)

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

Partition Function

```
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
```



(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

Proof of Correctness

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 $A[k] = x$.

Performance Analysis

- Worst-case Partitioning: Will happen when each time the partition function returns an element A[q] such that all the other elements are either less than or equal to A[q] or greater than or equal to A[q] thus always resulting into n-1 elements in one sub-array and 0 in the other
- The recurrence relation for the worst-case partitioning can be written as:

$$T(n) = T(n-1) + T(0) + \theta(n)$$

$$= T(n-1) + \theta(n)$$

Complexity Analysis of worst case

- Solution to the above recurrence is $T(n) = \theta(n^2)$
- Thus, quicksort take $\theta(n^2)$ time in the worst-case
- This worst case occurs only when the input array is already sorted
- It should be noted that insertion sort runs in $\theta(n)$ time in this situation

Best-case partitioning

- The best case occurs when the pivot element A[q] partitions the array A into two (almost) equal-size subarrays each time
- The performance of quicksort improves considerably in this case and it runs as fast as the merge sort
- The recurrence for the best case can be given as:

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$$

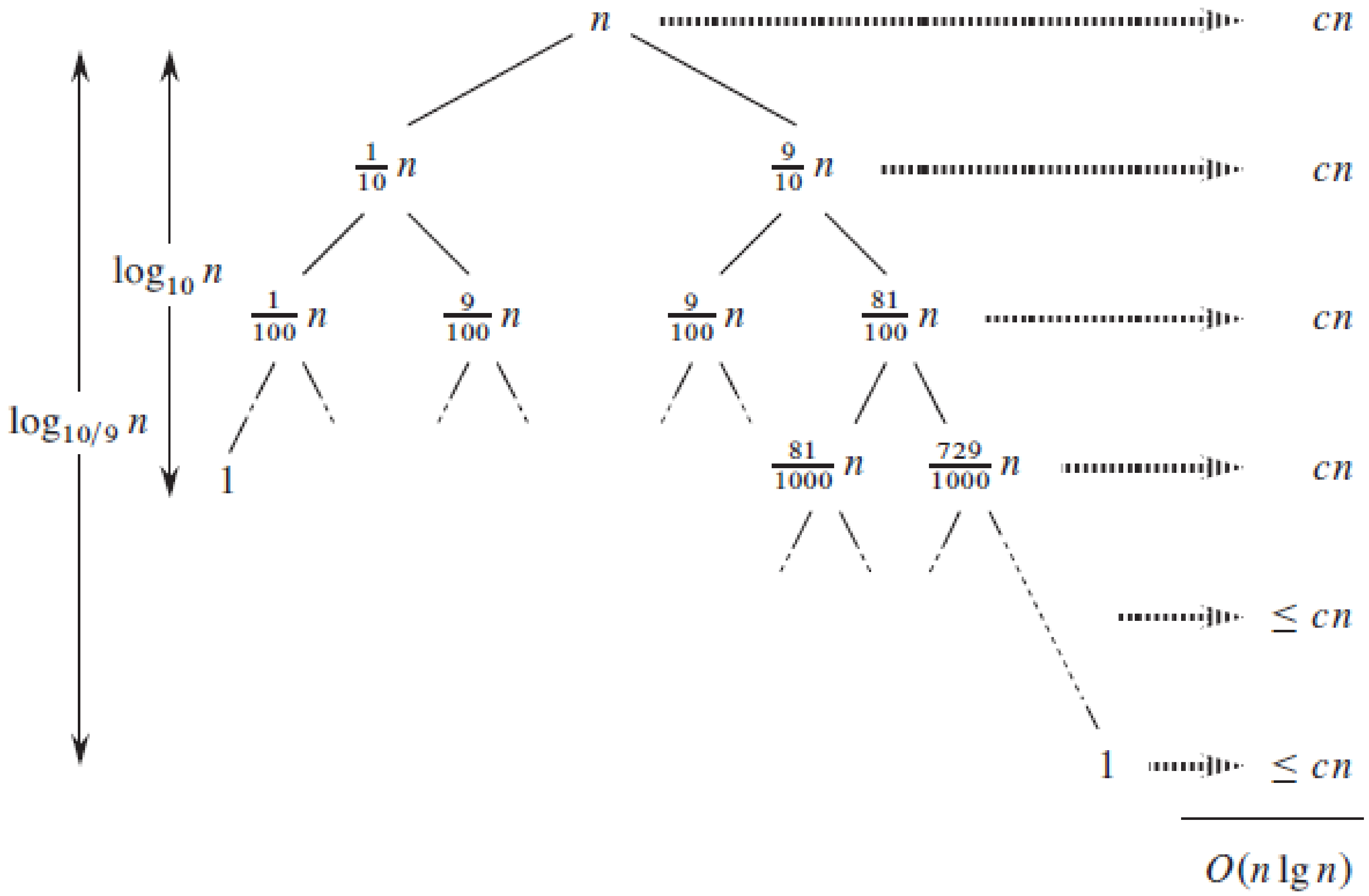
- Solution to this recurrence, $T(n) = \theta(nlgn)$
- Thus, we get asymptotically faster algorithm if the partitioning is balanced

Balanced Partitioning

- Even for very unbalanced partitioning like 9-to-1 at each level, we get good performance from quicksort i.e. O(nlogn)

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$

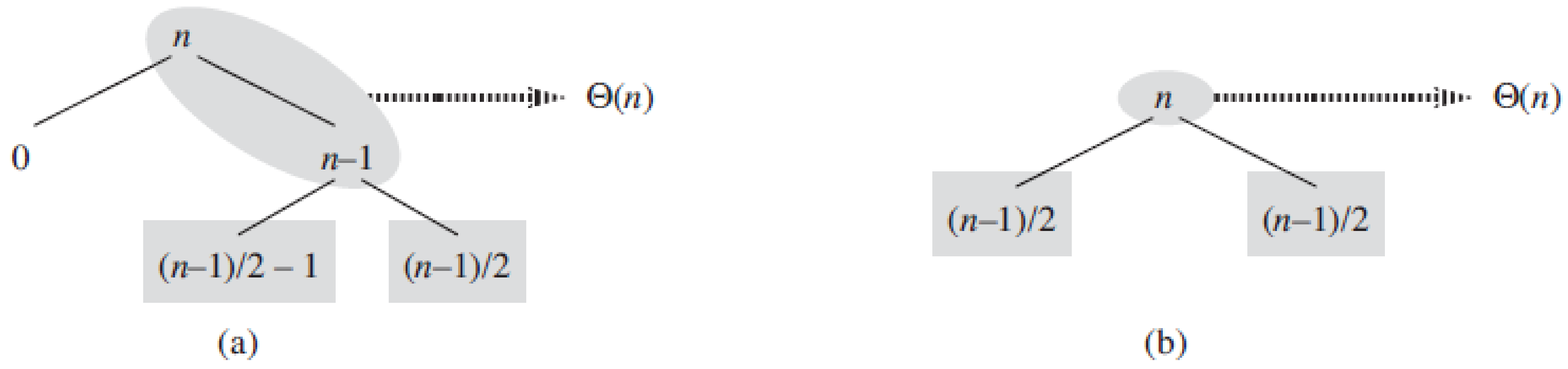
- For even bad split like 99-to-1 as well, we get similar performance
- O(nlgn) time complexity is always achieved if the partition has constant proportion



n
cn
1/10 n
9/10 n
cn
log_10 n
1/100 n
9/100 n
9/100 n
81/100 n
cn
log_10/9 n
1
81/1000 n
729/1000 n
cn
≤ cn
1
≤ cn
O(n lg n)

Average Case

- For a random input, partitioning in constant proportion at each level is highly unlikely
- Some partitions could be unbalanced, some could be reasonably balanced and some fairly balanced
- The performance depends on the relative ordering of input numbers and not their actual value
- We consider all combinations of the input sequence equally likely
- As in next example, combination of alternative good and bad splits also results into O(nlgn) time complexity



Partitioning: 0, (n-1)/2 – 1, (n-1)/2
Time = $\theta(n) + \theta(n-1) = \theta(n)$
The extra cost can be absorbed in the constant and thus the time complexity is O(nlgn)