Divide and Conquer

Text Book References:

Fundamentals of Computer Algorithms – Sartaj Sahni

Introduction to Algorithms – Thomas H Coremen

Unit II

Divide and conquer: The general method, Iterative and Divide and conquer for Binary search, Merge sort, Quick sort, Masters’ theorem.

Divide-and-Conquer

Divide the problem into a number of sub-problems

- Similar sub-problems of smaller size

Conquer the sub-problems

- Solve the sub-problems recursively
- Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner

Combine the solutions of the sub-problems

- Obtain the solution for the original problem

Divide and Conquer Strategy:

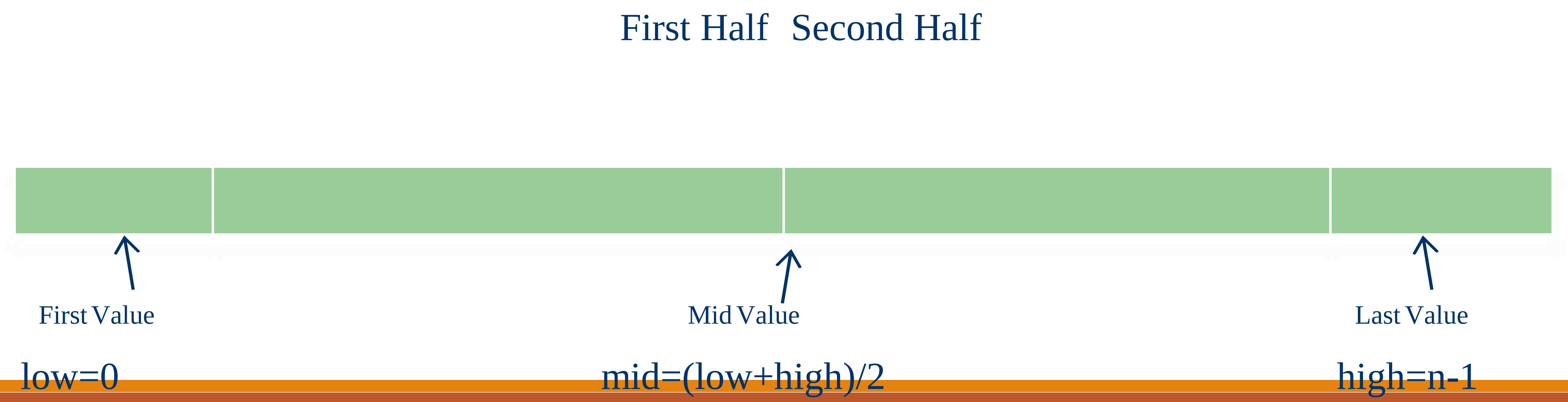
1) Divide : Breaking the problem into sub problems that are themselves smaller instances of the same type of problem.
2) Recursion : Recursively solving these sub problems.
3) Conquer : Appropriately combining their answers.

Control abstraction :

```
DAndC(P)
{
    if(small(P))
     // P is very small so that a solution
     // is obvious
    return solution(n);
    else
     divide the problem P into k
              sub problems P1, P2, ...Pk
     return (combine(DAndC(P1), DAndC(P2),
              ...DAndC(Pk)))
}
```

Binary Search

- This technique works better on sorted arrays and can be applied only on sorted arrays.
- Not applied on Linked Lists.
- Requires less number of comparisons than linear search.
- Efficiency: O(log2n).
- Logic behind the technique:



Binary Search

Iterative

```
BinarySearch (A, n, x)
{
    low ← 0
    high ← n-1
    while (low <= high)
    {
        mid ← (low + high)/2
        if (x == A[mid])
            return mid
        else if (x < A[mid])
            high ← mid - 1
        else
            low ← mid + 1
    }
    return -1
}
```

$O(\log_2 n)$

Recursive

```
BinarySearch (A, low, high, x)
{
    ~~mid ← low~~
    if (low > high)
        return -1
    mid ← (low + (high))/2
    if (x == A[mid])
        return mid
    else if x < A[mid]
        return BinarySearch(A, low, mid-1, x)
    else
        return BinarySearch(A, mid+1, high, x)
}
```

$T(n) = T(n/2) + c$

$T(0) = c$

$\Theta(\log_2 n)$

n	c
$n/2$	c
$n/4$	c
$\vdots$	$\vdots$
2	c
1	c
0	c

Binary Search

Recursive

```
BinarySearch(A, low, high, x)    — T(n)
{
    if (low > high) //  if (low = = high)
                    {   if (A[low] = = key)
                            return low;
        return -1       else return -1;
                    }
    else
        mid ← (low + mid)/2
        if (x = = A[mid])
            return mid
        else if (x < A[mid])
            return BinarySearch(A, low, mid-1, x)
        else
            return BinarySearch(A, mid+1, high, x)
}
```

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + 1 & n > 1 \end{cases}$$

$\Rightarrow \theta(\log n)$

or By using Master's theorem

$a = 1,\ b = 2,\ k = 0,\ p = 0$

$b^k = 2^0 = 1$

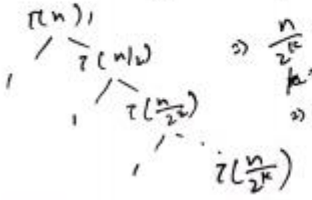
Case 2: $p > -1$

$T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

$= \theta(n^{\log_2 1} \log^1 n)$

$= \theta(n^0 \log n)$

$= \theta(\log n)$



$\Rightarrow \frac{n}{2^k} = 1$

$k = \log_2 n$

$\Rightarrow \theta(\log n)$

Merge Sort Approach

To sort an array $A[p\,..\,r]$:

Divide

- Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each

Conquer

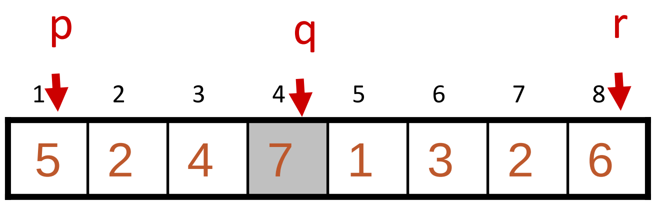
- Sort the subsequences recursively using merge sort
- When the size of the sequences is 1 there is nothing more to do

Combine

- Merge the two sorted subsequences

Merge Sort

Alg.: MERGE-SORT(A, p, r)



if p < r — Check for base case

then $q \leftarrow \lfloor (p + r)/2 \rfloor$ — Divide

MERGE-SORT(A, p, q) — Conquer

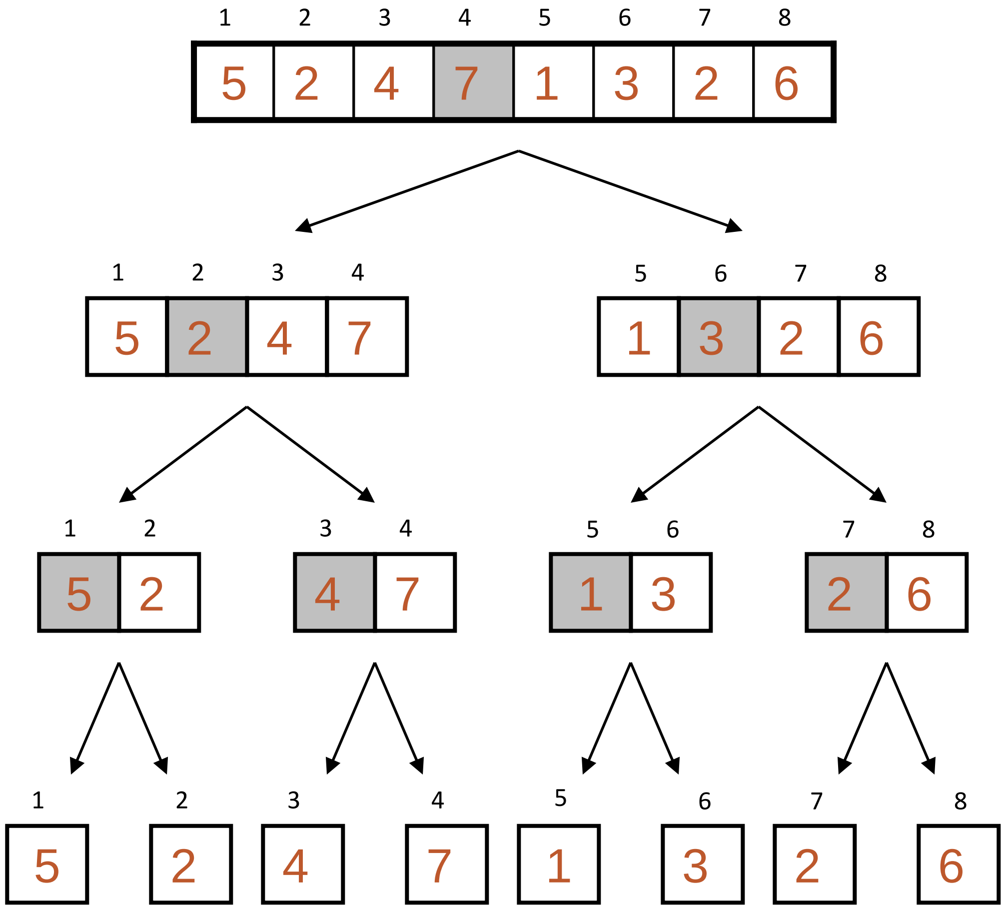
MERGE-SORT(A, q + 1, r) — Conquer

MERGE(A, p, q, r) — Combine

Initial call: MERGE-SORT(A, 1, n)

Example – n Power of 2

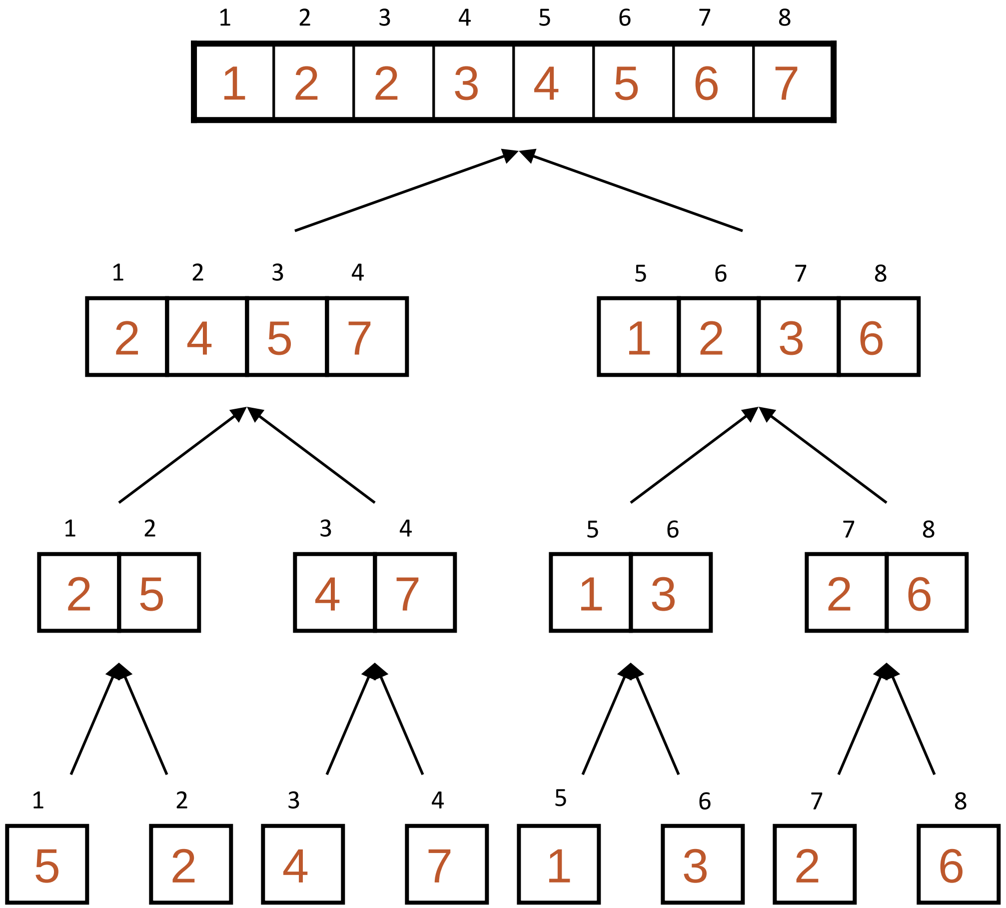
Divide



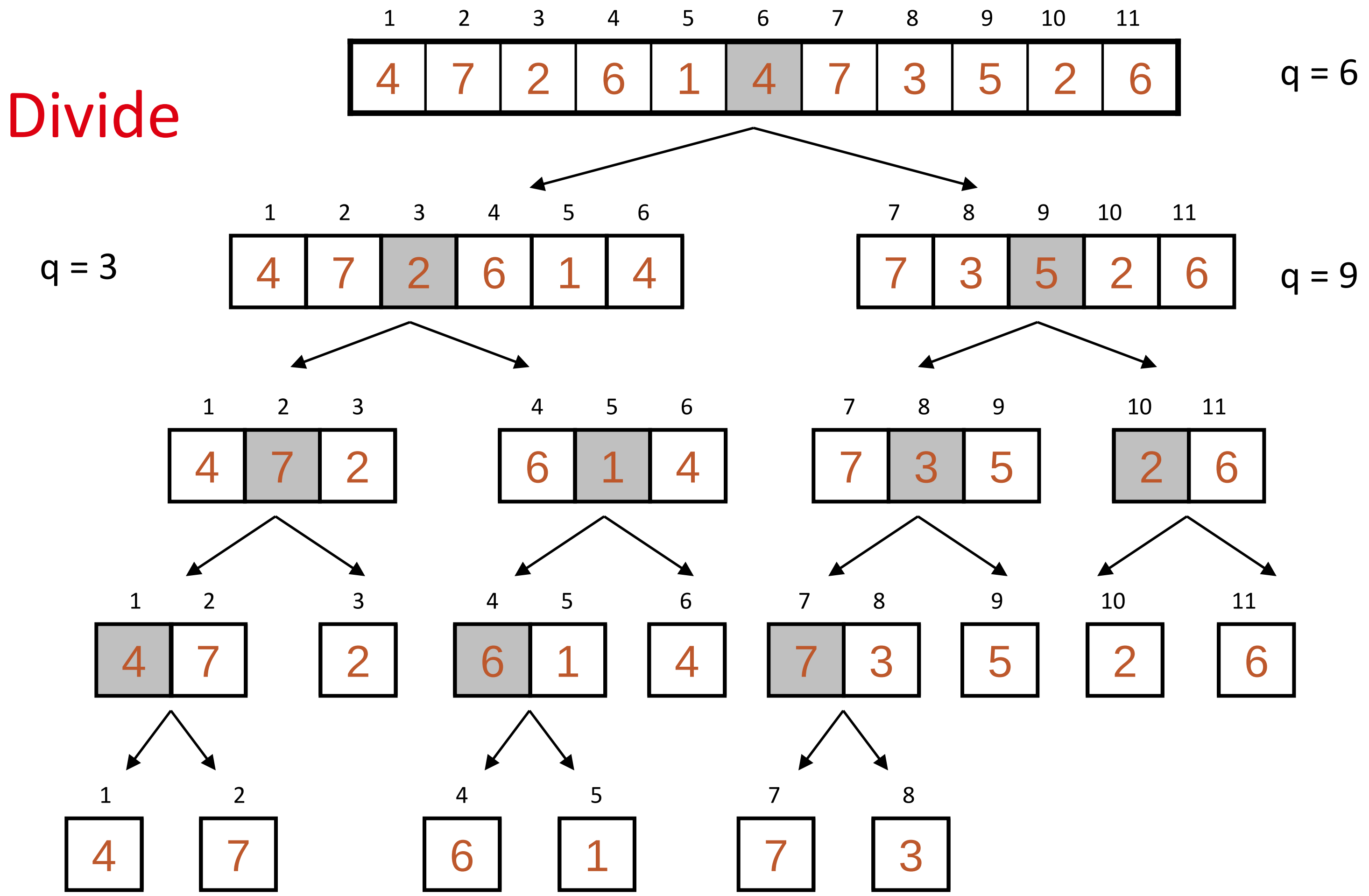
q = 4

Example – *n* Power of 2

Conquer
and
Merge

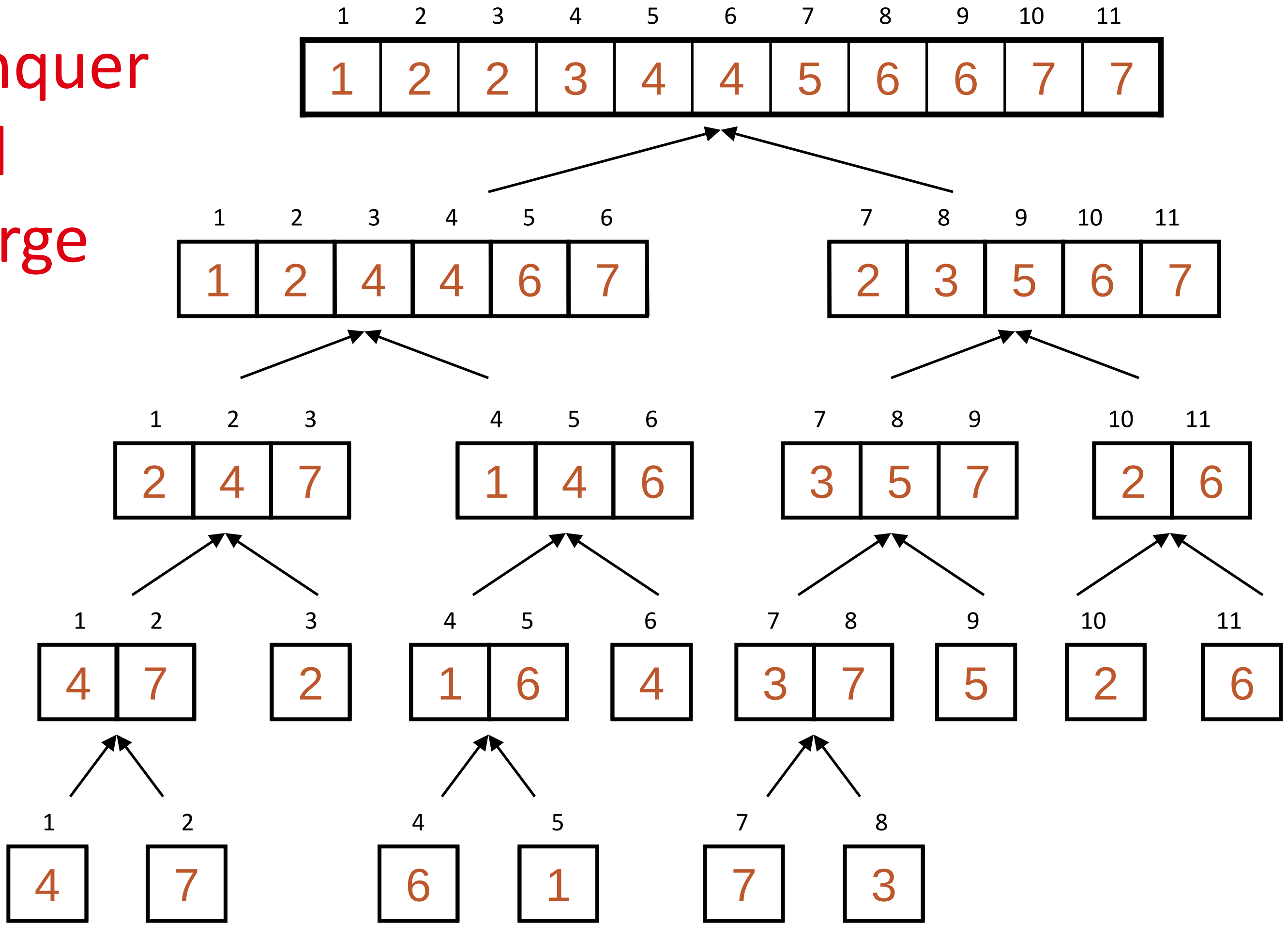


Example – *n* Not a Power of 2



Example – n Not a Power of 2

Conquer and Merge



Merging

p			q				r
1	2	3	4	5	6	7	8
2	4	5	7	1	2	3	6

Input: Array A and indices p, q, r such that $p \le q < r$

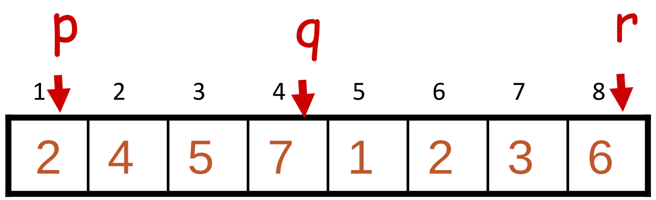
- Subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ are sorted

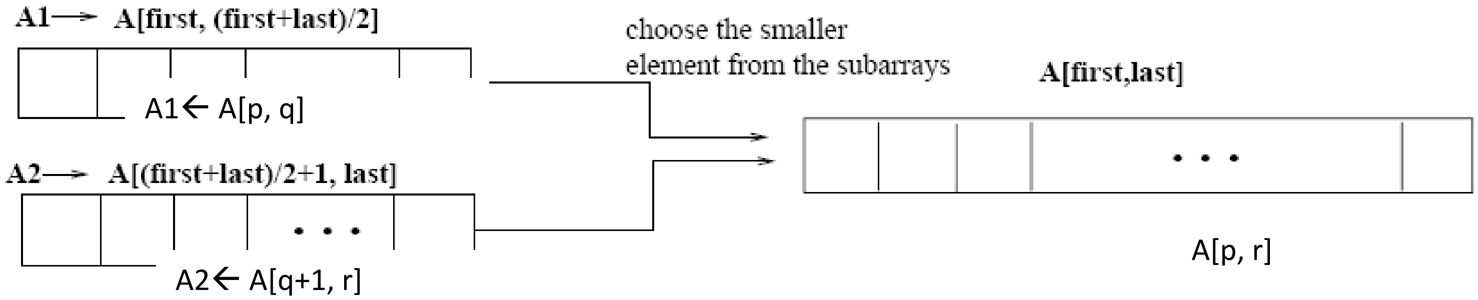
Output: One single sorted subarray $A[p \,..\, r]$

Merging

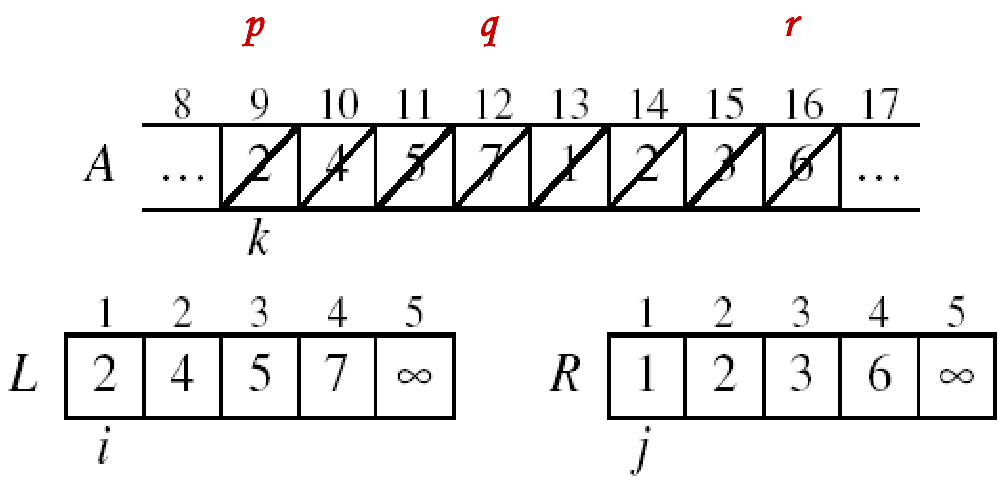
Idea for merging:

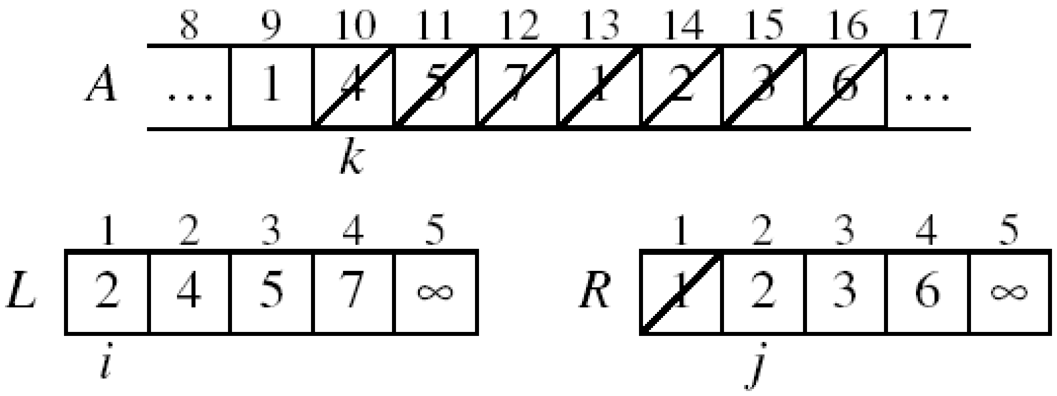
- Two piles of sorted cards
  - Choose the smaller of the two top cards
  - Remove it and place it in the output pile
- Repeat the process until one pile is empty
- Take the remaining input pile and place it face-down onto the output pile



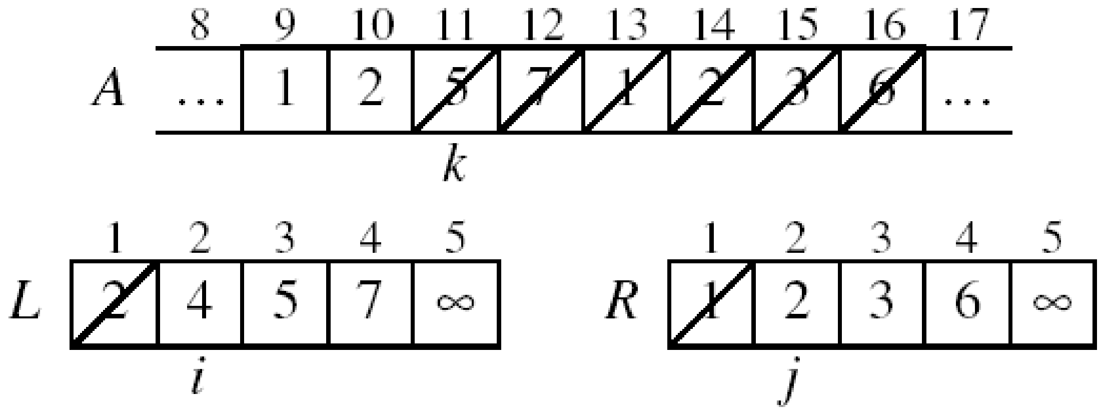


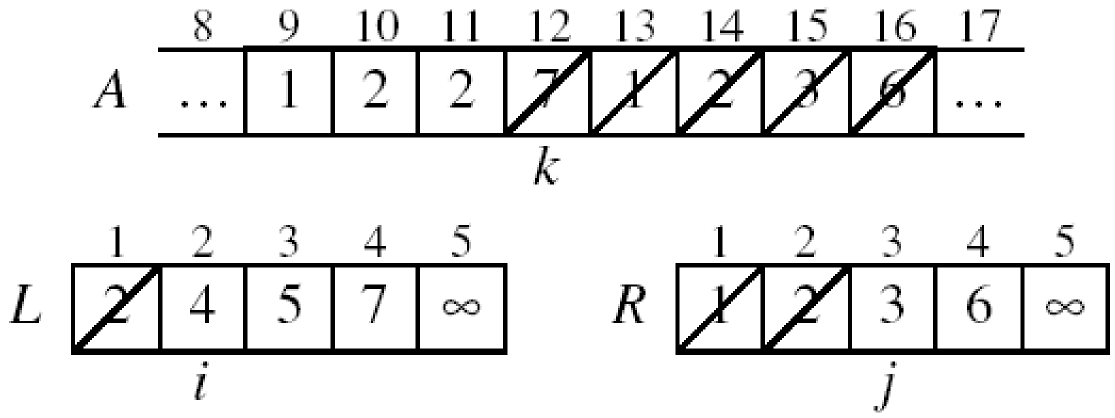
Example: MERGE(A, 9, 12, 16)



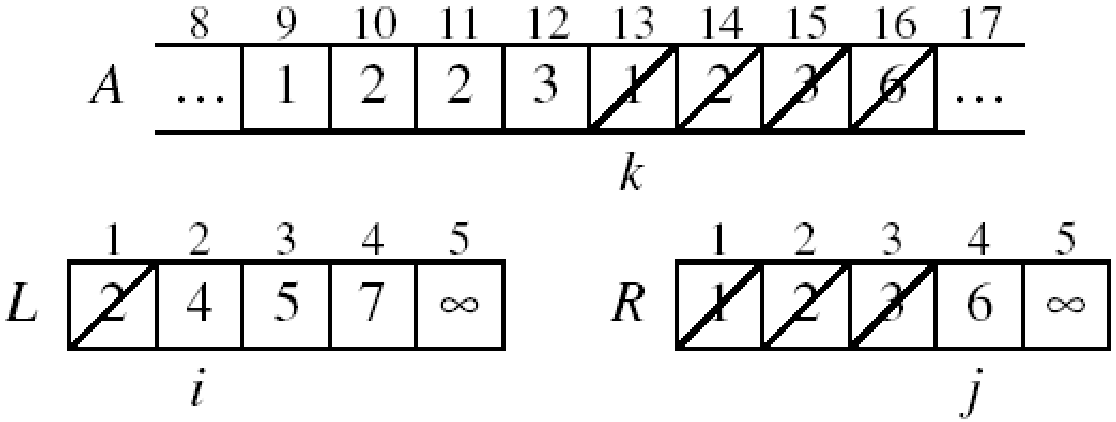


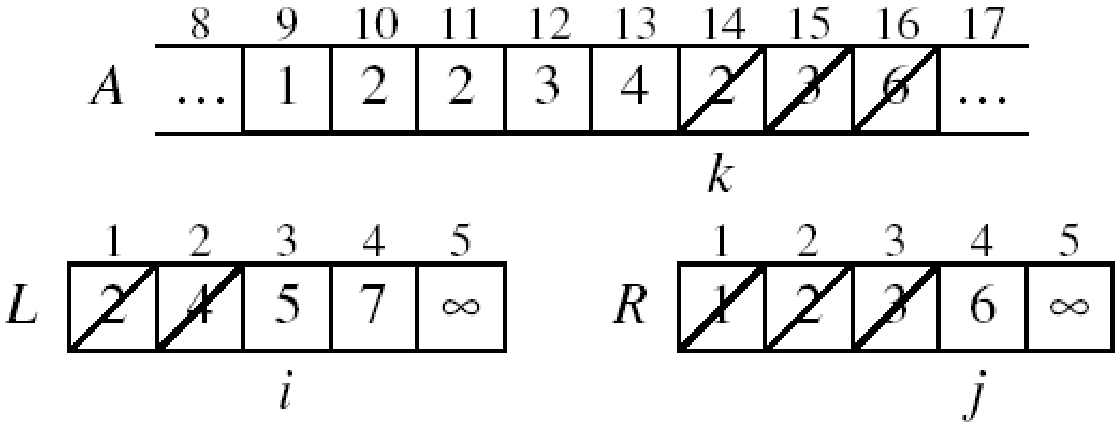
Example: MERGE(A, 9, 12, 16)



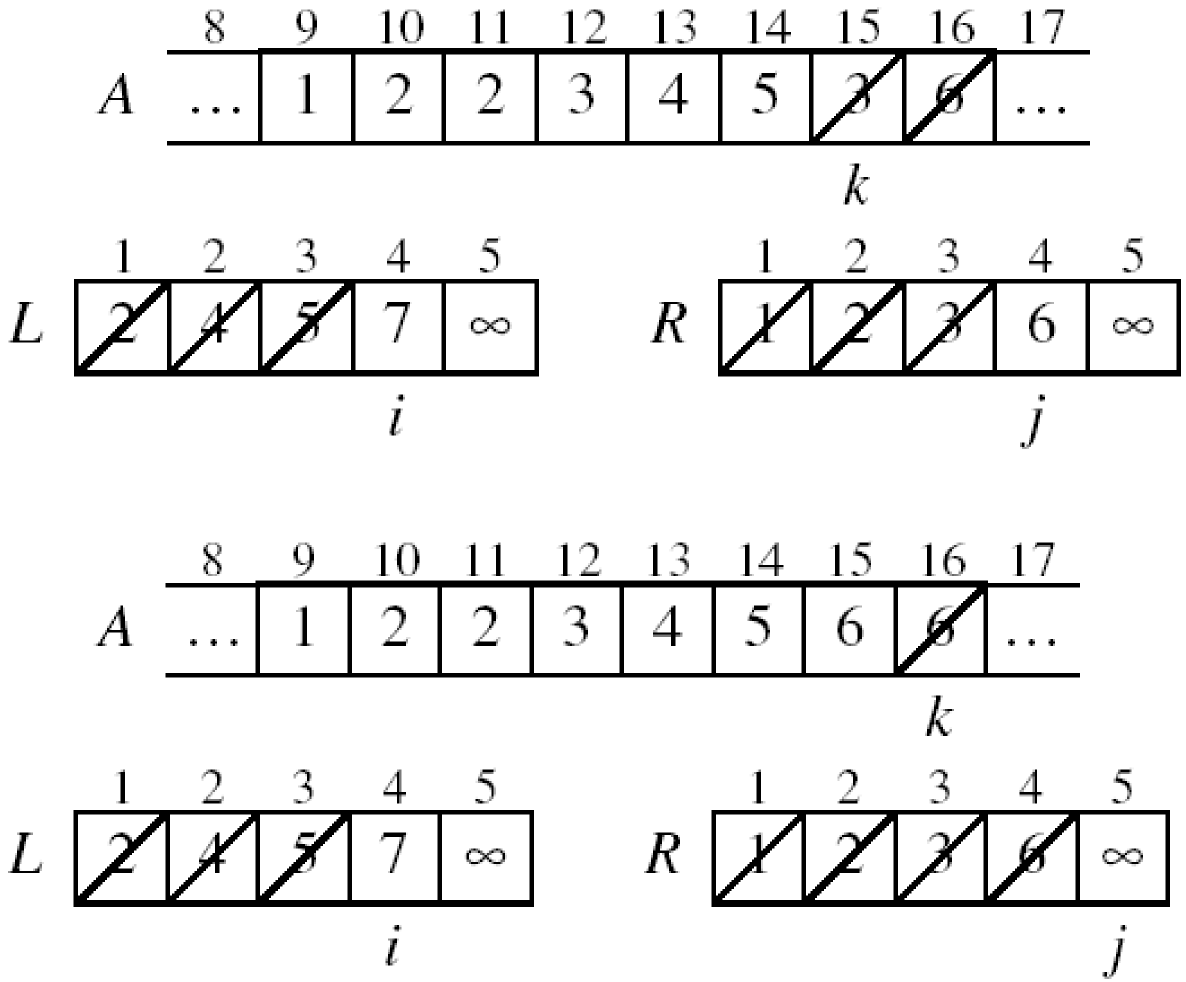


Example (cont.)

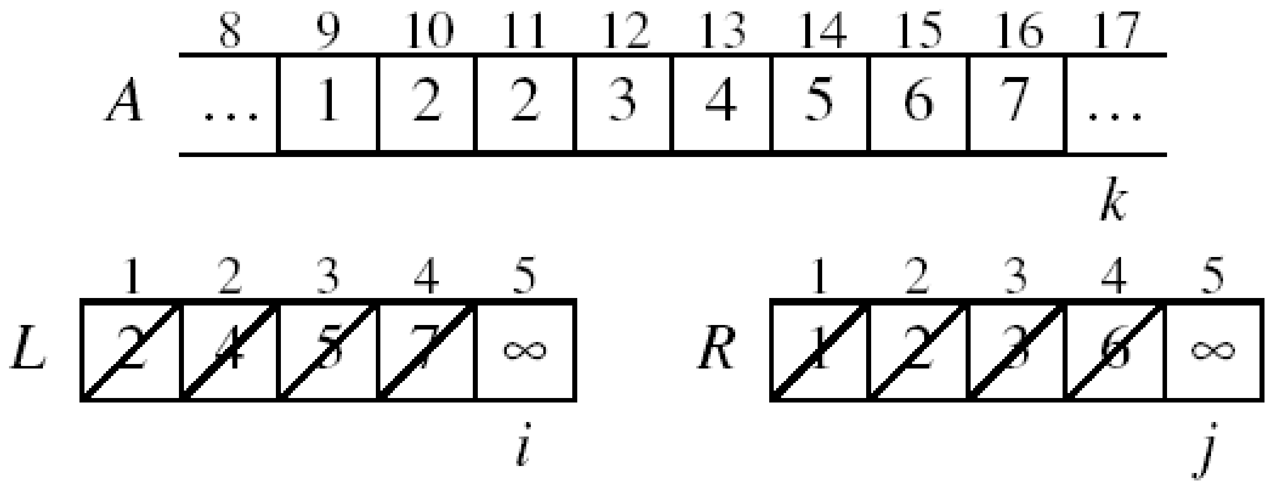




Example (cont.)



Example (cont.)

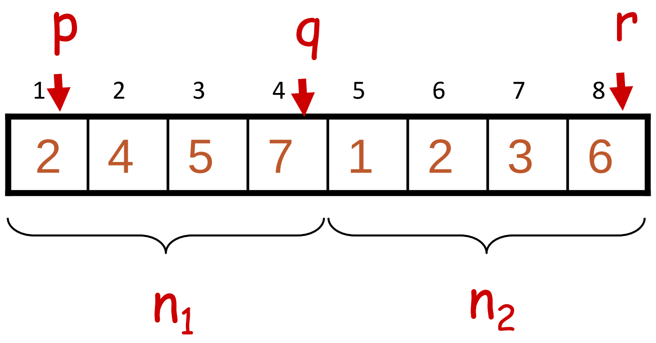


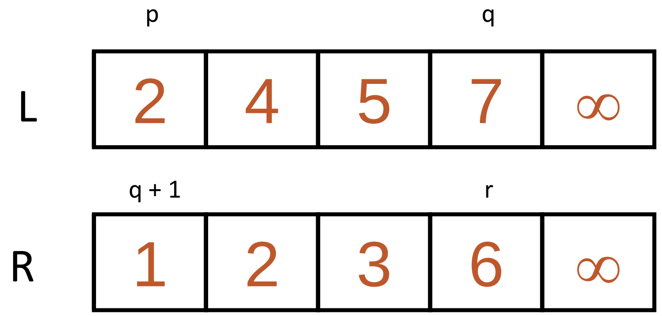
Done!

Merge - Pseudocode

Alg.: MERGE(A, p, q, r)

Compute n_1 and n_2
Copy the first n_1 elements into
$L[1 \,..\, n_1 + 1]$ and the next n_2 elements into $R[1 \,..\, n_2 + 1]$
$L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$; $j \leftarrow 1$
for $k \leftarrow p$ **to** r
    do if $L[\,i\,] \le R[\,j\,]$
        then $A[k] \leftarrow L[\,i\,]$
            $i \leftarrow i + 1$
        else $A[k] \leftarrow R[\,j\,]$
            $j \leftarrow j + 1$





Running Time of Merge (assume last **for** loop)

Initialization (copying into temporary arrays):

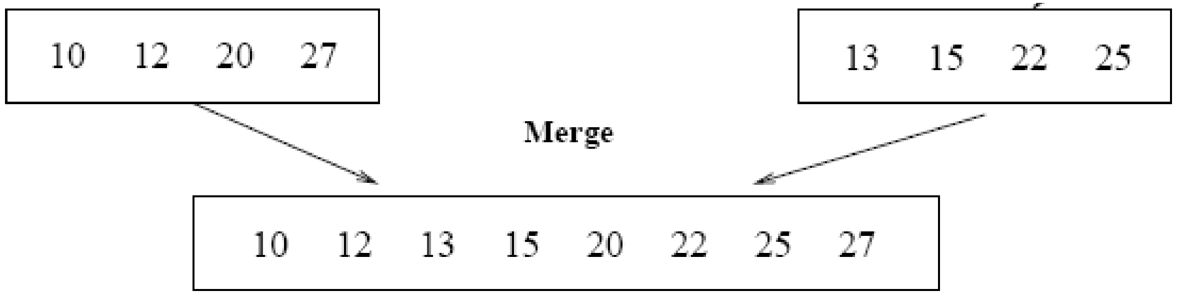
- $\Theta(n_1 + n_2) = \Theta(n)$

Adding the elements to the final array:

- n iterations, each taking constant time $\Rightarrow \Theta(n)$

Total time for Merge:

- $\Theta(n)$



Analyzing Divide-and Conquer Algorithms

The recurrence is based on the three steps of the paradigm:

- $T(n)$ – running time on a problem of size n
- **Divide** the problem into a subproblems, each of size n/b: takes $D(n)$
- **Conquer** (solve) the subproblems $aT(n/b)$
- **Combine** the solutions $C(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

MERGE-SORT Running Time

Divide:

- compute q as the average of p and r: $D(n) = \Theta(1)$

Conquer:

- recursively solve 2 subproblems, each of size $n/2 \Longrightarrow 2T(n/2)$

Combine:

- MERGE on an n-element subarray takes $\Theta(n)$ time $\Longrightarrow C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solve the Recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Use Master's Theorem:

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\lg n)$

Merge Sort - Discussion

Running time insensitive of the input

Properties:

1. Uses divide and conquer
2. It is stable
3. It is not an in-space algorithms.
4. Does not require random access of data.

Advantages:

- Guaranteed to run in $\Theta(nlgn)$

Disadvantage

- Requires extra space $\approx$N

Sorting Files That are Almost in Order

Selection sort?

- NO, always takes quadratic time

Bubble sort?

- NO, bad for some definitions of “almost in order”
- Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A

Insertion sort?

- YES, takes linear time for most definitions of “almost in order”

Mergesort or custom method?

- Probably not: insertion sort simpler and faster

Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- Lecture 2 provides several ways to find a good upper bound on $T(n)$.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$cn$$

$$T(n/2) \qquad\qquad T(n/2)$$

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

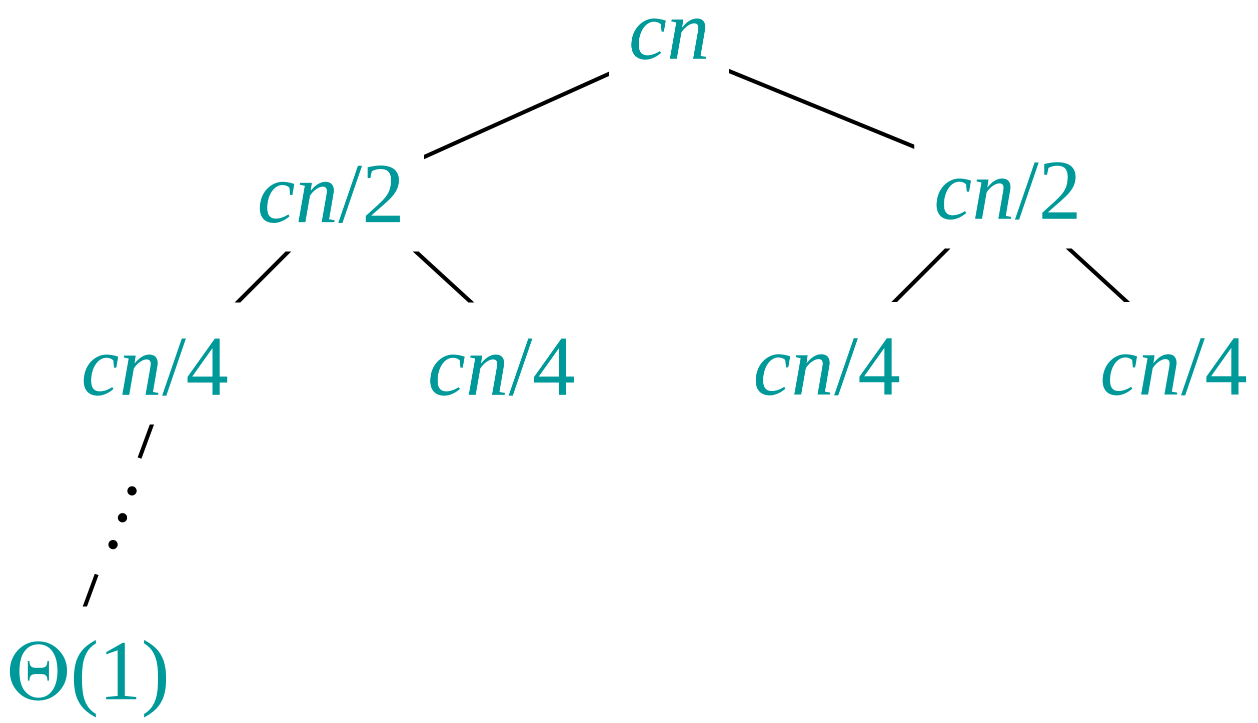
$$cn$$

$$cn/2 \qquad\qquad cn/2$$

$$T(n/4) \quad T(n/4) \quad T(n/4) \quad T(n/4)$$

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

cn

$cn/2$ $cn/2$

$cn/4$ $cn/4$ $cn/4$ $cn/4$

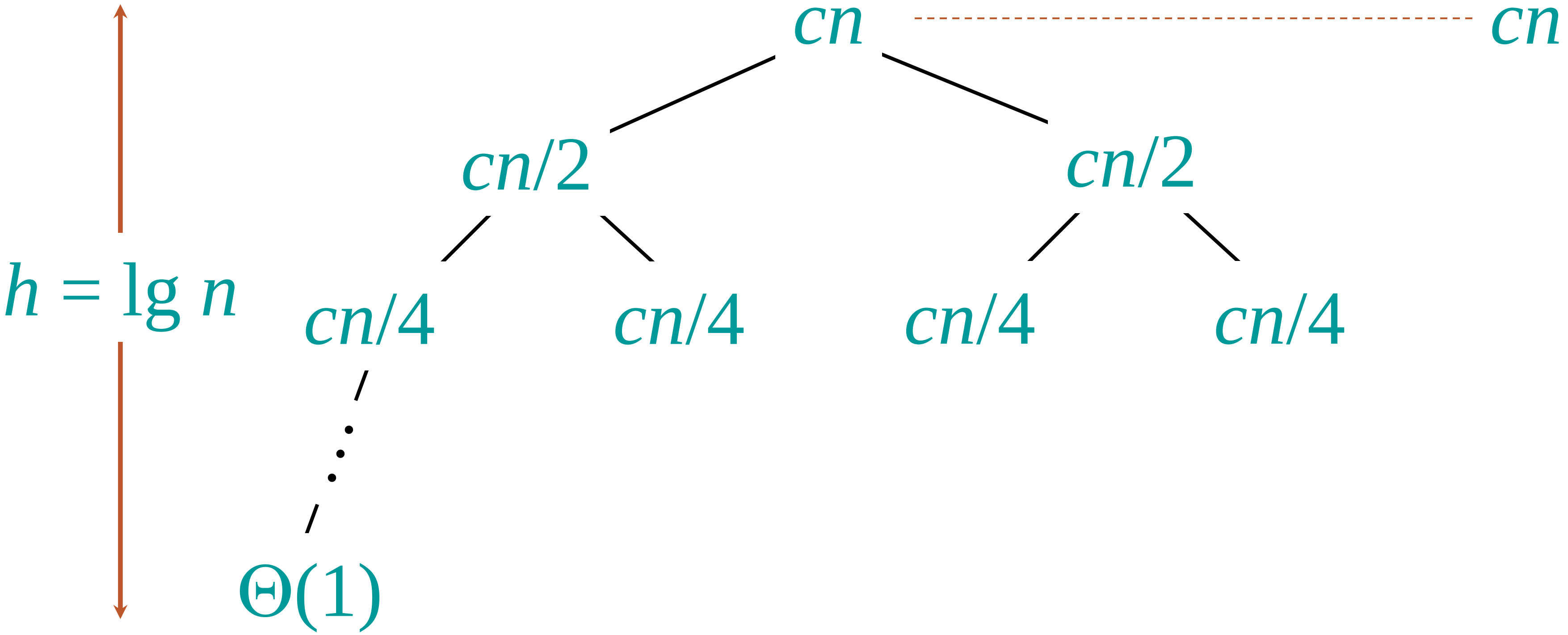
$\vdots$

$\Theta(1)$

$h = \lg n$

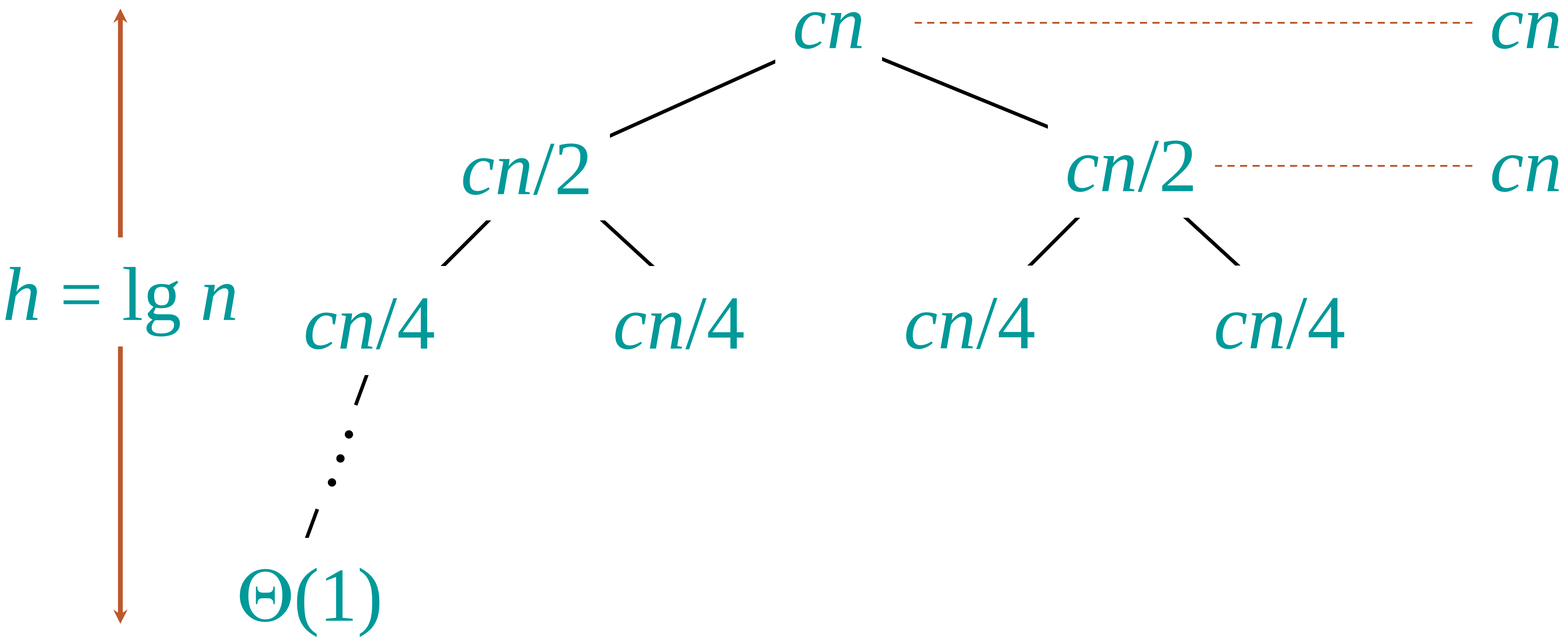
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



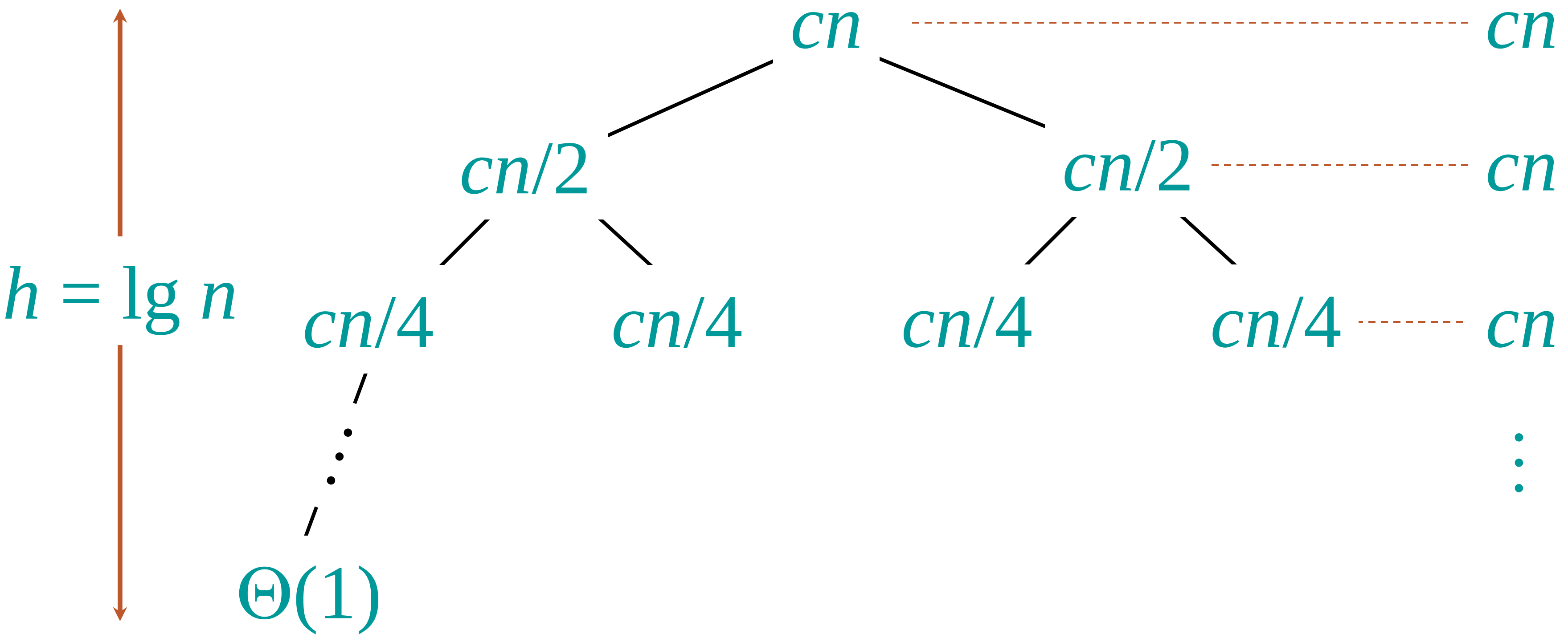
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



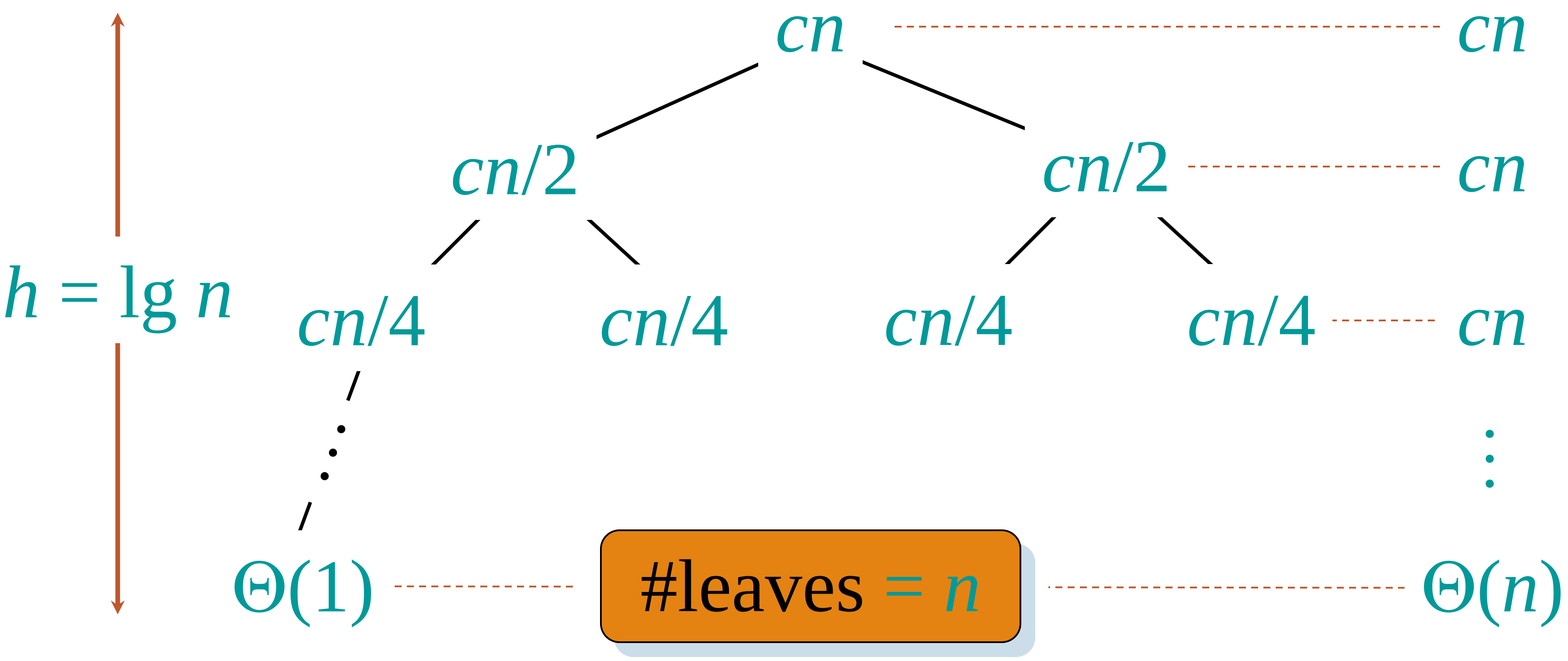
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



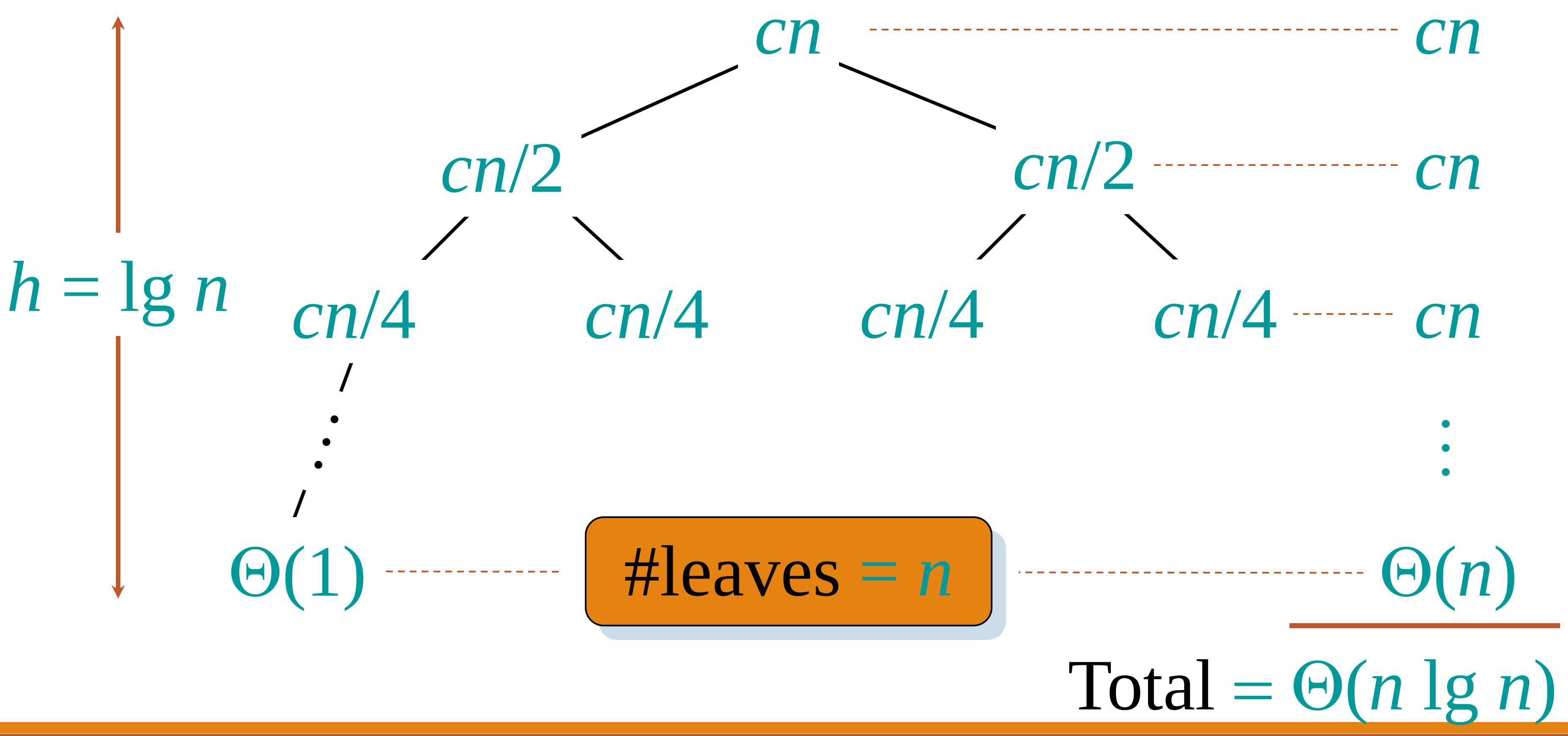
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

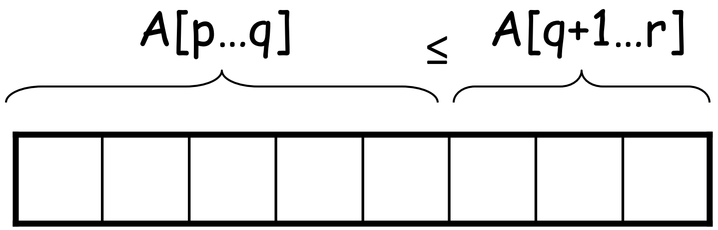


Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



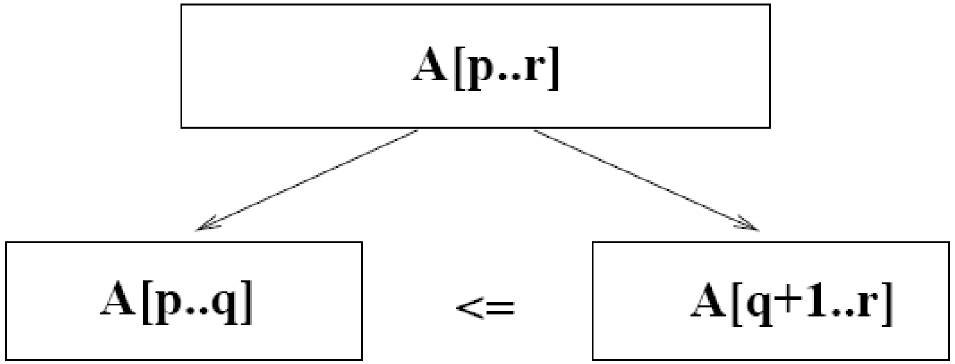
Quicksort



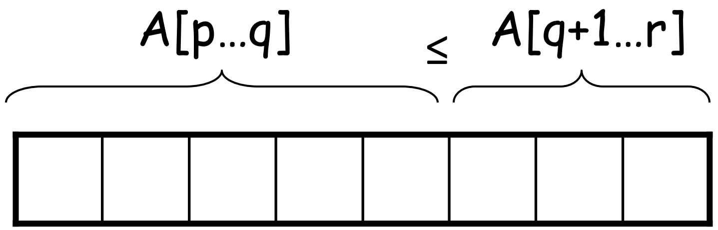
Sort an array $A[p \ldots r]$

Divide

- Partition the array A into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is smaller than or equal to each element in $A[q+1..r]$
- Need to find index q to partition the array



Quicksort



Conquer

- Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort

Combine

- Trivial: the arrays are sorted in place
- No additional work is required to combine them
- The entire array is now sorted

QUICKSORT

Initially: p=1, r=n

Alg.: QUICKSORT(A, p, r)

if p < r

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

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence:

T(n) = T(q) + T(n – q) + f(n)

($f(n)$ depends on PARTITION())

Partitioning the Array

Choosing PARTITION()

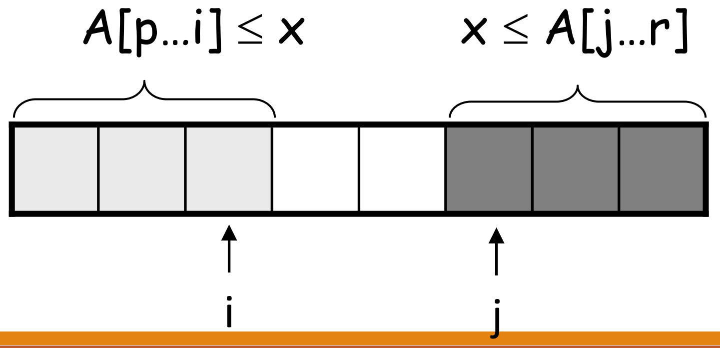
- There are different ways to do this
- Each has its own advantages/disadvantages

Hoare partition (see prob. 7-1, page 159)

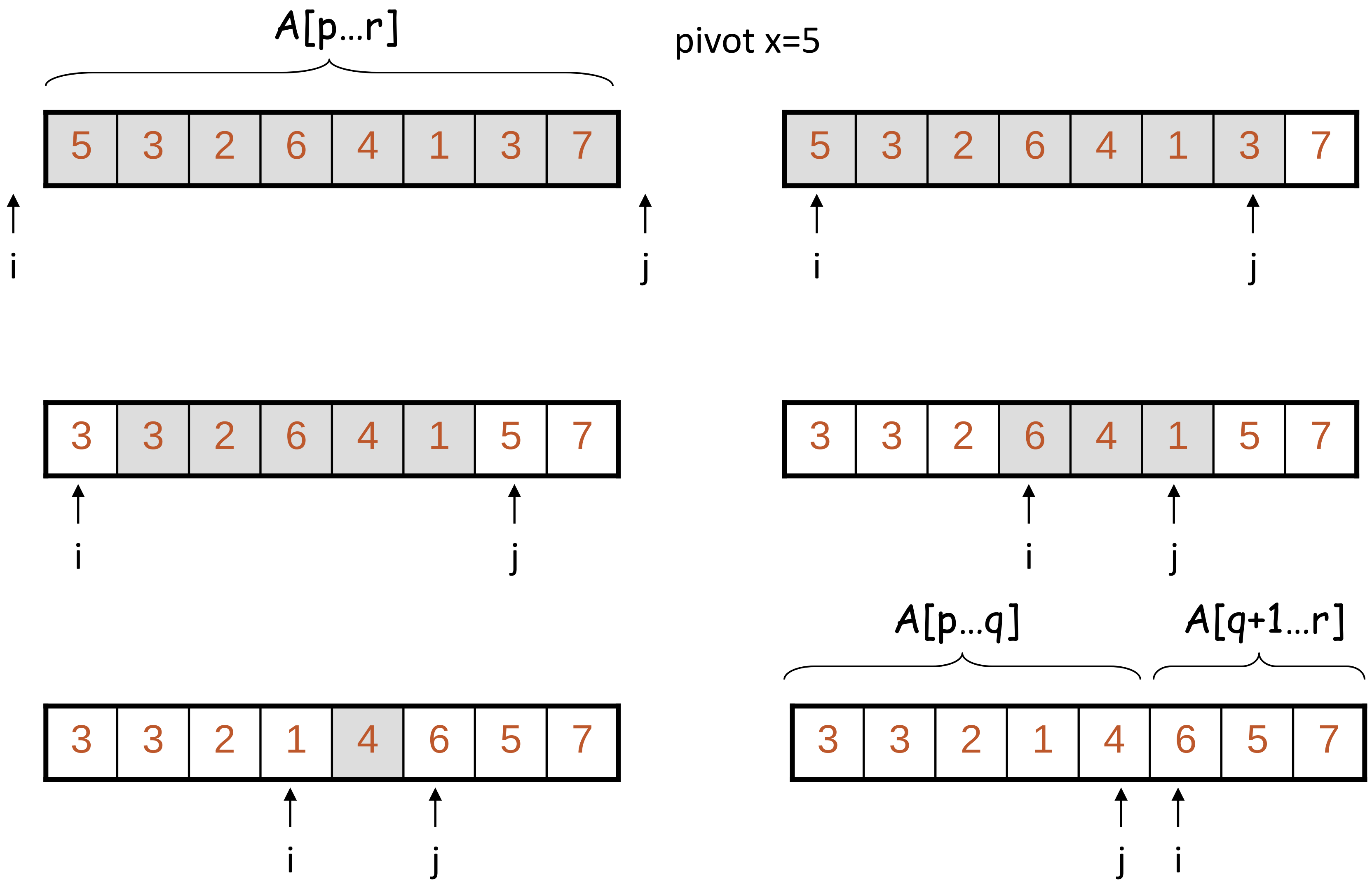
- Select a pivot element **x** around which to partition
- Grows two regions

$$A[p \ldots i] \leq \mathbf{x}$$

$$\mathbf{x} \leq A[j \ldots r]$$

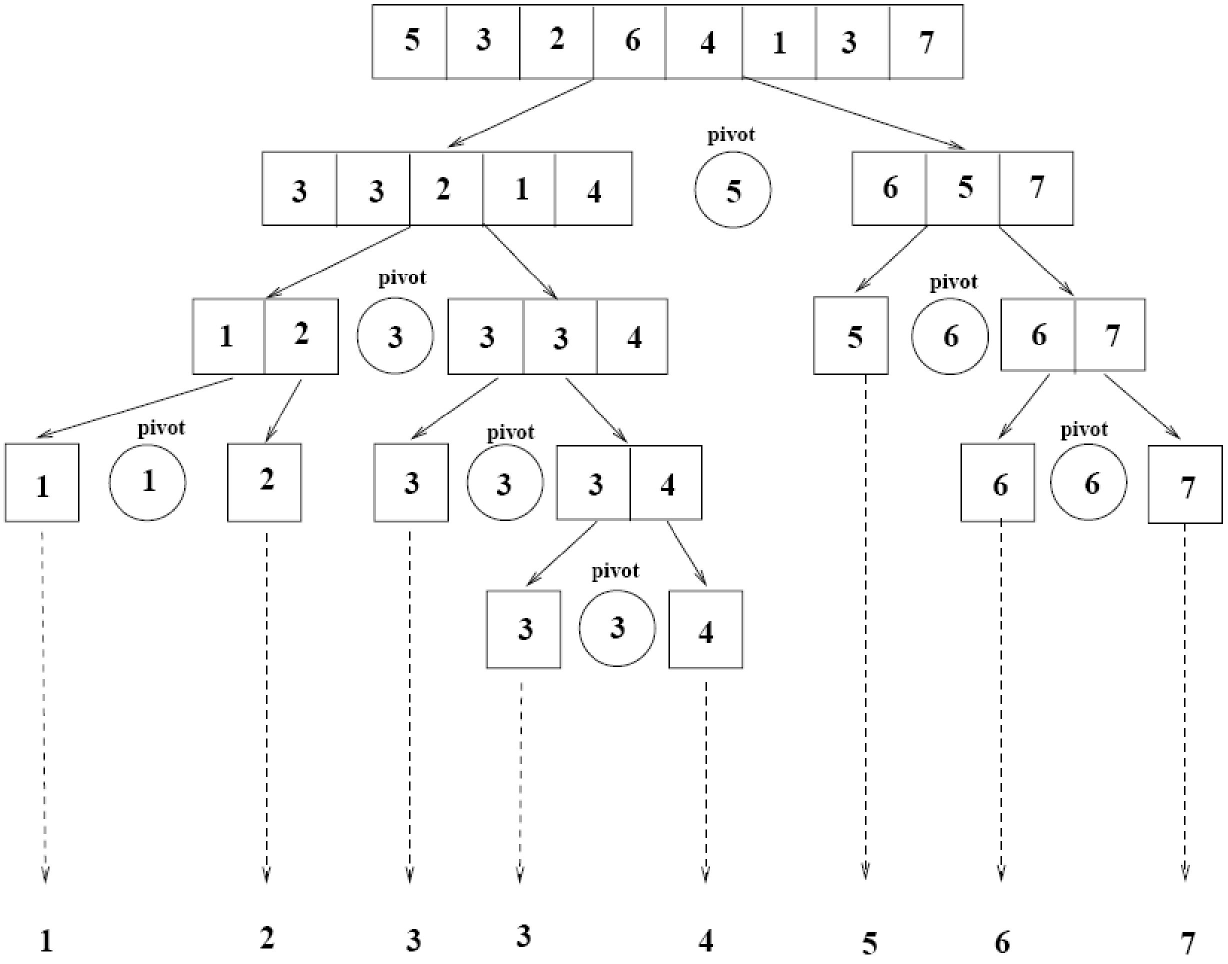


Example



A[p…r]
pivot x=5
5 3 2 6 4 1 3 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 1 4 6 5 7
i
j
A[p…q]
A[q+1…r]
3 3 2 1 4 6 5 7
j
i

Example

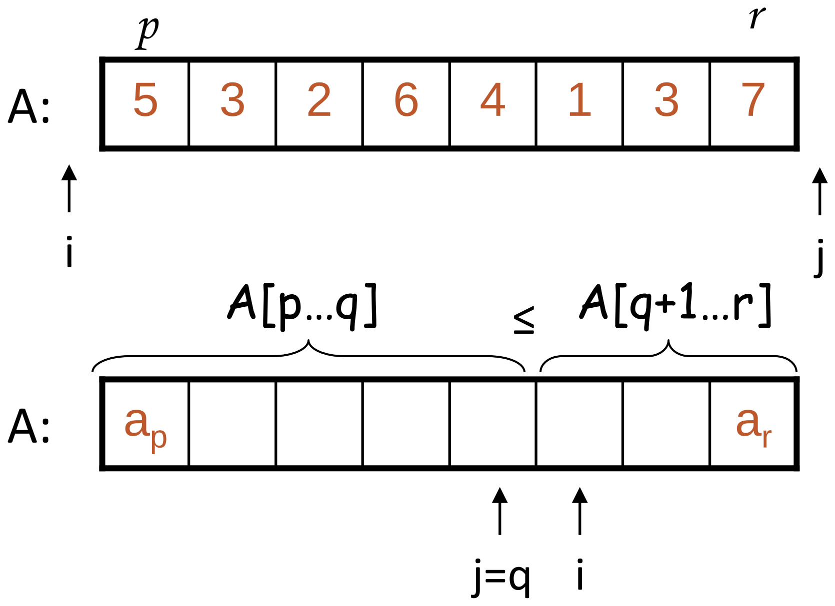


5 3 2 6 4 1 3 7
pivot
3 3 2 1 4
5
6 5 7
pivot
1 2
3
3 3 4
pivot
5
6
6 7
pivot
1
1
2
pivot
3
3
3 4
pivot
6
6
7
pivot
3
3
4
1 2 3 3 4 5 6 7

Partitioning the Array

Alg. PARTITION (A, p, r)

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



Each element is visited once!

Running time: $\Theta(n)$

$n = r - p + 1$

Recurrence

Initially: p=1, r=n

Alg.: QUICKSORT(A, p, r)

if p < r

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

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence:

$T(n) = T(q) + T(n – q) + n$

Worst Case Partitioning

Worst-case partitioning

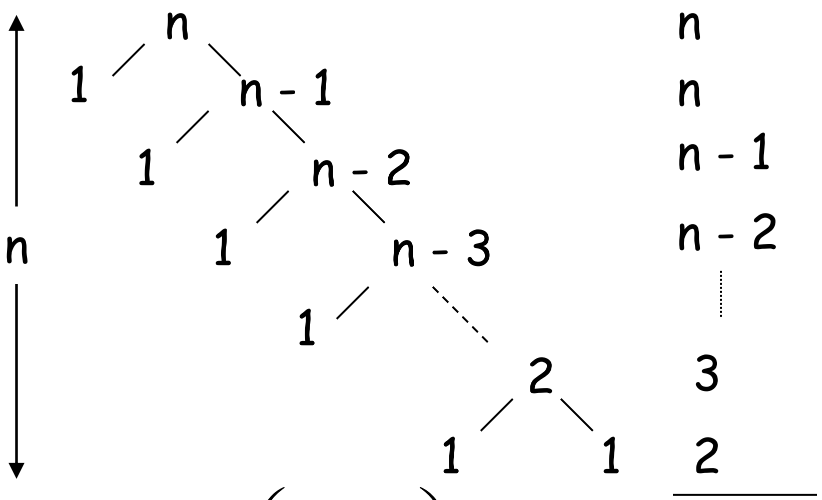
- One region has one element and the other has n – 1 elements
- Maximally unbalanced

Recurrence: q=1

T(n) = T(1) + T(n – 1) + n,

T(1) = Θ(1)

T(n) = T(n – 1) + n



$$n + \left(\sum_{k=1}^{n} k\right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

=

When does the worst case happen?

Best Case Partitioning

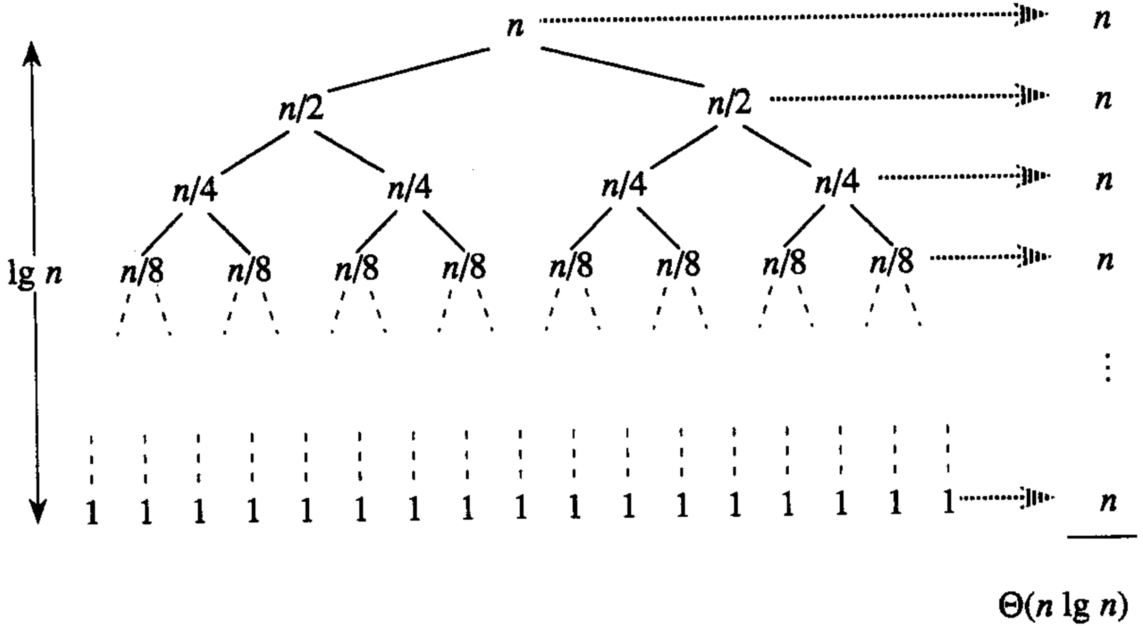
Best-case partitioning

- Partitioning produces two regions of size $n/2$

Recurrence: q=n/2

$T(n) = 2T(n/2) + \Theta(n)$

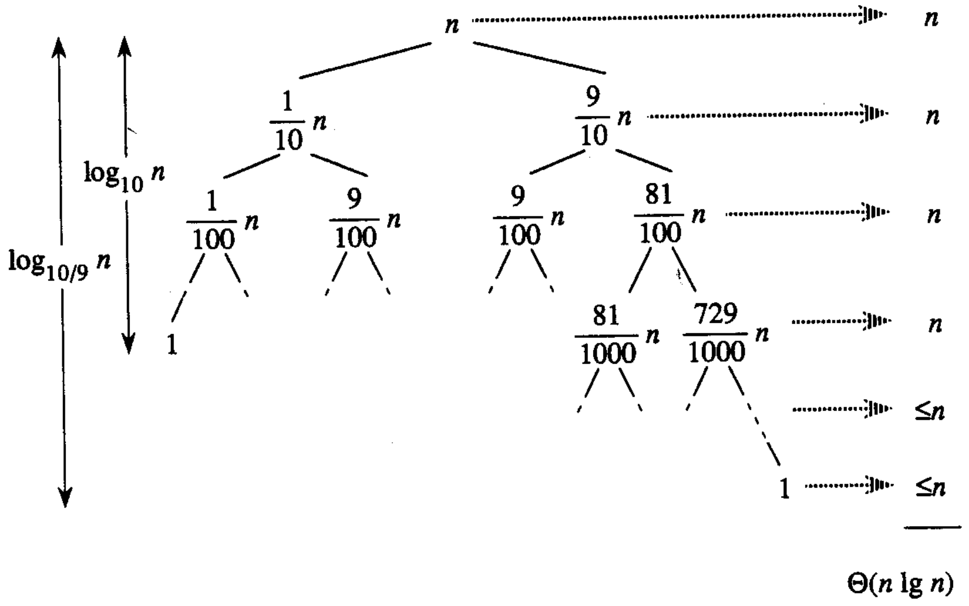
$T(n) = \Theta(n\lg n)$ (Master theorem)



Case Between Worst and Best

9-to-1 proportional split

$Q(n) = Q(9n/10) + Q(n/10) + n$



- Using the recursion tree:

longest path: $Q(n) \le n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 nlgn$

shortest path: $Q(n) \ge n \sum_{i=0}^{\log_{10} n} 1 = n \log_{10} n = c_1 nlgn$

Thus, $Q(n) = \Theta(nlgn)$

How does partition affect performance?

- **Any splitting of constant proportionality** yields $\Theta(nlgn)$ time !!!

- Consider the $(1 : n-1)$ splitting:

ratio=$1/(n-1)$ not a constant !!!

- Consider the $(n/2 : n/2)$ splitting:

ratio=$(n/2)/(n/2) = 1$ it is a constant !!

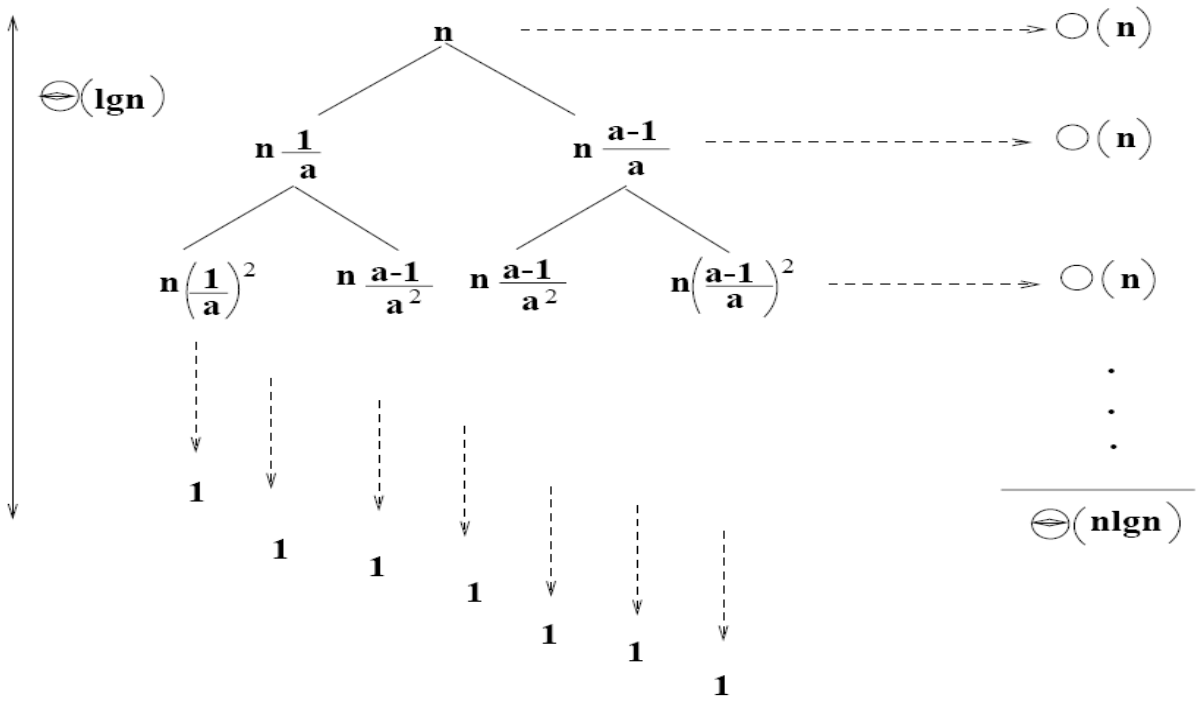
- Consider the $(9n/10 : n/10)$ splitting:

ratio=$(9n/10)/(n/10) = 9$ it is a constant !!

How does partition affect performance?

- Any $((a-1)n/a : n/a)$ splitting:

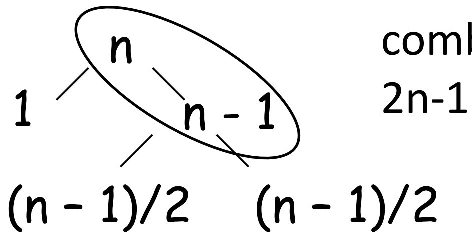
  ratio=$((a-1)n/a)/(n/a) = a-1$ it is a constant !!



Performance of Quicksort

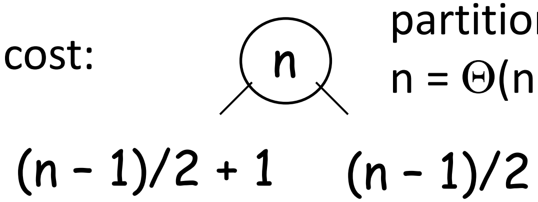
Average case

- All permutations of the input numbers are equally likely
- On a random input array, we will have a **mix** of well balanced and unbalanced splits
- Good and bad splits are randomly distributed across throughout the tree



combined partitioning cost:
$2n-1 = \Theta(n)$

Alternate of a good and a bad split



partitioning cost:
$n = \Theta(n)$

Nearly well balanced split

- Running time of Quicksort when levels alternate between good and bad splits is $O(n\lg n)$

Quick Sort - Discussion

Properties:

It is an in-space algorithm

It uses divide and conquer strategy

Not a stable sorting algorithm

Time complexity: Best and average case: $O(n\log n)$

Worst case: $O(n^2)$ which can be easily avoided by using randomized partition algorithm

Master's Theorem

$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$

$a \geq 1$, $b > 1$, $k \geq 0$, p is a real number then:

1) If $a > b^k$ then

$T(n) = \Theta(n^{\log_b a})$

2) If $a = b^k$ then

a) If $p > -1$, then

$T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

b) If $p = -1$, then

$T(n) = \Theta(n^{\log_b a} \log \log n)$

c) If $p < -1$, then

$T(n) = \Theta(n^{\log_b a})$

3) If $a < b^k$

a) If $p \geq 0$ then $T(n) = \Theta(n^k \log^p n)$

b) If $p < 0$, then $T(n) = O(n^k)$.

Q) $T(n) = 3T\left(\frac{n}{2}\right) + n^2$

$a = 3$, $b = 2$, $k = 2$, $p = 0$

$a = 3$, $b^k = 2^2 = 4$

$\Rightarrow a < b^k$ and $p = 0$

The 3.a. $\Rightarrow$ $T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^2 \log^0 n) = \Theta(n^2)$