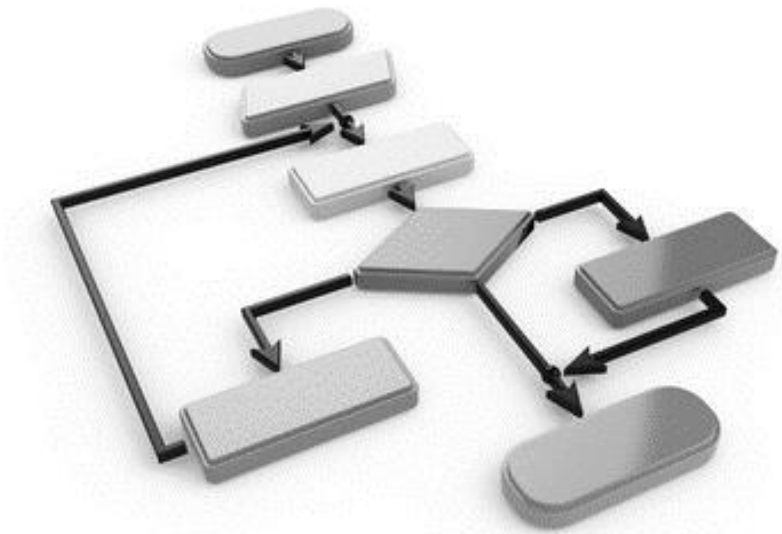


Lecture 03

Divide and Conquer

CSE373: Design and Analysis of Algorithms

Quicksort

Follows the **divide-and-conquer** paradigm.

Divide: Partition (separate) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$.

- Each element in $A[p..q-1] \leq A[q]$.
- $A[q] \leq$ each element in $A[q+1..r]$.
- Index q is computed as part of the partitioning procedure.

Conquer: Sort the two subarrays by recursive calls to quicksort.

Combine: The subarrays are sorted in place – no work is needed to combine them.

How do the divide and combine steps of quicksort compare with those of merge sort?

Partitioning

Select the last element A[r] in the subarray $A[p..r]$ as the *pivot* – the element around which to partition.

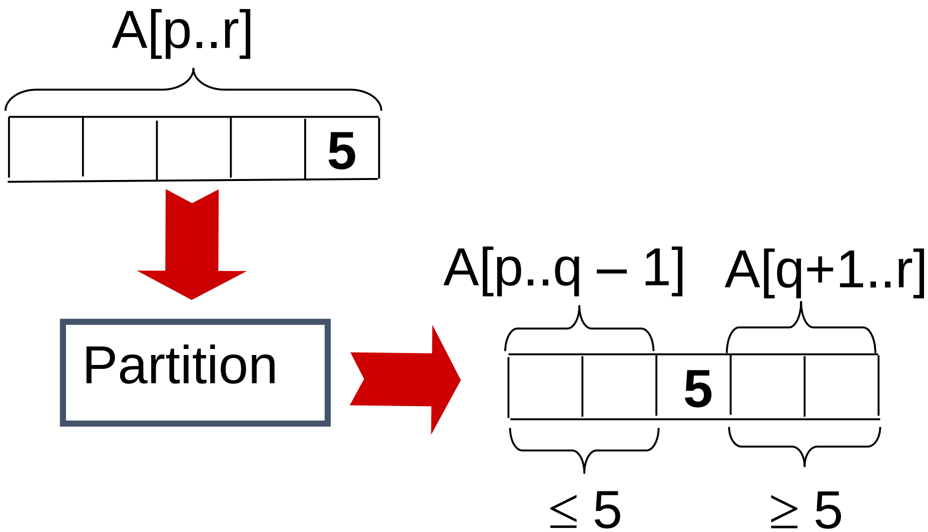
As the procedure executes, the array is partitioned into four (possibly empty) regions.

1. $A[p..i]$ — All entries in this region are $\leq$ ***pivot***.
2. $A[i+1..j-1]$ — All entries in this region are **>** ***pivot***.
3. $A[r] = pivot$.
4. $A[j..r-1]$ — Not known how they compare to *pivot*.

Partitioning

```
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 then
        q = PARTITION(A, p, r);
        QUICKSORT (A, p, q – 1);
        QUICKSORT (A, q + 1, r)
```



Example

```
                      p                             r
initially:            2  5  8  3  9  4  1  7  10  6        note: pivot (x) = 6
                    i j

next iteration:       2  5  8  3  9  4  1  7  10  6
                      i  j

next iteration:       2  5  8  3  9  4  1  7  10  6
                         i  j

next iteration:       2  5  8  3  9  4  1  7  10  6
                         i     j

next iteration:       2  5  3  8  9  4  1  7  10  6
                            i     j
```

```
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
```

Example (Continued)

next iteration:

2	5	3	8	9	4	1	7	10	**6**
		i		j					

next iteration:

2	5	3	8	9	4	1	7	10	**6**
		i			j				

next iteration:

2	5	3	4	9	8	1	7	10	**6**
			i			j			

next iteration:

2	5	3	4	1	8	9	7	10	**6**
				i			j		

next iteration:

2	5	3	4	1	8	9	7	10	**6**
				i				j	

next iteration:

2	5	3	4	1	8	9	7	10	**6**
				i					j

after final swap:

2	5	3	4	1	**6**	9	7	10	8
				i					j

```
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
```

Complexity of Partition

PartitionTime(n) is given by the number of iterations in the *for* loop.

$\Theta(n)$: $n = r - p + 1.$

```
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
```

Algorithm Performance

Running time of quicksort depends on whether the partitioning is balanced or not.

Worst-Case Partitioning (Unbalanced Partitions):

Occurs when every call to partition results in the most unbalanced partition.

Partition is most unbalanced when

- Subproblem 1 is of size $n - 1$, and subproblem 2 is of size 0 or vice versa.
- $pivot \geq$ every element in $A[p..r - 1]$ or $pivot <$ every element in $A[p..r - 1]$.

Every call to partition is most unbalanced when

- Array $A[1..n]$ is sorted or reverse sorted!

Worst-case of quicksort

Input sorted or reverse sorted.

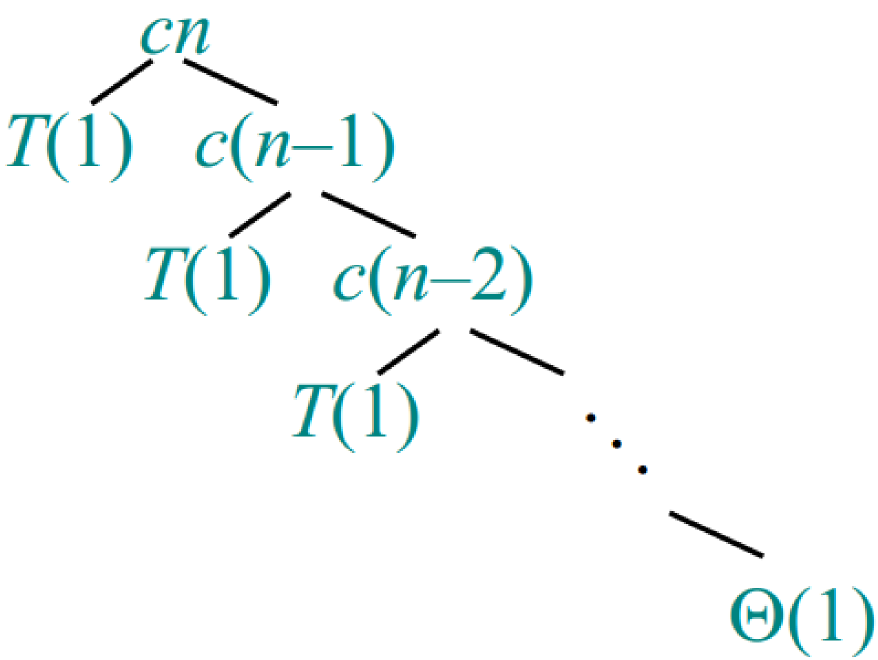
Partition around min or max element.

One side of partition always has one element.

$$
\begin{aligned}
T(n) &= T(1) + T(n-1) + \Theta(n) \\
&= \Theta(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2) \qquad \textbf{\textit{(arithmetic series)}}
\end{aligned}
$$

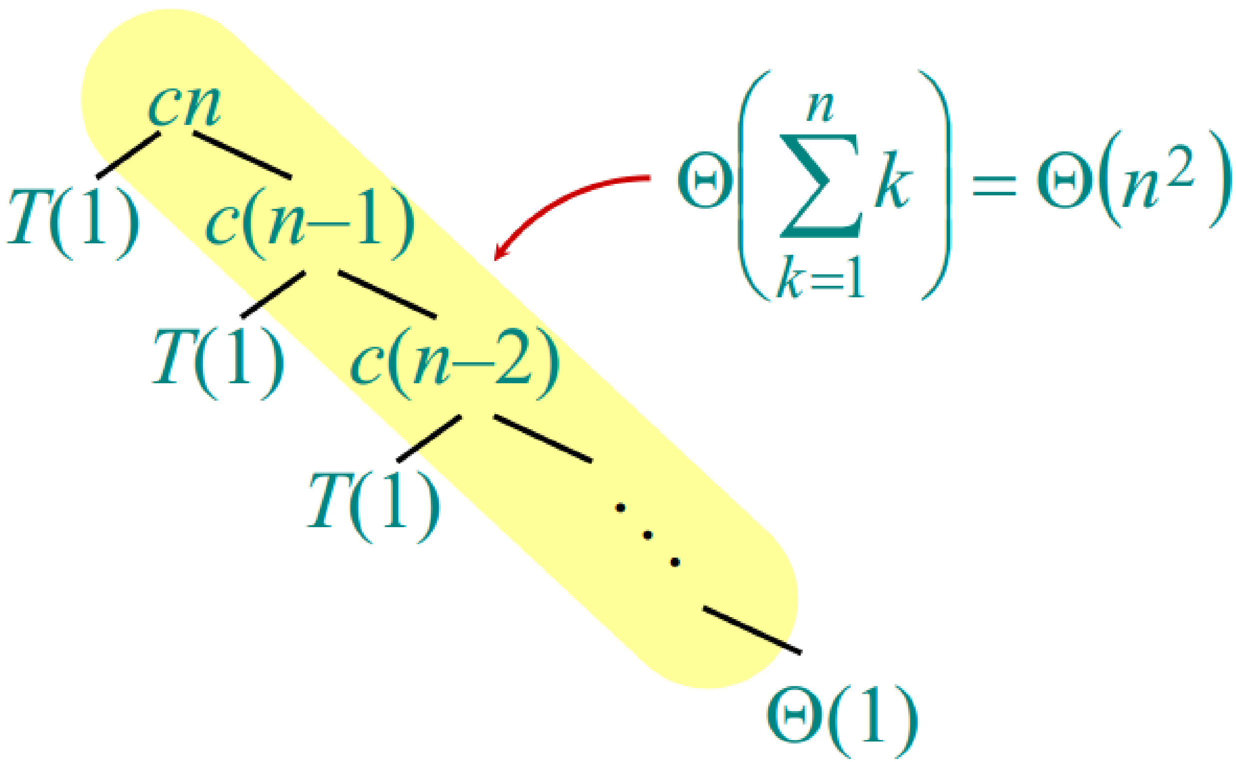
Worst-case recursion tree

$$T(n) = T(1) + T(n-1) + cn$$



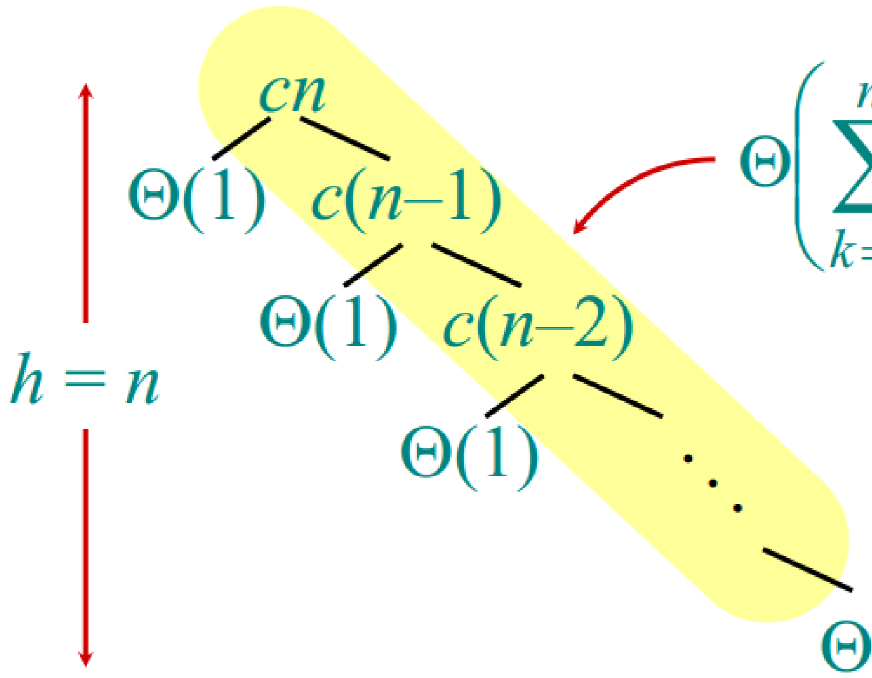
Worst-case recursion tree

$$T(n) = T(1) + T(n-1) + cn$$



Worst-case recursion tree

$$T(n) = T(1) + T(n-1) + cn$$



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

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$

Best-case Partitioning

Size of each subproblem $\leq n/2$.

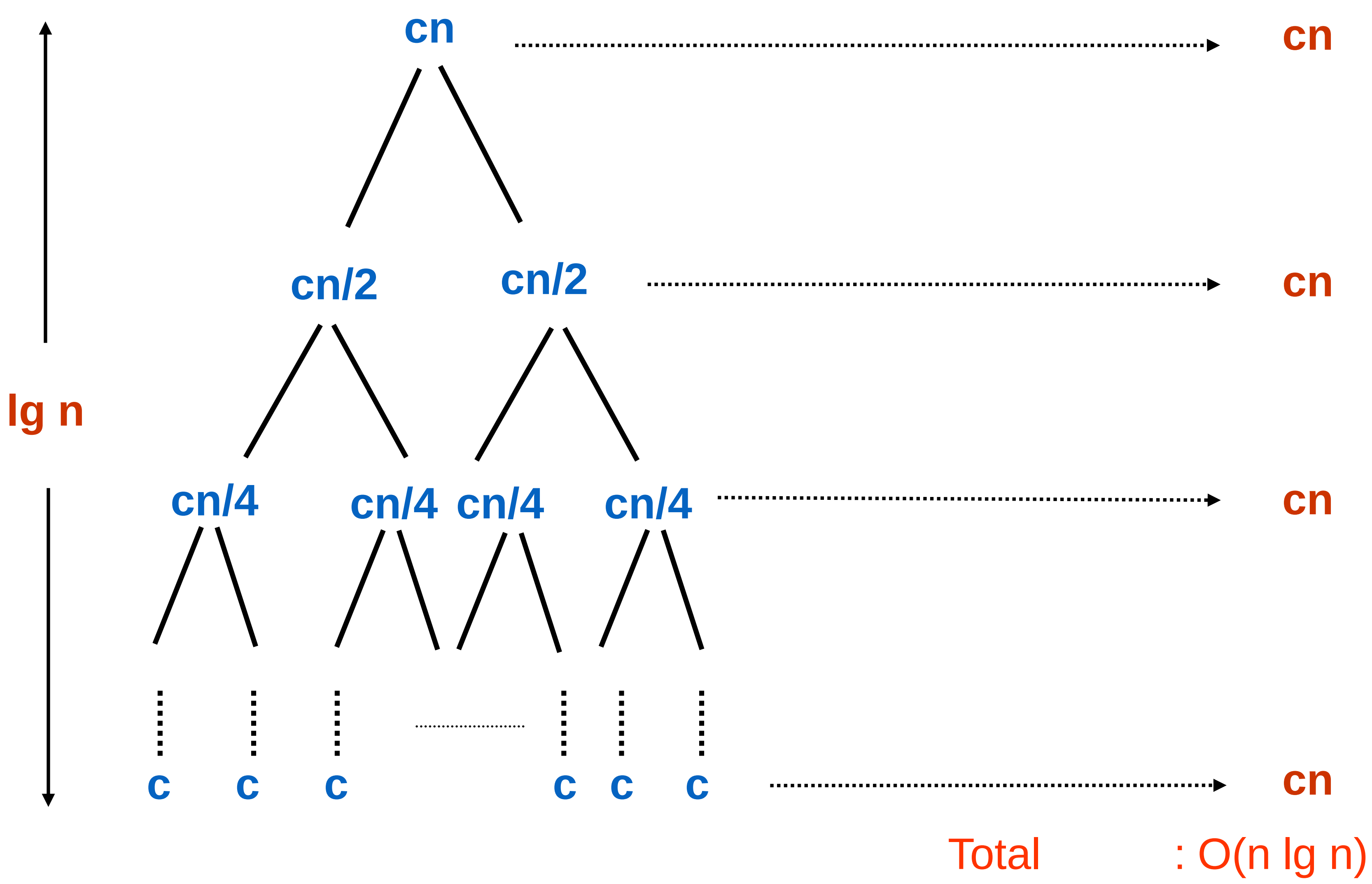
- One of the subproblems is of size $\lfloor n/2 \rfloor$
- The other is of size $\lceil n/2 \rceil - 1$.

Recurrence for running time

$$T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$$
$$= 2T(n/2) + \Theta(n)$$

$$\boldsymbol{T(n) = \Theta(n \lg n)}$$

Recursion Tree for Best-case Partition



Best-case analysis

If we're lucky, H-PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

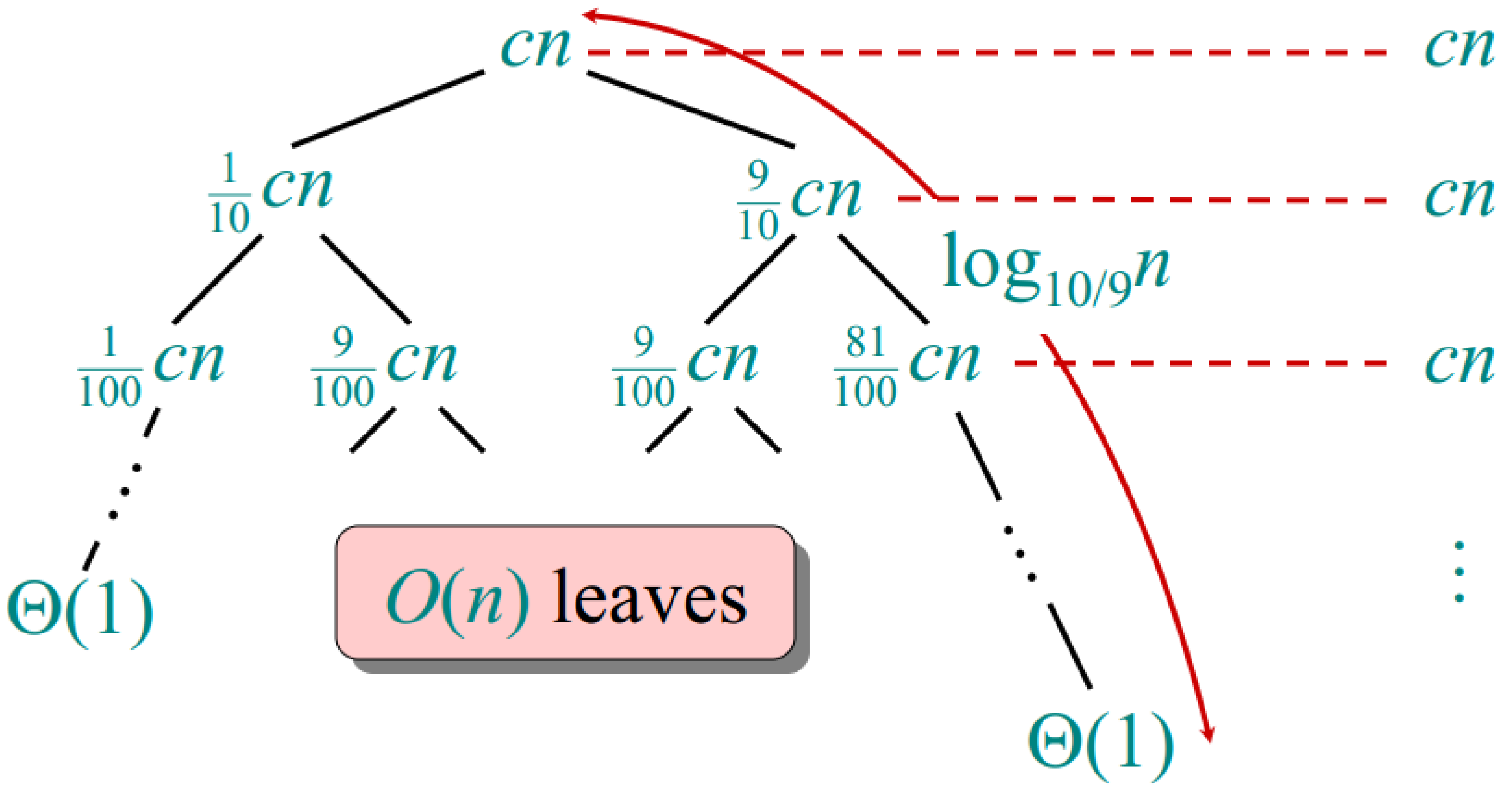
(same as merge sort)

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\tfrac{1}{10}n\right) + T\left(\tfrac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?

Analysis of “almost-best” case



Analysis of “almost-best” case