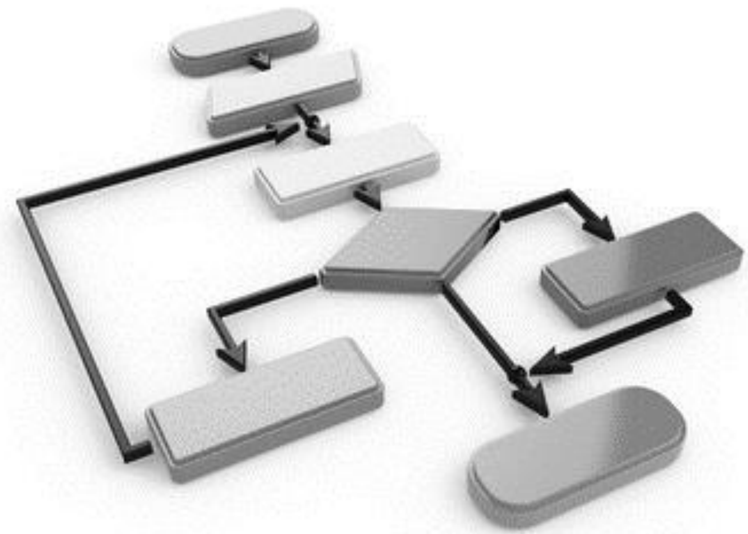


Lecture 04

Divide and Conquer (quicksort)

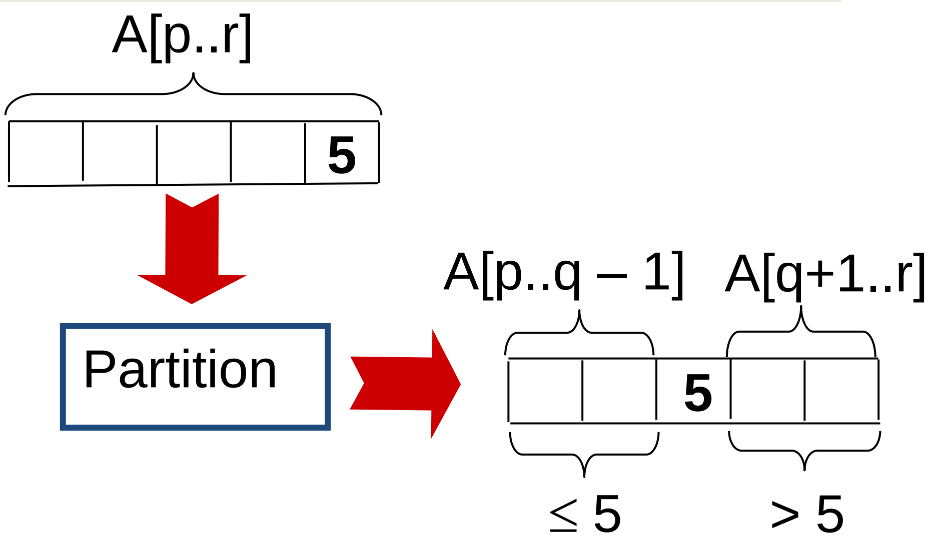
CSE373: Design and Analysis of Algorithms

Quicksort (Chapter 7)

- 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 $A[p..q-1] \leq A[q]$.
  - $A[q] <$ 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?

Pseudocode

```
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**
	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					

note: pivot (x) = 6

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

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
```

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*.

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)
    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
```

Algorithm Performance

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

– Best case: occurs when the recursion tree is always balanced

– Worst case: occurs when the recursion tree is always unbalanced

– Average case: computed by using expected value of running time assuming that the pivot elements are randomly distributed

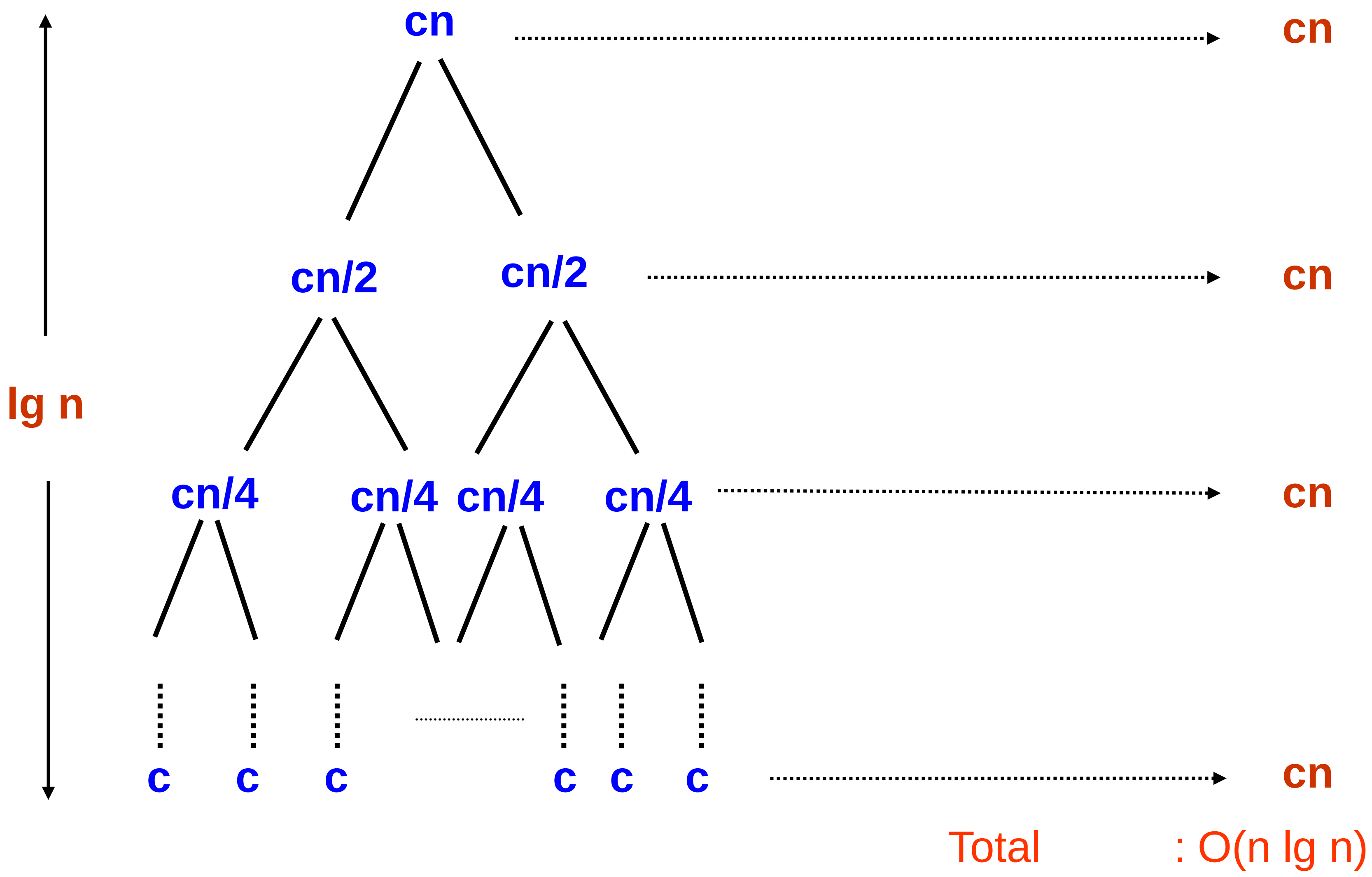
Best-case Partitioning

- Size of each subproblem ≈ $n/2$.
- Recurrence for running time
  - $T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$

    $= 2T(n/2) + \Theta(n)$, just like MergeSort
- **$T(n) = \Theta(n \lg n)$, like MergeSort**

```
QUICKSORT(A, p, r) //initial call: QuickSort(A,1,n)   …. Takes T(n) time (let)
    if p < r then                                      …… Θ(1)
        q = PARTITION(A, p, r);                        ….. Θ(n)
        QUICKSORT (A, p, q – 1);                       ….. T(n/2)
        QUICKSORT (A, q + 1, r);                       ….. T(n/2)
```

Recursion Tree for Best-case Partition



Worst-case of quicksort

- 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!**
    - One side of partition always has one element.

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

Expected Running time of QuickSort

- We can show that the expected running time of QuickSort is O(n lg n), same as that of MergeSort.
- So, even though, the worst case running time of QuickSort is worse than that of MergeSort, its Average running time is comparable to that of MaergeSort.
- For the sake of simplicity, we omit the mathematical details of the calculation of expected running time of QuickSort.
- In practice, QuickSort is much faster than MergeSort

Randomized quicksort

How can we **ENSURE** that each element of A are equally likely to be the pivot?

IDEA: Partition around a ***random*** element.

- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

Randomized quicksort

Standard Problematic Algorithm :

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

if $p < r$

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

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

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

Initial call: $\text{QUICKSORT}(A, 1, n)$

Randomized quicksort

RANDOMIZED-PARTITION (A, p, r)

$i \leftarrow$ RANDOM(p, r)
exchange A[r]$\leftrightarrow$A[i]
return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT (A,p,r)

if $p<r$
then $q \leftarrow$ RANDOMIZED-PARTITION (A, p, r)
    RANDOMIZED-QUICKSORT (A, p, q-1)
    RANDOMIZED-QUICKSORT (A, q+1, r)

In practice, Randomized-QuickSort is much faster than QuickSort