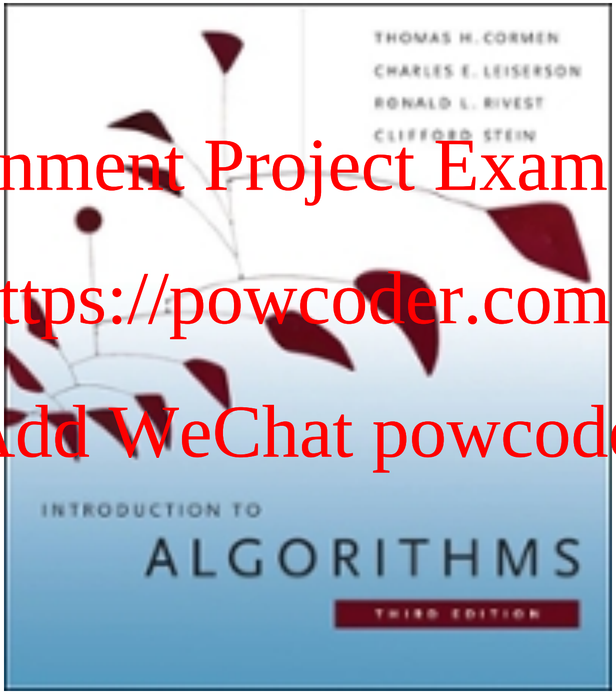
CS146 Data Structures and Algorithms



Chapter 7: Quicksort

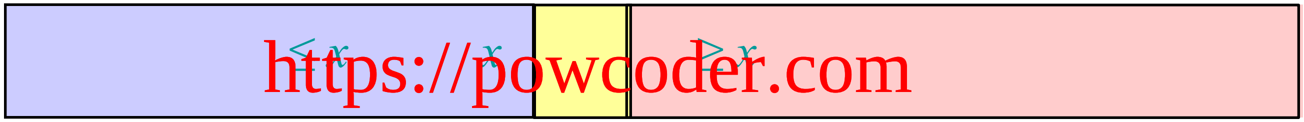
Quicksort

- Sorts in place
- Sorts O(n lg n) in the average case
- Sorts $O(n^2)$ in the worst case
  - But in practice, it's quick
  - And the worst case doesn't happen often (but more on this later…)
  - Empirical and analytical studies show that quicksort can be ***expected*** to be twice as fast as its competitors.

7.1 Description of Quicksort

Quicksort an *n*-element array:

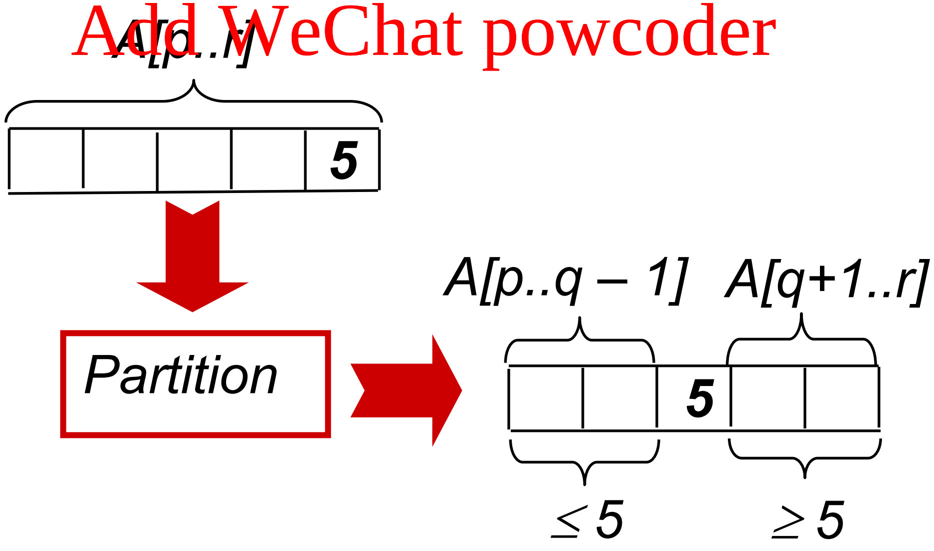
- *Divide:* Partition the array into two subarrays around a *pivot* x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.



- *Conquer:* Recursively sort the two subarrays.
- *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?

Design

- Key: Linear-time partitioning subroutine.
- ***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.



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

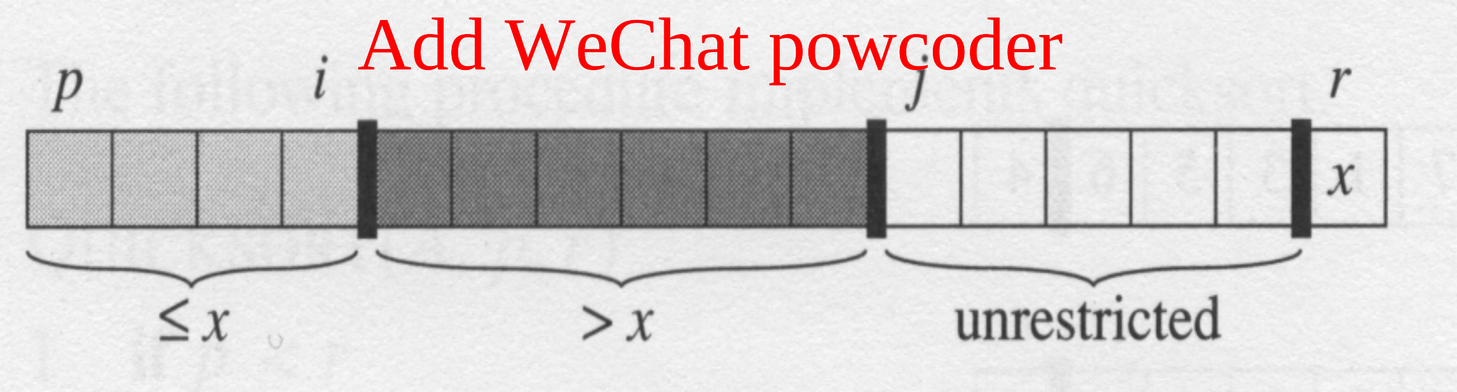
Sorting Animation

Initial call: QUICKSORT($(A, 1, n)$)

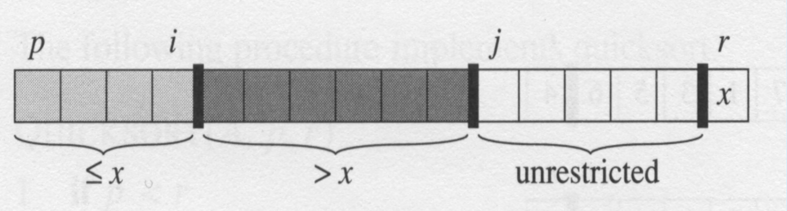
Loop Invariant

Loop Invariant

1. All entries in A[p..i] are ≤ pivot.
2. All entries in A[i+1..j-1] are > pivot.
3. A[r] = pivot.

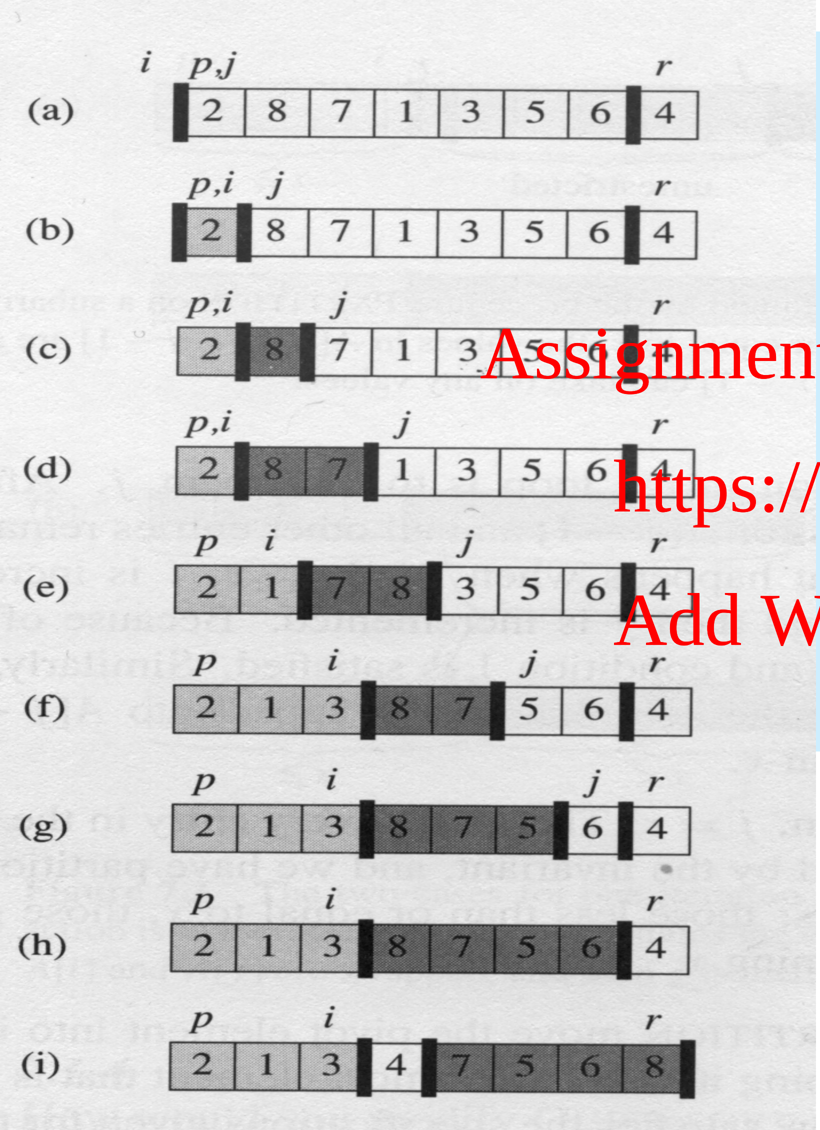


Partition(A, p, r)



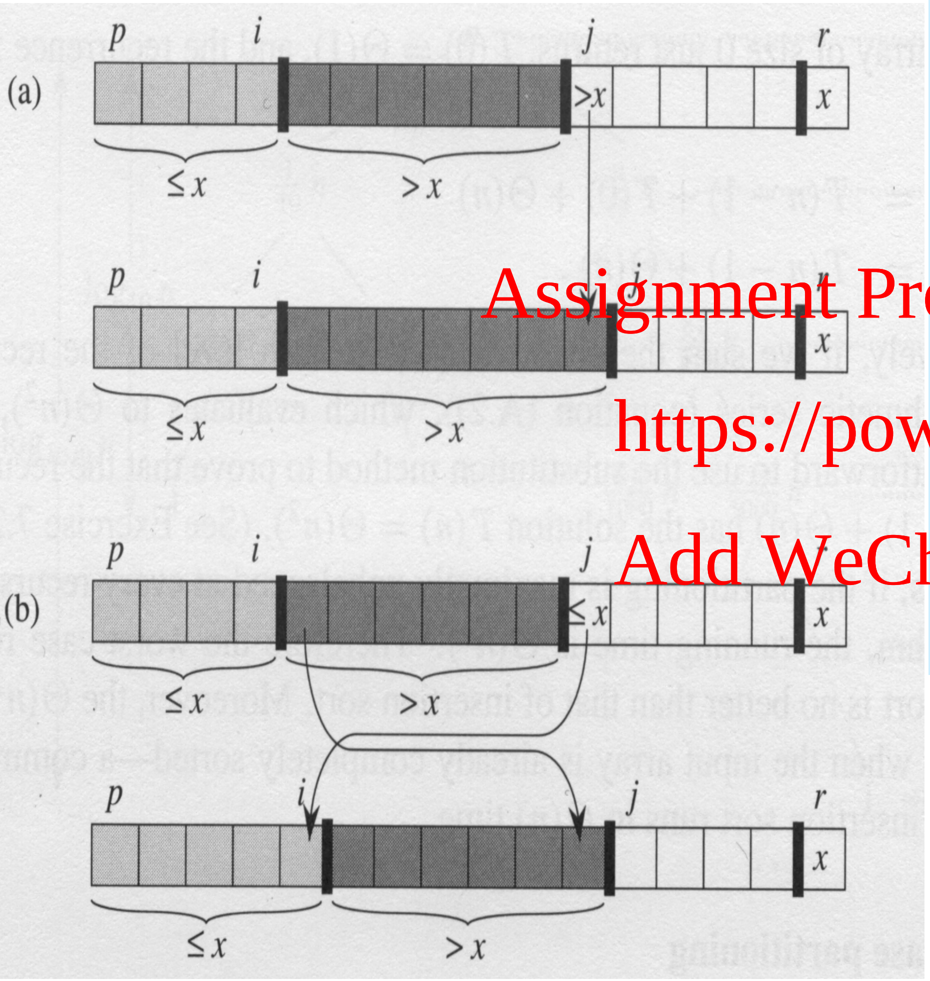
```
x = A[r]
i  = p - 1
for j = p to r -1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

The operation of *Partition* on a sample array



```
x = A[r]
i = p - 1
for j = p to r -1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

Two cases for one iteration of procedure *Partition*



```
x = A[r]
i = p – 1
for j = p to r -1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

Complexity:
Partition on A[p…r] is Θ(n)
where n = r –p +1

Why???
(Exercise 7.1-3)

Exercise 7.1-3

- Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

Answer:

The for loop makes exactly r − p iterations, each of which takes at most constant time. The part outside the for loop takes at most constant time. Since r − p is the size of the subarray, PARTITION takes at most time proportional to the size of the subarray it is called on.

Another partitioning example

initially:

```
  p                            r
  2  5  8  3  9  4  1  7  10  6
i j
```

note: *pivot (x) = 6*

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
```

```
x = A[r]
i = p − 1
for j = p to r −1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

Another 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

```
x = A[r]
i = p – 1
for j = p to r -1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] ↔
   A[j]
exchange A[i+1] ↔ 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 ≤ ***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*.

- The above hold before each iteration of the *for* loop, and constitute a *loop invariant*. (4 is not part of the LI.)

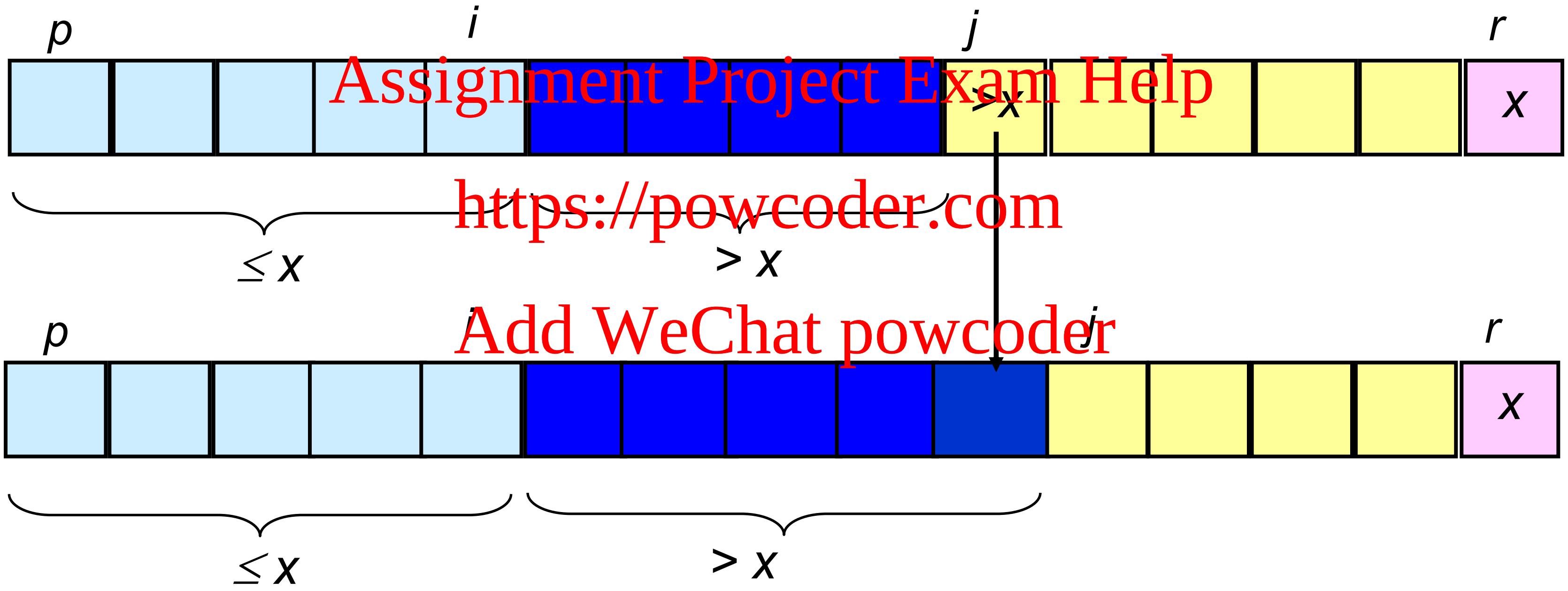
Correctness of Partition

- Use loop invariant.
- **Initialization:**
  - Before first iteration
    - $A[p..i]$ and $A[i+1..j-1]$ are empty – Conds. 1 and 2 are satisfied (trivially).
    - r is the index of the *pivot* – Cond. 3 is satisfied.
- **Maintenance:**
  - **Case 1:** $A[j] > x$
    - Increment j only.
    - LI is maintained.

```
x = A[r]
i = p - 1
for j = p to r -1
 if A[j] ≤ x
    i = i + 1
    exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

Correctness of Partition

Case 1:

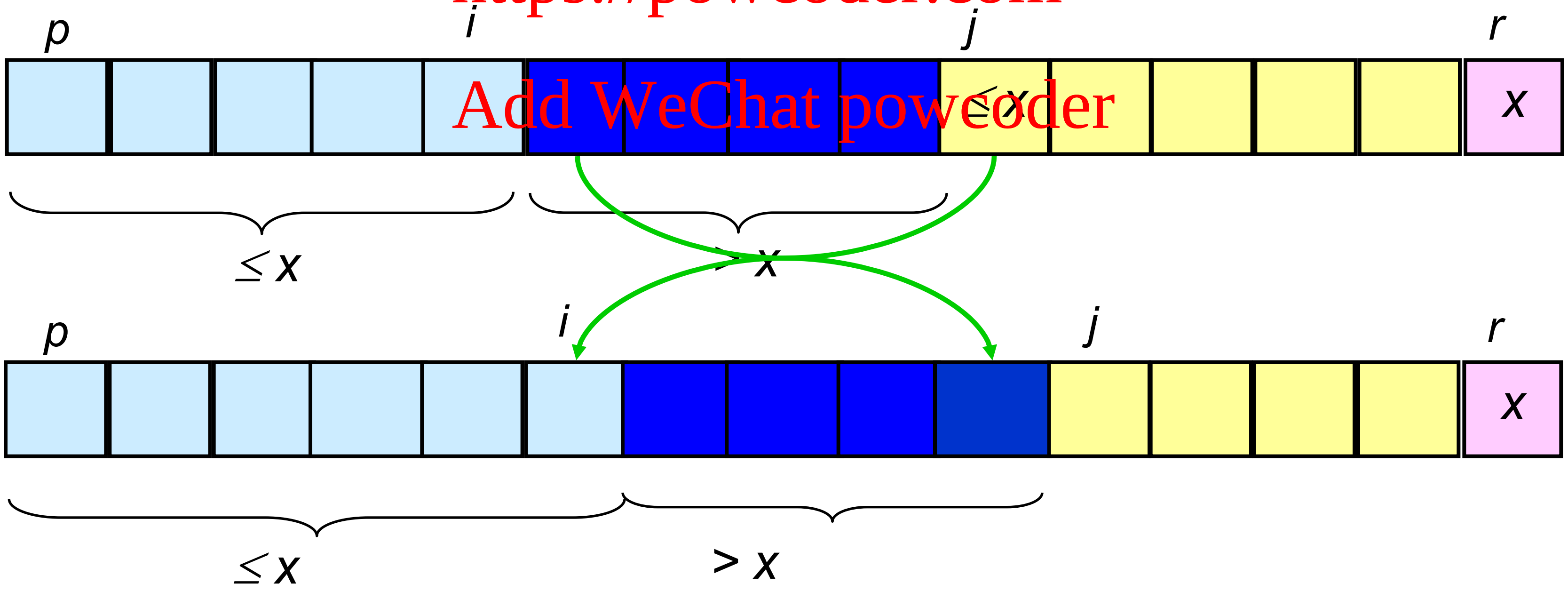


Correctness of Partition

- **Case 2:** $A[j] \leq x$
  - Increment i
  - Swap $A[i]$ and $A[j]$
    - Condition 1 is maintained.
  - Increment j
    - Condition 2 is maintained.

» A[r] is unaltered.

- Condition 3 is maintained.



Correctness of Partition

- **Termination:**
  - When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases:
    - $A[p..i] \le$ ***pivot***
    - $A[i+1..j-1] >$ ***pivot***
    - $A[r] =$ ***pivot***
- The last two lines swap $A[i+1]$ and $A[r]$.
  - *Pivot* moves from the end of the array to between the two subarrays.
  - Thus, procedure *partition* correctly performs the divide step.

Complexity of Partition

- PartitionTime(n) is given by the number of iterations in the *for* loop.
- $\Theta(n)$: $n = r - p + 1$.

```
x = A[r]
i = p - 1
for j = p to r -1
 if A[j] ≤ x
    i = i + 1
    exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

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

Sorting Animation

Initial call: QUICKSORT(A, 1, n)

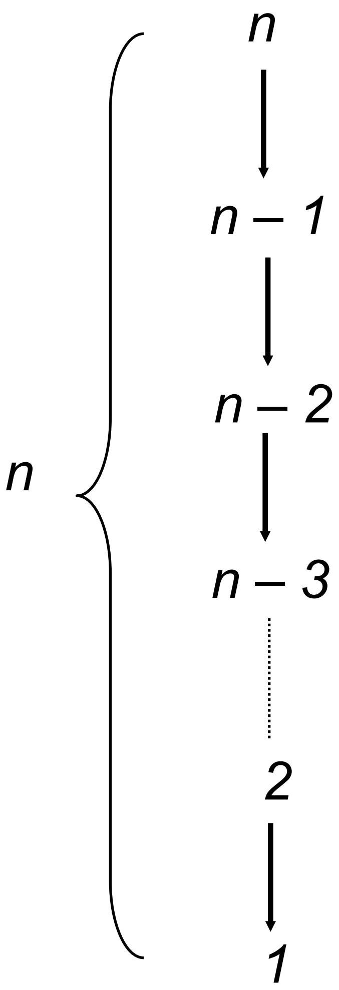
Algorithm Performance

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
    - Subarray 1 is of size $n - 1$, and subarray 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 Partition Analysis

Recursion tree for worst-case partition



Running time for worst-case partitions at each recursive level:

$$T(n) = T(n-1) + T(0) + \text{PartitionTime}(n)$$
$$= T(n-1) + \Theta(n)$$
$$= \textstyle\sum_{k=1 \text{ to } n} \Theta(k)$$
$$= \Theta(\textstyle\sum_{k=1 \text{ to } n} k) = \Theta(n(n+1)/2)$$
$$= \Theta(n^2)$$

Best-case Partitioning

- Size of each subarray $\le n/2$.
  - One of the subarray is of size $\lfloor n/2 \rfloor$
  - The other is of size $\lceil n/2 \rceil - 1$.

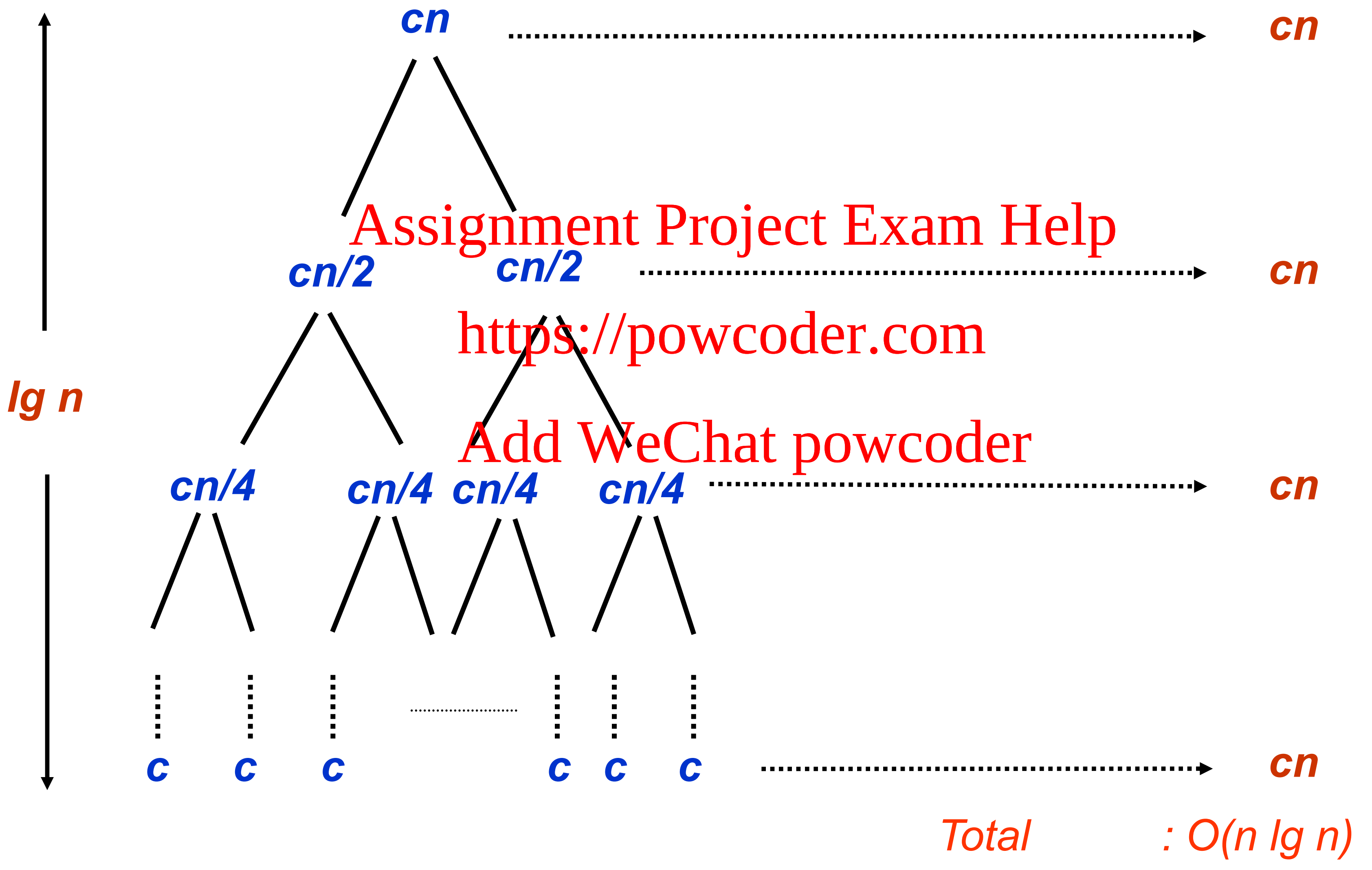
- Recurrence for running time

  - $\boldsymbol{T(n) \le 2T(n/2) + \textbf{PartitionTime}(n)}$

    $\boldsymbol{= 2T(n/2) + \Theta(n)}$

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

Recursion Tree for Best-case Partition



Variations

- Quicksort is not very efficient on small lists.
- This is a problem because Quicksort will be called on lots of small lists.

- **Fix 1:** Use Insertion Sort on small arrays.
- **Fix 2:** Leave small arrays unsorted. Fix with one final Insertion Sort at end.
  - **Note:** Insertion Sort is very fast on almost-sorted lists.

Unbalanced Partition Analysis

What happens if we get poorly-balanced partitions,

e.g., something like: $T(n) \leq T(9n/10) + T(n/10) + \Theta(n)$?

Still get $\Theta(n \lg n)$!! *(As long as the split is of constant proportionality.)*

Intuition: Can divide n by c > 1 only $\Theta(\lg n)$ times before getting 1.

n

↓

n/c

↓

n/c^2

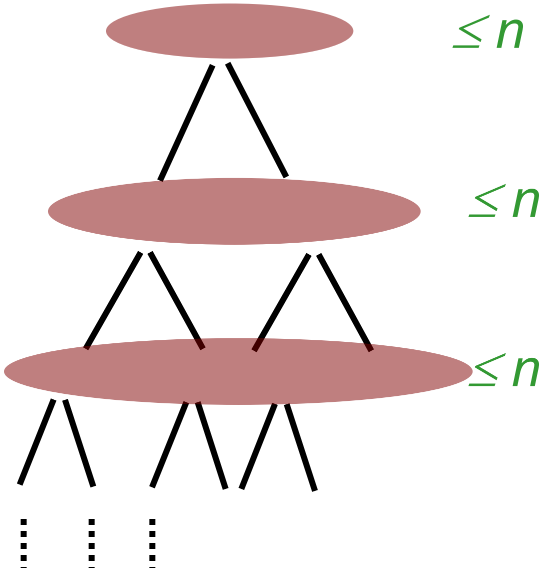
↓

⋮

↓

$1 = n/c^{\log_c n}$

Roughly $\log_c n$ levels;
Cost per level is O(n).



(**Remember:** Different base logs are related by a constant.)

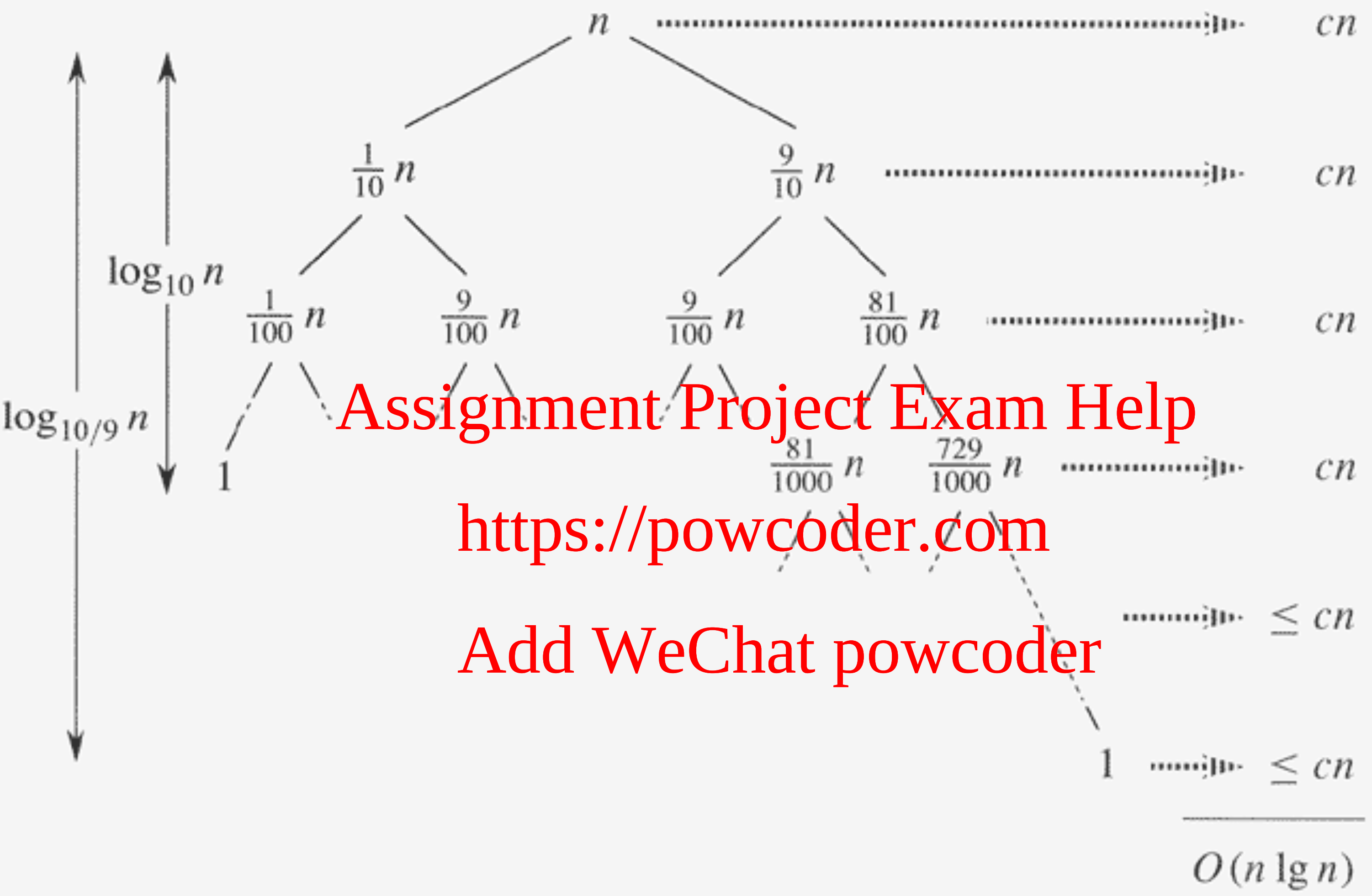


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

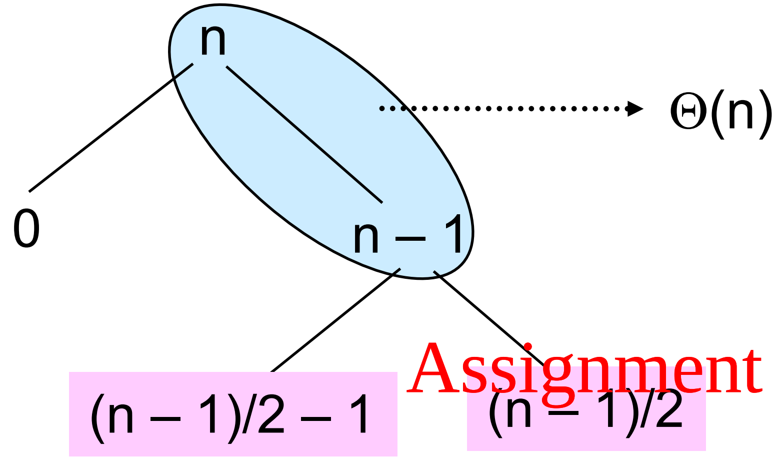
Intuition for the Average Case

- Partitioning is unlikely to happen in the same way at every level.
  - Split ratio is different for different levels. (Contrary to our assumption in the previous slide.)
- Partition produces a mix of "good" and "bad" splits, distributed randomly in the recursion tree.
- What is the running time likely to be in such a case?

Analyzing Quicksort: Average Case

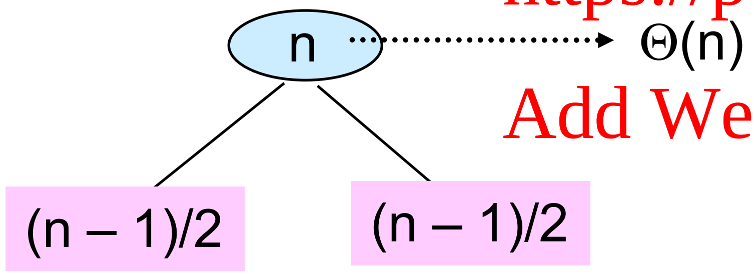
- Intuitively, a real-life run of quicksort will produce a mix of “bad” and “good” splits
  - Randomly distributed among the recursion tree
  - Pretend for intuition that they alternate between best-case (n/2 : n/2) and worst-case (n-1 : 1)
  - *What happens if we bad-split root node, then good-split the resulting size (n-1) node?*
    - We end up with three subarrays, size 1, (n-1)/2, (n-1)/2
    - Combined cost of splits = n + n -1 = 2n -1 = O(n)
    - No worse than if we had good-split the root node!

Intuition for the Average Case



Bad split followed by a good split:
Produces subarrays of sizes 0, (n – 1)/2 – 1, and (n – 1)/2.
Cost of partitioning :

$$\Theta(n) + \Theta(n-1) = \Theta(n).$$



Good split at the first level:
Produces two subarrays of size (n – 1)/2.
Cost of partitioning :

$$\Theta(n).$$

Situation at the end of case 1 is not worse than that at the end of case 2.
When splits alternate between good and bad, the cost of bad split can be absorbed into the cost of good split.
Thus, running time is O(n lg n), though with larger hidden constants.

Randomized Quicksort

- Want to make running time independent of input ordering.
- How can we do that?
  - Make the algorithm randomized.
  - Make every possible input equally likely.
    - Can randomly shuffle to permute the entire array.
    - For quicksort, it is sufficient if we can ensure that every element is equally likely to be the pivot.
    - So, we choose an element in A[p..r] and exchange it with A[r].
    - Because the pivot is randomly chosen, we expect the partitioning to be well balanced on average.

Randomized Version

Want to make running time independent of input ordering.

```
Randomized-Partition(A, p, r)
        i = Random(p, r);
        exchange A[r] =  A[i];
        Partition(A, p, r)
```

```
Randomized-Quicksort(A, p, r)
        if p < r
                q = Randomized-Partition(A, p, r);
                Randomized-Quicksort(A, p, q – 1);
                Randomized-Quicksort(A, q + 1, r)
```

7.4 Analysis of quicksort

7.4.1 Worst-case analysis

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

guess $T(n) \le cn^2$

$$T(n) \le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n)$$

$$= c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

$$\le cn^2 - 2c(n-1) + \Theta(n)$$

$$\le cn^2$$

pick the constant *c* large enough so that the $2c(n-1)$ **term dominates the** $\Theta(n)$ **term.**

$\Rightarrow T(n) = \Theta(n^2)$

7.4.2 Expected running time

- Running time and comparisons
- Lemma 7.1
  - Let X be the number of comparisons performed in line 4 of *partition* over the entire execution of *Quicksort* on an *n*-element array. Then the running rime of *Quicksort* is $O(n+X)$

we define

$$X_{ij} = I\ \{\mathrm{z_i\ is\ compared\ to\ z_j}\},$$

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr \quad \{\mathrm{z_i\ is\ compared\ to\ z_j}\}$$

$$\Pr\{z_i \text{ is compared to } z_j\} = \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}$$

$$= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\}$$

$$+ \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} \quad = \frac{2}{j-i+1}$$

$$\therefore E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}.$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

by pp. 1147 A.7

$$= \sum_{i=1}^{n-1} O(\lg n)$$

$$= O(n \lg n)$$