COT 6405: Analysis of Algorithms

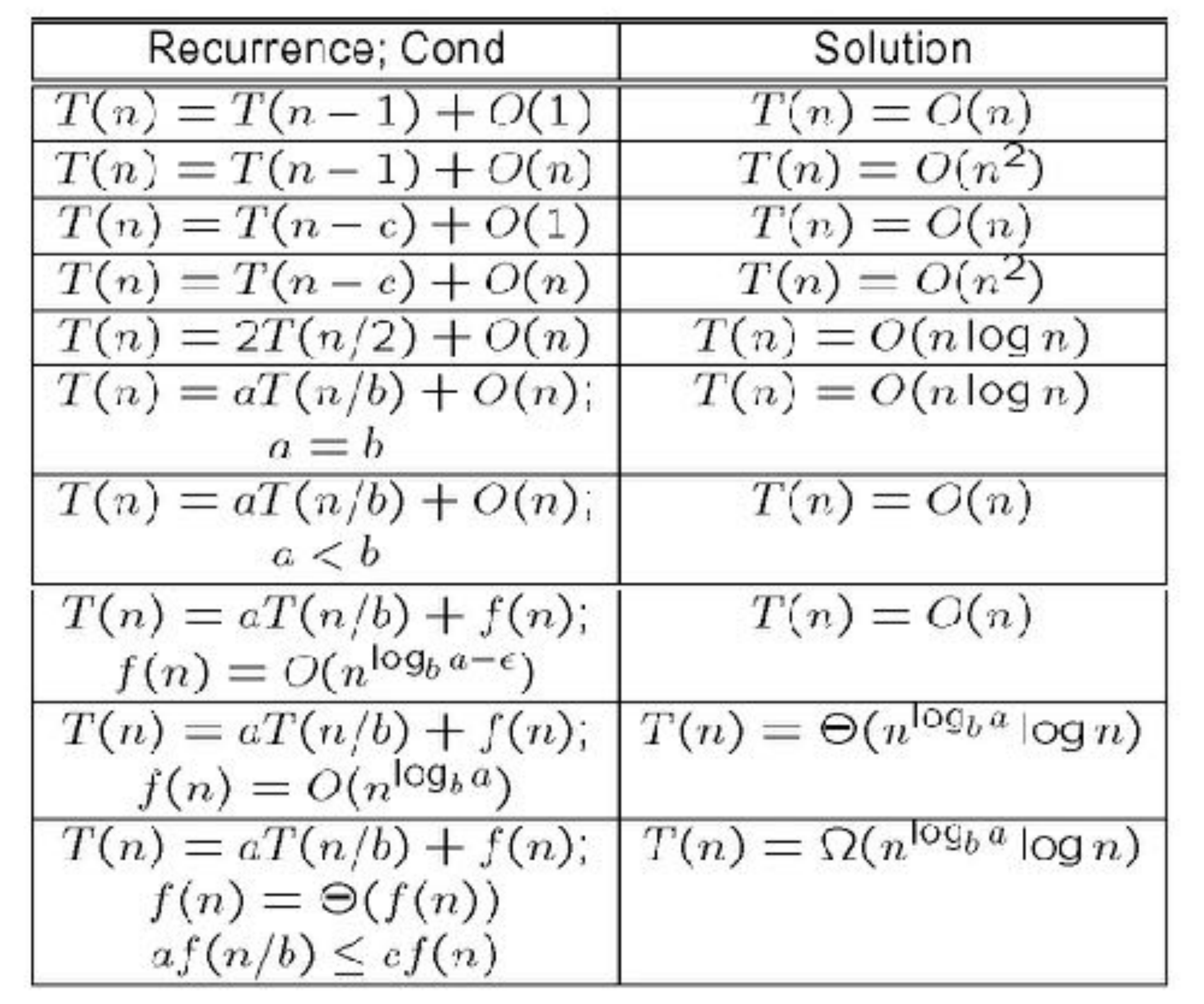
Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/6405F19.html

Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- ShakerSort
- MergeSort
- HeapSort
- QuickSort
- Bucket & Radix Sort
- Counting Sort

Solving Recurrence Relations



Recurrence; Cond	Solution
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-c) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-c) + O(n)$	$T(n) = O(n^2)$
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a = b$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a < b$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(f(n))$ $af(n/b) \leq cf(n)$	$T(n) = \Omega(n^{\log_b a} \log n)$

Solving Recurrences: Recursion-tree method

- Substitution method fails when a good guess is not available
- Recursion-tree method works in those cases
    - Write down the recurrence as a tree with recursive calls as the children
    - Expand the children
    - Add up each level
    - Sum up the levels
- Useful for analyzing divide-and-conquer algorithms
- Also useful for generating good guesses to be used by substitution method

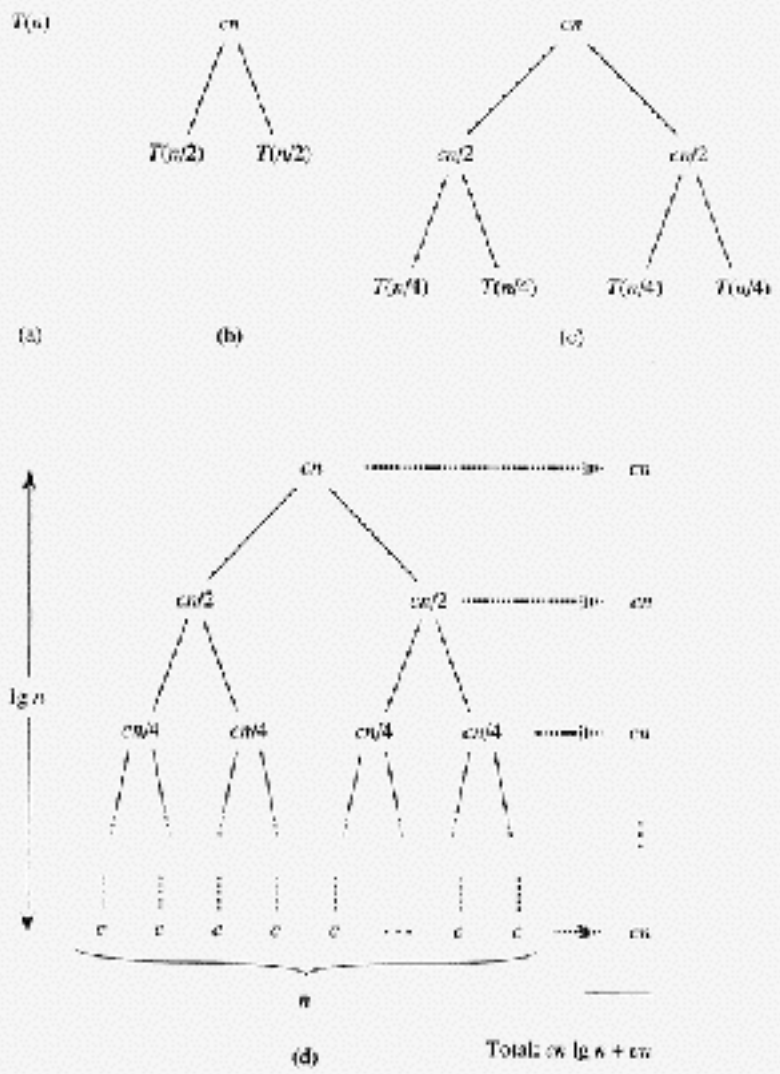


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

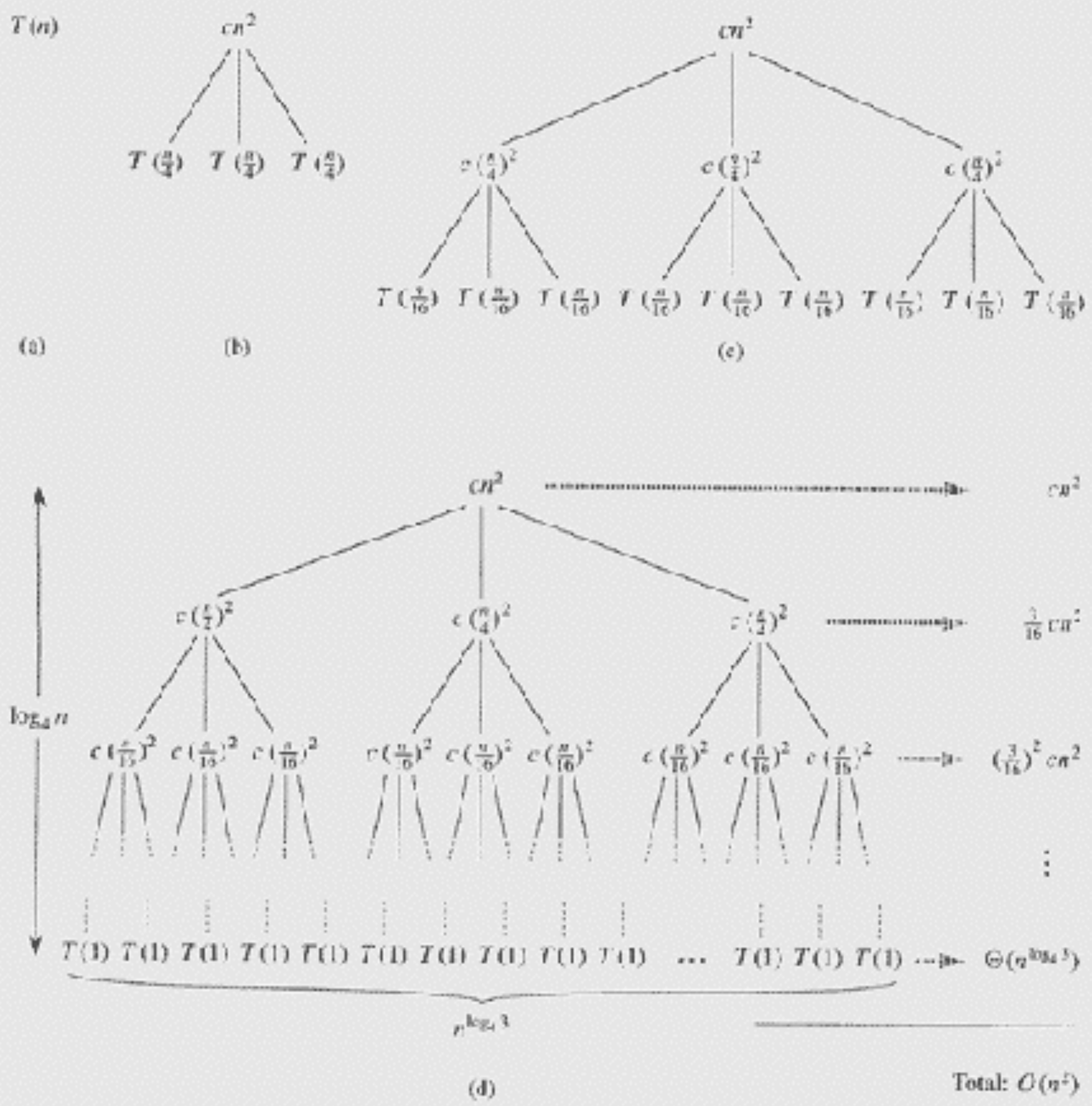


Figure 4.1 The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part (a) shows $T(n)$, which is progressively expanded in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

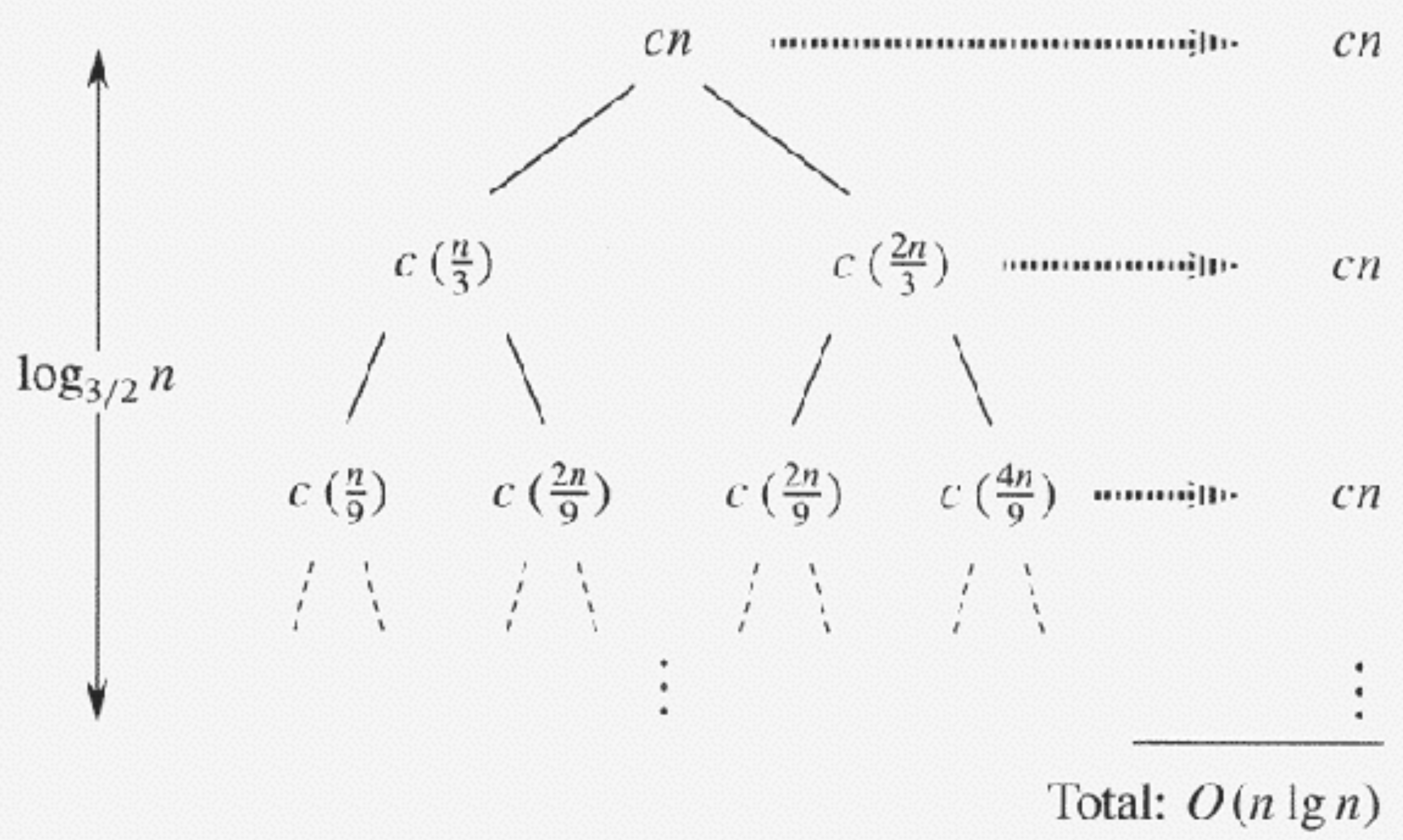


Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

Solving Recurrences using Master Theorem

Master Theorem:

Let a,b >= 1 be constants, let f(n) be a function, and let

$$T(n) = aT(n/b) + f(n)$$

1. If $f(n) = O(n^{\log_b a-e})$ for some constant e>0, then
   - ➨ $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then
   - ➨ $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a+e})$ for some constant e>0, then
   - ➨ $T(n) = \Theta(f(n))$

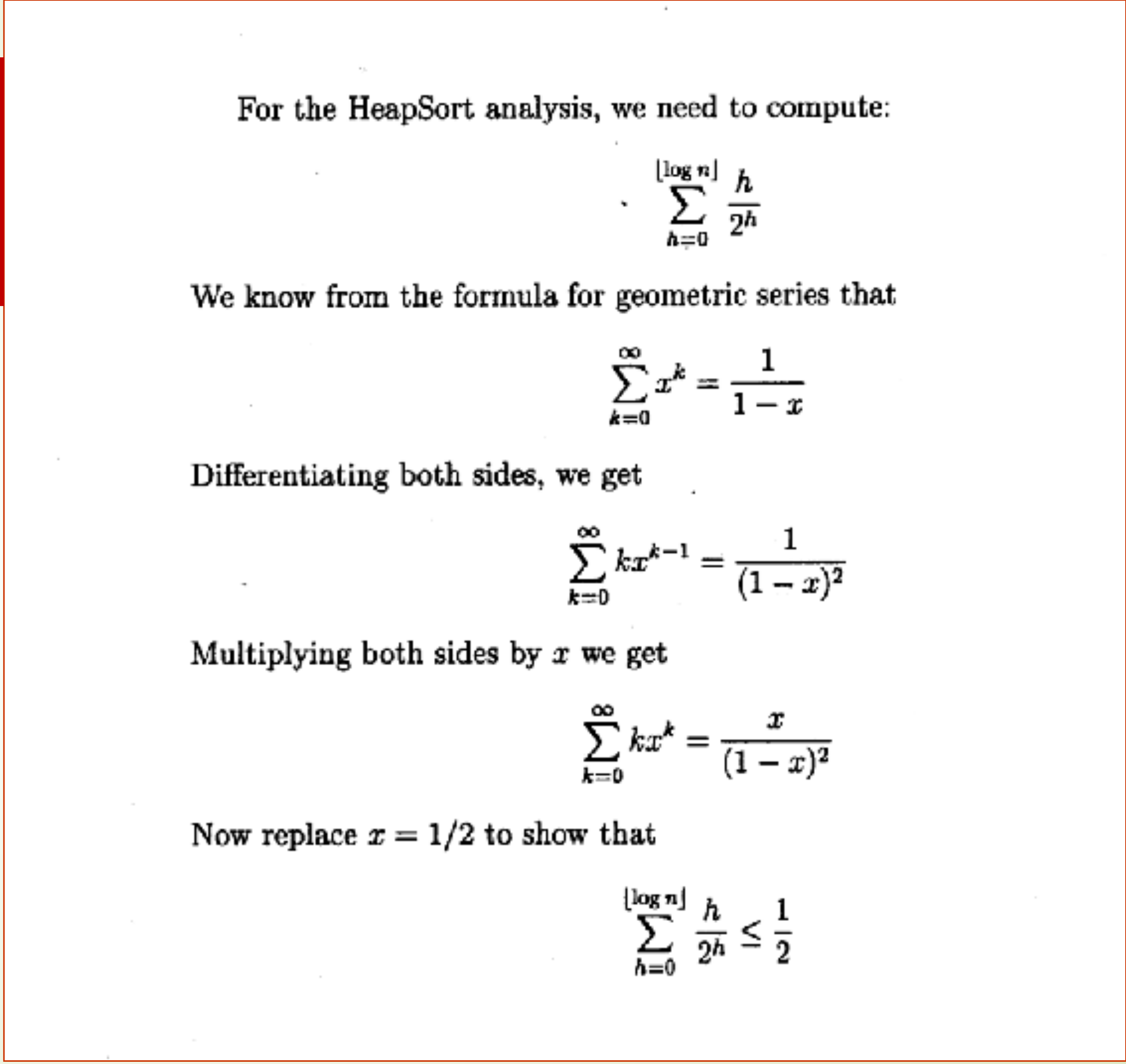
QuickSort

Page 146, CLR

```
QuickSort(A, p, r)
    if (p < r) then
        q = Partition(A, p, r)
        QuickSort(A, p, q-1)
        QuickSort(A, q+1, r)

Partition(A, p, r)
  x = A[r]
  i = p-1
  for j = p to r-1 do
      if A[j] <= x) then
          i++
          exchange(A[i], A[j])
exchange(A[i+1], A[r])
return i+1
```

HeapSort Analysis



For the HeapSort analysis, we need to compute:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

We know from the formula for geometric series that

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Differentiating both sides, we get

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

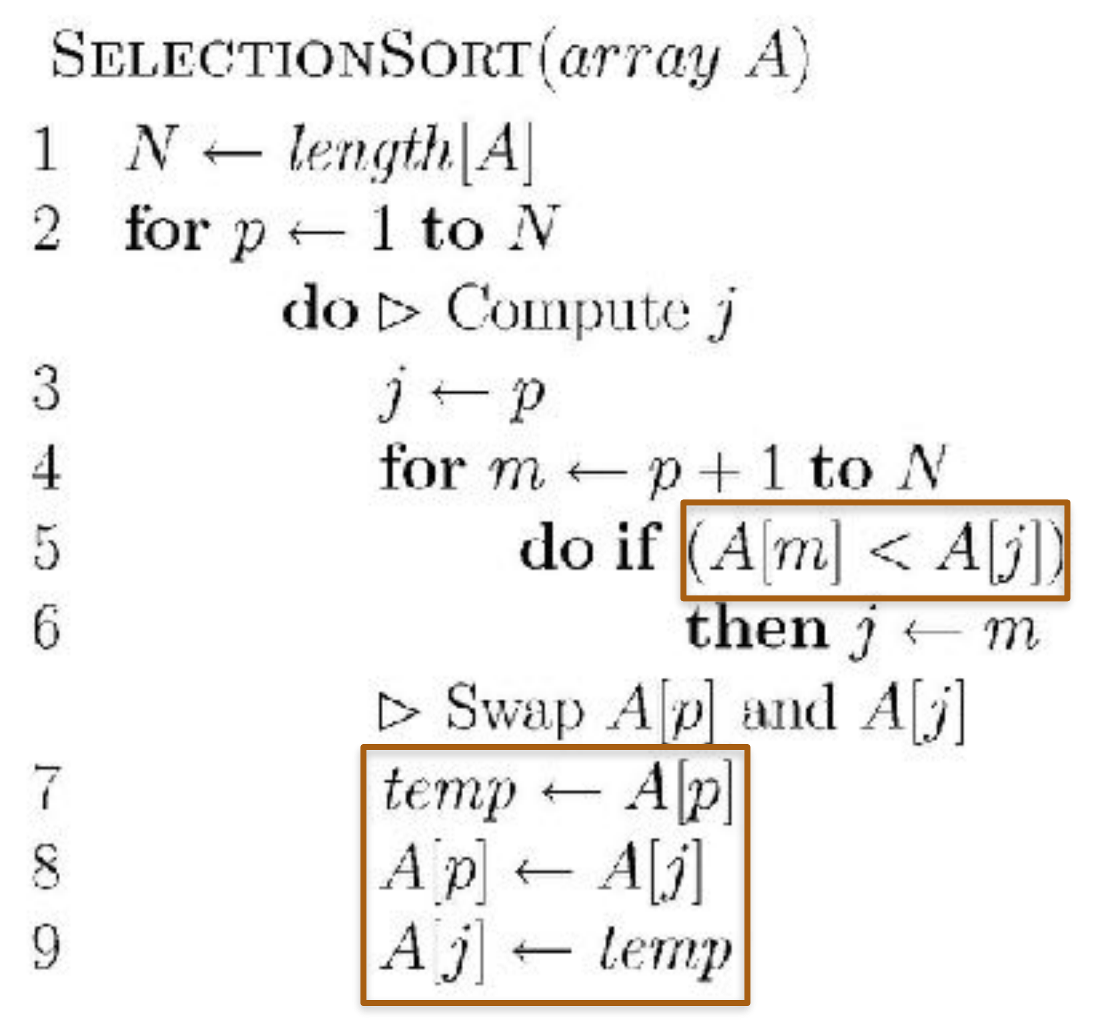
Multiplying both sides by x we get

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Now replace $x = 1/2$ to show that

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \frac{1}{2}$$

SelectionSort – Worst-case analysis



```
SELECTIONSORT(array A)
N ← length[A]
for p ← 1 to N
        do ▷ Compute j
           j ← p
           for m ← p + 1 to N
                do if (A[m] < A[j])
                      then j ← m
           ▷ Swap A[p] and A[j]
           temp ← A[p]
           A[p] ← A[j]
           A[j] ← temp
```

N-p comparisons

3 data movements

Invariant for SelectionSort

- An appropriate invariant has a parameter related to the progress of the algorithm (e.g., iteration number)
- An appropriate invariant helps in proving algorithm is correct
- "At the end of iteration p, the p smallest items are in their correct location"

Algorithm Invariants

- Selection Sort
  - iteration k: the k smallest items are in correct location.
- Insertion Sort
  - iteration k: the first k items are in sorted order.
- Bubble Sort
  - In each pass, every item that does not have a smaller item after it, is moved as far up in the list as possible.
  - Iteration k: k smallest items are in the correct location.
- Shaker Sort
  - In each odd (even) numbered pass, every item that does not have a smaller (larger) item after it, is moved as far up (down) in the list as possible.
  - Iteration k: the k/2 smallest and largest items are in the correct location.

Algorithm Invariants (Cont'd)

- Merge (many lists)
  - Iteration k: the k smallest items from the lists are merged.
- Heapify
  - Iteration with i = k: Subtrees with roots at indices k or larger satisfy the heap property.
- HeapSort
  - Iteration k: Largest k items are in the right location.
- Partition (two sublists)
  - Iteration k (with pointers at i and j): items in locations [1..I] (locations [i+1..j]) are at least as small (large) as the pivot.

Definition of big-Oh

- We say that
  - $F(n) = O(G(n))$

If there exists two positive constants, c and n_0, such that

  - For all $n \geq n_0$, we have $F(n) \leq c\, G(n)$

- Thus, to show that $F(n) = O(G(n))$, you need to find two positive constants that satisfy the condition mentioned above
- Also, to show that $F(n) \neq O(G(n))$, you need to show that for any value of c, there does not exist a positive constant n_0 that satisfies the condition mentioned above

Algorithm Analysis

- Worst-case time complexity*
  - Worst possible time of all input instances of length N
- (Worst-case) space complexity
  - Worst possible spaceof all input instances of length N
- Average-case time complexity
  - Average time of all input instances of length N

Computation Tree for A on n inputs

- Assume A is a comparison-based sorting alg
- Every node represents a comparison between two items in A
- Branching based on result of comparison
- Leaf corresponds to algorithm halting with output
- Every input follows a path in tree
- Different inputs follow different paths
- Time complexity on input x = depth of leaf where it ends on input x

Upper and Lower Bounds

- Time Complexity of a Problem
  - **Difficulty**: Since there can be many algorithms that solve a problem, what time complexity should we pick?
  - **Solution**: Define upper bounds and lower bounds within which the time complexity lies.
- What is the **upper** bound on time complexity of sorting?
  - **Answer**: Since SelectionSort runs in worst-case $O(N^2)$ and MergeSort runs in $O(N \log N)$, either one works as an upper bound.
  - **Critical Point**: Among all upper bounds, the best is the lowest possible upper bound, i.e., time complexity of the best algorithm.
- What is the **lower** bound on time complexity of sorting?
  - **Difficulty**: If we claim that lower bound is $O(f(N))$, then we have to prove that no algorithm that sorts N items can run in worst-case time $o(f(N))$.

Lower Bounds

- It's possible to prove lower bounds for many comparison-based problems.
- For comparison-based problems, for inputs of length N, if there are P(N) possible solutions, then
  - any algorithm needs $\log_2(P(N))$ to solve the problem.
- Binary Search on a list of N items has at least N + 1 possible solutions. Hence lower bound is
  - $\log_2(N+1)$.
- Sorting a list of N items has at least N! possible solutions. Hence lower bound is
  - $\log_2(N!) = O(N \log N)$
- Thus, MergeSort is an optimal algorithm.
  - Because its worst-case time complexity equals lower bound!

Beating the Lower Bound

- Bucket Sort
  - Runs in time O(N+K) given N integers in range [a+1, a+K]
  - If K = O(N), we are able to sort in O(N)
  - How is it possible to beat the lower bound?
  - Only because we know more about the data.
  - If nothing is know about the data, the lower bound holds.
- Radix Sort
  - Runs in time O(d(N+K)) given N items with d digits each in range [1,K]
- Counting Sort
  - Runs in time O(N+K) given N items in range [a+1, a+K]

Stable Sort

- A sort is stable if equal elements appear in the same order in both the input and the output.
- Which sorts are stable? Homework!

Order Statistics

- Maximum, Minimum
  - Upper Bound
    - O(n) because ??
    - We have an algorithm with a single for-loop: n-1 comparisons
  - Lower Bound
    - n-1 comparisons
- MinMax
  - Upper Bound: 2(n-1) comparisons
  - Lower Bound: 3n/2 comparisons
- Max and 2ndMax
  - Upper Bound: (n-1) + (n-2) comparisons
  - Lower Bound: Harder to prove

7	3	1	9	4	8	2	5

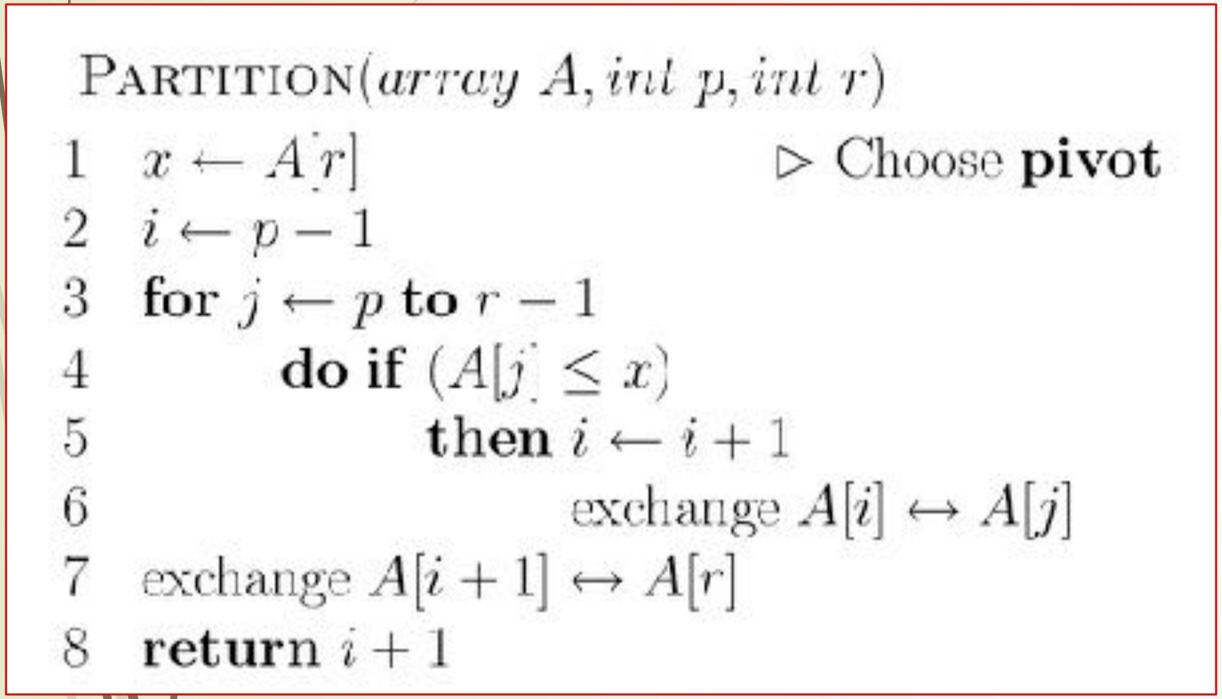
$\underline{\text{Rank}}_A(x)$ = position of x in sorted order o

k-Selection; Median

- Select the k-th smallest item in list
- Naïve Solution
  - Sort;
  - pick the k-th smallest item in sorted list.

    O(n log n) time complexity
- Idea: Modify Partition from QuickSort
  - How?
- Randomized solution: Average case O(n)
- Improved Solution: worst case O(n)

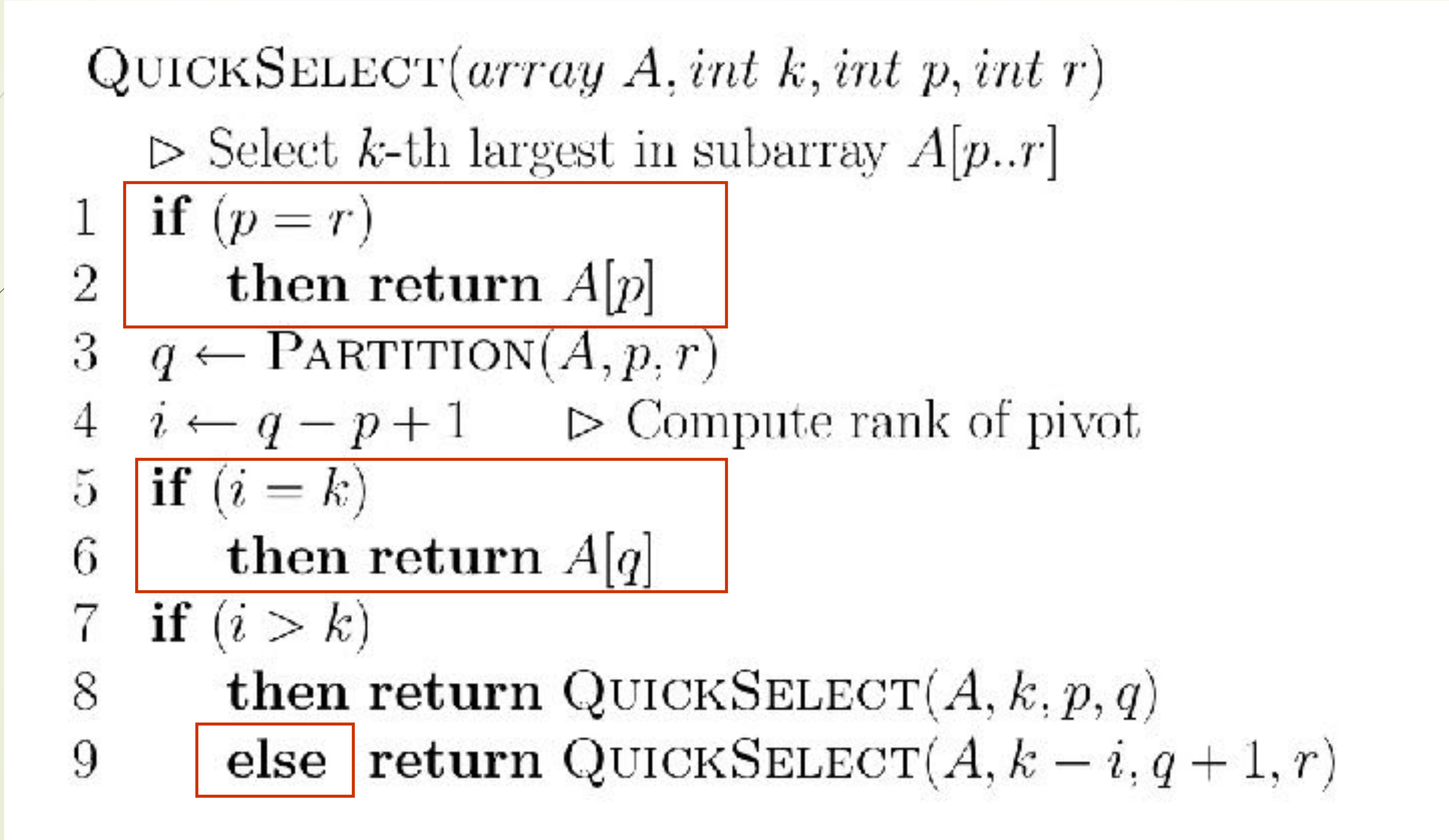
Using Partition for k-Selection



```
PARTITION(array $A$, int $p$, int $r$)
$x \leftarrow A[r]$                        ▷ Choose pivot
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r - 1$
    do if $(A[j] \leq x)$
        then $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$
```

- Perform Partition from QuickSort (assume all unique items)
- Rank(pivot) = 1 + # of items that are smaller than pivot
- If Rank(pivot) = k, we are done
- Else, recursively perform k-Selection in one of the two partitions

QuickSelect: a variant of QuickSort

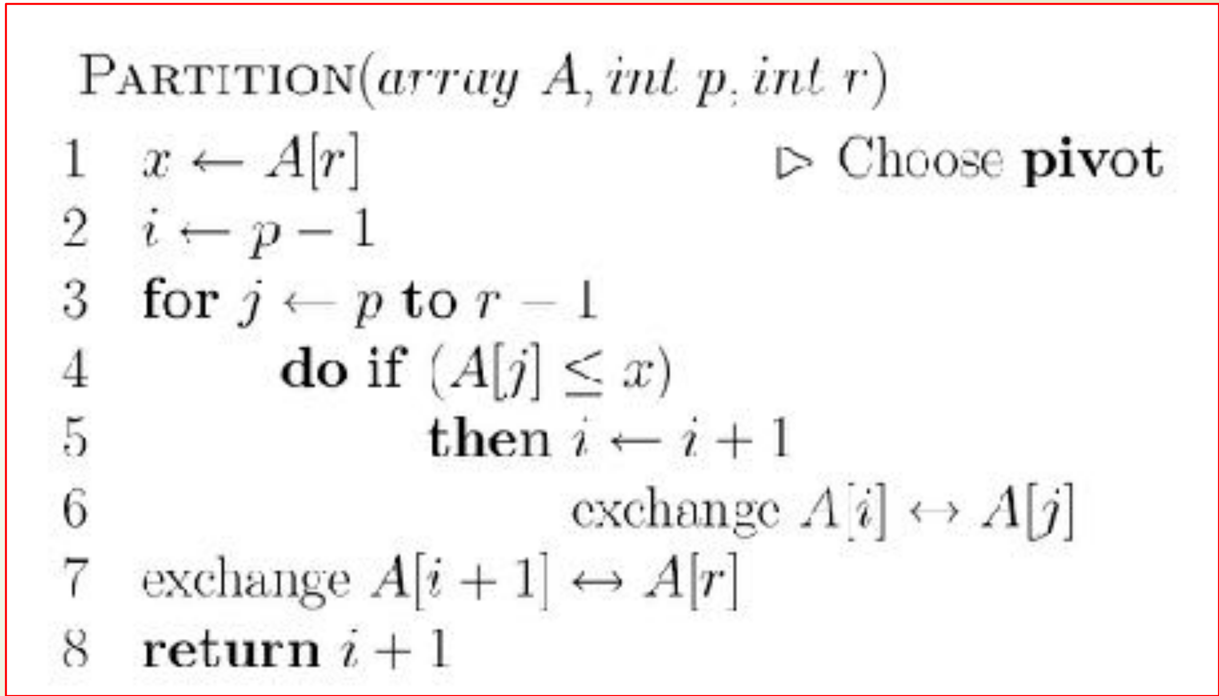


```
QUICKSELECT(array $A$, int $k$, int $p$, int $r$)
    ▷ Select $k$-th largest in subarray $A[p..r]$
    if ($p = r$)
        then return $A[p]$
    $q \leftarrow$ PARTITION($A, p, r$)
    $i \leftarrow q - p + 1$        ▷ Compute rank of pivot
    if ($i = k$)
        then return $A[q]$
    if ($i > k$)
        then return QUICKSELECT($A, k, p, q$)
        else return QUICKSELECT($A, k - i, q + 1, r$)
```

k-Selection Time Complexity

- Perform Partition from QuickSort (assume all unique items)
- Rank(pivot) = 1 + # of items that are smaller than pivot
- If Rank(pivot) = k, we are done
- Else, recursively perform k-Selection in one of the two partitions

- On the average:
  - Rank(pivot) = n / 2
- Average-case time
  - T(N) = T(N/2) + O(N)
  - T(N) = O(N)
- Worst-case time
  - T(N) = T(N-1) + O(N)
  - $T(N) = O(N^2)$



```
PARTITION($array\ A, int\ p, int\ r$)
$x \leftarrow A[r]$                                  ▷ Choose pivot
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r - 1$
        do if $(A[j] \leq x)$
                then $i \leftarrow i + 1$
                        exchange $A[i] \leftrightarrow A[j]$
exchange $A[i+1] \leftrightarrow A[r]$
return $i + 1$
```

Randomized Solution for k-Selection

- Uses RandomizedPartition instead of Partition
  - RandomizedPartition picks the pivot uniformly at random from among the elements in the list to be partitioned.
- Randomized k-Selection runs in $O(N)$ time on the average
- Worst-case behavior is very poor $O(N^2)$

Readings for next class

- Trees,
- Binary Trees,
- Binary Search Trees,
- Balanced Binary Search Trees