CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,

Sogang University

Chapter 7. Quicksort

Introduction

- The quicksort algorithm has a worst-case running time of $\Theta(n^2)$, and an average-case running time of $\Theta(n \lg n)$.

Algorithm	Worst-case	Average-case
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$

- Despite its slow worst-case running time, quicksort is often the best practical choice for sorting
  - Expected running time is $\Theta(n \lg n)$
  - Constant factors hidden in the notation are small

7.1 Description of Quicksort

Quicksort: Strategy

- Quicksort applies the divide-and-conquer method.

- Divide
  - Partition (rearrange) the array $A[p:r]$ into two (possibly empty) subarrays $A[p:q-1]$ (the low side) and $A[q+1:r]$ (the high side).
  - Each element in the low side is smaller than or equal to the pivot $A[q]$.
  - Each element in the high side is larger than or equal to the pivot $A[q]$.

- Conquer
  - Call quicksort recursively to sort each of the subarrays $A[p:q-1]$ and $A[q+1:r]$.

- Combine
  - Do nothing.

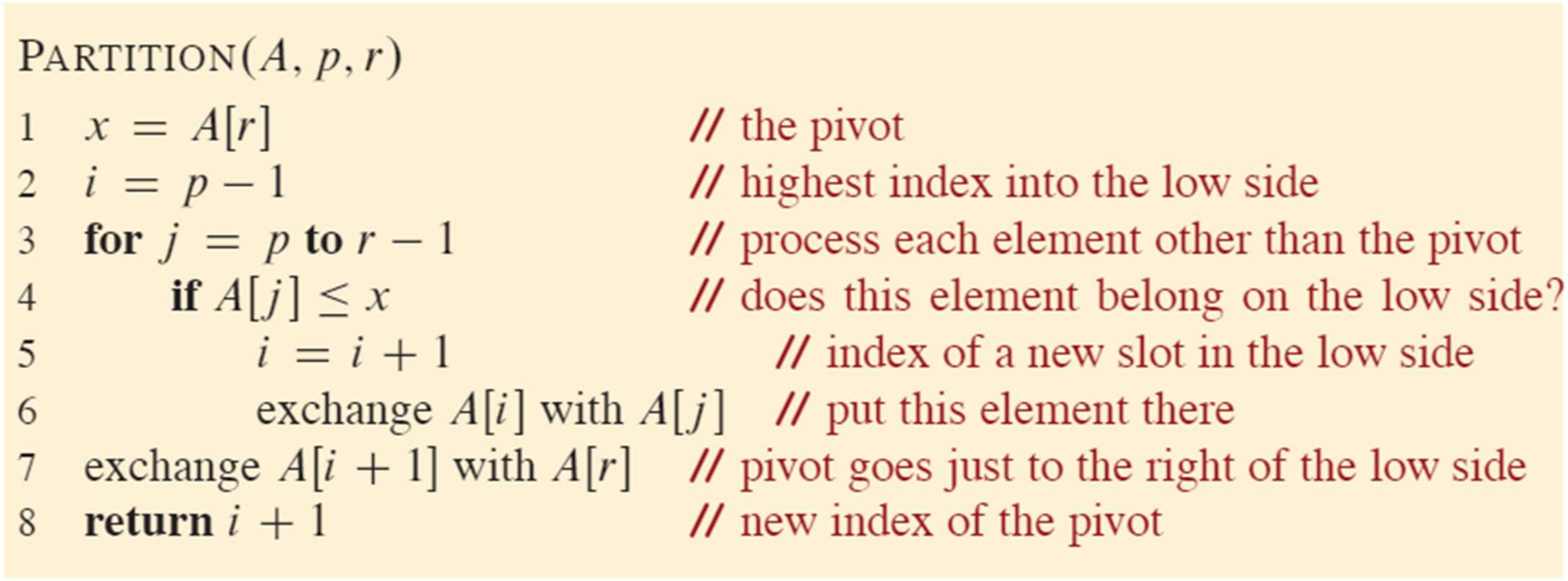
Quicksort: Algorithm

- To sort an entire n-element array $A[1:n]$, we call $\text{QUICKSORT}(A, 1, n)$.

```
QUICKSORT($A, p, r$)
if $p < r$
    // Partition the subarray around the pivot, which ends up in $A[q]$.
    $q$ = PARTITION($A, p, r$)
    QUICKSORT($A, p, q - 1$)  // recursively sort the low side
    QUICKSORT($A, q + 1, r$)  // recursively sort the high side
```

Partitioning the Array

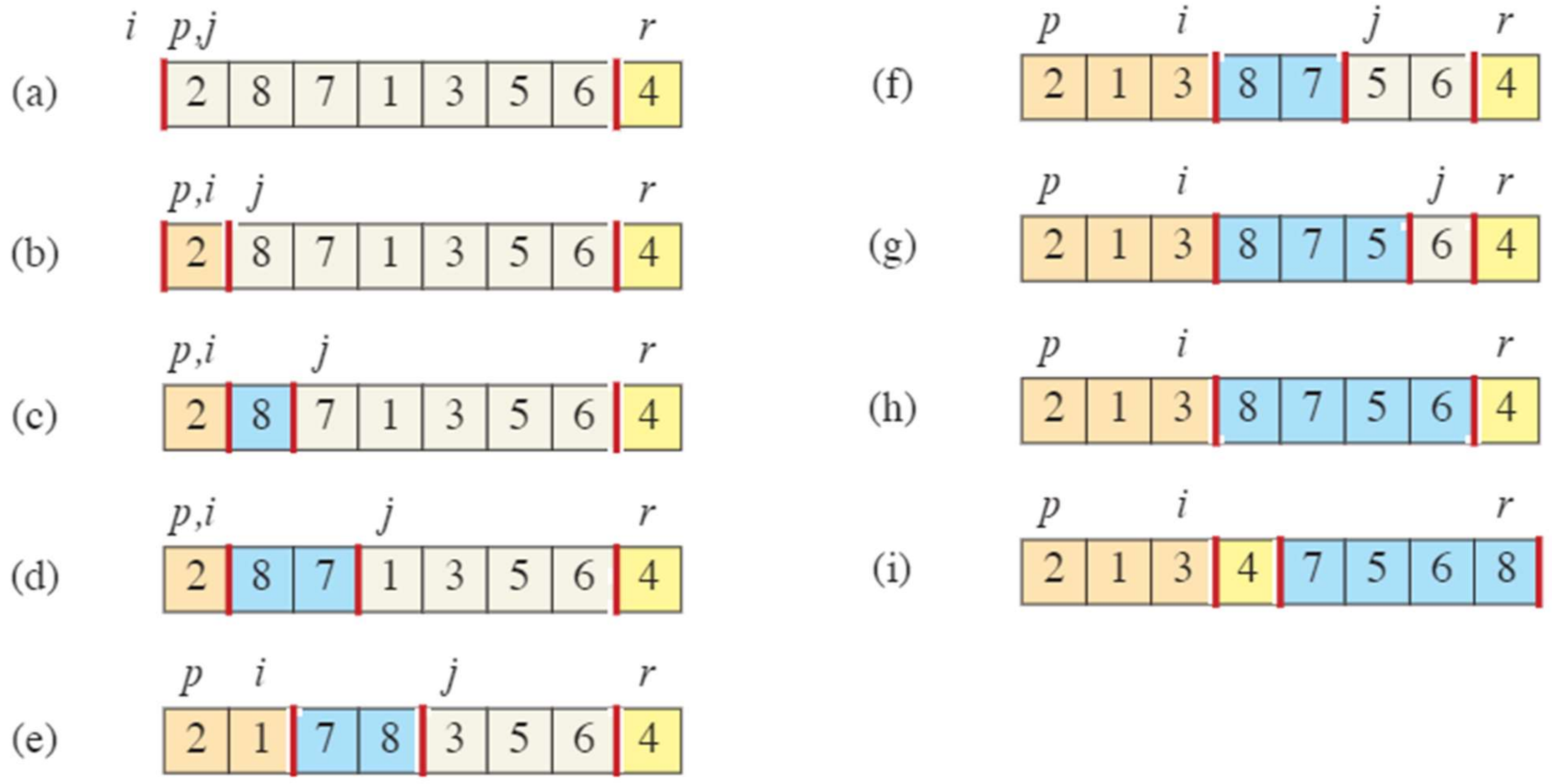
- The PARTITION procedure, we rearrange the subarray $A[p: r]$ in place.
- The procedure returns the index of dividing point.



```
PARTITION($A, p, r$)
$x = A[r]$                          // the pivot
$i = p - 1$                         // highest index into the low side
for $j = p$ to $r - 1$              // process each element other than the pivot
    if $A[j] \le x$                 // does this element belong on the low side?
        $i = i + 1$                     // index of a new slot in the low side
        exchange $A[i]$ with $A[j]$     // put this element there
exchange $A[i + 1]$ with $A[r]$     // pivot goes just to the right of the low side
return $i + 1$                      // new index of the pivot
```

서강대학교 SOGANG UNIVERSITY

Partitioning the Array: Example



(a) i p,j r: 2 8 7 1 3 5 6 4
(b) p,i j r: 2 8 7 1 3 5 6 4
(c) p,i j r: 2 8 7 1 3 5 6 4
(d) p,i j r: 2 8 7 1 3 5 6 4
(e) p i j r: 2 1 7 8 3 5 6 4
(f) p i j r: 2 1 3 8 7 5 6 4
(g) p i j r: 2 1 3 8 7 5 6 4
(h) p i r: 2 1 3 8 7 5 6 4
(i) p i r: 2 1 3 4 7 5 6 8

Partitioning the Array: Loop Invariant Property

- At the beginning of each iteration of the loop of lines 3-6, for any array index k, the following conditions hold:

- If $p \leq k \leq i$, then $A[k] \leq x$;
- If $i + 1 \leq k \leq j - 1$, then $A[k] > x$;
- If $k = r$, then $A[k] = x$.

Partitioning the Array: Loop Invariant Property

- Initialization
  - Prior to the first iteration of the loop, we have $i = p - 1$ and $j = p$.
  - Because no values lie between p and i and no values lie between $i + 1$ and $j - 1$, the first two conditions are trivially satisfied.
  - The assignment in line 1 satisfies the third condition.

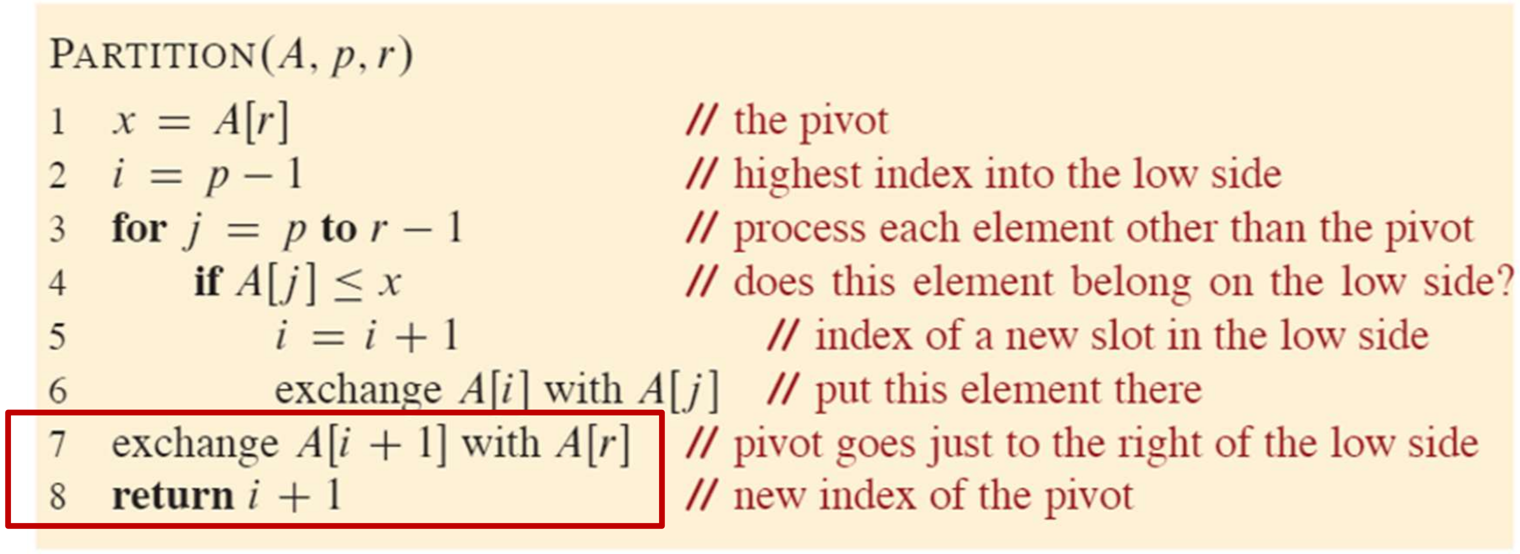
- Maintenance
  - When $A[j] > x$, the only action in the loop is to increment j. After j has been incremented, the second condition holds for $A[j - 1]$ and all other remain unchanged.
  - When $A[j] \leq x$, the loop increments i, swaps $A[i]$ and $A[j]$, then increments j. Because of the swap, we now have $A[i] \leq x$, and condition 1 is satisfied. Similarly we also have that $A[j - 1] > x$, since the item that was swapped into $A[j - 1]$ is, by loop invariant, greater than x.

Partitioning the Array: Loop Invariant Property

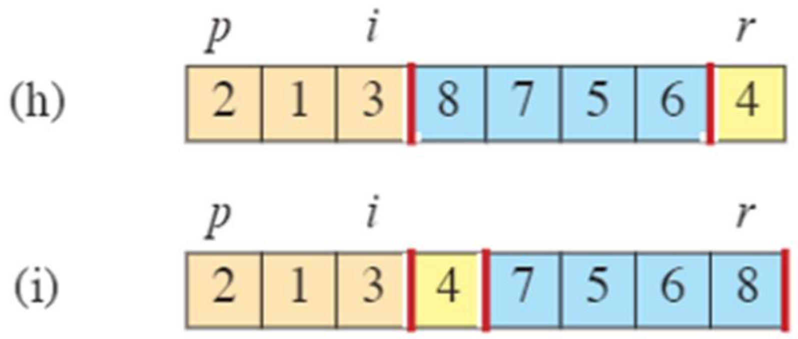
- Termination
  - Since the loop makes exactly $r - p$ iterations, it terminates, where upon $j = r$.
  - At termination, the unexamined subarray $A[j{:}r - 1]$ is empty, and every entry in the array belongs to one of the other three sets described by the invariant.
  - The values in the array have been partitioned into three sets: those less than or equal to x (the low side), those greater than x (the high side), and a singleton set containing x (the pivot).

Partitioning the Array

- In line 7-8, we swap the pivot with the left-most element greater than x.
- It moves the pivot into its correct place in the partitioned array.



```
PARTITION(A, p, r)
1  x = A[r]                              // the pivot
2  i = p − 1                             // highest index into the low side
3  for j = p to r − 1                    // process each element other than the pivot
4      if A[j] ≤ x                       // does this element belong on the low side?
5          i = i + 1                     // index of a new slot in the low side
6          exchange A[i] with A[j]       // put this element there
7  exchange A[i + 1] with A[r]           // pivot goes just to the right of the low side
8  return i + 1                          // new index of the pivot
```



Partitioning the Array: Exercise

- What is the result of PARTITION, when the input array is
- A = <13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11>

7.2 Performance of Quicksort

Running Time of Quicksort

- The running time of quicksort depends on how balanced each partitioning is.
- If the two sides of a partition are about the same size – the partitioning is balanced – then the algorithm runs asymptotically as fast as merge sort ($O(n \lg n)$).
- If the partition is unbalanced, it can run asymptotically as slowly as insertion sort ($O(n^2)$).

Quicksort: Worst-Case Partitioning

- The worst-case behavior for quicksort occurs when the partitioning produces one subproblem with $n-1$ elements and one with 0 elements.

- If this unbalanced partitioning arises in each recursive call, then the recurrence relation would be:

- $T(n) = T(n-1) + T(0) + \Theta(n)$
  - Partitioning takes $\Theta(n)$ time.

- Solving this recurrence relation, we get $T(n) = \Theta(n^2)$.

- If we are trying to sort an array which is already sorted
  - Insertion sort takes $O(n)$ time.
  - Quicksort takes $O(n^2)$ time.

Quicksort: Best-case Partitioning

- In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor (n-1)/2 \rfloor \leq n/2$ and one of size $\lceil (n-1)/2 \rceil \leq n/2$.

- In this case, quicksort runs much faster. An upper bound on the running time can be described as

- $T(n) = 2\left(\frac{n}{2}\right) + \Theta(n)$

- Using Master Theorem, we get $T(n) = \Theta(n \lg n)$.

Quicksort: Balanced Partitioning

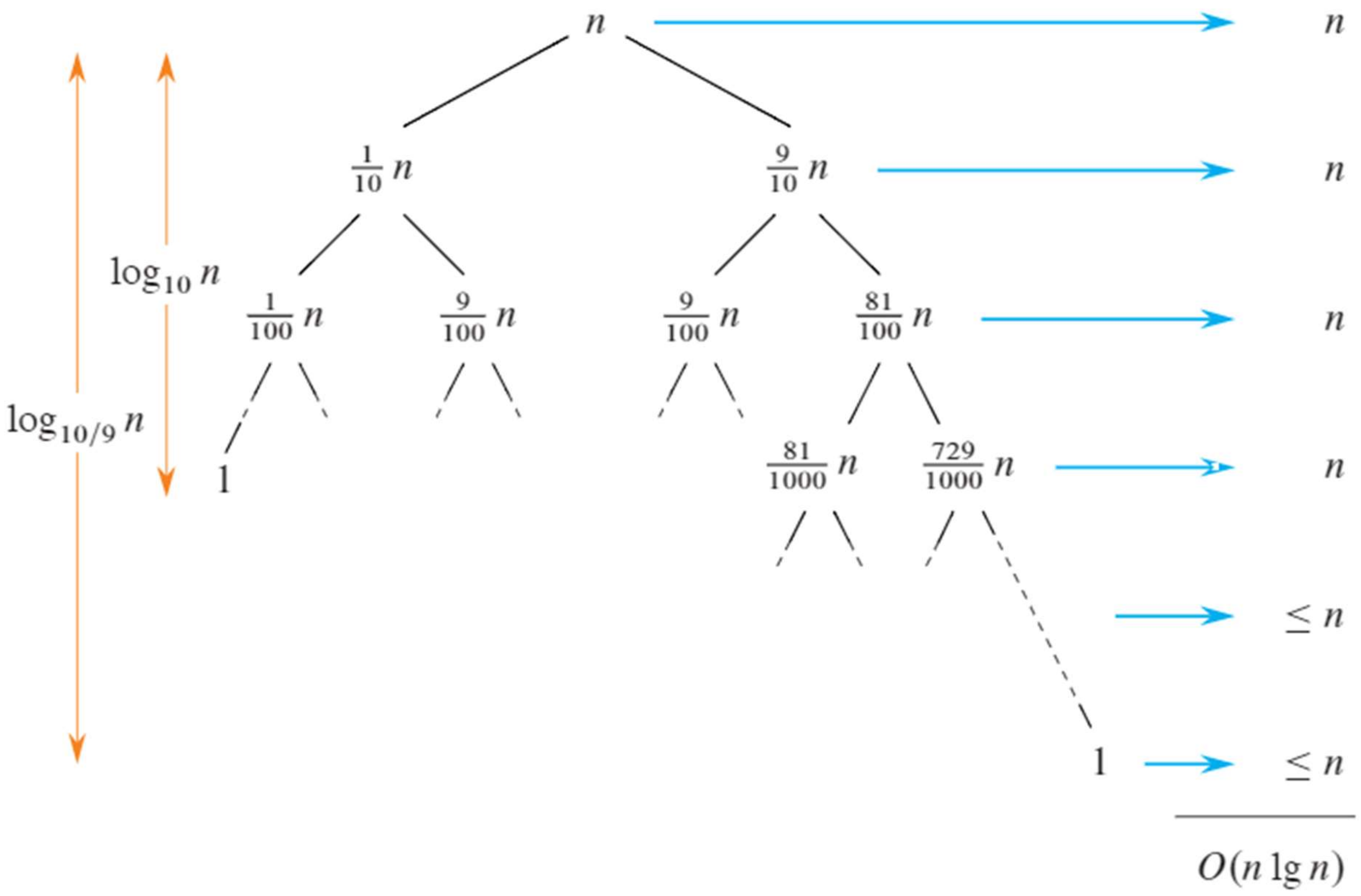
- For quicksort, the average-case running time of quicksort is much closer to the best case than the worst case.

- Suppose the partitioning algorithm always produces a 9-to-1 proportional split, which seems quite unbalanced. We obtain the recurrence:

- $T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$

Quicksort: Balanced Partitioning

- Recursion tree for $T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$



Quicksort: Balanced Partitioning

- Recursion tree for $T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$
- For simplicity, we use n instead of $\Theta(n)$ for the driving function.

- Every level of the tree has cost n, until the recursion bottoms out in a base case at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most n.

- The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$.

- Thus, with a 9-to-1 proportional split at every level of recursion, quicksort runs in $O(n \lg n)$.
  - 99-to-1 split yields $O(n \lg n)$ running time as well.

- Any split of constant proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. Therefore, the running time is $O(n \lg n)$.

7.3 A Randomized Version of Quicksort

Quicksort: Randomized Version

- We want to explore the average-case behavior of quicksort.
- We can assume that all permutations of the input numbers are equally likely, but we do not know whether it is actually true.
- We make the randomized version of quicksort.
  - Before calling PARTITION with A[r] as pivot, we choose a random element in the array and swap the element with A[r].
  - Since we randomly choose a pivot, it does not depend on a particular input pattern.

```
RANDOMIZED-PARTITION($A, p, r$)
$i$ = RANDOM($p, r$)
exchange $A[r]$ with $A[i]$
return 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($A, q + 1, r$)
```

서강대학교 SOGANG UNIVERSITY

7.4 Analysis of Quicksort

Quicksort: Average-Case Analysis

- We've seen that the worst-case running time of quicksort is $\Theta(n^2)$.
- Now we calculate the expected running time of RANDOMIZED-QUICKSORT.
- We assume that the values of the elements being sorted are distinct.
  - The result is true without this assumption as well.

- The QUICKSORT and RANDOMIZED-QUICKSORT procedures differ only in how they select pivot elements.
- We can therefore analyze RANDOMIZED-QUICKSORT by considering the QUICKSORT and PARTITION procedures, but with the assumption that pivot elements are selected randomly from the subarray.

Quicksort: Average-Case Analysis

- Relation between asymptotic running time of QUICKSORT and the number of times elements are compared (line 4 in PARTITION)

- Lemma 7.1
  - The running time of QUICKSORT on an n-element array is $O(n + X)$, where X is the number of element comparisons performed.

- Proof
  - The running time of QUICKSORT is dominated by the time spent in the PARTITION procedure.
  - Each time PARTITION is called, it selects a pivot element, which is never included in any future recursive calls to QUICKSORT and PARTITION.
  - Thus, there can be at most n calls to PARTITION over the entire execution of the quicksort algorithm.
  - Each time QUICKSORT calls PARTITION, it also recursively calls itself twice, so there are at most $2n$ calls to the QUICKSORT procedure itself.

Quicksort: Average-Case Analysis

- Proof (cont.)
  - One call to PARTITION takes $O(1)$ time plus an amount of time that is proportional to the number of iterations of the **for** loops in lines 3-6.
  - Each iteration of this **for** loop performs one comparison in line 4, comparing the pivot element to another element of the array A.
  - Therefore, the total time spent in the for loop across all executions is proportional to X.

  - Since there are at most n calls to PARTITION and the time spent outside for loop is $O(1)$ for each call, the total time spent in PARTITION outside of the for loop is $O(n)$.
  - Thus the total time for quicksort is $O(n+X)$.

Quicksort: Average-Case Analysis

- From Lemma 7.1, our goal for analyzing RANDOMIZED-QUICKSORT is to compute the expected value $E[X]$ of the random variable X denoting the total number of comparisons performed in all calls to PARTITION.

- To do so, we must understand when the quicksort algorithm compares two elements and when it does not.

- Let us index the elements of array A by their position in the sorted output. Although the elements in A may start out in any order, we will refer to them by $z_1, z_2, \dots, z_n$, where $z_1 < z_2 < \cdots < z_n$ with strict inequality.

- We denote $\{z_i, z_{i+1}, \dots, z_j\}$ by Z_{ij}.

Quicksort: Average-Case Analysis

- Lemma 7.2
  - During the execution of RANDOMIZED-QUICKSORT on an array of n distinct elements $z_1 < z_2 < \cdots < z_n$ an element z_i is compared with an element z_j, where $i < j$, if and only if one of them is chosen as a pivot before any other element in the set Z_{ij}. Moreover, no two elements are ever compared twice.

- Proof
  - Let's look at the first time that an element $x \in Z_{ij}$ is chosen as a pivot during the execution of the algorithm. There are three cases to consider. If x is neither z_i nor z_j - that is, $z_i < x < z_j$ - then z_i and z_j are not compared at any subsequent time, because they fall into different sides of partition around x.
  - If $x = z_i$, then PARTITION compares z_i with every other item in Z_{ij}.
  - Similarly, if $x = z_j$, then PARTITION compares z_j with every other item in Z_{ij}.
  - Thus, z_i and z_j are compared if and only if the first element to be chosen as a pivot from Z_{ij} is either z_i or z_j.
  - If z_i or z_j is chosen as pivot, they are never compared again because the pivot is removed from future comparisons.

Quicksort: Average-Case Analysis

- Example of Lemma 7.2
  - Suppose the input array is {8, 3, 7, 1, 4, 5, 9, 2, 10, 6}.
  - Suppose the first pivot element is 7.
  - Then the first call to PARTITION separates the numbers into two sets {1, 2, 3, 4, 5, 6} and {8, 9, 10}.
  - In the process, pivot 7 is compared with all other elements, but no number from the first set (e.g. 2) will ever be compared with any number from the second set (e.g. 9).

  - The values 7 and 9 are compared because 7 is the first item from $Z_{7,9}$ to be chosen as pivot.
  - In contrast, 2 and 9 are never compared because the first pivot element chosen from $Z_{2,9}$ is 7.

Quicksort: Average-Case Analysis

- Lemma 7.3
  - Consider an execution of the procedure RANDOMIZED-QUICKSORT on an array of n distinct elements $z_1 < z_2 < \cdots < z_n$. Given two arbitrary elements z_i and z_j where $i < j$, the probability that they are compared is $2/(j - i + 1)$.

- Proof
  - We examine the tree of recursive calls that RANDOMIZED-QUICKSORT makes, and consider the sets of elements provided as input to each call.
  - Initially, the root set contains all the elements of Z_{ij}, since the root set contains every element in A.
  - The elements belonging to Z_{ij} all stay together for each recursive call of RANDOMIZED-QUICKSORT until PARTITION chooses some elements $x \in Z_{ij}$ as a pivot.
  - From that point on, the pivot x appears in no subsequent input set.

Quicksort: Average-Case Analysis

- Proof (cont.)
  - The first time that RANDOM chooses a pivot $x \in Z_{ij}$ from a set containing all the elements of Z_{ij}, each element in Z_{ij} is equally likely to be x because the pivot is chosen uniformly at random.
  - Since $|Z_{ij}| = j - i + 1$, the probability is $1/(j - i + 1)$ that any given element in Z_{ij} is the first pivot chosen from Z_{ij}.
  - By Lemma 7.2, we have
  - $$\begin{aligned}\Pr\{z_i \text{ is compared with } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\ &= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\ &+ \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\ &= \frac{2}{j-i+1}\end{aligned}$$
  - The second line follows from the first because two events are mutually exclusive.

Quicksort: Average-Case Analysis

- Theorem 7.4
  - The expected running time of RANDOMIZED-QUICKSORT on an input of n distinct elements is $O(n \lg n)$.

- Proof
  - Let the n distinct elements be $z_1 < z_2 < \cdots < z_n$, and for $1 \leq i < j \leq n$, define the indicator random variable $X_{ij} = I\{z_i \text{ is compared with } z_j\}$.
  - From Lemma 7.2, each pair is compared at most once, and so we express X as:

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

Quicksort: Average-Case Analysis

- Proof (cont.)
  - By taking expectations of both sides and using linearity of expectation, we obtain

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared with } z_j\} && \text{(by Lemma 5.1)} \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} && \text{(by Lemma 7.3)} .
\end{aligned}$$

Quicksort: Average-Case Analysis

- Proof (cont.)
  - We can evaluate this sum using a change of variables ($k = j - i$) and the bound on the harmonic series.

$$
\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) .
\end{aligned}
$$

Quicksort: Average-Case Analysis

- This bound and Lemma 7.1 allow us to conclude that the expected running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$.

Quick sort: a simple implementation

```
// Sort a list from A[left] to A[right].

void quick_sort(item_type *A, int left, int right) {
  int pivot;

  if (right - left > 0) {
    pivot = partition(A, left, right);

    quick_sort(A, left, pivot - 1);
    quick_sort(A, pivot + 1, right);
  }
}
```

```
#define SWAP(a, b) { item_type tmp; tmp = a; a = b; b = tmp; }

int partition(item_type *A, int left, int right) {
  int i, pivot;

  pivot = left;
  for (i = left; i < right; i++) {
    if (A[i] < A[right]) {
      SWAP(A[i], A[pivot]);
      pivot++;
    }
  }
  SWAP(A[right], A[pivot]);
  return(pivot);
}
```

How is the pivot element chosen in this function?

```
18 20 28  0 38  8  2 16 10 14 24 30 34 12 32 22  6  4 36 26

18 20  0  8  2 16 10 14 24 12 22  6  4 26 32 38 30 34 36 28
                                     (13)
```

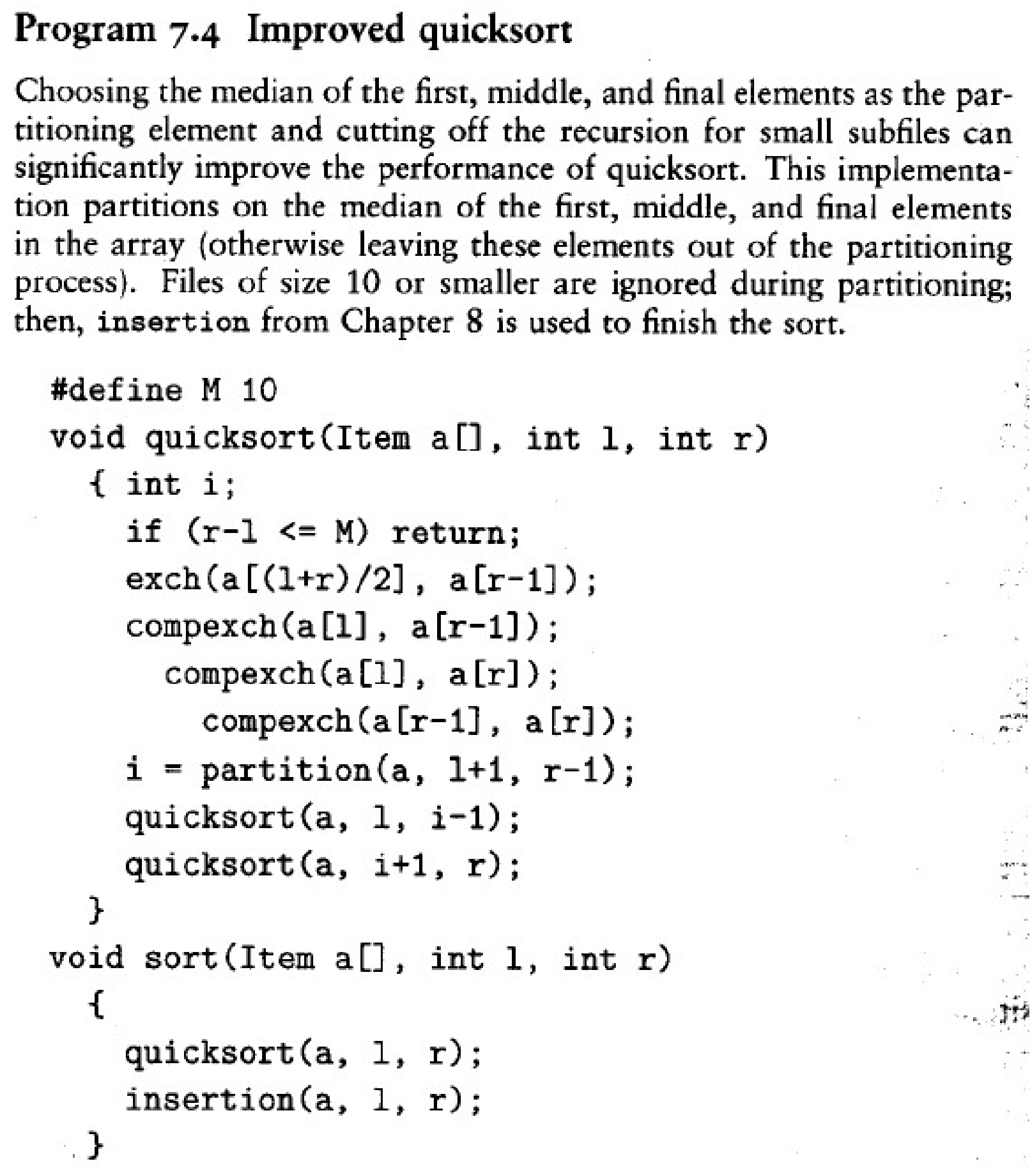
26 **(13)**

Improving performance of quick sort

- How can you select a "good" pivot element?
  - Choose a random element in the list.
  - Choose the median of the first, middle, and final elements in the list.
  - Etc.

- How can you minimize the bookkeeping cost involved in the recursive calls?
  - Much of the pushing and popping of the frame stack is unnecessary.
  - List of size m or smaller are ignored during quick sort, then do a single sorting pass at the end.

Improving performance of quick sort



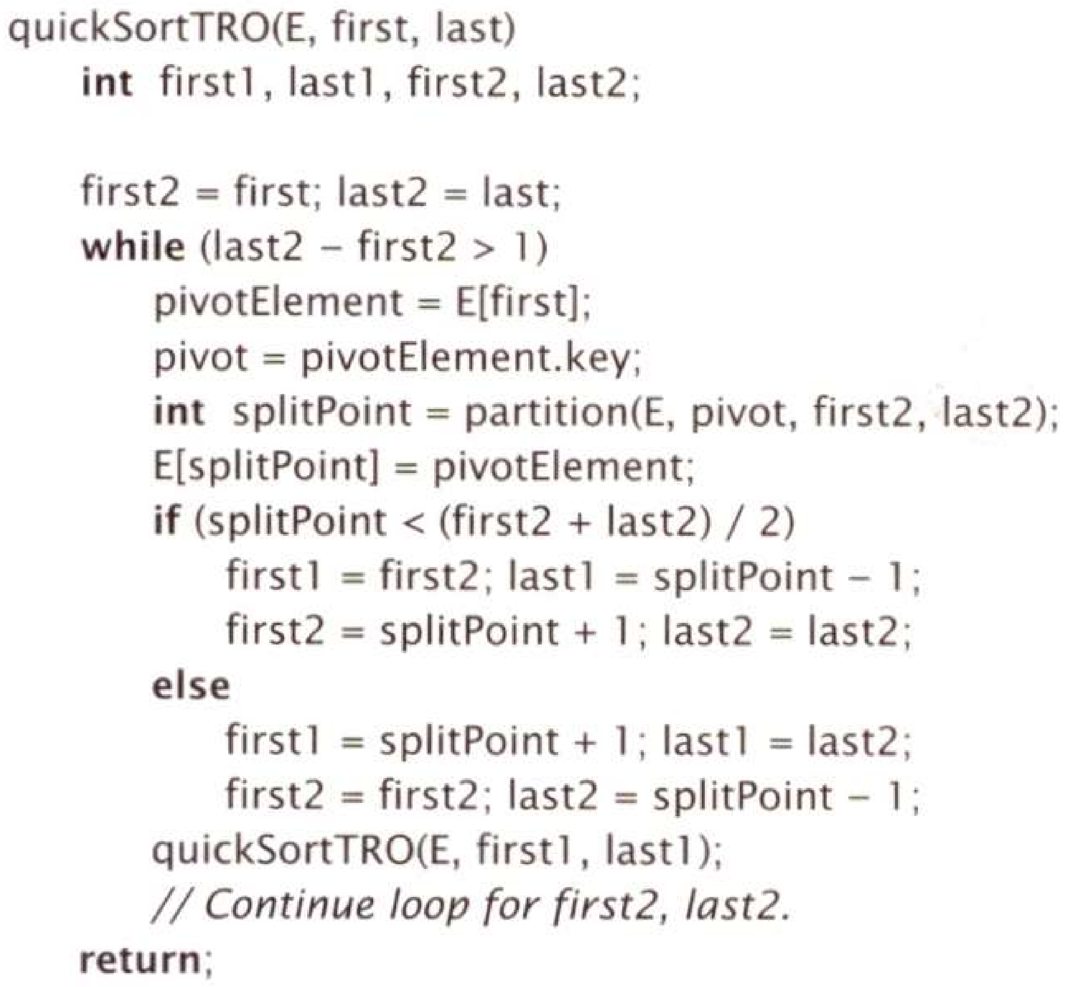
Program 7.4 Improved quicksort

Choosing the median of the first, middle, and final elements as the partitioning element and cutting off the recursion for small subfiles can significantly improve the performance of quicksort. This implementation partitions on the median of the first, middle, and final elements in the array (otherwise leaving these elements out of the partitioning process). Files of size 10 or smaller are ignored during partitioning; then, `insertion` from Chapter 8 is used to finish the sort.

```
#define M 10
void quicksort(Item a[], int l, int r)
  { int i;
    if (r-l <= M) return;
    exch(a[(l+r)/2], a[r-1]);
    compexch(a[l], a[r-1]);
      compexch(a[l], a[r]);
        compexch(a[r-1], a[r]);
    i = partition(a, l+1, r-1);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
void sort(Item a[], int l, int r)
  {
    quicksort(a, l, r);
    insertion(a, l, r);
  }
```

Improving performance of quick sort

- Tail recursion optimization
  - After each partition,
    - Smaller subrange ← recursive call
    - Larger subrange ← handled iteratively
  - Avoid making recursive calls on the larger subrange.
  - The depth of recursion ≤ O(log n)



```
quickSortTRO(E, first, last)
    int first1, last1, first2, last2;

    first2 = first; last2 = last;
    while (last2 − first2 > 1)
        pivotElement = E[first];
        pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first2, last2);
        E[splitPoint] = pivotElement;
        if (splitPoint < (first2 + last2) / 2)
            first1 = first2; last1 = splitPoint − 1;
            first2 = splitPoint + 1; last2 = last2;
        else
            first1 = splitPoint + 1; last1 = last2;
            first2 = first2; last2 = splitPoint − 1;
        quickSortTRO(E, first1, last1);
        // Continue loop for first2, last2.
    return;
```

Quick sort vs. Introspective sort: Quick sort

```
Algorithm QUICKSORT(A, f, b)
                    Inputs: A, a random access data structure containing the sequence
                        of data to be sorted, in positions A[f], ..., A[b − 1];
                      f, the first position of the sequence
                      b, the first position beyond the end of the sequence
                    Output: A is permuted so that A[f] ≤ A[f+1] ≤ . . . ≤ A[b − 1]
    QUICKSORT_LOOP(A, f, b)
    INSERTION_SORT(A, f, b)

Algorithm QUICKSORT_LOOP(A, f, b)
                    Inputs: A, f, b as in QUICKSORT
                    Output: A is permuted so that A[i] ≤ A[j]
                      for all i, j: f ≤ i < j < b and size_threshold < j − i
    while b − f > size_threshold
        do  p := PARTITION(A, f, b, MEDIAN_OF_3(A[f], A[f+(b−f)/2], A[b−1]))
            if (p − f ≥ b − p)
                then QUICKSORT_LOOP(A, p, b)
                     b := p
                else QUICKSORT_LOOP(A, f, p)
                     f := p
```

Average-case: $O(n \log n)$
Worst-case: $O(n^2)$

Quick sort vs. Introspective sort: Introspective sort

```
Algorithm INTROSORT(A, f, b)
                    Inputs: A, a random access data structure containing the sequence
                        of data to be sorted, in positions A[f], ..., A[b − 1];
                      f, the first position of the sequence
                      b, the first position beyond the end of the sequence
                    Output: A is permuted so that A[f] ≤ A[f+1] ≤ ... ≤ A[b − 1]
    INTROSORT_LOOP(A, f, b, 2 * FLOOR_LG(b − f))
    INSERTION_SORT(A, f, b)

Algorithm INTROSORT_LOOP(A, f, b, depth_limit)
                    Inputs: A, f, b as in INTROSORT;
                      depth_limit, a nonnegative integer
                    Output: A is permuted so that A[i] ≤ A[j]
                      for all i, j: f ≤ i < j < b and size_threshold < j − i
    while b − f > size_threshold
        do if depth_limit = 0
               then HEAPSORT(A, f, b)
                    return
           depth_limit := depth_limit − 1
           p := PARTITION(A, f, b, MEDIAN_OF_3(A[f], A[f+(b−f)/2], A[b−1]))
           INTROSORT_LOOP(A, p, b, depth_limit)
           b := p
```

Average-case: $O(n \log n)$
Worst-case: $O(n \log n)$

Performance comparison

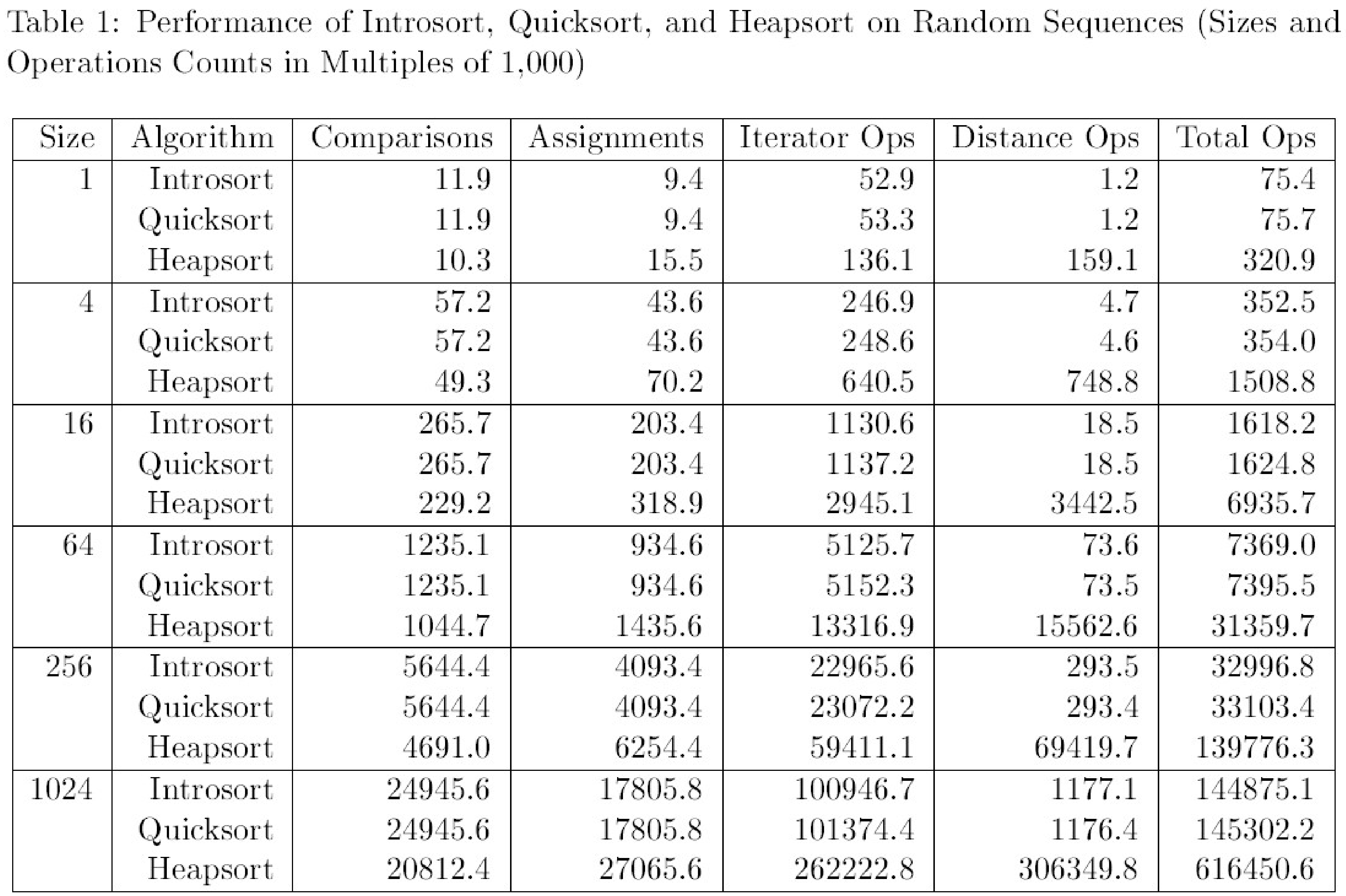


Table 1: Performance of Introsort, Quicksort, and Heapsort on Random Sequences (Sizes and Operations Counts in Multiples of 1,000)

Size	Algorithm	Comparisons	Assignments	Iterator Ops	Distance Ops	Total Ops
1	Introsort	11.9	9.4	52.9	1.2	75.4
	Quicksort	11.9	9.4	53.3	1.2	75.7
	Heapsort	10.3	15.5	136.1	159.1	320.9
4	Introsort	57.2	43.6	246.9	4.7	352.5
	Quicksort	57.2	43.6	248.6	4.6	354.0
	Heapsort	49.3	70.2	640.5	748.8	1508.8
16	Introsort	265.7	203.4	1130.6	18.5	1618.2
	Quicksort	265.7	203.4	1137.2	18.5	1624.8
	Heapsort	229.2	318.9	2945.1	3442.5	6935.7
64	Introsort	1235.1	934.6	5125.7	73.6	7369.0
	Quicksort	1235.1	934.6	5152.3	73.5	7395.5
	Heapsort	1044.7	1435.6	13316.9	15562.6	31359.7
256	Introsort	5644.4	4093.4	22965.6	293.5	32996.8
	Quicksort	5644.4	4093.4	23072.2	293.4	33103.4
	Heapsort	4691.0	6254.4	59411.1	69419.7	139776.3
1024	Introsort	24945.6	17805.8	100946.7	1177.1	144875.1
	Quicksort	24945.6	17805.8	101374.4	1176.4	145302.2
	Heapsort	20812.4	27065.6	262222.8	306349.8	616450.6

Performance comparison

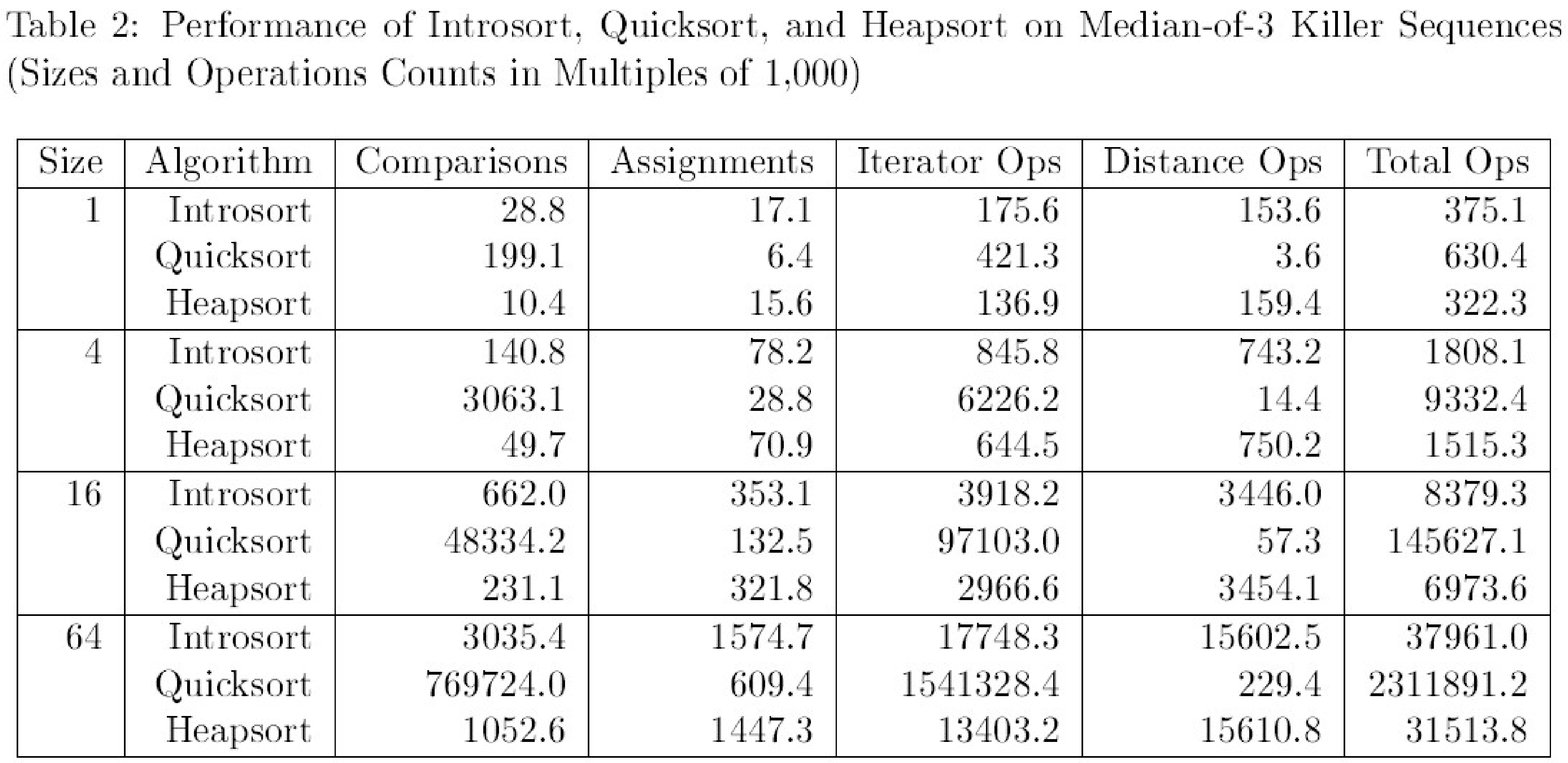


Table 2: Performance of Introsort, Quicksort, and Heapsort on Median-of-3 Killer Sequences (Sizes and Operations Counts in Multiples of 1,000)

Size	Algorithm	Comparisons	Assignments	Iterator Ops	Distance Ops	Total Ops
1	Introsort	28.8	17.1	175.6	153.6	375.1
	Quicksort	199.1	6.4	421.3	3.6	630.4
	Heapsort	10.4	15.6	136.9	159.4	322.3
4	Introsort	140.8	78.2	845.8	743.2	1808.1
	Quicksort	3063.1	28.8	6226.2	14.4	9332.4
	Heapsort	49.7	70.9	644.5	750.2	1515.3
16	Introsort	662.0	353.1	3918.2	3446.0	8379.3
	Quicksort	48334.2	132.5	97103.0	57.3	145627.1
	Heapsort	231.1	321.8	2966.6	3454.1	6973.6
64	Introsort	3035.4	1574.7	17748.3	15602.5	37961.0
	Quicksort	769724.0	609.4	1541328.4	229.4	2311891.2
	Heapsort	1052.6	1447.3	13403.2	15610.8	31513.8

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr