Sorting

- Data Structures and Algorithms
- Kyuseok Shim
- SoEECS, SNU.

Sorting Algorithms in General

Sorting: Permuting a sequence of numbers into ascending order

O(n^2) Sorting Algorithms:

- Insertion Sort, Bubble Sort

O($n\log n$) Sorting Algorithms

- Heap Sort: Based on Heap data structure
- Quick Sort: Widely regarded as the "fastest" algorithm
- Merge Sort: *Stable* algorithm; if two elements have the same value, then their relative position after sorting is the same

Is it possible to sort faster than O($n\log n$) time?

- Any comparison-based sorting must make at least O($n\log n$) Comparisons in the worst-case
- Linear-Time sorting algorithms for SMALL integers

Insertion Sort Algorithm

- Consists of N-1 passes
  - For pass p = 1 through N-1, it ensures that the elements in position 0 through p are in sorted order.
  - Use the fact that the elements 0 through p-1 are already known to be in sorted order.

Insertion Sort Algorithm

```
void insertionSort()
{
        int j;
        for (int p = 1; p < n; p++)
        {
                int tmp = a[p];
                for (j = p; j > 0 && tmp < a[j-1]; j--)
                        a[j] = a[j-1];
                a[j] = tmp;
        }
}
```

Insertion Sort Algorithm

Original	34	8	64	51	32	21	Position Moved
After p =1	8	34	64	51	32	21	1
After p = 2	8	34	64	51	32	21	0
After p = 3	8	34	51	64	32	21	1
After p = 4	8	32	34	51	64	21	3
After p = 5	8	21	32	34	51	64	4

Insertion Sort Algorithm

- THEOREM 7.1
  - The average number of inversion in an array of N distinct elements is $N(N-1)/4$.

Insertion Sort Algorithm

- THEOREM 7.1
  - The average number of inversion in an array of N distinct elements is $N(N-1)/4$.
- Proof:
  - For any list L, consider L', the list in reverse order.
  - Consider any pair of two elements in the list (x,y), with $y > x$.
  - In exactly one of L and L', this ordered pair represents an inversion
  - The total number of these pairs in a list L and its reverse L' is $N(N-1)/2$.
  - Thus, an average list has half this amount.

Insertion Sort Algorithm

- THEOREM 7.1
  - Any algorithm that sorts by exchanging adjacent elements requires Omega(N^2)
- Proof:
  - Each swap removes only one inversion so Omega(N^2) swaps are required.

Divide and Conquer

- This is more than just a military strategy, it is also a method of algorithm design that has created such efficient algorithms as Merge Sort, Quick Sort
- In terms or algorithms, this method has three distinct steps:
  - Divide: If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint subsets.
  - Recurse: Use divide and conquer to solve the subproblems associated with the data subsets.
  - Conquer: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem.

Merge Sort

- **Divide:**

  If S has at least two elements, remove all the elements from S and put them into two sequences, $S1$ and $S2$, each containing about half of the elements of S. (i.e. $S1$ contains the first $\lceil n/2 \rceil$ elements and $S2$ contains the remaining $\lfloor n/2 \rfloor$ elements.

- **Recurse:** Recursive sort sequences $S1$ and $S2$.
- **Conquer:** Merge the sorted sequences $S1$ and $S2$ into a unique sorted sequence S.

Merge(A,p,q,r)

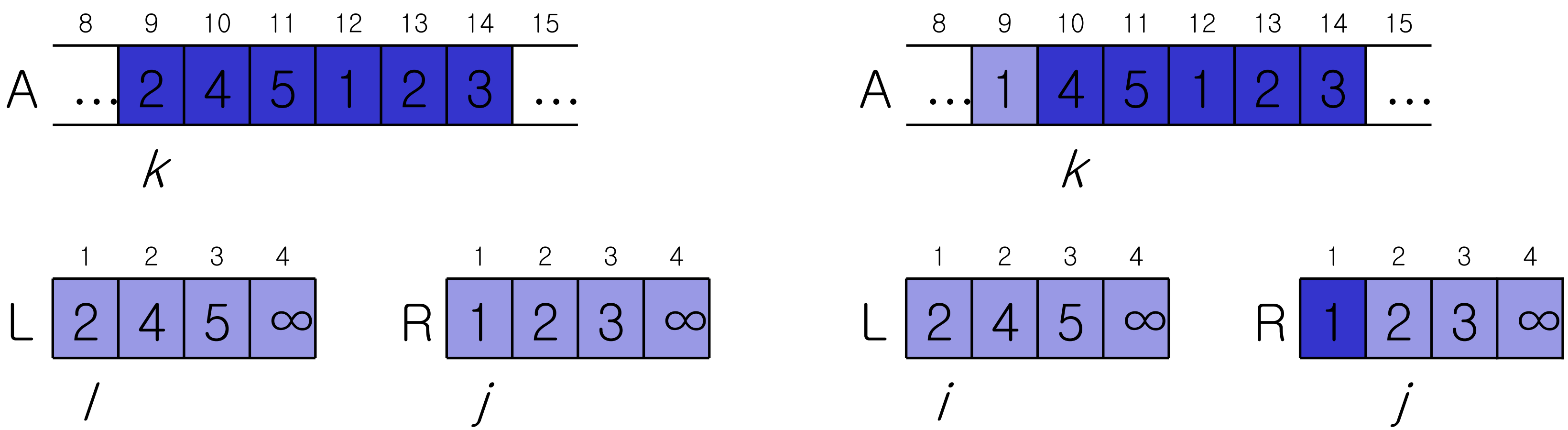
```
n1 <- q – p + 1
n2 <- r – q
create arrays L[1..n1+1] and R[1..n2+1]
for i <- 1 to n1
    do L[i] <- A[p+i-1]
for j <- 1 to n2
    do R[j] <- A[q+j]
L[n1+1] <- infinity
R[n2+1] <- infinity
```

Merge(A,p,q,r)

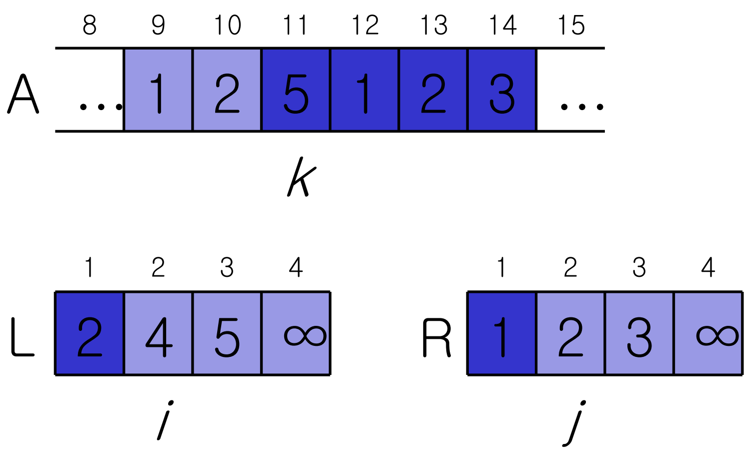
```
i <- 1
j <- 1
for k <- p to r
    do if L[i] <= R[j]
           then A[k] <- L[I]
                i <- i + 1
           else A[k] <- R[j]
                j <- j + 1
```

Loop Invariant:

At the start of each iteration for the for-loop above, the subarray A[p..k-1] contains the k-p smallest elements of L[1..n1+1] and R[1..n2+1], in sorted order. Moreover, L[I] and R[j] are the smallest elements of their arrays that have not been copied back into A.

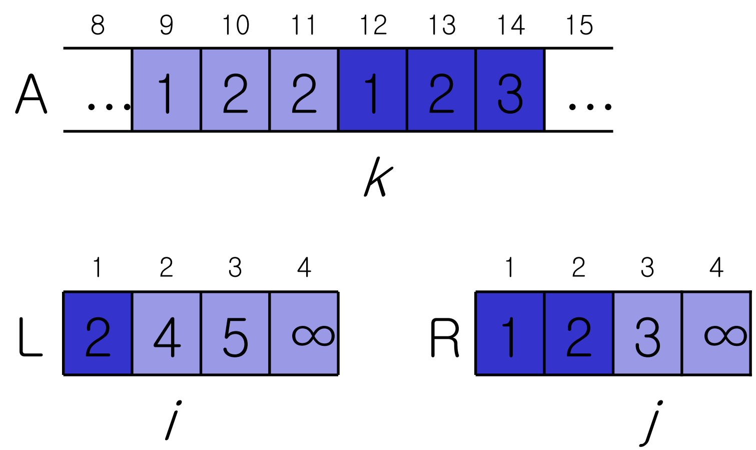

8 9 10 11 12 13 14 15
A … 2 4 5 1 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j
8 9 10 11 12 13 14 15
A … 1 4 5 1 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

(a) (b)



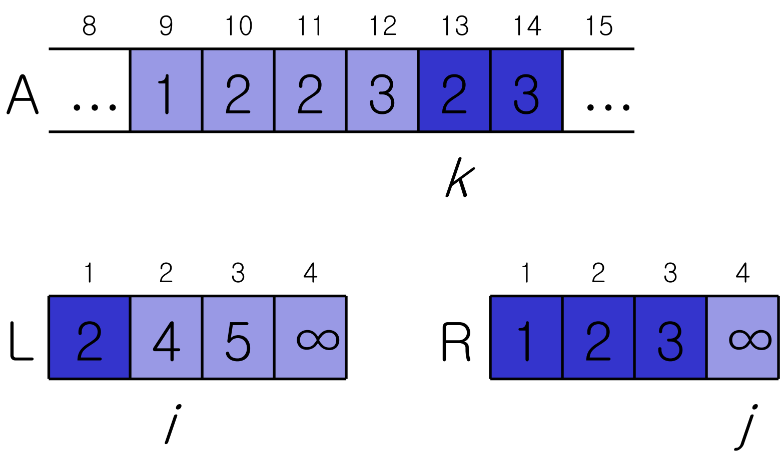
8 9 10 11 12 13 14 15
A … 1 2 5 1 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

(c)

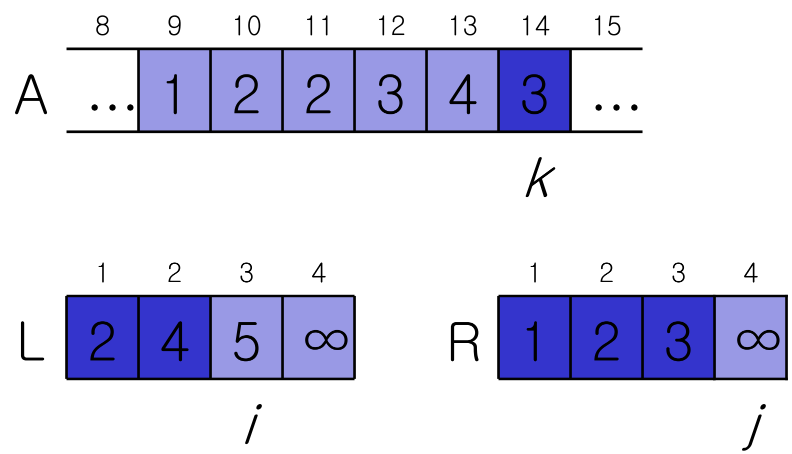


8 9 10 11 12 13 14 15
A … 1 2 2 1 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

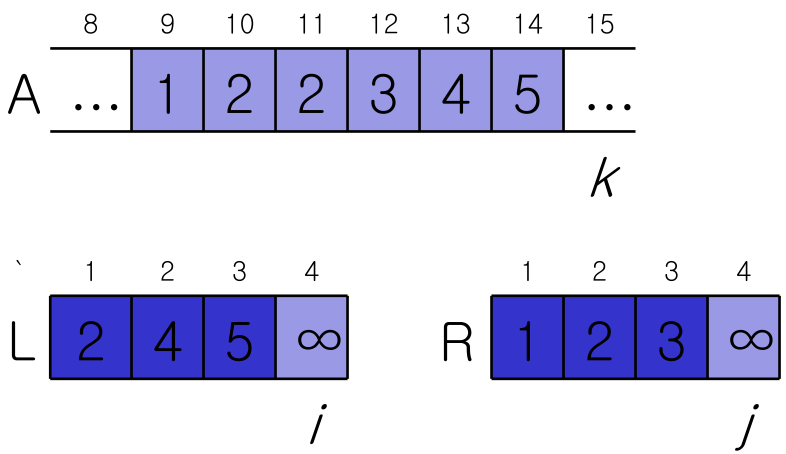
(d)


8 9 10 11 12 13 14 15
A … 1 2 2 3 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

(e)

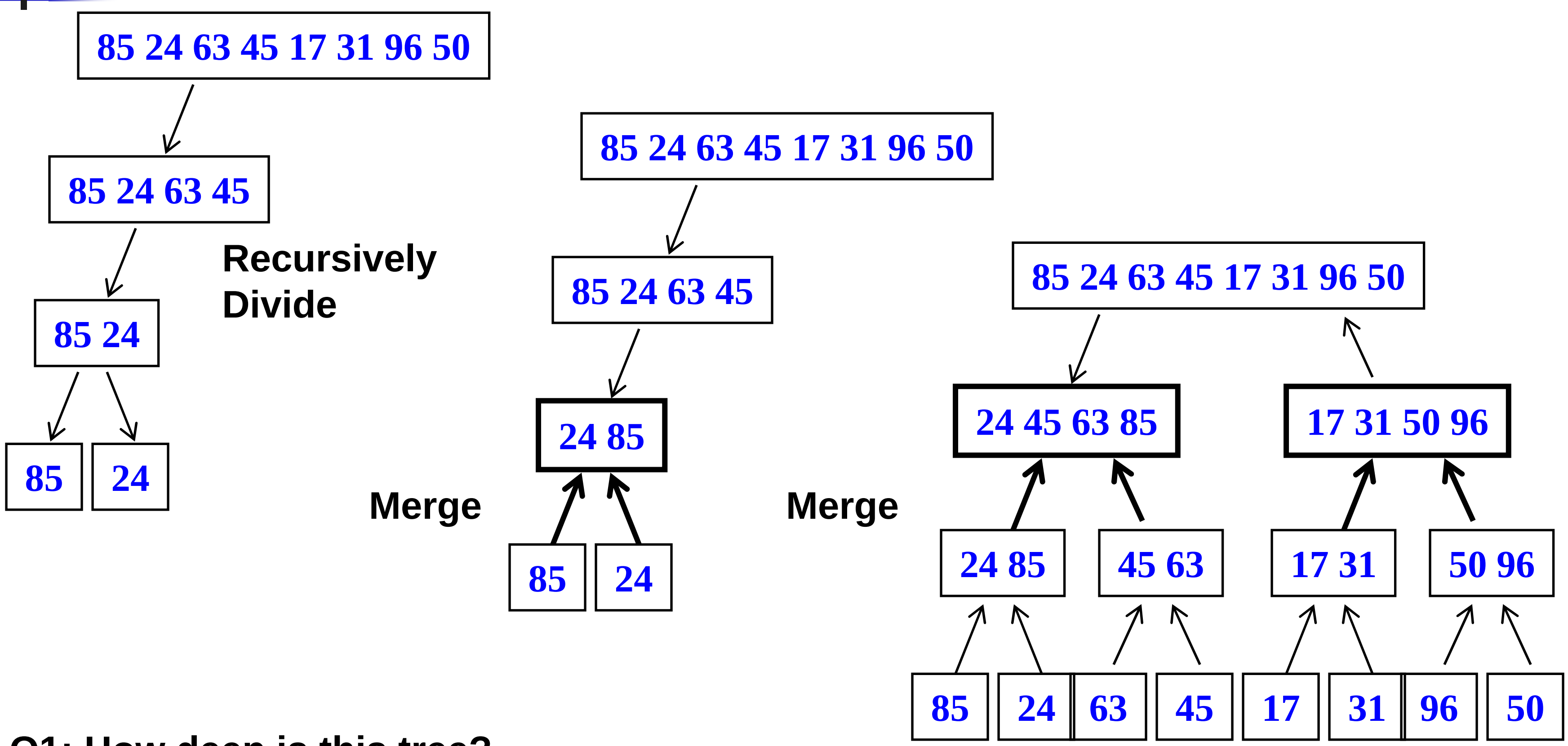

8 9 10 11 12 13 14 15
A … 1 2 2 3 4 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

(f)



(g)

Merge Sort Tree



Q1: How deep is this tree?

Q2: How much memory is needed for merge sort?

Merge Sort

MergeSort(A,p,r)

```
 if p < r
     Then q = floor((p+r)/2)
          MergeSort(A,p,q)
          MergeSort(A,q+1,r)
          Merge(A,p,q,r)
```

- Merge() is the procedure to merge two sorted lists.

Merge Sort Analysis

Recurrence equation :

$$T(1) = 1$$

$$T(n) = 2T(\frac{n}{2}) + n$$

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n$$

$$T(n) = n\log n + n = O(n\log n)$$

Merge Sort

- Merging two half arrays S1, S2 into a full array S requires three pointers, one for S1, another for S2, and the other for S.
- The formal analysis result coincides with the intuitive count of the big Oh, namely, the area taken by the merge sort tree.
- The amount of memory needed for merge sort
  - An extra array

Quick Sort

Given an array `A[1...r]`

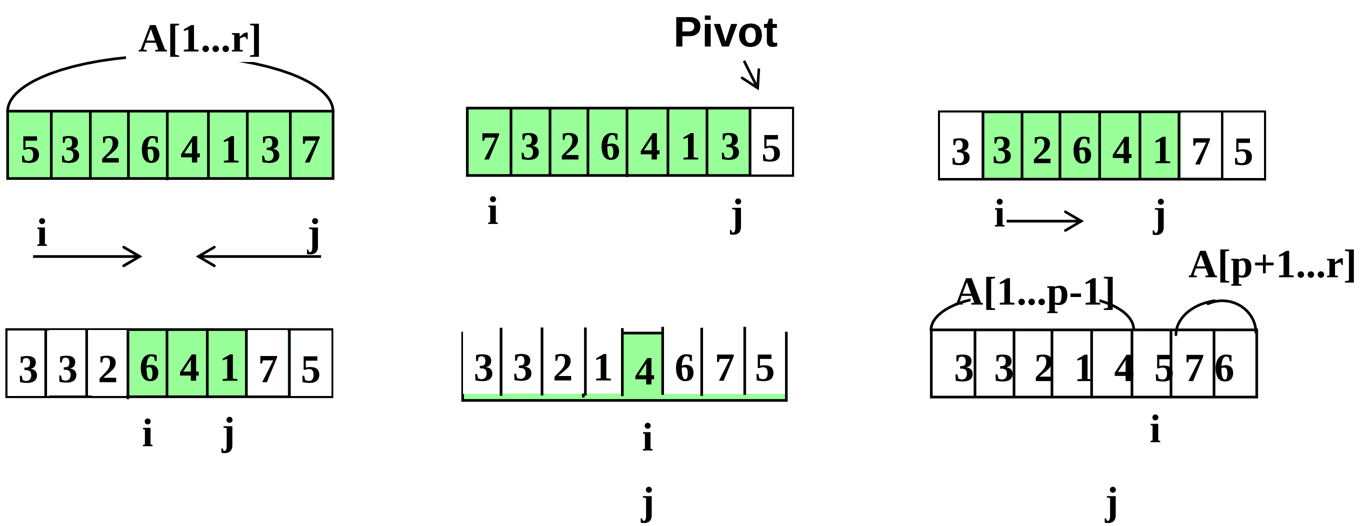
- **Divide:** The array `A[1...r]` is *partition*ed into two nonempty subarrays `A[1...p-1]` and `A[p+1...r]` around the pivot A[p] such that all elements in `A[1...p-1]` <= A[p] <= all elements in `A[p+1...r]`
- **Conquer:** Each of `A[1...p]` and `A[p+1...r]` are sorted by recursive calls to Quick sort

```
Qucksort(A,1,r) {
    if (1 >= r) return;
    p=Partition(A,1,r);
    Quicksort(A,1,p-1);
    Quicksort(A,p+1,r);
}
```

Quick Sort: Partition

Shaded region: not yet partitioned, white region: Partitioned

First, choose the pivot somehow, let's say, it is A[0]=5.
Second, Move the pivot at the end of the array.
Move i to the right until finding the element > the pivot, and
Move j to the left until finding the element < the pivot.



Finally, swap the pivot with the i-th element

Performance of Quick Sort

$$T(n) = T(i) + T(n-i-1) + n(Y(0)) = T(1) = 0)$$

Performance depends on the selection of pivot

worst-case partitioning divide $n-1$ and 1 element

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(1) + \sum_{i=2}^{n} i + n \\ &= O(n^2) \end{aligned}$$

best-case partitioning divide $\frac{n}{2}$ and $\frac{n}{2}$ elements

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n \\ &= 2T(\tfrac{n}{4}) + 2n \\ &= O(n \log n) \end{aligned}$$

Performance of Quick Sort-Cont.

Average-case partitioning:

Assume that the size of a partition is equally likely(that is probability is $\frac{1}{n}$)

The average value of $T(i)$ of $T(n-i-1)$ is $\frac{1}{n}\sum_{j=0}^{n-1} T(j)$

$T(n) = \frac{2}{n}[\sum_{j=0}^{n-1} T(j)] + n$

We already know $T(n) = O(n \log n)$ from the average case analysis of unbalanced binary search tree

This average performance requires good selection of pivot!

- Median-of-Partitioning: take the median of the left, right, and center elements in `A[l...r]`

Average Time Complexity

$$T_{avg}(n) \le cn + \frac{1}{n}\sum_{j=1}^{n}(T_{avg}(j-1) + T_{avg}(n-j)) = cn + \frac{2}{n}\sum_{j=0}^{n-1}T_{avg}(j),\ n \ge 2 \qquad (7.1)$$

We may assume $T_{avg}(0) \le b$ and $T_{avg}(1) \le b$ for some constant b. We shall now show $T_{avg}(n) \le kn\log_e n$ for $n \ge 2$ and $k = 2(b + c)$. The proof is by induction on n.

Induction base: For $n = 2$, Eq. (7.1) yields $T_{avg}(2) \le 2c + 2b \le kn\log_e 2$.

Induction hypothesis: Assume $T_{avg}(n) \le kn\log_e n$ for $1 \le n < m$.

Induction step: From Eq. (7.1) and the induction hypothesis we have

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2}{m}\sum_{j=2}^{m-1}T_{avg}(j) \le cm + \frac{4b}{m} + \frac{2k}{m}\sum_{j=2}^{m-1}j\log_e j \qquad (7.2)$$

Since $j\log_e j$ is an increasing function of j, Eq. (7.2) yields

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2k}{m}\int_2^m x\log_e x\,dx = cm + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2\log_e m}{2} - \frac{m^2}{4}\right]$$

$$= cm + \frac{4b}{m} + km\log_e m - \frac{km}{2} \le km\log_e m,\ \text{for } m \ge 2 \ \square$$

Average Time Complexity

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{n+1} 1/i$$

Sigma 1/i = O(log n)

Thus, T(n) = O(n log n)