COMP319 Algorithms Lecture 5 Merge and Quick Sort

Instructor: Gil-Jin Jang

Merge Sort

Quick Sort

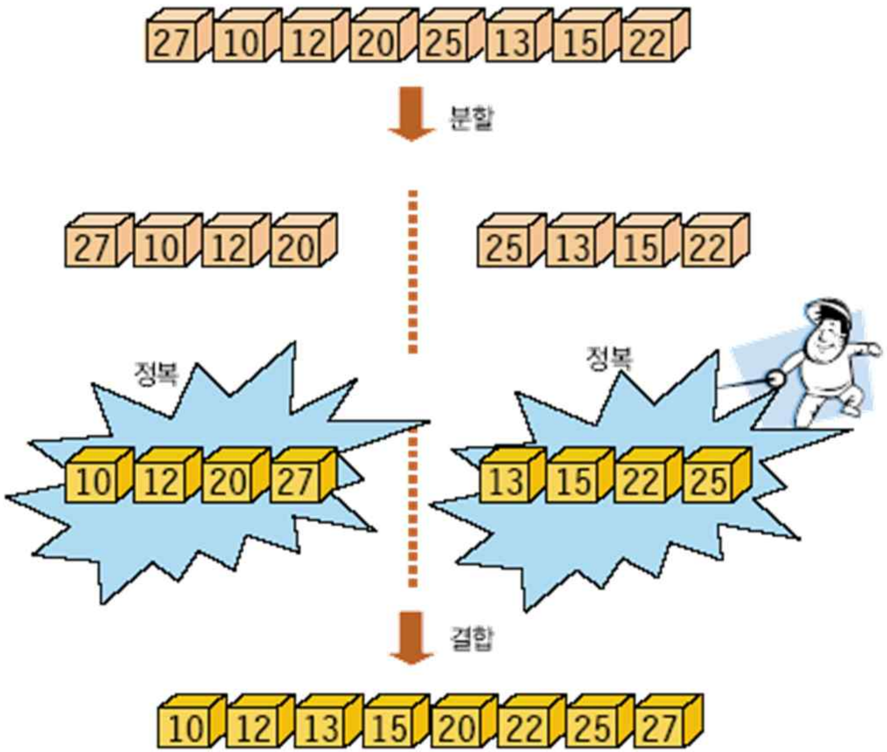
합병 정렬 or
병합 정렬

MERGE SORT

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

합병정렬

- 합병정렬은 리스트를 두개로 나누어, 각각을 정렬한 다음, 다시 하나로 합치는 방법
- 합병정렬은 분할정복기법에 바탕



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

분할정복법

- 분할정복법(divide and conquer)
  - 문제를 작은 2개(또는 n개)의 작은문제로 분리하고 각각을 해결한 다음, 결과를 모아서 원래의 문제를 해결하는 전략이다.
  - 분리된 문제가 아직도 해결하기 어렵다면, 즉 충분히 작지 않다면 분할정복방법을 다시 적용한다.
  - 주로 재귀호출을 이용하여 구현된다.

1. *분할(Divide): 배열을 같은 크기의 2개의 부분 배열로 분할한다.*
2. *정복(Conquer): 부분배열을 정렬한다. 부분배열의 크기가 충분히 작지 않으면 재귀호출을 이용하여 다시 분할정복기법을 적용한다.*
3. *결합(Merge): 정렬된 부분배열을 하나의 배열에 통합한다.*

병합정렬의 작동 예

정렬할 배열이 주어짐

31	3	65	73	8	11	20	29	48	15

배열을 반반으로 나눈다

31	3	65	73	8	11	20	29	48	15	— ①

각각 독립적으로 정렬한다

3	8	31	65	73	11	15	20	29	48	— ②③

병합한다 (정렬완료)

3	8	11	15	20	29	31	48	65	73	— ④

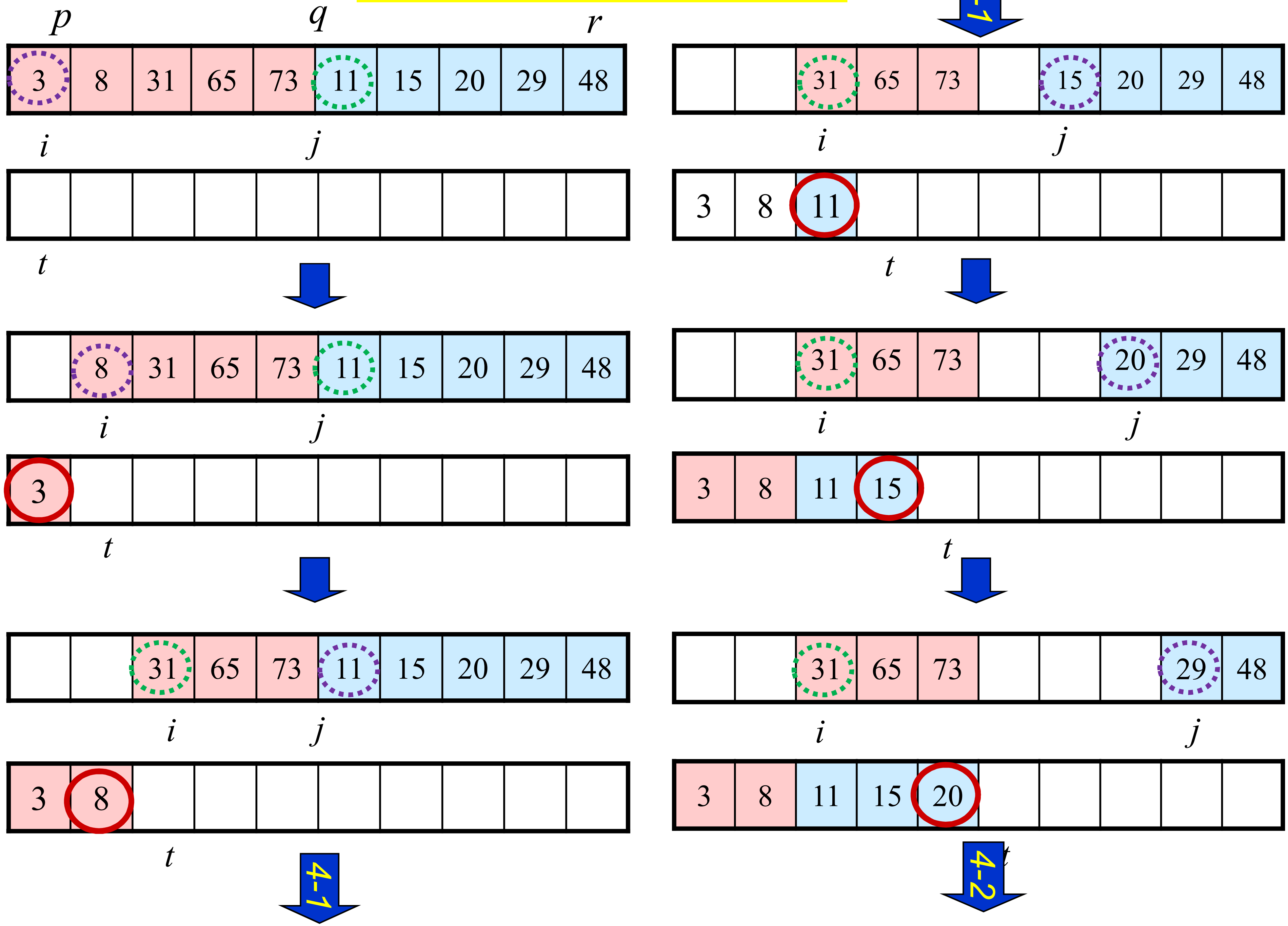
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

병합정렬

mergeSort(A[ ], *p*, *r*)
▷ A[*p* ... *r*]을 정렬한다.
{
    if (*p* < *r*) **then** {
        $q \leftarrow \lfloor (p+q)/2 \rfloor$; ---------------- 1 ▷ p, q의 중간 지점 계산
        mergeSort(A, *p*, *q*); ---------------- 2 ▷ 전반부 정렬
        mergeSort(A, *q*+1, *r*); -------------- 3 ▷ 후반부 정렬
        merge(A, *p*, *q*, *r*); ----------------- 4 ▷ 병합
    }
}

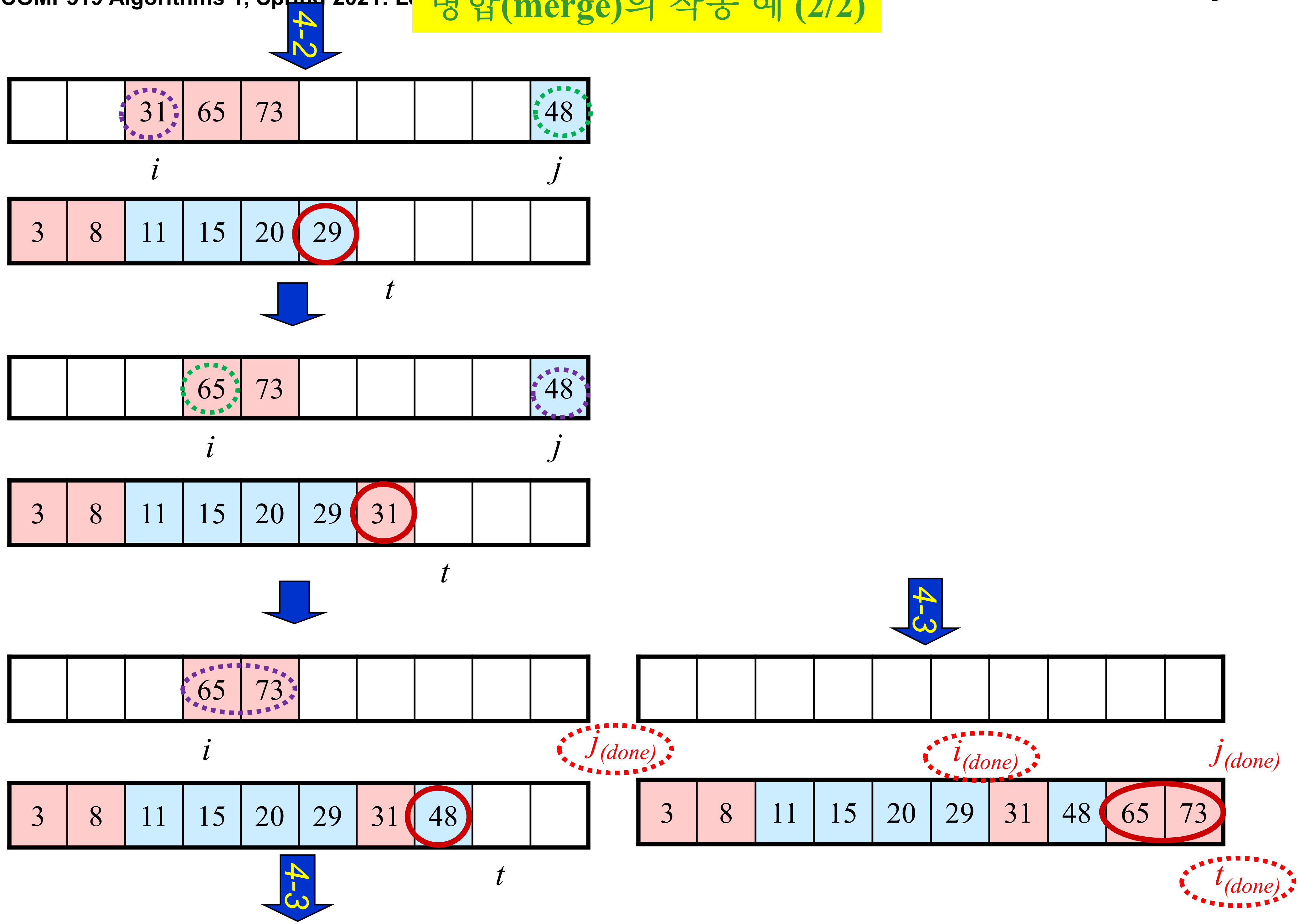
merge(A[ ], *p*, *q*, *r*)
{
    정렬되어 있는 두 배열 A[*p* ... *q*]와 A[*q*+1 ... *r*]을 합쳐
    정렬된 하나의 배열 A[*p* ... *r*]을 만든다.
}

병합(merge)의 작동 예 (1/2)



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

병합(merge)의 작동 예 (2/2)



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Merge

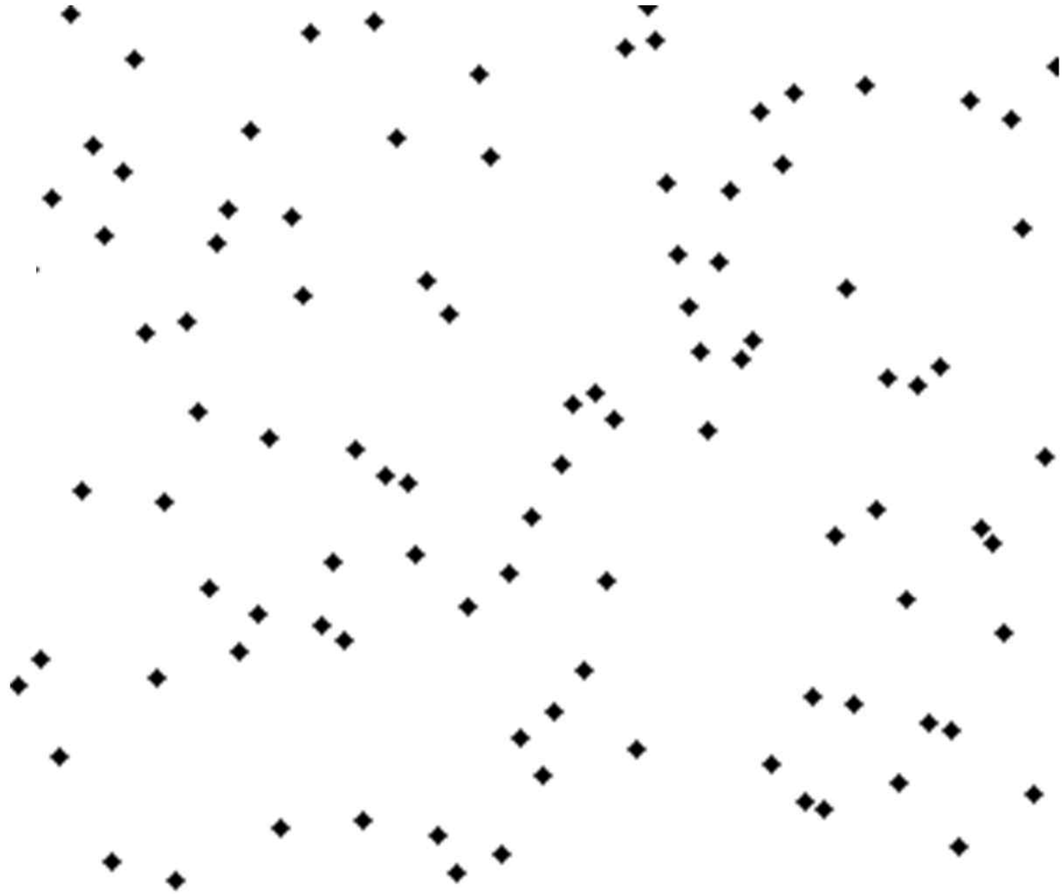
```
merge (A[ ], p, q, r)
▷ A[p ... q]와 A[q+1 ... r]를 병합하여 A[p ... r]을 정렬된 상태로 만든다.
▷ A[p ... q]와 A[q+1 ... r]는 이미 정렬되어 있다.
{
    i ← p; j ← q+1; t ← 1;
    while (i ≤ q and j ≤ r){
        if(A[i] ≤ A[j]) then tmp[t++] ← A[i++];
        else tmp[t++] ← A[j++];
    }

    while(i ≤ q) tmp[t++] ← A[i++];
    while(j ≤ r) tmp[t++] ← A[j++];

    i ← p; t ← 1;
    while(i ≤ r) A[i++] ← tmp[t++];
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Animating Merge Sort



Picture credit: Wikipedia, https://en.wikipedia.org/wiki/Merge_sort

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Merge Sort C code

```c
// i, j: 정렬된 왼쪽/오른쪽 리스트에 대한 인덱스
// k: 정렬될 리스트에 대한 인덱스
void merge(int list[], int tmp[],
              int left, int mid, int right){
  int i, j, k, l;
  i = left; j = mid+1; k = left;

  /* 분할 정렬된 list의 합병 */
  while(i<=mid && j<=right){
    if(list[i]<=list[j]) tmp[k++] = list[i++];
    else tmp[k++] = list[j++];
  }

  // 남아 있는 값들을 일괄 복사
  if(i>mid){
    for(l=j; l<=right; l++) tmp[k++] = list[l];
  }
  // 남아 있는 값들을 일괄 복사
  else{
    for(l=i; l<=mid; l++) tmp[k++] = list[l];
  }

  // 배열 tmp[](임시 배열)의 리스트를
  // 배열 list[]로 재복사
  for(l=left; l<=right; l++){
    list[l] = sorted[l];
  }
}
```

```c
// 합병 정렬
void merge_sort(int list[],
              int tmp[],
              int left, int right){
  int mid;
  if(left<right){
    mid = (left+right)/2;
    // 중간 위치를 계산하여
    // 리스트를 균등 분할(Divide)
    merge_sort(list, left, mid);
    // 앞쪽 부분 리스트 정렬(Conquer)
    merge_sort(list, mid+1, right);
    // 뒤쪽 부분 리스트 정렬(Conquer)
    merge(list, left, mid, right);
    // 정렬된 2개의 부분 배열을
    // 합병하는 과정 (Combine)
  }
}
```

Source: https://gmlwjd9405.github.io/2018/05/08/algorithm-merge-sort.html

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Sorting Algorithm Comparison

- Insertion/Selection/Bubble sort
  - Advantages: easy to understand, by increasing SORTED REGION one item at a time
  - Disadvantages: n^2에 비례하는 연산량
    - o $T(n) = T(n-1)+cn$ ➔ $O(n^2)$ … next class
- Merge sort
  - Advantages: divide-and-conquer 를 이용하여 연산량을 $n \log_2 n$을 비례하게 줄임
    - o $T(n) = 2T(n/2)+cn$ ➔ $O(n \lg n)$ … next class
  - Disadvantages: 정렬하고자 하는 배열과 같은 크기의 추가 메모리 공간($O(n)$ … next class)
- *Quicksort*
- *Heapsort*

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Merge sort 시간 복잡도 분석

ANALYSIS OF MERGE SORT

Merge Sort

```
MergeSort(A, left, right) {
   if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
   }
}
// Merge() takes two SORTED subarrays of A and
// merges them into a single sorted subarray of A
//      (how long should this take?)
// It requires O(n) time, and *does* require extra O(n)
   space
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Merge Sort: Example

- **A = {10, 5, 7, 6, 1, 4, 8, 3, 2, 9};**
  - [10 5 7 6 1 4 8 3 2 9]
  - [[10 5 7 6 1] [4 8 3 2 9]]
  - [[[10 5 7] [6 1]] [[4 8 3] [2 9]]]
  - [[[[10 5] [7]] [(6) (1)]] [[[4 8] [3]] [(2) (9)]]]
  - [[[[(10) (5)] [7]] (1 6)] [[[(4) (8)] [3]] (2 9)]]
  - [[[(5 10) (7)] (1 6)] [[(4 8) (3)] (2 9)]]
  - [[(5 7 10) (1 6)] [(3 4 8) (2 9)]]
  - [(1 5 6 7 10) (2 3 4 8 9)]
  - (1 2 3 4 5 6 7 8 9 10)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analysis of Merge Sort

```
Statement                                          Effort
MergeSort(A, left, right) {                        T(n)
   if (left < right) {                             Θ(1)
      mid = floor((left + right) / 2);             Θ(1)
      MergeSort(A, left, mid);                     T(n/2)
      MergeSort(A, mid+1, right);                  T(n/2)
      Merge(A, left, mid, right);                  Θ(n)
   }
}
```

So $T(n) =$ $\Theta(1)$ when $n = 1$, and

$2T(n/2) + \Theta(n)$ when $n > 1$

This expression is a recurrence

So what is $T(n)$?

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Merge Sort Complexity

```
                              T(n)

              T(n/2)   +   T(n/2)   +   Θ(n)

   T(n/4)+T(n/4)+Θ(n/2)  +  T(n/4)+T(n/4)+Θ(n/2)  +  Θ(n)

T(n/8)+T(n/8)+Θ(n/4)  +  T(n/8)+T(n/8)+Θ(n/4)  …  +  2Θ(n/2)  +  Θ(n)

                              …

T(1)+T(1)+Θ(2)  +  T(1)+T(1)+Θ(2)  +  …  +  4Θ(n/4)+2Θ(n/2)+Θ(n)
```

Depth $= \lfloor \log_2 n \rfloor$,

$$T(n) \in \Theta(n) \cdot \lfloor \log_2 n \rfloor + \Theta(n) = \Theta(n \lfloor \log_2 n \rfloor) = \Theta(n \lg \mathrm{n})$$

*In the textbook notation, 'lg' is used as natural logarithm, $\lg x = \log_e x$

* $\log_2 n = \frac{\log_e n}{\log_e 2} = \Theta(\log_e n) = \Theta(\lg n)$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

합병정렬의 분석

- 비교회수
  - 합병정렬은 크기 *n*인 리스트를 정확히 균등 분배하므로 정확히 logn개의 패스를 가진다.
  - 각 패스에서 리스트의 모든 레코드 *n*개를 비교하여 합병하므로 *n*번의 비교 연산이 수행된다.
  - 따라서 합병정렬은 최적, 평균, 최악의 경우 모두 큰 차이 없이 *n*log*n*번의 비교를 수행하므로 $\Theta(n\log n)$의 복잡도를 가지는 알고리즘이다. 합병정렬은 안정적이며 데이터의 초기 분산 순서에 영향을 덜 받는다.
- 이동회수
  - 배열을 이용하는 합병정렬은 레코드의 이동이 각 패스에서 2*n*번 발생하므로 전체 레코드의 이동은 2*n*log*n*번 발생한다. 이는 레코드의 크기가 큰 경우에는 매우 큰 시간적 낭비를 초래한다.
  - 그러나 레코드를 연결 리스트로 구성하여 합병 정렬할 경우, 링크 인덱스만 변경되므로 데이터의 이동은 무시할 수 있을 정도로 작아진다.
  - 따라서 크기가 큰 레코드를 정렬할 경우, 연결 리스트를 이용하는 합병정렬은 퀵정렬을 포함한 다른 어떤 정렬 방법보다 매우 효율적이다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Sorting Algorithm Comparison

- Insertion/Selection/Bubble sort
  - Advantages: using less extra memory
  - Disadvantages: $T(n) = T(n\text{-}1)+cn \Rightarrow O(n\^{}2)$
- Merge sort
  - Advantages: $T(n) = 2T(n/2)+cn \Rightarrow O(n \lg n)$
  - Disadvantages: extra memory of $O(n)$
- *Quicksort*
- *Heapsort*

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

효율성 비교

	Worst Case	**Average Case**
Selection Sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Mergesort	$\boldsymbol{n\log n}$	$\boldsymbol{n\log n}$
Quicksort	n^2	$n\log n$
Heapsort	$n\log n$	$n\log n$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

퀵 정렬

QUICK SORT

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Sorting Algorithm Comparison

- Insertion/Selection/Bubble sort
  - Advantages: using less extra memory
  - Disadvantages: $T(n) = T(n\text{-}1)+cn \rightarrow O(n\text{^}2)$
- Merge sort
  - Advantages: $T(n) = 2T(n/2)+cn \rightarrow O(n \lg n)$
  - Disadvantages: extra memory of $O(n)$
- *Quicksort*
  - $O(n \lg n)$ without extra memory
- *Heapsort*

Review: Insertion Sort

```
/* Pseudo code: not an actual code,
   index starts from 1 */
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

Review: Merge Sort

```
MergeSort(A, left, right) {
   if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
   }
}
// Merge() takes two SORTED subarrays of A and
// merges them into a single sorted subarray of A
//     (how long should this take?)
// It requires O(n) time, and *does* require extra O(n)
   space
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Quicksort Pseudo Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray $\leq$ all values in second
  - Returns the index of the "pivot" element separating the two subarrays
- *How do you suppose we implement this function?*

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Partition In Words

- Partition(A, p, r):
  - Select an element to act as the “pivot” (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

Partition Code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

`Illustrate on`

`A = {5, 3, 2, 6, 4, 1, 3, 7};`

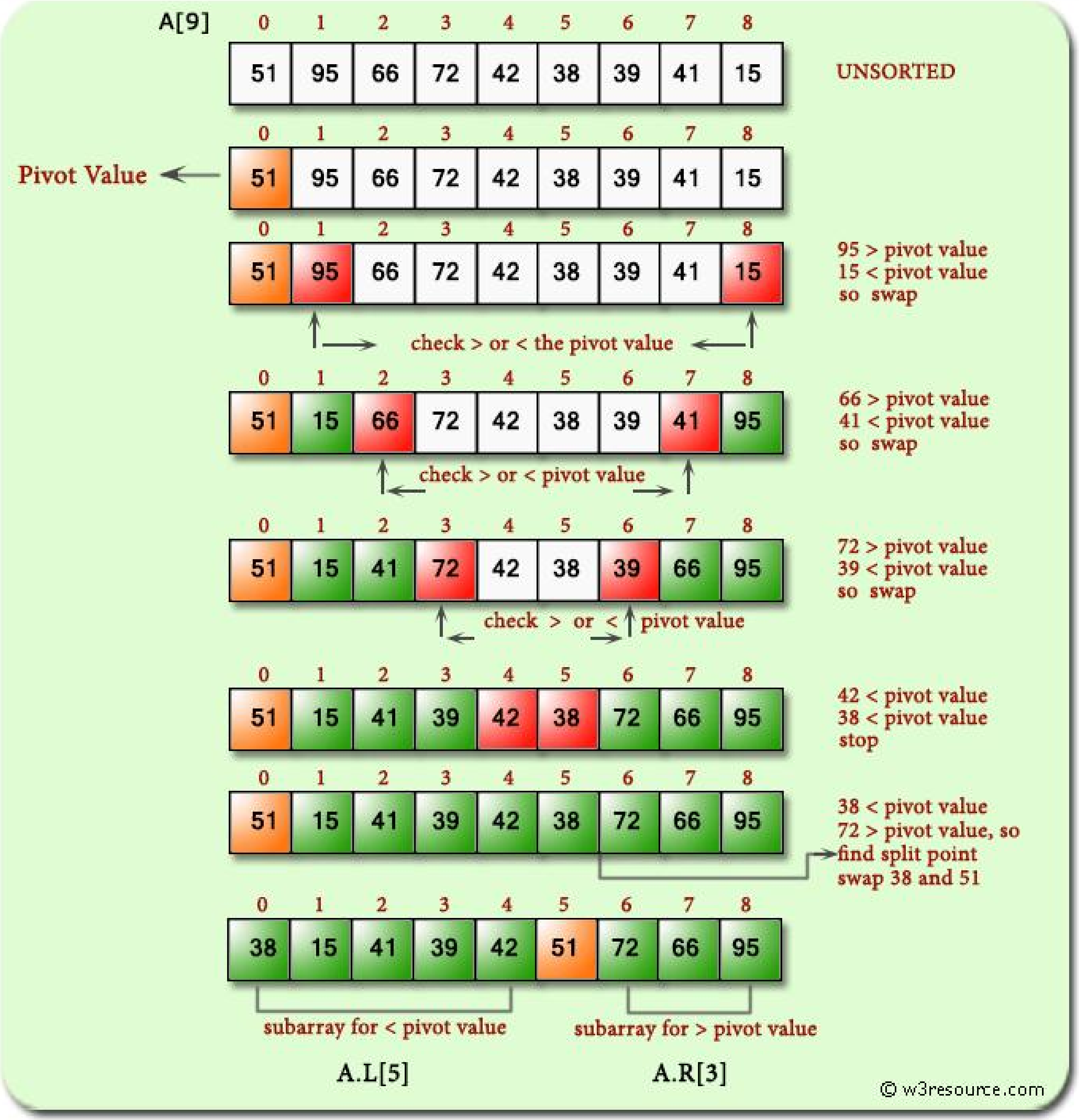
What is the running time of **`partition()`**?

`partition()` *runs in O(n) time*

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

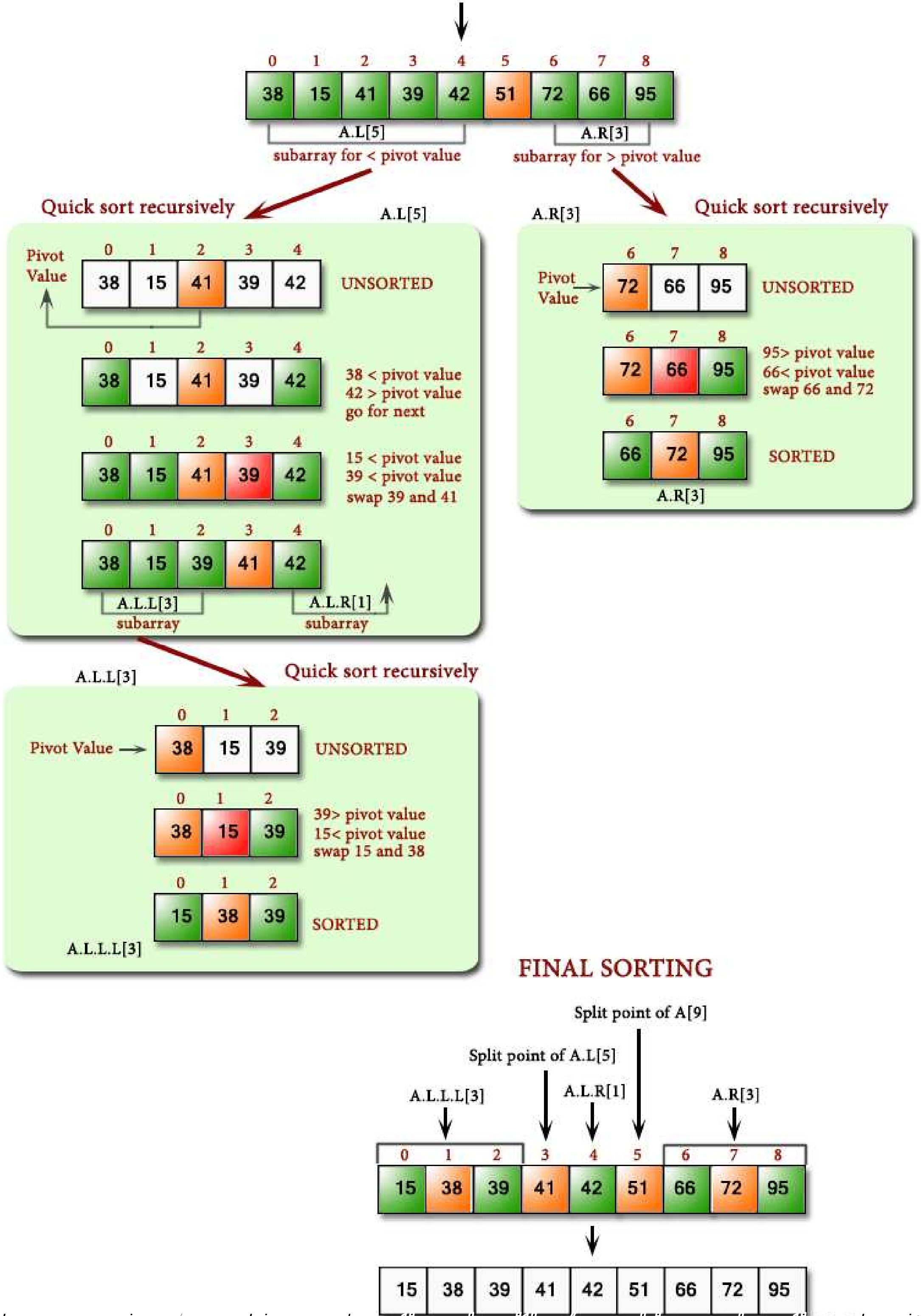
Quick Sort

Partition



https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-9.php

Recurrence



https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-9.php

Quicksort properties

- Sorts in place (i.e. requiring constant extra memory)
- Sorts $O(n \log_2 n)$ in the average case
- Sorts $O(n^2)$ in the worst case
- Another *divide-and-conquer* algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort

Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is perfectly balanced
- *Which is more likely?*
  - The latter, by far, except...
- *Will any particular input elicit the worst case?*
  - Yes: Already-sorted input

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort

- In the worst case:

  $T(1) = \Theta(1)$

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

- Works out to

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

- In the best case:

  $T(n) = 2T(n/2) + \Theta(n)$

- What does this work out to?

  $T(n) = \Theta(n \log_2 n)$

Improving Quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input
- Book discusses two solutions:
  - Randomize the input array, OR
  - *Pick a random pivot element*
- *How will these solve the problem?*
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Quicksort: Radom Pick of Pivots

```
Quicksort(A, left, right) {
    if (left < right) {
                // choose a random integer in [p, r]
        pivot = random(left, right);
        // swap the leftmost and chosen pivot in array A
        swap(A, left, pivot);

        q = Partition(A, left, right);
        Quicksort(A, left, q);
        Quicksort(A, q+1, right);
    }
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

- Assuming random input, average-case running time is much closer to O(n lg n) than O(n^2)

- First, a more intuitive explanation/example:
  - Suppose that partition() always produces a 9-to-1 split. This looks quite unbalanced!
  - The recurrence is thus:

    $T(n) = T(9n/10) + T(n/10) + n$
  - *How deep will the recursion go?*
    - 9n/10 → 9n/10 * 9/10 → … → 1

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!

Analyzing Quicksort: Average Case

- Intuitively, the O(n) cost of a bad split (or 2 or 3 bad splits) can be absorbed into the O(n) cost of each good split
- Thus running time of alternating bad and good splits is still O(n lg n), with slightly higher constants
- How can we be more rigorous?

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

- For simplicity, assume:
  - All inputs distinct (no repeats)
  - Slightly different **`partition()`** procedure
    - partition around a random element, which is not included in subarrays
    - all splits (0:n-1, 1:n-2, 2:n-3, … , n-1:0) equally likely
- *What is the probability of a particular split happening?*
  - Answer: 1/n

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

- So partition generates splits
  (0:n-1, 1:n-2, 2:n-3, … , n-2:1, n-1:0)
  each with probability 1/n
- If T(n) is the expected running time,

$$
\begin{aligned}
T(n) &= \frac{1}{n}\sum_{k=0}^{n-1}\left[T(k)+T(n-1-k)\right]+\Theta(n) \\
&= \frac{2}{n}\sum_{k=0}^{n-1}T(k)+\Theta(n)
\end{aligned}
$$

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - *What's the answer?*
  - Assume that the inductive hypothesis holds
    - *What's the inductive hypothesis?*
  - Substitute it in for some value < n
    - *What value?*
  - Prove that it follows for n

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \leq an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
    - The value k in the recurrence
  - Prove that it follows for n
    - Grind through it…

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$

The recurrence to be solved

$$\le \frac{2}{n}\sum_{k=0}^{n-1} (ak\lg k + b) + \Theta(n)$$

Plug in inductive hypothesis

$$\le \frac{2}{n}\left[b + \sum_{k=1}^{n-1} (ak\lg k + b)\right] + \Theta(n)$$

Expand out the k=0 case

$$= \frac{2}{n}\sum_{k=1}^{n-1} (ak\lg k + b) + \frac{2b}{n} + \Theta(n)$$

2b/n is just a constant, so fold it into Θ(n)

$$= \frac{2}{n}\sum_{k=1}^{n-1} (ak\lg k + b) + \Theta(n)$$

Note: leaving the same recurrence as the book

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1}(ak\lg k + b) + \Theta(n)$$

The recurrence to be solved

$$= \frac{2}{n}\sum_{k=1}^{n-1} ak\lg k + \frac{2}{n}\sum_{k=1}^{n-1} b + \Theta(n)$$

Distribute the summation

$$= \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + \frac{2b}{n}(n-1) + \Theta(n)$$

Evaluate the summation:
b+b+…+b = b (n-1)

$$\le \frac{2a}{n}\boxed{\sum_{k=1}^{n-1} k\lg k} + 2b + \Theta(n)$$

Since n-1<n, 2b(n-1)/n < 2b

This summation gets its own set of slides later

Analyzing Quicksort: Average Case

$$T(n) \le \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + 2b + \Theta(n)$$

The recurrence to be solved

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + 2b + \Theta(n)$$

We'll prove this later

$$= an \lg n - \frac{a}{4}n + 2b + \Theta(n)$$

Distribute the (2a/n) term

$$= an \lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right)$$

Remember, our goal is to get $T(n) \le an \lg n + b$

$$\le an \lg n + b$$

Pick a large enough that an/4 dominates $\Theta(n)+b$

Analyzing Quicksort: Average Case

- So $T(n) \leq an \lg n + b$ for certain a and b
  - Thus the induction holds
  - Thus $T(n) = O(n \lg n)$
  - Thus quicksort runs in $O(n \lg n)$ time on average (phew!)

- Oh yeah, the summation…

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k$$

Split the summation for a tighter bound

$$\le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg n$$

The $\lg k$ in the second term is bounded by $\lg n$

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move the $\lg n$ outside the summation

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

$$\le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg(n/2) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The* lg *k in the first term is bounded by* lg *n/2

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k(\lg n - 1) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

lg *n/2* = lg *n* - 1

$$= (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move (*lg *n* - 1*) outside the summation

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

$$= \lg n \sum_{k=1}^{\lceil n/2 \rceil - 1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Distribute the (lg n - 1)

$$= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summations overlap in range; combine them

$$= \lg n \left(\frac{(n-1)(n)}{2} \right) - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The Guassian series

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \leq \left(\frac{(n-1)(n)}{2}\right)\lg n - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summation bound so far

$$\leq \frac{1}{2}[n(n-1)]\lg n - \sum_{k=1}^{n/2-1} k$$

Rearrange first term, place upper bound on second

$$\leq \frac{1}{2}[n(n-1)]\lg n - \frac{1}{2}\left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right)$$

X Guassian series

$$\leq \frac{1}{2}\left(n^2 \lg n - n \lg n\right) - \frac{1}{8}n^2 + \frac{n}{4}$$

Multiply it all out

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}\left(n^2 \lg n - n \lg n\right) - \frac{1}{8}n^2 + \frac{n}{4}$$

$$\le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \text{ when } n \ge 2$$

Done!!!

효율성 비교

	Worst Case	**Average Case**
Selection Sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Mergesort	$n\log n$	$n\log n$
Quicksort	$\boldsymbol{n^2}$	$\boldsymbol{n\log n}$
Heapsort	$n\log n$	$n\log n$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Merge and quick sort

END OF LECTURE 5