CSE 230: Data Structures

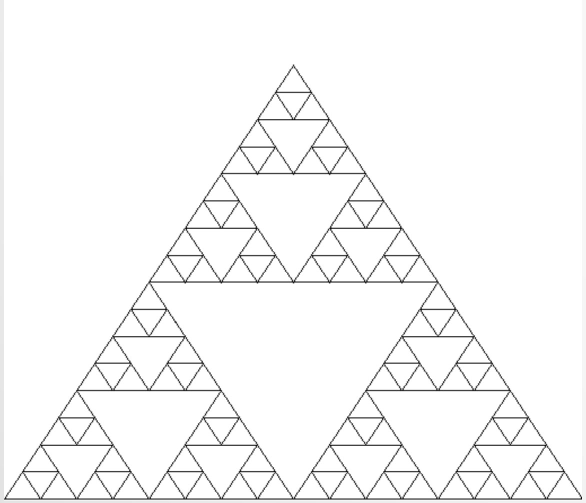
Lecture 3: Recursion and Sorting

Dr. Vidhya Balasubramanian

Amrita School of Engineering
Amrita Vishwa Vidyapeetham

Recursion

- Concept of defining a function that calls itself as a sub-routine
  - Allows us to take advantage of the repeated structure in many problems
  - e.g finding the factorial of a number



http://introcs.cs.princeton.edu/java/23recursion/images/sierpinski5.png

http://www.spektyr.com/Gallery/Recursion.jpg

Linear Recursion

- Function is defined so that it makes atmost one recursive call at each time it is invoked
- Useful when an algorithmic problem
  - Can be viewed in terms of a first or last element, plus a remaining set with same structure

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Src:mitpress.mit.edu

Algorithm LinearSum(A,n):
    Input: Integer array A and element n
    Output: Sum of first n elements of A
    if n=1 **then**
        return A[0]
    else
        return LinearSum(A,n-1)+A[n-1]

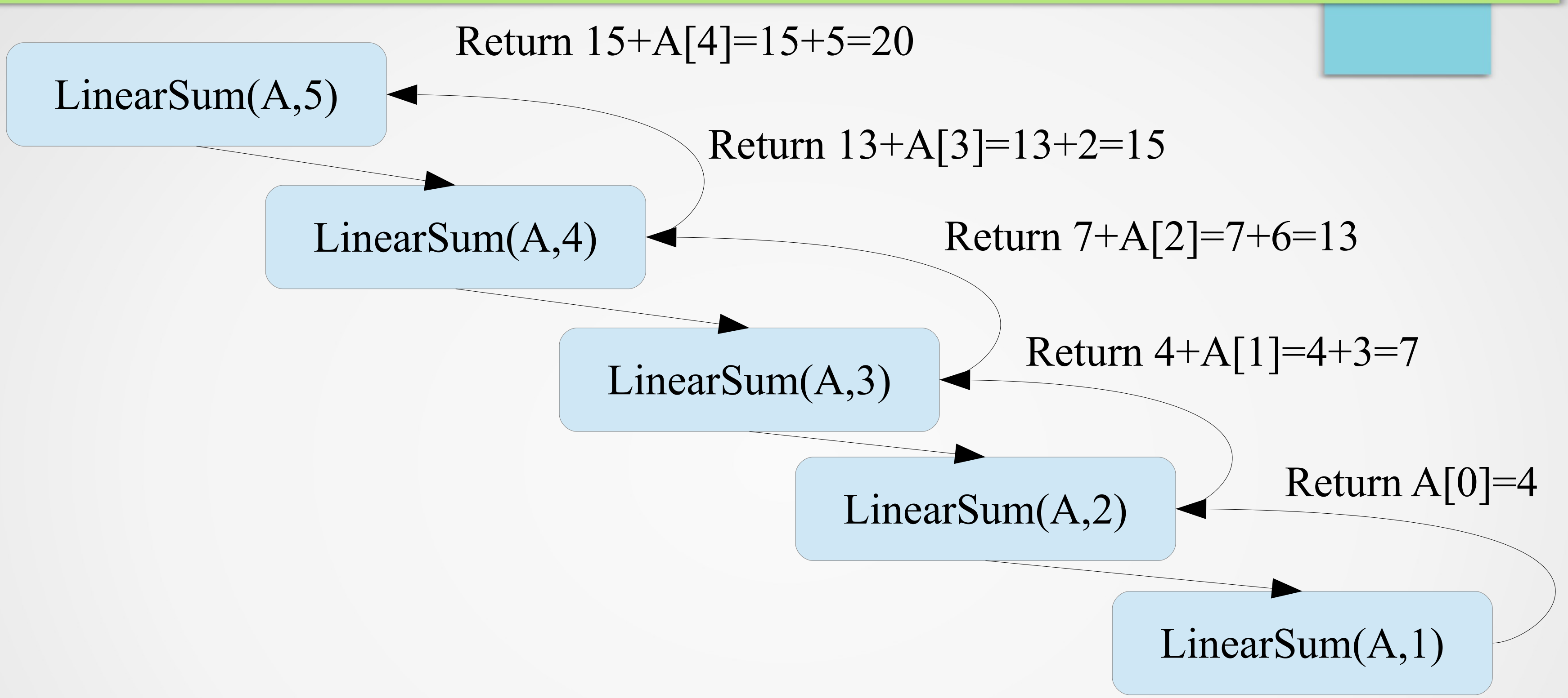
Linear Recursion

- Algorithm using linear recursion uses the following
  - Test for base cases
    - Base cases defined so that every possible chain of recursive call reaches base case
    - Helps in termination
  - Recurse
    - May decide on one of multiple recursive calls to make
    - Recursive calls must progress to base case

Analyzing Recursive Algorithms

- Use visual tool – recursion trace
  - Contains box for each recursive call
    - Contains parameters of the call
  - Links showing the return value
- Use recurrence relation
  - Mathematical formulation to capture the recursion process

Analyzing Recursive Algorithms



- Example LinearSum
  - Running time is O(n)
  - Space Complexity: O(n)

Stack Trace

- Example stack trace

LinearSum(A,1)	Return A[0]=4
LinearSum(A,1)+A[1]	Return 4+A[1]=4+3=7
LinearSum(A,2)+A[2]	Return 7+A[2]=7+6=13
LinearSum(A,3)+A[3]	
LinearSum(A,4)+A[4]	

Problem 1

- Reverse an array using linear recursion
- Solution
  - **Algorithm** ReverseArray(*A*,*i*,*n*):

    Input: Integer array *A* and integers *i*,*n*

    Output: Reversal of *n* integers in *A* starting from *i*

    if n<=1 **then**

        return

    else

        Swap *A*[i] and *A*[*i*+*n*-1]

        Call ReverseArray(*A*,*i*+1,*n*-2)

    return

Problem 2:

- Computing Powers via Linear Recursion

$$power(x,n)=\begin{Bmatrix} 1 \; if \; n=0 \\ \text{x.power}(x,n-1) \; otherwise \end{Bmatrix}$$

$$power(x,n)=\begin{Bmatrix} 1 \; if \; n=0 \\ \text{x.power}(x,(n-1)/2)^2 \; if \; n>0 \; is \; odd \\ \text{power}(x,n/2)^2 \; if \; n>0 \; is \; even \end{Bmatrix}$$

Problem 3

- Describe a linear recursive algorithm for finding the minimum element in an n-element array
- Write a function using recursion to print numbers from n to 0.
- Write a recursive function that computes and returns the sum of all elements in an array, where the array and its size are given as parameters.

Higher-Order Recursion

- Uses more than one recursion call
  - e.g 3-way merge sort
- Binary recursion
  - Two recursion calls
  - e.g BinarySum
    - **Algorithm** BinarySum(A,i,n):

      Input: Integer array A and integers i, n

      Output: Sum of first n elements of A starting at index i

      if n=1 then

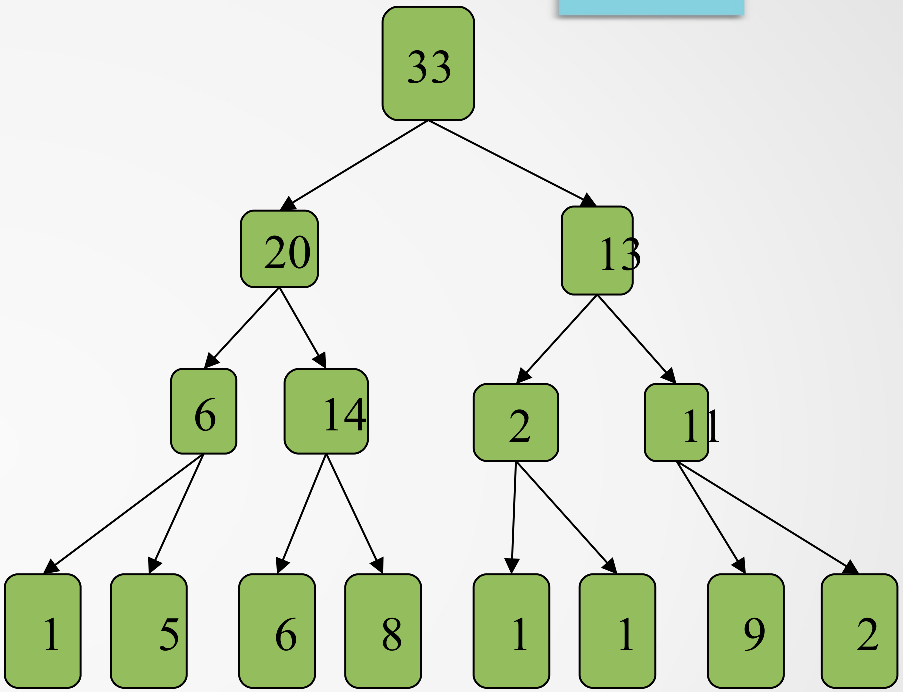
          return A[i]

      else

          return BinarySum(A,i,$\lceil n/2 \rceil$)+BinarySum(A,i+$\lceil n/2 \rceil$,$\lfloor n/2 \rfloor$)

Analysis

- Recursion trace is a tree
- Depth of recursion
  - O(logn)
  - Lesser additional space needed
- Running time
  - O(n)
  - Have to visit every element in the array



Problem

- Generate the kth Fibonacci number using Binary Recursion
- Solution (This method not recommended !! Exponential complexity)
  - **Algorithm** BinaryFib(*k*):

    Input: k

    Output: kth Fibonacci Number

    if k<=1 **then**

        return k

    else

        return BinaryFib(k-1)+BinaryFib(k-2)

Problem

- Describe a binary recursive method for searching an element x in an n-element unsorted array A.
  - Compute the running time and space complexity of your algorithm

Sorting

- Given a set of n numbers that may be in any order, the goal is to output the numbers in sorted order
  - A set of elements S are sorted in ascending order when $S_i < S_{i+1}$ for all $1 <= i < n$
  - A set of elements S are sorted in descending order when $S_i > S_{i+1}$ for all $1 <= i < n$
- Can sort just the key or entire records

Importance of Sorting

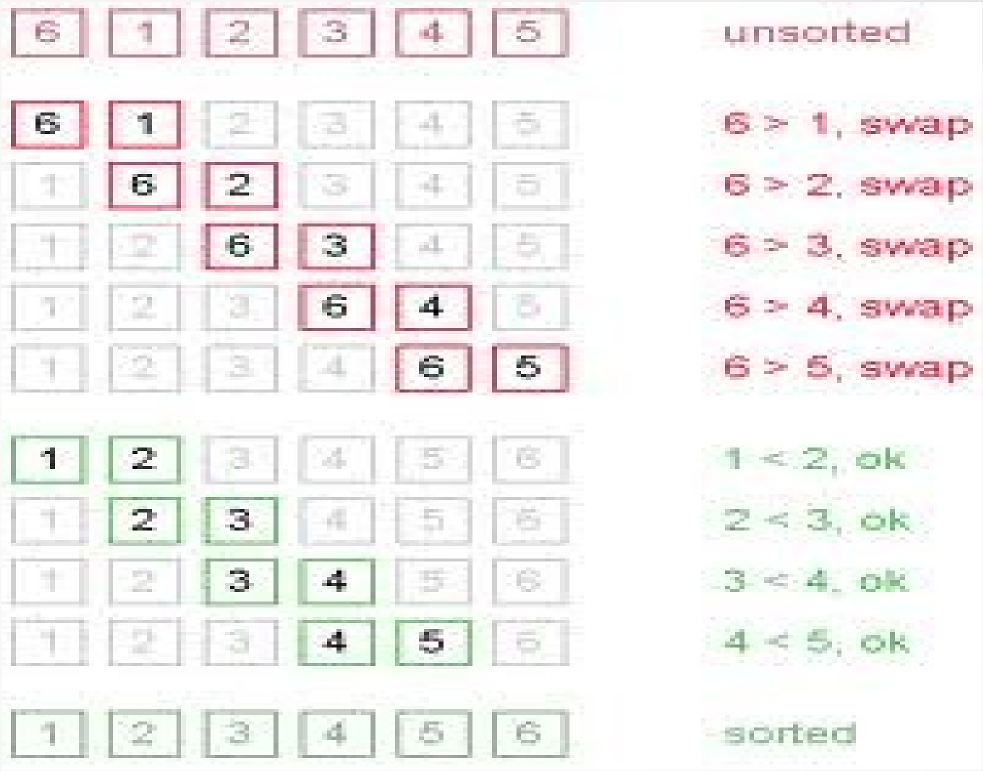
- Basic block around which many other algorithms are built on
- Interesting ideas in design of algorithms appear in the context of sorting
  - Divide and conquer, data structures, randomized algorithms
- Remains the most ubiquitous combinatorial algorithmic problem in practice
- Loads of study on this problem

Applications of Sorting

- Searching: Whether linear or binary, it is easier to search for an element if the list is sorted
- Closest Pair
  - Given a set of n numbers, find the pair of numbers that have the smallest difference between them
  - Closest pair lie next to each other somewhere in the sorted order
- Finding duplicates among n elements in a list
  - In sorted list, it is easy to find this by scanning adjacent elements
- Frequency Distribution
  - Find the number of occurrences of different elements in a list
  - Identical items together in a sorted list: Just scan list once
- Selection
  - Find the kth largest element in a list

Bubble Sort

- A basic sorting algorithm where corresponding elements are checked and swapped
  - The smallest elements bubble up the way
- BUBBLESORT(A)
  - ***for*** i = 1 ***to*** A.length-1
  - ***for*** j = A:length ***downto*** i – 1
  - ***if*** A[j] < A[j-1]
  - exchange A[j] with A[j -1]



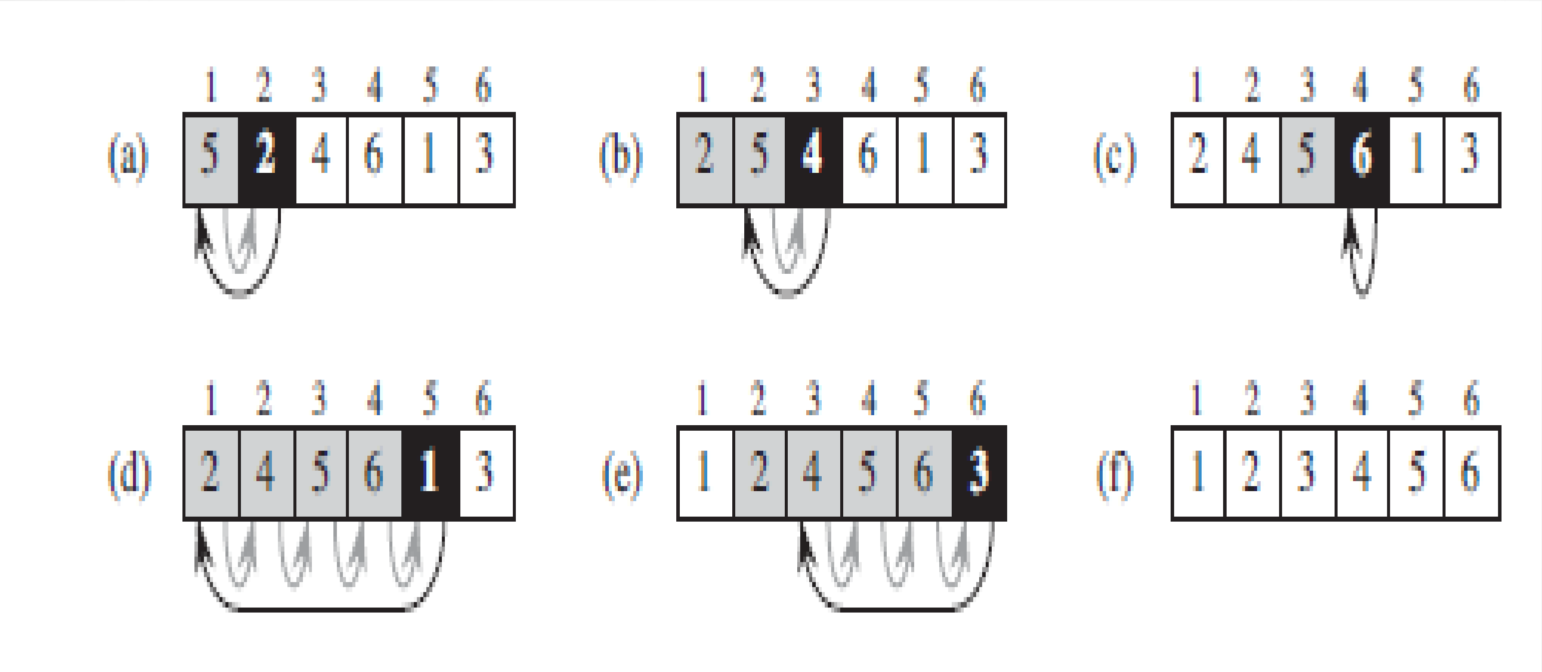
http://t2.gstatic.com/images?q=tbn:ANd9GcRnvLV-rVerCKzEZ7sN7WwBIUL09M_FqIkgX52w5mSLElDgtFVRX-FgHtSsWg

Bubble Sort: Analysis

- Best Case
  - O(n)
  - List already sorted, hence zero swaps after first iteration
- Worst case
  - $O(n^2)$
- On average
  - (n-1)*n/2 comparisons
  - $O(n^2)$

Insertion Sort

- At each iteration an element is removed from the list and inserted into the right place
- An any iteration the first i items are in place



Src: CLR – Ch2- Pg22

Insertion Sort

- INSERTION-SORT(A)

  1. ***for*** j = 2 ***to*** A.length
  2. key = A[j] // Insert A[j] into the sorted sequence A[1 : : j-1]
  3. i = j-1
  4. ***while*** i > 0 ***and*** A[i] > key
  5. A[i+1] = A[i]

  6 i = i-1

  7 A[i-1] = key

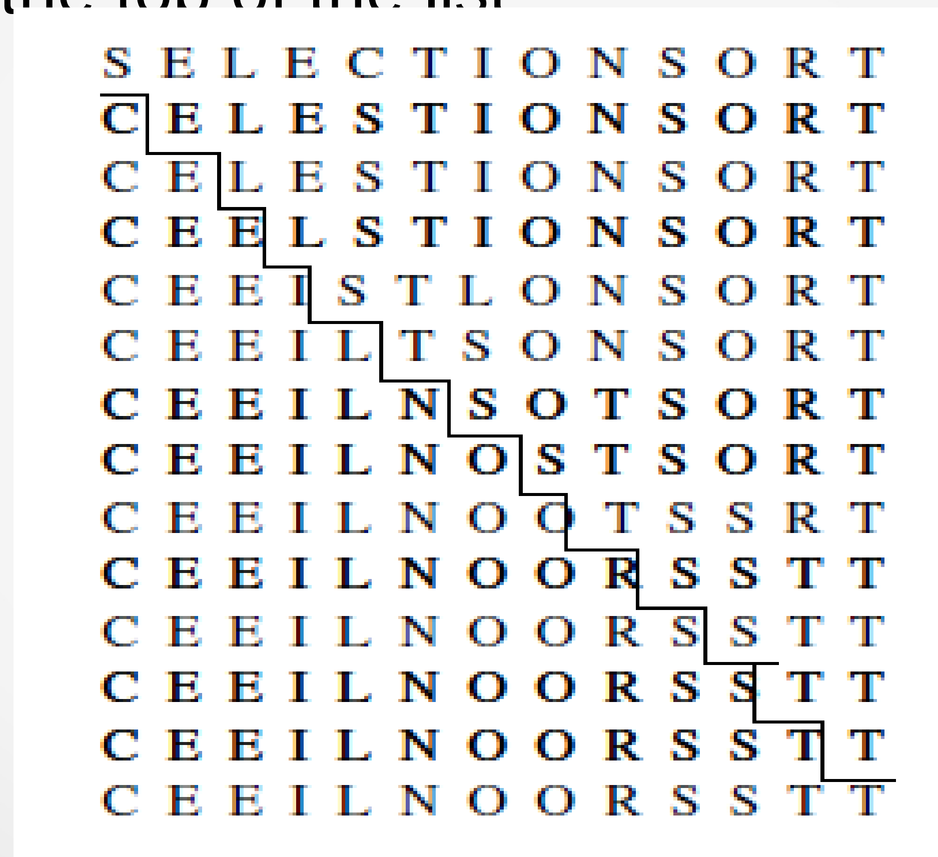
Src: CLR – Ch2

Insertion Sort: Analysis

- At each iteration
  - Element compared and/or swapped with atmost i elements
    - i varies from 1 to n
- n such elements inserted
- Average case and worst case- $O(n^2)$
  - Worst case when list sorted in reverse order
- Best Case – O(n)
  - When list is already sorted
- No swaps needed

Selection Sort

- At each iteration the minimum element is chosen and inserted in the top of the list



Src: Steven S. Skiena, “The Algorithm Design Manual”, Second Edition

Selection Sort

- selection_sort(int s[], int n){

```
int i,j; /* counters */
for (i=0; i<n; i++) {
   min=i;     /* index of minimum */
   for (j=i+1; j<n; j++)
            if (s[j] < s[min]) min=j;
   swap(s[i],s[min]);
}
}
```

Analysis of Selection Sort

- Complexity depends on cost of finding minimum element in remaining list
- Best Case – $O(n^2)$
  - Almost sorted still requires cost of finding min
- Average Case – $O(n^2)$
  - Cost of finding minimum element is i per iteration = $n(n-1)/2$
- Worst Case – $O(n^2)$
  - When list is in reverse sorted order
- The minimum is always at the end

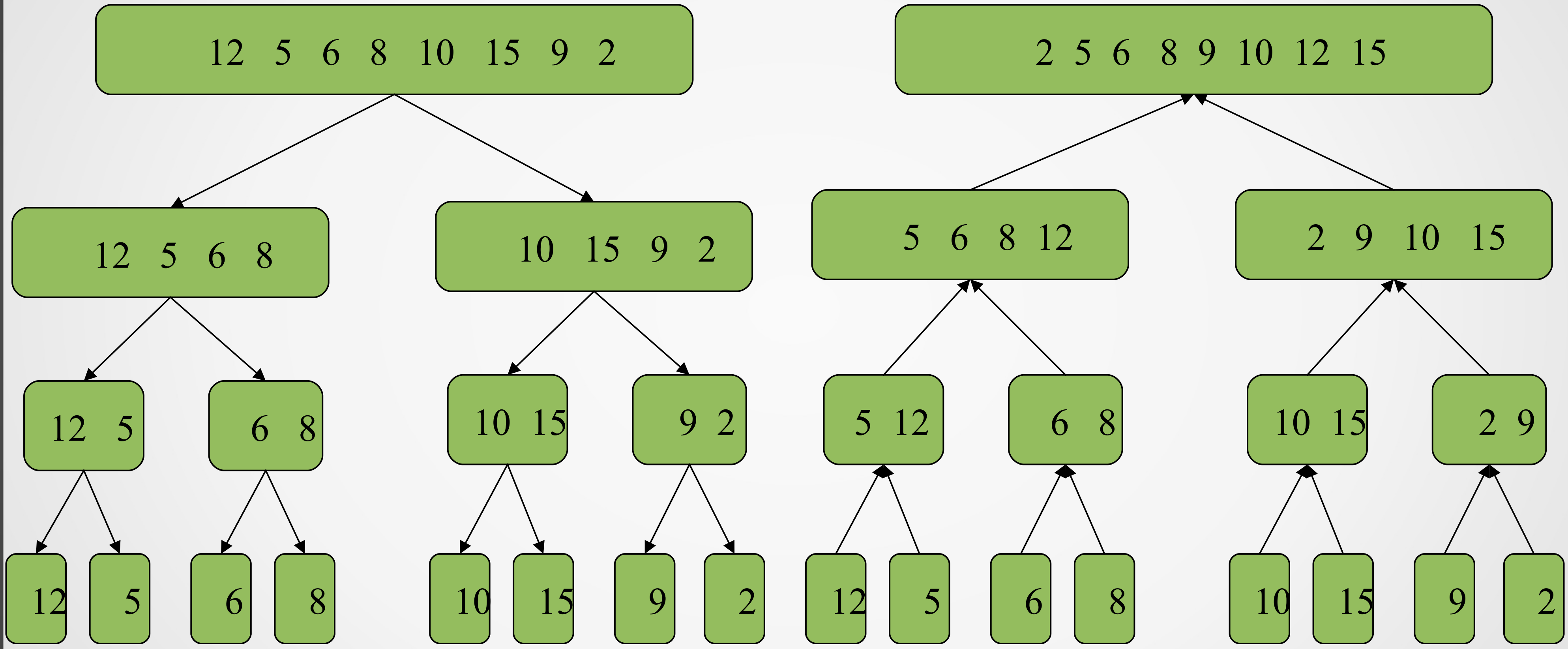
Merge Sort

- Uses divide and conquer strategy (multiple recursion) to sort a set of numbers
- Divide:
  - If S has zero or one element, return S
  - Divide S into two sequences S_1 and S_2 each containing half of the elements of S
- Recur
  - Recursively apply merge sort to S_1 and S_2
- Conquer
  - Merge S_1 and S_2 into a sorted sequence

Merging of Sorted Sequences

- Iteratively remove smallest element from one of the two sequences S_1 and S_2 and add it to end of output sequence S

Merge Sort



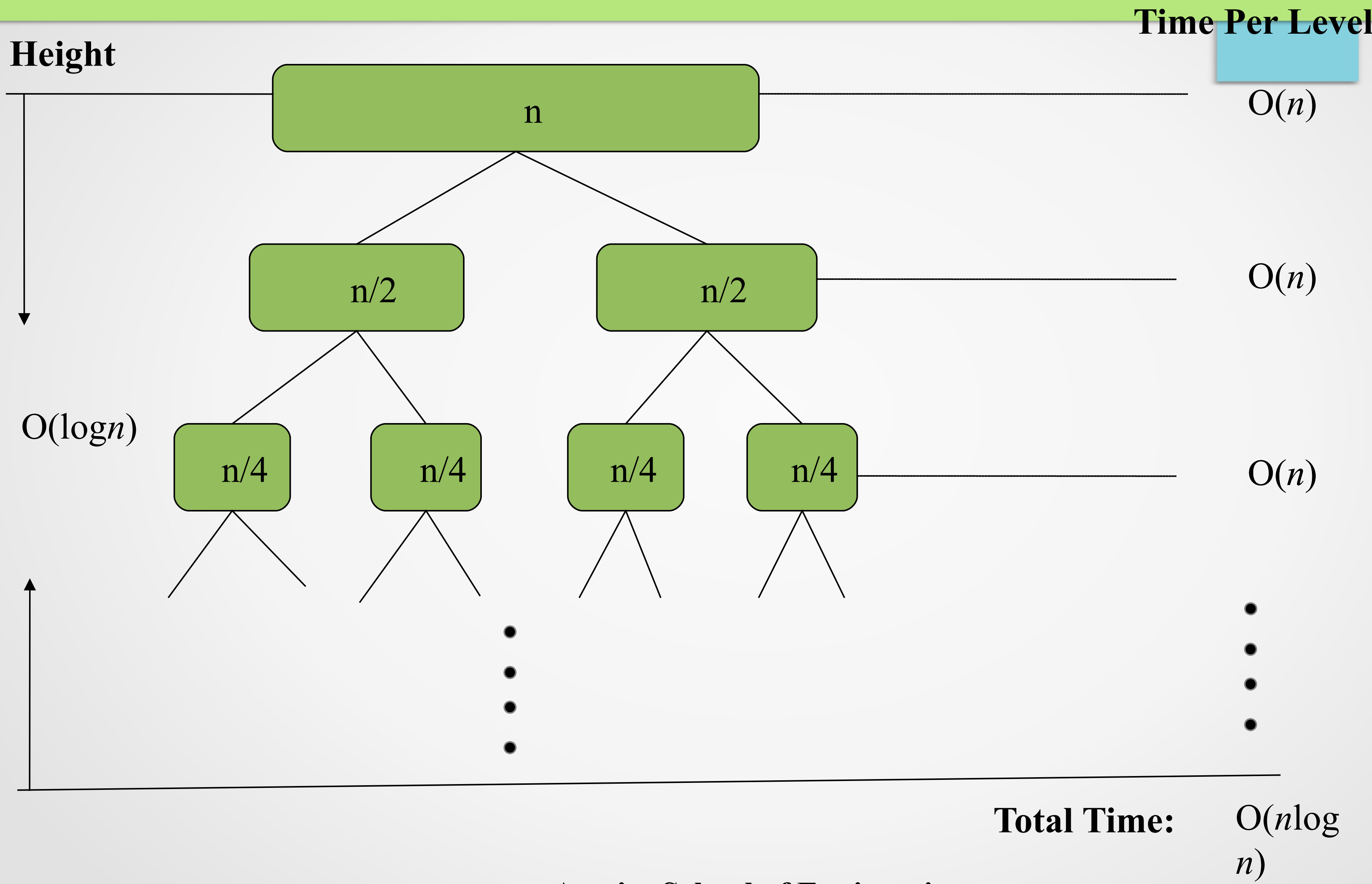
12 5 6 8 10 15 9 2
2 5 6 8 9 10 12 15
12 5 6 8
10 15 9 2
5 6 8 12
2 9 10 15
12 5
6 8
10 15
9 2
5 12
6 8
10 15
2 9
12
5
6
8
10
15
9
2
12
5
6
8
10
15
9
2

Merge Sort

```
• mergesort(S, low, high) {
      if (low < high) {
              middle = (low+high)/2;
              mergesort(s,low,middle);
              mergesort(s,middle+1,high);
              merge(s, low, middle, high);
          }
  }
```

Src: Skiena, Algorithm Design Manual, Chapter 4

Analysis of Merge Sort



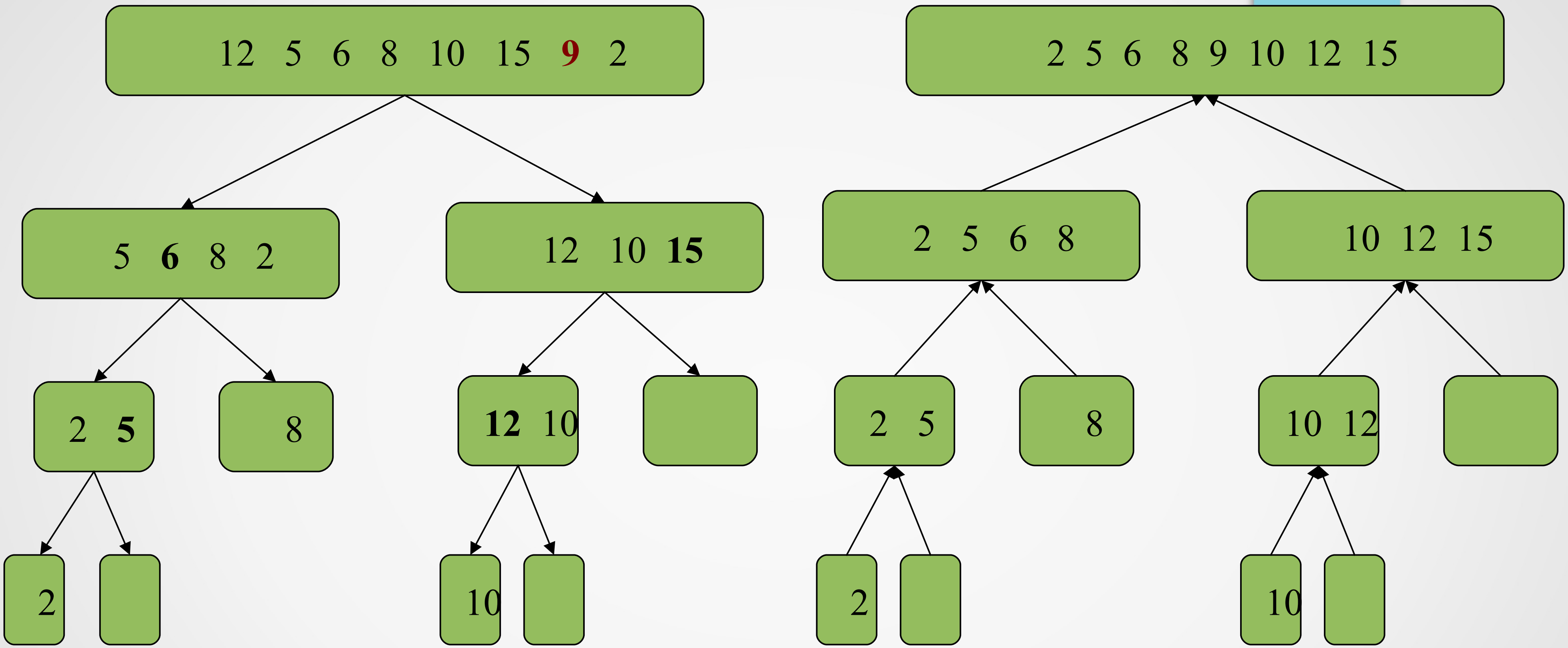
Merge Sort: Analysis

- work done on the kth level involves merging 2^k pairs sorted list, each of size $n/2^{k+1}$
  - A total of atmost n ie 2^k comparisons
- Linear time for merging at each level
  - Each of the n elements appear exactly in one of the subproblems at each level
- Requires extra memory

Quick Sort

- A divide and conquer strategy which also uses randomization
- Divide
  - Select a random pivot p. Divide S into two subarrays, where one contains elements < p and the other which are > p
- Recur
  - Sort subarrays by recursively applying quicksort on each subarray
- Conquer
  - Since the subarrays are already sorted, no work is needed in merge part

Quick Sort



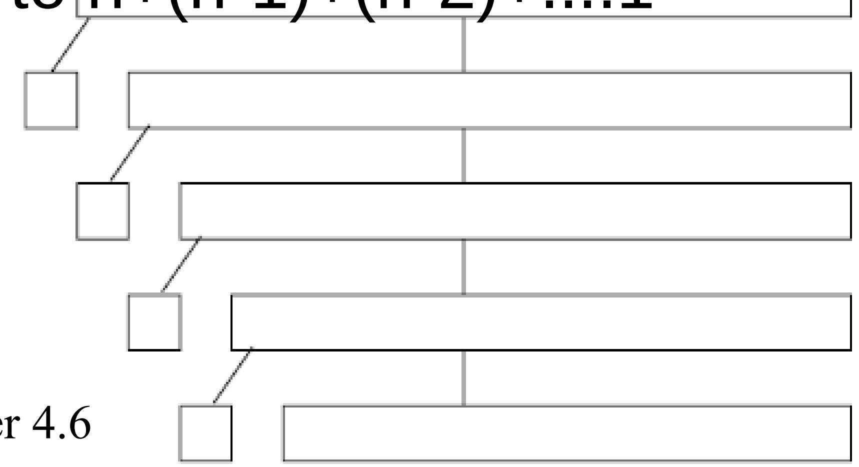
12 5 6 8 10 15 9 2
5 6 8 2
12 10 15
2 5
8
12 10
2
10
2 5 6 8 9 10 12 15
2 5 6 8
10 12 15
2 5
8
10 12
2
10

Quick Sort

- QUICKSORT(A, p, r)
  - *if* p < r
  - q = PARTITION(A,p,r)
  - QUICKSORT(A, p, q-1)
  - QUICKSORT(A,q +1, r)
- Selection of pivot
  - Can be first or last element (here)
  - Median element
  - Random element

Quick Sort Analysis

- n(logn) on average
- Worst Case Running Time
  - Occurs when pivot is always the largest element
  - Selecting the first element or last element as pivot causes this problem when list is already sorted
  - Running time proportional to n+(n-1)+(n-2)+....1
  - $O(n^2)$



Src: Skiena, Algorithm Design Manual, Chapter 4.6