Understanding Algorithms

Debswapna Bhattacharya, PhD
Assistant Professor
Computer Science and Software Engineering
Auburn University

Understanding the mechanics of algorithms

- Understanding non-recursive algorithms
- Understanding recursive algorithms
  - Drawing recursion trees
- Mentally simulating algorithms
  - Working out algorithm operations on problem instances, especially boundary cases

Understanding algorithm mechanics

1. Read and understand each step of the pseudocode

2. Simulate on a small problem instance to develop an initial sense of how the algorithm works

3. Determine boundary cases of inputs and loops, and simulate on those

4. See if all valid inputs can be categorized into distinct groups or ranges and simulate on representative inputs from each group

5. Repeat till you understand the algorithm!

Insertion Sort

Insertion-sort(*A: array [1…n] of number, n≥1*)

```
for j = 2 to n
        key=A[j]
        i = j - 1
        while i>0 and A[i]>key
                A[i+1] =A[i]
                i = i - 1
        A[i +1] = key
```

What is the inherent complexity of sorting?

An Interesting Property of Insertion Sort

- The numbers are sorted *in- place*, i.e., within the input data structure, without using any additional memory

- Therefore Insertion sort is an in-place algorithm

- In-place algorithms are the most space efficient algorithms

```
Mystery(x: real, a: non-negative integer) returns real
    temp=1
    while a>0
        temp=temp*x
        a = a - 1
    return temp
```

What does Mystery compute?

- Range of input?
- Simulate on a small problem instance
- Boundary cases?

Recursive and Non-Recursive Algorithms

- What is a recursive algorithm?
  - Calls itself until it hits the base cases
- What is an iterative algorithm?
  - Repeats some steps and stops by checking some condition
- Understanding non-recursive algorithms
  - which is what we did in the previous examples
- Understanding recursive algorithms

http://interactivepython.org/runestone/static/pythonds/Recursion/pythondsintro-VisualizingRecursion.html

https://www.youtube.com/watch?v=Ivg25WzNJ7o

Recursive Algorithms

- How can I understand (think about) a recursive algorithm?
  - Same technique but also draw a **Recursion Tree**

- Is a recursive algorithm always efficient/inefficient compared to its non-recursive counterpart?
  - not necessarily
  - the system cost of recursion: maintaining the **call stack**

- When should I use a recursive algorithm?
  - when the problem is amenable to a recursive solution strategy without sacrificing efficiency

- When should I not use a recursive algorithm?
  - avoid **tail recursion!**
  - avoid **duplicated work!**

Recursion Trees

- A Recursion Tree is a graphical representation of the operation of a recursive algorithm on a problem instance.
  - Each node represents one execution of the algorithm
    - The root node stands for the original call to the algorithm
    - The leaf nodes are executions that do not generate further recursive calls, called base cases
  - Each downward edge represents a new instance/execution
    - Each upward edge represents return of control to the calling algorithm when the new execution terminates
  - Each node is annotated with the inputs to that execution
  - Each upward edge is annotated with the values returned by an instance that finished executing

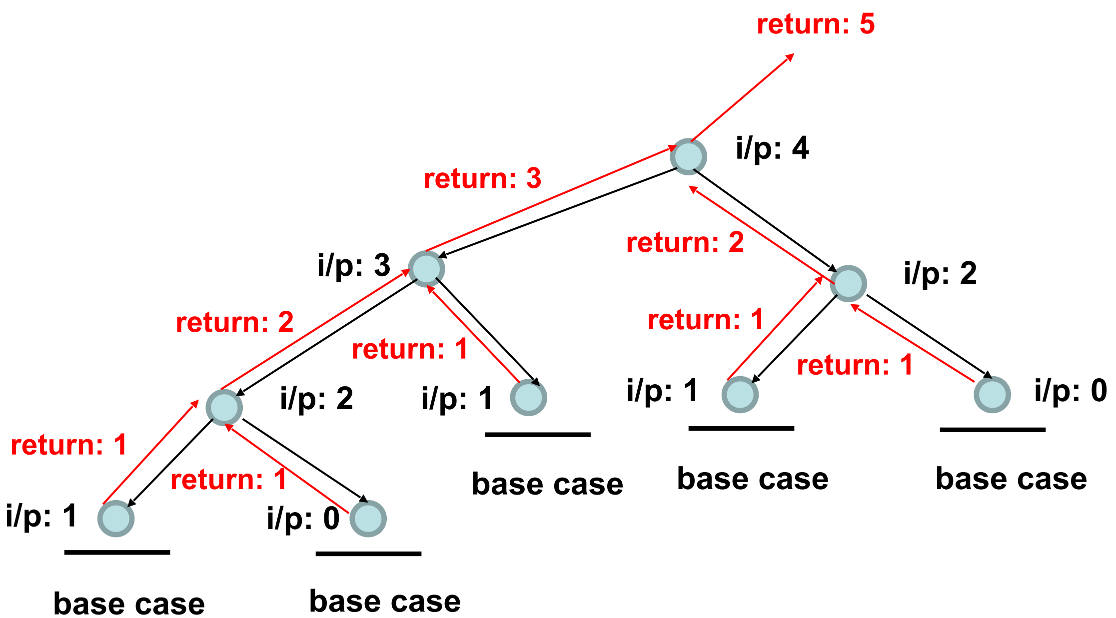
Recursion Tree Example

The Fibonacci Algorithm

Fib(n: non-negative integer) returns non-negative integer

1 if n==0 or 1 then return 1

2 return Fib(n-1)+Fib(n-2)



Tail Recursion

- General tail recursion – when there is only one recursive call that occurs toward the end of the algorithm
- Why it is inefficient:
  - call stack storage & processing
- How to make it more efficient:
  - replace recursion with iteration manually
- A stricter form of tail recursion - when there is only one recursive call that is the very last step of the algorithm
  - compilers can automatically replace recursion with iteration

Power-of-2-Alg1(n: non-negative integer)

```
if n==0 then return 1
else return 2*Power-of-2-Alg1(n–1)
```

General tail recursion: recursive call is not the very last thing that happens

Power-of-2-Alg2(n : non-negative integer)

```
Power-of-2-recursive(n,1)
```

Power-of-2-recursive(n, accum: non-negative integer)

```
if n==0 then return accum
else return Power-of-2-recursive(n–1, accum*2)
```

Strict tail recursion: recursive call is the very last thing that happens

In-class Exercise

Write non-recursive versions of:

Power-of-2-Alg1

and

Power-of-2-recursive

Example: Find-Max

Find-max-1(A:array [i...j] of number) returns number
k: number
1 if i==j then return A[i]
2 k= find-max(A:array [i+1...j])
3 if k>A[i] then return k else return A[i]

Thinking about Recursion

- How does this algorithm find the max?
- What are the legal inputs?
- What is the base case?
- Draw the recursion tree showing inputs and outputs if the input is the array [1,2,3,4,5]
- Why is this an example of tail recursion?

What does Mystery compute?

```
Mystery(n: non-negative integer)
    if n==0 then return 1
    else return 2*Mystery(n–1)
```

Thinking Assignments: Removing Tail Recursion

- Why are Find-Max and Mystery examples of tail recursion?

- How to turn them into iterative (and therefore more efficient) algorithms?

Fibonacci: Duplicated Work

Fib(n: non-negative integer) returns non-negative integer

1 if n==0 or 1 then return 1

2 return Fib(n-1)+Fib(n-2)

Thinking about Recursion

- How does this algorithm find the n^{th} Fibonacci #?
- What are the legal inputs?
- What is (are) the base case(s)?
- Draw the recursion tree of fib(4)
- Why is this an example of duplicated work?
- How to turn this into an iterative (and therefore more efficient) algorithm that does not duplicate work?

MERGE-SORT(A:array [p…r] of number)

```
if p < r
    then m= ⌊(p+r)/2⌋
        MERGE-SORT(A[p…m])
        MERGE-SORT(A[m+1…r])
        MERGE(A[p…r],m)
```

MERGE Procedure

$n_1 = m - p + 1$

$n_2 = r - m$

create arrays $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n_2 + 1]$

for $i = 1$ ***to*** n_1

    $L[i] = A[p + i - 1]$

for $j = 1$ ***to*** n_2

    $R[j] = A[m + j]$

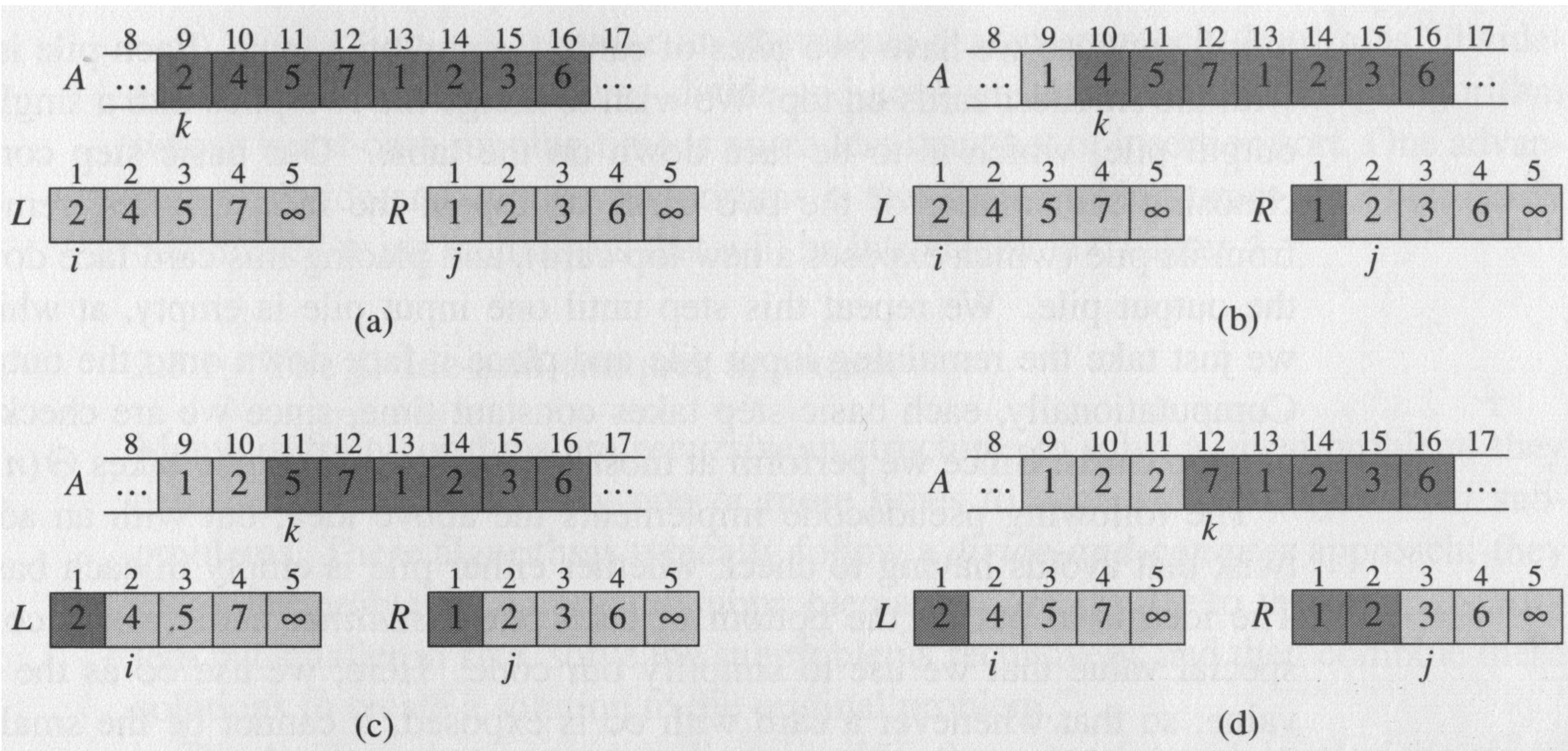
$L[n_1 + 1] = \infty$

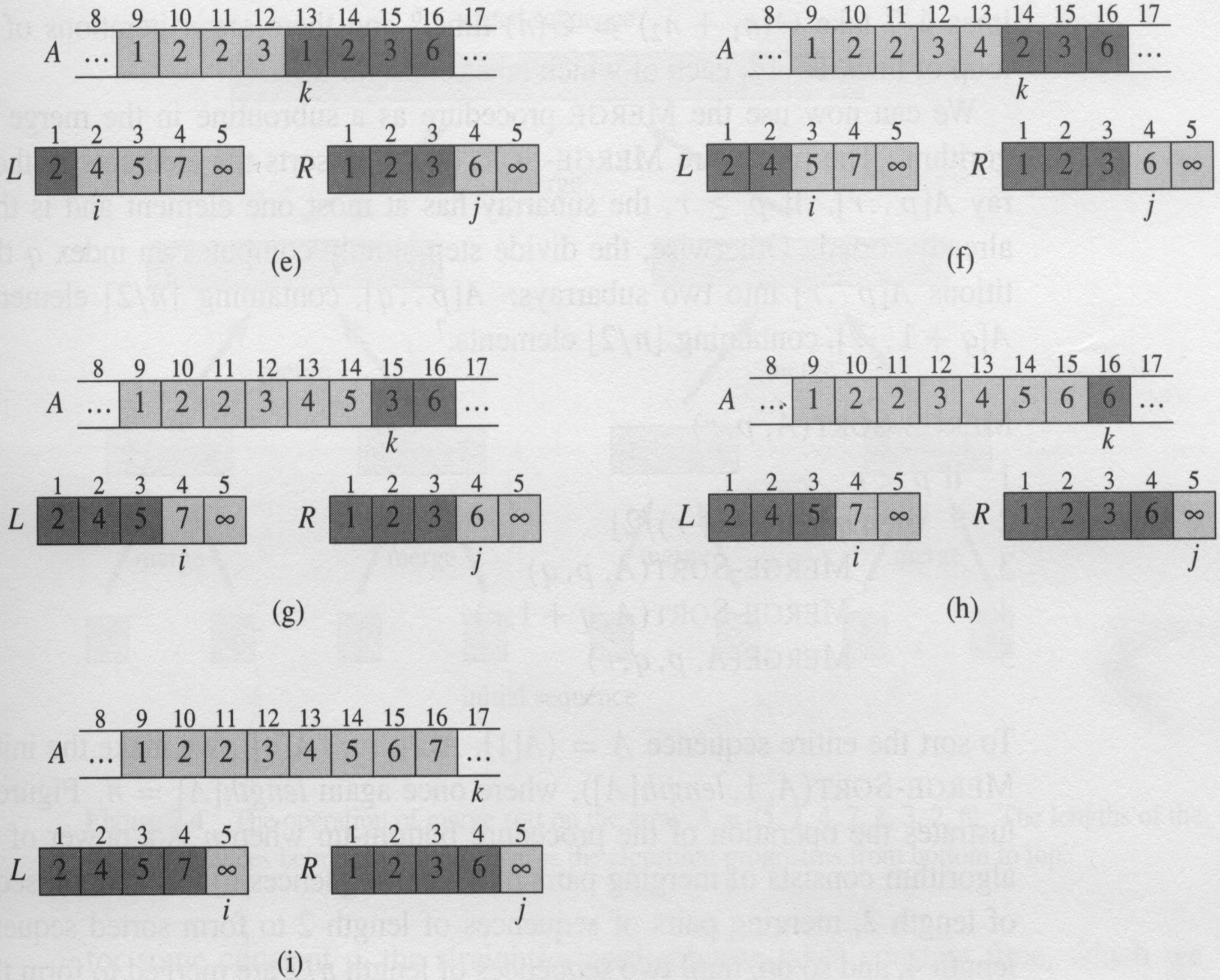
$R[n_2 + 1] = \infty$

MERGE Procedure

```
    i = 1
    j = 1
for k = p to r
    if L[ i ] ≤ R[ j ]
        then A[ k ] = L[ i ]
             i = i + 1
        else A[ k ] = R[ j ]
             j = j + 1
```

Operation of Merge





8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
(e)
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
(f)
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
(g)
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
(h)
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
(i)

Thinking about Recursion

- How does this algorithm sort?
- What are the legal inputs?
- What is/are the base case/s?
- Draw the recursion tree showing inputs and outputs if the input is the array [5,2,4,7,1,3,2,6]

Thinking Assignment

```
Mystery(A: array [p…q] of number)
left, right, temp: array [1…2] of number
if p==q then
        temp[1]=temp[2]=A[p]
        return temp
m=
left=Mystery(A[p…m])
right=Mystery(A[m+1…q])
if left[1]<right[1] then
        temp[1]= left[1]
        else temp[1]= right[1]
if left[2]>right[2] then
        temp[2]= left[2]
        else temp[2]= right[2]
return temp
```

What does this algorithm do?

How does it do it?

Thinking Assignment

```
Find-max-2(A:array [i...j] of number)
if i==j then return A[i]
m=
left-max= find-max-2(A [i...m])
right-max= find-max-2(A [m+1...j])
if left-max>right-max
    then return left-max
    else return right-max
```

- How does this algorithm find the max?
- What are the legal inputs?
- What is the base case?
- Draw the recursion tree showing inputs and outputs if the input is the array [1,2,3,4,5]
- Is this an example of tail recursion?

Thinking Assignment

```
MaxMin(A:array [1…n] of number)
if n is odd then
    then max=min=A[n]
    else max=
for i=1 to
    if A[2i1]≤A[2i]
        then small= A[2i1]; large=A[2i]
        else small= A[2i]; large=A[2i 1]
    if small<min then min=small
    if large>max then max=large
```

- Does the above algorithm <u>correctly</u> find the max and min numbers in the input array? What are its legal inputs?
- Write an algorithm to find max and min using the strategy of scanning the array left to right, keeping track of the max and min numbers using two local variables.
- How is the above given algorithm's strategy different from the left to right scanning? Which algorithm is more efficient – the above one or yours?

Reading Assignment

- Chapter 2
  - Section 2.1
    - Omit (for the time being) the discussion of loop invariants (p. 18-20)
    - You should already have read p. 20-22.
    - Try some of the problems at the end of this section.
  - Section 2.3
    - Omit (for the time being) the discussion of loop invariants (p. 32-33)
    - Omit (for the time being) Section 2.3.2