MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

15 Algorithms Analysis

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

February 17, 2022

Binary Recursion (from last lecture)

- Again! summing the **n** elements of an integer array **A**:
  - recursively
    - summing first half
    - summing second half
    - adding the two

Algorithm BinarySum(A, i, n):

    Input: An array A and integers i and n

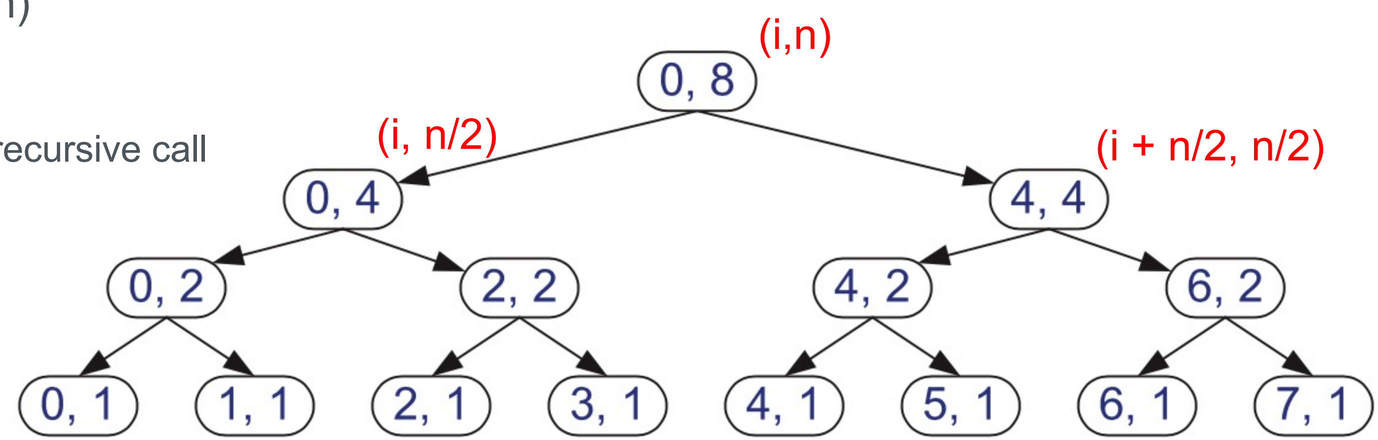
    Output: The sum of the n integers in A starting at index i

    if $n = 1$ **then**

        return $A[i]$

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

- Analysis of the algorithm:
  - We assume n is a power of 2
  - BinarySum(A, 0, n)
    - BinarySum(A, 0, 8)
  - *n* is halved at each recursive call

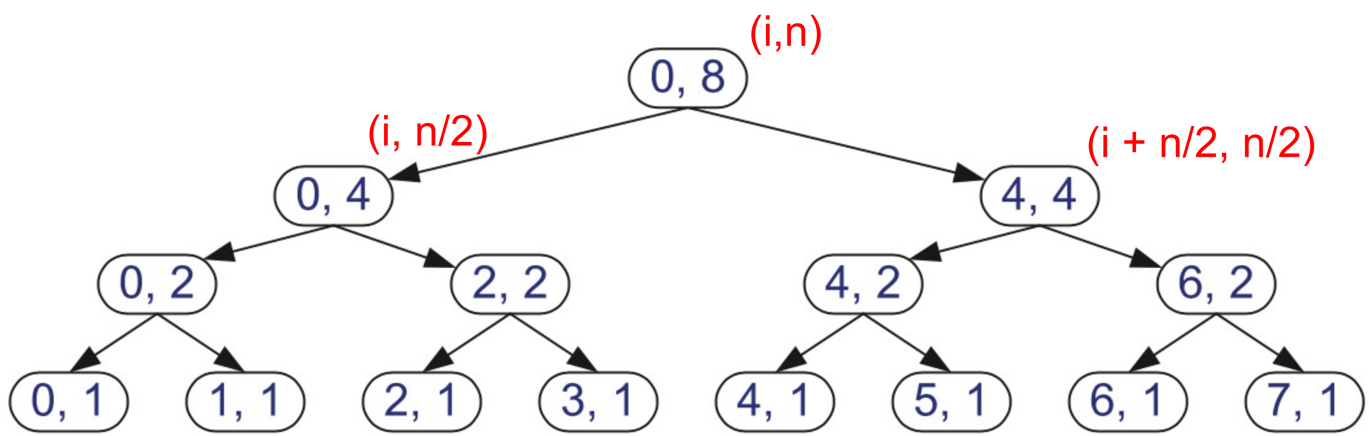


Binary Recursion (from last lecture)

- Analysis of the algorithm:
  - We assume n is a power of 2
  - BinarySum(A, 0, n)
    - BinarySum(A, 0, 8)
  - *n* is halved at each recursive call

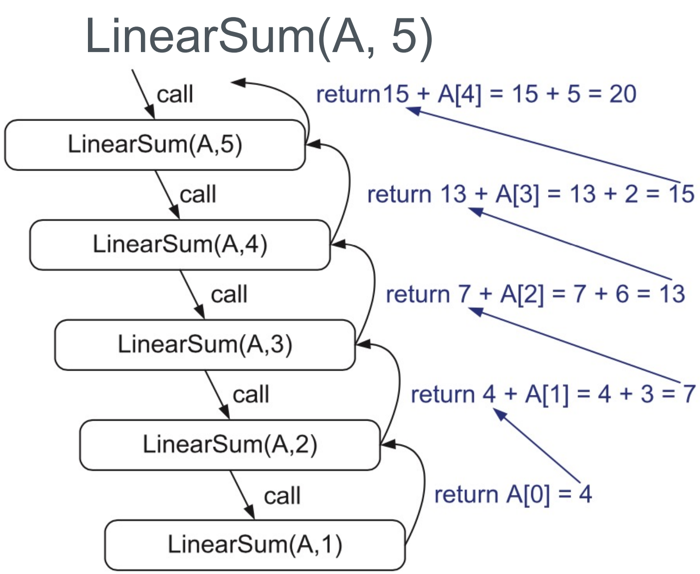
Algorithm BinarySum(A,i,n):
  Input: An array A and integers i and n
  Output: The sum of the n integers in A starting at index i
  if $n = 1$ **then**
    return $A[i]$
  return BinarySum$(A,i,\lceil n/2\rceil)$ + BinarySum$(A,i+\lceil n/2\rceil,\lfloor n/2\rfloor)$

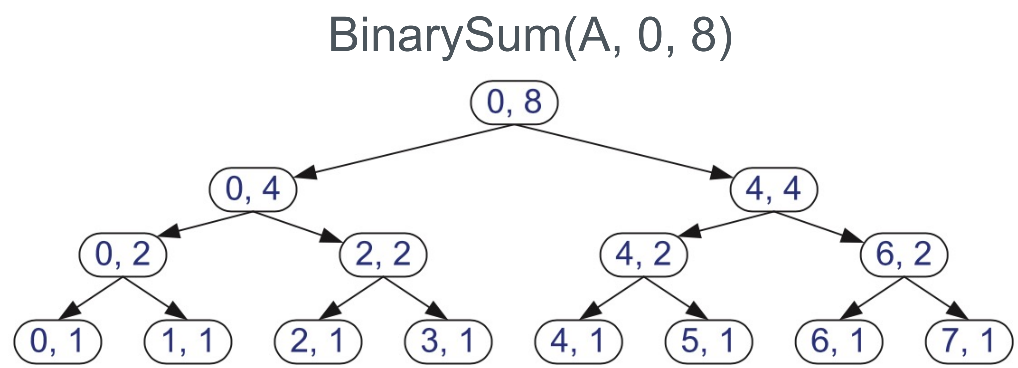
- The depth of the recursion, that is, the maximum number of function instances that are active at the same time, is $\mathbf{1 + \log_2} \boldsymbol{n}$



Binary Recursion (from last lecture)

- Analysis of the algorithm:
  - We assume n is a power of 2
- **Memory consumption** (Space complexity)
  - The depth of the recursion, that is, the maximum number of function instances that are active at the same time, is $\mathbf{1 + \log_2} \boldsymbol{n}$
  - Remember that this depth was **n** for **LinearSum**

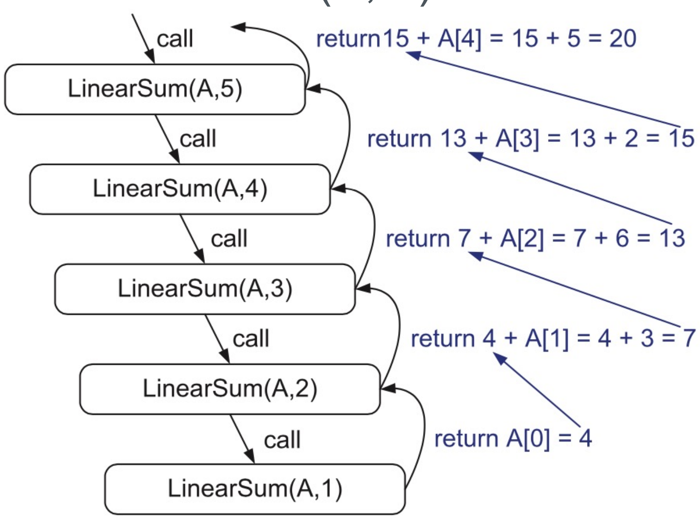




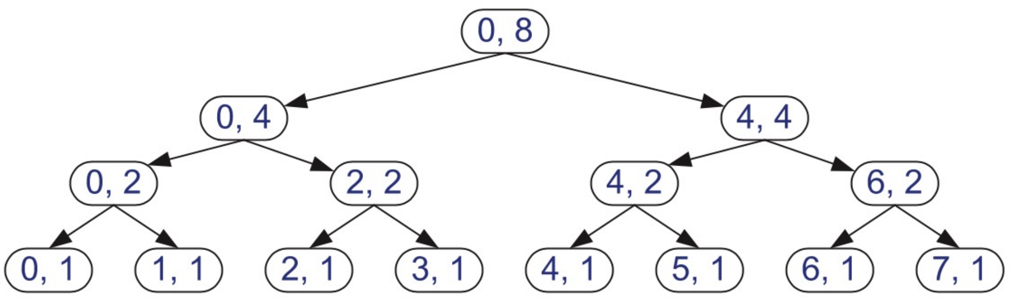
Binary Recursion (from last lecture)

- Analysis of the algorithm:
  - We assume n is a power of 2
- **Runtime** (Time complexity)
  - There are **2n-1** boxes (calls) in **BinarySum**
  - There are **n** boxes (calls) in **LinearSum**
- Assume each calls is visited in a constant time

LinearSum(A, 5)



BinarySum(A, 0, 8)

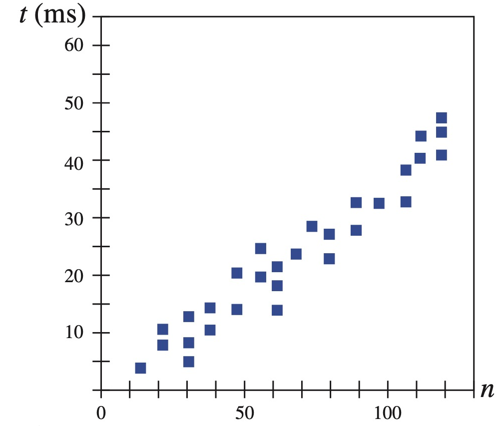


Comparing Algorithms

- Algorithm: An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.
- There are different solutions for a problem
- We need to identify. which solution is **better** than the other.
- Better in the sense of:
  - Running time
  - Memory space required
  - structure of programs (readability, simplicity, design)

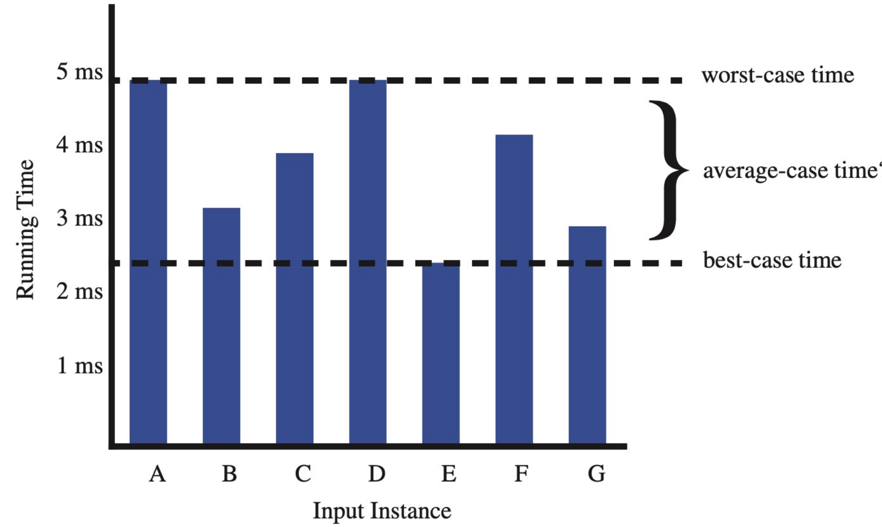
Running Time

- Most algorithms transform input objects into output objects.
  - We need to think about input
    - Insertion sort: array of size **n**
    - recursive factorial: **n**
- The running time of an algorithm typically grows with the input size.
  - best cases
  - average cases
  - worst cases
- We focus on the worst-case running time.
  - Easier to analyze
  - Crucial to applications such as games, embedded systems
- Average-case running time is often difficult to determine.
  - Why?



Average vs. Worst Case

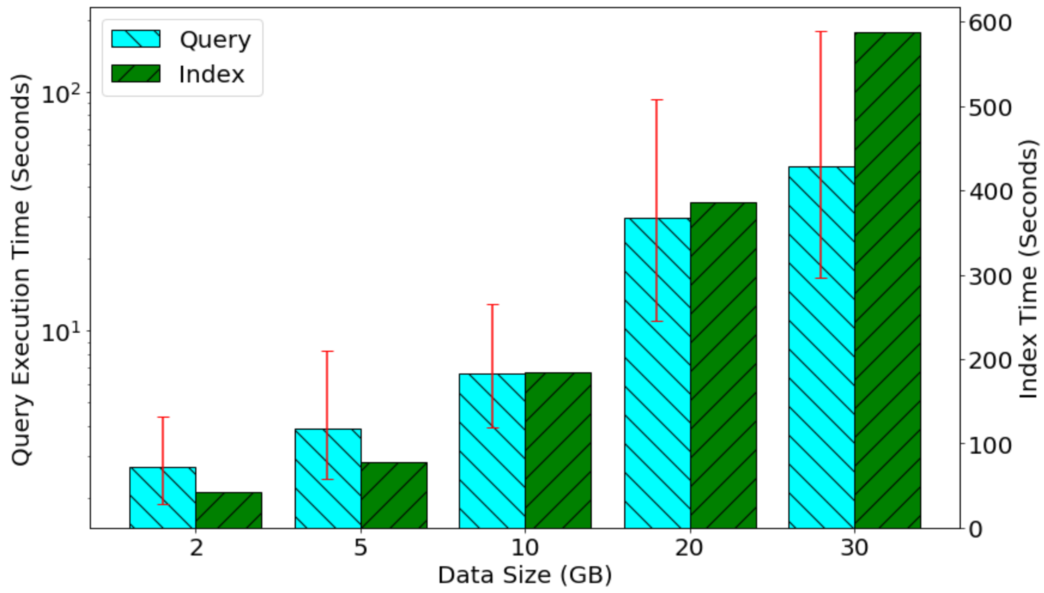
- The average case running time is harder to analyze because you need to know the probability distribution of the input.



- knowing the worst-case time is important in many applications
  - real-time systems
  - nuclear reactor
  - air traffic control

Experimental Approach

- Implement your algorithm
- Run the program with inputs of varying size and composition
- Use a wall clock to get an accurate measure of the actual running time
- Plot the results



Experimental Approach - Limitations

- It is necessary to implement your algorithm, which may be difficult and often time-consuming
  - Sometimes you just ideate!
- Results may not be indicative of the running time on other inputs not included in the experiment.
  - Your input may not be inclusive enough of all possible inputs
- Competing algorithms!
  - In order to compare your algorithm with a competing algorithm, the same hardware and software environments must be used
    - Your competitor may have ran on a high-end machine to which you don’t have access
      - You need to implement it!
      - Or buy the same machine

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
  - The algorithm receives a worth of data
  - Performs a few operations on the data
- Characterizes running time as a function of the input
  - Size of input: array of length **n**
  - input: **n** for factorial
- Considers all possible inputs
- Allows us to evaluate the **relative efficiency** of any two algorithms **independent of** the hardware/software environment
- For each algorithm, we will end up with a function **f(n)** that characterizes the running time of the algorithm as a function of the input size **n**

Primitive Operations

- Primitive operation corresponds to a low-level instruction with an execution time that is constant
  - Basic computations performed by an algorithm
  - Identifiable in pseudocode
  - Largely independent of the programming language
  - Exact definition is not important
- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$

for $i \leftarrow 1$ **to** $n-1$ **do**

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$

for $i \leftarrow 1$ **to** $n-1$ **do**

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$\boxed{currMax \leftarrow A[0]}$ <-------- 2 operations

for $i \leftarrow 1$ **to** $n-1$ **do**

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

1. accessing A[0] (indexing in array)
2. assigning A[0] to currMax

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

for $i \leftarrow 1$ **to** $n - 1$ **do**

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

The for loop repeats **n** times, why?
Each time it has **2** operations, why?

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations
$\boxed{\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do}}$
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

The for loop repeats **n** times, why?
Each time it has **2** operations, why?
  each time it involves an **assignment** and a **comparison**

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

for $i \leftarrow 1$ **to** $n-1$ **do**

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

The for loop will repeat **n** times, why?

We account for the last increment to **n** in which the for loop identifies it should exit before entering next iteration

for example: n = 4

for ***i*** from ***1*** to ***3*** ➔ in the last iteration ***i*** becomes ***4*** and will be compared to ***3***

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

$\boxed{\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do}}$ <----- 2n operations

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

The for loop will repeat **n** times, why?

We account for the last increment to **n** in which the for loop identifies it should exit before entering next iteration

for example: n = 4

for ***i*** from ***1*** to ***3*** ➔ in the last iteration ***i*** becomes ***4*** and will be compared to ***3***

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

for $i \leftarrow 1$ **to** $n-1$ **do** <----- 2n operations

  if $currMax < A[i]$ **then** <-- 2(n-1) operations

   $currMax \leftarrow A[i]$ <-- 2(n-1) operations

  i++ <-- 2(n-1) operations

return $currMax$

The body of for loop will repeat **n-1** times

The increment of ***i*** is performed at the end of each iteration

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

for $i \leftarrow 1$ **to** $n - 1$ **do** <----- 2n operations

if $currMax < A[i]$ **then** <-- 2(n-1) operations

$currMax \leftarrow A[i]$ <-- 2(n-1) operations

i++ <-- 2(n-1) operations

return $currMax$ <-- 1 operation

We will have a total of 8n - 2 operations

n is the input size!

Estimating Runtime

- Algorithm **arrayMax** executes 8n - 2 primitive operations in total

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

$currMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

- Suppose:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation

- The running time of **arrayMax** is bounded by two linear functions
  - **a**(8n - 2) <= **T(n)** <= **b**(8n - 2)
- Changing the hardware / software environment
  - Affects **T(n)** by a constant factor, but does not alter the growth rate of T(n)
- The linear growth rate of the running time **T(n)** is an **intrinsic property** of algorithm **arrayMax**

Questions?