CSCE 411: Design and Analysis of Algorithms

Lecture 1: Intro, Asymptotic Runtimes, Divide and Conquer

Date: *Lecturer: Nate Veldt*

Course Logistics

- Read section 2.3, and chapter 4 for first week of classes.
- Read (or skim) chapters 1-3 to ensure familiarity with prerequisites
- Syllabus quiz is due Sat, Aug 24. HW 1 and intro video due Fri, Aug 30

1 Computational Problems and Algorithms

Definition 1. A computational problem is a general task defined by a specific type of input and an explanation for a desired output

A specific case of the problem is called an instance

Example 1. Sorting

Input: A sequence of n numbers: $a_1, a_2, \ldots, a_n$

Output: A permutation σ of the input sequence so that

$$a_{\sigma(1)} \leq a_{\sigma(2)} \leq \cdots \leq a_{\sigma(n)}$$

An instance of this problem is the sequence 1, 0, -2, 3, 12

Example 2. Min element

Input: An array of n numbers: $[a_1, a_2, \ldots, a_n]$

Output: The smallest element in the array and its index.

instance: [10, -1, 0, 12, 13]

Definition 2. An **algorithm** is a computational procedure that

- takes an input for a problem
- applies a concrete set of steps to produce an output

An **algorithm** is said to be **correct** if it always produces the right output for every input.

↳ for a certain problem

2 Asymptotic Runtime Analysis (Chapter 3)

2.1 Rules for runtime analysis

- n denotes the size of the input

$O(1)$
- Each basic operation takes constant time $\times, -, +,$ comparing two numbers.

- We focus on the worst-case runtime

- We only care about the order of the runtime, i.e., the asymptotic runtime.

2.2 Some initial examples

Question 1. Given an array of n items, find whether the array contains a negative number using the following steps:

for $i = 1$ to n **do**
    if $a_i < 0$ **then**
        Return (true, i)
    end if
end for

What is the runtime of this method?

A $O(1)$

B $O(n)$

C $O(n^2)$

D It depends

E Other

F Don't know, I need a reminder for how this works.

Question 2. Given an $n \times n$ matrix A, what is the runtime of summing the upper triangular portion using the following algorithm? (same answers).

$sum = 0$
for $i = 1$ to n **do**
    for $j = i$ to n **do**
        $sum = sum + a_{ij}$
    end for
end for
Return sum

$$A = \begin{bmatrix} 1 & 2 & 5 \\ 9 & 13 & -10 \\ 0 & 4 & 3 \end{bmatrix}$$

$$\frac{n(n-1)}{2} + \underline{\underline{n}} = \binom{n}{2} + n = O(n^2)$$

2.3 Formal Definitions

Let n be input size, and let f and g be functions over $\mathbb{N}$.

$0 \leq f(n) \qquad 0 \leq g(n)$

Definition 3. Big O notation.

A function $g(n) = O(f(n))$ (we say, "g is big-O of $f(n)$") means:

There exists $c > 0$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$

$$g(n) \leq c f(n)$$

Definition 4. Big Ω notation.

$g(n) = \Omega(f(n))$ means:

There exists $c > 0$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$

$$c f(n) \leq g(n)$$

(same as $f(n) = O(g(n))$)

$$f(n) \leq \frac{1}{c} g(n)$$

Definition 5. Θ notation.

$g(n) = \Theta(f(n))$ means:

There exist $d > c > 0$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$

$$c f(n) \leq g(n) \leq d f(n)$$

Equivalently, this means

$$g(n) = O(f(n))$$
and $g(n) = \Omega(f(n))$

Additional runtime examples

1. $4n^4 + n^3 \log n + 100n \log n$ = $O(n^4)$

2. $n + 2(\log n)^2$ = $O(n)$

3. $2^n + 10^{100}n^45$ = $O(2^n)$

$\log = \log_{10}$

$\lg = \log_2$

$\ln = \log_e$

Logarithms in Runtimes Which of the following runtimes are the same asymptotically? Which are not?

- $O(n \log n)$ and $O(n \lg n)$

  Same

  $\lg n = \log_2 n = \dfrac{\log_{10} n}{\boxed{\log_{10} 2}}$

- $O(\log n)$ and $O(\log^2 n)$

  different

  $\log^2 n = (\log n)^2$

- $O(\log n)$ and $O(\log(n^2))$

  same

  $\log n^2 = 2 \log n$

- $O(n^{\log 100})$ and $O(n^{\lg 100})$

  different

$\log_{10} 100 = 2$

$\log_2 100 > 6$

3 How to Present an Algorithm

Presenting and analyzing can be broken up into four steps.

1. **Explain**: the approach in basic English
2. **Pseudocode**: for formally presenting the algorithmic steps
3. Prove: the **correctness**
4. Analyze: the **runtime complexity**

As a rule it's a good idea to go through all steps when presenting an algorithm. Sometimes we will focus more on just a subset of these (e.g., you may be asked to prove a runtime complexity of an algorithm on a homework but not a correctness proof).

We will go through all four steps when we present the *merge sort* algorithm.

4 The Divide and Conquer Paradigm

The divide and conquer paradigm has three components:

- **Divide**: the problem into subproblems
- **Conquer**: the subproblems by solving them (recursively)
- **Combine**: solutions to subproblems into a solution for the whole problem.

Example: Mergesort (Textbook, Chapter 2.3, 4) Given n numbers to sort, apply the following steps:

- Divide the sequence of length n into two arrays of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$
- Recursively sort the subarrays
- Combine the subarrays by repeatedly comparing the largest elements in each subarray.

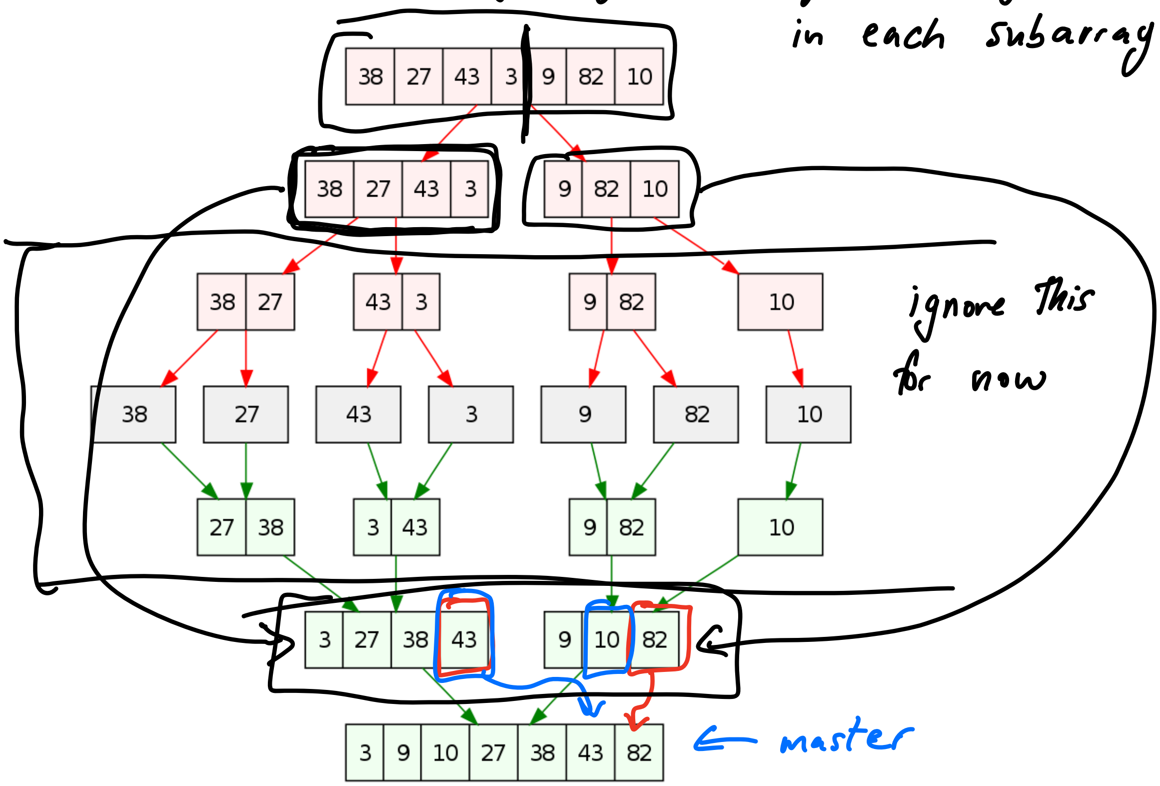
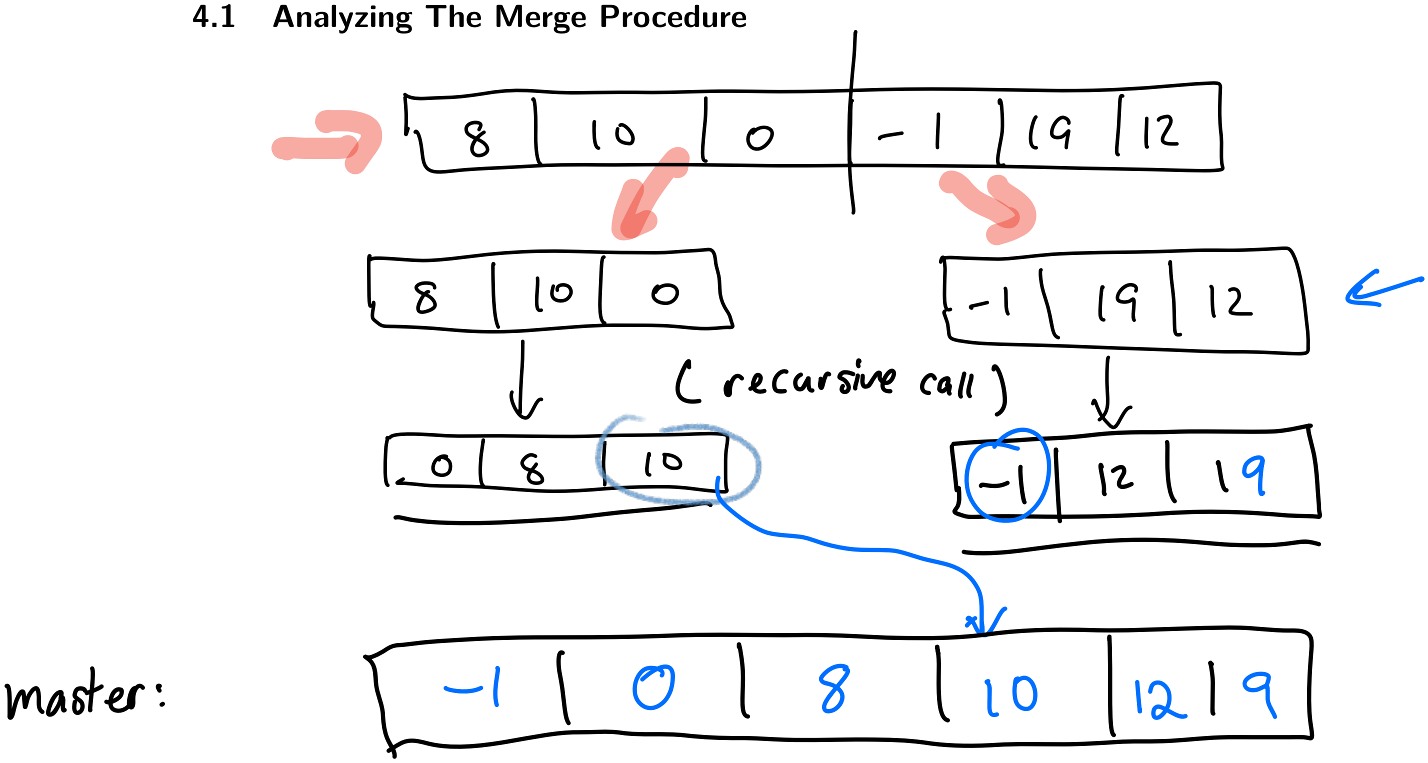


Image courtesy of Wikipedia: https://en.wikipedia.org/wiki/Merge_sort.

MERGESORT(A)
  n = **length**(A)
  if $n == 1$ **then**
    return A
  else
    $m = \lfloor n/2 \rfloor$
    $L = A[1 \ldots m]$
    $R = A[m+1, m+2, \ldots, n]$
    $L' = \text{MergeSort}(L)$
    $R' = \text{MergeSort}(R)$
    return $\text{Merge}(L', R')$
  end if

4.1 Analyzing The Merge Procedure



L and R

Correctness: To merge two sorted subarrays into a master array

- Maintain a pointer to the end (max value location) of each subarray L&R.
- At each step, compare the two numbers from the subarrays
- Since subarrays are sorted, one of these numbers is guaranteed to be the largest among all un-merged numbers in either subarray.
- so place that largest number in the next open position in the master array.
- At each step, we guarantee that: we place the next largest number in the master array.
- So continuing until both subarrays are empty, this yields a sorted master array.