Program Structure and Algorithms

Sid Nath

Lecture 4

Agenda

- Administrative
  - Quiz #2 today – solving recurrences

- Lecture

Hash Data Structures

- Data structure to store key/value pairs
- Dictionary is a data structure that you can store values by keys
  - E.g., A[key] = value

- Suppose you have an arbitrary object (e.g., string) and you want to assign a unique key to make searching easy.
  - E.g., Student ID for each student x major
  - E.g., Code for books in a library
- Hashing: Convert large keys to small keys using a function (hash function). Key/value stored in hash table
- Given a key, read/update O(1)

Hash Data Structures

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	

Hash Table

Frequency

Char	Index	Value
a	0	2
b	1	2
c	2	1
d	3	1
e	4	0
-	-	-
-	-	-
y	24	0
z	25	0

Divide and Conquer (D-Q)

- Basic idea
  - *divide* the problem into 2 or more sub problems
  - *conquer* the sub problems recursively
  - combine sub problems

- Binary search
- Merge sort
- Problems
  - Find Majority
  - Find index of first “1”

Binary Search

Given a **sorted** array $A[1:n]$ integers, find the position of x (another integer) if it exists in $A[1:n]$

Linear search?

- Scan through every element in A starting from index 1
- Return i if $A[i] == x$
- Return -1 if x cannot be found we index is n.

Running time is $O(n)$

Binary Search

Given a **sorted** array $A[1:n]$ integers, find the position of x (another integer) if it exists in $A[1:n]$

Binary search

- Find middle element mid = lo + (hi – lo)/2
- Return i if $A[mid] == x$
- If $A[mid] < x$, search $A[mid+1:hi]$ (subproblem)
- If $A[mid] > x$, search $A[lo:mid-1]$ (subproblem)
- Return -1 if x cannot be found in subproblem.
- Initially, lo = 1, hi = n

Binary Search – Running time

- In each iteration, we discard one half of the subproblem
- So, the size of each successive subproblem is halved
  - Let, $n = 2^k$
  - Subproblem sizes: $2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow \cdots \rightarrow 2^{k-k} = 1$
- In each iteration we do a constant amount of work
  - Either check for equality or inequality of $A[mid]$ with x
- We can write this down as a recurrence relation
  - $T(n) = T\left(\frac{n}{2}\right) + O(1)$
- Using Master theorem, we can solve this as:
  - $a = 1, b = 2, d = 0, k = 0 \Rightarrow \log a \,/ \log b \; = d \Rightarrow Case\ 2$
  - So, $\Theta\left(n^d \log^{k+1} n\right) = \Theta(\log n)$

Binary Search -- Example

$A = \{2, 5, 8, 12, 16, 23, 38, 56, 72, 91\}; x = 23$

Iteration #1 (lo = 0, hi = 9, x = 23)

mid = 0 + (9 – 0)/2 = 4

A[4] = 16 < 23, so look in upper half

that is, A[5:9]

Iteration #2 (lo = 5, hi = 9, x = 23)

mid = 5 + (9 - 5)/2 = 7

A[7] = 56 > 23, so look in lower half

that is, A[5:6]

Iteration #3 (lo = 5, hi = 6, x = 23)

mid = 5 + (6 - 5)/2 = 5

A[5] = 23 == 23, so return 5

Merge sort

- Merge sort on Sequence **S** of N elements
  - Divide: Divide **S** into disjoint subsets **S1** and **S2**
  - Conquer: Recursively merge sort **S1** and **S2**
  - Combine: Merge **S1** and **S2** into a sorted sequence

MERGE-SORT (*A*, *p*, *r*)
if *p* <*r* #Check base case
  then #Divide
  MERGE-SORT (A, *p*, *q*) #Conquer
  MERGE-SORT (*A*, *q* +*1*, *r*) #Conquer
  MERGE (*A*, *p*, *q*, r) #Combine

$$q = \lfloor (p + r) / 2 \rfloor$$

Initial call: MERGE-SORT (*A*, 1, *n*)

Merge procedure?

- **Input:** Array A and indices p, q, r, $p \leq q < r$
  - A[p ..q] & A[q +1 ..r]: sorted, neither sub array is empty.
- **Output:** two subarrays are merged into a single sorted subarray A[p ..r].
- ***Idea behind merging:*** Think of two piles of cards.
  - each pile is sorted and placed face-up on a table with the smallest cards on top
  - we will merge these into a single sorted pile, face-down on the table.
- Initially, p = 1, r = n

Merge Sort - Pseudocode

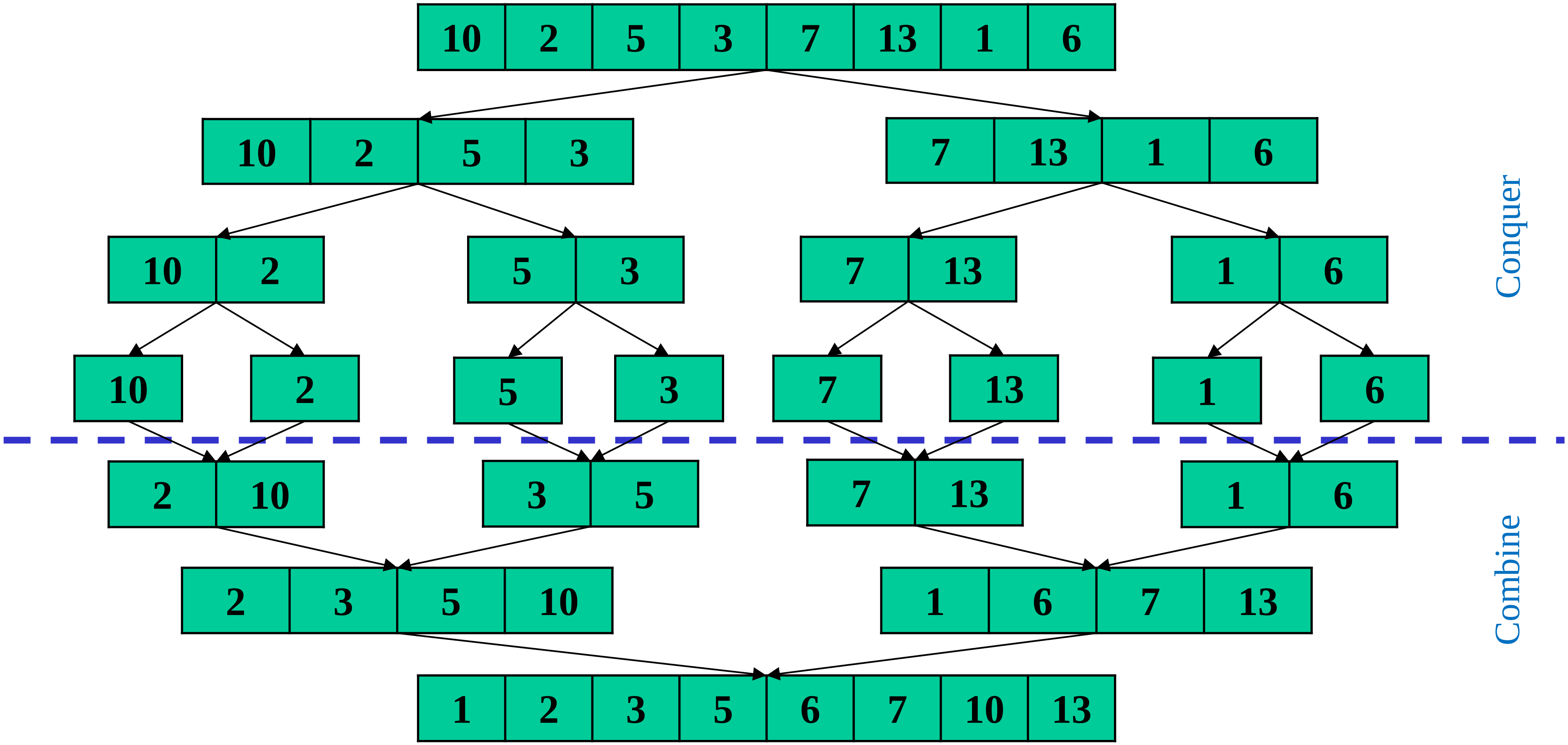
```
mergesort (A[1…n]) {
   if n > 1:
      q = floor(n/2)
      return merge(   mergesort(A[1..q]),
                       mergesort(A[q+1…n])
                    )
   else:
      return A
}
```

```
merge(x[1..p], y[1…q]) {
   if p == 0: return y[1…q]
   if q == 0: return x[1…p]
   if x[1] <= y[1]:
      return x[1] ° merge(x[2…p], y[1..q])
   else:
      return y[1] ° merge(x[1…p], y[2..q])
```

$T(n) = \quad 2T(n/2) \quad + \quad \theta(n)$

Merge does a constant amount of work per recursive call for a total running time of $\theta(p + q)$

Example



10 2 5 3 7 13 1 6
10 2 5 3
7 13 1 6
10 2
5 3
7 13
1 6
10
2
5
3
7
13
1
6
2 10
3 5
7 13
1 6
2 3 5 10
1 6 7 13
1 2 3 5 6 7 10 13
Conquer
Combine

Analysis of Merge Sort

- Recursive tree is a perfect binary tree, height is $\log n$
- At each depth k, need to merge 2^{k+1} sequences of size $n/_{2^{k+1}}$
  - Work at each depth is $\theta(n)$
  - Base case $T(1) = c$

- $T(n) = 2T(n/2) + \theta(n) = cn + n \log n = \theta(n \log n)$
- Best-, average-, worst-case complexity is $\theta(n \log n)$
- Space: $\theta(n)$; $O(1)$ for linked lists

DQ1: findMajority

You are given an array $A[1:n]$ elements. A *majority element* of A is any element that occurs strictly more than $\frac{n}{2}$ times.

If $n = 6\ or\ n = 7$, any majority element will occur in at least four positions.

Assume that the elements cannot be sorted but can only be compared for equality.

Please describe an efficient divide-and-conquer algorithm to find if there is a majority element in A.

What Qs to Ask?

-- Small example?

-- Brute-force algorithm?

-- Sort and find?

-- Different D/Q?

Tiny Testcase

A = [1, 5, 5, 5, 2, 6, 5, 1, 5]; majority element = 5

A = [2, 2, 1, 3]; no majority element

Ideas?

How many majority elements can A have?

One (at most one element can appear > n/2 times)

Split A into A1 and A2

If A has majority element, then it must be a majority element in at least one of A1 and A2

Either subproblem may or may not have a majority element

Our Algorithm

Split A into A1 and A2

If A has majority element, then it must be a majority element in at least one of A1 and A2

Either subproblem may or may not have a majority element

Case 1: No majority in A1 or A2

Case 2: Only A1 has a majority. Count of majority?

Case 3: Only A2 has a majority. Count of majority?

Case 4: Both A1 and A2 have majority.

Working on Tiny Testcase

A = [1, 5, 5, 5, 2, 6, 5, 1, 5]; majority element = 5

A = [2, 2, 1, 3]; no majority element

DQ2: Find Index of the first "1"

You are given an array with n elements that are equal either to 0 or +1 such that all 0 entries appear before +1 entries.

You need to find the index where the transition happens, i.e. you need to report the index with the last occurrence of 0.

Describe an efficient divide-and-conquer algorithm for this task?

Tiny Example

all 0 entries appear before +1 entries → input is sorted!

$A = \{0, 0, 0, 1, 1\}$ Output: 4

$A = \{0, 0, 0\}$ Output: -1

$A = \{1, 1\}$ Output: ?

Brute-Force Algorithm?

Also referred to as “naïve” approach / algorithm

- Iterate over the array elements from $1{:}n$ and return the index of the first “1”
- If there are no “1”s, return -1

- Running time?
  - $O(n)$

- Can we do better?

Ideas?

$$A = \{0, 0, 0, 1, 1\}; \text{mid} = 1 + \frac{5 - 1}{2} = 3; \text{A}[\text{mid}] = \text{A}[3] = 0$$

$$A = \{0, 1, 1, 1, 1\}; \text{mid} = 1 + \frac{5 - 1}{2} = 3; \text{A}[\text{mid}] = \text{A}[3] = 1$$

- Observations
  - Array is sorted, so 0s will be before 1s
  - If the middle element is 0, do I need to look $1{:}\,middle\ - 1$?
    - No, can discard the left half of the array
  - If the middle element is 1, do I need to look $middle\ +\ 1{:}\,n$?
    - No, can discard the right half of the array

Our Algorithm

find_first_one(A, start, end):

- If start > end, return -1
- $mid = start + (end - start)/2$
- If $A[mid]$ is 1 AND $mid == 1$ (i.e., first index in A) or $A[mid-1]$ is 0, then return mid
- Else
  - If $A[mid]$ is 0
    - Search in $A[mid+1: end]$, i.e., $\text{find\_first\_one(A, mid} + 1\text{, end)}$
  - Else
    - Search in $A[1: mid-1]$, $i.e.$, $\text{find\_first\_one(A, start, mid} - 1\text{)}$

- Running time: $T(n) = T\left(\frac{n}{2}\right) + O(1) = O(\log n)$

Notes on D/Q Algo Problems

- Broadly two templates
  - Binary search and variants
  - MergeSort and variants

- Typically, divide in the middle
  - Either conquer further unconditionally (e.g., mergesort, findMajority, …), or
  - Conquer conditionally (e.g., binary search, find first one, …)
  - Either merge is constant time comparison, i.e., O(1) (e.g., binary search, find first one, …), or
  - Merge uses all elements in the merged arrays (for comparison, ..), i.e., O(n) (e.g., mergesort, findMajority, …)

Lecture 4 summary

- D/Q algorithms
- Binary search
- Merge sort
- Two problems
- D/Q strategy