COMP-250 - Lecture 3

More on list intersection & search

a 7 1 4 10
b 3 4 9 1

} common elements (e.g. 2)

For each elem in a → look for it in b

listIntersection → search abstraction

suppose a, b sorted (increasing)

a 1 4 7 10

b 1 3 4 9

If always sorted - should be able to get a faster algorithm!

Binary search

Example:

1 3 7 [8] 9 12 13

val = 3 ↑l ↑r ≤ ↑ ≥

either left or right

val = 7 ↑l ↑r = found

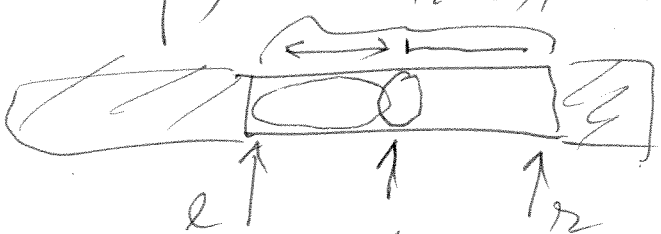
val = 14 Look at some elements but still get right result

Algorithm **binarySearch(a, n, val)**

Input: sorted array a of n elements; val (value we're looking for)

Output: true if val found, false otherwise

```
int l ← 0, r ← n-1
while l ≤ r // array not empty
    int mid ← (l+r)/2
    if a[mid] = val then
        return true
    if a[mid] > val // have to search left of the middle
        r ← mid - 1
    else l ← mid + 1
return false // I was left with an "empty" array to search
```



$mid = l + \frac{r-l}{2} = \frac{r+l}{2}$

Size n	1
Size $\frac{n}{2}$	1
size $\frac{n}{2^2}$	1
size $\frac{n}{2^{\log_2 n}}$	1

(2 comparisons really)

Stop! $\Rightarrow O(\log_2 n)$

n: Size of sorted array

Algorithm listIntersection(a, n, b, m)

Input: Array a of size n, b of size m

Output: return no. of elements in common

⇒ sort(b, m)

Abstraction!!

```
int intersect ← 0
for i ← 0 to n-1
    if binarySearch(b, m, a[i]) then
        intersect ← intersect + 1
return intersect
```

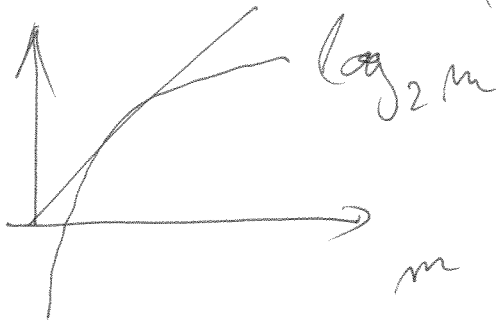
n · running time of binarySearch + sorting time
(size of a) on array b

$n \times \log_2 m$

Previous search: $O(m)$

$n \cdot O(\log_2 m) + O(m \log_2 m) = O((n+m)\log_2 m)$

$= O((m+n)\log_2 m)$



$m, n \simeq$ 1 mil

binarySearch version

$\simeq$ 25000 times faster!

"Look in the middle, then look either left or right

Algorithm binarySearch(a, val, l, r)

Input: Array a, value val, left & right indices between which we search.

// Call binarySearch(a, val, 0, n-1)

Output: same

// Do I have an array still? // Recursion //

if l > r return false "base case"

int mid ← (l+r)/2 // as before

if a[mid] = val then return true

// Search again in one side

if a[mid] > val then // search left

return binarySearch(a, val, l, mid-1)

return binarySearch(a, val, mid+1, r)

There is always an iterative algorithm that does same thing but recursion often

- more natural
- easier to analyze