Lecture 15 -
Makeup for ProgTest1

(≈ 90 minutes)

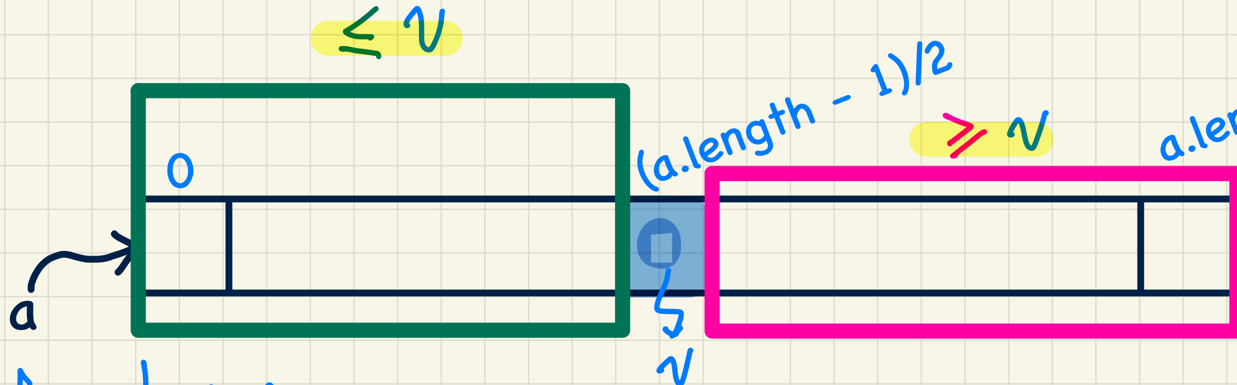
Lecture

Recursion: Part II

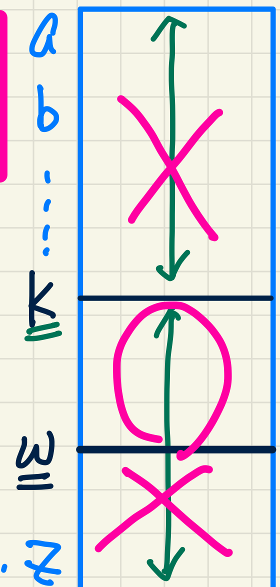
Examples on Recursion
Binary Search

Binary Search: Ideas

Precondition: Array **sorted** in non-descending order



Oxford dictionary



size of search space

n

$\frac{n}{2}$

$\frac{n}{4}$

$\vdots$

1

$\log n$ times to narrow the s.p. to size 1

Search: Does key k exist in array **a**?

① Keep accessing the middle of the search space (halved each time)

② Recur on the left: $k < m$

Recur on the right: $k > m$

Binary Search in Java

input array sorted

```
boolean binarySearch(int[] sorted, int key) {
  return binarySearchH(sorted, 0, sorted.length - 1, key);
}
boolean binarySearchH(int[] sorted, int from, int to, int key) {
  if (from > to) { /* base case 1: empty range */
    return false; }
  else if(from == to) { /* base case 2: range of one element */
    return sorted[from] == key; }
  else {
    int middle = (from + to) / 2;
    int middleValue = sorted[middle];
    if(key < middleValue) {
      return binarySearchH(sorted, from, middle - 1, key);
    }
    else if (key > middleValue) {
      return binarySearchH(sorted, middle + 1, to, key);
    }
    else { return true; }
  }
}
```

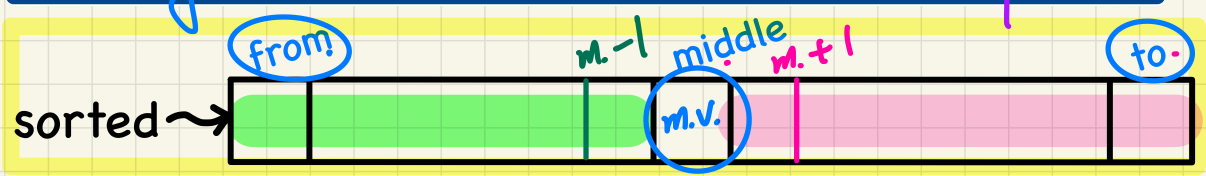
call by value

define the range of indices of the search space.

recursive case

key == middleValue

narrowed search spaces represent a ~~strictly~~ smaller problem to solve.



Binary Search: Tracing

a

0	1	2	3	4	5	6	7	8
3	6	9	12	15	18	21	24	27

initial search space

search space of 1st recursion

search space of 2nd recursion

key

search(a,18)

binarySearchH(a,0,8,18)

$m. = \frac{(0+8)}{2} = 4$

m.v. $a[4] == 15$

$>$

binarySearchH(a,5,8,18)

m.+1

fr.

$m. = \frac{(5+8)}{2} = 6$

m.v. $a[6] = 21$

binarySearchH(a,5,5,18)

↳ true

from

m.-1

Exercise

a

0	1	2	3	4	5	6	7	8
3	6	9	12	15	18	21	24	27

search(a,7)

binarySearchH(a,0,8,7)

binarySearchH(a,0,3,7)

binarySearchH(a,2,3,7)

binarySearchH(a,2,1,7)

Running Time: Ideas

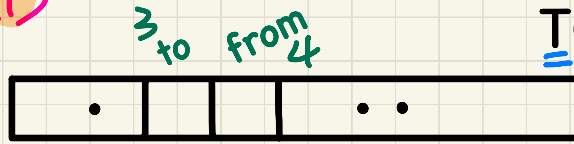
Recurrence Relation

```
boolean allPositive(int[] a) { return allPosH(a, 0, a.length - 1); }
boolean allPosH (int[] a, int from, int to) {
  if (from > to) { return true; }
  else if (from == to) { return a[from] > 0; }
  else { return a[from] > 0 && allPosH (a, from + 1, to); } }
```

O(1) O(1) O(1)

subproblem of size n-1

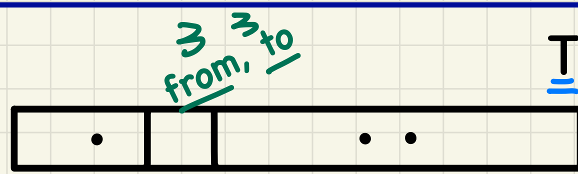
Base Case:
Empty Array



$T(0) = 1$

[4, 3] has 0 numbers

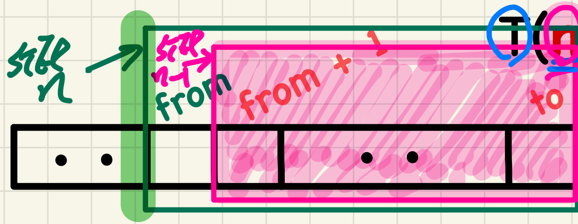
Base Case:
Array of Size 1



$T(1) = 1$

[3, 3] has 1 number

Recursive Case:
Array of size > 1



$T(n) = T(n - 1) + 1$

strictly smaller problem

Running Time: Unfolding Recurrence Relation

$$T(0) = 1$$
$$T(1) = 1$$
$$T(n) = T(n-1) + 1$$

recurrence relation derived from Java imp. of recursive algorithm.

$$T(n) = T(n-1) + 1$$

$T(n-1)$

$$= T((n-1)-1) + 1 + 1$$

$n-2$

$T(n-2)$

$$= T((n-2)-1) + 1 + 1 + 1$$

$n-3$

$n-(n-1)$

$$= \dots$$

$$= T(1) + \underbrace{1 + 1 \dots + 1}_{\text{How many? } (n-1)}$$

$= 1$

$$\therefore T(n) = (n-1) + 1$$
$$= n$$

$$O(n)$$

Binary Search: Running Time

```java
boolean binarySearch(int[] sorted, int key) {
  return binarySearchH(sorted, 0, sorted.length - 1, key);
}
boolean binarySearchH(int[] sorted, int from, int to, int key) {
  if (from > to) { /* base case 1: empty range */
   return false; }  O(1)
  else if(from == to) { /* base case 2: range of one element */
   return sorted[from] == key; }  O(1)
  else {
   int middle = (from + to) / 2;
   int middleValue = sorted[middle];
   if(key < middleValue) {
    return binarySearchH(sorted, from, middle - 1, key);
   }
   else if (key > middleValue) {
    return binarySearchH(sorted, middle + 1, to, key);
   }
   else { return true; }
  }
}
```

Running Time as a Recurrence Relation

$T(0) = 1$

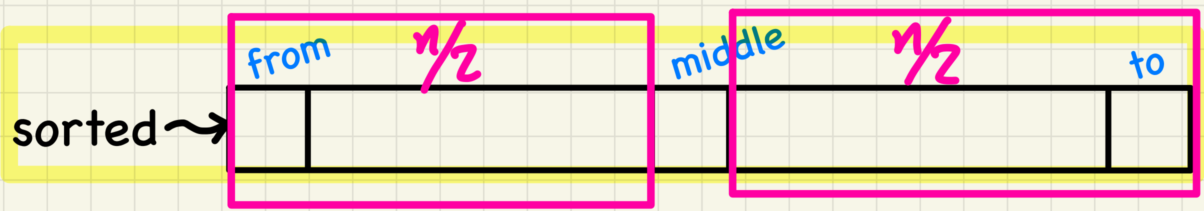
$T(1) = 1$

$T(n) = T(n/2) + 1$

wrong:

$T(n) = \underbrace{T(\frac{n}{2})}_{L} + \underbrace{T(\frac{n}{2})}_{R}$ X

!! either L or R but not both



Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

$T(n) = T(n/2) + 1$

once reaching here, no more unfoldings

Assume : $n = 2^x$ for $x \geq 0$

↳ without loss of generality.

$2^{\log 8} = 2^3 = 8$

$\frac{n}{2^{\boxed{\log n}}} = \frac{n}{n}$

$T(n) = T\left(\frac{n}{2}\right) + 1$

$T\left(\frac{n}{2}\right)$

$\frac{n}{2^1} = \left(T\left(\frac{n}{4}\right) + 1\right) + 1$

$T\left(\frac{n}{4}\right)$

$\frac{n}{2^2} = \left(T\left(\frac{n}{8}\right) + 1\right) + 1 + 1$

$T\left(\frac{n}{8}\right)$

$\frac{n}{2^3} = \left(T\left(\frac{n}{16}\right) + 1\right) + 1 + 1 + 1$

$\frac{n}{2^4}$

$\vdots$

$= T(1) \underbrace{+1 + 1 \cdots + 1}_{\text{How many? } \log n}$

$O(\log n)$