Lecture 24

Monday December 2

Inserting into an Array

$O(n+1) = O(n)$

```
String[] insertAt(String[] a, int n, String e, int i)
  String[] result = new String[n + 1];
  for(int j = 0; j <= i - 1; j ++){ result[j] = a[j]; }
  result[i] = e;
  for(int j = i + 1; j <= n - 1; j ++){ result[j] = a[j-1]; }
  return result;
```

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

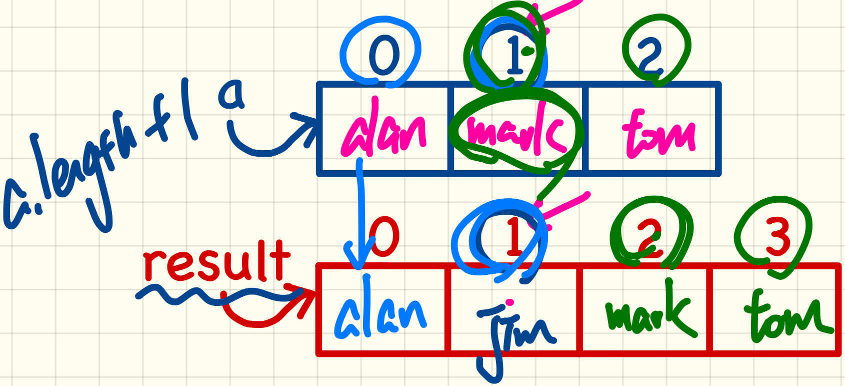
$0 \,..\, i-1 \quad i \quad i+1 \,..\, n-1 \quad n$

$(n-1)-(i+1)+1$

$i+1$ f

Example:

insertAt({alan, mark, tom}, 3, jim, 1)



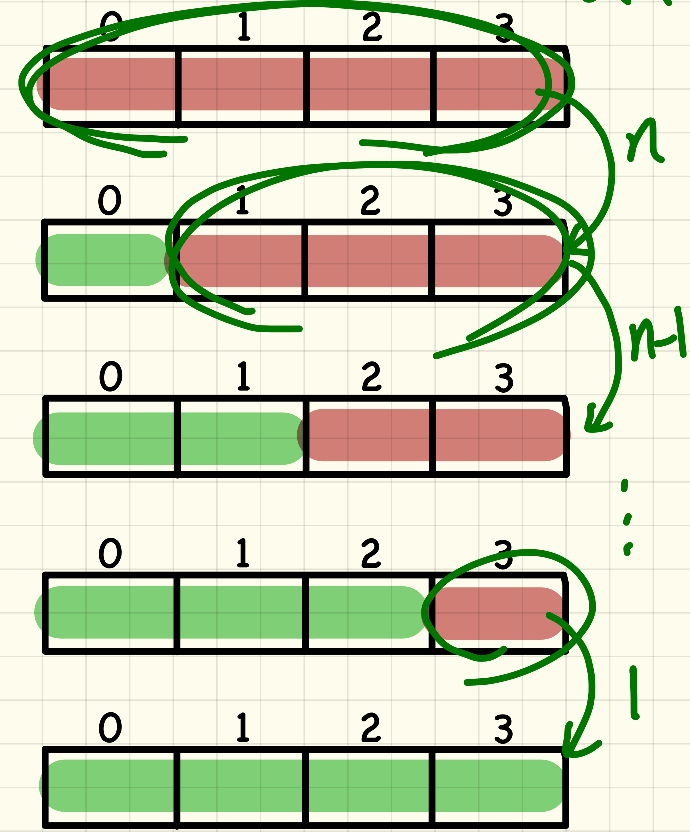
result[2] = a[1]

[3] = a[2]

Selection Sort

$O(n + (n-1) + (n-2) + \dots + 1)$

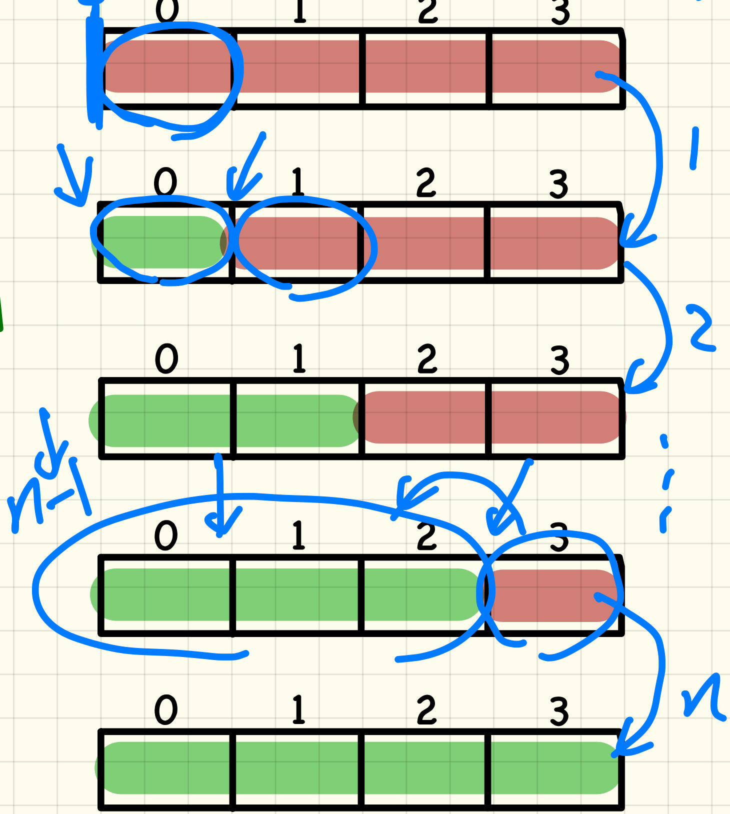
$O(n^2)$



Insertion Sort

$O(1 + 2 + 3 + \dots + n)$

$= O(n^2)$



$O(n^2)$

Input size 1000

$\hookrightarrow (1000)^2 = 1M$

$O(n \cdot \log n)$

$\hookrightarrow 1000 \cdot \log 1000 = 1000 \cdot 10 =$

$= 10000$

Selection Sort in Java

$O((n-1) + (n-2) + \cdots + 2)$

$O\left(\frac{((n-1)+2) * (n-2)}{2}\right)$

```
selectionSort(int[] a, int n)
  for (int i = 0; i <= (n - 2); i ++)
    int minIndex = i;
    for (int j = i; j <= (n - 1); j ++)
      if (a[j] < a[minIndex]) { minIndex = j; }
    int temp = a[i];
    a[i] = a[minIndex];
    a[minIndex] = temp;
```

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

$O(n^2)$

$i = 0$: $j = 0 \; 1 \; .. \; n-1$ → $n-1$

1: $j = 1 \; 2 \; ... \; n-1$ → $n-2$

2: $j = 2 \; 3 \; ... \; n-1$ → $n-3$

$\vdots$

$n-2$: $j = n-2 \; .. \; n-1$ → 2

$(n-1)$

Insertion Sort in Java

```
insertionSort(int[] a, int n)
  for (int i = 1; i < n; i ++)
    int current = a[i]; O(1)
    int j = i;
    while (j > 0 && a[j - 1] > current)
      [a[j] = a[j - 1]; O(1)]
      j --;
    a[j] = current; O(1)
```

$i = 1$ $j = 1$ $O(1 + 2 + 3 + \ldots +$

2 $j = 2 \quad 1$

3 $j = 3 \quad 2 \quad 1$ $= O\left(\frac{(1 + (n-1)) \times (n-1)}{2}\right)$

$n-1$ $j = n-1 \quad n-2 \quad n-3 \quad \ldots \quad 1$ $O(n^2)$.

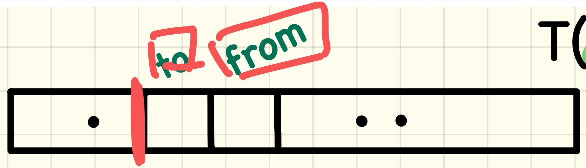
Running Time: Ideas

running time ← T(t) → input size.

```
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); } }
```

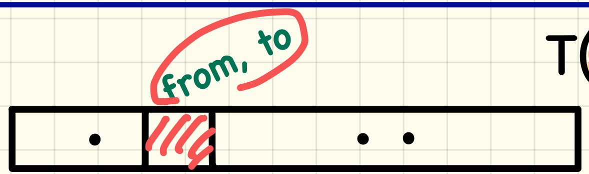
input size

Base Case:
Empty Array



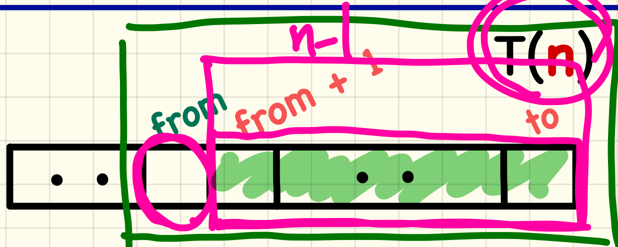
$T(0) = 1$

Base Case:
Array of Size 1



$T(1) = 1$

Recursive Case:
Array of size > 1



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

Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

$\checkmark\ T(\cancel{n}) = T(\cancel{n} - 1) + 1$

$n-1 \qquad n-1$

$n-2 \qquad n-2$

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

$$= \underbrace{(T(n-2) + 1)}_{T(n-1)} + 1$$

$$= ((T(n-3) + 1) + 1) + 1$$

$T(n-n)$

$$= \ldots$$

$$= (\underbrace{T(0)}_{T(1)} + 1) + \cdots + 1 + 1 + \underbrace{1}_{n \text{ terms.}}$$

$O(n)$.

Correctness Proofs: Ideas

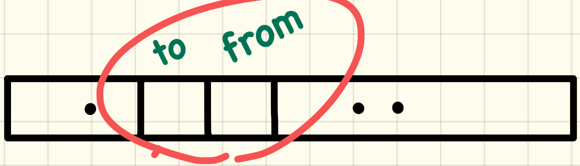
```
boolean allPositive(int[] a) { return allPosH (a, 0, a.length - 1); }
boolean allPosH (int[] a, int from, int to) {
  if (from > to) { return true; }
  else if(from == to) { return a[from] > 0; }
  else { return a[from] > 0 && allPosH (a, from + 1, to); } }
```

assumed to be correct

I.H.

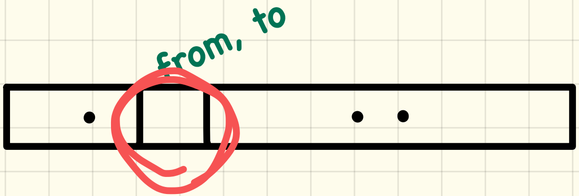
Base Case:

Empty Array



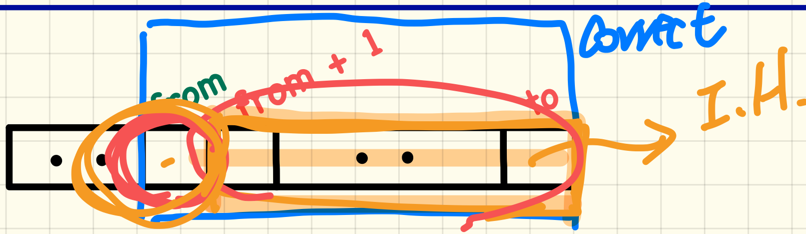
Base Case:

Array of Size 1



Recursive Case:

Array of size > 1



Correctness Proofs

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

- Via mathematical induction, prove that `allPosH` is correct:

  Base Cases

  - In an empty array, there is no non-positive number ∴ result is ***true***. **[L3]**
  - In an array of size 1, the only one elements determines the result. **[L4]**

  Inductive Cases

  - **Inductive Hypothesis**: `allPosH(a, from + 1, to)` returns ***true*** if a[from + 1], a[from + 2], . . . , a[to] are all positive; ***false*** otherwise.
  - `allPosH(a, from, to)` should return ***true*** if: **1)** a[from] is positive; and **2)** a[from + 1], a[from + 2], . . . , a[to] are all positive.
  - By *I.H.*, result is $a[from] > 0 \land$ `allPosH(a, from + 1, to)`. **[L5]**

- `allPositive(a)` is correct by invoking `allPosH(a, 0, a.length - 1)`, examining the entire array. **[L1]**

Entire range of array.

Binary Search: Ideas

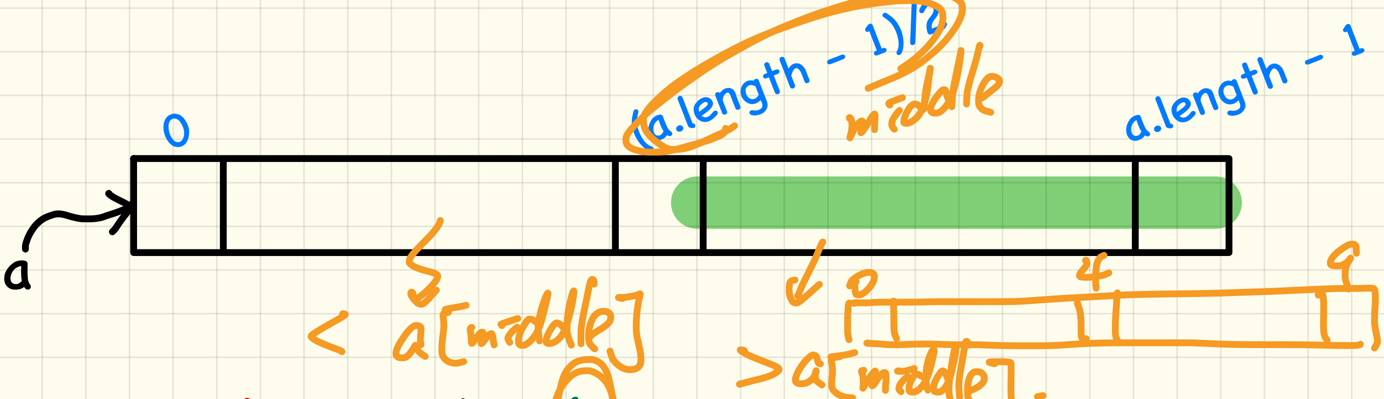
n

search(k)

k   O(n)

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

1 2 4 6 7 8 - -



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

$\boxed{k > a[middle]}$

Binary Search in Java

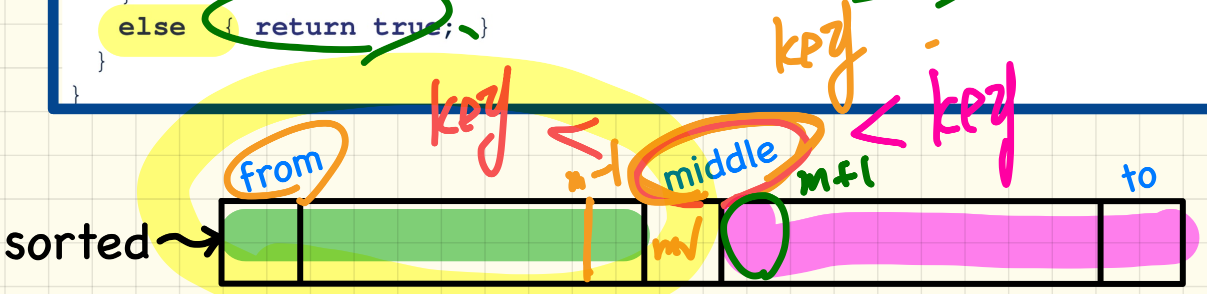
```java
boolean binarySearch(int[] sorted, int key) {
  return binarySearchHelper(sorted, 0, sorted.length - 1, key);
}
boolean binarySearchHelper(int[] sorted, int from, int to, int key
  if (from > to) { /* base case 1: empty range */
   return false; }
  else if(from == to) { /* base case 2: range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

O(1)

T(0) = 1

T(1) = 1

O(1)



Binary Search: Tracing

0 1 2 3 4 5 6 7 8

m

Say a = {3,6,9,12,15,18,21,24,27}

18 > a[4]

search(a,18)

f t

search(a,0,8,18)

m = (0+8)/2 = 4

search(a,5,8,18)

m = (5+8)/2 = 6

18 < a[6]

search(a,5,5,18)

0 1 2 3 4 5 6 7 8

Say a = {3,6,9,12,15,18,21,24,27}

search(a,7)

search(a,0,8,7)

search(a,0,3,7)

search(a,2,3,7)

search(a,2,1,7)

Binary Search: Running Time

```
boolean binarySearch(int[] sorted, int key) {
  return binarySearchHelper(sorted, 0, sorted.length - 1, key);
}
boolean binarySearchHelper(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 binarySearchHelper(sorted, from, middle - 1, key);
    }
    else if (key > middleValue) {
      return binarySearchHelper(sorted, middle + 1, to, key);
    }
    else { return true; }
  }
}
```

$$T(0) = 1$$
$$T(1) = 1$$
$$T(n) = T(n/2) + 1$$

Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

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

$\frac{n}{2}$ $\quad$ $\frac{n}{2}$

$\frac{n}{4}$ $\quad$ $\frac{n}{4}$

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

$1 = \frac{n}{2^{\log n}}$

$O(\log n)$.

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

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

$$= \left(\left(T\left(\frac{n}{8}\right) + 1\right) + 1\right) + 1 \qquad 2^3$$

$$\vdots$$

$$1 \qquad \frac{n}{2^{\log n}}$$

$$= \underbrace{T(1) + 1 + \cdots + 1 + 1 + 1}_{\log n}$$