Search

Input: a set $A$ (array, list, ...) of elements and an element $x$

Output: YES if $x$ is in $A$, NO otherwise

possibly + position of $x$
Linear Search

**Input:** an array $A$ and an element $x$

**Output:** position of $x$ if it is in $A$, or -1 otherwise

**Pseudocode:**

```plaintext
i = 0
while ((i < n) & (A[i] != x)) {
    i++
}
if (i == n) then output -1
else output i
```

**Running time:** $O(n)$ in the worst case (e.g. if $x$ is not in the array)
Binary Search

Input: a sorted array $A$ and an element $x$

Output: position of $x$ if it is in $A$, or -1 otherwise

High-level pseudocode:

1. look at the middle element, let it be $y$
2. if $x < y$, continue with the same algorithm in the left "half" $A[0...posy-1]$ 
3. if $x > y$, continue with the same algorithm in the right "half" $A[posy+1...n-1]$
4. if $x = y$, output the position of $y$

Note: this means $k = \log_2 n$ every iteration is a constant number of steps.

Run-time/ time complexity:
$O(\log n)$

Some remarks:
- Can't copy the array content into smaller arrays, but can keep the beginning and end index of the part of the (original) array currently under consideration.

Suggested algorithm:
- Stop if $x$ is the element in the array, then output the position of $x$.
- Else output -1.

Examples:
- $A = [1, 3, 7, 10, 11, 12]$
- $x = 11$
- $y = \frac{n}{2^k}$ elements to look at
- Stop if $x$ found or if $\frac{n}{2^k} = 1$

Some additional notes:
- As $n = 2^k$, then $\frac{n}{2^k} = 1$.
**Binary Search**

**Input:** a sorted array $A$ and an element $x$

**Output:** position of $x$ if it is in $A$, or -1 otherwise

**Detailed pseudocode - attempt 1:**

1. $left = 0$;
2. $right = n-1$;
3. while ($left < right$) do {
4.   $middle = \text{roundDown}((left+right)/2)$;
5.   if ($x \geq A[middle]$) $left = middle$;
6.   else $right = middle$;
7. }
8. if ($A[left] == x$) output $left$;
9. else output -1
Binary Search

Input: a sorted array $A$ and an element $x$

Output: position of $x$ if it is in $A$, or -1 otherwise

Detailed pseudocode - attempt 2:

1. $left = 0$;
2. $right = n-1$;
3. while ($left < right$) do {
4.   $middle = \text{roundDown}((left+right)/2)$;
5.   if ($x > A[middle]$) $left = middle$;
6.   else $right = middle$;
7. }
8. if ($A[left] == x$) output $left$;
9. else output -1
**Binary Search**

**Input:** a **sorted** array $A$ and an element $x$

**Output:** position of $x$ if it is in $A$, or -1 otherwise

**Detailed pseudocode - attempt 3:**

1. left = 0;
2. right = n-1;
3. while (left < right) do {
4.    middle = roundDown((left+right)/2);
5.    if ($x \geq A[middle]$) left = middle+1;
6.    else right = middle;
7. }
8. if ($A[left] == x$) output left;
9. else output -1
Binary Search

Input: a sorted array $A$ and an element $x$

Output: position of $x$ if it is in $A$, or -1 otherwise

Detailed pseudocode - attempt 4:

1. $left = 0$; 
2. $right = n-1$; 
3. while ($left < right$) do { 
4.  $middle = \text{roundDown}((left+right)/2)$; 
5.  if ($x > A[middle]$) $left = middle + 1$; 
6.  else $right = middle$; 
7. } 
8. if ($A[left] == x$) output $left$; 
9. else output -1
Binary Search

Moral of the story:

(I.e., how to test a code and how to prove its correctness)

**TEST WITH ALL SPECIAL CASES**

IN BIN.SEARCH:

- small size array 0, 1, 2, 3
- A FEW RANDOM CASES, SOME LARGE
  - e.g. an array of size 1000
    - with x in the array
      - any position
      - first/last/middle
    - with x not in the array
      - x is smaller than the smallest
      - x is larger than the largest
      - x somewhere in between but not in A

**Checking that no infinite loops occur**
(typically through finding a “shrinking” quantity)

- checking that the program behaves according to the specifications/expectations
  
  e.g. for the bin.search, we verified that
  
  "if x is in A, then in every iteration, x is between left and right"
Introduction to Recursion

Implementing linear search recursively

Pseudocode:

Input: array A, element x
Output: YES/NO depending on if x ∈ A

1. if the size of A is 0 then return NO
2. if A[0] == x then return YES
3. solve the same problem with A[1...n-1]

A

Disclaimer: 1) most people would use for implementation of linear search, this is just a toy example (although it's very natural for functional languages)
2) if x ∈ A then we need n recursive calls and in the i-th call we use O(n-i) steps

Running time:

Attempt 1 at detailed implementation:

```
linSearchRec (array A, element x) {
    1. if size of A == 0 then return NO
    2. if A[0] == x then return YES
    4. return linSearchRec (B, x)
}
```

example: A = 5 7 3 8 4 x = 3

B = 7 3 8 4 returns YES (first)
newA = 7 3 8 4 x = 3
newB = 3 8 4 returns YES (second)
newA = 3 8 4 x = 3
newB = 8 4 returns YES (fourth)
newA = 8 4 if x = 6 we continue until size of newA is 0, return NO
Introduction to Recursion

Implementing binary search recursively

Running time: