COMP10002 Workshop Week 6

	lec05.pdf : what's there?
1	Arrays & Pointers Binary Search: Exercise 1 (at the end of lec05.pdf) Quicksort: Exercise 4 (at the end of lec05.pdf)
2	Main Room: do 7.8 together BreakOut Rooms: do LAB
LAB	7.3, 7.4, 7.6, 7.7, 7.8, 7.9, 7.10, and 7.11;
Canvas	Workshops: • Discuss Exercises 1 and 4 at the end of the lec05.pdf slides; then • Continue on with the Chapter 7 exercises from last week, you need to be "array-peaking" this week ready to tackle Assignment 1 next week; • ASSIGNMENT 1

Topics in lec05.pdf

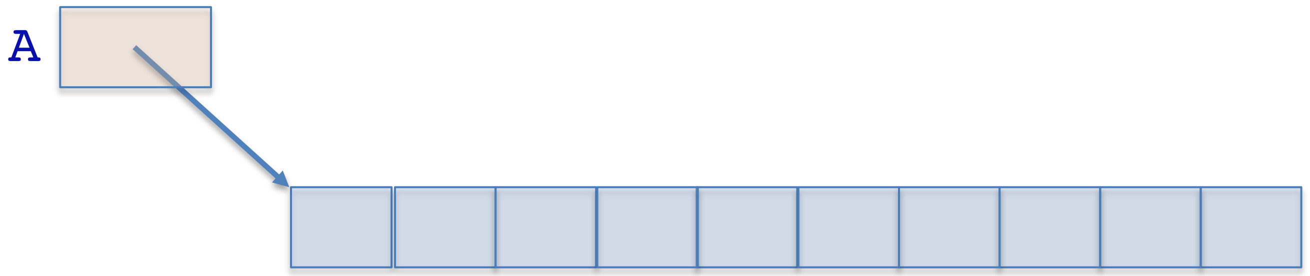
- Arrays:
  - Array – function - pointer
  - 2D arrays
  - Array Search
  - *Linear Search* and Correctness (using assertions)
- Defining *efficiency* with *big-O*, efficiency of linear search
- *Binary Search* (in sorted arrays):
  - Algorithm & Correctness
  - Recursive binary search
  - Binary Search Efficiency
  - Note: C convention: function `int cmp(data_t *x, data_t *y)`
- Sorting: *Quick Sort* (with `fe`, `fg`):
  - algorithm
  - partition & analysis
  - picking *pivot*: random is good

array name is a *pointer*

```
int A[10], n=0;
```

Here the name `A` :

- represents the address of the first element of the block of 10 integer cells reserved for the array → `A` is a pointer

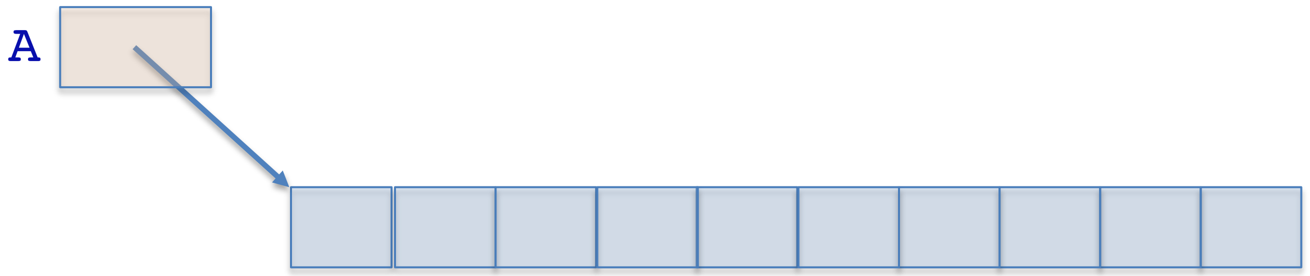


array name is a *constant pointer*

```
int A[10], B[10], n=0;
```

Here the name `A` :

- represents the address of the first element of the block of 10 integer cells reserved for the array → `A` is a *constant* pointer



- We cannot change `A`, for example by (wrong) `A= &n;`, or by `A= B;`
- but we can change the value at address `A`, we can change the values in the array.

Chorus: “array names are constant pointers”

With declaration:

```
#define MAX_N 1000
int A[MAX_N]= {10, 20, 30}, n= 3, i= 1;
int *p;    // p is a variable pointer
p= A;      // p now points to A[0], just like A
```

- now variable `p` can be used as a substitute for constant `A`
- `p` ⇔ `A`,   `p[i]` ⇔ `A[i]`

  but `p++` is valid, while `A++` isn’t

Array Notation and Pointer Notation
“array names are constant pointers”

With declaration:

```
#define MAX_N 1000
int A[MAX_N]= {10, 20, 30}, n= 3, i= 1;
int *p= A;
```

- `*A` and `*p` both equivalent to `A[0]`
- `A+1`, as well as `p+1`, points to the next int cell, ie. to `A[1]`

- `*(A+i)`, as well as `*(p+i)`, is `A[i]`

Check your understanding

1. What is the value of `*(A+2)`?
2. What will be printed by the loop:

```
for (i=0, p=A; i<n; i++, p++) {
      printf(“%d “, *p);
}
```

Searching in an array: Linear Search Revisited

The Task: searching for `x` in an array `A[ ]` of `n` elements.

Input: `?`

Output: `?`

Complexity of linear search: ?

Linear Search: search for x in array A of n element

```
algorithm= ???
```

Searching in an array: Linear Search

The Task: searching for `x` in an array `A[]` of `n` elements.

Input: (`A`, `n`), `x`

Output:

- index `i` such that `A[i]==x`, OR
- `NOTFOUND` (`#define`-ed as `-1`)

Complexity of linear search:

- O(n)

Searching in an array: Binary Search in Sorted Array

The Task: searching for `x` in a *sorted* array `A[]` of `n` elements.

Input: (`A`, `n`), `x`

Output:

- index `i` such that `A[i]==x`, OR
- `NOTFOUND` (`#define`-ed as `-1`)

Complexity of binary search:

- O(?)

Binary Search: Recursive Algorithm

The Task: searching for `x` in a *sorted* array `A[ ]`

Recursive Binary Search: search for x in an array A which is sorted in non-decreasing order

```
int binSearch(int A[], int n, int x) {
   if (base cases) {
   }
   // general case



}
```

Exercise 1

Suppose that the sorted array may contain duplicate items. Linear search will find the first match.

Modify the binary search algorithm so that it also identifies the first match if there are duplicates.

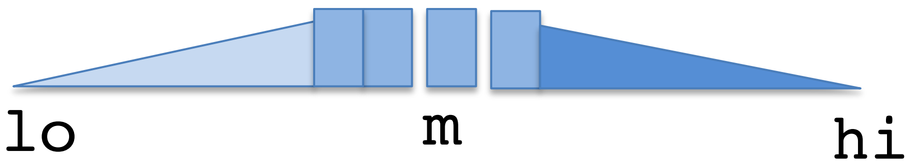
Modify the demonstration of correctness so that it matches your altered algorithm.

Modifying the following algorithm:

The original algorithm: searching for `x` in the sorted array `A[lo..hi]`

```
lo,hi ← 0,n
while lo < hi
    m ← (lo + hi)/2
    if x < A[m]
        hi ← m
    else if x > A[m]
        lo ← m + 1
    else
        return m
return not_found
```

Modification



…

Question: Modify the above binary search algorithm so that it also identifies the first match if there are duplicates.

Correctness: Original → New Version

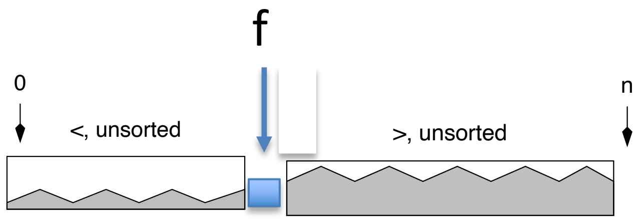
```
m ← (lo + hi)/2
assert: P and lo≤m<hi
if x < A[m]
    assert: P and x < A[m . . . n − 1]
    hi ← m
    assert: P
else if x > A[m]
    assert: P and x > A[0...m]
    lo ← m + 1
    assert: P
else
    assert: P and x ≮ A[m] and x ≯ A[m] ⇒ x = A[m]
    return m
```

Question: Modify the the above (for binary search algorithm) to fit the new version of returning the first match.

Exercise 4: on the partitioning of the quicksort

A slightly different approach to partition is described by this more relaxed specification:

```
assert: n > 1 and p ∈ A[0 . . . n − 1]
f ← partition(A, n, p)
assert: 0 < f < n − 1 and
        A[0 . . . f − 1] ≤ p and
        A[f ] = p and
        A[f + 1 . . . n − 1] ≥ p
```

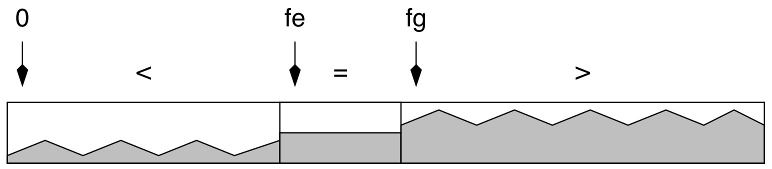


4a. Design a function that meets this specification.

4b. Does this approach have any disadvantages or advantages compared to the first one presented?

Quicksort: The original logic with fe, fg

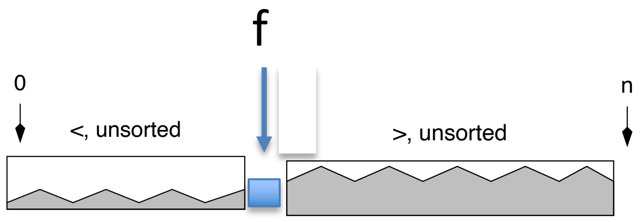
```
define R ≡ 0 ≤ fe < fg ≤ n and A[0 . . . fe − 1] < p and
A[fe . . . fg − 1] = p and A[fg . . . n − 1] > p

if n ≤ 1
   return

p ← any element in A[0 . . . n − 1]

assert: n>1 and p∈A[0...n-1]
(fe, fg) ← partition(A, n, p)
assert: R
quicksort (A[0 . . . fe − 1])
quicksort (A[fg . . . n − 1])
assert: A[0...fe-1]is sorted and A[fg...n-1]is sorted and
R ==> A[0...n-1] is sorted
```



Exercise 4

A slightly different approach to partition is described by this more relaxed specification:

```
assert: n > 1 and p ∈ A[0 . . . n − 1]
f ← partition(A, n, p)
assert: 0 < f < n − 1 and
        A[0 . . . f − 1] ≤ p and
        A[f ] = p and
        A[f + 1 . . . n − 1] ≥ p
```

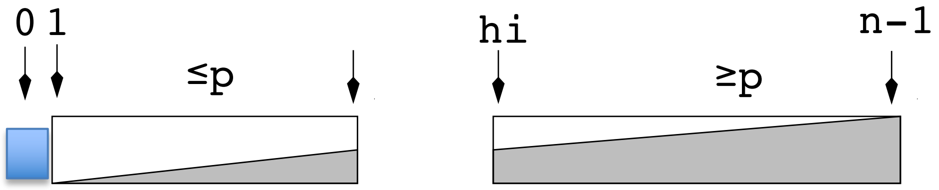


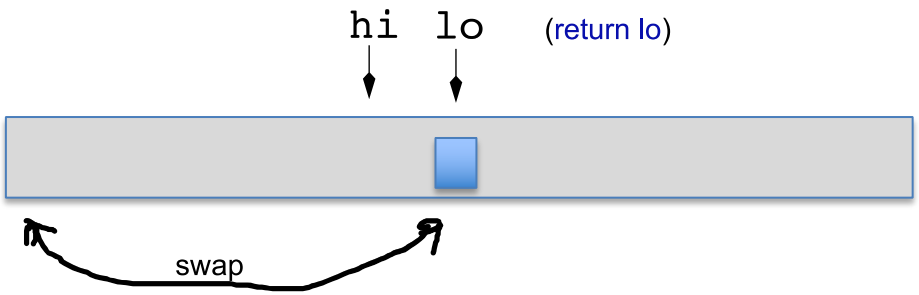
4a. Design a function that meets this specification.

4b. Does this approach have any disadvantages or advantages compared to the first one presented?

4a. Design a function that meets this specification.

```
assert: n > 1
        f ← partition(A, n)
        p ← A[f]
assert: A[f] = p and A[0 . . . f − 1] ≤ p and
                     A[f + 1 . . . n − 1] ≥ p
```





Hoare's Partitionning

```
int partition(int A[], int n) {
   p= ?; lo=?, hi=?
}
```

4b. Does this approach have any disadvantages or advantages compared to the first one presented?

- The first version: partitioning into 3 segments: <p, ==p, and >p
- The new version: Partitioning into 3 segments: <=p, ==p, and >=p and the ==p segment has only one element.

- So ???:
  - is there any difference on complexity?
  - is there any difference on simplicity?
  - which one do you prefer?

Lab: Group+Individual Works: Ex 7.7-7.9 and others.

Write a *function* that

7.2: [W06] sorts an array in deceasing order

7.3: [W06] sorts an array and removes duplicates

7.4: [W06] computes frequency of each value in an array

7.6: [W06] performs selection sort

7.7: [W05]

array `A` of i

may not be

7.8: [W5X]

7.9: [W05]

For examp

Main Room	Break-out Rooms
Together with Anh, do (relatively low speed): • exercise 7.8, or • any exercise you want	Do as many exercises as possible. Summon Anh when having questions

7.10: [W5X] returns the number of inversions in `int A[]`. For example, the above array has 3 inversions: 2 caused by **`15`**, and 1 by **`21`**.

`***` and all other exercises from `W05, W05X, W06, W06X`

Wrap-Up

Practice with processing arrays!
Know the algorithms:

- Sequential Search
- Binary Search
- Insertion Sort
- Quick Sort

Assignment 1:

- Start early (ASAP)
- *Do it incrementally!*
- Be active in Discussion Forum
- Ask Questions
- By workshop next week: At least finish the first stage.