COMP10002 Workshop Week 6

Cut in half, the Search runs fast, Quicksort sorts the lot at last!

Topics:

1. 2D-arrays ?
2. Recursive Functions?
3. Binary Search, discuss Ex 1 (at the end of lec05.pdf)
4. Quick Sort: discuss Exercise 4 (at the end of lec05.pdf)

LAB:

- array exercises from Chapter 7

Don't Miss:

Assignment 1 will be released this Wed.

- Make sure to explore and start *before* the workshop next week.

2D-arrays [recall: an array name is a pointer constant]

Each data type has a corresponding pointer type, for example

```
int A[5],  B[5][4];
//      B is an  array of 5      "array of 4 int"
//      B is a   pointer to      pointer-to-int
int *pa, **pb;
//      pb is a "pointer to pointer-to-int"
pa = A;        ✓ ✗     OK or not-OK  ?
pa = A+3;      ✓ ✗
pb = B;        ✓ ✗
pb = &B[0];    ✓ ✗
pb = &B[0][0]; ✓ ✗
A = A+3;       ✓ ✗
```

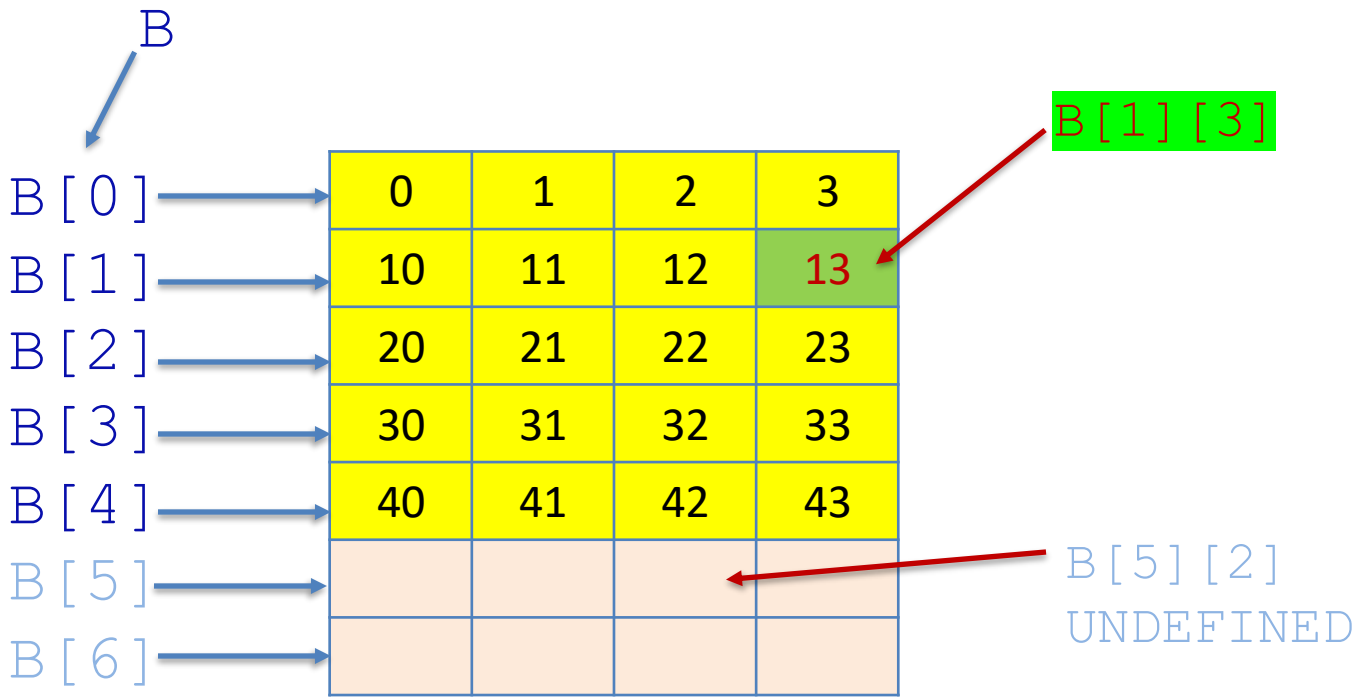
2D array – as we normally understand

int **B[5]**[4] = {{0,1,2,3},{10,11,12,13},…};

B is an array of 5 "array of 4 int"

B[0] is an array of 4 int: {0,1,2,3}

and similarly for B[1], B[2], B[3]

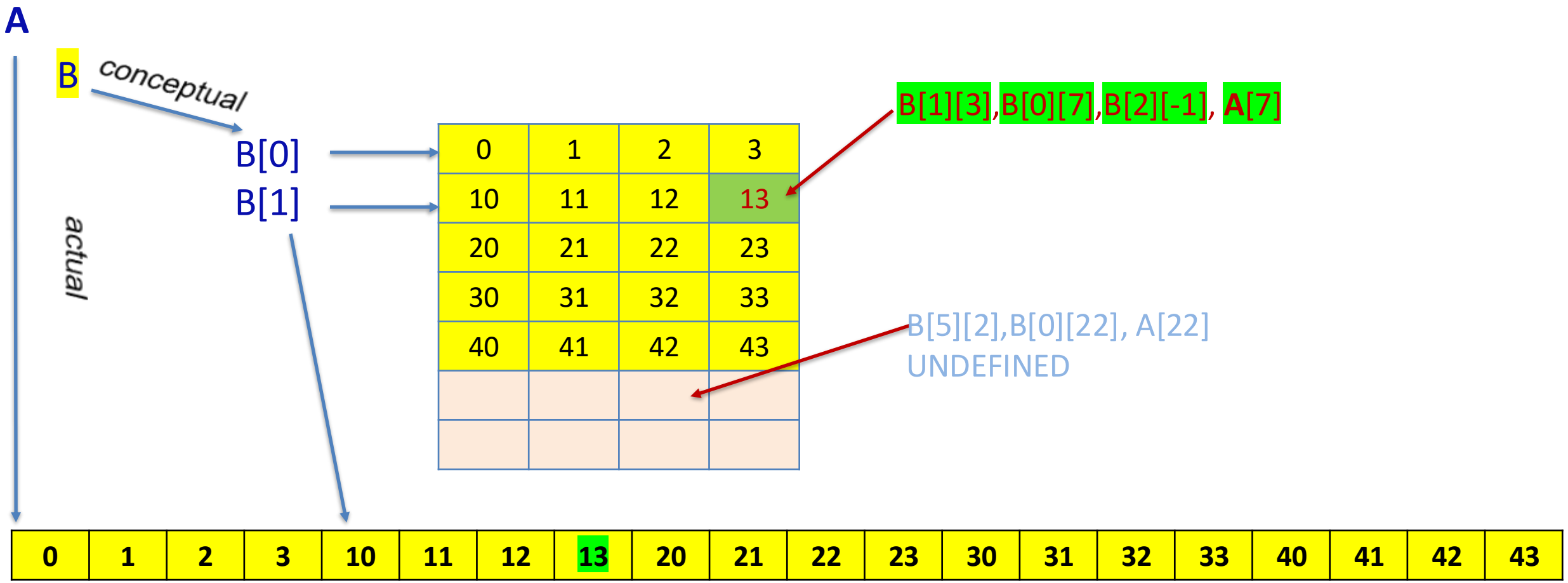


? what if we do:

```
B[5][2]= 777;
```

if A is an 1D array, i is an int, A[i] is valid, but might be undefined!
if B is an 2D array, B[i][j] is valid, but **undefined**!

```
int B[5][4]={{0,1,2,3},{10,11,12,13},{20,21,22,23},{30,31,32,33},{40,41,42,43}};
int *A= &B[0][0];
```



example

```
int A[5]={4,3,5,2,1};
int B[3][5]
  ={{1,12,3,14,5},{15,4,6,2,13},{10,7,8,11,9}};
```

What's the value of

1. B[A[4]][A[1]]
2. B[A[4]][A[2]]
3. B[A[1]][A[2]]
4. B[A[3]][A[2]]

Recursive functions

Recursive function: function that calls itself

```
int factorial( int n ) {
  if (n==1) {
    return 1;
  }
  return n*factotial(n-1);
}
```

Recursive functions: How

- general case: reduce the task of size n to the same tasks of smaller sizes
- clearly describe the base cases where the solutions are trivial
- when writing code, start with *solving the base cases first*

Example1: factorial (n)

- general case: factorial(n) can be computed by using factorial(n-1)
- base case: when n==1 the solution is trivial
- writing code:

```
int factorial( int n ) {
  if (n==1) { // base case
    return 1;
  }
  // general case   [note: else is not needed]
  return factorial(n-1) * n;
}
```

Example 2: **5.13**: int pow(int m, int n)

- the task: returns m^n (m>0, n >=0)
- what's: relationship between the input size a smaller size?
- what's: the base case?

Other Example: Recursive Binary Search

Recursive Selection Sort:

General Case:

Base Case:

Binary Search

.

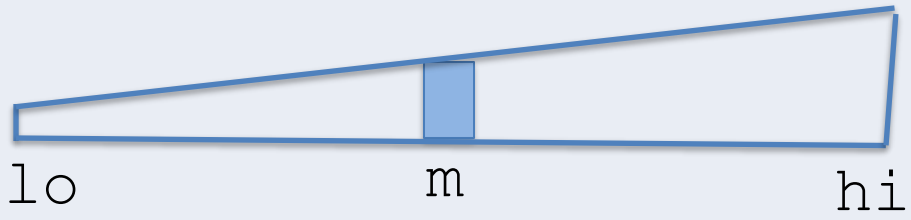
Input: sorted array `A[]`, `n`, `x`

Output:

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

`Recursive Algorithm:`

Iterative Algorithm




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

lec05 - Exercise 1

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

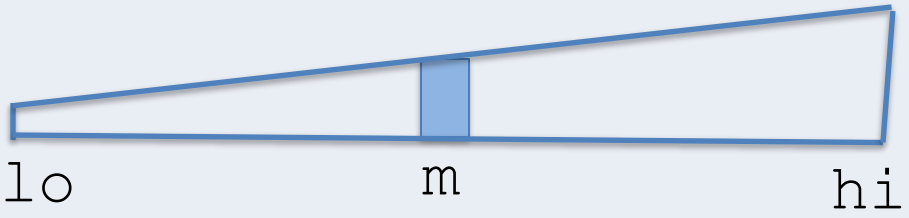
Modify the binary search algorithm so that it also identifies the first match if there are duplicates.

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

Modifying the following algorithm:

The original algorithm: search for `x` in the sorted array `A[0...n-1]`

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

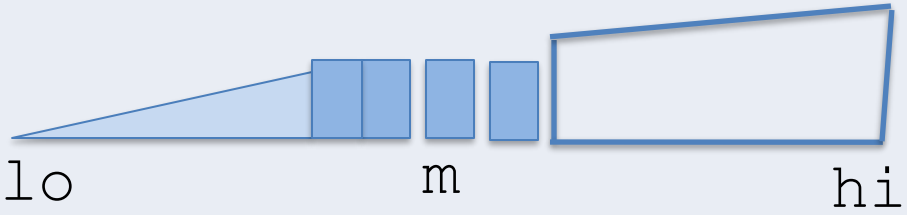


Question: Modify the algorithm so that it also *identifies the **first match** if there are duplicates*.

Modifying the following algorithm:

The original algorithm: search for `x` in the sorted array `A[0...n-1]`

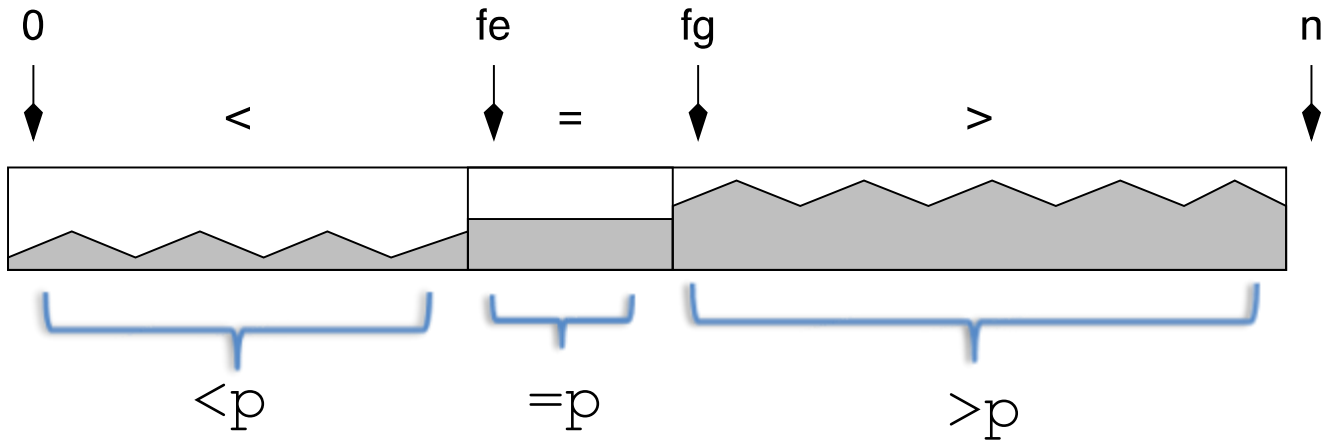
```
lo,hi ← 0,n-1
while lo <= hi
    m ← (lo + hi)/2
    if x < A[m]
        hi ← m - 1
    else if x > A[m]
        lo ← m + 1
    else
        return m
return not_found
```



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

Quicksort: Lecture Review

```
if n ≤ 1
    return
p ← any element in A[0..n-1]
(fe, fg) ← partition(A, n, p)
```




```
quicksort (A[0 . . . fe - 1])
quicksort (A[fg . . . n - 1])
```

Sorting= re-arrange array elements in *increasing* (or *decreasing*) *order*

Efficiency

	Worst Case	Average Case
Insertion Sort:	$O(n^2)$	$O(n^2)$
Selection Sort:	$O(n^2)$	$O(n^2)$
Quick Sort :	$O(n^2)$	O(nlogn)

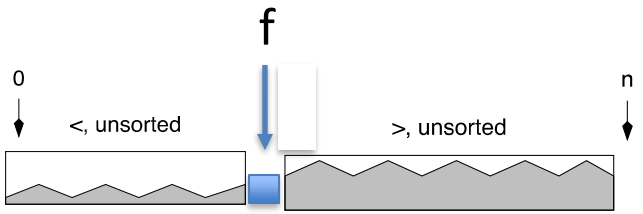
note: partition has O(n) efficiency

lec05 - 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?

Exercise 4 – lecture algorithm (in C)

```
void partition(int A[], int n, int *fe, int *fg) {
   int p= A[0];  // for simplicity, fix p as A[0]
   int next;
   next= *fe= 0; *fg= n;
   while (next < fg) {
         if (A[next] < p) {
         swap (&A[*fe],  &A[next]);
         (*fe)++;   next++;
      } else if (A[next] > p) {

         swap(&A[next], &A[*fg-1]);
         (*fg)--;

      } else {
         next++;
      }
   }
}
```

4b. Disadvantages/Advantages compared to the first method?

- 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?

Exercise 7.05 – Sorting Parallel Arrays

Task

Suppose that a set of "student number, mark" pairs are provided, one pair of numbers per line, with the lines in no particular order.

Write a program that reads this data and outputs the same data, but **ordered by student number**.

Sample input

```
823678  66
765876  94
864876  48
785671  68
854565  89
```

Sample output

```
765876  94
785671  68
823678  66
854565  89
864876  48
```

Sorting Parallel Arrays: Method 1 – performing swap in all arrays

using 1 array for stud_nums, 1 array for marks

index	stud_nums	marks
0	**823678**	**66**
1	765876	94
2	864876	48
3	785671	68
4	854565	89

When sorting:
- need to use stud_nums to decide swapping
- but when swapping, need to swap rows in both arrays!
- **what if we have many parallel arrays?**

index	stud_nums	marks
0	**765876**	**94**
1	785671	68
2	823678	66
3	854565	89
4	864876	48

Sorting Parallel Arrays: Method 2 – indirect sort

ndirect sort uses an additional array to store sorted indices.

index	stud_nums	marks
0	**823678**	**66**
1	765876	94
2	864876	48
3	785671	68
4	854565	89

```
When sorting:
- need to use stud_nums
  to decide swapping
- but when swapping,
  need to swap rows in
  both arrays!
- what if we have many
  parallel arrays?
- Other benefits?
```

index	P	stud_nums	marks
0	1	**823678**	**66**
1	3	765876	94
2	0	864876	48
3	4	785671	68
4	2	854565	89

Lab: Ex 7.8-7.9 and others.

Last Week:

Ex7.04: Count frequencies

Ex7.06: Selection sort

Ex7.07: Most frequent element

Ex7.x1: Is the array sorted?

Ex7.x2: Count distinct values

This Week:

Ex7.08: k'th smallest in array

Ex7.09: Count ascending runs

Ex7.x3: Longest ascending run

Ex7.10: Count inversions

Ex7.11: Trace a 2d array program

Ex3.06-X: Giving change with arrays

Next Week: string processing

Today's Wrap-up

- Recursion
- Array Algorithms
- Searching and Sorting Algorithms, and Big-O
  - *Sequential Search*
  - *Binary Search*
  - *Insertion Sort*
  - *Quick Sort*

Assignment 1 due Friday Week 8: start early (ASAP)

- Read, Understand, Know What to Do, and
- Prepare a Workspace for A1, upload all needed (=given) files, start coding
- Bring questions to next week's workshop
- Use Discussion Forum & FYC

Additional Slides