COMP10002 Workshop Week 6

	lec05.pdf review
	Binary Search: Exercise 1 at the end of lec05.pdf Quicksort: Exercise 4 at the end of lec05.pdf
	Q&A over the sample quiz
LAB	Quiz, 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;

Topics in lec05.pdf

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

Data type of pointers

```
int n= 10;   double z= 1.5;
int *pi;
double *pz;
```

Valid? Invalid?

```
pi = &n;
pi = n;
pi = z;
pi = &z;
pz = &z;   pi= pz;
```

Data type of pointers

```
int n= 10;   double z= 1.5;
int *pi;
double *pz;
```

Valid? Invalid?

pi = &n;

~~pi = n;~~

~~pi = z;~~

~~pi = &z;~~

pz = &z;   ~~pi= pz;~~

Pointer type is quite privilege! In:

```
int *p;
```

p cannot be mixed up with any other type, including with double* , float* , char*.

Consider the following declarations:

```
  int n=44;    int c='a';  double z=2.5;
  int *pn=&n; int *pc=&c; double *pz=&z;
```

In the context of the declarations, which of the following expressions contains a type incompatibility that means that it is not valid:

```
A.  pc == pz
B.  *pn == *pc
C.  *pn == *pz
D.  n == z
E.  &pn == &pc
```

Array name is a pointer constant!

Suppose that `A` is an array of int, and that `B` is declared to be a pointer of type `int *`. Which of the following is not a valid assignment statement, assuming that both `B` and the elements of `A` have been initialized to suitable values:

```
1. B = &(*A);
2. B = A + *B;
3. *A = A[1];
4. B = *(*A);
5. *B = A[1];
6. A= &A[2];
7. B= &A[2];
```

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

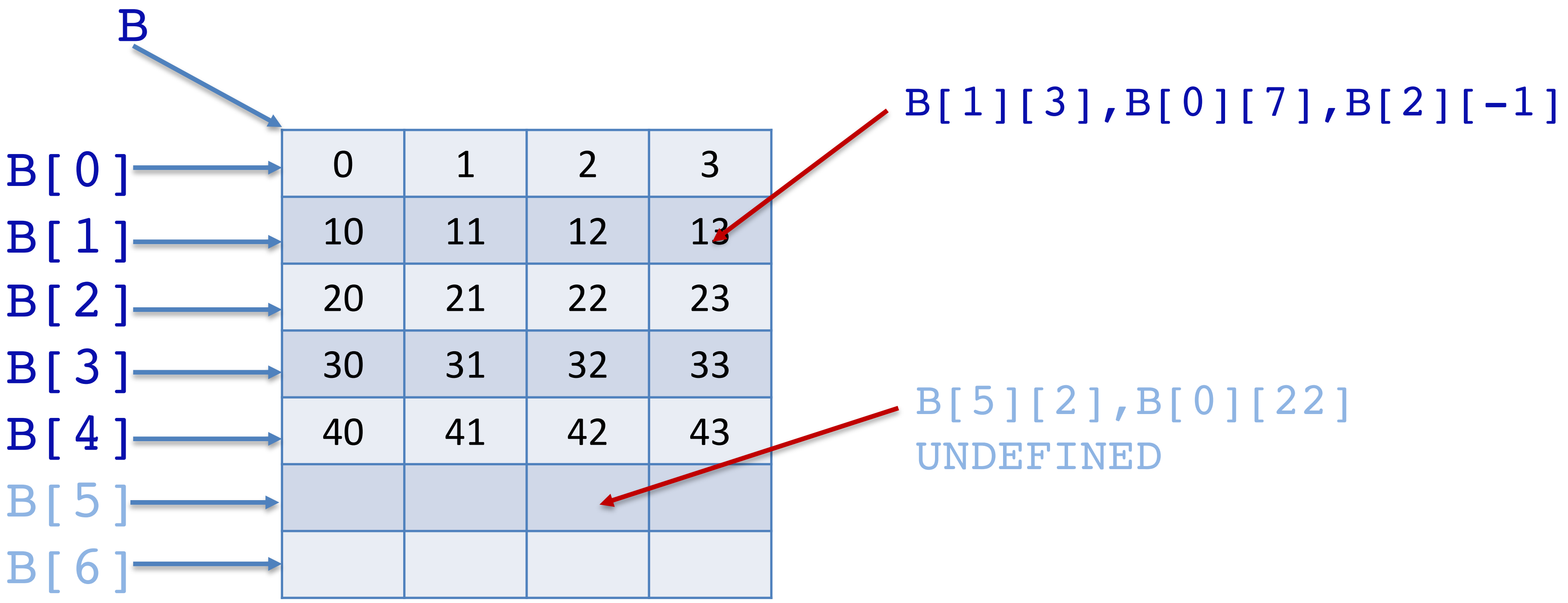
```
int B[5][4]={{0,1,2,3},{10,11,12,13},…};
```

`B` points to the location of `B[0][0];`

`B[i]` points to `B[i][0],` ie `B[i]` has the value of `&B[i][0];`

`B[i]+j` points to the `j`-th element after `B[i][0].` Depending on the value of `j`, this element can be at any position and can be undefined.

`B[i][j]` is valid, but might not be defined.



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

Big-O

$f(n) \in O(g(n)) \iff f(n) \le c.\ g(n)$ for large n

Big-O: $f(n) \in O(g(n))$

Important classes of $g(n)$ (supposing $0<\varepsilon<1,\ a>1,\ 0<\alpha<\beta$)

$1 \quad \log n \quad n^{\varepsilon} \quad n \quad n\log n \quad n^{a} \quad a^{n}$

$(\log n)^{\alpha} \quad (\log n)^{\beta} \quad n^{\alpha} \quad n^{\beta}$

Thumb rules when finding the *best* order of $f(n)$:

- if $f(n)$ is a sum : keep only *the most dominant* member
- if $f(n)$ is a product: delete all *constant* members
- base of log is not important

Big-O: $f(n) \in O(g(n))$

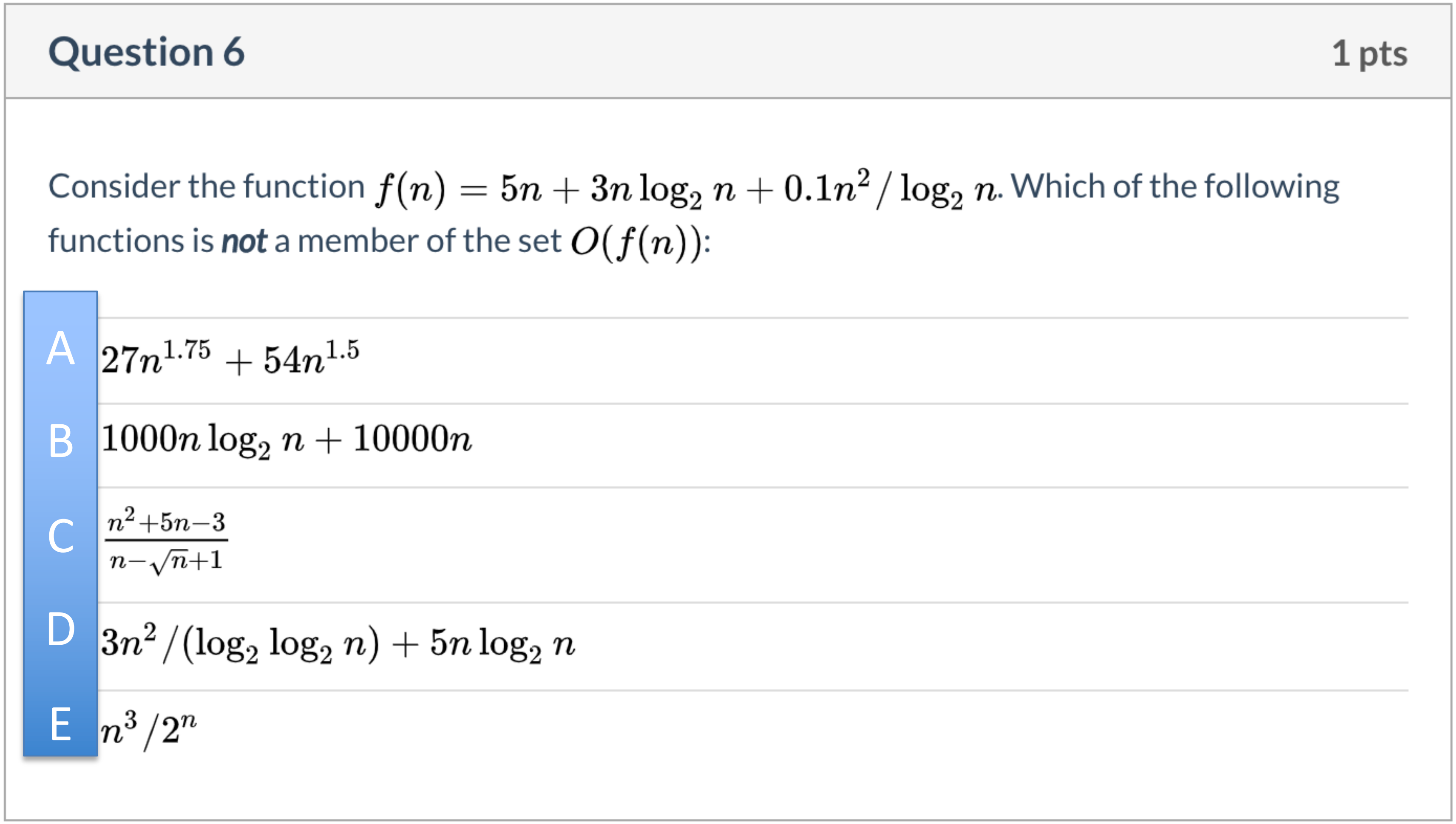
Find the best $g(n)$ for:

$f_1(n) = 9n + 3n\sqrt{n} + 10^9$

$f_2(n) = 1000n + n^2 + 100n\log n + n(\log n)^2$

$f_3(n) = n^2 \log\log n + 2n\log n^{20}$

Big-O



Question 6 **1 pts**

Consider the function $f(n) = 5n + 3n\log_2 n + 0.1n^2/\log_2 n$. Which of the following functions is ***not*** a member of the set $O(f(n))$:

A $27n^{1.75} + 54n^{1.5}$

B $1000n\log_2 n + 10000n$

C $\frac{n^2+5n-3}{n-\sqrt{n}+1}$

D $3n^2/(\log_2\log_2 n) + 5n\log_2 n$

E $n^3/2^n$

Binary Search

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

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
```

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

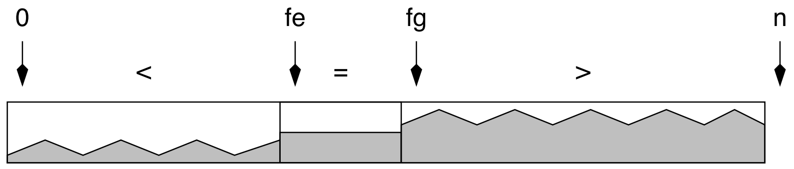
Correctness: Original → New Version

```
define P ≡ (0 ≤ i ≤ n) and (x∉ A[0 . . . i − 1])
i←0
assert: P
while i < n and A[i] ≠ x
    assert: P and (i < n and A[i] ≠ x)
    i←i+1
    assert: P

assert: P and (i ≥ n or A[i] = x)
if i ≥ n
    assert: P and(...)and(i≥n) =⇒ x ∉ A[0...n−1]
    return not found
else
    assert: P and (...) and (i ≱ n) =⇒ A[i] = x
    return i
```

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

Quicksort: The original logic with fe, fg

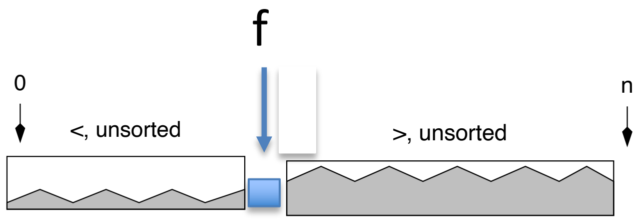
```
define R ≡ 0 ≤ fe < fg ≤ n and A[0 . . . fe − 1] < p and
A[fe . . . fg − 1] = p and A[fg . . . n − 1] > p

if n ≤ 1
   return
else
   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 **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
```

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.

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]** returns the value that appears most frequently in an array `A` of integers. On tie, returns the smallest one. The array `A` may not be modified.

7.8: [W5X] returns the *k*-smallest of `int A[]`, not modifying `A`

7.9: [W05] returns the number of *ascending runs* in `int A[].` For example, array **`{10,13,16,18,15,22,21}`** has 3 runs.

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`