AY1718 S1 Midsem Test

Selected Answers

Qn A

1. O(N)
2. O(1)
3. O(1)
4. O(N)
5. O(N)
6. O(N)
7. O(1)
8. O(1)
9. O(N log N)
10. O(N)

Qn A2

Search for the k-th smallest item in a **sorted** Array.

O(1)

Look at array[k-1].

Qn A3

Delete an item at index i of a C++ STL vector of size *n* and ***n-i* ≤ 3**.

O(1)

n-i ≤ 3 → (up to) last 3 elements

Regardless how long the vector is,

it will only take (up to) “3 steps” to remove the item.

Qn A9

Extract ***minimum*** item of a Binary ***Max*** heap using ***ExtractMax()***.

O(N log N)

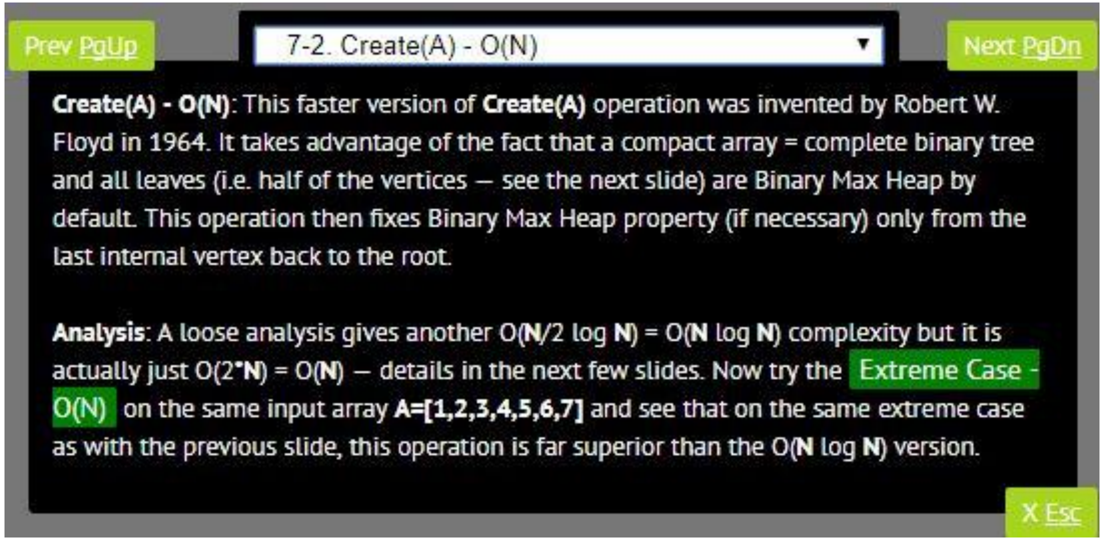
Each *ExtractMax()* is O(log N).

Since you are ***constrained*** to only use *ExtractMax()*, you must call it **N-1** times before you can extract the minimum item and then push all **N-1** of it back.

Qn A10

Build a Binary Max Heap from any integer array *A* of size *N*.

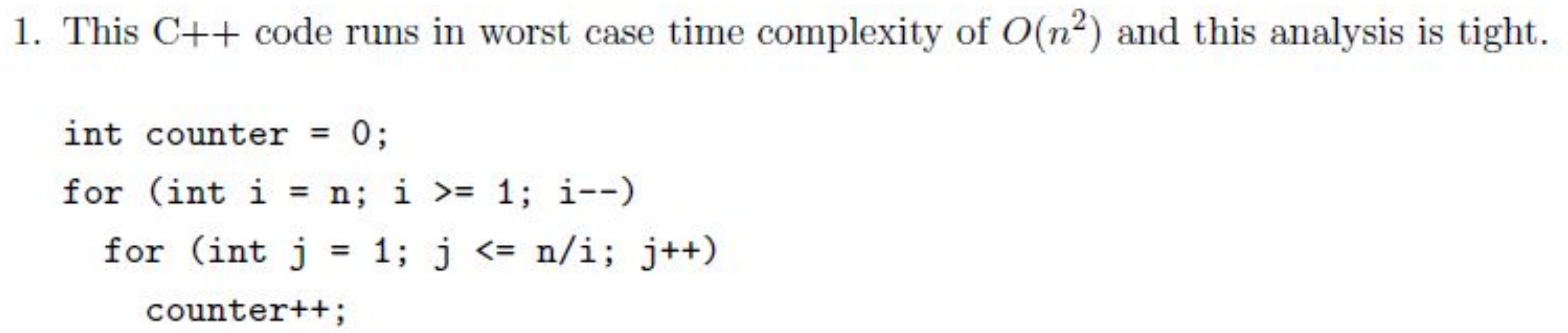
O(N)



Create(A) - O(N): This faster version of **Create(A)** operation was invented by Robert W. Floyd in 1964. It takes advantage of the fact that a compact array = complete binary tree and all leaves (i.e. half of the vertices — see the next slide) are Binary Max Heap by default. This operation then fixes Binary Max Heap property (if necessary) only from the last internal vertex back to the root.

Analysis: A loose analysis gives another O(**N**/2 log **N**) = O(**N** log **N**) complexity but it is actually just O(2*N) = O(**N**) — details in the next few slides. Now try the Extreme Case - O(N) on the same input array **A=[1,2,3,4,5,6,7]** and see that on the same extreme case as with the previous slide, this operation is far superior than the O(**N** log **N**) version.

Qn B1



1. This C++ code runs in worst case time complexity of $O(n^2)$ and this analysis is tight.

```
int counter = 0;
for (int i = n; i >= 1; i--)
  for (int j = 1; j <= n/i; j++)
    counter++;
```

When **i** = n, **j** from 1 to 1.

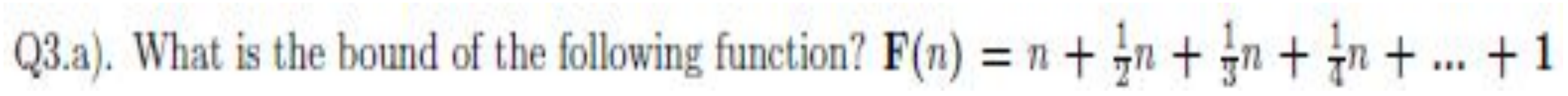
When **i** = n-1, **j** from 1 to 1???.

When **i** = 3, **j** from 1 to n/3.

When **i** = 2, **j** from 1 to n/2.

Qn B1

Complexity = O(**n/n** + **n/(n-1)** + ... + **n/3** + **n/2** + **n/1**)



Q3.a). What is the bound of the following function? $\mathbf{F}(n) = n + \frac{1}{2}n + \frac{1}{3}n + \frac{1}{4}n + ... + 1$

If you can mention **Harmonic Series**: Good :)

Else, we need to see that you show explicitly

O(1 + ½ + ⅓ + ... + 1/N) = O(log N)

Factor the **N** out.

Qn B2

(Randomized) Quick Sort is the *best* sorting algorithm to sort <u>*any* set of **n** integers</u>.

No.

Counting Sort -- small range

Optimized bubble sort/insertion sort -- already sorted

Qn B2

(Randomized) Quick Sort is the *best* sorting algorithm to sort *any* set of **n** integers.

Telling us that merge sort is *also* O(n log n) is not a strong enough argument and was given 2 out of 3 marks.

Qn B3

False. Both can be equally good as removing last element from Vector is also O(1).

Adding to the back of Linked List and Vector are both O(1).

Qn B4

True. LinkedList can do O(1) insert/delete from front and back. Vector can only do O(1) insert/delete from the back end.

To implement a queue, we need to insert on one end, and delete from the other. Hence, either the push or pop operations will be O(N) if Vector is used. This is slower compared to Linked List.

Qn B5

False. The smallest element can be anywhere among the leaf vertices, e.g. A = [-, 3, 1, 2], see that 1 is not at A[3] but at A[2].

Note: index 0 is not used, hence its “-”.

In exam, you can also draw the binary heap as a counter-example.

Qn C1

Observation

If two strings **A** and **B** are anagrams:

They must have the **<u>same number of every character</u>** from ‘A’ to ‘Z’.

(As a consequence, when you sort the characters, you should get the same sorted string).

Qn C1

STL sort → O(**N** log **N**)

[12 marks if no other deductions]

Counting sort → O(**N** + *26*)

[Only 26 uppercase characters]

Qn C2

constrained to just use the given implementation

Vector, array, priority queue... **NO**

If you used STL List and <u>only insert to front and back</u>... we close one eye and deduct some marks

list.sort() ... **NO**

Qn C2

Approach

Push the head (pivot) into SLL result.

For all the remaining elements:

    If > pivot, push to the *back* of result

    If **<u><=</u>** pivot, push to *front* of result

Those that never handle equality cases... :O

Qn C3

Observation

Let M_K be the max of the bottom K elements in the stack.

Let S_K be the Kth element from the bottom of the stack.

Then,

$$\mathbf{M_{K+1} = max(M_K, S_K)}$$

In essence, this is a *running max* of the stack.

Qn C3

Approach

Keep 2 stacks, for **M** and **S** respectively.

When adding a new number **X**:

    Push **X** into **S**

    Push max(**X**, **M.top()**) into **M** *

When there is no number in **M**, *what to do?**

Qn C3

Approach

When popping a number from the stack:

- Pop from **S**
- Pop from **M**

You can use **STL stack**!

Qn D

Foreword

Since this is a bonus question, we are very strict regarding this.

Direct *implementation* is already O(**N**^{2}) and does not demonstrate much algorithmic thinking.

Low marks for this question: ~5 marks

Qn D

Observation

Lets say we only reverse the array without ‘dropping’ any character.

Odd number of times: Reversed

Even number of times: Not-reversed

Qn D

Observation

Now when we have *drop*,

Dropping from the front in *reversed*

        ==> dropping from the back

Dropping from the front in *not reversed*

        ==> dropping from the front

Qn D

Approach

We need a data structure that can drop from front and back.

pop_front

pop_back

Deque or Linked List

Qn D

Approach

Keep track of how many times ‘R’ has appeared.

Drop from front/back respectively.

Print from front/back respectively.

AY1718 S2 Midsem Test

Selected Answers

Qn A

1. O(1)
2. O(N)
3. O(N) or O(N log N)
4. O(N log N)
5. O(log N)
6. O(N)
7. O(1)
8. O(1)
9. O(N)
10. O(1)

Qn A2

Erase the *first* element of a **sorted** std::vector.

O(n)

Vector will ‘copy’ the remaining n-1 items forward.

Qn A3

Compare if two **equal-size** std::vectors have the same elements.

If elements can be hashed: O(N) using Hash Table

If not able to hash: O(N log N) sort, then O(N) loop.

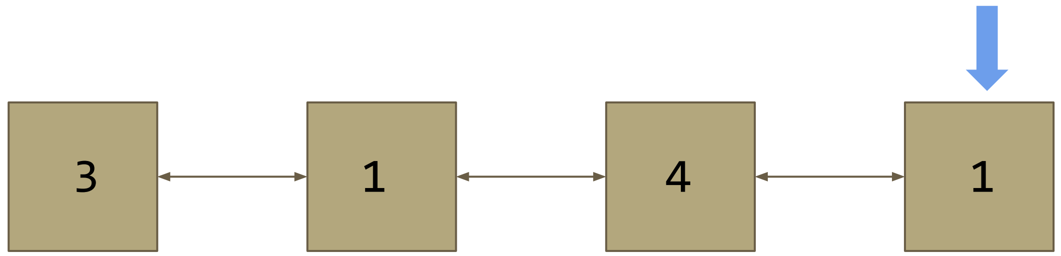
Both O(N) and O(N log N) are accepted.

- Steven will make the wording tighter next time

Qn A7

Get the *third last* element from **std::list**.

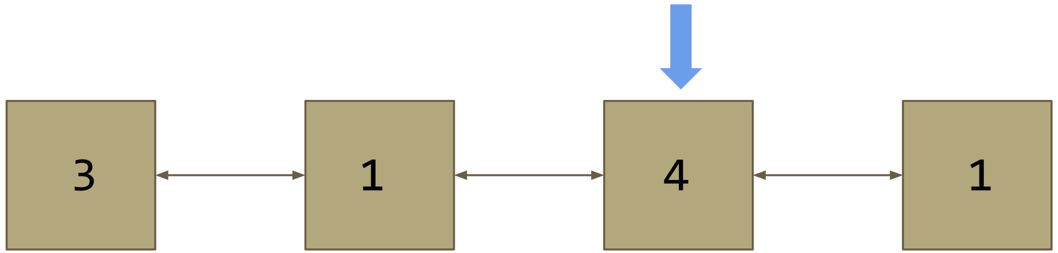
- Doubly linked list



Qn A7

Get the *third last* element from **std::list**.

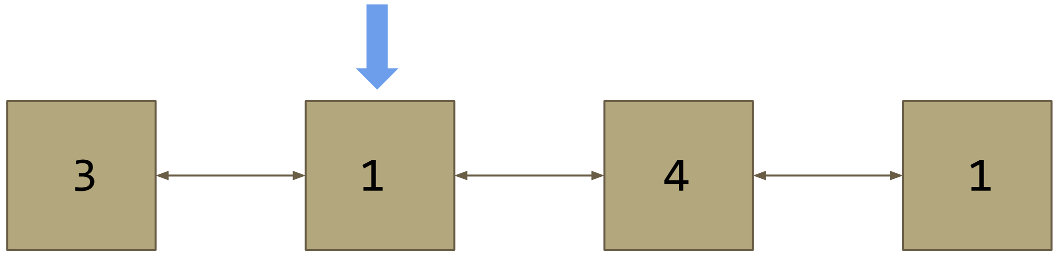
- Doubly linked list



Qn A7

Get the *third last* element from **std::list**.

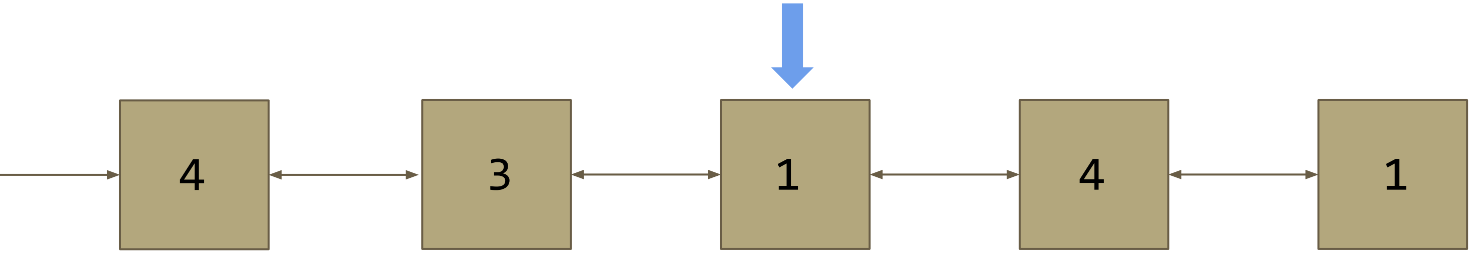
- Doubly linked list



Qn A7

No matter how long the linked list is...

removeLast() 3 times. 3 operations. → O(1)



Qn B1

False.

Both cin/cout and scanf/printf can be used.

C++ supports many C functions as well.

Qn B2

False.

Randomized **quick sort** is not stable.

Most stable sort implementations uses merge sort or its variants.

Qn B3

True.

DLL needs to store both the next and previous object references to the next vertex.

SLL only stores the reference to the next vertex.

Qn B4

Stack ADT with *vector*; Random access.

True, if you expose additional methods from the underlying *vector*.

False, if you argue that Stack ADT do not have random access methods exposed. Hence, require multiple pops to retrieve the element.

Qn B5

Contextualised meaning of **superset**:

Deque has all the methods that vector has.

(+ more)

Qn B5

We can *generally* replace.

Question is intentionally vaguely worded.

Can argue both ways.

Most features of vector are also in deque.

Some specific ones are not, e.g. reserve

Section C Remarks

Implications

Can comment about things that *remain at the same time/memory complexity*.

But needs to be **coded differently**.

Qn C1

List ADT backed by vector.

2	5	3	4	7	9

Qn C1

List ADT backed by vector.

When deleting an element.

If we close the resulting gap:

2	5	3	4	7	9

Qn C1

List ADT backed by vector.

When deleting an element.

If we close the resulting gap:

2	**5**	**4**	**7**	**9**

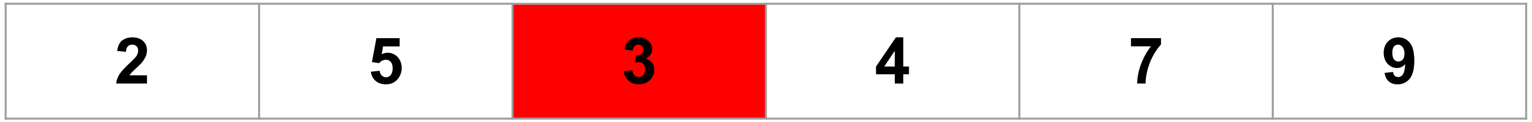
O(N) loop + pop_back

Qn C1

List ADT backed by vector.

When deleting an element.

If we **do not** close the resulting gap:



Qn C1

List ADT backed by vector.

When deleting an element.

If we **do not** close the resulting gap:

2	**5**	**-**	**4**	**7**	**9**

O(1) delete, we *generally* just mark the deleted cell

Qn C1

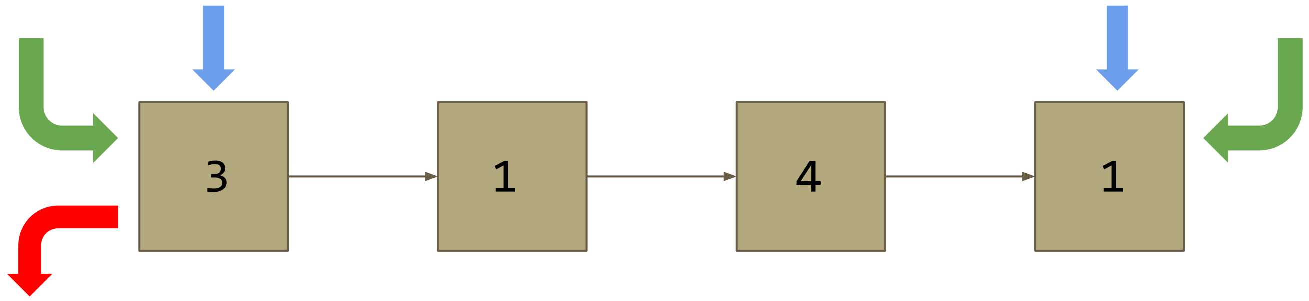
- Memory wastage
  - If there are many inserts and deletes?
- Search / Insert is still **O(*N*)** ???
  - But '***N***' is now the total number of inserted elements (including deleted ones)
    - **N** here is not the number of elements in the linked list
  - Harder to code, need to correctly skip blank elements

Qn C1

- Only the first remove(i) is O(1)
  - Future removes need to count from index 0 to get the new index to remove → still O(N)
- Cannot sort directly
- Extra memory usage for ‘isDeleted’ flag
  - Or requires the use of dummy value (which may cause problems if that value is actually used)

Qn C2

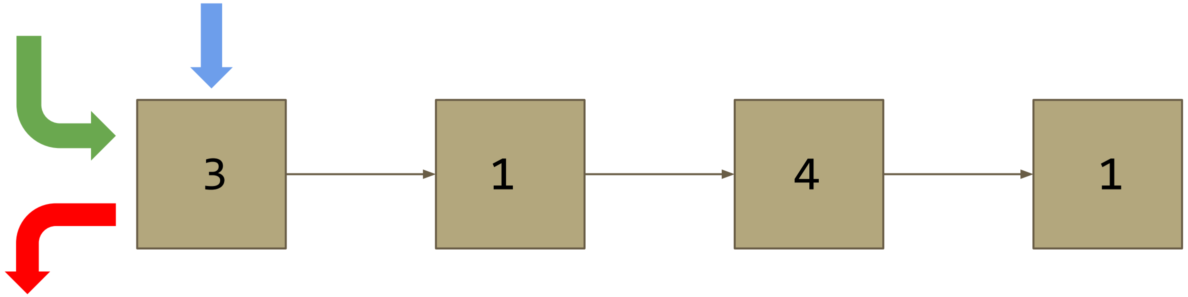
Singly Linked List **with** tail pointer.



Qn C2

Singly Linked List **without** tail pointer.

Access tail is not O(1) anymore.



Qn C2

Common Mistake

- Deleting the tail is now **O(N)** instead of **O(1)**.

In the original SLL, deleting tail is already **O(N)**.

Qn C2

Suggested Answers

- Save 4 (or 8) bytes of memory... lol
- Insert at tail is **O(N)** instead of **O(1)**
- Remove is less complex as we don’t need to update the tail pointer
- Remove **remains** as O(**N**)
- OK for stack ADT, not good for queue ADT
- Still not good for deque ADT

Qn D1

Approach

– Input **N** strings, reverse them.	O(NM)
– Sort **N** strings.	O(NM log N)
– Reverse the **N** strings.	O(NM)
– Output **N** strings.	O(NM)

Why O(NM log N)?

Qn D1

Reversing a string using STL

```
string s = "rarthecat";
reverse(s.begin(), s.end());
```

Qn D1

Reversing a string without using STL

```
string s = "rarthecat";
for (int i = 0; i < N/2; ++i) { // /2 :O
    char t = s[i];
    s[i] = s[N-i-1];
    s[N-i-1] = t;
}
```

Qn D1

Sorting strings

```
vector<string> names;
sort(names.begin(), names.end());
```

C-style strings... a bit harder

Search up strcmp

Qn D2

Observation

- Only values 1, 2 or 3.

How will the *sorted array* look like?

1, 1, 1, ….., 1, 2, 2, 2, ….., 2, 3, 3, 3, …, 3

Qn D2

How will the *sorted array* look like?

1, 1, 1, ..., 1, 2, 2, 2, ..., 2, 3, 3, 3, ..., 3

[0 or more 1s][0 or more 2s][0 or more 3s]

We can use *counter* to track how many of each number! → Counting Sort (the frequency counting part only)

Qn D2

Approach

For each element:

- Keep track of how many ‘1’, ‘2’ and ‘3’. O(1)
- Calculate the median(s) based on the counters. O(1)
- Sum the median. O(1)

Qn D2

Approach

- Calculate the median(s) based on the counters. O(1)

$\rightarrow$ Some cases…

We shall go through the general form.

Qn D2

Approach

How to find the number at *any* index **K**?

0 ≤ **K** < counter[1]

- 1

counter[1] ≤ **K** < counter[1] + counter[2]

- 2

counter[1] + counter[2] ≤ **K**

- 3

Number	**Counter**
1	17
2	28
3	20

Qn D2

Approach

How to find the median?

Vary value of **K**.

- **K = N/2** and/or **K = N/2-1**

Compute the number at **K**.

Then, compute the median.

Overall complexity: **O(N) * O(1) = O(N)**

AY1718 S4 Midsem Test

Selected Answers

Note: S4 is conducted in Java

Qn A

1. $O(N)$
2. $O(N^2)$
3. $O(N)$
4. $O(N)$
5. $O(1)$
6. $O(N)$
7. $O(N)$
8. $O(\log^2 N)$, which is $O((\log N)^2)$
9. $O(N \log N)$
10. $O(N)$

Qn B1

True.

It is possible to take more than O(N) to compare 2 elements.

For example, if you are comparing strings of length **N**, then the total complexity of sort is $O(N \log N) * O(N) \rightarrow O(N^2 \log N)$

Qn B2

True.

It is possible to do so in O(N log N) using Mergesort, using the merging part.

It is also possible for Optimized Bubble Sort to run in O(N) for a sorted array.

Qn B3

False.

There are only 26 alphabets.

We can count number of ‘A’ .. ‘Z’ in each string and then compare the counters subsequently.

This takes O(N + M) time complexity.

Qn B4

False.

If we insert the sequence [1, 1] into both Stack and Queue, we will get [1, 1] out when we pop both of them.

Qn B5

True.

We can use an array/vector.

Append new elements to the back of the vector.

Before any dequeue operation, sort the vector in increasing order.

Dequeue is simply returning the back of the vector and pop_back.

Qn C

1. Dequeue operation remains O(1).
2. Will never encounter case where we cannot enqueue an item.
3. More memory efficient if queue is near max capacity, as we do not store pointers unlike a LL.
4. Less memory efficient if queue is near empty, as we have many empty slots in the array.

Qn C

5. We can access any element in O(1) time if the underlying array methods are exposed.
6. Can replace any element in O(1) as well.

Qn D1

Insert {1, 2, 7, 6, 5, 4, 3}.

Since {1, 2} is already sorted, we can ignore them.

{7, 6, 5, 4, 3} is ‘*reverse sorted*’ and requires 4 + 3 + 2 + 1 Bubble Sort swaps to sort them.

Qn D1

One other way to get a correct input array is to...

- Start off with a sorted array of 1 .. 7
- Make 10 swaps of two adjacent numbers that are in the correct order.
    - Eg: Swap A[i] and A[i+1] only if A[i] < A[i+1]
- Bubble sort will hence need 10 swaps to undo what you have just done. :)

Qn D2

X contains 15 integers of the **same value**.

It does not matter what the random pivot is, if it is always the *same value*...

For VisuAlgo, if elements compare equal to the pivot, it *always go to the right partition*.

Qn D2

So the first partition will compare 14 elements with the chosen pivot.

The second partition will compare 13 elements with the chosen pivot.

Total number of comparisons

$= 14 + 13 + 12 + \ldots + 3 + 2 + 1$

$= 14 * (14 + 1) / 2$

$= 105$

Qn D3

In Java, “Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.”

Hence, to force Java to make many pointer advancements, insert 4 times into the middle of the Linked List. (i.e. index 50)

AY1819 S1 Midsem Test

Question and Solutions

AY1819 S1 Midsem

Question PDF: https://goo.gl/Psx2Jc

Solution PDF: https://goo.gl/CrZzYy