CS2040 2023/2024 Sem 1 Midterm

MCQ: 40 Marks, 10 Questions

1. What is the time complexity of calling f(N) for some N > 1?

```
void f(int N) {
      while (N > 0) {
            while (N > 1) {
                  System.out.print("*");
                  N /= 2;
            }
            System.out.print("*");
            N -= 1;
      }
}
```

a. O(log N)
b. O(sqrt(N))
c. O(N)
d. O(N log N)
e. O(N^2)

Ans: a
Outer loop only performs 1 iteration

2. What is the time complexity of calling g(N) for some N > 1?

```
long g(int N) {
      long sum = 0;
      for (int jump = 1; jump <= N; jump++) {
            long prod = 1;
            for (int i = 1; i <= N; i *= 2)
                  prod *= jump;
            sum += prod;
      }
      return sum;
}
```

a. O(N)
b. O(N log N)

c. $O(N^2)$
d. $O(2^N)$
e. $O(N^N)$

Ans: b
Don't confuse the computed return value (sum) with the number of operations performed

3. For this question, you may **NOT** utilize the function call-stack (that means you cannot use recursion).

   Given a (pure ADT, language-independent) stack **S** = [5, 2, 4, 1, 3, 9, 2, 8, 7 (where element 7 is at the top and 5 the element at the bottom), which of the following is/are a possible result after performing operations on **S** and ONLY the given data structure instance, with NO additional (variables or data structures) to store elements from **S**?

a. **S** = [7, 8, 2, 9, 3, 1, 4, 2, 5 is possible using ONLY **S** and nothing else
b. **S** = [5, 2, 9, 3, 1, 4, 2, 8, 7 is possible using ONLY one additional stack
c. S = [5, 2, 9, 3, 1, 4, 2, 8, 7 is possible using ONLY one additional queue
d. All of (a)-(c) are **NOT** possible
e. More than one of (a)-(c) are possible

Ans: c
Flip [2, 8, 7] using the queue back into the stack, enqueue 7 elements, rotate [2, 8, 7] in the queue, then push all 7 elements back into the stack

4. The lecture version of quicksort is called on an array **A**, and at some point **WITHIN** the sorting algorithm **A** = [2, 1, 7, 5, 4, 8, 3, 6]. Which option is a valid initial ordering of the elements in **A**?

a. [7, 1, 2, 5, 4, 8, 3, 6]
b. [8, 1, 7, 5, 4, 2, 3, 6]
c. [6, 1, 7, 5, 4, 8, 3, 2]
d. [2, 7, 5, 1, 4, 8, 3, 6]
e. [2, 6, 7, 5, 4, 8, 3, 1]
f. None of (a)-(e) is a valid initial ordering

Ans: e
The snapshot of A shows that the first partition has NOT been completed as there is no such

candidate pivot
Therefore the snapshot is taken WITHIN the partition algorithm, i.e. the first element has to be 2 and is the pivot
Between (d) and (e), the partition algorithm used in lecture works by swapping, not shifting elements

5. Consider the following 3 sorting algorithms:

   S1 – Insertion sort with the sorted region expanding from the higher-indexed region down to the lower-indexed region after each pass (instead of the other way round which is what is given in the lecture notes)

   S2 – Selection sort with the sorted region expanding from the higher-indexed region down to the lower-indexed region after each pass

   S3 – Bubble sort with the sorted region expanding from the higher-indexed region down to the lower-indexed region each pass

   A sorting algorithm is called on an array **A** but has not run to completion. After 3 outer loop passes have run fully, **A** = [3, 4, 1, 2, 9, 5, 6, 7, 8]. Which algorithm(s) could have been running on **A**?

a. **S1 only**
b. S2 only
c. S3 only
d. S1 or S3 could have possibly been running but not S2
e. Any of S1, S2, S3 could have possibly been running

Ans: a
Both selection and bubble sort will not leave the largest element (9) unsorted after 1 outer loop pass

6. What is the time complexity of calling Fun(n) for some n > 1?

```
void Fun(int n) {
      System.out.println("*");
      if (n == 0)
            return;
      for (int i = 1; i < n+1; i++)
            Fun(n-i);

}
```

a. O(n^2) time
b. **O(2^n) time**
c. O(n!) time
d. O(n^n) time

Ans: b

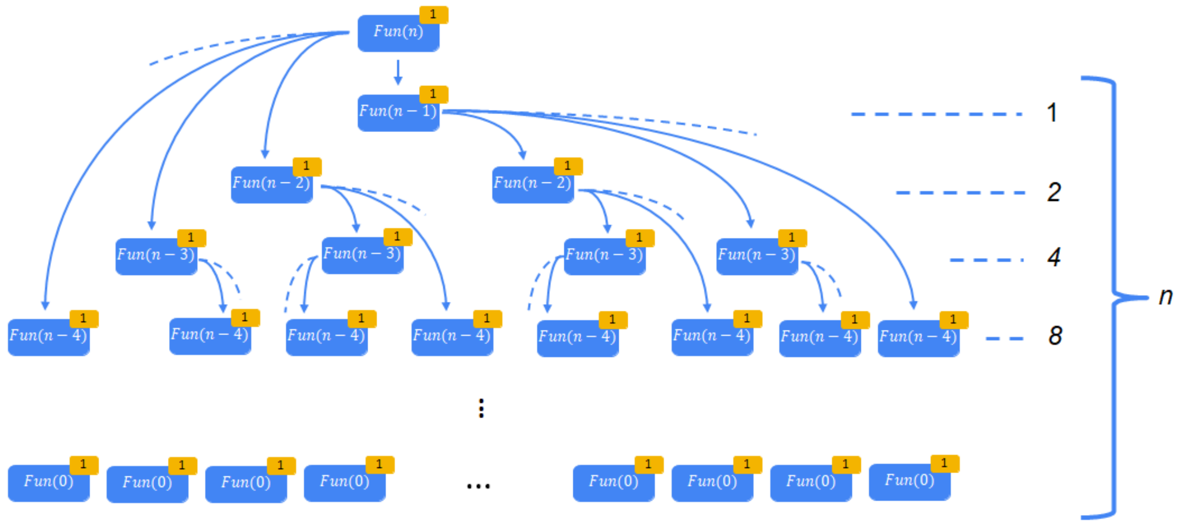
We note that Fun(i) is directly called once for every Fun(j), $j > i$. Counting the number of calls for each Fun(i)we have:

Function	Fun(n)	Fun(n-1)	Fun(n-2)	Fun(n-3)	Fun(n-4)	…	Fun(0)
Number of calls	1	1	1+1=2	1+1+2=4	1+1+2+4= 8	…	2^{n-1}

In general (with the exception of $k = 0$), Fun(n-k) is called 2^{k-1} times. From the geometric series, we can compute the number of times Fun() is called:

$$\underbrace{\frac{1\cdot(2^n-1)}{2-1}}_{\texttt{Fun(n-1) to Fun(0)}} + \underbrace{1}_{\texttt{Fun(n)}} = 2^n$$

Since each call to Fun() does $O(1)$ operations, we have a total running time of $O(2^n)$. An illustration is given below



7. How many data structures below can support all queue operations in worst case O(1) time? The options are independent, e.g. picking (i) and (v) means that it can be done with either a basic linked list alone, or a stack alone.

   You may assume that if you use an array, the number of valid elements in the ADT will always be less than the array size, at any point in time and no resizing is required.

i. Basic linked list
ii. Doubly linked list with tail reference
iii. Circular linked list with only head reference, no tail reference, singly linked
iv. Circular array
v. Stack

a. 1
b. **2**
c. 3
d. 4

Ans: b.

i. Using basic linked list, enqueue takes O(N) time.
ii. With the tail reference, all queue operations can be done in O(1) time.
iii. Since the circular list is singly linked and does not have tail reference, enqueue takes O(N) time.
iv. Using circular array, all queue operations can be done in O(1) time by keeping track of the front and back index of the queue.
v. Depends on the implementation, when queue is implemented using 2 stacks, either enqueue takes O(N) and dequeue takes O(1) or vice versa.

8. Suppose we want to implement a special stack that has a **push_mid(x, pos)** operation, which inserts the element **x** to **pos**, where **pos** can be either:
   - an integer index of the position you want to insert **x** if you are using an array based implementation, or
   - a pointer/ reference to the node that you want to insert **x** immediately on top of in the stack if you are using a linked list based implementation. For example if there are 4 entries in the stack and **pos** is a reference to the 3rd entry from the top of the stack then **x** should be inserted between the 2nd and 3rd entry from the top of the stack.

   You may assume that if you use an array, the number of valid elements in the ADT will always be less than the array size, at any point in time and no resizing is required. Which of the following data structures can you use to make this operation cost O(1) time in the worst case, while still allowing a worst-case time complexity of O(1) time for the usual **push (to the top of the stack)** and **pop (from the top of the stack)** operations.

   Note that if a linked list based implementation is used, the element stored in each node cannot be modified.

   i. Array
   ii. Circular array
   iii. Singly linked list without tail reference
   iv. Doubly linked list with tail reference

a. i and iii
b. ii only
c. iii and iv
d. iv only

Ans: d

i. Array requires shifting the existing elements to make space to insert the new element in.
ii. Same as i.
iii. When using singly linked list without tail reference, the head pointer references the top of stack. Since we want to insert **x** on top of **pos** in the stack, we need to obtain and change the reference of the previous node of **pos** to **x**, which cannot be done in O(1) time via **push_mid(x, pos)**.
iv. It can be done by modifying the pointers of **pos.prev**, **x**, and **pos**.

9. Given a hash table T with 25 slots that stores 1,000 entries, the load factor α for T is:

a. 40
b. 0.025
c. 4000
d. 2.5
e. 1.0

Ans: a

Simple application of the formula for load factor.

10. A hash table of length 10 uses hash function h(k)=k mod 10, and linear probing for collision resolution. After inserting 6 keys into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the keys could have been inserted in the table?

a. 46, 42, 34, 52, 23, 33
b. 46, 34, 42, 23, 52, 33
c. 34, 42, 23, 52, 33, 46
d. 42, 46, 33, 23, 34, 52

Ans: b

We will check whether the sequence given in option a. can lead to hash table given in the question. Option a. inserts 46, 42, 34, 52, 23, 33 as:

For key 46, h(46) is 46%10 = 6. Therefore, 46 is placed in slot (index) 6 in the hash table.
For key 42, h(42) is 42%10 = 2. Therefore, 42 is placed in slot 2 in the hash table.
For key 34, h(34) is 34%10 = 4. Therefore, 34 is placed in slot 4 in the hash table.
For key 52, h(52) is 52%10 = 2. However, slot 2 is occupied with 42. Therefore, 52 is placed in slot 3 in the hash table (the first empty slot after 2). But in the given hash table, 52 is placed in slot 5. Therefore, the sequence in option a. cannot generate the hash table given in the question.

For option c. the first 3 numbers can be directly inserted with no collisions into their respective slots: 34 into 4, 42 into 2, and 23 into 3. 52 has a collision with 2 so it goes into the next empty slot after 2: 5. 33 also has a collision and goes into the next empty slot: 6. However, in the given table 46 is in slot 6, so the sequence c. does not result in the given table.

For option d. the first 3 numbers go into their respective slots without collisions. 23 should go into slot 3, but that is occupied, so it goes into slot 4. However, in the given table slot 4 is occupied by 34. So, the sequence of d. cannot result in the given table.

Finally, if we check the option b. it is the only sequence that results in the given table. The first 4 number are inserted without collisions into slots 6, 4, 2, and 3. 52 has a collision with slot 2, and 3 and 4 are occupied, so it goes into slot 5. Finally, 33 has a collision with slot 3, and the closest empty slot is 7.

Analysis: 18 marks, 3 problems (6 marks each)

Question 11 and 12 refers to the problem given below:

11. **[4 marks]** Let there be a non-empty sequence of integers in an array **a[0]a[1]a[2]a[3]a[4]** … **a[N-1]**. Initially, the sequence **is sorted**. Then, zero, one or more continuous subsequences (**a[i..j]** for **i** < **j**) are reversed. No two of such subsequences overlap.

    For example, initially **a** = [1, 2, 4, 5, 5, 7, 8, 11, 11, 15, 18]. After reversing some non-overlapping subsequences, **a** = [2, 1, 4, 5, 8, 7, 5, 18, 15, 11, 11]. Here the continuous subsequences [2,1], [8,7,5] and [18,15,11,11] are the reversed subsequences.

    After the reverse(s) have taken place, an algorithm sort(**a**) is used to attempt to recover the initial sequence in **a**, placing the result in a stack **S**. The top of **S** should be the first element that was initially in **a**.

```
Stack sort(int[] a) {
    Stack S = new Stack<>(), T = new Stack<>();
    for (int i = 0; i < a.length; i++) { //a.length is N
        while ( !S.isEmpty() && S.peek() < a[i])
            T.push(S.pop());
        S.push(a[i]);
    }
    while ( !T.isEmpty())
        S.push(T.pop());
    return S;
}
```

    It is guaranteed that the sort algorithm compiles correctly, there is no syntax error.

    Claim: The sort algorithm will solve the problem of recovering the initial sequence correctly.

a. The claim is true because the algorithm performs insertion sort by repeatedly expanding **S** (the sorted region)
b. **The claim is true because the algorithm stores any subsequence that could potentially be reversed in one stack, only moving to the other stack when it is safe to do so**
c. The claim is false because at times, the algorithm **does not work** properly when there are two elements having the same value
d. The claim is false because reversing **T** into **S** causes **S** to be in the **wrong order** (largest element ends up at the top of **S**)
e. The claim is false because there are still elements in **S** at the end of the for loop, so it **does not work**

Ans: b

Option b is correct because the invariant of **S** is that elements closer to the top get smaller or equal to previous, but never larger, i.e. reversed subsequences. This allows any larger element to easily confirm that the smaller elements in **S** should appear first in the output.

T stores confirmed elements, but the top of the stack is the largest element, hence it needs to be reversed back into **S**

Option a is not correct, because this is not insertion sort, you aren't repeatedly expanding **S** as the sorted region till you get the sorted sequence

Option c is not correct, because equal elements are intentionally left in **S**, as it is still unsafe to conclude that these 2 elements are already in the sorted region e.g. [5, 5, 2, 1, 9]. It is not a requirement for this "sort"/unscrambling algorithm to be stable

Option d is not correct because before reversing, the top of **T** is the largest element, hence the need to reverse into **S**

Option e is not correct because there may be elements in **S** after the for loop, but elements in **T** would all be smaller than those in **S**

12. **[2 marks]** Regardless of whether the sort(**a**) algorithm works or not, the worst-case time complexity of the sort(**a**) algorithm is

a. O(N)
b. O(N log N) as that is the best possible worst-case time complexity you can get for a sorting algorithm
c. O(N^2) as there are nested loops, each performing iteration
d. Some polynomial time complexity but worse than O(N^2) as you are sorting the array repeatedly
e. Algorithm does not terminate – infinite loop

Ans: a

Each element in **a** gets copied into **S**, possibly gets moved into **T** and eventually back into **S**. Since each element is moved up to 3 (a constant number of) times, and not compared more times, this is an O(N) algorithm

Question 13 to 14 refers to the following problem

13. **[3 marks]** Let there be a non-empty sequence of arbitrarily large integers in an array **b[0] b[1] b[2] b[3] b[4]** … **b[N-1]**.

    Suppose you are also given a brokenSort(**a**) function that sorts and mutates only even-indexed elements in **a** in O(length(**a**)) time. For example, brokenSort(**a**) for **a** = [**7**, 99, **5**, 97, **2**, 98, **5**] changes **a** into [**2**, 99, **5**, 97, **5**, 98, **7**].

    Claim: If brokenSort(**a**) indeed runs in O(length(**a**)) time, it is possible to sort **b** in O(**N**) time with the help of the brokenSort function

a. True

b. False

14. **[3 marks]** Give your rationale for your answer to the previous question.

Ans:

We can run brokenSort(b) twice to create 2 sorted sublists within b in O(N) time. Copying out the 2 sublists and then merging them will take another O(N) time.

Algo:

```
brokenSort(b)
swap pairs of (b[i], b[i+1]) for even i where possible
brokenSort(b) again
copy out sublists of even-indexed elements and odd-indexed elements
merge the 2 sublists back into a
```

Grading Scheme: (note that Q13 and Q14 are graded independently)

3 marks for correct answer

1 mark for valid explanation/ example of how to brokenSort elements at even indexes in O(N)

1 mark for valid explanation/ example of how to brokenSort elements at odd indexes in O(N)

1 mark for explaining how to compare elements at even indexes and odd indexes to obtain a completely sorted array in O(N)

Notes:

- Approaches, where all elements of a given array b of size N are turned into elements at even indexes of a new array c of size 2N, then brokenSort(c) to sort elements of original array b, are also accepted.
- 1 mark is deducted for incorrect time complexity analysis.

Common mistakes:

- Claiming the array is completely sorted after applying brokenSort twice on the even indexes and odd indexes. For example, let b = [10, 4, 9, 2, 3]. After applying brokenSort twice, b = [3, 2, 9, 4, 10], which is not completely sorted.
- Claiming that since brokenSort(a) takes O(N), brokenSort(b) is also O(N) without further explanation. This does not explain clearly how you can completely sort a given array with the help of brokenSort.
- Radix sort is not applicable in this question since it is stated that you are given a non-empty sequence of arbitrarily large integers.
- Only after extracting out the unsorted even indexes and odd indexes into 2 arrays, brokenSort is applied on these 2 arrays. Since brokenSort only sorts elements at even indexes of a given array, this process does not guarantee that you will have 2 sorted subarrays so that you can merge them in the last step to obtain a completely sorted array.
- After applying brokenSort, claiming that applying insertion sort or optimized bubble sort will completely sort this "almost sorted" array in O(N) time.

Question 15 to 16 refers to the following problem

15. **[3 marks]** Given the following functions:

```
void A(int n) {

    if (n <= 0)
        return;
    System.out.println("*");
    B(n-1);
}

void B(int n) {
    if (n <= 0)
        return;
    System.out.println("*");
    A(n);
    B(n-1);
}
```

Claim: Calling A(n) for n > 1 will run in O(2^n) time.

a. True
b. False

16. **[3 marks]** Give the rationale for your answer to the previous question.

Ans: True.

We have a total of n layers, with layer k doing 2^{k-1} operations. From the geometric series we get that the total number of operations is $2^n - 1 = O(2^n)$.

Grading Scheme: (note that Q15 and Q16 are graded independently)

0 Marks:

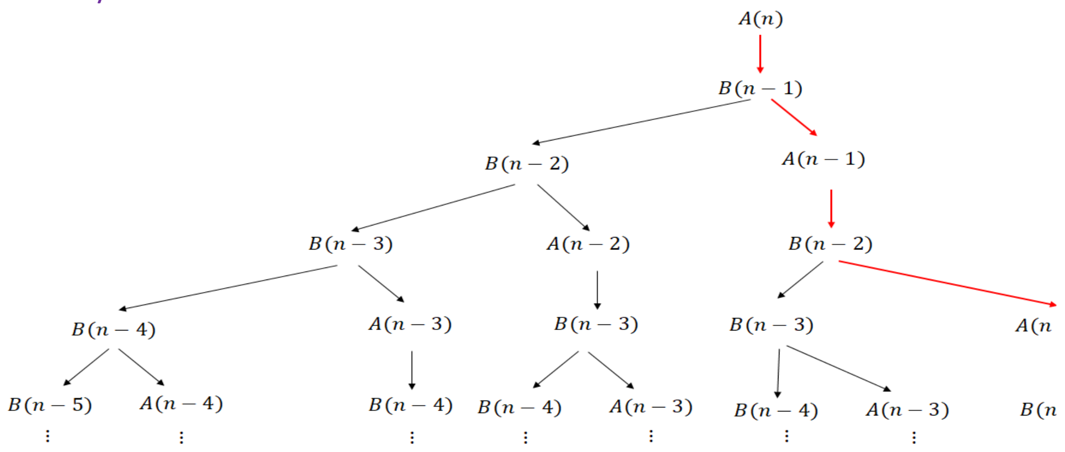
Justifying wrong time complexity

1 Mark:

Substitution and guessing patterns
Expanding recursion tree directly and claiming without justification
Incorrect understanding of Big-O
Number of nodes at i-th layer is i-th Fibonacci number

2 Marks:

Analysing the naïve tree with direct calls being children nodes and observing it is binary-like:

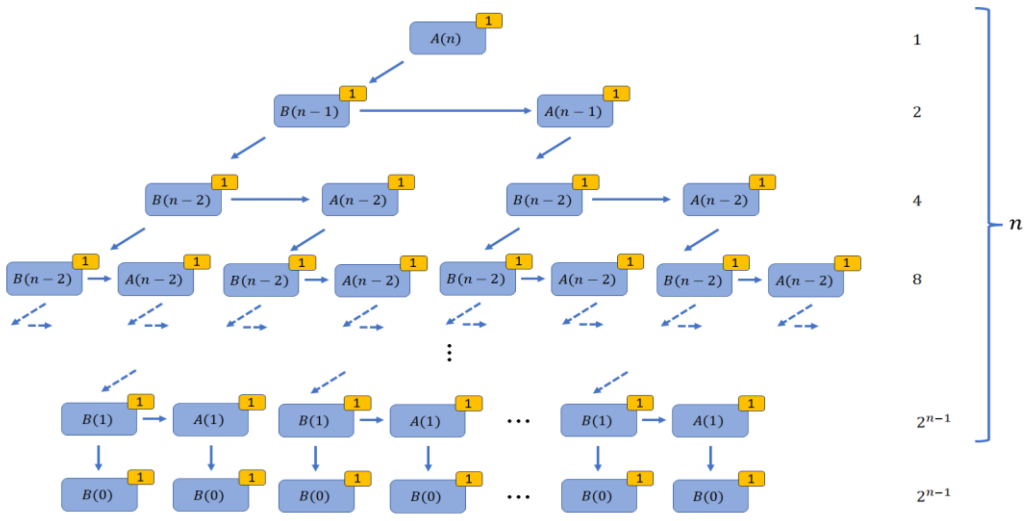


This includes answers like

- Claiming tree is binary-like, incomplete binary tree, skewed
- $A(i)$ eventually branches into 2 branches
- Upper bounded by $B(n)$ which calls 2 functions
- Nodes either branch once or twice, upper bounded by twice

The key issue with analysing this tree is that nodes at the same level have different parameters and thus terminate at different heights! This tree has a height of $2n$ and if each node either branches into 2 or 1, we get an upper bound of $O(2^{2n}) = O(4^n) \neq O(2^n)$. However, answers that correctly analyse this tree and account for the different branching factors of A, B and different heights of subtrees are awarded 3marks.

3 Marks:



Intended answer involves observing the following re-arrangement of nodes to form a complete binary tree with height n followed by an $(n+1)$-th layer of $B(0)$ calls. Note the final $B(0)$ call terminates before calling $A(0)$. Total work done is thus:

$$\underset{GP \text{ from tree}}{\frac{1(2^n-1)}{2-1}} \quad \underset{\text{Final layer}}{+2^{n-1}} \quad = 2^n + 2^{n-1} - 1 = O(2^n)$$

Other accepted answers include ***correctly*** showing manipulation to get something like $T_B(n) = 2T_B(n-1) + c$ or $T_A(n) = 2T_A(n-1) + c$ or $T_A(n) = 2T_B(n-1) + c$

Structured Questions: 4 questions

This section is worth 42 marks. Answer all questions.

Write in **pseudo-code**.

Any algorithm/data structure/data structure operation not taught in CS2040 must be described, there must be no black boxes. For the written questions partial marks will be awarded for correct answers not meeting the time complexity required.

17. **[6 marks]**

Tom keeps a record of bitcoin prices regularly. Assume the price of a bitcoin is a positive floating-point number which may be very small/large and have many digits. He is not interested in reading all the prices he has recorded, but he is only interested in the **maximum** bitcoin price among them.

Besides recording these prices, he also wants to be able to undo one or more of such recordings.

Each time Tom records a price, the operation Record(price) will be called.
Each time Tom undoes the latest recording, the operation Undo() will be called.
Each of the 2 functions should return the **maximum** bitcoin price among the prices still recorded.

Example	**Output**	**Explanation**
Record(0.0001)	0.0001	Max of [0.0001]
Record(1.23)	1.23	Max of [0.0001, 1.23]
Record(0.9)	1.23	Max of [0.0001, 1.23, 0.9]
Record(2.001)	2.001	Max of [0.0001, 1.23, 0.9, 2.001]
Undo()	1.23	Max of [0.0001, 1.23, 0.9]
Undo()	1.23	Max of [0.0001, 1.23]
Undo()	0.0001	Max of [0.0001]
Record(0.234)	0.234	Max of [0.0001, 0.234]

Select all statements about implementing Record / Undo that are correct. Assume that you can only use any ADT/data structure taught in CS2040 up to hashtables.

a. Using O(1) space throughout the entire program, **each call** of Record and Undo can be implemented using worst case O(1) time if **only the prices added by the most recent 20 Record** operations can be Undone
b. Using O(1) space throughout the entire program, **each call** of Record and Undo can be implemented using worst case O(1) time if we allow **any number of prices to be Undone**
c. A hashtable or array-based list can be used to implement Record and Undo such that **each call** takes worst case O(1) time if we allow **any number of prices to be Undone**
d. Any linked list variant can be used to implement Record and Undo such that **each call** takes worst case O(1) time if we allow **any number of prices to be Undone**

Ans: option a and d.

A Stack that stores the historical maximum price can be used. There is no need to store every single recorded price.

When Recording to an empty stack, just push the newly recorded price (which will be the maximum). Otherwise, push the maximum of the price on top of the stack (which is the historical max price so far) and the newly recorded price.

Undo just pops from the stack, restoring the previous maximum.

Option a is correct because we only need to save a doubly-linked list of the topmost 20 elements. For eg the tail end could be used as the top of the "stack", for Record and Undo operations. Once the size of the list becomes 21, we remove the oldest element from the head of the list. As the list never has more than 21 nodes, one operation takes O(1) space and O(1) time.

Option b is not correct as storing O(N) prices takes worst case O(N) space.

Option c is not correct as collision resolution in hashtable may cause an operation to take more than O(1) time in the worst case. Array-based list will occasionally take O(N) time to resize the underlying array when it is full.

Option d is correct. Singly liked list without tail pointer is good enough to function as a stack, with push and pop operations running in worst case O(1) time.

18. **[6 marks]** Ivan is playing a trading card game and has many cards. Instead of each card having a name, it has an integer ID which may be negative or positive (the values are not bounded). It is possible that Ivan has many cards having the same ID. Ivan needs to be able to perform the following 5 operations:

1. void add(ID): called when Ivan adds a card with the given ID to his collection
2. void update(oldID, newID): called in case a card with the given oldID was added by mistake. If at least one such card is found, only **ONE** such card should be changed to a card with newID instead
3. int count(ID): returns the number of cards with the given ID
4. int countCards(freq): returns the total number of cards in which Ivan has freq number of such cards
5. int countCardIDs(freq): returns the number of distinct card IDs in which Ivan has freq number of such cards

For example:

initially Ivan has cards [7, 2, 4, 2, -10]
after add(2), Ivan has cards [7, 2, 4, 2, -10, 2]
after update(2, 4), Ivan has cards [7, 4, 4, 2, -10, 2]

count(2) returns 2 since there are 2 cards with ID 2
count(3) returns 0 since there are no cards with ID 3
count(-10) returns 1 since there is 1 card with ID -10

countCards(1) returns 2 as there is 1 card with ID 7 and 1 card with ID -10
countCards(2) returns 4 as there are 2 cards with ID 4, 2 cards with ID 2
countCards(3) returns 0 as there is no card ID where Ivan has 3 cards of that ID

countCardIDs(1) returns 2 as there is 1 card with ID 7 and 1 card with ID -10
countCardIDs(2) returns 2 as there are 2 cards with ID 4 and 2 cards with ID 2 (2 and 4 are the distinct IDs)
countCardIDs(3) returns 0 as there is no card ID where Ivan has 3 cards of that ID

Which is the best approach that allows a sequence of Q operations (Q may be very large) to be performed correctly and most efficiently, starting with Ivan having an empty collection?

Assume that worst case O(log n) is not as good as average O(1) and amortized case is the same as worst case.

a. Use only a few Lists (array implementation) and some worst or amortized case O(1) time algorithms to implement each of the 5 operations
b. Use only a few Lists (array implementation) and some worst case O(log n) time algorithms to implement each of the 5 operations
c. Use only a few Lists (array implementation) and some worst case O(n) time algorithms to implement each of the 5 operations
d. Use a double-ended queue, LIFO stack, or FIFO queue (all using linked list implementation) to implement each of the 5 operations most efficiently
e. **Use only ONE hash table and few Lists (array implementation) to implement each of the 5 operations most efficiently**
f. Use only **ONE** hash table and the help of comparison-based sorting to implement each of the 5 operations such that they run in worst case O(n log n) time or better
g. Use only **ONE** List (array implementation) and the help of O(1) radix sort to implement each of the 5 operations such that they run in worst case O(1) time

Ans: e

Use a HashMap of ID (key) to freq (value), and an ArrayList of integers as a DAT of freq (key) to numDistinctCardIDs (value)

Each time a card ID's frequency is changed, it affects one key-value pair in the HashMap and up to 2 elements in the DAT. Therefore, an update operation can affect 2 key-value pairs in the HashMap and up to 4 elements in the DAT

Each add operation should first add one more 0 element to the back of the DAT (except for the first add operation which should add 2 such elements) to ensure the DAT is large enough

The countCards operation just returns freq*countCardIDs(freq), since each distinct card ID which matches always has the same number of cards

Runs in O(Q) average time

```
init()
      idToFreqMap = new HashMap<Integer, Integer>();
      freqToCardIDs = new ArrayList<Integer>(); // to distinct card
IDs
      freqToCardIDs.add(0);// so that next add op will let DAT have
                                // index 1

increaseDAT(freqKey) // private
      freqToCardIDs.set(freqKey, 1 + freqToCardIDs.get(freqKey));

decreaseDAT(freqKey) // private
      freqToCardIDs.set(freqKey, freqToCardIDs.get(freqKey) - 1);



add(ID)
      int oldFreq = count(ID);
      idToFreqMap.put(ID, 1 + oldFreq);

      freqToCardIDs.add(0); // expand DAT to prevent out of bounds
      if (oldFreq > 0) decreaseDAT(oldFreq);
      increaseDAT(1 + oldFreq);

update(oldID, newID)
      int oldFreq = count(oldID);
      if (oldID == newID || oldFreq == 0) return; // no update, or no
                                                  // such card

       // decrease freq for oldID first, update DAT
       if (oldFreq == 1)
             idToFreqMap.remove(oldID);
       else
             idToFreqMap.put(oldID, oldFreq - 1);
             increaseDAT(oldFreq - 1);
```

```
        decreaseDAT(oldFreq);

        // then increase freq for newID, update DAT
        oldFreq = count(newID);
        idToFreqMap.put(newID, 1 + oldFreq);

        if (oldFreq > 0)
             decreaseDAT(oldFreq);
        increaseDAT(1 + oldFreq);

count(ID)
        return idToFreqMap.getOrDefault(ID, 0);

countCards(freq)
        return freq * countCardIDs(freq);

countCardIDs(freq)
        return freqToCardIDs.get(freq);
```

IDs are not bounded, so creating a DAT over ID, or performing radix sort is considered very inefficient (it will not be O(1) time) thus option g is out.

Option a is not most correct because maintaining unsorted lists doesn’t let you aggregate / search by frequency or ID quickly so it cannot implement the 5 operations in worst/amortized O(1) time.

Option b is not correct because even though binary search can help with finding an element in a sorted list, insertion still requires shifting in worst case O(n) time.

Option c works but not the most efficient. One list can be used to store (ID, freq) pairs, and another to store (freq, numDistinctCardID) pairs

Option d is similar to c, still have to perform traversal each operation

Option f is worse than c due to repeated sorting. Remember that the best comparison-based sort is more expensive than linear searching an element in a list

19. **[16 marks]** You are given an array A of size N (N >= 1000) containing non-repeated integer of unbounded values. You are also given 2 indices i and j (0 < i < j < N-1 and j-i > 1) where A[i] and A[j] are already in their correct sorted position (assume sorted in increasing order), and the rest of the integers may or may not already be in their correct sorted position. You want to arrange A such that A[0] to A[i-1] contains all integers < A[i], A[i+1] to A[j-1] contains all integers > A[i] but < A[j] and A[j+1] to A[N-1] contains all integers > A[j]. Give an algorithm to do so that will take worst case O(N) time and use only extra O(1) space.
    If your algorithm uses extra O(n) space you will get half of the marks.

Ans:

1. t = A[j]
2. Swap A[i] and A[0] and call partition(A,0,N-1) where partition is the partition function of Quicksort as given in the lecture notes // O(N) time.
3. Go from A[i+1] to A[N-1] to find t. Let k be the index where it is found. // O(N)
4. Swap A[i+1] and A[k] and call partition(A,i+1,N-1) // O(N)

Grading Scheme:
Note: minor typos (e.g.: mistyped indices or variable names) are not penalized and the below scheme applies.

16 marks: Solutions that are correct, except for typos, and efficient (i.e., O(N) execution time and O(1) extra space). Possible solutions for the given problem are using 2 times a 2-way partition() algorithm, using a 3-way partition() algorithm, using 2 times quickselect() (which is basically partition()), or using a single loop with multiple if-statements that correctly identify the source **and** target indices for item swaps in the different sections.

12 marks: Solutions that were close to being correct, however with some deficiencies other than typos.

8 marks: Solutions that had > O(N) execution time and/or used > O(1) extra space. For example, solutions that used extra arrays, lists, queues, or stacks, or solutions that used comparison-based O(N log N) sorting algorithms. Also, solutions that used if-statements and identified the right source indices for swaps, but incorrect (or somewhat arbitrary) target indices. For a swap, both the source and the target indices need to be in the correct sections, especially if there is only a 1-pass loop.

4 marks: Solutions that used radix sort, which is not usable for integers of unbounded values (mentioned in the problem statement). Solutions that had very unclear descriptions, e.g., such as “inserting” a value into a section without describing how this would affect the execution time (i.e., shifting of other values?).

2 marks: Very incomplete, not understandable, solutions with many missing details. Partial solutions.

0 mark: No description or solution provided. Or the provided description cannot really be matched to the problem statement.

Common Mistakes for Q19:

- Some students realized that the partition() method (or quickselect) can be used to move data items to the left and right of a pivot. Two calls to partition() can be done for the two pivots. Some students got the boundaries of the partition() method wrong, but there was generally no marks deduction for that. Some students also designed a 3-way partition() function as an extension of the standard 2-way partition() function. These are all valid solutions.

- Some students said that they used a “modified quicksort”. However, unless it was clear from the code that it was a 1-pass quicksort, or partition() or quickselect (which all can run in O(N)), the assumption was that quicksort runs in at least O(N log N). If you say “modified quicksort” please explain how it is modified. If the modification is that the pivots are given versus the pivots are not given, this will not change the execution time.

- Some students used a single-pass solution that used if-statements to move items that were in the wrong section to the right section. These can be valid solutions with execution time O(N) and extra space of O(1). The best way to do this is with swap() operations because the array A[] is full of data and this way no extra storage is needed. The main mistake by students was that they correctly looked for the source of the swap() but then didn’t correctly identify the target location of the swap. For example, we can find an element that is on the left of A[i] with a value that is larger than A[i] (i.e., it is in the wrong section). Then we swap it with an element that is on the right of A[i]. However, the element on the right, between A[i+1] and A[N-1], needs to have a value that is **less than A[i]**. Only in this case it makes sense to swap the two elements and thus making progress. Many students just used some index starting from i+1 and incremented by +1 at each loop-step to find a target element. There is no guarantee that such an element has a value of < A[i]. This is especially important if we only do one pass through the whole array A.

- Some students used radix sort to get O(N) for sorting. However, radix sort cannot handle integers of ‘unbounded values’ (as mentioned in the problem statement).

- Some students used regular sorting algorithms (insertion sort, quicksort, mergesort, etc.). I think most students were aware that comparison-based sorting is at least O(N log N), and thus > O(N). Therefore, there was some marks deduction. Hopefully also students recognized that the problem statement does not require a full sort of the array.

Some students used additional data structures (lists, queues, stacks, etc.) and of course these use more than O(1) space. These solutions can work correctly, but marks were deducted for the additional space.

20. **[14 marks]** Mary is playing a game with N (N > 1) special magnets which are placed a small distance apart from each other in a straight line from left to right. The magnet are labelled with an ID from 1 to N (not necessarily in that order from left to right).

Given a scenario where the magnets from 1 to N are indeed placed in that order from left to right, Mary starts by using another magnet in her hand to magnetically push magnet 1 towards magnet 2.

If magnet 1 is of the same polarity as magnet 2, magnet 1 will flip around and then attach to magnet 2 so that now you have magnet 1,2 attached together in a so called **"magnet train"**.

Mary will continue to push the "magnet train" towards magnet 3 using the magnet in her hand. If now the polarity of magnet 2 and magnet 3 are opposite, magnet 3 will attach to the end of the "magnet train" to form 1,2,3.

If the next magnet which is magnet 4 is the same polarity as magnet 3, the entire train will flip before magnet 4 attaches to it to form 3,2,1,4. This is because the magnets have increasing weights from left to right so that the magnet train is always lighter than the next magnet that will attach to it, thus it will always be the train that is flipped when the same polarity is encountered.

The game will proceed until all magnets are attached into the "magnet train".

Another example is as follows:

3+,1+,4-,15-,7- (where + and - indicate the polarity of the magnets).

- First you have 3+ in the magnet train
- Second you have 3+,1+ in the magnet train (flipping 3+ is still 3+)
- Third you have 3+,1+,4- in the magnet train
- Fourth you have 4-,1+,3+,15- in the magnet train (flip 3+,1+,4- before attaching 15- to the end of the magnet train)
- Lastly you have 15-,3+,1+,4-,7- (flip 4-,1+,3+,15- before attaching 7- to the end of the magnet train)

Mary wants to know even before she plays the game what the magnets in the final magnet train will be like from left to right.
She knows that you are studying CS2040 and asks you to simulate the process given a array **A** of the magnets when each slot in the array contains a pair <x,y> where x is the magnet ID and y is the polarity which is either -1 (negative polarity) or 1 (positive polarity).

Give the most efficient algorithm **<u>using only any of the linked list variants taught</u>** (you may use multiple such linked list) to implement a function called **magnetTrain(A)** that will print out the IDs of all magnets in the final magnet train from left to right given the array **A**.

Ans:

```
1. Let D be a doubly linked list that stores the pair representing a magnet.
2. D.addLast(A[0].x)
3. flipped = false
4. for i from 1 to N-1 inclusive
      if D[i-1].y == D[i].y
        flipped = !flipped // flip the magnet train
      if flipped
        D.addFirst(A[i].x)
      else
        D.addLast(A[i].x)
5. while !D.isEmpty()
      if flipped print(D.removeLast())
      else print(D.removeFirst())
```

Step 1 to 3 takes O(1) time, step 4 takes O(N), step 5 takes O(N) time so in all it takes O(N) time.

Other ways not involving a doubly linked list but other list variants, every reverse/flip requires you have to take each of the magnet in the current train out and re-insert them. Thus if all the polarity of the magnets are the same in array A, this will take a worst case of O(N^2) time.

Grading Scheme:

14 marks for correct O(N) solution

9 marks for correct O(N^2) solution

5 marks for correct O(N^3) solution

5 marks - solution works in O(N) or O(N^2) but did not mention LL implementation of Stack/Queue

2 marks - works but uses other DS

Incorrect solutions:

5 marks: swapping head and tail references to reverse doesn't work as LL is now corrupted

3 marks: algorithm compares polarities properly, decides to add to ends of linked list but doesn't properly addFirst / addLast

0 marks: rotating linked list instead of reversing

Deductions (in cases where algorithm is not totally off):

-4 marks:

Always comparing to tail instead of last logical element

OR

same polarity requires flipping, not always adding to head

-3 marks: did not properly add back elements from stack to linked list

-2 marks: when handling same polarity case, wrong order of adding the newest magnet to the back of train if flipped

-2 marks: clearly mentioned and showed intention to reverse/flip, in O(N) time per flip, magnet train with a TLL/DLL but did not show how to (This doesn't apply to answers that use CLL/DCLL, or don't provide other steps aside from the examples already mentioned in the question)

-2 marks: need to determine whether to traverse from head or tail while printing

Common Mistakes (besides those already appearing in the marking scheme):

- Wasting time and increasing risk for making errors by using pointer manipulation instead of an already defined LinkedList operation e.g. addFirst() addLast()
- Thinking that CLL / DCLL makes it easier since there is circularity. In actual fact there are many more links to maintain. By the way, CLL doesn't need a head pointer, just having a tail pointer is good. You should have realized this from the CLL question in tutorial
- Confusing element (magnet / Pair) with Node (the object that stores the magnet/Pair and has a next attribute), or thinking that the array is pre-populated with Node objects or LinkedList objects
- Confusing LinkedList with Node, thinking that LinkedList has next/prev attributes, or thinking that Node has head/tail attributes
- Not linking backward pointers in a DLL
- Attempting to find A[i] or access index i in the linked list - Can do so if done properly, but inefficient
- Using LinkedList[i] or LinkedList.get(i) to traverse a for loop sequentially
- Attempting to reverse list using 2 stacks only (ignore recursive stack)
- Thinking that since one reverse takes O(N) time (for some inefficient implementations), the algorithm takes O(N) time. If you perform N-2 reverses, you need O(N^2) time