Quiz 1 Solutions

- Don't Panic. Often, things aren't as difficult as they may first appear.
- Write your student id *clearly* in the top right on every odd page (Do this now).
- The quiz contains 3 problems. You have 75 minutes to earn 60 points.
- The quiz contains 14 pages, including this one and 4 pages of scratch paper.
- The quiz is open book. You may not bring any magnification equipment. You may **not** use a calculator, your mobile phone, or any other electronic device.
- Write your solutions in the space provided. If you need more space, please use the scratch paper at the end of the quiz. Do not put part of the answer to one problem on a page for another problem.
- Read through the problems before starting. Do not spend too much time on any one problem.
- Show your work. Partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- If you finish early, drop off your quiz in the front, and leave quietly.
- You may not bring any part of this quiz (even the scratch pages) out of the room.
- Good luck!

Problem #	**Name**	**Possible Points**	**Achieved Points**
1	Algorithm Analysis	20	
2	Reverse Polish Calculator	20	
3	List Partitioning	20	
Total:		60	

Student id.: ______________________________
IMPORTANT: Please ensure your student id is correct and legible.
Please circle your tutorial group:

Ayush 9am-10am	Irham 10am-11am	Irham 11am-12pm	Irham 12pm-1pm	Ryan 1pm-2pm	Ryan 2pm-3pm	Enzio 3pm-4pm

Problem 1. Algorithm Analysis [20 points]

For each of the following, choose the best (tightest) asymptotic upper bound from among the given options. Some of the following may appear more than once, and some may appear not at all. Each problem is worth 5 points. **Please write the letter in the blank space beside the question.**

A. $O(1)$	B. $O(\log n)$	C. $O(n)$	D. $O(n \log n)$
E. $O(n^2)$	F. $O(n^3)$	G. $O(2^n)$	H. None of the above.

Problem 1.a.

$$T(n) = \frac{n^3 - n}{3n^2} + \left(\frac{1}{5n}\right)\left(\frac{n^2}{2}\right) \qquad T(n) = \boxed{\text{C} : O(n)}$$

Problem 1.b. The running time of the `mycode`, as a function of n:

```
public static int mycode(int n){
  return bar(n);
}

public static int bar(int n) {
  if (n <= 1) return 1;

  return fu(n);
}

public static int fu(int n) {
  if (n <= 1) return 0;
  return bar(n/2);
}
```

$T(n) =$ B: $O(\log n)$

Problem 1.c. $T(n)$ is the running time of a divide-and-conquer algorithm that divides the input of size n into two *unequal-sized parts* and recurses on both of them. The first part is of size $4n/5$ and the second part is $n/5$. It uses $O(1)$ work in dividing/recombining the two parts (and there is no other cost, i.e., no other work done). The base case for the recursion is when the input is of size 1, which costs $O(1)$.

$T(n) =$ C : $O(n)$

Problem 1.d. The running time of the following code, as a function of n:

```
public static int recursiveloops(int n){

  if (n <= 1) return 1;

  int a = doWork(n);
  return recursiveloops(a/3);
}

public static int doWork(int n) {
  int j=0;
  for (int i=0; i< n; i++) {
    j = j + 2;
  }
  return j;
}
```

$T(n) = \boxed{\text{C} : O(n)}$

Hint: Given a real number $|r| < 1$, the sum of the series $\sum_{i=0}^{\infty} r^i = \frac{1}{1-r}$.

Grader Feedback:

a. There were quite a number of attempts that wrong here. Common mistakes included not dividing the powers of n properly, resulting in final answers being of the wrong order, such as writing $\frac{n^3}{n^2} = n^2$.

b. The recurrence relation is $T(n) = T(\frac{n}{2}) + c$. Here, the function calls go from `bar(n)` to `fu(n)` to `bar(n/2)` to `fu(n/2)` to `bar(n/4)` and so on. Essentially we see that the problem size is always reducing by half in each `bar(x)` and `fu(x)` pair.

Solution presentation: Given the recurrence relation $T(n) = T(\frac{n}{2}) + c$, we get the expansion $T(n) = T(\frac{n}{4}) + c + c$ and so on, where this gives us a final summation of $T(n) = \sum_{i=1}^{\log(n)} c = \log(n)$.

c. The recurrence relation is $T(n) = T(\frac{4n}{5}) + T(\frac{n}{5}) + c$. One major thing to note here is that many attempts were not able to even write down the recurrence relation at all. Here, we are testing a recurrence of a similar form to that covered for paranoid quicksort in lecture, just that we change the work done to $O(1)$ instead of $O(n)$. The easiest way to solve this will be to use substitution. We realise that this is in fact not easy to solve given the current knowledge students have, so everyone will be given full marks for this question.

Solution presentation (Extra Content): Given the recurrence $T(n) = T(\frac{4n}{5}) + T(\frac{n}{5}) + c$, we will want to show that this is $O(n)$. The technique used here will be via substituion/induction. The solutions here will take the chance to show a faulty version first, before showing the actual version.

Faulty attempt: We make the "guess" that $T(i) \le ki$ for some positive constant k and for all $i < n$. Then proceeding by induction, we get that:

$$\begin{aligned} T(n) &= T(\frac{4n}{5}) + T(\frac{n}{5}) + c \text{ (by definition of } T(n)) \\ &\le k\frac{4n}{5} + k\frac{n}{5} + c \text{ (by our guess)} \\ &= kn + c \end{aligned}$$

and therefore we conclude $T(n) \le kn$ for some constant k. Notice this doesn't work our guess was that $T(i) \le ki$, for i smaller than n. Yet in our final equation, we seem to have gained a $+c$, which means we haven't shown our guess actually also works for n. So instead, what we are going to do, is something called "subtraction by lower order terms".

Working attempt: We instead make the "guess" that $T(i) \le ki - d$ for any $i < n$, and some positive constants k and d. Now we get:

$$\begin{aligned} T(n) &= T(\frac{4n}{5}) + T(\frac{n}{5}) + c \text{ (by definition of } T(n)) \\ &\le k\frac{4n}{5} - d + k\frac{n}{5} - d + c \text{ (by our guess)} \\ &= kn + c - 2d \\ &\le kn - d \text{ (so long as we guess } d \text{ to be larger than } c) \end{aligned}$$

Which fulfills our induction!

d. The recurrence relation is $T(n) = T(\frac{2n}{3}) + cn$. The `doWork` function does a total of n additions in the for loop. We can thus treat each call of `recursiveloops` as having $O(n)$ cost. The result of `a = doWork(n)` is also easily calculated to be $2n$, and thus the recursive call is made to `recursiveloops(2n/3).`

There are several types of mistakes made here:

- Wrongly identifying the runtime of the `doWork` function.
- Not identifying that there would be $O(\log n)$ recursive calls / levels from `recursiveloops`.
- Not being able to add up the work properly. Here, we expected students to make use of the hint given for the sum of geometric series. In fact, solving this is very similar to what is covered in the tutorial and the tutorial solutions.

Solution Presentation : By expanding the recurrence, we obtain:

$$\begin{aligned}
T(n) &= cn + \frac{2}{3}cn + \left(\frac{2}{3}\right)^2 cn + \cdots \left(\frac{2}{3}\right)^{\log_{\frac{3}{2}}(n)-1} cn \\
&= \sum_{i=0}^{\log_{\frac{3}{2}}(n)-1} \left(\frac{2}{3}\right)^i cn \\
&\leq \sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^i cn \\
&= cn \sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^i \\
&= cn \frac{1}{1-\frac{2}{3}} \\
&= \frac{3}{2}cn
\end{aligned}$$

Problem 2. Reverse Polish Calculator [20 points]

There are several ways we can write arithmetic expressions. You should be familiar with the commonly-used *infix notation*, e.g.,"$2 + 3$" (which gives 5) and "$(2 + 3) \times 2$" (which gives 10).

In Reverse Polish notation (RPN), the operator symbol is placed *after* the arguments being operated on. For example, rather than "$2 + 5$", the RPN expression is: "$2, 5, +$". RPN has the nice property that parentheses (or brackets) are not required to specify order — RPN expressions are always evaluated from left to right. For example, the arithmetic expression "$(2+3) \times 4$" in RPN is "$2, 3, +, 4, \times$". More precisely, a list is in RPN if it follows the following (recursive) rules:

1. it contains a single integer
2. it is of a form "$a, b, \circ$" where a and b are RPN expressions and $\circ$ is a (binary) operator, i.e., $+, -, \times, \div$.

Your task in this problem is to evaluate a given RPN. Rather than a string, assume that an RPN expression is provided as a *list* A of size n, and the first element is the left-most element in the RPN. Each of the elements $A[i]$ is an integer or a binary operator. Here are some more examples of infix and RPN, and how the RPN expressions are evaluated:

Infix	Reverse Polish (RPN)	Result	Remarks
$(2+3) \times 4$	$2, 3, +, 4, \times$	20	$+$ is applied to 2 and 3 to give a. Then, $\times$ is applied to a and 4.
$2 \times (3+4)$	$2, 3, 4, +, \times$	14	$+$ is applied to 3 and 4 to give a. Then, $\times$ is applied to 2 and a.
$(3+6) \div (1+2)$	$3, 6, +, 1, 2, +, \div$	3	$+$ is applied to 3 and 6 to give a. Then, $+$ is applied to 1 and 2 to give b. Finally, $\div$ is applied to a and b, i.e., $a \div b$.

Problem 2.a. [6 points] Among the data structures and ADTs we have learnt thus far, which one would be helpful in evaluating an RPN expression? Provide a brief justification for your answer.

Solution: A stack would be helpful in this case, as it can be used to recursively evaluate the expression from the innermost operation.
Feedback: Common mistakes included

- As long as you mentioned stack or any data-structure that was used like a stack (double-linked list), we gave you 4 marks. The last 2 marks were given for stating that it was used to recursively solve it, or that we used stacks for its LIFO properties.
- Using a queue instead of a stack, with the justification being that a queue would preserve the order of the operators and operands - we already have the operators and operands in a list, so we do not need an additional structure to ensure the order.
- Using a list, without stating if it is singly or doubly linked, since using a singly-linked list as a stack is non-trivial.

Problem 2.b. [7 points] **Describe the most time-efficient algorithm you can think of to evaluate the RPN A and return the result.** Assume that A is *guaranteed* to be a valid RPN expression. Be precise, but pseudocode or Java is not necessary unless it helps you to explain. You can assume you already have access to all the algorithms and data structures we have discussed in class. Unless you make a modification, you do not have to describe how the standard methods work. Write the time and space complexity of your method below:

Running time: $\boxed{O(n)}$ Space: $\boxed{O(n)}$

Your Algorithm:

Solution: Using a stack, we will look at all the elements in the RPN one by one starting from the left most element. If the current elements is an integer, we push it onto the top of the stack. Otherwise, if it is a operator denoted as **o**, we pop the top element and call it **B**. We then pop another element and call it **A**. Evaluate **A o B** and push the result back onto the stack. This is repeated until we have looked at all elements in the RPN. At the end, there will be one element left in the stack which is our final result.

Feedback: Common mistakes include:

- Reversing the order of evaluation from **A o B** to **B o A**, which will not work for non-commutative operations like division and subtraction.
- Giving space complexity as $O(1)$ instead of $O(n)$. Students who did this made the assumption that your stack will only contain at most 3 (or some other constant) amount of elements at one time. A counter-example would be a sequence like "$1, 2, 3, ..., n, +, +, +, +, +, ..., +$"/
- Assuming that the input is already inside a stack, even though question already stated that the input comes in a list. This usually resulted in wrong complexities.

Problem 2.c. [7 points] In the previous subproblem, we assumed that the list A was a valid RPN. Let us remove that assumption: **describe the most time-efficient algorithm you can think of to check if A is a valid RPN, i.e., it obeys the previously stated rules.** Be precise, but pseudocode or Java is not necessary unless it helps you to explain. You can assume you already have access to all the algorithms and data structures we have discussed in class. Unless you make a modification, you do not have to describe how the standard methods work. Write the time and space complexity of your method below:

Running time: $O(n)$ Space: $O(1)$

Your Algorithm:

Solution: The idea is the same as that of 2(b), but here we do not need to do any evaluation. Let x denote the total number of current integers and y denote the total number of current operators. For every element we see, we increase the respective value of x or y depending on whether it is an integer or operator, then we check to make sure that the inequality $y + 1 <= x$ holds. If at any time the inequality is false, it means the RPN is invalid. After all elements have been considered, we need to do one final check to make sure $y + 1 = x$ for the RPN to be valid.
Another possible solution is to only use one single counter variable. When the current element is an integer, increase the value by 1. When it is an operator, and decrease the value by 2 and check if it is negative. If it is negative, the RPN is invalid and we terminate. Otherwise we increment the value by 1 and continue. At the end, the value of the counter should be 1 for the RPN to be valid. (1 because the last element should always be an operator)

Feedback: Common mistakes include:

- Directly using the original algorithm in 2(b), and adding additional checks for validity of evaluations. This is not ideal since you will require $O(n)$ for the stack space.
- Not checking for the last equality before the end of the algorithm.
- Not checking for the inequality when considering each element in the RPN. This is more serious than the former because it is possible for the final check to be valid even though the intermediate checks are invalid.

Problem 3. List Partitioning [20 points]

As we have learnt, partitioning (or pivoting) plays a crucial role in Quicksort. We focussed mainly on sorting *arrays* in the lectures, but what if we had to apply Quicksort on a singly linked list? For the sub-problems below, assume that you are given a singly linked list A that has n integer elements; the elements may or may not be sorted. You may assume you have head and tail pointers.

Problem 3.a. [10 points] **Describe the most time-efficient algorithm you can think of to partition A.** For this sub-problem, **use the first element as the pivot**. Be precise, but pseudocode or Java is not necessary unless it helps you to explain. You can assume you already have access to all the algorithms and data structures we have discussed in class. Unless you make a modification, you do not have to describe how the standard methods work. Write the time and space complexity of your method below:

Running time: $\boxed{O(n)}$ Space: $\boxed{O(1)}$

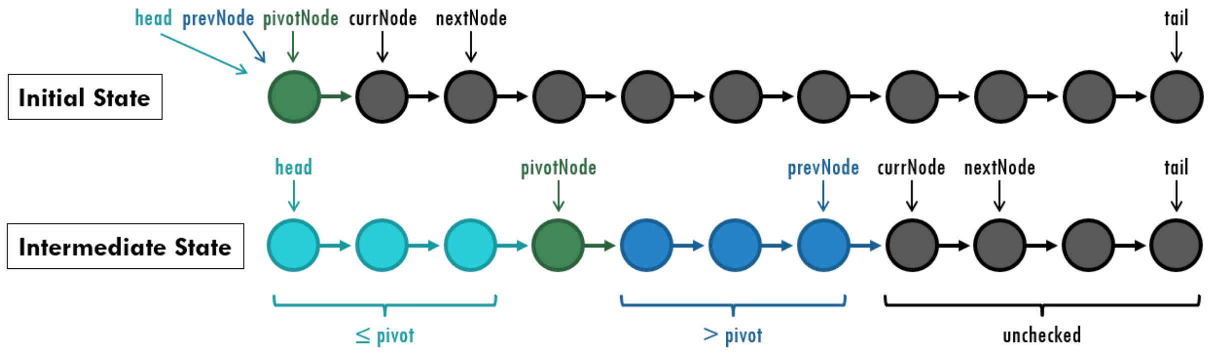
Your Algorithm:

Solution: See the next 3 pages.
Feedback: Common mistakes include:

- Some students have recursive partition algorithms, or use quicksort in their algorithms. This is a sign of a lack of understanding of how quicksort and partition works.
- Directly using the algorithm for Lomuto on arrays, or just stating the Lomuto scheme. This will not get full marks because swapping of two non-adjacent vertices in a singly-linked list is not a procedure covered during lecture, and is also non-trivial.
- Using Hoare partitioning scheme directly. This will not work because it will require you to move backwards, but this is not possible in a singly-linked list unless you iterate from the head of the list, which is $O(n)$ time and not $O(1)$.
- Missing elements or parts of the list because the swap procedure was not defined correctly.
- Creating new nodes when partitioning into two sub-lists, which makes space complexity $O(n)$.
- Some students also change the value of the pivot, or use another element other than the first as the pivot. Please do read the instructions carefully.

Possible Solution 1:

We can perform a slightly modified version of Lomuto's Partition. Instead of swapping values in an array, we are moving nodes between segments. Here, the segment of nodes with values smaller than the pivot is located before the pivot instead of between the pivot and the segment of nodes with values larger than the pivot for ease of manipulation.

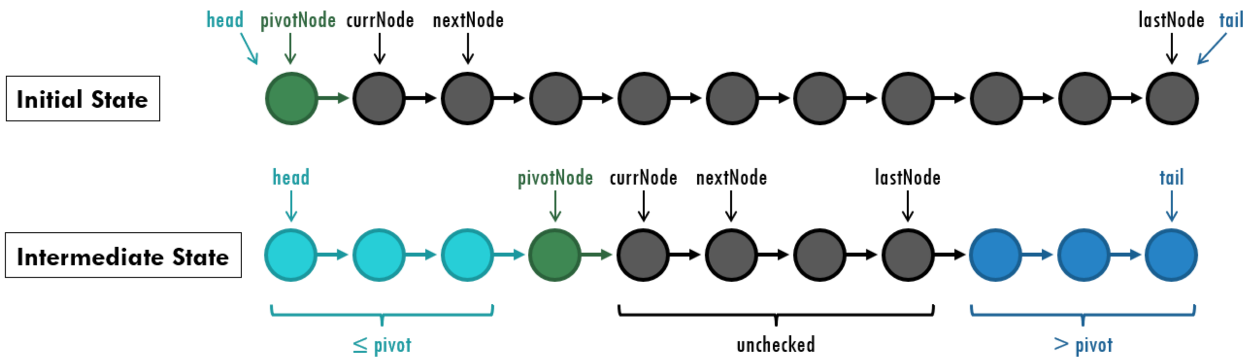


```
pivotNode = head
prevNode = head
currNode = head.next

while (currNode != null) {
    nextNode = currNode.next

    if (currNode.value <= pivotNode.value) {
        if (currNode == tail) {
            tail = prevNode
        }
        prevNode.next = currNode.next
        currNode.next = head
        head = currNode
    } else {
        prevNode = currNode
    }

    currNode = nextNode
}
```

Possible Solution 2:

We can perform a slightly modified version of Hoare's Partition. Instead of swapping values in an array, we are moving nodes between segments. Here, the segment of nodes larger than the pivot will grow forwards (rather than from the back of an array to the middle).

Note: This code assumes there is at least 1 element to partition (aside from the pivot)

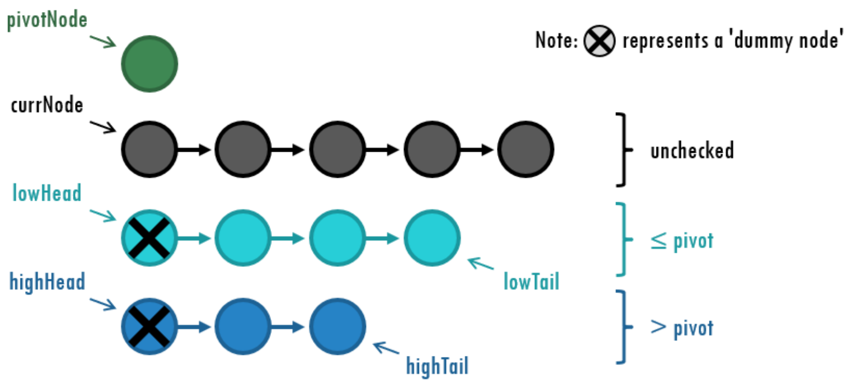


```
pivotNode = head
currNode = head.next
lastNode = tail

do {
    nextNode = currNode.next
    pivotNode.next = nextNode

    if (currNode.value <= pivotNode.value) {
        currNode.next = head
        head = currNode
    } else {
        tail.next = currNode
        tail = currNode
    }
} while (currNode != lastNode)
```

Possible Solution 3:

Instead of swapping values around, like in Lomuto's or Hoare's partition, we can simply create two new linked lists - one for the nodes with values smaller than the pivot and the other for values larger than the pivot, and simply move the nodes from the original linked list to these ones. At the end of the partition, simply join the linked lists together.

There will be dummy nodes at the start of each new linked list for ease of manipulation.



```
lowHead = lowTail = new Node()    // dummy node
highHead = highTail = new Node() // dummy node
pivotNode = head
currentNode = head.next

while (currentNode != null) {
    if (currentNode.value <= pivotNode.value) {
        lowTail.next = currentNode
        lowTail = currentNode
    } else {
        highTail.next = currentNode
        highTail = currentNode
    }

    currentNode = currentNode.next
}

head = tail = pivotNode
if (lowHead != lowTail) {
    head = lowHead.next
    lowTail.next = pivotNode
}
if (highHead != highTail) {
    pivotNode.next = highHead.next
    tail = highTail
}
```

Problem 3.b. [5 points] **Given your partitioning method in the previous subproblem, what would be worst-case time complexity of applying quicksort to a singly linked list?** Briefly justify your answer.

Worst-Case Running time: $\boxed{O(n^2)}$

Brief justification/explanation:

Solution: The worst-case time complexity of quicksort on a singly-linked list is the same as that as quicksort on an array, because the partition function in both cases takes $O(n)$ time. An example of this is we have a sorted (ascending or descending) linked list, or a linked list with all elements being the same. This results in the recurrence relation $T(n) = T(n-1) + cn$.

Feedback: Common mistakes include:

- Error-carried forward from 3(a). Students should base their arguments for 3(b) based on their answer for 3(a). Even if the answer in 3(a) is wrong, we still gave marks for 3(b) based on the time complexity that the student stated in 3(a). For example, if you gave your answer for time complexity in 3(a) as $O(n^2)$, then the answer for 3(b) should be $O(n^3)$.
- Some students were not able to identify that there is almost no change in the quicksort algorithm when the partition is $O(n)$, and that the analysis would be exactly the same as that given in lecture.

Problem 3.c. [5 points] **How would your partitioning algorithm given in problem 3.a. change if you had to pick a random element to be the pivot?** Briefly explain the changes and the impact the changes would have on the *average-case running time of using the random partitioning in Quicksort on a singly-linked list.*

Average-case Running Time: $\boxed{O(n \log n)}$

Brief justification/explanation:

Solution: As before, the average-case time complexity of quicksort on a singly-linked list is the same as that as quicksort on an array, because the partition function in both cases takes $O(n)$ time. In class, we analyzed paranoid quicksort which had the recurrence relation $T(n) = T(\frac{9n}{10}) + T(\frac{n}{10}) + cn$, which yields $O(n \log n)$.

Feedback: The common mistakes were similar to 3(b):

- Again, we attempted to not double penalize students so, answers should be based on the stated time complexity in 3(a). Some students simply stated $O(n \log n)$ despite their stated bound in 3(a).
- Some students were not able to identify there was no change since the partitioning still takes $O(n)$ time.
- Another common mistake was to give the average case running time for the partition algorithm instead of quicksort. We tried to give marks, e.g., if the overall quicksort complexity was clearly given in the explanation.