Office Use Only

MONASH University

Solutions

FIT1008 – SAMPLE paper

Semester 2 2018

Question 1: [3 marks]

What must be true of a list to perform binary search? Why?

The list must be sorted. Binary search only works as we can guarantee that where a target is not the middle (and it's in the list in the first place) then if greater than mid it's to the right and otherwise to the left. If this isn't certain then we could discover the item is to the left of mid when binary search chooses to search within the right half.

- 1 mark for sorted list
- 2 marks for valid justification
  - 1 mark where vague

Question 2: [1 marks]

Why is it not valid to treat $n = 0$ as a best case scenario for some an algorithm's time complexity?

Big O notation provides an upper bound on the behaviour of some algorithm for values of some n where n **approaches infinity** (or is suitably large or for all n greater than some initial value etc.)

$n = 0$ breaches this and hence is no longer Big O.

- 1 mark for recognising n must be large

1

Question 3: [2 marks]

What would happen if a recursive function didn't have a base case? Why?

If a function didn't have a base case, then it would never terminate; the base case is used to stop further calls to the function kind of like the condition in a while loop when attempting iteratively.

- 2 marks for clear argument
  - 1 mark where vague

2

Question 4: [2 marks]

What are the ***best*** and ***worst*** case time complexities for:

i Heap sort
ii Quick sort

i $O(nlog(n))$ for both best **and** worst case
ii (n^2) worst case and $O(nlog(n))$ best case

2

Question 5: [4 marks]

Implement the following faithfully in MIPS:

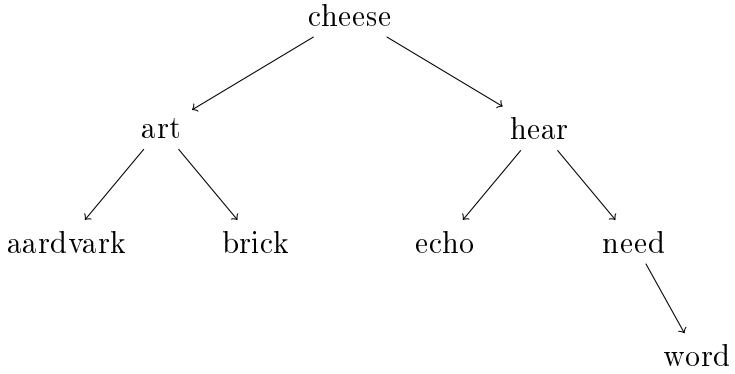
```
num = 100
while num > 0:
        num = num - 1
print(num)
```

```
.data
num: .word 100

.text
loop:
lw $t0, num
ble $t0, $0, endloop

lw $t0, num
addi $t0, $t0, -1
sw $t0, num

j loop
endloop:

lw $a0, num
addi $v0, $0, 1
syscall
```

- 0.5 marks for assigning 100 to num (either in data or text)
- 0.5 marks num is lw each time needed
- 0.5 marks num is sw each time needed
- 0.5 marks num -= 1 is implemented in MIPS
- 0.5 marks print(num) implemented in MIPS
- 0.5 marks for a branch to an end loop area
- 0.5 marks for correct unconditional jump back to start of loop
- 0.5 marks for correct condition

Question 6: [2 marks]

Show what would be printed if we output a **post-order** traversal of the binary search tree below:



In the tree above, 'cheese' is the root with left child 'art' and right child 'hear'.
'art' has a left child of 'aardvark' and a right of 'brick' 'hear' has 'echo' and 'need' as its left and right children.
'need' has a right child of 'word'

```
aardvark brick art echo word need hear cheese
```

- 2 marks for a correct postorder traversal
  - 1 mark if a different kind of traversal but otherwise correct

0.5 marks deducted for minor mistakes

2

Question 7: [2 marks]

Why should function arguments be stored in a stack-frame rather than held in registers (i.e. in MIPS)?

we only have a set number of registers that we can use for arguments, if we exceed that for some functions then those functions cannot be run in their current setup. Using the stack allows us to have an arbitrary number of arguments for any function.

- 2 marks for valid argument
  - 1 mark where vague

2

Question 8: [2 marks]

*Students of **FIT1008** should answer the following:*

Why is it that when performing a jal command, $ra gets set to PC+4?

$ra is set to PC+4 as PC was the location of the jal command so +4 ensures that after returning to $ra we run the line of code *beneath* the jal command

- 1 mark for PC being where we were (jal commend)
- 1 mark for +4 to run the line AFTERWARDS

2

Question 9: [2 marks]

*Students of **FIT1008** should answer the following:*

Provide a useful test case (input state, expected output and reason) for the __len__ method of an arrayList class.

some possible cases (for array of max capacity n):

input state	**expected**	**reason**
an empty array	0	should be sensible where no elements exist
an array of 1 item	1	should update after an item is added
an array of n items	n	a valid case just on the edge of an invalid case
an array with > n items	an error	invalid case just on boundary of valid

- 0.5 marks for a valid input state
- 0.5 marks for a valid output
- 1 mark for a clear rationale

*Students of **FIT1008** should answer the following:*

Question 10: [2 marks]

Consider the idea of a class to represent a line on the x-y plane. Give an example of a useful attribute and method for this class.

Attributes will likely be related to the coordinates on the x-y plane; e.g. *self.x*, *self.y*, *self.coords*, etc.
methods could include getting the length of the line (__*len*__), finding the intersection with another line, checking for equality (__*eq*__) with another line; basically anything useful that you could do with a line is a valid answer.

- 1 mark for a valid attribute
- 1 mark for a valid method

Question 11: [4 marks]

Consider the ***Mystery*** method (of the *linkedList* class) as defined below.

```
def Mystery(self, k):
        self._mystery_aux(self.head)

def _mystery_aux(self, current):
        S = None
        while not current is None:
                S = current.next
                if not S is None and not S.next is None:
                        S = S.next
                current.next = S
                current = current.next
```

Given the linked list below:

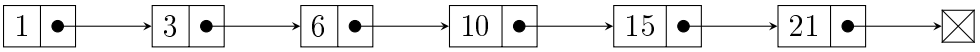
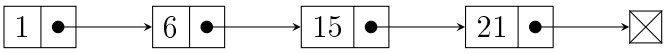


Figure 1: a linked list containing the elements 1,3,6,10,15,21,28,36

Show or describe the effect of ***Mystery*** on this list. What does ***Mystery*** do in general?



Given any linked list, ***Mystery*** will remove every even positioned element from the list (with the exception of the last element)

- 2 marks for connecting nodes with those two afterwards
  - 1 mark where inconsistent
- 1 mark for connecting 15 to 21
- 1 mark for valid suggestion of task

Question 12: [10 marks]

Consider the sequence represented by the following recurrence relation:

$$A(n) = \begin{cases} A(n-1) + 2 \times A(n-2); & n > 1 \\ 1; & n = 0, 1 \end{cases}$$

(a) (4 marks) Create an iterator (*A_iter*) capable of generating the first k elements of this sequence

For instance, if we ran the following

```
for val in A_iter(6):
    print(val)
```

We would expect the output of `1,1,3,5,11,21`

```
class A_iter:
    def __init__(self,k):
        self.k = k
        self.cPos = 0
        self.nless1 = 1
        self.nless2 = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.cPos >= self.k:
            raise StopIteration
        elif self.cPos<2:
            self.cPos+=1
            return 1
        else:
            self.cPos+=1
            computed = self.nless1 + 2*self.nless2
            self.nless2 = self.nless1
            self.nless1 = computed
            return computed
```

- 0.5 marks for correct iter definition
- 0.5 marks for setting up end of sequence in init
- 0.5 marks for appropriate initialisation in init
- 0.5 marks for raising StopIteration when sequence ended
- 1 mark for correct computation of next element in next
- 1 mark for next correctly shifting within the sequence

(b) (6 marks) Define a recursive **dynamic programming** solution to this problem (*dp_A*) which returns an array *A_table* where A_table[i] = A(i). What benefits does this have over a **naive** recursive implementation?

If you aren't sure how to apply dynamic programming in code, you should explain the approach with a written description (in enough detail that it can be followed) or as pseudocode

One possible implementation is given below:

```
def dp_A(k):
        solved = [-1]*(k)
        solved[0] = 1
        solved[1] = 1
        return dp_A_aux(k-1,solved)

def dp_A_aux(k,solved):
        if solved[k]==-1:
                kless1 = dp_A_aux(k-1,solved)
                kless2 = dp_A_aux(k-2,solved)
                solved[k] = kless1+2*kless2
        return solved[k]
```

In a dynamic programming approach we would have to solve the subproblems first and use this to solve later ones *rather than* recomputing whereas in the naive recursive implementation we **must** recompute *A(n)* each time it is triggered, hence the complexity changes from exponential (each call triggers 2 calls but the number of levels is n) to linear (any call results in an immediate return rather than further recursion if already computed – each *A(n)* is computed exactly once)

- 4 marks for dynamic programming approach
  - 1 mark first checking if solution already computed
  - 1 mark for using existing values to compute new values
  - 1 mark for updating memorised solutions when available
  - 1 mark otherwise correct implementation
- 2 marks for clear comparison with naive recursion
  - 1 mark where vague

Question 13: [6 marks]

Abby Sträctip is developing an online taxi manager for the hotel chain *RoomE*. The purpose is to allow hotel guests to request a taxi from their hotel room (with time and destination). In the system Abby wants a structure to hold each guest organised so that they can be easily retrieved in order of departure time (with no particular order for taxis departing at the same time). Guests are then added to the structure **when** they make a booking and removed from the structure at the time of their taxi's departure.

Abby is planning for at most 3400 guests to be in the system at once (the maximum capacity of the hotel) but it's likely there will be significantly fewer than this in actually (perhaps a few hundred at once).

(a) (3 marks) What kind of ADT is appropriate in this situation? Justify your answer.

In this case, it seems that items are to be added in order of creation but removed in order of departure time (which won't necessarily match the order they are added in). As such we want a structure which can efficiently organise them by a different key to their append order. Hence a standard queue is not appropriate but a priority queue (implemented as a minheap) would be.

In a priority queue (heap) we are able to add in new elements in $O(log(n))$ time where n is the number of guests currently in the heap and we are also able to extract out the next guest (by departure time) in $O(log(n))$ time. We are expecting the same number of appends as serves (everyone added must eventually be removed). Other options could be a sorted list ($O(n)$ add/remove due to need to shuffle elements, $O(1)$ remove if implemented via linked nodes but potentially still $O(n)$ insertion. Another option is an unsorted list which is $O(1)$ to add a guest but $O(n)$ to find the next guest in number of guests currently in the structure. A stack and queue won't work at all as it's not possible to extract guests based on departure time. Implementing a BST is also not appropriate as we can't guarantee the balance is appropriate to maintain logarithmic time (without concepts not covered in this unit). All in all, the best time possible consistently is logarithmic hence the heap is the best choice.

- 1 mark for priority queue/heap
- 2 marks for valid justification
  - 1 mark where vague

(b) (3 marks) Should this ADT be implemented using an array, nodes, or would either be equally appropriate? Justify you answer.

either would be appropriate in terms of a heap for correctness and time efficiency as both support logarithmic time complexity (a node could define left child, right child and parent or handle accessing the parent via returning from recursive calls) however the memory cost would differ between array and linked representation.

Arrays are fixed size, which means that the full 3400 places must be planned for in advance despite not all places being required the full time. So if typical use is a few hundred (say 300) that's well under 10% usage of the space allocated; if the same was done using nodes, we would allocate for exactly the number needed rather than the excessive overallocation here. Any given node requires more space than any given cell in an array (by a factor of number of attributes – e.g. 3 to 4 in this case) however unless the typical utilisation becomes closer to a quarter or a third, the node representation is better.

- 1 mark for nodes preferred
- 2 marks for valid justification
  - 1 mark where vague

3

Question 14: [6 marks]

Consider the task of computing the 'width' of nodes in a binary search tree. In this case, the 'width' refers to the difference between the minimum and maximum values in a given subtree. For instance, given a tree as below

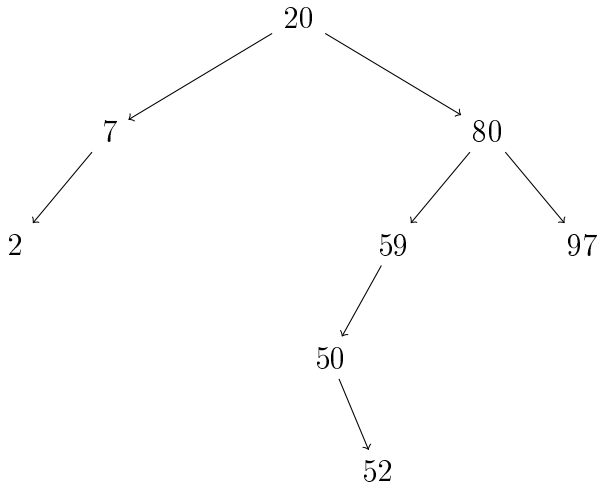


Figure 2: binary search tree containing $20, 7, 80, 2, 59, 97, 50, 52$

The root node (20) would have a 'width' of 95 as 2 is the minimum element that is a descendent of 20 and 97 is the maximum element which is a descendent of 20. In the same way, 7 has a width of 5 as the minimum element is 2 but 7 has no larger children so we use the 7 itself. Any nodes with no children have a 'width' of 0 as the minimum and maximum element in their subtree is the node itself.

(a) (4 marks) Prepare a recursive method ***findWidth*** of the ***Binary Search Tree*** class which determines the 'width' of each node of the tree and assigns this as a property to that node when computed.

```
def findWidth(self):
        self.findWidthAux(self.root)

def findWidthAux(self, current):
        if current.left is None and current.right is None:
                current.width = 0
                return [current.value, current.value]
        else:
                minVal = current.value
                if not current.left is None:
                        minVal = self.findWidthAux(current.left)[0]
                maxVal = current.value
                if not current.right is None:
```

```
                maxVal = self.findWidthAux(current.right)[1]
            current.width = (maxVal - minVal)
            return [minVal, maxVal]
```

- 0.5 marks for starting at the root
- 1 mark for finding the min and max for any subtree
- 0.5 marks for assigning the difference in min,max to that node
- 1 mark for using the current nodes value where a right or left doesn't exist
- 0.5 marks correct recursive application of findWidth to children
- 0.5 marks correct base case

0

(b) (2 marks) What is the **worst case** complexity of the recursive function you prepared in 14(a)? Justify your answer. If you make any assumptions about the shape of the tree, you should mention these in your answer. ***No marks without explanation***

Regardless of their implementation, if it is correct it will need to visit every single node in the tree at least once making the minimum complexity $O(n)$ where n is the number of nodes in the tree. Each node triggers the computation for both of their children and computes in $O(1)$ the width to assign to that node based on the result of recursive calls. As the work involved in each recursive call is $O(1)$ and there are no additional triggers of calls on nodes once they have been already seen there is a single call per node, hence $O(n)$.

If their code is particularly complicated (e.g they run a findMin and findMax on each node before triggering findWidth on the children) they may end up with something performing $O(n)$ work with each recursive call for $O(n)$ recursive calls in the case of an unbalanced tree yielding $O(n^2)$ worst case and $O(nlogn)$ best case where the tree is balanced so findMin and findMax become $O(logN)$ but are still performed on each node.

- 2 marks for a valid and correctly justified complexity matching their implementation
  - 1 mark justification is vague or incomplete

2

Question 15: [6 marks]

In managing hash tables, we often refer to both

1. primary clustering and
2. secondary clustering

What are these? Is it possible to reduce the impact of them? Justify your answer.

1. primary clustering is where two items with a different key and hash value get stuck in the same probe chain (joining somewhere later in the chain)
2. secondary clustering is where two items hash to the same position and get stuck in the same probe chain.

Primary clustering can be mitigated with careful choice of collision resolution method - e.g. something where the position in the chain has an impact on where the next place to go is (such as with quadratic probing);

Resolving secondary clustering can typically only be done by improving the quality of the hash function, as a poor hash function can result in multiple different keys hashing to the same position (such as a hash function only using the first character in a string's ascii value rather than considering the full key). Further, use of double hashing can result in better results as a different hash function is used where any collision occur, this should mean that the new hash is different to the hash for the key we are colliding with so that they follow different paths.

both can also be mitigated by increasing the size of the table, in the former this results in most probe chains being shorter in general (as secondary clustering is reduced) which reduces the chances of encountering another probe chain in your first hashed position

For each of primary and secondary clustering (6 marks in total)...

- 1 mark for explaining the type of clustering
- 2 marks for a valid resolution
  - 1 mark where vague

*Students of **FIT1008** should answer the following:*

Question 16: [6 marks]

Explain in words and/or with the aid of a diagram the best and worst case time complexities of merge sort.

in merge sort each level is either a split or a merge; the splits occur all the way until the base case of len 1 (or 0) lists and the merges occur symettrically with the same elements which were split originally. As such there are as many splits as there are merges – the same number of levels e.g. recursive depth in each direction.

the number of levels (recursive depth) depends on how long it takes to get from size N to size 1 and since each split cuts the items exactly in half, the depth will be how many times N can be halved, which is the same as what power of 2 it is;
$N = 2^{depth}$
$log_2(N) = log_2(2^{depth}) = depth$
hence the depth is $O(log_2(N))$.

At each level on splitting we perform $O(1)$ work as the split is trivial but the merge must consider every single item over the full level as each must be moved into an ordered place within that sublist so during the merge we do $O(N)$ work over the full list so we have $O(log_2(N))$ levels with $O(N)$ work per level (for merges) hence the complexity is $O(Nlog_2(N))$

In this case, the best and worst case complexity are the same as regardless of the ordering of the input list, each merge stage must still consider each element in the list for the full level (there is no early termination) and the split and merge stage are always done for each level in the same way.

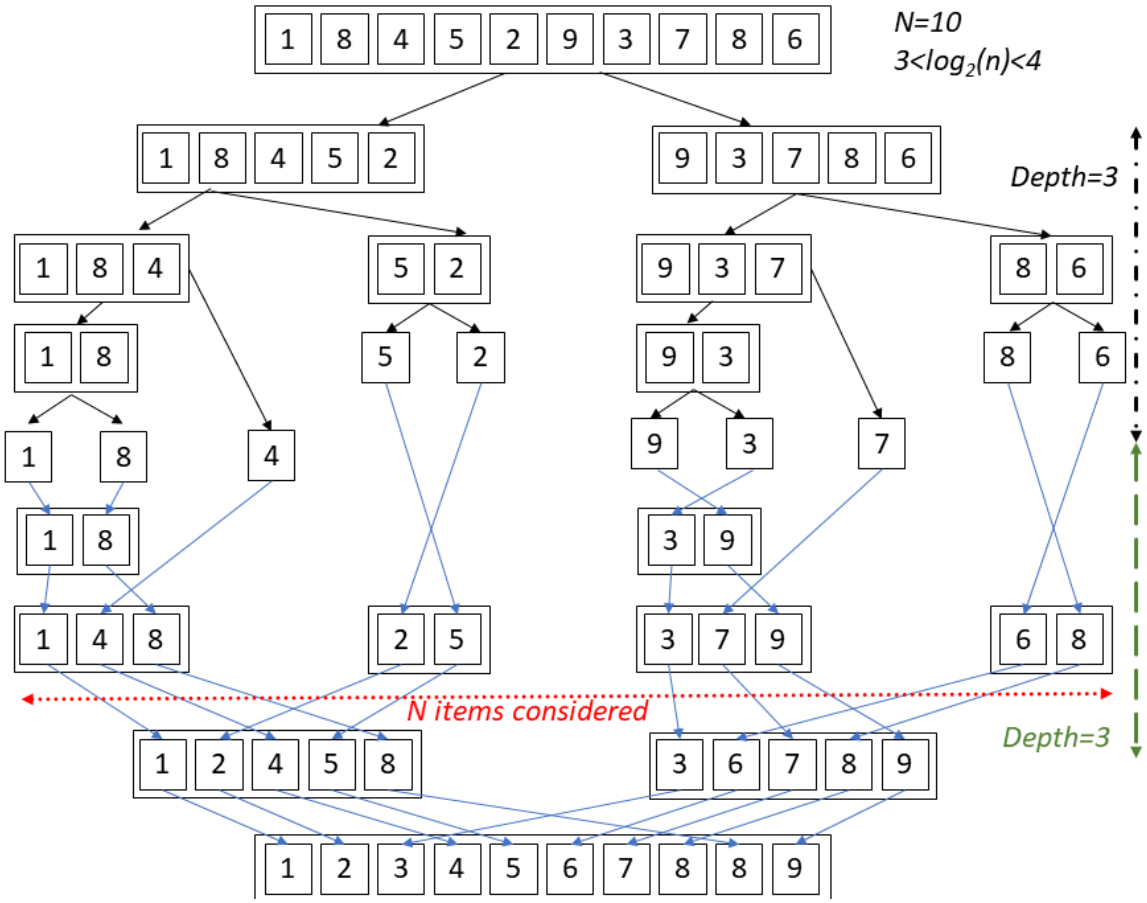


image of recursive levels of merge sort for the list $1, 8, 4, 5, 2, 9, 3, 7, 8, 6$

- 2 marks for explaining the work involved in merging two lists
  - 1 mark where vague
- 2 marks for explaining the recursive depth
  - 1 mark where vague
- 2 marks for explaining why the best and worst case are the same
  - 1 mark where vague

0

End of Exam

0

This page intentionally left blank.
Answers on this page will not be marked.