CS2040S — Data Structures and Algorithms
School of Computing
National University of Singapore

Midterm Assessment

10 Mar 2022 **Time allowed:** 2 hours

Instructions — please read carefully:

1. Do not open the midterm until you are directed to do so.
2. Read **all** the instructions first.
3. The quiz is closed book. You may bring one double-sided sheet of A4 paper to the quiz. (You may not bring any magnification equipment!) You may NOT use a calculator, your mobile phone, or any other electronic device.
4. The **QUESTION SET** comprises **TEN (10) questions** and **TWENTY (20) pages**, and the **ANSWER SHEET** comprises of **SEVEN (7) pages**.
5. The time allowed for solving this test is **2 hours**.
6. The maximum score of this test is **100 marks**. The weight of each question is given in square brackets beside the question number.
7. All questions must be answered correctly for the maximum score to be attained.
8. All questions must be answered in the space provided in the **ANSWER SHEET**; no extra sheets will be accepted as answers.
9. You must submit only the **ANSWER SHEET** and no other documents. The question set may be used as scratch paper.
10. An excerpt of the question may be provided above the answer box. It is to aid you to answer in the correct box and is not the exact question. You should refer to the original question in the question booklet.
11. You are allowed to use pencils, ball-pens or fountain pens, as you like as long as it is legible (no red color, please).
12. Unless otherwise stated in the question, when we ask for the worst-case big-O running time of an algorithm we always mean to give the tightest possible answer.
13. Unless otherwise stated in the question, we will assume that operators behave as per Java (e.g., 5/2 will evaluate to 2). However, pseudocode does not necessarily satisfy Java syntax (unless stated otherwise) and things that do not compile as legal Java are not necessarily bugs (as long as they are clear pseudocode).
14. Unless otherwise stated in the question, we are not concerned with overflow errors, e.g., storing the value $2^{245,546,983}$ in an integer.

GOOD LUCK!

This page is intentionally left blank.

It may be used as scratch paper.

Question 1: Fruit Jumble [12 marks]

The first column in the table below contains an unsorted list of words. The last column contains a sorted list of words. Each intermediate column contains a partially sorted list.

Each intermediate column was constructed by beginning with the unsorted list at the left and running one of the sorting algorithms that we learned about in class, stopping at some point before it finishes. Each algorithm is executed exactly as described in the lecture notes. One column has been sorted using a fake sorting algorithm. (Recursive algorithms recurse on the left half of the array before the right half. QuickSort uses the first element as the pivot and uses in-place 2-way partitioning.)

Unsorted	**A**	**B**	**C**	**D**	**E**	**F**	**Sorted**
Elderberry	Apple	Banana	Apple	Banana	Apple	Banana	Apple
Orange	Banana	Elderberry	Cherry	Elderberry	Banana	Elderberry	Banana
Guava	Cherry	Guava	Banana	Guava	Elderberry	Guava	Cherry
Banana	Durian	Honeydew	Durian	Honeydew	Guava	Orange	Durian
Honeydew	Honeydew	Orange	Elderberry	Imbe	Honeydew	Honeydew	Elderberry
Mango	Mango	Mango	Mango	Jackfruit	Mango	Imbe	Fig
Imbe	Imbe	Imbe	Imbe	Kiwi	Imbe	Mango	Guava
Papaya	Papaya	Papaya	Papaya	Fig	Jackfruit	Papaya	Honeydew
Jackfruit	Jackfruit	Jackfruit	Jackfruit	Lychee	Kiwi	Jackfruit	Imbe
Kiwi	Kiwi	Kiwi	Kiwi	Durian	Fig	Kiwi	Jackfruit
Fig	Fig	Fig	Fig	Cherry	Lychee	Fig	Kiwi
Lychee	Lychee	Lychee	Lychee	Mango	Durian	Lychee	Lychee
Durian	Orange	Durian	Honeydew	Apple	Cherry	Durian	Mango
Cherry	Guava	Cherry	Guava	Nectarine	Nectarine	Cherry	Nectarine
Nectarine	Nectarine	Nectarine	Nectarine	Orange	Orange	Nectarine	Orange
Apple	Elderberry	Apple	Orange	Papaya	Papaya	Apple	Papaya
Unsorted	**A**	**B**	**C**	**D**	**E**	**F**	**Sorted**

Identify, below, which column was (partially) sorted with which of the following algorithms:

1. BubbleSort
2. SelectionSort
3. InsertionSort
4. MergeSort
5. QuickSort (with first element pivot)
6. None of the above.

Hint: Do not just execute each sorting algorithm, step-by-step, until it matches one of the columns. Instead, think about the invariants that are true at every step of the sorting algorithm.

A. What sort was used on Column A? [2 marks]

B. What sort was used on Column B? [2 marks]

C. What sort was used on Column C? [2 marks]

D. What sort was used on Column D? [2 marks]

E. What sort was used on Column E? [2 marks]

F. What sort was used on Column F? [2 marks]

Question 2: Asymptotically Approaching Answers [8 marks]

A. Choose the tightest possible bound for the following function:

$$T(n) = 24n^2 \log^2(n) + 1.7 \log^3 n$$

1. $O(n)$
2. $O(n \log n)$
3. $O(n \log^2 n)$
4. $O(n^2)$
5. $O(n^3)$
6. $O(2^n)$

[3 marks]

B. $\log_2 n = O(\log_3 n)$: True or False? [2 marks]

C. Suppose you have developed a new algorithm `NeuralDeepCatFinder`, and you have run some experiments to see how long it takes on different sized inputs. You observe the following, where N is the input size and time is measured in seconds:

N	time
10,000	0.2 seconds
20,000	1.2 seconds
40,000	3.9 seconds
80,000	16.1 seconds
160,000	63.8 seconds

Based on this experimental data, you conclude that your algorithm has what asymptotic complexity? (Choose the best answer. You may assume that you were testing the algorithm on the worst case inputs.)

1. $O(\sqrt{n})$
2. $O(n)$
3. $O(n \log n)$
4. $O(n^2)$
5. $O(n^3)$
6. $O(2^n)$

[3 marks]

Question 3: Recurring Recurrences [12 marks]

A. Solve the recurrence: $T(n) \leq T(n/4) + n$, where $T(n) = 3$ for all $n < 8$. Choose the tightest possible bound.

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n \log n)$
5. $O(n^2)$
6. $O(n^3)$
7. $O(2^n)$

[3 marks]

B. Which of the following algorithms is best described by the recurrence: $T(n) \leq T(n/2) + n$:

1. Binary Search
2. MergeSort
3. QuickSelect (where the pivot is exactly the median value)
4. QuickSort (where the pivot is exactly the median value)
5. SelectionSort
6. None of the above.

[3 marks]

C. Which recurrence best describes the following algorithm? (Assume that $A[u \ldots v]$ creates a new array containing a copy of only the slots in the range $[u, v]$. Do not worry about the correctness of the algorithm.)

```
loopyloop(A[0..n-1]){
    if (n < 17) return 42;
    x = n/3;
    y = 2n/3;
    for (i = 0; i < x; i++)
        A[i] = 42;
    for (j = x+1; j < n; j++)
        A[j] = 42;
    return loopyloop(A[x+1...y])
 }
```

1. $T(n) \leq T(n/2) + O(1)$.
2. $T(n) \leq T(n/2) + O(n)$.
3. $T(n) \leq T(n/3) + O(1)$.
4. $T(n) \leq T(n/3) + O(n)$.

5. $T(n) \leq 3T(n/3) + O(n)$.
6. $T(n) \leq T(n/3) + O(n^2)$.
7. None of the above.

[3 marks]

D. What is the asymptotic running time of the following code, as a function of n, when you execute `doubleLoopy(n)`? (Give the tightest bound possible.)

```
public int doubleLoopy(int n){
    int k = 0;
    for (int i=0; i<n; i++)
        for (int j=1; j<n; j=2*j)
            k += j;
    return k;
}
```

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(n)$
4. $\Theta(n \log n)$
5. $\Theta(n^2)$
6. $\Theta(n^3)$
7. $\Theta(2^n)$

[3 marks]

Question 4: How fast is it? [12 marks]

For each of the following algorithms, what is the running time? Assume that each data strucutre is implemented as described in class (without any additional augmentation, aside from what is specified). For randomized algorithms, specify the expected running time. Choose the best (tightest) asymptotic function from among the following options. (Some choices may be used more than once, while others may not be used at all.)

1. $O(1)$
2. $O(\log n)$
3. $O(\sqrt{n})$
4. $O(n)$
5. $O(n \log n)$
6. $O(n^2)$
7. $O(n^3)$
8. $O(2^n)$

A. Listing all the elements in a binary search tree. (Note: there is no assumption that the tree is balanced.)

[2 marks]

B. Finding the median element in an AVL tree.

[2 marks]

C. The expected running time of QuickSort on a sorted list.

[2 marks]

D. The expected running time of QuickSelect on a sorted list.

[2 marks]

E. The expected query time in a hash table with chaining where the table size $m = n/\log n$, where n is the number of keys in the table.

[2 marks]

F. The worst-case number of rotations after a deletion in an AVL tree.

[2 marks]

Question 5: Where can I find the answer? [9 marks]

For all the questions in this section, we will assume (unless otherwise stated) that the input arrays are sorted, and that the array is 0-index and of size $n > 0$. We will assume that integer division follows Java rules. You should make no assumption on the items in the array, e.g., there may be duplicate values, etc. There are also no assumptions about the key, e.g., as to whether the key k is in the array, etc. A search routine for a given key k in an array A should return the index of the element in the array, if it exists, and -1 otherwise.

A. Consider the following code for a version of binary search. The strings SSS and TTT represent code that you need to determine. The search is invokedy initially by executing `myGoodSearch(A, key, 0, n-1)` on an array of size n. The algorithm is considered correct if it efficiently returns the correct answer, while not doing anything that would cause an exception or an error during the execution.

```
myGoodSearch(A, key, start, end)
1. if (SSS) then
        if (A[TTT] == key) then return TTT
        else return -1
2. mid = start + (end-start)/2
3. if (key > A[mid]) return myGoodSearch(A, key, mid+1, end)
4. else if (key < A[mid]) return myGoodSearch(A, key, start, mid-1)
5. else if (A[mid] == key) return mid
6. else return -1
```

Consider the following possible expressiongs for SSS and TTT:

I. SSS = `(start > end)`, TTT = `start`

II. SSS = `(start >= end)`, TTT = `end`

III. SSS = `(start == end)`, TTT = `start`

Which of the above lead to a correct implementation of binary search?

1. Only I.
2. Only II.
3. Only III.
4. I and II.
5. I and III.
6. All three: I and II and III.
7. None of the listed options.

[3 marks]

B. What if we want to perform a binary search in an array that is almost sorted, but not perfectly sorted? Here we consider one possible assumption:

> Array A is not necessarily sorted, but a guaranteed precondition is that for all $j < n-1$, for all $k \geq j+1$: $A[j] \leq A[k]+1$.

This means that item j is not guaranteed to be smaller than everything after it in the array, but it is guaranteed to be at most 1 larger. For example, here is an array that satisfies this condition:

$$A = [1,5,4,6,9,10,9,12,15]$$

Notice that $A[1] = 5$ and $A[2] = 4$, so they are out of order, but $A[1] \leq A[2] + 1$.

Can you search for a key in this type of almost sorted array (using a variant of binary search) in sub-linear time?

1. Yes, a relatively simple variant of binary search still works in sub-linear time.
2. No, binary search does not work. But we can still find the maximum item in the array using a variant of the peak finding algorithm in sub-linear time.
3. No, it is impossible to solve in sub-linear time.

[3 marks]

C. As we are not sure whether we can sort the almost sorted list from the previous part, we now consider a different type of almost sorted list. Assume that the array satisfies the following property:

> Array A is not necessarily sorted, but a guaranteed precondition is that for all $j < n-2$, $A[j] < A[j+2]$.

The following is an example of an array that satisfies that requirement:

$$A = [3,2,8,7,23,21]$$

For example, $A[0] = 3$ and $A[1] = 2$, so $A[0] > A[1]$, but $A[0] < A[2] = 8$.

Consider the following binary search algorithm that is intended to work for this type of almost sorted array.

```
almostBinarySearch(A, key, start, end)
1. while (start < end-1):
2.      mid = start + (end-start)/2
3.      if key == A[mid] then return mid
4.      else if key == A[mid-1] then return mid-1
5.      else if key == A[mid+1] then return mid+1
6.      else if key < A[mid] or key < A[mid+1], then end = mid-1
7.      else if key > A[mid-1] or key > A[mid], then start = mid+1
8.      else return -1
9. return -1
```

We will now try to prove this correct. For the purpose of the proof, assume that the key k is somewhere in the array A. We will try to prove by induction the following invariant:

> For some j where `start` $\leq j \leq$ `end`, `A[j] = k`.

Because we have assumed that key k is in the array initially, it is true when the execution begins. Consider the following proof of the inductive step which consists of three statements:

1. If key k is in slot $A[mid]$ or $A[mid+1]$ or $A[mid-1]$, then the execution ends while our invariant still holds (because *mid* is always between *start* and *end*).
2. If $A[j] = k$ where $j \leq mid-2$ (i.e., j is in the first half), then $A[j] \leq A[j+2] \leq \ldots A[mid]$ or $A[j] \leq A[j+2] \leq \ldots A[mid+1]$ (depending on whether the distance from j to mid is even or odd). Therefore, whenever we update $end = mid-1$ on line 6, the invariant still holds.
3. If $A[j] = k$ where $j \geq mid+2$ (i.e., j is in the second half), then $A[j] \geq A[j-2] \geq \ldots A[mid]$ or $A[j] \geq A[j-2] \geq \ldots A[mid-1]$ (depending on whether the distance from j to mid is even or odd). Therefore, whenever we update $start = mid+1$ on line 7, the invariant still holds.

Since either way in which we modify start and end maintains the invariant, we conclude that the invariant is maintained.

Which of the following is the most accurate statement about this proof. (Note that the goal here is to show that the search will find element k if it is in the array. It does not matter what happens if the element is not in the array, or if the algorithm does not terminate, or if it is slower than expected.)

1. Statement 1 is the first statement in the proof that is not true. (Statements 2 and 3 may or may not be true.)
2. Statements 2 is the first statement in the proof that is not true. (Statement 3 may or may not be true.)
3. Statement 3 is the first statement in the proof that is not true.
4. All three statements in the proof are true. However, proving that the invariant is maintained is not sufficient to prove that the search will find the key k if it is in the array.
5. All three statements in the proof are true. The invariant is correct, and the algorithm will successfully find key k if it is in the array.

[3 marks]

Question 6: Putting everything in order [12 marks]

Consider sorting an array that consists of only two different values. (For example, the array $[4,2,4,2,4,4,2,2,2,4,4]$ only contains 2's and 4's.) What is the asymptotic running time of each of the following algorithms? Give the tightest bound possible. Assume the algorithm is executed as presented in class (and not optimized for this special case).

1. $O(1)$
2. $O(\log n)$
3. $O(\sqrt{n})$
4. $O(n)$
5. $O(n \log n)$
6. $O(n^2)$
7. $O(n^3)$
8. $O(2^n)$

A. BubbleSort:

[2 marks]

B. SelectionSort:

[2 marks]

C. InsertionSort:

[2 marks]

D. MergeSort:

[2 marks]

E. QuickSort (using 3-way partitioning, pivot chosen at random, expected running time):

[2 marks]

F. Assume that the following array has just been partitioned used some in-place partitioning algorithm:

14	1	2	5	42	1430	132	16796	4862	429

Which of the following values could have been the value of the pivot for this partition operation? (Just for fun: do you know where these numbers come from?)

1. 14
2. 5
3. 42
4. 1430
5. 132
6. 16796
7. 4862
8. None of the above.

[2 marks]

Question 7: Ash, Birch, Chestnut, Dogwood, Elm [17 marks]

A. In the best case, inserting into an AVL tree containing n keys may take $< \log n$ time.[1] True or false?

1. True
2. False

[2 marks]

B. Which of the following possible invariants for an AVL Tree is true? (Remember that a leaf has height 0.)

I. After every insertion or deletion, for every node u at height $h(u)$, the subtree rooted at u contains at least $2^{h(u)}$ nodes (including u itself).

II. After every insertion or deletion, for every node u at height $h(u)$, the subtree rooted at u contains at most $2^{h(u)}$ nodes (including u itself).

III. After every insertion or deletion, for every node u at height $h(u)$, the subtree rooted at u contains at least $2^{h(u)+1}$ nodes (including u itself).

IV. After every insertion or deletion, for every node u at height $h(u)$, the subtree rooted at u contains at most $2^{h(u)+1}$ nodes (including u itself).

1. I only.
2. I and III.
3. II and IV.
4. IV only.
5. I and II.
6. III and IV.
7. None of the four possible invariants are correct.

[2 marks]

C. The depth of a node u in a tree is the number of edges on the path from the root to the node. (More precisely, define the depth of the root to be zero; the depth of node u is equal to `1 + depth(u.parent)`.) If u and v are two leaves in an AVL tree, then their depth differs by at most 10. True or false?

1. True
2. False

[2 marks]

D. A FibTree is defined as follows: FibTree(1) consists of a single node. FibTree(2) consists of two nodes: the root node, and one left child. FibTree(n) consists of: a root node, which

[1] Remember, when we use $\log n$ without specifying, we always mean base-2, i.e., $\log_2 n$.

has a left subtree of FibTree($n-1$) and a right subtree of FibTree($n-2$). Every FibTree is height-balanced, by definition. True or false?

1. True
2. False

[2 marks]

E. A 3AVL-tree is a binary tree with a slightly different balance property. We say that a node u is 3h-good if the height of each child of u is at least $height(u)-3$. In a 3AVL-tree, every node is 3h-good. What is the worst-case height of a 3AVL tree? Let $h(n)$ be the maximum height of a 3AVL tree with n nodes. Choose the tightest bound. (None of the answers is exact, but one is derived in a similar manner to the approximation used in class for AVL trees.)

1. $h(n) \leq \log_2(n)+3$
2. $h(n) \leq 2(\log_3(n)+1)$
3. $h(n) \leq 3(\log_2(n)+1)$
4. $h(n) \leq (\log_2(n))^2+3$
5. $h(n) \leq (\log_2(n))^3+3$
6. $h(n) \leq 2n^{1/3}+3$
7. $h(n) \leq 3n^{1/2}+3$
8. $h(n) \leq n$

[2 marks]

F. Assume you have an empty AVL and insert keys in the following order:

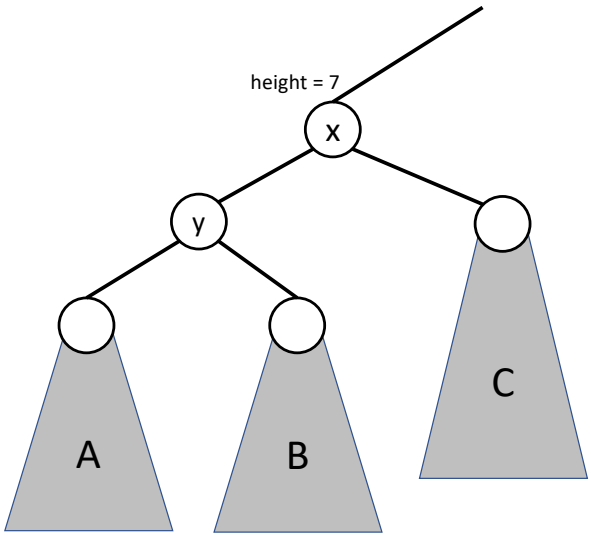
$$1,2,3,4,5,6$$

Which key is the root after all six insertions complete?

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6

[3 marks]

G. For this question and the next one, consider the following AVL tree:



In this tree, A, B, and C are unknown subtrees. The node with key x is the left child of the root and has height 7. The left child of x has key y. When a new key $w < x$ is inserted, node x is the lowest node in the tree that is out of balance, and a double-rotation occurs.

What do we know about the height of node y before the insertion occurs?

1. Node y has height 5.
2. Node y has height 6.
3. Node y has height 7.
4. We cannot determine the height of node y from the information given.

[2 marks]

H. Referring to the same scenario as the previous question, what do we know about key w?

1. Key $w < y$.
2. Key $w > y$.
3. We cannot determine the relationship of w to y from the information given.

[2 marks]

Question 8: Making a hash of things [6 marks]

A. A symbol table (the abstract data type) can be implemented with an AVL tree, true or false?

1. True
2. False

[2 marks]

B. A dictionary (the abstract data type) can be implemented with a hash table, true or false?

1. True
2. False

[2 marks]

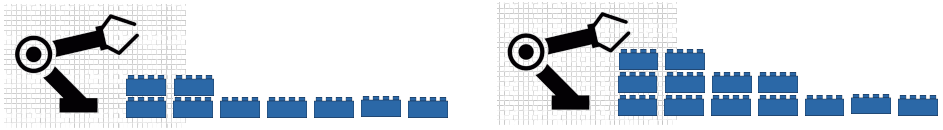
C. If three items are inserted into a hash table of size m and the hash function satisfies the simple uniform hashing assumption, what is the probability of at least one collision?

1. $1/m^2$
2. $2/m$
3. $2/m - 1/m^2$
4. $2/m^2$
5. $3/m$
6. $3/m^2$
7. $3/m - 2/m^2$
8. None of the above.

[2 marks]

Question 9: 3D Printing [12 marks]

Tom Swift has built a 3d-printer for building structures out of lego bricks. The printer consists of a robotic arm that adds one layer of legos at a time to a given row of legos. Unfortunately, it is a bit limited right now. It only moves along one axis (from left to right), and it can only add bricks in a sequence from the end of the row to a given point to the right.



Luckily, programming this printer is easy: you only need to specify the size of the row to print. For example, the structure above on the left was built by the following two commands:

(build, 7), (build, 2)

Each piece is dropped on top of the previous one, so the structure above on the right was built by the following three commands:

(build, 7), (build, 2), (build, 4)

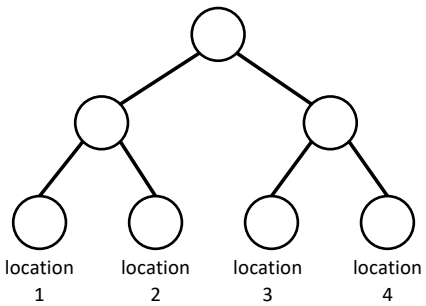
That is, by doing a (build, 4) on the structure on the left, we get the structure on the right.

Our goal is to design a data structure that will determine the height at any given point along the line after a sequence of build commands. That is, the data structure supports two operations:

- `build(distance)`: Specifies a build command.
- `height(location)`: Returns the height of the tower at the specified location.

Note that distances are 1-indexed, i.e., 1 is the location of the first lego. We say that a `build(x)` command includes location d if $d \leq x$, i.e., the build adds a lego to location d. A `build(x)` commands ends at location x.

To implement this, we build a balanced binary tree, augmented with a little bit of extra information. Assume the tree is perfect, i.e., for any non-leaf node u, the two subtrees have exactly the same number of nodes. Each leaf in the tree represents a location, i.e., the leftmost leaf represents location 1, its sibling represents location 2, etc. There are n locations, and the tree has n leaves. (We can assume that n is a power of 2.)



In the following four subparts, we are going to design the algorithm for implementing this data structure. You should look at all four parts before choosing your answers; the correct answers

are the ones that yield a complete correct algorithm. (Some wrong answers may be part of a different correct algorithm, but will not be compatible with the other parts.)

A. We augment each node u in the tree to contain a value `u.legos`. The invariant that we maintain is that after each build operation, which of the following is always true:

1. `u.legos` = number of build commands that include all the leaves in the subtree rooted at node u.
2. `u.legos` = number of build commands that include at least one leaf in the subtree rooted at node u.
3. `u.legos` = number of build commands that end in the subtree rooted at node u.
4. `u.legos` = number of build commands that include all the leaves in the left subtree rooted at node u.
5. `u.legos` = number of build commands that include at least one leaf in the left subtree rooted at node u.
6. `u.legos` = number of build commands that end in the left subtree rooted at node u.
7. `u.legos` = number of build commands that include all the leaves in the right subtree rooted at node u.
8. `u.legos` = number of build commands that include at least one leaf in the right subtree rooted at node u.
9. `u.legos` = number of build commands that end in the right subtree rooted at node u.

[3 marks]

B. Imagine that Tom Swift tries to build a tower on a strip with maximum distance 16. (Assume the balanced binary tree has 16 leaves.) Tom issues the following sequence of commands:

(build, 7)

(build, 7)

(build, 11)

(build, 4)

(build, 4)

(build, 4)

(Remember that locations are 1-indexed.) What is the value stored in the root node `u.legos`?

1. `u.legos` = 7
2. `u.legos` = 6
3. `u.legos` = 5
4. `u.legos` = 4

5. `u.legos` = 3
6. `u.legos` = 2
7. `u.legos` = 1
8. `u.legos` = 0

[3 marks]

C. The following is the algorithm for `build(d)`. Certain values (x, y, z) are omitted.

```
build(d)
1. Traverse the tree to the leaf at location d.  Call that node u.
2. u.legos = u.legos + x;
3. while u is not the root:
4.      v = u.parent
5.      if u is a left child of v, then v.legos = v.legos + y
6.      if u is a right child of v, then v.legos = v.legos + z
7.      u = v
```

Which of the follow values for (x, y, z) yields a correct implementation?

1. $(x, y, z) = (0, 1, 0)$
2. $(x, y, z) = (0, 0, 1)$
3. $(x, y, z) = (0, 1, 1)$
4. $(x, y, z) = (1, 1, 0)$
5. $(x, y, z) = (1, 0, 1)$
6. $(x, y, z) = (1, 1, 1)$
7. $(x, y, z) = (1, 0,$ `u.legos`$)$
8. $(x, y, z) = (1,$ `u.legos`$, 0)$

[3 marks]

D. The following is the algorithm for `height(d)`. Certain values (y, z) are omitted.

```
height(d)
1. Traverse the tree to the leaf at location d.  Call that node u.
2. total = u.legos;
3. while u is not the root:
4.      v = u.parent
5.      if u is a left child of v, then total = total + y
6.      if u is a right child of v, then total = total + z
7.      u = v
8.      return total
```

Which of the follow values for (y, z) yields a correct implementation?

1. $(y, z) = (1, 0)$

2. $(y,z) = (0,1)$
3. $(y,z) = (1,1)$
4. $(y,z) = (\texttt{u.legos}, 0)$
5. $(y,z) = (0, \texttt{u.legos})$
6. $(y,z) = (\texttt{u.legos}, \texttt{u.legos})$
7. $(y,z) = (\texttt{v.legos}, 0)$
8. $(y,z) = (0, \texttt{v.legos})$
9. $(y,z) = (\texttt{v.legos}, \texttt{v.legos})$

[3 marks]