AY9018-19 Term 1 Test
November 9018

CS602 Algorithm Design and Implementation

Instructors: DAI Bing Tian & LAU Hoong Chuin

Student Name: _____

SMU Email: _____

INSTRUCTIONS TO STUDENTS

1. The time allowed for this test paper is **2 hours**.
2. You are only allowed to reference a double-sided A4-size cheat sheet.
3. You are NOT allowed to use Internet or any other messaging tools.
4. You are allowed to use a non-programmable and non-graphical scientific calculator.
5. You are required to answer **ALL** questions and hand in all sheets including cheat sheet and rough paper.

Question	Mark	Question	Mark
1		3	
2		4	
Total (80 marks)			

This page is intentionally left blank.

Question 1

(a) (8 marks) True or False: Circle Your Answers.

i. [T/F] All NP-complete problems are NP-hard.

ii. [T/F] The Master Theorem can be used to derive a tight time complexity of any divide and conquer algorithm so long as its recurrence can be stated in the form T(n) = a T(n/b) + f(n) for any function f(n).

iii. [T/F] If an algorithm has a time complexity $\Theta(n)$, then we can also say it is O(n).

iv. [T/F] If P≠NP, then there exist problems that cannot be solved in polynomial time.

(b) (4 marks) Using the Master Theorem, solve the following recurrences:

i. $T(n) = 8\,T(n/2) + n \log n$

ii. $T(n) = 3\,T(n/3) + n^2$

(c) (8 marks) Suppose problems A is in P, and B and C are in NP. Under what condition is

(i) $A \leq_p B$

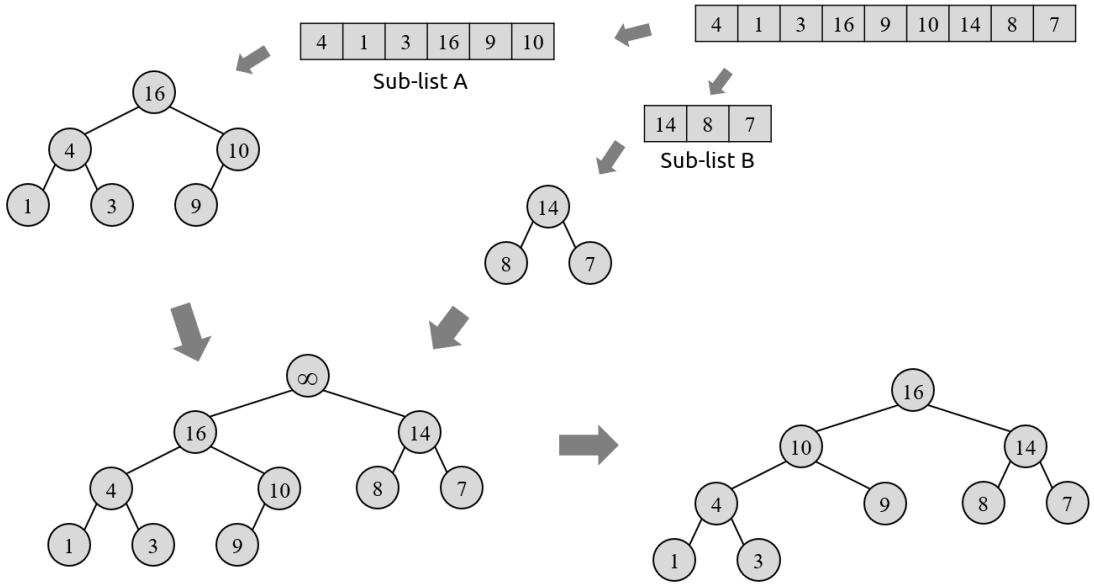
(ii) $B \leq_p C$

Question 2

After learning binary trees, Timothy thinks the heapify process discussed in class is too complicated, and would like to build a divide-and-conquer algorithm for the heapify process that constructs a binary tree from a list with max-heap properties. This is what he plans to do given a list of positive integers in arbitrary order:

1. Divide the list into a sub-list A on the left and a sub-list B (containing the remaining elements) on the right.
2. Heapify sub-list A and sub-list B by the same divide-and-conquer heapify process recursively. Let S and T be the root element of the heaps constructed from sub-lists A and B respectively.
3. Construct a new node with key math.inf, left child S and right child T.
4. Pop the maximum element from the heap.
5. The resultant heap is a binary tree with max-heap properties. It can be converted to a list by BFS if necessary.

The base case for the above recursive process is as simple as to construct a singleton node for a list with only one element. For example, with the list [4, 1, 3, 16, 9, 10, 14, 8, 7]. We may construct sub-lists A and B as [4, 1, 3, 16, 9, 10] and [14, 8, 7] to construct two smaller heaps, and then merge them into one under the node with key infinity. After a pop operation, we get a binary tree as shown at the bottom right corner with max-heap properties.

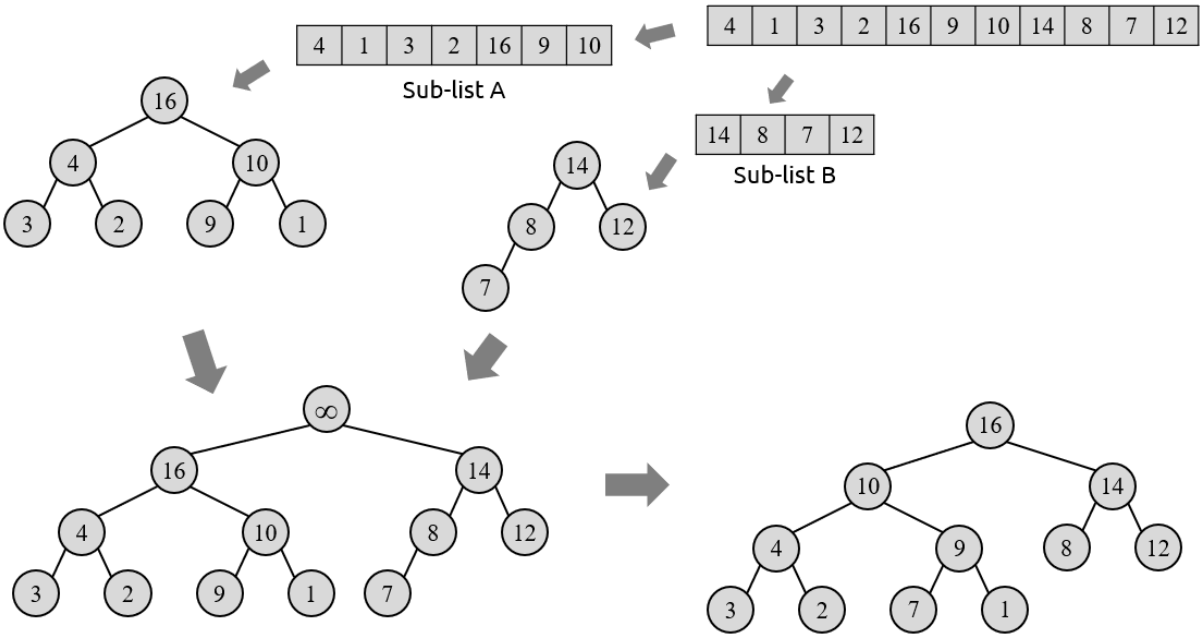


(a) (2 marks) There are two major steps in a typical divide-and-conquer algorithm: divide and combine. Some algorithms take more effort in the "divide" step, while others take more in the "combine" step. In this algorithm, which step takes more effort?

____divide____ / ____combine____ (circle one of the two)

(b) (8 marks) Timothy soon realizes it is not always the case that a full binary tree is constructed from sub-list B. It is also possible to have a full binary tree representing sub-list A, as shown in the figure below. However, at least one tree must be a full binary tree. Otherwise, it fails to construct a complete binary tree at step 3. Therefore, it is crucial to decide how many elements are to be divided into sub-list A. Implement a function "len_of_left_side" to determine the number of elements in the sub-list A. Remaining elements will be partitioned into sub-list B.

```
def dnc_heapify(lst):
    n = len(lst)
    if n == 1:
        return Node(lst[0])
    m = len_of_left_side(n)
    A, B = lst[:m], lst[m:]
    S, T = dnc_heapify(A), dnc_heapify(B)
    R = Node(math.inf, lchild = S, rchild = T)
    R.pop()
    return R
```



(c) (3 marks) Analyze in big-O notation the complexity of the function “len_of_left_side” that you implemented.

(d) (7 marks) Analyze in big-O notation the complexity of the function “dnc_heapify” from your analysis in part (c).

Question 3

Design an algorithm to compute the number of ways to make up a change of n cents, given m coin denominations $\{d_1, d_2, \ldots, d_m\}$ in cents. For example, given coin denominations {1, 2, 5}, the ways to make up $n = 7$ cents are {1, 1, 1, 1, 1, 1, 1}, {1, 1, 1, 1, 2, 1}, {5, 2}, {2, 1, 2, 1, 1}, {1, 5, 1}, {2, 2, 1, 2}, so there are 6 ways to make up 7 cents. Note that the sequence of coins in each set does not matter, so each set is considered one way to make up 7 cents.

(a) (3 marks) What are the subproblems if you plan to solve this question with dynamic programming?

(b) (1 mark) Do subproblems overlap?

(c) (6 marks) Formulate the recurrence relationship to solve this problem.

(d) (3 marks) Point out the overlapping part of the recurrence relationship.

(e) (3 marks) What is the time complexity of your algorithm?

(f) (4 marks) The algorithm below is provided by ChatGPT to solve this problem. Do you think this is a correct algorithm? If you think so, justify this algorithm, otherwise, give a counter example to show this algorithm does not work.

```
def count_change_ways(denominations, target):
    # Sort denominations in descending order
    denominations.sort(reverse=True)

    # Initialize count of ways to make change
    ways = 0

    # Iterate through each denomination
    for coin in denominations:
        # Choose the maximum number of times the coin can be used
        while target >= coin:
            target -= coin
            # If target becomes 0, we found a way to make change
            if target == 0:
                ways += 1

    return ways

# Example usage:
denominations = [1, 2, 5]  # Set of denominations
target = 10  # Target amount
ways = count_change_ways(denominations, target)
print("Number of ways to make change:", ways)
```

Question 4

Given a set of *n* items, each with a weight w_i and a value v_i for $1 \leq i \leq n$, determine the items to include in a collection so that the total weight is less than or equal to a given limit *L* and the total value is as large as possible.

Study the algorithm below and comments about its correctness and efficiency.

```
w = [1, 3, 7, 10, 14, 19]
v = [10, 40, 80, 100, 200, 350]
n, L = len(w), 22
r = [0] * n
best_collection, curr_best = 0, 0

def f(m, p, q, s, t):
    global w, v, n, L, r, best_collection, curr_best
    if q == m:
        if t >= curr_best:
            best_collection = r[:m].copy()
            curr_best = t
    else:
        for i in range(p, n - m + q + 1):
            if s + w[i] <= L:
                r[q] = i
                if t + v[i] >= curr_best:
                    best_collection = r[:q+1].copy()
                    curr_best = t + v[i]
                f(m, i+1, q+1, s + w[i], t + v[i])

for i in range(1, n+1):
    f(i, 0, 0, 0, 0)

print('best collection:', best_collection)
```

(a) (1 mark) Is this algorithm correct?

(b) (2 marks) Identify the redundant part of the algorithm if there is any. Answer “Nil” if there is no redundancy.

(c) (3 marks) What is the complexity of this algorithm?

(d) (3 marks) Give a tuple of w, v and L such that the algorithm runs in the worst case complexity.

(e) (9 marks) If you were to design a dynamic programming algorithm to solve this problem, what is the recurrence relationship for the dynamic programming algorithm? Write down the recurrence relationship and explain your algorithm. If a table is used in your algorithm, draw the table with the input in the code.

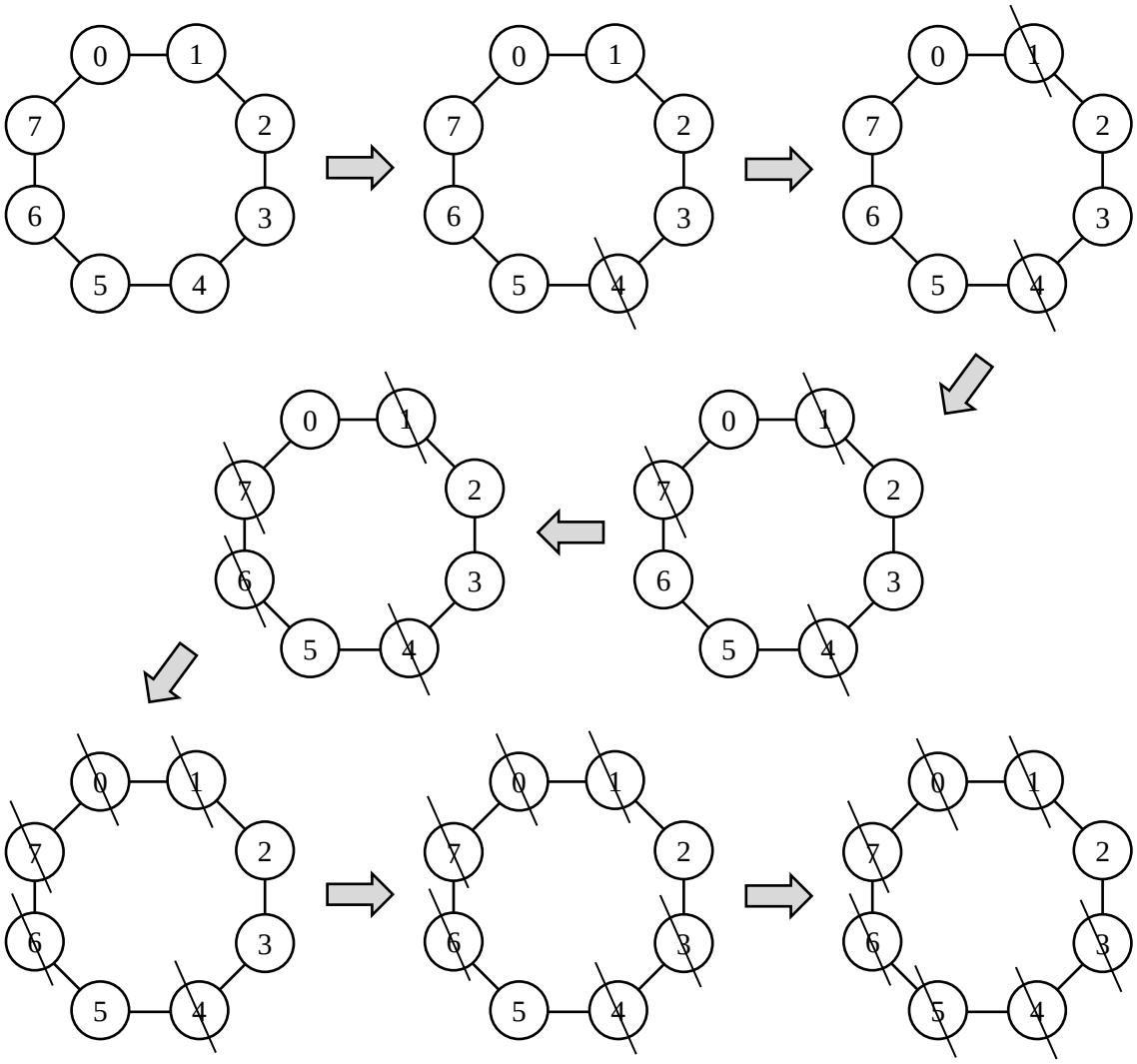
(f) (2 marks) What is the result of your dynamic programming with the input in the code?

Question 5

[Only 4 questions in the exam, this is an extra question on recursion, on the difficult side.]

This is a sad story. An explosion happened in a space station and the auxiliary oxygen system can only support one astronaut to go back to the earth. However, there were *n* astronauts in the space station, so they decided to draw lots with a special number to determine the lucky guy. They sat down in a circle clockwise facing the center of the circle, with the first astronaut at position 0 and the last astronaut at position *n-1*. For every position *p*, the position on the left is *(p+1)%n* and the position on the right is *(p-1)%n*. They then randomly selected an integer *m*, and started counting with this special number *m*. The first astronaut (position 0) counted number 1, the second astronaut (position 1) counted number 2, until one of them counted number *m*, who would then selflessly suicide to give other astronauts chances to survive. After that, they started counting number 1 from the next astronaut (who is the first alive on the left side of the one who just suicided) and chose the next astronaut to suicide. The draw process stopped when there was only one astronaut left.

The figure below is an illustration for *n* = 8 and *m* = 5.



As the only computer scientist among all astronauts, you told everyone this is a deterministic process and you can design an algorithm to quickly decide who would be the lucky astronaut given the special number m.

(a) (10 marks) Design and describe an O(nm) algorithm to decide the final lucky astronaut. You may use data structures, pseudocode or formulae to help describe the algorithm.

(b) (10 marks) When number m is much larger than n, the above algorithm is inefficient. Design and describe an O(n) algorithm to decide the final lucky astronaut. You may use data structures, pseudocode or formulae to help describe the algorithm.