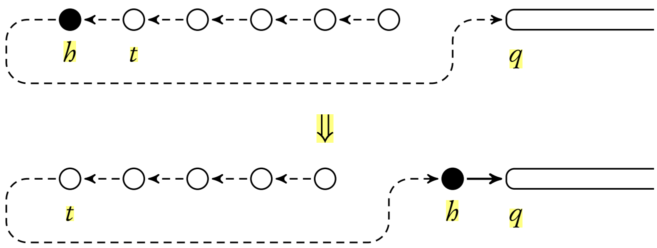
A zero mark will be given
if you copy someone else's
work or you let someone
copy your work.

COMP122/22-10 Data Structures and Algorithms	0 – 67 points
Recursion and Tail Recursion	2022-02-24 *Due Date* — 2022-03-03
Class Code	
Student No.	DO NOT WRITE YOUR NAME

1. To reverse a singly linked list without creating new nodes, we can keep detaching the head node and pushing it to another initially empty list. This can be done recursively as follows,

   - detach the head node h of the first list and mark the tail as t,
   - link h to the head node of the second list, and
   - recursively reverse t with the updated second list.



a) Write a tail-recursive function $reverse(h, q)$ to reverse the nodes of linked list h and push them to linked list q. The function returns the new head node of the reversed linked list. 6 points

```
def reverse(h, q):


    if h is None:                 1
        return q                  1
    else:
        t = h.nxt                 1
        h.nxt = q                 1
        return reverse(t, h)      2
```

(1) .

b) Convert the tail-recursion to a loop and define a function $reverse\_i(h)$ using this loop to reverse linked list h. Notice that q is the accumulator and initially empty, so we can use it as a local variable with the loop. 7 points

```
def reverse_i(h):


    q = None                      1
    while h is not None:          1
        t = h.nxt                 1
        h.nxt = q                 1
        q, h = h, t               2
    return q                      1
```

(2) .

2. Based on the above tail recursion scheme in Question 1, we can split a linked list in a similar way, with two accumulators.

   a) Write a function *reverse_cut*(h, i, j, p, q) to cut a sub-linked list out of a linked list h, the sub-linked list consists of the nodes of h from index i to index j, the node at index j is not included. The nodes of the sub-linked list must be reversely joined to the front of the accumulator linked list p, and the rest of nodes of h must be reversely joined to the other accumulator q. The function returns the pair of the sub-linked list and the remaining linked list. For example, if h is `0->1->2->3->4->5-/`, *reverse_cut*(h, 2, 5) updates p and q to `4->3->2->`p and `5->1->0->`q. You can assume $0 \leqslant i \leqslant j \leqslant count(h)$. (11 points)

```
def reverse_cut(h, i, j, p, q):

        if h is None:                     1
            return (p, q)                 1
        else:
            t = h.nxt                     1
            if i == 0 and j > 0:          2
                h.nxt = p                 1
                return reverse_cut(t, 0, j-1, h, q)
                                          3
            else:
                h.nxt = q                 1
                return reverse_cut(t, i-1, j-1, p, h)
                                          1
```

(3)

   b) Convert the tail-recursion to a loop and define a function *reverse_cut_i*(h, i, j). Notice that p and q are both the accumulators and initially empty, so we can use them as local variables with the loop. (12 points)

```
def reverse_cut_i(h, i, j):

        p = q = None                      1
        while h is not None:              1
            t = h.nxt                     1
            if i == 0 and j > 0:          2
                h.nxt = p                 1
                p, h = h, t
                j = j-1                   3
            else:
                h.nxt = q                 1
                q, h = h, t
                i, j = i-1, j-1           1
        return (p, q)                     1
```

(4)

3. Let s be a list of n unique elements. To generate all the combinations of r elements ($0 \leqslant r \leqslant n$) from s, we consider the following analysis.

   - If $r = n$, we have to choose all the elements as the only combination.
   - If $r = 0$, we have to choose no element as the only empty combination.

- Otherwise, we have $1 \leqslant r \leqslant n-1$. For the head element in s, say h, we have two cases. Joining the combinations from the two cases gives us the full answer.
  (i) We include h in the combinations, and we must choose $r-1$ elements from the remaining $n-1$ elements. This is a smaller problem, we can do it recursively.
  (ii) We don't include h in the combinations, thus we must choose r elements from the remaining $n-1$ elements. This is also a smaller problem, we can do it recursively.

Write a recursive generator function *combinations*(*s*, *r*) to generate all the combinations of r elements from s, each as a list of length r. You must keep the original order of the chosen elements in the combinations. 12 points

```
def combinations(s, r):
```

```
if r == 0:                                          1
    yield []                                        1
elif r == len(s):                                   1
    yield [*s]                                      1
else:
    h, *t = s                                       2
    yield from ([h, *c] for c in combinations(t, r-1))
                                                    4
    yield from combinations(t, r)                   2
```

(5)

4. Let s be a list of n unique elements. A *displacement* of s is a permutation of all the elements in s such that no element is at its original position. For example, if s is `['Ada', 'Bob', 'Cara']`, then `['Bob', 'Cara', 'Ada']` is a displacement, but `['Bob', 'Ada', 'Cara']` is not. To generate all the displacements of s, we consider the following analysis.

- If $n=0$, we have only one empty displacement.
- If $n=1$, we have no displacement.
- Otherwise, we have $n \geqslant 2$. For the head element s_0, we must relocate it to somewhere else and put one of the remaining $n-1$ elements to the head position. Then, for each chosen new head element s_i, we have two cases. Joining the displacements from the two cases gives us the full answer.
  (i) We place s_0 to position i, and we must displace the remaining $n-2$ elements. This is a smaller problem, we can do it recursively.
  (ii) We don't place s_0 to position i, therefore we must also displace s_0 away from position i, thus we must displace all the remaining $n-1$ elements. This is also a smaller problem, we can do it recursively.

a) Define a recursive function $f(n)$, in the form as on Page 7 of Lesson 10, to compute the number of displacements for a given list of n unique elements. 6 points

$$f(n) = \begin{cases} 1 & \text{if } n = 0, \\ 0 & \text{if } n = 1, \\ (n-1)[f(n-2)+f(n-1)] & \text{if } n \geqslant 2. \end{cases}$$

(6)

b) Write a recursive generator function *displacements*(s) to yield all the displacements of list s. You need to think about how to construct the sublists to displace, and how to join the elements that have been placed to the displacements of the sublists. 13 points

```
def displacements(s):

        if not s:                                  1
            yield []                               1
        else:
            for i in range(1, len(s)):             2
                yield from (
                    [s[i], *t[:i-1], s[0], *t[i-1:]] for t in
                                                   3
                    displacements(s[1:i]+s[i+1:]))
                                                   2
                yield from (
                    [s[i], *t] for t in            2
                    displacements([*s[1:i], s[0], *s[i+1:]]))
                                                   2
```

(7) .