Data Structure and Algorithm

Midterm Reference Solution

TA email: dsa1@csie.ntu.edu.tw

Problem 1. To prove $\log_2 n! = \Theta(n \log n)$, it suffices to show $\exists N \in \mathbb{N}, c_1, c_2 > 0$ such that

$$c_1 n \ln n \leq \ln n! \leq c_2 n \ln n \tag{1}$$

for all $n \geq N$, since constants can be ignored when considering asymptotic complexity.

For the second inequality, since

$$\ln n! = \sum_{k=1}^{n} \ln k \leq \sum_{k=1}^{n} \ln n = n \ln n \tag{2}$$

for $n \geq 1$, we can take $c_2 = 1$ and the constraint $N \geq 1$ is trivial.

For the trickier first inequality, calculus come in handy! Remember that the upper sum is greater than the area under the curve $y = \ln x$, i.e.

$$\ln n! = \sum_{k=1}^{n} \ln k \geq \int_1^n \ln x dx = (x \ln x - x)|_1^n = n \ln n - n + 1 \tag{3}$$

for $n \geq 1$, with the help of integration by parts. Now we want $n \ln n - n + 1 \geq c_1 n \ln n$, or equivalently $(1 - c_1) n \ln n \geq n - 1$, for $n \geq N$. If we take $c_1 = 0.5$ (actually all $0 < c_1 < 1$ are fine, and note that different c_1 result in different N_0 below), then what we want becomes $n \ln n \geq 2n - 2$, for $n \geq N$. We already know (and are familiar with) that $n \ln n = \omega(n)$, so there must be some $N_0 \in \mathbb{N}$ such that $n \ln n \geq 2n - 2$, for $n \geq N_0$. The formal arguments are not shown here.

To sum up, take $c_1 = 0.5, c_2 = 1, N = N_0$, and with (2), (3) we have (1), which completes the proof.

Remark 1. Another simpler way to prove $\log_2 n! = \Omega(n \log n)$ (the first inequality) is by

$$\begin{aligned}
\log n! &= \log n + \log(n-1) + \cdots + \log \frac{n}{2} + \cdots + \log 1 \\
&\geq \log n + \log(n-1) + \cdots + \log \frac{n}{2} \\
&\geq \log \frac{n}{2} + \log \frac{n}{2} + \cdots + \log \frac{n}{2} \\
&= \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n).
\end{aligned}$$

Remark 2. This problem can be solved easily with Stirling's formula, which this margin is too narrow to contain.

Problem 2. (a) Code provided as follow:

```
int is_linked_list_with_loop( struct list_node *head ){
  list_node* one_step = head;
  list_node* two_steps = head;
  while( one_step != NULL && two_steps != NULL ){
    one_step = one_step->next;
    two_steps = two_steps->next;
    if( two_steps == NULL ) return 0;
    two_steps = two_steps->next;
    if( one_step != NULL && one_step == two_steps )
      return 1;
  }
  return 0;
}
```

(b)
- Suppose the linked list is without loop. Then, for each `one_step` and `two_steps` will at most go through $O(N)$ nodes then encounter NULL pointer.
- Suppose the linked list has a loop. Assume that the size of loop is C and total number of nodes is N. Then, for each iteration, the number of steps between `one_step` and `two_steps` will differ by one. Therefore, after K iterations, their difference will be exactly K. Then, after $N - C$ iterations, both of them will go into the loop. Once their steps' difference is a multiple of C, they will be on the same node. We need at most C more iterations to reach a multiple of C. Therefore, total time complexity is $O(N-C)+O(C) = O(N)$.

(c) We modify the code of (a) to provide more information to solve this problem. That is, the number of step go through by `one_step`. For convenience, we encode the step by multiplying it by two. Then, we can still have the information that the linked list with loop or not from least significant bit.

```
int is_linked_list_with_loop( struct list_node *head ){
  list_node* one_step = head;
  list_node* two_steps = head;
  int step = 0;
  while( one_step != NULL && two_steps != NULL ){
    step = step + 1;
    one_step = one_step->next;
    two_steps = two_steps->next;
    if( two_steps == NULL ){
      while( one_step != NULL ){
        step = step + 1;
        one_step = one_step->next;
      }
      return 0 + step * 2;
    }
    two_steps = two_steps->next;
    if( one_step != NULL && one_step == two_steps )
      return 1 + step * 2;
  }
  return 0 + step * 2;
}
int chain_length( struct list_node *head ){
  int result = is_linked_list_with_loop( head );
  if( ( result & 1 ) == 0 )
    return result / 2;
  int step_of_one = result / 2;
  list_node* one_step = head;
  list_node* two_step = head;
  for( int i = 0 ; i < step_of_one ; i ++ )
    two_step = two_step->next;
  for( int i = 0 ; i < step_of_one ; i ++ ){
    if( one_step == two_step ){
      return i;
    }
    one_step = one_step->next;
    two_step = two_step->next;
  }
  return -1; // this should not happen
}
```

Problem 3. Space-efficient doubly linked list.

(a) (6%)

```
first_node->xor_pointer = (int)NULL ^ (int)(&next_node)
middle_node->xor_pointer = (int)(&prev_node) ^ (int)(&next_node)
last_node->xor_pointer = (int)(&prev_node) ^ (int)NULL
```

Note that XORing `(int)NULL` can be removed, since a bit remains the same after being XORed with 0.

Each node's `xor_pointer` pointer stores the XOR value of its previous node's address and its next node's address. During traversal, whatever the direction is, calculating the XOR value of the `xor_pointer` and its previous encountered node's address results in the next node's address.

(b) (9%)

```
int number_of_nodes(struct db_list_node *head) {
  struct db_list_node *prev, *next;
  prev = NULL;
  int count = 0;
  while(head != NULL) {
    next = (struct db_list_node *)((int)head->xor_pointer ^ (int
       )prev);
    prev = head;
    head = next;
    count ++;
  }
  return count;
}
```

Note that the `int` type should be replaced by `long long int` type, if the computer architecture is 64-bit instead of 32-bit, or you can use `uintptr_t` type, which automatically detects the length of address, to avoid this problem.

Problem 4.

(a) (5%)

Because insertion sort runs in $O(n^2)$ worst-case time for sorting n elements, sort a sequence of k elements runs in $O(k^2)$ worst-case time. For sorting $\frac{n}{k}$ sublists, the total running time is $O(\frac{n}{k} * k^2)$ $= O(nk)$

(b) (10%)

By problem 4.(a), we know that the running time of insertion sort part is $O(nk)$ (the bottom of the tree). The running time for sorting n elements from each level to higher level in merge sort is $O(n)$ (because in each level there are n elements). And we have $\frac{n}{k}$ sublists, so the number of levels of the tree are $\log_2 \frac{n}{k}$. The running time of merge sort part is $O(n * \log \frac{n}{k})$, and the time complexity of the hybrid algorithm is $O(nk + n \log \frac{n}{k})$.

(c) (10%)

Our target is to find the optimal k so that intersion sort can run faster than merge sort. At fisrt, we can find that sorting k elements by insertion sort is faster than sorting by merge sort. When the value of k grows and reaches some value, the running time of insertion sort is larger than the running time of merge sort.

In practice, we can choose any platform and compare the running time of merge sort and insertion sort by trying various values of k (e.g. k from 2 to n). The optimal value of k is the largest list length on which insertion sort is faster than merge sort.

Problem 5.

(a) (5%)

(1), (2), (4): valid; (3), (5): invalid.

For each list, split it in two lists: one with numbers smaller than 363 and one with bigger (in order). The smaller list should be sorted ascending, the bigger should be descending. For example, in case (1), the bigger list is 401, 398, and 397. It is sorted descending. The smaller list is 2, 252, 330, and 344. It is sorted ascending. Therefore, case (1) is valid. You can use the same method for case (2) and (4) and found that they are all valid, too.

In case (3), the bigger list is 925, 911, and 912. It is not sorted descending, so it is invalid. In case (5), the smaller list is 278, 347, 299, and 358. It is not sorted ascending, so it is invalid.

(b) (5%)

Given that x has two child nodes, x's successor must be in x's right subtree because

The x's successor must in the x's right subtree. Since x have right child, there exist one node greater than x. For the other nodes which is not in the x's right subtree, if it is greater than x, then it must greater than any node in subtree of x, so there is no any node can between x and the smallest node in the x'right subtree, hence that the smallest node in x'right subtree is x's successor. If x's succesor has a left child, the left child is greater than x since it is in x's right subtree and smaller then x's successor, we can find a contradiction that there is a node between x and x's successor, so x's successor have no left child. Similar, the smallest node in x's left subtree is x's predecessor and if it have right child will lead to the same contradiction.

(c) (10%)

```
struct bst_node *bst_successor(struct bst_node* x) {
    bst_node* successor = NULL;w
    if (x->right != NULL) {
        successor = x->right;
        while(successor->left != NULL){
            successor = successor->left;
        }
    } else {
        while (x->p != NULL){
            if (x->p->data > x->data){
                successor = x->p;
                break;
            }
            x = x->p;
        }
    }
    return successor;
}
```

Problem 6. String compression.

First of all, we can see that string AB (directly concatenate string A and string B) is a valid choice of S. Knowing the answer is no more then $|A| + |B|$ ($|S|$ denotes the length of string S), let's focus on finding shorter S.

If $|S| = x < |A| + |B|$,

A[1]	...	A[$x - |B| + 1$]	......	A[$|A|$]		
S[1]	...	S[$x - |B| + 1$]	......	S[$|A|$]	...	S[x]
		B[1]	......	B[$|A| + |B| - x$]	...	B[$|B|$]

There should be a string A's suffix equals to a string B's prefix.

If the string A's suffix with length y is equal to the string B's prefix with length y, then one can construct a string with length $|A| + |B| - y$ that is a valid choice of S.

The task becomes to find the longest A's suffix that is also a string B's prefix.

One can get 30% of scores by enumerating every A's suffix and directly comparing it with B's prefix that has the same length.

To get 100% of scores, one possible way is using *Rabin-Karp Algorithm* to compare a pair of A's suffix and B's prefix, which get time complexity to $O(|A| + |B|)$.

Another solution uses the property of *Prefix function* from *Knuth-Morris-Pratt Algorithm.*

For any string C, the prefix function $\pi[|C|]$ is the largest number $k < |C|$ such that the suffix of C with length k is a prefix of C. This seems similar to our task here. But we got two strings here, how to manage this problem?

Concatenate them!

Let's make $C = B\$A$ where \$ is a character not being any of the lowercase English letters. Because of the strange character \$, $\pi[|C|]$ will not exceed $min\{|A|, |B|\}$, which is actually matching A's suffix to B's prefix. So we got another solution with time complexity $O(|A| + |B|)$.

***Problem* 7.** Post-order Sequence Verification.

Overview

Basically, any sequence of numbers could be a post-order sequence of a binary tree. However, there are some characteristics that one must possess to be a post-order sequence of a *binary search tree.*

The definition of post-order is outputting the content of a binary tree in the following order: (1) output the left sub-tree, (2) output the right sub-tree, and (3) output the root. Now, consider the characteristics of a binary search tree. The left sub-tree contains all nodes that are smaller than root's value, and the right sub-tree contains all those larger than root's value.

Therefore, a post-order sequence, when we take the last number as the value of root, the rest of the sequence can be divided into two parts, where the first part is all the numbers smaller than root's value, *i.e.*, left sub-tree, and the numbers in the second part would be all greater than root's value, forming the right sub-tree.

Algorithm

Based on the observation and deduction above, we can now develop a more structured set of steps for our post-order sequence verification:

- Take the last number as root's value.
- Try split the rest of the sequence into two parts.
  - If successful, recursively check the two sub-sequence.
  - Otherwise, it's not a valid post-order sequence.

Sample Code

The following code snippet implements the algorithm above. The function `check()` would check a sub-sequence of the given integer array, `sequence`, specified by `start` and `end`, both inclusive.

The return value of `check()` is the height of the tree formed by the specified sub-sequence if it is possible, -1 otherwise. Additionally, the height of the tree formed by current sub-sequence is: 1 + the height of its higher sub-tree, either left or right.

```
function check(sequence, start, end):
    # the terminal condition
    if start > end:
        return 0
    # get the value of root
    root_value = sequence[end]
    # find the position of separation
    for i = start to end:
        if sequence[i] > root_value:
            break
    separate_index = i
    for i = separate_index to end:
        if seq[i] < root_value:
            return -1 # not separable
    # recursive checking
    left_sub = check(sequence, start, separate_index - 1)
    right_sub = check(sequence, separate_index, end - 1)
    if left_sub == -1 or right_sub == -1:
        return -1
    else:
        return max(left_sub, right_sub) + 1
```