DATA STRUCTURES AND ALGORITHMS HOMEWORK #2

B07902143 陳正康

2.1

(1) Pseudo code:

```
count[1:n] := 0
for i = 0 to len(A)-1
    if count[A[i]] > 0
        return A[i]
    count[A[i]] += 1
return NOT_FOUND
```

Time complexity: $\Theta(n)$

(2) Let A be an array with $\frac{(N+1)(N+2)}{2}$ elements, and define:

$$A[\frac{n(n+1)}{2} + k] := \binom{n}{k}, \quad 0 \le n \le N, \quad 0 \le k \le n$$

The index of A is as below (for N=5):

n\k	**0**	**1**	**2**	**3**	**4**	**5**
0	0					
1	1	2				
2	3	4	5			
3	6	7	8	9		
4	10	11	12	13	14	
5	15	16	17	18	19	20

(3) Let A be an array with $\lfloor \frac{N}{2} \rfloor (N - \lfloor \frac{N}{2} \rfloor) + N + 1$ elements, and define:

$$A[(h+1)(n-h)+k] := \binom{n}{k} = \binom{n}{n-k}, \quad 0 \le n \le N, \quad 0 \le k \le h, \quad h = \lfloor \frac{n}{2} \rfloor$$

The index of A is as below (for N=5):

n\k	0	1	2	3	4	5
0	0					
1	1					
2	2	3				
3	4	5				
4	6	7	8			
5	9	10	11			

(4) Consider simply linked-list for simplicity

Pseudo code:

```
slow := L.head
fast := L.head
while fast != L.head
    slow := slow.next
    fast := fast.next.next
fast.next := slow.next
slow.next := L.head.next
L'.head := slow
return (L, L')
```

(5) Pseudo code:

```
function Arrange(a)
    n := len(a)
    if n <= 1
        return a
    else if a[0] is odd
        return Arrange(a[1:n-1]) + a[0]
    else
        return a[0] + Arrange(a[1:n-1])
```

(6) Pseudo code:

```
function Arrange(a)
    n := len(a)
    i, j := 0, 0
    while i < n and j < n
        if i > j: swap(i, j)
        if a[i] is odd and a[j] is even
            swap(a[i], a[j])
        if a[i] is even: i += 1
        if a[j] is odd: j += 1
    return a
```

2.2

(1) Counterexample: $d(n) = 3n^2, e(n) = 2n^2, f(n) = n^2 + 1, g(n) = n^2$
Then $d(n) - e(n) = n^2 \neq O(f(n) - g(n)) = O(1)$

(2) If $h(n) = O(f(n) + g(n))$

$\exists c, n_0 > 0 \quad \forall n \geq n_0 \quad |h(n)| \leq c\,|f(n) + g(n)|$

$\implies |h(n)| \leq 2c\,|\max\{f(n), g(n)\}|$

$\implies h(n) = O(\max\{f(n), g(n)\})$

If $h(n) = O(\max\{f(n), g(n)\})$

$\exists c, n_0 > 0 \quad \forall n \geq n_0 \quad |h(n)| \leq c\,|\max\{f(n), g(n)\}|$

$\implies |h(n)| \leq c\,|f(n) + g(n)|$ (this is true for positive function f, g)

$\implies h(n) = O(f(n) + g(n))$

(3) $f(n) = n^2 \sin(\frac{\pi}{2}n)$

Suppose $f(n) = \Omega(n)$

$\exists c, n_0 \quad n \geq n_0 \rightarrow n^2 \sin(\frac{\pi}{2}n) \geq cn$

Consider $n = 4m + 3,\ m \in \mathbb{Z}^+ \implies f(n) = -(4m+3)^2$ cannot be lower-bounded by cn when $m \rightarrow \infty$

Contradiction.

Similarly, suppose $f(n) = O(n)$

$\exists c, n_0 \quad n \geq n_0 \rightarrow n^2 \sin(\frac{\pi}{2}n) \leq cn$

Consider $n = 4m + 1,\ m \in \mathbb{Z}^+ \implies f(n) = (4m+1)^2$ cannot be upper-bounded by cn when $m \rightarrow \infty$

Contradiction.

$$\sum_{i=1}^{n} \lceil \log_2 i \rceil \leq \sum_{i=1}^{n} \log_2 n + 1 = n \log_2 n + 1 = O(n \log n) \tag{4}$$

2.3

(1) The implementation of our data structure can be characterized in a few aspects:

A) **Cache-Friendly Memory Scheme**: separate columns into values and keys
Among the columns we need, we classify them into:
value: [sale, product_price, product_age_group, product_gender]
key: [user_id, product_id, click_timestamp]

Keys will later be sorted, but values won't move, so they are stored in different arrays.
We have defined the following 4 arrays:

```
struct EntryValue {
    bool sale;
    char product_price[8];
    char product_age_group[32];
    char product_gender[32];
};
struct EntryKeyUPT {
    uint8_t user_id[16];
    uint8_t product_id[16];
    int click_time;
    EntryValue* value;
};
```

```
struct EntryKeyPU {
    uint8_t product_id[16];
    uint8_t user_id[16];
    EntryValue* value;
};
struct EntryKeyUT {
    uint8_t user_id[16];
    int click_time;
    EntryValue* value;
};
```

One for storing values, 3 for storing keys, and a key should point to the associated values.
During sorting and binary search, the key they depend on is stored in the same struct, so it's cache-friendly.

B) **Data Compression**: from 32 to 16 bytes
As we observed that product_id and user_id are hexadecimal strings, every two characters can be converted into one byte. Modern CPUs can copy and compare a 16-byte string very efficiently with xmm register.
This make it faster on swap-based sorting algorithms and binary search.

C) **Pre-processing and Binary Search**: Sorting by different keys
We do 3 kinds of pre-processing: sort the loaded data by (user_id, product_id, click_timestamp), (product_id, user_id) and (user_id, click_timestamp), let's call **sorted_upt**, **sorted_pu** and **sorted_ut**, respectively. Then the 4 actions can be done by binary search on one of the array.

get(): binary search user_id, product_id, and click_timestamp in order in **sorted_upt**

purchased(): can binary search the user_id, then scan through in **sorted_upt**

clicked(): binary search two intervals, one by product_id1 and the other by product_id2 in **sorted_pu**, then scan through to find intersections.

profit(): scan through users in **sorted_ut**, for each user, binary search the start time and scan though to accumulate clicks and sales.