Data Structures and Algorithms: Homework #2

Due on April 14, 2015 at 16:20

Instructors Hsuan-Tien Lin, Roger Jang

Tim Liou (b03902028)

2.1 More about c++

(1) The **sub1** may result in a run-time error. Why?

This function return a reference to a local varible, whose lifetime limited to the scope of the function call. Once **sub1** return, **int c** is dead. Reference to a dead object is useless. Therefore, it would result in a run-time error.

(2) The **sub2** does not result in a run-time error, but there may be some other problem. What is the problem?

int *pc is allocated on the heap memory not on the stack memory, that is, its lifetime doesn't limited to the function call. Therefore, this function works. However, clients who call this function need to free memory by themselves. That is annoying since clients do not allocte anything but have to free something out. If clients forget to free memory, terrible memory leak occurs.

2.2 Arrays, Linked List, and Recursion

(1) Do Exercise C-3.4 of the textbook. (The faster the better!)

Sort the array A first, and go through the whole array to find which values are as same as their neighbors.

Quick-Sort(A)
$i \leftarrow 1$
$old \leftarrow A[0]$
while $i \neq n$ **do**
    if old is equal to $A[i]$ **then**
        old is one of the *repeated integers*
    end if
    $old \leftarrow A[i]$
    $i \leftarrow i + 1$
end while

Algorithm 1: Find the repeated integers

The time complexity of this **while loop** is $O(n)$ and the time complexity of **Quick-Sort** is $O(n \log n)$. Therefore, the time complexity of this algorithm is $O(n \log n)$.

Problem 2.2 continued on next page...

(2) Describe the memory layout and the function for getting/putting values from/to the matrix.

$$
A_n = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ \vdots \\ n-2 \\ n-1 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 0 & 1 & 2 & \cdots & n-2 & n-1 \end{array} \\
\begin{pmatrix}
a_{0,0} & 0 & 0 & \cdots & 0 & 0 \\
a_{1,0} & a_{1,1} & 0 & \cdots & 0 & 0 \\
a_{2,0} & a_{2,1} & a_{2,2} & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
a_{n-2,0} & a_{n-2,1} & a_{n-2,2} & \cdots & a_{n-2,n-2} & 0 \\
a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-2} & a_{n-1,n-1}
\end{pmatrix}
\end{array}
$$

We store $a_{i,j}$ with a dense one-dimensional array B at the position of $B[(\sum_{k=0}^{i} k) + j] = B[\frac{i(i+1)}{2} + j]$.

$$
\begin{array}{cccccccc}
B[\,] & 0 & 1 & 2 & 3 & 4 & 5 & \cdots \\
A & \big(a_{0,0} & a_{1,0} & a_{1,1} & a_{2,0} & a_{2,1} & a_{2,2} & \cdots \big)
\end{array}
$$

Functions below show how to input or output the value in this data structure.

```
function GET(B, row, col)
    return B[row(row+1)/2 + col]
end function
```

Algorithm 2: Get the value from this matrix

```
function PUT(B, row, col, value)
    B[row(row+1)/2 + col] ← value
    return
end function
```

Algorithm 3: Put the value in this matrix

(3) Do Exercise C-3.22 of the textbook.

Check the sizes of this two circularly linked list. If they are not in the same size, they obviously do not contain the same list of nodes. If they are in the same size, find the same node first, then check for the rest of the nodes.

```
function Check-Two-Circularly-Linked-Lists(L, M)
    if Lsize is equal to Msize then
        i ← 0
        while i ≠ Lsize do
            if LCursorElem is equal to MCursorElem then
                if the rest of nodes are the same then
                    if n ≠ 0 then
                        return true
                    end if
                end if
                LCursor ← LCursorNext
            end if
            i ← i + 1
        end while
    end if
    return false
end function
```

Algorithm 4: Check whether the two circularly linked lists are same or not

The time complexity of this algorithm is $O(n^2)$ since it also need a loop to check whether the rest of nodes are the same. Note that **if** $n \neq 0$ is to prevent the situation that they contain the same starting points.

(4) Do Exercise C-3.18 of the textbook using either C/C++ or pseudo code.

Listing 1: Rearrange integer array by recursion.

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

void rearrage(int *array, int start, int end)
{
    while (array[start] % 2 == 0) { start++; }
    while (array[end] % 2 != 0) { end--; }

    if (start >= end) return;

    swap(&array[start], &array[end]);
    rearrage(array, start, end);

    return;
}
```

(5) Do Exercise C-3.18 of the textbook, but use one single loop instead of recursion.

Listing 2: Rearrange integer array by one single loop

```
void rearrage(int *array, int start, int end)
{
    while (start < end)
    {
        if (array[start] % 2 != 0)
        {
            if (array[end] % 2 == 0)
            {
                swap(&array[start], &array[end]);
                start++;
                end--;
            }
            else { end--; }
        }
        else { start++; }
    }
    return;
```

```
    }
```

2.3 Asymptotic Complexity

(1) Do Exercise R-4.24 of the textbook, under the assumption that both $d(n) - e(n) \geq 0$ and $f(n) - g(n) \geq 0$.

Consider $d(n) = 2n^2$ and $e(n) = n^2$. Note that both $d(n)$ and $e(n)$ are $O(n^2)$, $d(n) - e(n) \geq 0$, and $f(n) - g(n) = n^2 - n^2 = 0 \geq 0$

$$\begin{aligned} d(n) - e(n) &= n^2 \\ f(n) - g(n) &= 0 \end{aligned}$$

We can easily find that $d(n) - e(n)$ is $O(n^2)$ not $O(f(n) - g(n))$.

Consider another case, $d(n) = 2n^3$ and $e(n) = n^2$. Note that $d(n)$ is $O(n^3)$, $e(n)$ is $O(n^2)$, $d(n) - e(n) \geq 0$, and $f(n) - g(n) \geq 0$

$$\begin{aligned} d(n) - e(n) &= 2n^3 - n^2 \\ f(n) - g(n) &= n^3 - n^2 \end{aligned}$$

We can easily find that $d(n) - e(n)$ is $O(n^3)$, which is also $O(f(n) - g(n))$.

Thus, We can conclude that $d(n) - e(n)$ is not necessarily $O(f(n) - g(n))$.

(2) Do Exercise R-4.29 of the textbook.

By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $(n+1)^5 \leq cn^5$ for every integer $n \geq n_0$. Take $n_0 = 1$, for every $n \geq n_0$, we have

$$\begin{aligned} (n+1)^5 = & \\ &= n^5 + 5n^4 + 10n^3 + 10n^2 + 5n^1 + 1 \\ &\leq 10n^5 + 10n^5 + 10n^5 + 10n^5 + 10n^5 \\ &= 50n^5 \end{aligned}$$

A possible choice is $c = 50$ and $n_0 = 1$. Thus, we conclude that $(n+1)^5$ is $O(n^5)$.

(3) Do Exercise R-4.39 of the textbook.

Consider $f(n) = n \log n + 97.5n$ and $g(n) = n^2$. Note that $f(n)$ is $O(n \log n)$, and $g(n)$ is $O(n^2)$. For every integer n, $1 \leq n \leq 97$ we can easily find that

$$
\begin{aligned}
f(n) = & \\
&= n \log n + 97.5n \\
&= n \cdot (\log n + 97.5) \\
&> n \cdot 97.5 \\
&> n^2 \\
&= g(n)
\end{aligned}
$$

Problem 2.3 continued on next page. . .

When $n = 98$,

$$\begin{aligned} f(98) &= 98 \cdot (\log 98 + 97.5) \\ &> 98 \cdot (0.5 + 97.5) \\ &= 98^2 \\ &= g(98) \end{aligned}$$

When $n = 99$,

$$\begin{aligned} f(99) &= 99 \cdot (\log 99 + 97.5) \\ &> 99 \cdot (1.5 + 97.5) \\ &= 99^2 \\ &= g(99) \end{aligned}$$

When $n \geq 100$,

$$\begin{aligned} f(n) &= \\ &= n \log n + 97.5n \\ &= n \cdot (\log n + 97.5) \\ &< n^2 \\ &= g(n) \end{aligned}$$

We found the possible functions satisfying the statements from question. There is $97.5n$ in $f(n)$, which isn't negligible when $n < 100$. When n is getting larger, this term would become negligible. Therefore, A1 and Bob find the result that $n < 100$, the $O(n^2)$-time algorithm runs faster, and when $n \geq 100$ is the $O(n \log n)$-time one better.

2.4 Playing with Big Data

(1) Describe your design of the data structure. Emphasize on why you think the data structure would be (time-wise) efficient for the four desired actions.

I use Data Structure 2 from HW2 Hints. I create two 2-D Array.
First Array for **Get**, **Clicked**, **impressed**,

UserID	vector<data>
0	data with u = 0...
1	data with u = 1...
⋮	⋮
23907634	data with u = 23907634...

Second Array for **Profit**,

AdID	vector<data> (Click, Impression, UserID)
0	data with a = 0...
1	data with a = 1...
⋮	⋮
22238287	data with a = 22238287...