ICCS200: Assignment homework-number-2

Poonnarat Nakartit poonpptn@gmail.com
Recitation: Your recitation section
The date

Exercise 1: : Poisoned Wine (2 points)

You own n bottles of wine, exactly one of which has been poisoned. You dont know which bottle, however. What you know is, if someone drinks just a tiny amount of wine from the poisoned bottle, s/he will start laughing uncontrollably after 30 days. In fact, the poison is so potent that even the faintest drop, diluted over and over, will still cause the symptom. Design a scheme that determines exactly which one bottle was poisoned. You are allowed 31 days and can expend O(logn) testers (people). Explain why your scheme meets the O(logn)-tester requirement.

- Since I can expend O(logn), I can assume that the $\log n$ is $\log_2 n$, and the number of tester is equal to t, meanging that

  $$\begin{aligned} t &= \log_2 n \\ 2^t &= n \end{aligned}$$

  I am going to use just t testers, so this garuntees that the number of my testers is always meets $O(logn)$ requirment.

- Here is how it works, the 2^t can be representing in so many forms, but the way that I am going use is representing it in the binary form, where t is the number of bits I need to labeling each bottle. Note that if the value of $log_2 n$ is not an integer, I will round it to an integer.

  For example, If my $n = 4$, the number of testers I need is $\log_2 4$, so t is equal to 2. Each person represent the number of binary bits I need. In this case, I need 2 bits. 1 means that person drink, and 0 means the opposit.

Bottle number	My scheme(Bianry form)
1	00
2	01
3	10
4	11

  After 30 days pass you can detect that which bottle is posioned by looking at the pattern of the person who is laughing uncontrollably. In this case, if the first bottle is posioned , these two guys will be fine nothing happen. If only the first guy (from right hand side) laughs, the second bottle is posioned. If only the other guy laughs, the third bottle is posioned. Therefore if they both meaing that the fourth bottle is posioned.

Exercise 2: : More Running Time Analysis (6 points)

For the most part, we have focused almost exclusively on worst-case running time. In this problem, we are going to pay closer attention to these Java methods and consider their worst-case and best-case behaviors. The best-case behavior is the running time on the input that yields the fastest running time. The worst-case behavior is the running time on the input that yields the slowest running time.

(1) Determine the $best - case$ running time and the $worst - case$ running time of `method1` in terms of $\Theta()$.

```
static void method1(int[] array) {
       int n = array.length; \\ O(1)

       for (int index=0;index<n-1;index++) {\\O(n-1) just for the loop, so O(n^2) in total
            int marker = helperMethod1(array, index, n - 1);
            swap(array, marker, index); \\O(1)
       }
}

static void swap(int[] array, int i, int j) { \\ This function takes O(1)
       int temp=array[i]; ---> \\O(1)
       array[i]=array[j];  ---> \\O(1)
       array[j]=temp; ----> \\O(1)
       }

static int helperMethod1(int[] array, int first, int last) {
       int max = array[first]; \\ O(1)
       int indexOfMax = first; \\O(1)

       for (int i=last;i>first;i--) { \\ O(first - last)
              if (array[i] > max) {
                     max = array[i]; \\O(1)
                     indexOfMax = i; \\O(1)
              }
       }
       return indexOfMax;
}
```

The `method1`runnig time depends on the length of the input arrays. Therefore, the $worse-case$ would be the case where the input array is very long. The running time of the $worse - case$ is $\Theta(n^2)$ as well as the $best - case$. I could answer that is no different in $worse - case$ and $best - case$

(2) Determine the *best − case* running time and the *worst − case* running time of `method2` interms of Θ.

```
static boolean method2(int[] array, int key) {
    int n = array.length; \\ O(1)
    for (int index = 0; index < n; index++) { \\O(n)
            if (array[index] == key)
                    return true;
            }
      return false;
}
```

The *worse−case* of this `method2` is where the `key` is at the last index of the `array`. The running time in this case is $\Theta(n)$ where n is the length the `array`. On the other hand, *best − case* is the case where the `key` is at the first index of the array. The running time will be $\Theta(1)$

(3) Determine the *best − case* running time and the *worst − case* running time of `method3` in terms of Θ.

```
static double method3(int[] array) {
      int n = array.length; \\O(1)
      double sum = 0; \\ O(1)
      for (int pass=100; pass >= 4; pass--) { O(1) this run as constance time
             for (int index=0;index < 2*n;index++) { \\ This is O(n)
                    for (int count=4*n;count>0;count/=2) \\This is O(logn)
                           sum += 1.0*array[index/2]/count;
             }
       }
      return sum;
}
```

The *worse − case* of this `method1` is when the size of `array` is very long, the running time is $\Theta(n \log n)$, I think in this case there are no differnt in*worse − case* and *best − case*

Exercise 3: : Recursive Code (6 points)

---

(1) Describe how you will measure the problem size in terms the input parameters. For example, the input size is measured by the variable n, or the input size is measured by the length of array a.
(2) Write a recurrence relation representing its running time. Show how you obtain the recurrence.
(3) Indicate what your recurrence solves to (by looking up the recurrence in our table).

Poonnarat Nakartit
poonpptn@gmail.com

```
//Code 1
// assume xs.length is a power of 2

int halvingSum(int[] xs) {
       if (xs.length == 1)
             return xs[0];
       else {
              int[] ys = new int[xs.length/2];
              for (int i=0;i<ys.length;i++)
                     ys[i] = xs[2*i]+xs[2*i+1];
              return halvingSum(ys);
        }
}
```

(1) The input size is measured by the length of the `xs`.
(2) I am going to define a recursive function that help me analyse this function.

Define:

$$\begin{aligned} T(w) &= \text{how long dose it take to run } \texttt{halvingSum(int[]}w\texttt{)} \\ T(1) &= \text{The first element in the } \texttt{int[]} \text{ .This step takes } O(1) \end{aligned}$$

For $w > 1$:

- make a new `int[] ys` with the length of `xs.length/2`. This takes $O(n)$
- perform for-loop with `length.ys` times. This takes $n/2$ times, which is still $O(n)$

(3) $T(w) = T(w/2) + O(n)$. Therefore, this code is $O(n)$.

```
//Code 2
int anotherSum(int[] xs) {
       if (xs.length == 1)
              return xs[0];
       else {
              int[] ys = Arrays.copyOfRange(xs, 1, xs.length);
              return xs[0]+anotherSum(ys);
              }
}
```

(1) The input size is measured by the length of the `xs`.
(2) I am going to define a recursive function that help me analyse this function.

Define:

$$\begin{aligned} T(w) &= \text{how long doset it take to run } \texttt{anotherSum(int[]}w\texttt{)} \\ T(1) &= \text{How long does it takes if the input size is one } \texttt{int[]} \text{ .This step takes } O(1). \end{aligned}$$

For $w > 1$:

- make a new `int[] ys` with `Arrays.copyOfRange(xs, 1, xs.length)`. This takes $O(t)$, where $t = (xs.length - 1)$, so this $O(n)$

(3)$T(w) = T(n-1) + O(n)$. Therefore, this code is $O(n^2)$

```
//Code 3
int[] prefixSum(int[] xs) {
       if (xs.length == 1)
              return xs;
       else {
              int n = xs.length;
              int[] left = Arrays.copyOfRange(xs, 0, n/2);
              left = prefixSum(left);
              int[] right = Arrays.copyOfRange(xs, n/2, n);
              right = prefixSum(right);

              int[] ps = new int[xs.length];
              int halfSum = left[left.length-1];
              for (int i=0;i<n/2;i++) {
                     ps[i] = left[i];
                     }
              for (int i=n/2;i<n;i++) {
                     ps[i] = right[i - n/2] + halfSum;
                     }
              return ps;
              }
}
```

(1) The input size is measured by the length of the `xs`.
(2) I am going to define a recursive function that help me analyse this function.

Define:

$T(w)$ = how long doset it take to run `anotherSum(int[]`w`)`
$T(1)$ = The first eliemetnt in the `int[]` .This step takes $O(1)$

For $w > 1$:

- create a new `int n` equal to `xs.length`. This step takes $O(n)$
- create a new`int[] left` by using `Arrays.copyOfRange(xs, 0, n/2)`. This takes $O(t)$, where $t = ((n/2) - 0)$, so this is $O(n)$.
- do the recursive call of the `int[] left`. This is $T(n/2)$
- create a new`int[] right` by using `Arrays.copyOfRange(xs, n/2, n)`. This takes $O(t)$, where $t = ((n) - n/2)$, so this is $O(n)$.
- do the recursive call of the `int[] right`. This is $T(n/2)$

Poonnarat Nakartit
poonpptn@gmail.com

- make a new `int ps` with the length of `xs.length`.
- do for-loop $n/2$ times, so this is $O(n)$.
- do for-loop $n/2$ times, so this is $O(n)$.

(3)$T(w) = T(n/2) + O(n) + T(n/2) + O(n)$. Therefore this code is $O(n \log n)$.