CS170–Fall 2014 — Solutions to Homework 2

Quoc Thai Nguyen Truong, SID 24547327, `cs170-ig`

September 12, 2014

Collaborators: Tho Truong

1. Big-Theta running time

(1 page)

(a) No

(b) Yes

(c) Yes

(d) No

2. Practice with running time analysis

(2 pages)

(a) We know that at step 2 takes i^2 steps of computation for each i^{th} iteration. So we have

$$0+1+4+3+9\cdots(n-1)^2 = 0^2+1^2+2^2+3^3+\cdots+(n-1)^2$$

$$= \sum_{i=0}^{n-1} i^2 = \frac{(n)(2n-1)(n-1)}{6}$$

Which equivalent to

$$\boxed{\Theta(n^3)}$$

(b) We know that at step 2 takes (n-2i) steps of computation in the i^{th} iteration excep when $2i \geqslant n$, so when $i \geqslant \frac{n}{2}$ step 2 will take 1 step of computation. Therefore, we have:

$$n+(n-2)+(n-4)+\cdots+(n-\frac{2n}{2})+C$$

where C is some constant of computations when $i \geqslant \frac{n}{2}$

$$\Longrightarrow \sum_{i=0}^{\frac{n}{2}}(n-2i)+C$$

$$= \sum_{i=0}^{\frac{n}{2}} n - 2\sum_{i=0}^{\frac{n}{2}} i + C$$

$$= \frac{n^2}{2} - (\frac{n}{2})(\frac{n}{2}+1)+C$$

Which equivalent to

$$\boxed{\Theta(n^2)}$$

(c) We know that step 2 takes $\frac{n}{(i+1)}$ step of computation in the i^{th} iteration. So we have

$$\frac{n}{2^0}+\frac{n}{2^1}+\frac{n}{x^2}+\cdots+\frac{n}{2^k}$$

where $k = log(n)-1$

$$\Longrightarrow n \sum_{i=0}^{log(n)-1} \frac{1}{2^i}$$

We know that

$$\sum_{i=0}^{log(n)-1} \frac{1}{2^i} \leqslant \sum_{i=0}^{\infty} \frac{1}{2^i} = 1$$

$$\implies n \sum_{i=0}^{log(n)-1} \frac{1}{2^i} = n \cdot 1 = n$$

Which is equivalent to

$$\boxed{\Theta(n)}$$

(d) We know at step 2 takes $\frac{n}{i+1}$ steps of computation for each iteration. So we have

$$\frac{n}{1} + \frac{n}{2} + \frac{n}{2} + \cdots + \frac{n}{n-1} + \frac{n}{n}$$

$$\implies n \sum_{i=0}^{n-1} \frac{1}{i+1}$$

We know that:

$$\sum_{i=0}^{n-1} \frac{1}{i+1} < \int_1^n \frac{1}{x} dx \text{ , } \textit{for all } n \geqslant 1, \textit{ harmonic series property}$$

$$\sum_{i=0}^{n-1} \frac{1}{i+1} < \ln(x)\Big|_1^n$$

$$\sum_{i=0}^{n-1} \frac{1}{i+1} < \ln(n) \ln(1)$$

$$\sum_{i=0}^{n-1} \frac{1}{i+1} < \ln(n)$$

Multiply both side by n

$$\implies n \sum_{i=0}^{n-1} \frac{1}{i+1} < n \ln(n)$$

$$\implies \textit{the run time is}: \ \Theta(n \log(n))$$

3. Out of sorts

(1 page)

(a) The base case when n = 2, it's doing the swap which take a constant time $\rightarrow \theta(1)$
There are 3 recursive calls at line 4, 5 and 6, each of them take a list of $\frac{2 \cdot n}{3}$ elements
Therefore, the recurrence relation for $T(n)$ is:

$$\boxed{T(n) = 3 \cdot T(\frac{2n}{3}) + \Theta(1)}$$

(b) Using the Master theorem, we can write T(n) as

$$T(n) = a \cdot T(\frac{n}{b}) + O(n^d)$$

Since $a = 3,\ b = \frac{3}{2},\ and\ d = 0$
We have $d = 0 < \log_b(a) = \log_{3/2}(3) \approx 2.71$
From the Master theorem

$$\Longrightarrow T(n) = \Theta(n^{\log_b(a)})$$

which is equivalent to

$$\boxed{\Theta(n^{2.71})}$$

(c) We know that Insertion sort takes $\Theta(n^2)$, and $T(n) = \Theta(n^{2.71})$
Therefore, I would expect Algorithm S to be $\boxed{\textit{slower than}}$ Insertion sort

4. Merge asymptotics

(2 pages)

We have:

$$n = number\ of\ exams$$

$$k = number\ of\ GSIs\ or\ number\ of piles$$

$$\frac{n}{k} = number\ of\ exams\ in\ a\ pile$$

Merging two plies take $\Theta(a+b)$ work

(a) Use the method 1, we know that David has to merge all k of piles. For each time he has to merge the (pile) result from previous step with the new pile. Assume we have Merge2 function which merging two plies, one with a exams and other with b exams takes $\Theta(a+b)$
So we have an pseudo code algorithm:

```
Method_1($A(k_0, k_1, k_2, \cdots, k_k)$):
    if(length(A) = 2) : return Merge2
    return Merge2(Method_1($A(k_0, k_1, k_2, \cdots k_{k-1})$), $k_k$)
```

Let T(k) = the number of works (steps) that merge k piles using method 1.
Assume that there is only 1 exam in each pile.

So it would take $\Theta(k)$ to merge the result of $(1st\ pile\ \rightarrow\ (k-1)^{th}\ pile)$ and the last pile $(k^{th} pile)$
Therefore, the recurrence relation is:

$$T(k) = T(k-1) + \Theta(k)$$

1st merge takes $1+1=2$ steps
2nd merge takes $2+1=3$ steps
3rd merge takes $3+1=4$ steps
$\cdots$
$(k-1)^{th}\ merge$ takes $k-1+1=k$ steps

$$\implies Total\ steps = \sum_{i=1}^{k-1}((i+1))$$
$$= (\sum_{i=1}^{k-1}(i+1)$$
$$= (\frac{k(k-1)}{2} - 1)$$
$$= (\frac{k^2+k-2}{2})$$

Which is equivalent to

$$\Theta(k^2)$$

Since there are $\frac{n}{k}$ exams in each pile.
So it would take $(\frac{n}{k})\Theta(k^2) = \Theta(nk)$
Therefore, it takes $\boxed{\Theta(nk)}$ for David to do if he uses method 1

(b) Use the method 2, split the piles into two roughly equal halves, recursively merge each half, then use the merge procedure to combine the two halves.

We have k piles, and assume that each pile have 1 exam.
Let T(k) = number of work (steps) that merge k piles using method 2.
Split the piles into two roughly equals halves, and recursively merge each half $\implies$ 2 calls recursive with size of $\frac{k}{2}$.
And it would takes $\Theta(k)$ to merge the two sorted piles with size of $\frac{n}{2}$ for each pile before return the list

$$\implies T(k) = 2T(\frac{k}{2}) + \Theta(k)$$

From Master Theorem:
Since a = b = 2, and $\log_b(a) = \log_2(2) = 1 = d = 1 \implies T(k) = k\log(k)$
Since there are $\frac{n}{k}$ exams for each pile, so:

$$T(n,k) = (\frac{n}{k})(k\log(k))$$
$$\boxed{T(n,k) = n\log(k)}$$

(c) Since method 1 take $\Theta(nk)$ and method 2 takes $\Theta(nlog(k))$, so $\boxed{method\ 2\ is\ better}$

5. Plurality finding: divide-and-conquer

(3 pages)
Explanation:
The base cases are: if n = 3 we want to return plurality element in A, if there is no plurality element, then return Empty Array.
If $n > 3$, split the array into half of n (set1, set2), recursively finding the plurality element for each set. Then, combine the all 2 sets into "sets", count the number of appearance of each element in "sets" with the array A. If there's at least an element in the sets that has the appearance more than $\frac{n}{3}$, output "yes" and return that element. Otherwise, output "No" and return empty array.
Algorithm:

```
Line 0: Plurality($A[0 \cdots n-1]$):
Line 1:     If $n == 3$:
Line 2:         If there is plurality element in A, return it
Line 3:         Else: output "No" return Empty Array
Line 4:     Else :
Line 5:         $k = \frac{n}{2}$
Line 6:         $set1 = $ Plurality($A[0 \cdots k-1]$)
Line 7:         $set2 = $ Plurality($A[k \cdots n-1]$)
Line 8:         $sets = set1 + set2$
Line 9:         For e in sets:
Line 10:            $count = 0$
Line 11:            For $i := 0, 1, \cdots, n-1$ :
Line 12:                If $e = A[i]$:
Line 13:                    $count = count + 1$
Line 14:            If $count > \frac{n}{3}$ :
Line 15:                print "Yes"
Line 16:                return e
Line 17:     print "No"
```

Running time:
Let T(n) = the run time of Plurality algorithm, and n is the size of the array.

In the algorithm, there are 2 recursive calls, and each call take in an array of length $\frac{1}{2}$ of the current array's length. Also, *sets* has at most 2 elements, and we compare the values in the *sets* with each element in array A which takes $\Theta(n)$

Therefore
$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$
from Master Theorem: $a = b = 2, d = 1, \ so \log_2(2) = d = 1$
Therefore:
$$\boxed{T(n) = \Theta(n\log(n))}$$

Proof of correctness:
_ Invariant:

If an array $A[0 \cdots n-1]$ has at least a $\frac{1}{3}$ plurality element v. Hence, at the start of recursive call at k level, v must be plurality element and v appears $> \frac{n}{3^k}$
(where $1 \leqslant k \leqslant \log_2(n)$) in at least 1 sub-array in 2^k sub-arrays (set1(s) and set2(s)). Therefore, at least 2^k must return the value v.

_ Base Case: At first we split the array into 2 halves arrays(k = 1 and $\frac{n}{3}$ for each), and if a value v is a plurality element in array $A[0 \cdots n-1]$, we know that v must be plurality element in these 2 sub-arrays, so one of these 2 halves arrays must return v. Therefore, the invariant holds true for the base case.
_ Inductive Case:
Assume that array $A[0 \cdots n-1]$ has at least a $\frac{1}{3}$ plurality element v. At k level recursive call, algorithm has $2^k sub-arrays$ with size of $\frac{n}{3^k}$, and there is at least 1 of these sub-array has the plurality value v.

We know that $k+1$ level recursive has 2^{k+1} sub-arrays. Since at k level recursive , there is at least 1 of 2^k sub-array with size $\frac{n}{3^k}$ has the plurality value v. So at $k+1$, the sub-array has the plurality value v split the array into halves, and we know that v must appear more than $\frac{n}{3^{k+1}}$ in at least one of these halves, so that sub-array must return value v. Hence, it't true for $k+1$.
Therefore it's true for all k levels, and $1 \leqslant k \leqslant \log_2(n)$. $\Rightarrow$ the invariant is

inductive.

_ Prove correctness property:
At the $\log(n)$ level where the algorithm hits the base case $n = 3$, all the recursive calls will pop-up to level 0, and return the solution either "Yes" with a plurality value or "No" . We know that if Array A with size of n has at least 1 plurality value v, then v be must be the return value of either $set1$ or $set2$.Then compare the value v with all n values in array A which also the plurality value, also output "Yes". Otherwise, if array A doesn't have any plurality, and $set1$ or $set2$ could return some value z, but compare z with all n elements in A which is less than or equal to $\frac{n}{3}$, so the algorithm will output "No". Therefore, the algorithm is true for finding the plurality element in an array

6. More divide-and-conquer

(3 pages)

Explanation:

First we sort the array by the start value of intervals.Split the list into two roughly equal halves(left and right), recursively finding the pairs that are overlap of each half.After we have the pairs of interval for each half (left and right), then we find the pairs between these halves (left and right). To find the pairs between these halves, we find an interval that has the highest value of end point interval in the left half, and compare that interval with all the intervals ($\frac{n}{2}$) in the right half. At the end, we have 3 pairs of interval (left, between left and right, and right), and we return the pair that has the maximum overlap.

Algorithm:

```
Line 1:  OverLap($[x_1, y_1], [x_2, y_2] \cdots, [x_n, y_n]$):
Line 2:      If n == 2:
Line 3:          return ($[x_1, y_1], [x_2, y_2]$)
Line 4:      Else If n == 3:
Line 5:          p1 = $min(y_1, y_2) - max(x_1, x_2)$
Line 6:          p2 = $min(y_1, y_3) - max(x_1, x_3)$
Line 7:          p3 = $min(y_2, y_3) - max(x_2, x_3)$
Line 8:              If $p1 = max(p1, p2, p3)$ : return ($[x_1, y_1], [x_2, y_2]$)
Line 9:              If $p2 = max(p1, p2, p3)$ : return ($[x_1, y_1], [x_3, y_3]$)
Line 10:              Else: return ($[x_2, y_2], [x_3, y_3]$)
Line 11:     Else: Sort(A)
Line 12:         left_pair = OverLap($[x_1, y_1][x_2, y_2] \cdots, [x_{\frac{n}{2}}, y_{\frac{n}{2}}]$)
Line 13:         right_pair = OverLap($[x_{(\frac{n}{2}+1)}, y_{(\frac{n}{2}+1)}][x_{(\frac{n}{2}+2)}, y_{(\frac{n}{2}+2)}] \cdots, [x_n, y_n]$)
Line 14:         $k = 1$
Line 15:         For $i := 2,\ 3, \cdots \frac{n}{2}$:
Line 16:             if($y_i > y_k$): $k = i$
Line 17:         $Z = (x_k, y_k)$
Line 18:         temp_interval = find the most overlap with Z from interval $\frac{n}{2}+1 \to n-1$
Line 19:         middle_pair = [Z,temp_interval]
Line 20:         left = number of element(s) overlap in left_pair
Line 21:         right = number of element(s) overlap in right_pair
```

Line 22:	mid = number of element(s) overlap in middle_pair
Line 23:	If left = max(left,mid,right): return left_pair
Line 24:	Else If right = max(left,mid,right): return right_pair
Line 25:	Else: return middle_pair

Running time: Let T(n) = the run time of "OverLap" algorithm, and n is the size of the array.
In the algorithm, we sort the list A which take $\Theta(n \log n)$. There are 2 recursive calls, and each call take in an array of length $\frac{1}{2}$ of the current array's length. Also, to find the middle pairs interval, we have to go though the list on the left side to find the interval has the highest end point, and use it to find the most overlap interval on the right side, this process takes $\Theta(n)$
Therefore,

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

from Master Theorem: $a = b = 2, d = 1,\ so \log_2(2) = d = 1$
Therefore:

$$\boxed{T(n) = \Theta(n \log(N))}$$

Proof of correctness:
_ Invariant:

At the end of each loop on k recursive call ($1 \leqslant k \leqslant \log(n)$), the function "OverLap" return a pair of interval which has most overlap elements by comparing the max overlap elements between pair intervals in $left\_pair, right\_pair$, and $middle\_pair$

_ Base case: Assume there is exist an over lap pair of interval which has most overlap in n intervals. The algorithm sorted the list and splits the list of interval into $\frac{n}{2}$, one half intervals on left side and other half on the right side. We also find the middle pair by finding the pair that has the highest end point value on the left side, and compare that interval with every interval on the right side. We can see that the most over lap pair of interval must be in either right, left side , or middle. So the Invariant is true for the base case.

_ Inductive Step: Assume that the Invariant is true at k level, we prove that it's also true for $k + 1$ level.
At k level, we know that the most over lap pair of interval must be in either

right side ($\frac{n}{2^k}$ intervals), left side($\frac{n}{2^k}$ intervals), or middle pair. If the most over lap pair of interval appear in the left side or right side at level k, Therefore, at $k+1$ level, it's must be in either left side ($\frac{n}{2^{k+1}}$ intervals), right side ($\frac{n}{2^{k+1}}$ intervals), or middle pair. Hence, the most over lap pair of interval must be the return value of one of these pair (left, right, or middle pair). Therefore, it's true for k + 1 , so it's true for all k ($1 \leqslant k \leqslant \log(n)$). $\Rightarrow$ The invariant is inductive.

_ Prove correctness property:
At the $\log(n)$ level where the algorithm hits the base case $n = 2$ or $n = 3$, all the recursive calls will pop-up to level 0, and return the pair of interval that has most overlap elements. We know that the most overlap pair of interval must be in either left pair, right pair, or middle pair. Since we compare the max overlap elements between these 3 pairs, so it must return the pair that has maximum overlap elements. Therefore, the algorithm is true for finding the pair of interval with highest overlap.