

CSOR W4231 Analysis of algorithms I
Assignment 1

Xijiao Li (xl2950)

September 29, 2020

Problem 1

a.

code:

```
s = 0;
for i=1 to n                \\ n
    for j=1 to i            \\ i
        if (A[i] = A[j]) then s = s+1 \\ 1
    return s
```

$$\sum_{i=1}^n \sum_{j=1}^i 1 = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

Hence the time complexity is $\Theta(n^2)$, $f(n) = n^2$.

b.

code:

```
s = 0;
for i=1 to n                \\ n
    for j=1 to i            \\ i
        for k=j to i        \\ i-j
            if (A[k] = A[i]+ A[j]) then s = s+1 \\ 1
    return s
```

We have $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i 1 = (1)+(1+2)+\dots+(1+2+\dots+n) = \frac{n(n+1)(n+2)}{6}$,
So the time complexity is $\Theta(n^3)$, $f(n) = n^3$.

Problem 2

Orders (smallest growth rate first):

- $\Theta(1)$: 10
- $\Theta(\log(\log n))$: $\log(\log n)$
- $\Theta(\log n)$: $\log_{10} n < \log(n^2)$
- $\Theta(\log^2 n)$: $(\log n)^2$
- $\Theta(n^{1/4})$: $\sqrt[4]{n}$
- $\Theta(n \log^2 n)$: $n(\log n)^2$

- $\Theta(n^2)$: $2^{2\log n} < (3n - 2)^2$
- $\Theta(n^4)$: n^4
- $\Theta(n^{\log n})$: $n^{\log n} = 2^{(\log n)^2}$
- $\Theta(4^{(n^{1/2})})$: $4^{\sqrt{n}}$
- $\Theta((\sqrt{2})^n)$: $\sqrt{2}^n$
- $\Theta(8^n)$: 2^{3n}
- $\Theta(9^n)$: 3^{2n}
- $\Theta(n!)$: $n!$
- $\Theta(n^n)$: n^n

Justification for same group:

- $\Theta(\log n)$: $\log_{10} n < \log(n^2)$

$$\lim_{n \rightarrow \infty} \frac{\log_{10} n}{\log(n^2)} = \lim_{n \rightarrow \infty} \frac{\frac{\log n}{\log 10}}{2 \log n} = \frac{1}{2 \log 10}, \text{ which is a constant.}$$

- $\Theta(n^2)$: $2^{2\log n} < (3n - 2)^2$

$$\begin{aligned} 2^{2\log n} &= (2^{\log n})^2 = n^2 \\ (3n - 2)^2 &= 9n^2 - 12n + 4 \\ \lim_{n \rightarrow \infty} \frac{2^{2\log n}}{(3n - 2)^2} &= \lim_{n \rightarrow \infty} \frac{n^2}{9n^2 - 12n + 4} = \frac{1}{9}, \text{ which is a constant.} \end{aligned}$$

- $\Theta(n^{\log n})$: $n^{\log n} = 2^{(\log n)^2}$

$$2^{(\log n)^2} = 2^{(\log n)(\log n)} = n^{\log n}, \text{ these two are equal.}$$

Problem 3

a.

$$a = 9, b = 3, f(n) = n^2$$

because $n^{\log_b a} = n^{\log_3 9} = n^2$, $f(n) = \Theta(n^2)$, case 2 of the Master Theorem applies and we have $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^2 \lg n)$

b.

$$a = 2, b = 3, f(n) = n$$

because $n^{\log_b a + \epsilon} = n^{\log_3 2 + 1} = n$, $f(n) = \Omega(n)$, case 3 of the Master Theorem applies and we have $T(n) = \Theta(f(n)) = \Theta(n)$

c.

$$a = 3, b = 2, f(n) = n \log^2 n$$

because $n^{\log_b a} = n^{\log_2 3} = n^{1.58}$, $f(n) = O(n^{1.58})$ since

$$\lim_{n \rightarrow \infty} \frac{n \log^2 n}{n^{1.58}} = \lim_{n \rightarrow \infty} \frac{\log^2 n}{n^{0.58}} = 0$$

case 1 of the Master Theorem applies and we have $T(n) = \Theta(n^{\log_b a}) = O(n^{\log_2 3})$

d.

Unfold the recurrences:

$$\begin{aligned} T(n) &= 4T(n-2) + 1 \\ &= 4(4T(n-4) + 1) + 1 = 16T(n-4) + 4 + 1 \\ &\dots \\ &= 4^h T(n-2h) + (1 + 4 + \dots + 4^{h-1}) \end{aligned}$$

where $h = n/2$, so $T(n) = 2^n + \frac{2^n - 1}{3} = O(2^n)$

e.

Let $m = \log n$, and $n = 2^m$. So we can change the equation to

$$\begin{aligned} T(2^m) &= T(2^{m/2}) + 1 \\ S(m) &= S(m/2) + 1 \end{aligned}$$

where we have $a = 1, b = 2, f(m) = 1$, so $m^{\log_b a} = m^0 = 1$. Since $f(m) = \Theta(1)$, case 2 of the Master Theorem applies and we have $S(m) = \Theta(\log m)$.

So $T(n) = T(2^m) = S(m) = \Theta(\log m) = \Theta(\log \log n)$.

f.

We have a lopsided recursion tree. Nodes in the same level of the recursion tree have different values, and different leaves are at different levels. However, the nodes in any complete level sum to n . Moreover, every leaf in the recursion tree has depth between $\log_2 n$ and $\log_3 n$.

To derive an upper bound, we overestimate $T(n)$ by ignoring the base cases and extending the tree downward to the level of the deepest leaf. Similarly, to derive a lower bound, we overestimate $T(n)$ by counting only nodes in the tree up to the level of the shallowest leaf. These observations give us the upper and lower bounds $n \log_2 n \leq T(n) \leq n \log_3 n$.
 So $T(n) = O(n \log n)$.

Problem 4

a.

```
TwoSum(A, B)
  initial set s1, s2                                \\ 1
  set len <- max(length(A), length(B))              \\ 1
  for i from 0 to len:                              \\ len
    if i < length(A) and (100-A[i]) in s2:
      return True                                   \\ 1
    else:
      put A[i] in s1                                \\ 1
    if i < length(B) and (100-B[i]) in s1:
      return True                                   \\ 1
    else:
      put B[i] in s2                                \\ 1
  return False
```

The run time is bound to the times of executing the for loop, which is $length(A) + length(B) = n$ in the worst case, so the algorithm runs in $O(n)$ time.

b.

```
MergeSort(A, left, right)
if right > left:
  middle = (left+right)/2
  mergeSort(A, left, middle)
  mergeSort(A, middle+1, right)
  merge(A, left, middle, right)

Merge(A, left, middle, right)
initial array temp with length (right-left+1)
set p1 <- left
set p2 <- middle+1
for i from 0 to length(temp):
  if p2 == right+1 or A[p1] < B[p2]:
```

```

        temp[i] = A[p1]
        set p1 <- p1+1
    else:
        temp[i] = B[p2]
        set p2 <- p2+1
return temp

TwoSum(C)
initial set s
MergeSort(C)                \\ nlog n
for c in C:                  \\ n
    if (100-c) in s:
        return True        \\ 1
    else:
        put c in s          \\ 1
return False

```

The run time of the sorting part is bound to $O(n \log n)$ (for merge sort), and the run time of the searching part is bound to the times of executing the for loop, which is $length(C) = n$, so the algorithm runs in $O(n \log n)$ time in the worst case.

Problem 5

1.

The run time of this algorithm is bound to the number of the executions of the while loop, which is a in the worst case, so the algorithm runs in $O(a) = O(2^n)$ time. Since the growth rate of 2^n is greater than n^c for any constant c because $\lim_{n \rightarrow \infty} \frac{n^c}{2^n} = 0$, this algorithm does not run in polynomial time.

2.

Note: In order to distinguish the values of x, y in different iterations of the while loop, I will write x_i and y_i as the values of x, y at the start of the i th iteration.

a.

i. $x \geq y$

- At the first iteration of the while loop, since $x = \max(a, b), y = \min(a, b)$, so $x \geq y$.

- At the other iterations of the while loop, we only need to prove that $r \leq y$ after $r := x \bmod y$, which is obvious since remainder is smaller than divisor.

ii. For all positive integers c , c divides both a and b if and only if c divides both x and y .

(\rightarrow) for all positive integers c , c divides both a and b if c divides both x and y .

Proof by induction.

- Base case:

At the first iteration of the while loop, since $x_1 = \max(a, b)$, $y_1 = \min(a, b)$, so c divides both a and b .

- Induction Hypothesis:

At the k th iterations of the while loop, if c divides both x_k and y_k , c divides both a and b .

- Inductive Step:

At the $k+1$ th iterations of the while loop, if c divides both x_{k+1} and y_{k+1} , c divides both a and b .

Write $x_{k+1} = pc$ and $y_{k+1} = qc$ for some constants p, q .

Since we have:

$$x_{k+1} = y_k$$

$$y_{k+1} = (x_k \bmod y_k) = x_k - sy_k, \text{ for some constant } s$$

$$\text{Write } y_k = pc \text{ and } x_k = pc + qc,$$

so c divides both x_k and y_k , and by IH, c divides both a and b .

(\leftarrow) for all positive integers c , c divides both x and y if c divides a and b .

Proof by induction.

- Base case:

At the first iteration of the while loop, since $x_1 = \max(a, b)$, $y_1 = \min(a, b)$, so if c divides both a and b , c divides both x_1 and y_1 .

- Induction Hypothesis:

At the k th iterations of the while loop, if c divides both a and b , c divides both x_k and y_k .

- Inductive Step:

At the $k + 1$ th iterations of the while loop, if c divides both a and b , c divides both x_{k+1} and y_{k+1} .

By IH, if c divides both a and b , c divides both x_k and y_k , write $x_k = pc$ and $y_k = qc$ for some constants p, q .

Since $x_{k+1} = y_k$ and $y_{k+1} = (x_k \bmod y_k) = x_k - sy_k$ for some constants s , $x_{k+1} = qc$, $y_{k+1} = pc - sqc$, and c divides both x_{k+1} and y_{k+1} .

Thus, we have already proved both directions, and for all positive integers c , c divides both a and b if and only if c divides both x and y .

b.

Assume that after k iterations, the while loop breaks and value v is returned, so $v = x_k \bmod y_k = x_k - sy_k$ for some constant s . From the condition of the while loop, we know that v divides $x_{k+1} = y_k$, so write $y_k = qc$ for some constants q . Thus, $x_k = v + sy_k = (sq + 1)v$, v divides x_k . Use the fact we have proved in 2.a, we know that v divides both a, b , so v is a common divisor of a, b .

Now we need to prove that v is the greatest common divisor of a, b . Proof by contradiction. Suppose there is a value $w > v$ that is also a common divisor of a, b . Use the fact we have proved in 2.a, we know that w divides both $x_{k+1}, y_{k+1} = v$, so w is less or equal to v , which is a contradiction. Thus v is the greatest common divisor of a, b .

c.

First, we want to show that $r = x \bmod y \leq x/2$. Proof by contradiction. Suppose that $r > x/2$. Since y divides $x - r$, $y < x - r < x - x/2 = x/2 < r$, which is a contradiction since remainder is smaller than divisor. So, our assumption fails and $r = x \bmod y \leq x/2$.

With this fact, we know that x, y decrease at least by a factor of 2 after every two iterations. So $f(n) = O(2 \log n) = O(n)$.