ICCS 313: Assignment 2

Tawan Chaeyklinthes u5980963

Date: 23 Sep 2018

Problem 1

(a) $T(n) = 2T(n/3) + 1$

Using Master Theorem, we have that $a = 2$, $b = 3$, $d = 0$. Thus, $\log_3 2 > 0$ and by the theorem, $T(n) = \mathcal{O}(n^{\log_3 2})$.

(b) $T(n) = 5T(n/4) + n$

Using Master Theorem, we have that $a = 5$, $b = 4$, $d = 1$. Thus, $\log_4 5 > 1$ and by the theorem, $T(n) = \mathcal{O}(n^{\log_4 5})$.

(c) $T(n) = 7T(n/7) + n$

Using Master Theorem, we have that $a = 7$, $b = 7$, $d = 1$. Thus, $\log_7 7 = 1$ and by the theorem, $T(n) = \mathcal{O}(n \log n)$.

(d) $T(n) = 9T(n/3) + n^2$

Using Master Theorem, we have that $a = 9$, $b = 3$, $d = 2$. Thus, $\log_3 9 = 2$ and by the theorem, $T(n) = \mathcal{O}(n^2 \log n)$.

(e) $T(n) = 8T(n/2) + n^3$

Using Master Theorem, we have that $a = 8$, $b = 2$, $d = 3$. Thus, $\log_3 8 = 3$ and by the theorem, $T(n) = \mathcal{O}(n^3 \log n)$.

(f) $T(n) = T(n-1) + 2$

$$
\begin{aligned}
T(n) &= T(n-1) + 2 \\
&= T(n-2) + 2 + 2 \\
&= T(n-3) + 2 + 2 + 2 \\
&= T(n-n-2) + 2(n-2) \\
&= T(2) + 2(n-2) && \text{; where } T(2) = k \text{ ,for some constant.} \\
&= \mathcal{O}(n)
\end{aligned}
$$

(g) $T(n) = T(n-1) + n^c$

$$
\begin{aligned}
T(n) &= T(n-1) + n^c \\
&= T(n-2) + (n-1)^c + n^c \\
&= T(n-3) + (n-2)^c + (n-1)^c + n^c \\
&= T(2) + n^c + (n-1)^c + ... + 4^c + 3^c && \text{; where } T(2) = k \text{ ,for some constant.} \\
&= k + \mathcal{O}(n^{c+1} - 2^c - 1^c) \\
&= \mathcal{O}(n^{c+1})
\end{aligned}
$$

(h) $T(n) = T(n-1) + c^n$

$$\begin{aligned}
T(n) &= T(n-1) + c^n \\
&= T(n-2) + c^{n-1} + c^n \\
&= T(n-3) + c^{n-2} + c^{n-1} + c^n \\
&= T(0) + c^1 + c^2 + c^3 + ... + c^n \qquad \text{; where } T(0) = k \text{ ,for some constant.} \\
&= k + \mathcal{O}(c^n) \\
&= \mathcal{O}(c^n)
\end{aligned}$$

(i) $T(n) = 2T(n-1) + 1$

$$\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2(T(n-2) + 1) + 1 \\
&= 2T(n-2) + 2 + 1 \\
&= 2(2(T(n-3) + 1)) + 2 + 1 \\
&= 4T(n-3) + 4 + 2 + 1 \\
&= (2^{n-1})T(2) + 2^{n-3} + 2^{n-4} + 2^1 + 2^0 \qquad \text{; where } T(2) = k \text{ ,for some constant.} \\
&= k2^{n-1} + \mathcal{O}(2^n) \\
&= \mathcal{O}(2^n)
\end{aligned}$$

(j) $T(n) = T(\sqrt{n}) + 1$

$$\begin{aligned}
T(n) &= T(\sqrt{n}) + 1 \\
&= T(n^{1/2}) + 1 \\
&= T(n^{1/4}) + 1 + 1 \\
&= T(n^{1/8}) + 1 + 1 + 1 \\
&= T(n^{1/m}) + m
\end{aligned}$$

Now we have to solve for m. We know that $n^{1/2^m} = 2$ so :

$$\begin{aligned}
n^{1/2^m} &= 2 \\
\log_n 2 &= 2^{-m} \\
\frac{1}{\log_2 n} &= \frac{1}{2^m} \\
2^m &= \log_2 n \\
m &= \log_2 \log_2 n
\end{aligned}$$

Therefore,

$$\begin{aligned}
T(n) &= T(2) + \log_2 \log_2 n \\
&= \mathcal{O}(\log_2 \log_2 n)
\end{aligned}$$

Problem 2

Algorithm A :

Running time: We can write its recurrence as $T(n) = 3T(n/3) + n$. And by Master Theorem, $a = 3,\ b = 3,\ d = 1$, which means $\log_3 3 = 1$. Thus, its running time is $\mathcal{O}(n \log n)$.

Algorithm B :

Running time: We can write its recurrence as $T(n) = T(n-1) + n$. And its running time is:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &= T(0) + 1 + 2 + ... + n && ; \text{where } T(0) = k \text{ for some constant.} \\ &= k + \mathcal{O}(n^2) \\ &= \mathcal{O}(n^2) \end{aligned}$$

Algorithm C :

Running time : We can write its recurrence as $T(n) = 2T(n/3) + n^2$. And by Master Theorem, $a = 2,\ b = 3,\ d = 2$, which means $\log_3 2 < 2$. Thus, its running time is $\mathcal{O}(n^2)$.

Algorithm D :

Running time : We can write its recurrence as $T(n) = 5T(n/4) + n$. And by Master Theorem, $a = 5,\ b = 4,\ d = 1$, which means $\log_4 5 > 1$. Thus, its running time is $\mathcal{O}(n^{\log_4 5})$.

From the asymptotic values,Algo B and C are definitely slower than Algo A, thus they are out.To compare Algo A and D we need to take their limits.

$$\begin{aligned} \lim_{n\to\infty} \frac{n \log n}{n^{\log_4 5}} &= \lim_{n\to\infty} \frac{1/n(n) + \log(n)}{\log_4 5 \cdot n^{\log_4 5 - 1}} \\ &= \lim_{n\to\infty} \frac{1/n}{\log_4 5 \cdot \log_4 5 - 1 \cdot n^{\log_4 5 - 2}} \\ &= 0 \end{aligned}$$

This shows that $n \log n \in \mathcal{O}(n^{\log_4 5})$. Therefore Algorithm A is the most efficient.

Problem 3

We can write a recurrence for it as : $T(n) = 3T(n/2) + 1$

Which can be solve to :

$$\begin{aligned} T(n) &= 3T(n/2) + 1 \\ &= 3(3T(n/4) + 1) + 1 \\ &= 9T(n/4) + 3 + 1 \\ &= 3^{\log n}T(1) + 3^{\log n - 1} + ... + 3^1 + 3^0; \text{ where } T(1) = 0 \\ &= 3^{\log n - 1} + ... + 3^1 + 3^0 \\ &= \frac{3^{\log n} - 1}{2} \end{aligned}$$

So, the total number of times "hello" printed is $\frac{3^{\log n}-1}{2} = \mathcal{O}(3^{\log n})$.

Problem 4

At the start we will have a variable max which stores the maximum value. Suppose we have an array $A[l...r-1]$, we first find the middle element $m = (l+r)/2$ where r and l are the indexes. If $A[m] > max$, we let $max = A[m]$. We then look at the $A[m-1]$. If $A[m] > A[m-1]$, we let $l = m+1$ and we repeat the steps again. Else, if $A[m] < A[m-1]$, we let $r = m$ and we repeat the steps again. Keep doing this until $r - l \leqslant 0$ and the we return max which will be the maximum value.

Problem 5

(a) The way to solve is as follows:

```
def majority(A):
  n = len(A)
  if (n <= 1):
    return A[0]
  else:
    m = n/2
    l_major = A[0...m]
    r_major = A[m+1...n]
    major = merge(l_major,r_major,A)
  return major

def merge(l_major,r_major,A):
  if(l_major == r_major) return l_major
  else if(l_major == None & r_major == None) return None
  else if(l_major == None) return r_major
  else if(r_major == None) return l_major
  else:
    for elm in A[0...len(A)]:
      l_count = r_count = 0
      if (elm == l ):
        l_count += 1
      else if (elm == r):
        r_count += 1
```

```
if (l_count == r_count): return None
else if (l_count > r_count): return l_major
else: return r_major
```

From the code, our merge function will take $\mathcal{O}(n)$ time. Therefore we can write the recurrence for "majority" function as $T(n) = 2T(n/2) + \mathcal{O}(n)$ which can be solve to $\mathcal{O}(n \log n)$. **(b)** Using the algorithm provided in the question paper, we can solve this problem in linear-time by just changing the merge function from part (a) as follows :

```
def majority(A):
   n = len(A)
   if (n <= 1):
      return A[0]
   else:
      m = n/2
      l_major = A[0...m]
      r_major = A[m+1...n]
      major = constant_merge(l_major,r_major,A)
   return major

def constant_merge(l_major,r_major):
   if(l_major == r_major): return l_major
   else if (r_major == None): return l_major
   else if (l_major == None): return r_major
   else: return None
```

With reference to the algorithm in the question paper, using the Pigeon-hole Principle, let the number of pairs be number of holes and the number of majority element be the number of pigeons. If more than half of the elements are the same, there will be more pigeons than holes. Thus, at least 1 element of the majority will be left which is not canceled out.

There will also be at most $n/2$ element left because in the case where all the elements are pair to the same type, only 1 element will be canceled out and we will have $n/2$ elements left.