Solutions - Problem Set #1

Alice Bob (uni). Collaborators: A. Turing. - `uni@columbia.edu`

September 16, 2019

1 Problem 1

(a) $f(n) = n^4 + 3n^2 - 4n + 1$ and $g(n) = (5n^4 + 1)(1 + \frac{1}{n})$. We have that $f = \Theta(g)$, since $g(n) \leq 10^{10} f(n) \leq 10^{20} g(n)$ for $n \geq 10^{30}$; hence, we have $f = O(g)$ and $f = \Omega(g)$, but not the other asymptotics. More formally, we have that:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{1}{5}.$$

(b) $f(n) = 4^{\log_2 n + 0.5}$ and $g(n) = n^2$. We may re-write:

$$\begin{aligned} f(n) &= 4^{\log_2 n + 0.5} \\ &= \sqrt{4} \cdot 4^{\frac{\log_4 n}{\log_4 2}} \\ &= 2 \cdot n^2 \end{aligned}$$

Thus, we have that $g(n) \leq \frac{1}{2} f(n) \leq 2g(n)$ for all $n \geq 0$. So we have $f = \Theta(g)$ (and hence $f = O(g)$ and $f = \Omega(g)$), but not the other asymptotics.

(c) $f(n) = \sum_{i=1}^{n} 1/i^2$ and $g(n) = 100^{1/n}$. We note that $1 \leq f(n) \leq 10$ for all $n \geq 0$. In addition, we have $g(n) \in [1, 2]$ for all $n \geq 10$. Thus, we have $g(n) \leq 10f(n) \leq 1000g(n)$ for all $n \geq 10$, so $f = \Theta(g)$ (and hence $f = O(g)$ and $f = \Omega(g)$), but not the other asymptotics.

(d) $f(n) = n^{n\%2}$ and $g(n) = n$ where $n\%c$ for an integer c represents $n \mod c$. We note that when n is even, $f(n) = 1$. On the other hand, for every $k > 100$, we have that $g(2^k) = 2^k$. So assume for contradiction that $f(n) \geq c \cdot g(n)$ for some constant c and $n \geq n_0$ (if we try to show Ω bound). Then, we may find a large enough value of k so that $2^k \geq n_0$. When this happens, we have: $g(2^k) = 2^k$ and $f(2^k) = 1$, which contradicts our inequality.

We can show that $f(n) = O(g(n))$. We have that $f(n) = n$ when n is odd and $f(n) = 1$ when n is even. Thus, $f(n) \leq 1g(n)$ for all $n \geq 1$, showing that $f(n) = O(g(n))$.

2 Problem 2

- Constants: π^{2019}.
- Doubly logarithmic: $\log\log n$
- Logarithmic: $\log n + \log\log n$, $\log(n^{0.19})$
- Super-logarithmic but sub-polynomial: $2^{\sqrt{\log n}}$.
- Square-root: $\log(2^{\sqrt{n}})$
- $n\log n$: $n\log n$, $\log(n!)$, since:

$$\log(n!) \leq \log(n^n) \leq n\log(n)$$
$$\log(n!) \geq \log((n/2)^{n/2}) \geq (n/2)\log(n/2)$$

- Polynomial: $n^{3+\sin(n)}$
- Polynomial (but with higher degree): $\binom{n}{n-19} = \binom{n}{19} = \Theta(n^{19})$
- Super-polynomial but sub-exponential: $n^{\log\log n}$, $(\log n)^{\log n}$; they are equal.
- Mild exponential dependence: $3^{\sqrt{n}}$.
- Exponential with power 2: 2^n, $2^{2^{\log n}+19} = 2^{n+19} = 2^{19}2^n$.
- Exponential with higher base: $2^{2^{(\log n+19)}} = 2^{2^{19}n}$.

3 Problem 3

(a) We have as input two sorted arrays A and B, as well as an integer c and want to determine if there exists integers $1 \leq i, j \leq n$ such that $2A[i] + 3B[j] = c$. This is equivalent to determining if there exists integers $1 \leq i, j \leq n$ such that $2A[i] = -3B[j] + c$. Note that when $B[j]$ is positive, $-3B[j]$ is negative and vice versa, so if we copy the array $-3B + c$ into array D, but reversed, it will be a sorted array. Similarly, consider the array $C[i] = 2A[i]$. Then, we may solve the above problem by solving the following simplified problem:

> Given two sorted arrays D and C of size at most n, determine whether there exists indices i and j such that $D[i] = C[j]$.

We solve the above simplified problem with the following algorithm:

1. We initialize $i \leftarrow 1$ and $j \leftarrow 1$, and as long as $i \leq n$ and $j \leq n$, we do the following:

2. We consider comparing $D[i]$ and $C[j]$:
   - If $D[i] = C[j]$, we return YES.
   - If $D[i] > C[j]$, we set $j \leftarrow j + 1$.
   - If $D[i] < C[j]$, we set $i \rightarrow i + 1$.
3. If we have not returned, return NO.

For correctness, note that if the algorithm returns an answer, then it has found a match. In addition, if i_0 and j_0 is a pair where $D[i_0] = C[j_0]$, then $i \leq i_0$ and $j \leq j_0$ throughout the execution of the algorithm. In order to see this, note that initially, $i \leq i_0$ and $j \leq j_0$, and if at some point i becomes greater than i_0, then it was because $D[i] < C[j]$, however,

$$D[i_0] < D[i] < C[j] < C[j_0],$$

which is a contradiction. The same argument holds when j becomes higher than j_0.

The above algorithm runs in linear time since at each step, i or j increases, and these can only increase n times each.

(b) For this problem, we first sort the array A in $O(n \log n)$ time, and then may run the above algorithm n times for each $c = A[k]$ for $k = \{1, \ldots, n\}$.

4 Problem 4

(a) $T(n) = 6T(n/9) + n$

Master Theorem. $a = 6$, $b = 9$, $f(n) = n$, which falls in Case 3. Thus, we have $O(n)$

(b) $T(n) = 3T(n/3) + n^{1.5}$

Master Theorem $a = 3$, $b = 3$, $f(n) = n^{1.5}$, which falls in Case 3. Thus, we have $O(n^{1.5})$.

(c) $T(n) = T(n-2) + 1/n$

Draw a tree. Get sum $1/n + 1/(n-2) + ... + 1$.

This is every other term of a harmonic series. Therefore, we have $O(\log n)$.

(d) $T(n) = 4T(n/2) + n^2 \lg n$

Master theorem doesn't apply. Draw a tree and compute. The height of this tree is $\log(n)$, and computing the sum of the work done at each level in the tree gives $O(n^2 \log^2(n))$.

(e) $T(n) = \sqrt{n}T(\sqrt{n}) + n$

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

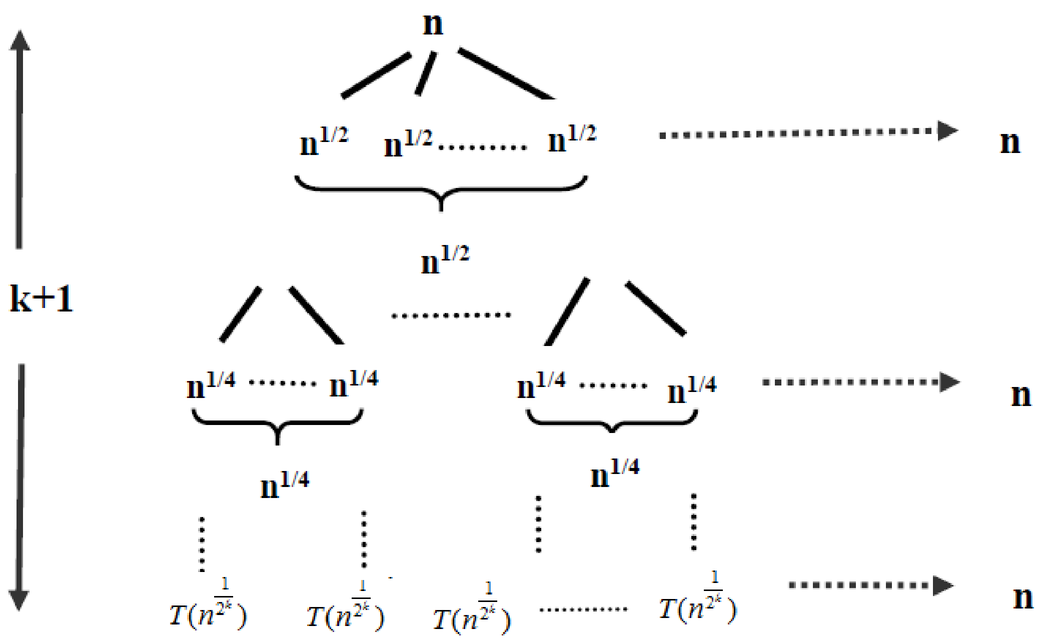

Figure 1: Recursion Tree of $T(n) = \sqrt{n}T(\sqrt{n}) + n$

The tree has $(k+1)$ levels. In each level, the sum is n. Assume $n^{\frac{1}{2^k}}$ is a small constant a such that $T(n^{\frac{1}{2^k}})$ is a constant. Therefore, $2^k = lg_a^n = \Theta(lgn)$. $k = \Theta(lglgn)$. $T(n) = (k+1)n = \Theta(nlglgn)$.

Or we can compute on the recurrence expression directly.

$$\begin{aligned}
T(n) &= \sqrt{n}T(\sqrt{n}) + n \\
&= \sqrt{n}(n^{\frac{1}{4}}T(n^{\frac{1}{4}}) + n^{\frac{1}{2}}) + n \\
&= n^{\frac{1}{2}+\frac{1}{4}}T(n^{\frac{1}{4}}) + 2n \\
&= n^{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}}T(n^{\frac{1}{8}}) + 3n \\
&= n^{\frac{1}{2}+\frac{1}{4}+\cdots+\frac{1}{2^k}}T(n^{\frac{1}{2^k}}) + kn \\
&= nT(n^{\frac{1}{2^k}}) + kn
\end{aligned}$$

Suppose $T(n^{\frac{1}{2^k}})$ is a constant c. Let $n^{\frac{1}{2^k}} = a$, $2^k = lg_a^n = \Theta(lgn)$. $k = \Theta(lglgn)$. Hence, $T(n) = (c+k)n = \Theta(nlglgn)$.

5 Problem 5

1. Run through the list from the beginning of the list to the end until you find the first 1.

2. Run through the list from the end of the list to the beginning until you find the first 1.

3. You run a binary search algorithm, which returns the index of the first 1:

   BinarySearch(A[1...n]):

   ```
   1 Let $p = \lceil n/2 \rceil$.
   2 **if** length of array is 1
   3     return 1
   4 **if** $A[p]$ is 0
   5     Return BinarySearch($A[p+1, \dots, n]$) + p
   6 **if** $A[p]$ is 1
   7     Return BinarySearch($A[1, \dots, p-1]$)
   ```

   Binary Search takes $O(\log n)$ runtime, so you don't run out of dollars or FATC.

4. The idea is to divide the array into bins of size $n^{0.5}$.

   ```
   1 Let $i = 1$. run = True
   2 **while** run is True and $n - in^{1/2} > 0$
   3     **if** $A[n - in^{0.5}]$ is 1
   4         i = i+1
   5     **if** $A[n - in^{0.5}]$ is 0
   6         run = False
   7 Search the bin $A[n - in^{0.5}, n - (i-1)n^{0.5}]$ from the end to the
     beginning until you find the first 1.
   ```

   The worst case runtime is when the first 1 is in the second position. In this case, I use up roughly $2\sqrt{n}$ FATC, which is still $O(\sqrt{n})$

5. We define $FATC_k$ to be the algorithm you use where you have k dollars and $O(n^{1/k})$ FATC. We use a recursive algorithm. The idea is that when you have k dollars, you want your bins to be size $n^{\frac{k-1}{k}}$. In using $\leq n^{1/k}$ FATC, I can reduce my problem to an array of size $n^{k-1/k} =: n_2$. From here I apply $FATC_{k-1}$ on this subproblem of size n_2.

   I claim that $FATC_k$ can be solved using $kn^{1/k}$ FATC, which would be $O(n^{1/k})$. This is easy to show by induction. The base case was already proved. By the induction hypothesis, $FATC_{k-1}$ when applied on an array of size n_2 can be solved using $(k-1)n_2^{1/(k-1)} = (k-1)n^{(\frac{k-1}{k})(\frac{1}{k-1})} = (k-1)n^{1/k}$ FATC. Thus, adding the initial $n^{1/k}$ FATC, I get that $FATC_k$ can be solved using $kn^{1/k}$ FATC.

$FATC_k$:

```
Let $i = 1$. run = True
while run is True and $n - in^{1/k} > 0$
    if $A[n - in^{\frac{k-1}{k}}]$ is 1
        i = i+1
    if $A[n - in^{\frac{k-1}{k}}]$ is 0
        run = False
return $FATC_{k-1}(A[n - in^{\frac{k-1}{k}}, n - (i-1)n^{\frac{k-1}{k}}])$
```