CSOR 4231, Section 3: Analysis of Algorithms I - Problem Set #1

Kangwei Ling - kl3076@columbia.edu

February 3, 2019

Collaborators

Zefeng Liu (zl2715), Kunyan Han (kh2931), Luoyao Hao (lh2913).

0.1

Assume all $\log = \log_2$. From slowest growing to fastest growing (same growing speed if in same line):

1. $\log 2n, \log 3n, 10 \log n, \log(n^2)$
2. $(\log n)^3$
3. $(\log n)^{10}$
4. $n^{0.1}$
5. $n^{1/2}, \sqrt{n}, n^{1/2}$
6. $n^{2/3}$
7. $n/\log n$
8. $n - 100, n - 200, 100n + \log n, n + (\log n)^2$
9. $n \log n, 10n \log 10n$
10. $n(\log n)^2, n(\log n)^2$
11. $n^{1.01}$
12. $n^2/\log n$
13. $5^{\log_2 n} \quad (5^{\log_2 n} = n^{\log_2 5})$
14. $(\log n)^{\log n}, (\log n)^{\log n} \quad ((\log n)^{\log n} = n^{\log \log n} = 2^{\log n \log \log n})$

15. $2^{(\log_2 n)^2}$

16. $2^n, 2^{n+1}, 2^n$

17. $n2^n$

18. 3^n

19. $n!$

$\sum_{i=1}^n i^k = \Theta(n^{k+1})$, since $(n/2)(n/2)^k < \sum_{i=1}^n i^k < n^{k+1}$.

2.4

A $T(n) = 5T(n/2) + O(n)$, solve this by the master theorem, we have have $T(n) = O(n^{\log_2 5})$

B $T(n) = 2T(n-1) + O(1)$,

$$\begin{aligned} T(n) = 2T(n-1) + O(1) &= 4T(n-2) + 2O(1) + O(1) \\ &= 2^{n-1}T(1) + \sum_{i=0}^{n-2} 2^i O(1) \\ &= \sum_{i=0}^{n-1} 2^i O(1) \quad \text{(assume } T(1) = O(1)\text{)} \\ &= O(2^n) \end{aligned}$$

C $T(n) = 9T(n/3) + O(n^2)$, solve this by the master theorem, we have $T(n) = O(n^2 \log_3 n) = O(n^2 \log n)$

I would choose between algo A and algo C, since algo B has an exponential complexity. And because $\log n = O(n^\epsilon), \epsilon > 0$, algo C is the best choice.

2.5

(a) $a = 2, b = 3, d = 0$, using the master theorem, we have $T(n) = \Theta(n^{\log_3 2})$

(b) $a = 5, b = 4, d = 1$, using the master theorem, we have $T(n) = \Theta(n^{\log_4 5})$

(c) $a = 7, b = 7, d = 1$, using the master theorem, we have $T(n) = \Theta(n \log_7 n)$

(d) $a = 9, b = 3, d = 2$, using the master theorem, we have $T(n) = \Theta(n^2 \log_3 n)$

(e) $a = 8, b = 2, d = 3$, using the master theorem, we have $T(n) = \Theta(n^3 \log n)$

(f) at level k, there are 49^k subproblems, each with size $\frac{n}{25^k}$, total work done at level k is,

$$49^k \times (\frac{n}{25^k})^{3/2} \log(\frac{n}{25^k}) = (\frac{49}{125})^k O(n^{3/2} \log n)$$

since $49/125 < 1$ therefore,

$$T(n) = O(n^{3/2} \log n)$$

also,

$$T(n) = 49T(n/25) + n^{3/2} \log n = \Omega(n^{3/2} \log n)$$

thus, $T(n) = \Theta(n^{3/2} \log n)$

(g) $T(n) = T(n-1) + 2 = T(n-2) + 2 + 2 = \Theta(n)$

(h) $T(n) = n^c + (n-1)^c + \cdots + 2^c + T(1) = \Theta(n^{c+1})$
(in 0.1, we know $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$)

(i) $T(n) = c^n + c^{n-1} + \cdots + c^2 + T(1) = \Theta(c^n)$

(j) $T(n) = 2T(n-1) + 1 = 4T(n-2) + 2 + 1 = 2^{n-1}T(1) + \sum_{i=0}^{n-2} 2^i = \Theta(2^n)$

(k) $T(n) = T(n^{1/2}) + 1 = T(n^{1/4}) + 2 = T(n^{1/2^k}) + k$, let $n = 2^m$ (for simplicity), we have $T(n) = \Theta(T(1) + k)$, where $k = \log m$, thus, $T(n) = \Theta(\log \log n)$

2.22

Let two list be $A = A[0], A[1], ..., A[M-1]$ and $B = B[0], B[1], ..., B[N-1]$, let $0 < k <= M + N$.

$findKth(A, B, s_A, s_B, m, n, k)$:

The input to the algorithm $findKth$ is A, B and the start index $(s_A, s_B)$and size of the current searching region size (m, n) of A and B respectively, together with the target k.

In each recursion step, if $m = 0$ or $n = 0$ we return the kth element of the other list respectively. Else if $k = 1$, we return $min(A[s_A], B[s_B])$.

Otherwise, we calculate $mid_A = (m+1)/2, mid_B = (n+1)/2$.

Case $mid_A + mid_B > k$:

If $A[s_A + mid_A - 1] >= B[s_B + mid_B - 1]$, then we are sure that the k th smallest will not be in $A[s_A + mid_A - 1], ...A[s_A + m - 1]$, therefore we can discard half of current search region of A and recursively call $findKth(A, B, s_A, s_B, mid_A - 1, n, k - (m - mid_A + 1))$.

Similarly, if $A[s_A + mid_A - 1] < B[s_B + mid_B - 1]$, we can discard half of current search region of B and recursively call $findKth(A, B, s_A, s_B, m, mid_B - 1, k - (n - mid_B + 1))$.

Case $mid_A + mid_B <= k$:

If $A[s_A + mid_A - 1] <= B[s_B + mid_B - 1]$, then we are sure that the k th smallest will not be in $A[s_A], ..., A[s_A + mid_A - 1]$, therefore we can discard half of current search region of A and recursively call $findKth(A, B, s_A + mid_A, s_B, m - mid_A, n, k - (m - mid_A))$.

Similarly, if $A[s_A + mid_A - 1] > B[s_B + mid_B - 1]$, we can discard half of current search region of B and recursively call $findKth(A, B, s_A, s_B + mid_B, m, n - mid_B, k - (n - mid_B))$.

This algorithm is $O(\log m + \log n)$ because one of the list A, B 's size is halved.

2.33

(a) Suppose $\boldsymbol{m}$ is a nonzero row of M. Then $Pr(M\boldsymbol{v} = 0) \leq Pr(\sum_{i=0}^{n} m_i v_i = 0)$. Since $\boldsymbol{m}$ is nonzero, there must exist a $m_k \neq 0$, then we can write $\sum_{i=0}^{n} m_i v_i = m_k v_k + \sum_{i \neq k} m_i v_i$.

$$\sum_{i=0}^{n} m_i v_i = 0 \Leftrightarrow v_k = \frac{-\sum_{i \neq k} m_i v_i}{m_k}$$

Note that for any randomly chosen $\boldsymbol{v}$, $Pr(v_k = \frac{-\sum_{i \neq k} m_i v_i}{m_k}) \leq 1/2$ because v_k can only be 0 or 1, each with probability 1/2. Therefore $Pr(M\boldsymbol{v} = 0) \leq Pr(\sum_{i=0}^{n} m_i v_i = 0) \leq 1/2$

(b) If $AB \neq C$, then $AB - C \neq \mathbf{0}$, therefore $Pr((AB - C)\boldsymbol{v} = 0) \leq 1/2$, which is exactly $Pr(AB\boldsymbol{v} = C\boldsymbol{v}) \leq 1/2$.

This statement implies that for a randomly chosen $\boldsymbol{v}$, we calculate $(AB - C)\boldsymbol{v}$, if it does not equal to $\mathbf{0}$, then we are sure that $AB \neq C$. If it does equal to $\mathbf{0}$, then there is a probability $\leq 1/2$ that we are wrong if we draw the conclusion that $AB = C$, we can continue to test to be more confident about this.

To use this conclusion in testing $AB = C$ in $O(n^2)$:

The calculation of $AB\boldsymbol{v}$ can be done in $O(n^2)$, since we can calculate $B\boldsymbol{v}$, then $A(B\boldsymbol{v})$, same for $C\boldsymbol{v}$. Checking $(AB - C)\boldsymbol{v} = \mathbf{0}$ is $O(n)$. Thus, each single test is $O(n^2)$.

After k independent test, we can draw the conclusion with the probability to make a mistake less than $1/2^k$, this is very small even with $k = 20$.

The total actual complexity for testing is $O(kn^2)$, but k can be viewed as a constant because $1/2^k$ decreases really fast.

Therefore we have an algorithm of $O(n^2)$ to check $AB = C$.

6

a. When $k = 1$, we can only try throwing the mug starting from $1, 2, 3, ..., n - 1$, thus we need $n - 1$ steps for the worst case. If we can use fewer steps (by skipping some millimeters or some other strategy), at the point of a mug breaks, we cannot get the exact largest number of millimeters that the mug will not break.

b. Suppose we test the first mug at height h, if it breaks, we need $h-1$ more tests from 1 to $h-1$ linearly in the worst case using the second mug, this implies that at least $1+h-1=h$ steps are required for the worst case. In order to minimize the worst case number of steps, we can make sure that all other cases (described below) does not use more than h steps.

If the first mug does not break at the first try at h, then we can try another height h', h' must be as high as possible because we want to use as fewer steps as possible to reach $n-1$. As a result, $h' = h+h-1$, h' cannot be larger, otherwise we need more than h steps in total if it breaks at h'.

In this reasoning, we must arrange our dropping at $h, h+h-1, h+h-1+h-2, ..., h+h-1+h-2+\cdots+1$. To make sure that we can cover height $n-1$, it must hold that,

$$h+h-1+h-2+\cdots+1 \geq n-1$$

Solve this, we have $h_{min} = \lceil \frac{-1+\sqrt{8n-7}}{2} \rceil$. Thus, we proceed our test at height $h_{min}, h_{min}+h_{min}-1, ..., h_{min}+h_{min}-1+\cdots+1$, once break at some height, we make linear trials between the previous safe height and the current height.

By this way, the minimum number of steps in worst case $= O(h_{min}) = O(\sqrt{n})$

c. Let $T(n,k)$ be the minimum steps needed in the worst case for searching height $[0,n]$ and k mugs. Suppose p is the first height we try in the optimized testing procedure. Depending on whether the mug breaks or not, we end up with two branch to recursion. If it breaks, the remaining problem is $T(p-1,k-1)$. If it is still intact, then $T(n-p,k)$.

Therefore $T(n,k) = 1 + max(T(p-1,k-1), T(n-p,k))$. To make p the optimal height to try, it must hold that

$$max(T(p-1,k-1), T(n-p,k)) = \min_{1\leq i\leq n}\{max(T(i-1,k-1), T(n-i,k))\}$$

Thus,

$$T(n,k) = 1 + \min_{1\leq i\leq n}\{max(T(i-1,k-1), T(n-i,k))\}$$

The base cases are easy: $T(0,k)=0, T(1,k)=1, T(n,1)=n$. When calculating the T, we can store a back pointer to find out the exact best way to test the mugs.