# CS21201 Discrete Structures

## Tutorial 9

---

### Recurrences

**1.** [*Stooge sort*] We want to sort an array $A$ of $n$ integer. We run the following algorithm.

> If $n \leqslant 2$, sort $A$ manually, otherwise do the following.
> Recursively sort the first $\lceil 2n/3 \rceil$ elements of $A$.
> Recursively sort the last $\lceil 2n/3 \rceil$ elements of $A$.
> Recursively sort the first $\lceil 2n/3 \rceil$ elements of $A$.

**(a)** Prove that this algorithm sorts $A$ correctly.

*Solution* The first two recursive calls bring the largest one-third elements of $A$ to their correct positions.

**(b)** Find the running time of this algorithm.

*Solution* Neglecting the ceilings, we can write $T(n) = 3T(2n/3) + c$, where $c$ is a constant. We have $\delta = \log_{3/2} 3 = 2.709511\ldots$ and $d = 0$, so by the master theorem, $T(n) = \Theta(n^{\log_{3/2} 3}) = \Theta(n^{2.709511\ldots})$.

**2.** Solve for the following divide-and-conquer recurrence: $T(n) = 2T(n/2) + \dfrac{n}{\log n}$ with $T(1) = 1$.

*Solution* Dividing both the sides of the given recurrence by $n$, we obtain

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + \frac{1}{\log n}.$$

Assuming $n = 2^k$ and $S(k) = \frac{T(n)}{n} = \frac{T(2^k)}{2^k}$, we can rewrite the above as

$$S(k) = S(k-1) + \frac{1}{k} \quad \text{with } S(0) = 1.$$

Unwinding gives

$$\begin{aligned} S(k) &= S(k-1) + \frac{1}{k} \\ &= S(k-2) + \frac{1}{k-1} + \frac{1}{k} \\ &= S(k-3) + \frac{1}{k-2} + \frac{1}{k-1} + \frac{1}{k} \\ &= \cdots = S(0) + \sum_{i=1}^{k} \frac{1}{i}, \end{aligned}$$

that is, $T(n) = nS(k) = n\left[1 + \sum_{i=1}^{\log n} \frac{1}{i}\right]$. We have $H_m = \sum_{i=1}^{m} \frac{1}{i} = \Theta(\log m)$. Therefore $T(n) = \Theta(n \log\log n)$.

**3.** Solve the following recurrence relation, and deduce the closed-form expression for $T(n)$.

$$T(n) = \begin{cases} \sqrt{n}\,T(\sqrt{n}) + n(\log_2 n)^d, & \text{if } n > 2 \\ 2 \qquad , & \text{if } n = 2 \end{cases} \quad (d \geqslant 0).$$

*Solution* Given that $T(n) = \sqrt{n}\,T(\sqrt{n}) + n\log_2^d n$ (where $d \geqslant 0$) and $T(2) = 2$, we have:

$$\frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + \log_2^d n \qquad \ldots\left[\textit{dividing both sides by } n\right]$$

$$\Rightarrow \quad S(n) = S(\sqrt{n}) + \log_2^d n \qquad \ldots\left[\textit{assuming } S(n) = \frac{T(n)}{n}\right]$$

$$\Rightarrow \quad S(2^{2^k}) = S(2^{2^{(k-1)}}) + (2^k)^d \qquad \ldots\left[\textit{substituting } n = 2^{2^k}\right]$$

$$\Rightarrow \quad R(k) = R(k-1) + (2^k)^d \qquad \ldots\left[\textit{let } R(k) = S(2^{2^k})\right]$$

$$\Rightarrow \quad R(k) = R(0) + (2^d)^1 + (2^d)^2 + \cdots + (2^d)^{k-1} + (2^d)^k \qquad \ldots\left[\textit{because } (2^k)^d = 2^{kd} = (2^d)^k\right]$$

$$\Rightarrow \quad R(k) = 1 + \sum_{i=1}^{k} (2^d)^i \qquad \ldots\left[S(2) = \frac{T(2)}{2} = 1,\ \textit{implying } R(0) = S(2^{2^0}) = 1\right]$$

$$\Rightarrow \quad R(k) = \begin{cases} \frac{(2^d)^{(k+1)}-1}{2^d-1}, & \text{if } d > 0 \\ 1+k, & \text{if } d = 0 \end{cases}$$

$$\therefore\ R(k) = S(2^{2^k}) = \begin{cases} \frac{(2^d)^{(k+1)}-1}{2^d-1}, & \text{if } d > 0 \\ 1+k, & \text{if } d = 0 \end{cases}, \quad \text{where } n = 2^{2^k} \text{ and } S(n) = \frac{T(n)}{n}.$$

Finally,

$$S(n) = \begin{cases} \frac{2^d \log_2^d n - 1}{2^d-1}, & \text{if } d > 0 \\ 1 + \log_2 \log_2 n, & \text{if } d = 0 \end{cases} \quad \Rightarrow \quad T(n) = \begin{cases} \frac{2^d n \log_2^d n - n}{2^d-1}, & \text{if } d > 0 \\ n + n\log_2 \log_2 n, & \text{if } d = 0 \end{cases}$$

**4.** Suppose that a recursive algorithm on an input of size $n$ makes two recursive calls: the first one is on an input of size $\lceil n/5 \rceil$, and the second one is on an input of size $\lceil 7n/10 \rceil$. In addition to these call, the algorithm takes time proportional to $n$. Express the running time as a recurrence, and solve for $T(n)$ in the big-$\Theta$ notation. (**Remark:** This recurrence appears in the analysis of a worst-case linear-time median-finding algorithm.)

*Solution* We have

$$T(n) = T(\lceil n/5 \rceil) + T(\lceil 7n/10 \rceil) + cn$$

for some constant of proportionality $c$. First of all, this implies that $T(n) \geqslant cn$. Next, we show that we can choose a positive constant $d$ such that $T(n) \leqslant dn$. We proceed by induction on $n$. The recurrence gives

$$\begin{aligned} T(n) &= T(\lceil n/5 \rceil) + T(\lceil 7n/10 \rceil) + cn \\ &\leqslant d\lceil n/5 \rceil + d\lceil 7n/10 \rceil + cn \\ &\leqslant d((n/5)+1) + d((7n/10)+1) + cn \\ &= d(9/10)n + 2d + cn \\ &= n[(9/10)d + (2/n) + c]. \end{aligned}$$

Let us choose any $d \geqslant 10(2+c)$. But then, we have

$$T(n) \leqslant n[(9/10)d + (2/n) + c] \leqslant n[(9/10)d + 2 + c] \leqslant n[(9/10)d + (1/10)d] = dn.$$

* **5.** [*Quick select*] You are given an array $A$ of $n$ distinct integers, and an integer $r$ in the range $1 \leqslant r \leqslant n$. Your task is to find the $r$-th smallest element of $A$ (that is, the element of rank $r$ in $A$). To this end, you choose a uniformly random element $p$ of $A$, and partition $A$ using $p$ as the pivot. Let the pivot go to position $k$ (assume one-based indexing of arrays). If $k = r$, then you return $p$. If $k > r$, you recursively find the $r$-th smallest element in the array consisting of elements of $A$ smaller than $p$. Finally, if $k < r$, you recursively find the $(r-k)$-th smallest element in the array consisting of elements of $A$ larger than $p$.

**(a)** Prove that the expected running time $T(n)$ of this algorithm satisfies the recurrence

$$T(n) \leqslant cn + \frac{1}{2}T(3n/4) + \frac{1}{2}T(n-1),$$

where $c$ is a constant.

*Solution* For simplicity, we may assume that $n$ is a multiple of 4. If not, you can find a few smallest elements of $A$, remove them from $A$, and adjust $k$ accordingly. This takes $O(n)$ time which when added to the partitioning stage does not change the complexity of the algorithm. $A$ consists of three parts: the smallest quarter $S$, the largest quarter $L$, and the middle half $M$. With probability $\frac{1}{2}$ the pivot $p$ is an element of $M$, and with probability

$\frac{1}{2}$ it is an element in $S$ or $L$. If $p$ is in $M$, then either $L$ or $R$ is eliminated from the array in the recursive call. Some elements of $M$ may also be eliminated, but we can ignore it because we are computing an upper bound on $T(n)$. On the other hand, if $p$ is from $S$ or $L$, then the worst-case array size in the recursive call is $n-1$.

**(b)** Prove that $T(n) = \Theta(n)$.

*Solution* Since partitioning already takes $\Theta(n)$ time, we have $T(n)$ is $\Omega(n)$. Using the recurrence, we show that $T(n)$ is $O(n)$ too. To that end, we show that $T(n) \leqslant dn$ for some constant $d$ for all sufficiently large $n$. Substituting this form in the recurrence gives

$$\begin{aligned} T(n) &\leqslant cn + \frac{1}{2}T(3n/4) + \frac{1}{2}T(n-1) \\ &\leqslant cn + \frac{1}{2} \times \frac{3dn}{4} + \frac{1}{2} \times d(n-1) \\ &= \left[c + \left(\frac{7}{8} - \frac{1}{2n}\right) d\right] n \\ &\leqslant \left[c + \frac{7}{8} d\right] n. \end{aligned}$$

It we choose any constant $d \geqslant 8c$, we have $T(n) \leqslant dn$.

## Additional Exercises

**6.** Find big-$\Theta$ estimates for the following positive-real-valued increasing functions $f(n)$.

**(a)** $f(n) = 125f(n/4) + 2n^3$ whenever $n = 4^t$ for $t \geqslant 1$.
**(b)** $f(n) = 125f(n/5) + 2n^3$ whenever $n = 5^t$ for $t \geqslant 1$.
**(c)** $f(n) = 125f(n/6) + 2n^3$ whenever $n = 6^t$ for $t \geqslant 1$.

**7.** Let the running time of a recursive algorithm satisfy the recurrence

$$T(n) = aT(n/b) + cn^d \log^e n$$

for some $e \in \mathbb{N}$. Let $t = \log_b a$. Deduce the running time $T(n)$ in the big-$\Theta$ notation for the three cases: (i) $t < d$, (ii) $t > d$, and (iii) $t = d$.

**8.** Let $t$ be the number of one-bits in $n$. Suppose that the running time of a divide-and-conquer algorithm satisfies the recurrence $T(n) = 2T(n/2) + nt$. When $n$ is a power of 2, we have $t = 1$, so $T(n) = 2T(n/2) + n$. Why does this not imply that $T(n) = \Theta(n \log n)$? Find a correct estimate for $T(n)$ in the big-O notation. (**Remark:** There exist algorithms whose running times depend on $t$. Example: Left-to-right exponentiation.)

**9.** Let the running time of a recursive algorithm satisfy the recurrence

$$T(n) = aT(\sqrt{n}) + h(n).$$

Deduce the running time $T(n)$ in the big-$\Theta$ notation for the cases: (i) $h(n) = n^d$ for some $d \in \mathbb{N}$, and (ii) $h(n) = \log^d n$ for some $d \in \mathbb{N}_0$.

**10.** [*Karatsuba multiplication*] You want to multiply two polynomials $a(x)$ and $b(x)$ of degree (or degree bound) $n-1$. Each of the input polynomials is stored in an array of $n$ floating-point variables. The product $c(x) = a(x)b(x)$ is of degree (at most) $2n-2$, and can be stored in an array of size $2n-1$.

**(a)** Use the school-book multiplication method to compute $c(x)$ (use the convolution formula). Deduce the running time of this algorithm.

**(b)** Let $t = \lceil n/2 \rceil$. Divide the input polynomials as $a(x) = x^t a_{hi}(x) + a_{lo(x)}$ and $b(x) = x^t b_{hi}(x) + b_{lo(x)}$, where each part of $a$ and $b$ is a polynomial of degree $\leqslant t-1$. But then

$$c(x) = a_{hi}(x)b_{hi}(x)x^{2t} + \Big(a_{hi}(x)b_{lo}(x) + a_{lo}(x)b_{hi}(x)\Big)x^t + a_{lo}(x)b_{lo}(x).$$

The obvious recursive algorithm uses this formula to compute $c(x)$ by making four recursive calls on polynomials of degrees $\leqslant t-1$. Deduce the running time of this algorithm.

**(c)** Reduce the number of recursive calls to three (how?). Deduce the running time of this algorithm.

**11.** In the quick-sort algorithm, two recursive calls are made on arrays of sizes $i$ and $n-i-1$ for some $i \in \{0,1,2,\ldots,n-1\}$ (assuming that there are no duplicates in the input array). Suppose that all these values of $i$ are equally likely. Deduce the expected running time of quick sort under these assumptions.

**12.** Suppose that an algorithm, upon an input of size $n$, recursively solves two instances of size $n/2$ and three instances of size $n/4$. Let the "divide + combine" time be $h(n)$. Find the running times of the algorithm if

**(a)** $h(n) = 1$, **(b)** $h(n) = n$, **(c)** $h(n) = n^2$, **(d)** $h(n) = n^3$.

**13.** Deduce the running times of divide-and-conquer algorithms in the big-$\Theta$ notation if their running times satisfy the following recurrence relations.

**(a)** $T(n) = T(2n/3) + T(n/3) + 1$.
**(b)** $T(n) = T(2n/3) + T(n/3) + n$.
**(c)** $T(n) = T(2n/3) + T(n/3) + n\log n$.
**(d)** $T(n) = T(2n/3) + T(n/3) + n^2$.

**14.** Deduce the running times of divide-and-conquer algorithms in the big-$\Theta$ notation if their running times satisfy the following recurrence relations.

**(a)** $T(n) = T(n/5) + T(7n/10) + 1$.
**(b)** $T(n) = T(n/5) + T(7n/10) + n$.
**(c)** $T(n) = T(n/5) + T(7n/10) + n\log n$.
**(d)** $T(n) = T(n/5) + T(7n/10) + n^2$.

**15.** Consider the following variant of stooge sort for sorting an array $A$ of size $n$.

1. Recursively sort the first $\lceil 3n/4 \rceil$ elements of $A$.
2. Recursively sort the last $\lceil 3n/4 \rceil$ elements of $A$.
3. Recursively sort the first $\lceil n/2 \rceil$ elements of $A$.

**(a)** Prove that this algorithm correctly sorts $A$.
**(b)** Derive the asymptotic running time of this algorithm.