باسمه تعالی
دانشگاه تهران - دانشکده مهندسی برق و کامپیوتر

ساختمان‌داده‌ها و الگوریتم‌ها

پاسخ تمرین اول - پیچیدگی و الگوریتم های بازگشتی

کوروش سجادی

تاریخ تحویل: ۱۴۰۲/۱۲/۲۷

۱. ۲۰ نمره

پیچیدگی زمانی هر یک از قطعه کدهای زیر را محاسبه کنید.

الف)

```
int i, j, k = 0;
for(i = n/2; i <= n; i++) {
    for(j = 2; j <= n; j = j * 2) {
        k = k + n/2;
    }
}
```

ب)

```
int i = n;
while(i > 1) {
    int j = 1;
    while(j < n) {
        j = j*5;
        i = i/3;
        cout << "*";
    }
}
```

ج)

```
void function(int n) {
    int count = 0;
    for(int i = 0; i < n; i++)
        for(int j = i; j < i*i; j++)
            if(j%i == 0) {
                for(int k = 0; k < j; k++)
                    print("*");
            }
}
```

پاسخ :

الف) درونی ترین حلقه هر بار ۲ برابر میشود و تا وقتی که به n برسد log(n) بار تکرار میشود و حلقه بیرونی نصف n تکرار میشود که در کل چون حلقه ها تو در تو هستند nlog(n) بار تکرار میشود که پیچیدگی قطعه کد ما است.

ب) با هر بار تکرار حلقه بیرونی (حلقه ای که شرط خاتمه اش روی i است). حلقه داخلی log_5^n بار اجرا میشود و در هر بار اجرا حلقه داخلی i تقسیم بر ۳ میشود و این بدان معناست که i در هر بار اجرای حلقه بیرونی تقسیم بر $3^{log_5^n}$ میشود و این این کار تا زمانی ادامه دارد که i > ۱ باشد بنابراین حلقه خارجی $log_{3^{log_5^n}}^n$ بار اجرا میشود که هر بار اجرا آن حلقه داخلی log_5^n بار اجرا می شود بنابراین در کل پیچیدگی زمانی برنامه ما :

$$log_{3^{log_5^n}}^n * log_5^n = \frac{1}{log_5^n} * log_3^n * log_5^n = log_3^n$$

ج) درونی ترین حلقه k هایی است که j آن بر i بخش پذیراند و چون خود j در مرتبه i^2 است در ماکسیموم حالت (بدترین حالت) میتوان آن را در مرتبه n^2 دانست حلقه میانی نیز همان گونه که گفته شد در مرتبه n^2 است اولین حلقه نیز n بار اجرا میشود پس در کل بدلیل تو در تو بودن حلقه ها پیچیدگی زمان کلی $O(n^5)$ میشود.

۲. ۱۰ نمره

روابط زیر را رد یا اثبات کنید برای مورد آخر گزاره را بررسی کنید.

الف) $log_2 f(n) \in \theta(log_2 g(n)) \Rightarrow f(n) \in \theta(g(n))$

ب) $f(n) \in \Omega(g(n)),\ g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$

ج) $f(n) \in O(g(n))$ or $f(n) \in \Omega(g(n))$ or $f(n) \in \theta(g(n))\ ExactlyOneOfTheseRelationsOccur$

پاسخ :

الف) نادرست مثال نقض:

$$f(n) = n^2, g(n) = n^3 \Rightarrow n^2 \neq \theta(n^3)$$

ب) درست اثبات:

$$f(n) = \Omega(g(n)) : \exists c, n_0 ; n > n_0 \Rightarrow f(n) \geq c(g(n))$$
$$g(n) = \Omega(h(n)) : \exists c, n_0 ; n > n_0 \Rightarrow g(n) \geq c(h(n))$$
$$\Rightarrow f(n) \geq c(h(n)) \Rightarrow f(n) \in \Omega(h(n))$$

ج) نادرست مثال نقض:
در نظر بگیرید که $f(n) = 2n$ و $g(n) = n$. در این حالت، $f(n) \in O(g(n))$ و همچنین $f(n) \in \Omega(g(n))$، ولی هیچکدام از این روابط به تنهایی رخ نمی‌دهند بلکه هر دو رخ می‌دهند که با گزاره "تنها یکی از این روابط رخ می‌دهد" در تضاد است.

۳. ۲۰ نمره

پیچیدگی روابط بازگشتی زیر را با استفاده از روش‌های گفته شده به دست آورید.

الف) $T(n) = T(\sqrt{n}) + O(log(log(n)))$

ب) $T(n) = ۲۵T(n/۵) + n^۲$

ج) $T(n) = T(۳n/۴) + T(n/۴) + ۱/۲n^۲(۱ - sin(n))$

پاسخ :

الف) برای حل این رابطه بازگشتی، از تغییر متغیر $n = ۲^{۲^m}$ استفاده می‌کنیم. با این تغییر متغیر، داریم:

$$T(۲^{۲^m}) = T(۲^{۲^{m-۱}}) + O(m).$$

با نام‌گذاری دوباره $S(m) = T(۲^{۲^m})$، رابطه به شکل $S(m) = S(m-۱) + O(m)$ در می‌آید. این نشان می‌دهد که پیچیدگی زمانی برابر با $O(m^۲)$ است، که با بازگرداندن $m = \log(\log(n))$، پیچیدگی نهایی $O((\log(\log(n)))^۲)$ خواهد بود.

ب) با استفاده از قضیه اصلی ، می‌توانیم پیچیدگی زمانی این رابطه را به دست آوریم. در اینجا $a = ۲۵$، $b = ۵$، و $f(n) = n^۲$. مقایسه‌ی $n^{\log_b a} = n^{\log_۵ ۲۵} = n^۲$ با $f(n) = n^۲$ نشان می‌دهد که ما در حالت دوم قضیه اصلی هستیم. بنابراین پیچیدگی زمانی $T(n) = \Theta(n^۲ \log n)$ است.

ج) این رابطه بازگشتی به طور مستقیم توسط قضیه اصلی پوشش داده نمی‌شود به دلیل وجود تابع $\sin(n)$. با این حال، می‌توان توجه داشت که $۱/۲n^۲(۱ - \sin(n))$ در بدترین حالت (که $\sin(n) = -۱$) $\Theta(n^۲)$ می‌شود. بنابراین، می‌توان گفت که پیچیدگی این رابطه حداقل $\Theta(n^۲)$ است. با توجه به این که $T(n) = T(۳n/۴) + T(n/۴) + O(n^۲)$، انتظار می‌رود پیچیدگی آن بیشتر از $\Theta(n^۲)$ باشد، اما برای دقت بیشتر نیاز به تحلیل عددی یا روش‌های تحلیلی پیچیده‌تری داریم .

برای تحلیل این رابطه بازگشتی، می‌توانیم از روش درخت استفاده کنیم. در هر مرحله، هزینه‌ای معادل $n^۲$ (به طور تقریبی) داریم. با توجه به تقسیم بندی مسئله به دو زیر مسئله با اندازه‌های $۳n/۴$ و $n/۴$ و هزینه‌ای تقریبی $n^۲$ در هر سطح، کل هزینه را می‌توان به صورت زیر نوشت:

$$T(n) = n^۲ \sum_{i=۰}^{\infty} \left(\frac{۵}{۸}\right)^i = n^۲ \cdot \frac{۸}{۳} = \frac{۸}{۳}n^۲,$$

که نشان می‌دهد پیچیدگی زمانی این رابطه $O(n^۲)$ است.

۴.　　۱۰ نمره

توابع زیر را براساس پیچیدگی زمانی آنها مرتب کنید.

الف) $log(n)!, log(log*(n)), log*(log(n)), n^{۹}, ۵^{n}, n۲^{n}, ۱۰۰n^{۷}$

ب) $n^{n(log(log(n)))}, ۱۰^{۱۰۰}, \sum_{i=۰}^{n} \frac{n^i}{i!}, n^{۱۰۰}, ۱۰^{۹}n, log(n), log*(n)$

ج) $\sum_{j=۱}^{n} \sum_{i=۱}^{j} i, n^{\frac{۱}{log(n)}}, \frac{۱}{۱۰۰}logn, ۱۰n, n^{۴}, nlog(n), n^{n}$

پاسخ:

الف) $log^*(log(n)) < log(log^*(n)) < log(n)! < ۱۰۰n^{۷} < n^{۹} < n۲^{n} < ۵^{n}$

ب) $۱۰^{۱۰۰} < log*(n) < log(n) < ۱۰^{۹}n < n^{۱۰۰} < \sum_{i=۰}^{n} \frac{n^i}{i!} < n^{n(log(log(n)))}$

ج) $n^{\frac{۱}{log(n)}} < \frac{۱}{۱۰۰}log(n) < ۱۰n < nlog(n) < \sum_{j=۱}^{n} \sum_{i=۱}^{j} i < n^{۴} < n^{n}$

$$\sum_{i=۰}^{n} \frac{n^i}{i!} = e^n$$

برای نشان دادن اینکه $n^{\frac{۱}{\log(n)}}$ یک ثابت است، در نظر بگیرید که:

$$n^{\frac{۱}{\log_n(۲)}} = ۲$$

این بیان به ما می‌گوید که اگر پایه لگاریتم n باشد و $n = ۲$، آنگاه عبارت به یک ثابت ساده می‌شود. این اصل را می‌توان به هر پایه n تعمیم داد تا نشان دهیم $n^{\frac{۱}{\log(n)}}$ مستقل از n و در نتیجه یک ثابت است.

$$\sum_{j=۱}^{n} \sum_{i=۱}^{j} i = n^{۳}$$

۵.　　۲۰ نمره

پیچیدگی زمانی قطعه کد های زیر را با نوشتن رابطه بازگشتی آن محاسبه کنید.

الف)

```
int Sum(int n) {
    if(n == 1)
        return 1;
    return n + Sum(n-1);
}
```

ب)

```
int Find(int a[], int x) {
    switch (len(a)) {
        case 0:
            return 0;
        case 1:
            if(x <= a[0])
                return 0;
            return 1;
    }
    mid = 1 + (len(a) - 1) / 2;
    if(x <= a[mid -  1])
        return Find(a[:mid], x);
    return mid + Find(a[mid:], x);
}
```

ج)

```
void f(int A[]) {
    n = len(A);
    sq = sqrt(n) // square root
    for(i = 1; i < n; i++)
        cout << "*";
    if(n == 1)
        return;
    itr = 0;
    while(itr < n) {
        f(A[itr:itr+sq])
        itr += sq;
    }
}
```

پاسخ :

الف) تابع Sum یک تابع بازگشتی ساده است که جمع اعداد تا n را محاسبه می‌کند. رابطه بازگشتی آن به صورت زیر است:

$$T(n) = T(n - ۱) + O(۱)$$

با حل این رابطه بازگشتی، پیچیدگی زمانی $O(n)$ به دست می‌آید.

ب) تابع Find به نظر می‌رسد که یک الگوریتم جستجوی دودویی را پیاده‌سازی می‌کند. رابطه بازگشتی برای این تابع:

$$T(n) = T\left(\frac{n}{۲}\right) + O(۱)$$

با حل این رابطه، پیچیدگی زمانی $O(\log n)$ به دست می‌آید.

ج) تابع f یک عملیات تکراری روی آرایه انجام می‌دهد و سپس به صورت بازگشتی خودش را روی زیرآرایه‌هایی با اندازه ریشه دوم n فراخوانی می‌کند. رابطه بازگشتی برای این تابع می‌تواند به صورت زیر نوشته شود:

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

این رابطه نشان دهنده پیچیدگی زمانی است که تحلیل دقیق‌تری نیاز دارد، اما نشان می‌دهد که این تابع با سرعتی بیشتر از خطی ولی کندتر از $O(n^۲)$ اجرا می‌شود.

با استفاده از تغییر متغیر $m = \log n$ و تقسیم رابطه بر n و نوشتن مجدد رابطه بازگشتی، ما به رابطه زیر می‌رسیم:

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

تبدیل می‌شود به:

$$S(m) = S(m/۲) + ۱$$

که در آن $S(m) = \frac{T(۲^m)}{۲^m}$. با حل این رابطه جدید، به پیچیدگی زمانی $O(n \log \log n)$ می‌رسیم. این نشان می‌دهد که تابع با پیچیدگی زمانی بالاتر از خطی ولی کمتر از $O(n^۲)$ اجرا می‌شود، که دقیقاً $O(n \log \log n)$ است.

۶. ۱۰ نمره

توضیح دهید که چگونه میتوان پیچیدگی زمانی الگوریتمی که هم جزء iterative و هم جزء recursive دارد را محاسبه نمود سپس راه بیان شده خود را بر روی قطعه کد زیر اجرا نمایید .

```
int modifiedBSearch(arr, target):
    if (len(arr) == 0)
        return -1;

    mid = len(arr) / 2;
    if(arr[mid] == target)
        return mid;

    for(i = mid + 1; i < len(arr); i++)
        if(arr[i] == target)
            return i;

    return modifiedBSearch(arr[:mid], target);
```

پاسخ :

برای تعیین پیچیدگی زمانی یک الگوریتم که هم دارای جزء تکراری (iterative) و هم جزء بازگشتی (recursive) است، ابتدا باید هر بخش را به طور جداگانه تحلیل کرده و سپس پیچیدگی‌های آنها را ترکیب نماییم. ابتدا، مورد پایه بازگشتی را شناسایی کرده و پیچیدگی زمانی آن را محاسبه می‌کنیم. سپس، پیچیدگی بخش تکراری را با بررسی حلقه‌ها یا ساختارهای تکراری دیگر تحلیل می‌کنیم. در نهایت، این پیچیدگی‌ها را با استفاده از نمادگذاری مناسب ریاضی ترکیب می‌کنیم.

در قطعه کد داده شده، بخش تکراری حلقه for دارای پیچیدگی $O(n)$ است، زیرا در بدترین حالت، کل نیمه دوم آرایه را جستجو می‌کند. بخش بازگشتی الگوریتم، با تقسیم آرایه به دو نیمه در هر مرحله، دارای پیچیدگی $O(logn)$ است.

بنابراین، با ترکیب این دو بخش، به پیچیدگی کلی $O(n + logn)$ می‌رسیم. با این حال، از آنجا که در اصطلاحات اصولی پیچیدگی زمانی، عبارت دارای بزرگ‌ترین رشد را در نظر می‌گیریم، پیچیدگی کلی الگوریتم را می‌توان O(n) در نظر گرفت.

۷. ۱۰ نمره

تصور کنید سه پایه و تعدادی دیسک با اندازه‌های مختلف داریم که بر اساس اندازه روی یکی از این پایه‌ها قرار گرفته‌اند، به طوری که هیچ دیسک بزرگ‌تری بر روی دیسک کوچک‌تری قرار نگیرد. میخواهیم تمام دیسک‌ها را به پایه دیگر با استفاده از پایه واسطه ببریم با این شرط که فقط مجاز به جابجایی دیسک‌ها بین پایه‌های مجاور هستیم و نمی‌توانیم دیسک‌ها را مستقیماً از پایه اول به پایه سوم منتقل کنیم. همچنین، در هر حرکت تنها می‌توان یک دیسک جابجا کرد و همیشه باید قاعده دیسک کوچکتر روی دیسک بزرگتر را رعایت کرد. یک تابع بازگشتی برای این مسئله نوشته و پیچیدگی زمانی تابع خود را تحلیل کنید. (با نوشتن شبه کد نیز میتوانید آن را تحلیل کنید)

پاسخ:

```
void diskMoving(int n, int from, int with, int to) {
    if(n == 1) {
        cout << ": {from} ----> {with}" << endl;
        cout << ": {with} ----> {to}" << endl;
    } else {
        diskMoving(n-1, from, with, to);
        cout << ": {from} ----> {with}" << endl;
        diskMoving(n-1, to, with, from);
        cout << ": {with} ----> {to}" << endl;
        diskMoving(n-1, from, with, to);
    }
}
```

برای تحلیل پیچیدگی زمانی، ابتدا نحوه فراخوانی توابع را در نظر بگیریم. در هر مرحله، برای یک دیسک با $n > ۱$، تابع 'diskMoving' سه بار با ورودی $n - ۱$ فراخوانی می‌شود. بنابراین، می‌توانیم رابطه بازگشتی زمان اجرای T(n) را به صورت زیر بنویسیم:

$$T(n) = ۳T(n-۱) + O(۱)$$

که در آن $O(۱)$ زمان ثابت مربوط به عملیات‌های چاپ و جابجایی دیسک‌ها است. این رابطه نشان می‌دهد که با هر افزایش واحد در تعداد دیسک‌ها، تعداد فراخوانی‌های تابع سه برابر می‌شود. بنابراین، پیچیدگی زمانی این الگوریتم از نوع اکسپوننشیال است و به صورت دقیق‌تر می‌توان آن را $O(۳^n)$ نشان داد.

$$\begin{aligned}
T(n) &= ۳T(n-۱) \\
&= ۳(۳T(n-۲)) \\
&= ۳^۲T(n-۲) \\
&= ۳^۲(۳T(n-۳)) \\
&= ۳^۳T(n-۳) \\
&\vdots \\
&= ۳^{n-۱}T(۱)
\end{aligned}$$