دانشکده مهندسی کامپیوتر

جزوه درس

ساختمان‌های داده

استاد درس: سید صالح اعتمادی

پاییز ۱۳۹۸

جلسه ۵

مقایسه الگوریتم ها

محمد امین قسوری جهرمی - ۱۳۹۸/۷/۰۶

جزوه جلسه ۵ام مورخ ۱۳۹۸/۷/۰۶ درس ساختمان‌های داده تهیه شده توسط محمد امین قسوری جهرمی.

۱.۵ مقدمه

جلسه قبل در مورد این که الگوریتم چیست و چرا مهم است توضیح دادیم. (مثال فیبوناچی) اگر یادتان باشد دیدیم که که مثال به دست آوردن n امین عدد این دنباله را با دو الگوریتم بدست آوردیم و دیدیم که با الگوریتم بازگشتی از جایی به بعد محاسبات بسیار زیاد میشود و بسیار طول میکشد تا به جواب برسیم. در حالی که با استفاده از الگوریتم دوم قادر بودیم خیلی سریع تر به جواب برسیم.

در این جلسه به بیان دو مطلب می پردازیم. اولین مطلب بیان یک مصداق کمی پیچیده تر است برای اهمیت الگوریتم و وجود الگوریتم های مختلف برای حل یک مسئله و مطلب بعدی هم در مورد روش مقایسه ی دو الگوریتم و بیان معیارمان برای آن ها است که با notation ای به نام big O notation آشنا خواهید شد.

۲.۵ بزرگترین مضرب مشترک دو عدد [۱] دو عدد

مصداق و مثال دوم پیدا کردن GCD یا ب.م.م دو عدد است. GCD یا ب.م.م دو عدد با بزرگترین مقسوم علیه مشترک دو عدد برابر است. برای مثال GCD دو عدد ۴۵ و ۱۵ برابر ۱۵ است چون ۱۵ بزرگترین عددی است که هم بر ۱۵ هم بر ۴۵ بخش پذیر است. برای بدست آوردن GCD دو عدد مثل ۲۴ و ۱۶ یک راه حل ریاضی بدین شکل دارد: ابتدا اعداد را تجزیه میکنیم : ۲۴ = ۲×۲×۲×۳ و ۱۶ = ۲×۲×۲×۲ و سپس از هر کدام از اجزای مشترک آن ها کمترین توان را بر میداریم. در این مثال جز مشترک عدد ۲ است که در یکی توان ۳ و در دیگری توان ۴ دارد . پس توان ۳ انتخاب می شود یعنی ۸ بزرگترین عدد بخش پذیر بر ۲۴ و ۱۶ است و برابر GCD آن دو عدد است.

۳.۵ نوشتن الگوریتم بدیهی برای GCD

خب الان با مفهوم ب.م.م آشنا شدیم ولی برای این که برنامه ای بنویسیم که بتواند ب.م.م حساب کند ما نیاز به الگوریتم داریم نه صرفا تعریف ریاضی. یک الگوریتم بدیهی به شکل زیر است که بگوییم: از عدد ۱ تا عدد کوچکتر بین دو عدد a و b جلو می رویم و برای همه ی آن اعداد محاسبه میکنیم که ایا باقیمانده دو عدد a و b بر آن صفر میشود یا خیر. اگر صفر نشد که ادامه میدهیم و گرنه آن را به عنوان بهترین جواب در آن مرحله در جایی ذخیره میکنیم. بدین ترتیب در آخر عددی که در متغییر می ماند بزرگترین عدد بخش پذیر بر a و b است

[۱] GCD

که همان تعریف GCD است.

```
Data: long a and long b
Result: GCD(a,b)
initialization;
i = 1;
GCD = 1;
while i < Min(a,b) do
    if Max(a,b) % i equals 0 then
        GCD = i;
    else
        Continue;
    end
end
return GCD;
End;
```

Algorithm 1: Simple Solution For GCD

ولی همیشه بعد از نوشتن الگوریتم از خودمان می پرسیم آیا الگوریتمی بهینه تر وجود دارد یا خیر؟ آیا این الگوریتمی که نوشته‌ام اندازه کافی سرعت بالایی دارد یا خیر؟ یکی از اهداف ما در این درس مقایسه الگوریتم ها و پیدا کردن بهترین است.

۴.۵ الگوریتم اقلیدسی GCD

الگوریتمی بهتر برای حل این مسئله وجود دارد. الگوریتم اقلیدس، روشی موسوم به روش نردبانی یا تقسیمات متوالی برای یافتن بزرگترین مقسوم علیه مشترک دو عدد است که در ادامه، با مثالی آن را شرح می‌دهیم. مثال: برای محاسبه GCD(۸۴۶، ۲۰۴) عدد بزرگتر یعنی ۸۴۶ را بر ۲۰۴ تقسیم می‌کنیم و سپس ۲۰۴ را بر باقی ماندهٔ تقسیم قبل تقسیم می‌کنیم و این عمل را تا جایی که باقی مانده صفر شود ادامه می‌دهیم، آخرین باقی‌مانده غیرصفر، بزرگترین مقسوم علیه مشترک دو عدد مزبور است بنابراین GCD(۸۴۶، ۲۰۴) = ۶ نکته: ب.م.م هر عددی با ۰ برابر خود آن عدد میشود.

```
A = KB + X
GCD(A,B)= GCD(A,X) = GCD(B,X)
```

در نتیجه الگوریتم ما به شکل زیر می شود در ابتدا عدد بزرگتر را a و عدد کوچکتر را b می نامیم اگر عدد کوچکتر برابر صفر بود عدد بزرگتر ما برابر است با GCD(a،b) وگرنه مراحل زیر را برایGCD(b، a

برای مثال اگر ب.م.م دو عدد ۱۵ و ۶ را بخواهیم حساب کنیم میگوییم جواب مسئله برابر است با ب.م.م عدد ۶ و باقی مانده ۱۵ بر ۶ که میشود ۳ و در ادامه میگوییم جواب مسئله برابر است با ب.م.م عدد ۳ و باقی مانده صفر که میشود ۳.

```
Data: long a and long b
Result: GCD(a,b)
if a < b then
    Swap(a,b);
end
if b == 0 then
    return a;
else
    return GCD(b,a%b);
end
```

Algorithm 2: Fast Algorithm For GCD

این الگوریتم از الگوریتم قبلی ما سریع تر است چون اعداد به سرعت ریز می شوند و محاسبه راحت تر می گردد پس انتخاب الگوریتم بهینه بسیار حائز اهمیت است.

۵.۵ مقدمه ای بر مقایسه الگوریتم ها

دومین مطلب مربوط به مقایسه الگوریتم ها است. سرعت اجرای برنامه ها به چه چیزی وابسته هست. عوامل موثر در سرعت اجرای برنامه قدرت پردازنده یا سرعت دسترسی به حافظه یا ... است. پس زمان اجرای برنامه درکامپیوتر های مختلف فرق دارد پس نمی‌توانیم بر اساس زمان اجرای الگوریتم بر روی کامپیوترها آن ها را مقایسه کنیم و نیاز به معیار بهتری دارد. شاید بر روی کامپیوتری که قدرت بیشتری دارد سریعتر اجرا شود در حالیکه درکامپیوتری با پردازنده ضعیف تر زمان اجرای برنامه بیشتر است الگوریتم‌های مختلفی برای حل یک مسئله ممکن است طراحی شده باشند. برای انتخاب بهترین الگوریتم باید معیاری جهت مقایسه کارایی الگوریتم‌ها داشته باشیم. آنالیز کارایی یک تخمین اولیه‌است با دو معیار سنجیده می شود :

- پیچیدگی حافظه [۲]
- پیچیدگی زمانی [۳]

---

[۲] complexity space
[۳] complexity time

۶.۵ پیچیدگی زمانی

زمان اجرای یک برنامه به موارد زیر بستگی دارد:

- سخت‌افزار
- سیستم‌عامل
- کمپایلر
- نوع الگوریتم
- آرایش داده‌های ورودی

زمان اجرای برنامه‌ها به صورت رابطه بین بزرگی سایز ورودی و زمان مورد نیاز برای پردازش ورودی است. زمان اجرا یکی از ملاک‌های مقایسه چند الگوریتم برای حل یک مسئله می‌باشد. منظور از واحد زمانی، واحدهای زمانی واقعی مانند میکرو یا نانو ثانیه نمی‌باشد بلکه منظور واحدهای منطقی است که رابطه بین بزرگی (n) یک فایل یا یک آرایه و زمان مورد نیاز برای پردازش داده‌ها را شرح می‌دهد. (توجه کنید که هر دستور یک واحد زمانی اشغال می‌کند)

مثلاً دستورهای a=b c/d=e؛ هر کدام یک واحد زمانی را دربردارند. بنابراین تعداد مراحل برای هر عبارت یک برنامه بستگی به؛ نوع عبارت دارد، بطوریکه در عبارات توضیحی برابر صفر و در دستور انتسابی بدون فراخوانی برابر یک می‌باشد؛ و در دستورهای غیربازگشتی حلقه for، while، repeat until به تعداد تکرار حلقه در نظر گرفته می‌شود. هدف از محاسبه پیچیدگی زمانی یک الگوریتم این است که بفهمیم نیازمندی یک الگوریتم به زمان با چه تابعی رشد می‌کند و هدف اصلی بدست آوردن این تابع رشد است. برای مثال هرچه زبان برنامه‌نویسی به زبان ماشین نزدیک تر باشد، برنامه با سرعت بیشتری به جواب خواهد رسید زمان اجرا مقدار زمانی از کامپیوتر است که برنامه برای اجرای کامل مصرف می‌کند. برای محاسبه پیچیدگی زمان الگوریتم ابتدا تعداد قدم‌های الگوریتم به صورت تابعی از اندازه مسئله مشخص می‌شود، برای انجام این کار تعداد تکرار عملیات اصلی الگوریتم محاسبه می‌شود و به صورت تابع f بیان می‌شود. سپس تابع g، که مرتبه بزرگی تابع f را وقتی اندازه ورودی به اندازه کافی بزرگ است نشان می‌دهد، بدست می‌آید. در نهایت پیچیدگی الگوریتم برای نشان دادن رفتار الگوریتم با ورودی‌های مختلف با استفاده از نمادها O، ? و Ω که در بخش بعدی با آن‌ها آشنا می‌شویم، بیان می‌شود.

برای مثال:

```
int func(int n)
int i;
int sum=0;
for (i=1;i<=n;i++) sum=sum+i;
return sum;
```

عبارت مساوی $T(N) = 2n + 3$ می‌شود. همان‌طور که مشاهده می‌کنید زمان اجرای هر عبارت جایگزینی یا محاسباتی را مساوی ۱ واحد زمانی فرض می‌کنیم. هم چنین دستور داخل حلقه n بار انجام می‌شود ولی آزمایش کردن شرط حلقه در خط for به تعداد n+1 بار صورت می‌گیرد. دستور Return نیز مساوی یک واحد زمانی است.

۷.۵ Ω and Big-O

برای نمایش پیچیدگی الگوریتم‌ها از تعاریف زیر استفاده می‌شود:

Big-O (حدبالا) تابع f(n) را برای $n\geq 0$ در نظر بگیرید. می‌گوئیم $O(g(n) = f(n)))$ است اگر ثابت مثبت و حقیقی c و عدد صحیح و غیر منفی N وجود داشته باشند به طوریکه به ازای تمام مقادیر $n\geq N$: $f(n)\leq cg(n)$ برقرار باشد. این نماد حد بالائی برای تابع f(n) می‌دهد و وقتی بکار می‌رود که رفتار الگوریتم بدترین حالت و بیشترین زمان اجرا را برای مقادیر معین ورودی دارد. برای مثال داریم: $F(n) = n^2 = .O(n^3)$ به ازای N=۱ همیشه رابطه $n^2 < cn^3$ برقرار است یا $F(n) = n^2 + 3n + 4 = .O(n^2)$ در نتیجه داریم> $8n^2 + n^2 + 3n^2 + 4n^2 < n^2 + 3n + 4$ و به ازای $N > 1$ و $C = 8$ در نتیجه از $O(n^2)$ است. در نتیجه برای پیدا کردن اوردر یک عبارت بزرگترین جمله را از لحاظ رشد در نظر میگیریم. در مثال ۳-۱ که یک کد ساده را بررسی کردیم عبارت مساوی $T(N) = 2n + 3$ شد و میتوانیم بگوییم که از O(n) است. توجه داشته باشید که مثلا n^2 هم $O(n^3)$ هم $O(nn)$ یا $O(n^2)$ ... است.

امگا/Ω (حدپائین) بر عکس big O notation تابع f(n) را برای $n\geq 0$ در نظر بگیرید. می‌گوئیم $(f(n)) = \Omega(g(n))$ اگر ثابت مثبت و حقیقی c و عدد صحیح و غیر منفی N وجود داشته باشند به طوریکه به ازای تمام مقادیر $n\geq N$: $(f(n)) \geq c(g(n))$ برقرار باشد. این نماد حد پائینی برای تابع f(n) می‌دهد و وقتی بکار می‌رود که رفتار الگوریتم بهترین حالت و کمترین زمان اجرا را برای مقادیر معین ورودی دارد.

برای مثال داریم: $F(n) = n^2 = \Omega.(n)$ به ازای N=۱ همیشه رابطه $cn < n^2$ برقرار است. توجه داشته باشید که مثلا n^2 هم Ω (۱) یا Ω (n) ... است.

۸.۵ سرعت رشد توابع

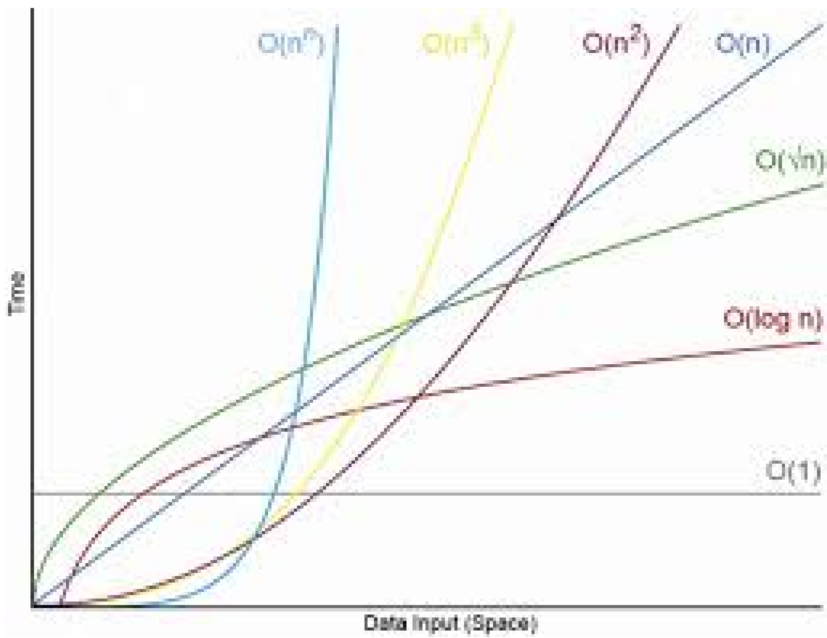
همانطور که گفتیم برای بدست آوردن اوردر یک عبارت باید بزرگترین عبارت را از لحاظ رشد پیدا کنیم یعنی زمانی که ورودی ما بزرگ می شود کدام توابع سریعتر و کدام کند تر پیش میروند: رشد توابع بصورت زیر است:

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

شکل ۱.۵: جدول رشد توابع

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n$$

شکل ۲.۵: سرعت رشد توابع



شکل ۳.۵: نمودار سرعت رشد توابع

Bibliography