فایل دوم درس ساختمان های داده

دانشجویان عزیز کلاس درس ساختمان های داده

- با آرزوی سلامتی برای تمامی شما، در فایل ارسالی سعی گردیده تمامی مطالب، جهت مطالعه دقیق و انجام تمرین با جزییات بصورت تفصیلی و ارایه مثال مطرح گردد.

- درس ساختمان های داده را با مبحث الگوریتم آغاز نموده و به بررسی معیار های ارزیابی الگوریتم ها پرداختیم. مبحث محاسبه تابع پیچیدگی زمانی T(n) تا بررسی الگوریتم های شامل حلقه for و مثال های مرتبط در کلاس بطور کامل بررسی شد. در فایل ارسالی هفته گذشته، تمامی مباحث مورد نیاز برای فهم بیشتر مطالب درس ساختمان های داده مشخص گردیده و امیدوارم مطالعه کافی صورت گرفته باشد.

موفق و سلامت باشید-اسبقی

تابع پیچیدگی زمانی T(n):

طبق مباحث قبلی، برای محاسبه زمان و پیچیدگی زمانی الگوریتم ها، نیاز به شمارش دستور العمل های الگوریتم البته دستوراتی با بیشترین تعداد تکرار وجود دارد. بر همین اساس و در ادامه درس، تابع پیچیدگی زمانی برای ساختارهای تکرار در الگوریتم ها محاسبه می گردد.

ساختار های تکرار: - حلقه for (قبلا بررسی شده است.)

- حلقه while/do while
- الگوریتم های بازگشتی

حلقه while /do while:

حلقه while را با ذکر مثال بررسی نموده تا نحوه محاسبه تابع پیچیدگی زمانی مشخص گردد:

تابعی طراحی نمایید که یک آرایه ای از اعداد صحیح و یک عدد صحیح را بعنوان ورودی گرفته و بررسی نماید که عدد مذکور در آرایه وجود دارد یا خیر؟

```
int  function(int   A[n] ,int   x)
{ int  i=0;
    while( i<n && A[i]≠ x)
       i++;
   if( i >= n)
     return  -1;
 else
    return   i;}
```

برای محاسبه تابع پیچیدگی زمانی در الگوریتم هایی که شامل حلقه while می باشند، نمیتوان مانند حلقه for یک راهکار ثابت را بکار برد زیرا در ساختار حلقه while ، تعداد تکرار حلقه کاملا متاثر از شرط حلقه بوده و برای آن لازم است دو حالت بهترین و بدترین بسته به شرط حلقه محاسبه گردد.

در این الگوریتم اگر بعنوان مثال آرایه A و عدد مورد جستجوی x بصورت حالت اول باشد:

A	50	30	10	20	70	60	100	90

بعنوان مثال اگر عدد x=50 باشد، حلقه while در اولین تکرار x را یافته و متوقف می گردد. پس فقط یک دور تکرار خواهد داشت.(بهترین حالت)

ولی بعنوان مثال اگر x=80 یا x=90 باشد، حلقه while بطور کامل اجرا شده و حداکثر دور تکرار را خواهد داشت.(بدترین حالت)

$B(n) = 1$ (Best case)

$W(n) = n$ (worst case)

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

البته در مورد حلقه while، همیشه محاسبه تابع پیچیدگی به این سادگی نبوده و نیاز به درنظرگرفتن قوانین احتمال وقوع هر کدام از حالتهای بهترین و بدترین وجود دارد.

الگوریتم های بازگشتی(Recursive Algorithms):

برای حل یک مساله بصورت بازگشتی بایستی بصورت زیر عمل نمود، هر الگوریتم بازگشتی دارای

1) شرط توقف

2)فراخوانی های بازگشتی

می باشد. در واقع هر تابع یا زیر برنامه ای که بتواند داخل بدنه اش، خودش را دوباره فراخوانی نماید را تابع یا زیر برنامه بازگشتی گویند.

مثال: برای محاسبه فاکتوریل یک عدد صحیح دو راهکار بازگشتی و غیر بازگشتی وجود دارد:

نحوه حل مساله فاکتوریل بصورت غیربازگشتی ⟸ 4!= 4 * 3 * 2 * 1 = 24

نحوه حل مساله فاکتوریل بصورت بازگشتی ⟸

4!= 4 * 3!

3!= 3 * 2!

2!= 2 * 1!

1!= 1 * 0!

0!= 1

همانطور که مشخص است در حل مساله فاکتوریل بصورت بازگشتی، در هر مرحله پاسخ، مجددا صورت مساله که فاکتوریل است، تکرار شده است. صورت مساله 4! بوده که در مرحله اول با 4*3! پاسخ داده شده است، که به آن فراخوانی های بازگشتی گویند. این فراخوانی های بازگشتی در هر مرحله تکرار می شود ولی بدیهی است که بایستی در جایی این فراخوانی ها متوقف گردد،

که آن را شرط توقف نامند. در مثال فاکتوریل 1 = !1 یا 1= !0 شرط توقف می باشد. برای روشن تر شدن مطلب، توابع محاسبه فاکتوریل یک عدد صحیح در دو حالت بازگشتی و غیر بازگشتی بررسی می گردد.

تابع محاسبه فاکتوریل بصورت غیر بازگشتی

```
int  fact( int  n){
    int  i , f=1;
    for(i=1; i<=n; i++)
      f*= i;
return  f;}
```

تابع محاسبه فاکتوریل بصورت بازگشتی

```
int  fact(int  n){
  if(n==0 || n==1)
      return  1;
return (n * fact(n-1));}
```

در الگوریتم بازگشتی ابتدا شرط توقف بررسی می شود که اگر مقدار n برابر 0 یا 1 باشد که مقدار 1 بازگردانده شده و فراخوانی ها متوقف می گردد، ولی در صورتیکه مقدار عدد n بیشتر از 1 باشد که فراخوانی های بازگشتی به صورت n*fact(n-1) ادامه می یابد تا جاییکه مقدار n به 1 تقلیل یافته و الگوریتم خاتمه یابد.

برای محاسبه تابع پیچیدگی زمانی برای الگوریتم های بازگشتی، تابع پیچیدگی نیز بایستی بصورت بازگشتی دنبال گردد.

$$T(n) = \begin{cases} c & n<=1 \\ T(n-1) + d & n>1 \end{cases}$$

مقدار c برای نمایش مقادیر ثابت در نظر گرفته می شود و مقدار d در این تابع پیچیدگی، به معنای زمان اتلافی فراخوانی بازگشتی می باشد. در واقع برای فهم بیشتر مطلب می توان چنین بیان نمود که اگر نیاز به محاسبه تابع پیچیدگی برای مقدار n وجود دارد فقط کافیست تابع پیچیدگی برای n-1 محاسبه شده و با مقدار d جمع گردد. این روند تا زمانیکه مقدار n به 1 برسد ادامه یافته و سپس در n=1 متوقف می گردد.

جواب نهایی تابع پیچیدگی بایستی فقط براساس n بدست آید، درحالیکه پاسخ تابع پیچیدگی فاکتوریل بازگشتی بصورت T(n-1) + d محاسبه شده است که خود نیاز به حل دارد. برای رسیدن به نتیجه نهایی، از روش جایگذاری های مکرر استفاده می شود.

T(n) = T(n-1) + d

T(n-1) = T(n-2)+ d

⟹ T(n) = T(n-2)+ 2d

T(n-2) = T(n-3) + d

⟹ T(n) = T(n-3) + 3d

T(n-3) = T(n-4) + d

⟹ T(n) = T(n-4) + 4d

.

.

$\Longrightarrow T(n) = T(n-10) + 10d$

.

.

$T(n) = T(n-(n-1)) + (n-1)d \longrightarrow T(n) = T(1) + (n-1)d \longrightarrow \boxed{T(n)= c + nd - d}$

در این حالت با استفاده از روش جایگذاری، در هر مرحله مقدار n را یک واحد کاهش میدهیم تا در نهایت به کشف الگوی مربوطه می رسیم. برای رسیدن به مرحله نهایی نیز باید ؟ را در تابع پیچیدگی(T(n-? تا جایی کم کنیم که مقدار ورودی تابع پیچیدگی به 1 برسد ، تا از T(1) = c استفاده نموده و c را جایگزین آن نمود.

مثالی دیگر برای محاسبه تابع پیچیدگی زمانی برای الگوریتم های بازگشتی:

$$T(n) = \begin{cases} c & n<=1 \\ T(n-1) + n + d & \end{cases}$$

حل:

$\underline{T(n) = T(n-1) + n + d}$

$T(n-1) = T(n-2)+ (n-1) + d$

$\Longrightarrow T(n) = T(n-2)+ n + n-1 + 2d$

$T(n-2) = T(n-3) + (n-2) + d$

$\Longrightarrow T(n) = T(n-3) + n + (n-1) + (n-2) + 3d$

$T(n-3) = T(n-4) + (n-3) + d$

⟹ T(n) = T(n-4) + n + (n-1) + (n-2) + (n-3) + 4d

.

.

⟹ T(n) = T(n-10) + n + (n-1) + (n-2) + ……..+ (n-9) + 10d

.

.

⟹ T(n) = T(n-(n-1)) + n + (n-1) + (n-2) + ……..+ (n-(n-2)) + (n-1)d

⟹ T(n) = T(1) + n + (n-1) + (n-2) + ……..(2) + (n-1)d

n + (n-1) + (n-2) + ……..+ 2 یک سری می باشد که می توان با یکی از روشهای زیر محاسبه نمود.

1)$\sum_{i=2}^{n}(\mathrm{i})$

2) $\sum_{i=0}^{n-2}(\mathrm{n}-\mathrm{i})$

در روش 1، داریم:

(2 / جمله آخر + جمله اول) * (1+ حدپایین – حدبالا)

(n – 2 +1) * ((2 + n) /2) ⟹ (n – 1)(n+2)/2

در روش 2 نیز به همان پاسخ می رسیم:

(n – 2 – 0 + 1) * ((n + 2)/2) ⟹ (n – 1)(n + 2) /2

جواب نهایی تابع پیچیدگی T(n) را براین اساس بدست می آوریم:

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

پس از بررسی سه ساختار تکرار در محاسبه تابع پیچیدگی الگوریتم ها، حال این سوال مطرح می شود که بعنوان مثال از بین توابع پیچیدگی بدست آمده زیر، کدام بهینه تر بوده و زمان بهتری را نتیجه می دهد؟

$T(n) = n^2 + n$

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

$T(n) = n^2 + n/4$

.

برای پاسخ به این سوال، باید اصطلاحاتی در محاسبه مرتبه زمانی الگوریتم ها و نشانه گذاری های زمانی را بدانیم.

نشانه گذاری O بزرگ:

تعریف: برای تابع پیچیدگی f(n)، O(f(n)) مجموعه تمام توابع پیچیدگی g(n) است که برای آنها یک عدد ثابت حقیقی مثبت c و یک عدد صحیح غیر منفی N وجود دارد به قسمی که :

$g(n) <= c * f(n) \qquad n >= N$

در اینحالت گوییم، g(n) عضو O بزرگ f(n) است و به اینصورت نمایش داده می شود:

$g(n) \in O(f(n))$

مثال:

n^2 +10 n <= 2 n^2 n>=10

درستی این رابطه همواره برای n>=10 برقرار است، بطوریکه اگر n^2 +10 n را معادل g(n)، بگیریم، عدد 2 معادل ثابت c بوده و n^2 نیز معادل f(n) خواهد بود پس گوییم:

n^2 +10 n € O(n^2)

به همین ترتیب داریم:

n^2 +10 n <= 3 n^3 n>=2

پس نتیجه می گیریم:

n^2 +10 n € O(n^3)

و همچنان می توان برای n های با توان بیشتر، این مفهوم را ادامه داد:

n^2 +10 n <= n^4 n>=3

نتیجه میگیریم:

n^2 +10 n € O(n^4)

نشانه گذاری O بزرگ در واقع بع معنای کران های بالا برای تابع پیچیدگی می باشد.

نشانه گذاری Ω بزرگ:

تعریف: برای تابع پیچیدگی f(n)، Ω(f(n)) مجموعه تمام توابع پیچیدگی g(n) است که برای آنها یک عدد ثابت حقیقی مثبت c و یک عدد صحیح غیر منفی N وجود دارد به قسمی که :

g(n)>= c* f(n) n>=N

در اینحالت گوییم، g(n) عضو Ω بزرگ f(n) است و به اینصورت نمایش داده می شود:

$g(n) \in \Omega(f(n))$

مثال:

$n^2 + 10n >= n^2 \qquad n>=0$

درستی این رابطه همواره برای n>=0 برقرار است، بطوریکه اگر $n^2 + 10n$ را معادل g(n)، بگیریم، عدد 1 معادل ثابت c بوده و n^2 نیز معادل f(n) خواهد بود پس گوییم:

$n^2 + 10n \in \Omega(n^2)$

به همین ترتیب داریم:

$n^2 + 10n >= n \quad n>=0$

پس نتیجه می گیریم:

$n^2 + 10n \in \Omega(n)$

نشانه گذاری Ω به مفهوم کران های پایین برای تابع پیچیدگی g(n) می باشد.

مرتبه زمانی Θ:

مرتبه زمانی Θ، اشتراک میان دو نشانه O بزرگ و Ω می باشد بدین معنی که بزرگترین کران پایین و کوچکترین کران بالا را برای یک تابع پیچیدگی زمانی، مرتبه زمانی آن تابع گویند.

$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

$C * f(n) <= g(n) <= d * f(n)$

در اینصورت گوییم:

$g(n) \in \Theta(f(n))$

پس در نتیجه برای مثال های قبل داریم:

$n^2 + 10n \in \Theta(n^2)$

بدین معنی که $n^2 + 10n$ از درجه n^2 می باشد. و همینطور برای سایر مثال های زیر نیز داریم:

$T(n) = n^2 + n \in \Theta(n^2)$

$T(n) = n(n+1))/2 \in \Theta(n^2)$

$T(n) = n^2 + n/4 \in \Theta(n^2)$

درنهایت می توان نتیجه گرفت که تمامی توتبع پیچیدگی بالا، همه از درجه n^2 می باشند.

چند نکته در مورد مرتبه زمانی:

- اگر داشته باشیم $g(n) \in O(f(n))$ پس نتیجه می گیریم: $f(n) \in \Omega(g(n))$
- اگر داشته باشیم $g(n) \in \Theta(f(n))$ پس نتیجه می گیریم: $f(n) \in \Theta(g(n))$
- برای تمامی توابع لگاریتمی داریم:

$\log_a(n) \in \Theta(\log_b(n))$ a>1,b>1

در کل اولویت زمانی مقادیر توابع پیچیدگی بصورت زیر است:

$$C < \log n < n < n^2 < n^3 < \ldots\ldots < a^n < n!$$

الگوریتم هایی با پیچیدگی زمانی n را الگوریتم های زمان خطی می گویند زیرا پیچیدگی زمانی آنها با اندازه ورودی n رابطه خطی دارد، ولی الگوریتم هایی با پیچیدگی زمانی n^2 را الگوریتم های زمانی درجه دوم گویند، زیرا پیچیدگی زمانی آنها نسبت به اندازه ورودی n یک تابع درجه دوم است. هر الگوریتم زمانی خطی در نهایت کارایی بیشتری از الگوریتم های زمانی درجه دوم داشته و به همین ترتیب.

مثال: مقادیر توابع پیچیدگی زیر را مرتب کنید.

الف) $2n^2$ ، $\log n$، 2^n، $n^2 \log n$

پاسخ: $\log n < 2n^2 < n^2 \log n < 2^n$

ب) n^2 ، $\log n$، 100^n، $n^{100} \log n$ ، 100n

پاسخ: $\log n < 100n < n^2 < n^{100} \log n < 100^n$