به نظر شما هدف از ارائه ی درس طراحی الگوریتم ها در مقطع کارشناسی فناوری اطلاعات چیست ؟

برای اینکه بتوانیم مرتبه ی زمانی الگوریتم هایمان را محاسبه کنیم و سعی کنیم الگوریتم هایی بنویسیم که هزینه ی زمانی کمتری دارند .

در تولید یک محصول نرم افزاری محاسبه ی پیچیدگی زمانی و فضایی یکی از مهمترین متریک های محاسبه شده توسط مهندسین نرم افزار است .
الف) به نظر شما چرا پیچیدگی الگوریتم و نرم افزار را محاسبه می کنند ؟

تا قبل از اجرای عملی بر روی سخت افزار بتوانند تحلیل کنند که آیا این الگوریتم در مقادیر داده های بزرگ بر روی سخت افزارمان توان اجرا دارد یا خیر .

ب) منظور از $f(n) = O(g(n))$ و $f(n) = \Omega(g(n))$ و $f(n) = \theta(g(n))$ چیست ؟

big-O Notation
عبارت $g(n) \in O(f(n))$ یعنی برای تابع پیچیدگی مفروض $f(n)$ ، $O(f(n))$ به مجموعه ای از توابع اشاره دارد که برای آنها ثابت های c و n_0 وجود دارند ، بطوریکه برای همه ی $n \geq n_0$ داریم :

$$g(n) \leq cf(n)$$

big-Ω Notation
عبارت $g(n) \in \Omega(f(n))$ یعنی برای تابع پیچیدگی مفروض $f(n)$ ، $\Omega(f(n))$ به مجموعه ای از توابع اشاره دارد که برای آنها ثابت های c و n_0 وجود دارند ، بطوریکه برای همه ی $n \geq n_0$ داریم :

$$g(n) \geq cf(n)$$

θ Notation

عبارت $g(n) \in \theta(f(n))$ یعنی :

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

و

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

ج) کدامیک از موارد زیر می تواند درست باشد .

$$\left.\begin{array}{l} f(n) = O(g(n)) \\ \\ f(n) = \theta(g(n)) \end{array}\right\} \Rightarrow f(n) = \Omega(g(n)) \qquad \checkmark$$

$$\left.\begin{array}{l} f(n) = \Omega(g(n)) \\ \\ f(n) = \theta(g(n)) \end{array}\right\} \Rightarrow f(n) = \Omega(g(n)) \qquad \checkmark$$

$$\left.\begin{array}{l} f(n) = \theta(h(n)) \\ \\ g(n) = \Omega(f(n)) \end{array}\right\} \Rightarrow g(n) = \Omega(h(n)) \qquad \checkmark$$

$$\left.\begin{array}{l} f(n) = \Omega(g(n)) \\ \\ f(n) = O(g(n)) \end{array}\right\} \Rightarrow f(n) = \theta(g(n)) \qquad \checkmark$$

$$\left.\begin{array}{l} f(n) = \theta(g(n)) \\ \\ g(n) = \Omega(f(n)) \end{array}\right\} \Rightarrow f(n) = \theta(g(n)) \qquad \checkmark$$

تفاوت میان الگوریتم های از نوع **تقسیم و غلبه** و **حریصانه** چیست ؟

تقسیم و غلبه

تقسیم مسئله به یک یا چند نمونه کوچکتر و سپس حل نمونه های کوچکتر و سپس در صورت نیاز ترکیب راه حل ها برای حل کل مسئله

حریصانه

الگوریتم حریصانه با انجام یک سری انتخاب ، که در جای خود بهینه است ، عمل کرده به امید اینکه یک حل بهینه کلی یافت شود .

در روش حریصانه ، تقسیم به نمونه های کوچکتر صورت نمی پذیرد .

الگوریتم حریصانه ، کار را با یک مجموعه ی تهی آغاز کرده و به ترتیب بهینه ترین عناصر را به مجموعه اضافه می کند تا به راه حل نهایی دست یابد .

اگر الگوریتمی روی سیستمی با اندازه ورودی ۱۰ به مدت ۸ میلی ثانیه اجرا شود ، همین الگوریتم با اندازه ورودی ۱۰۰ روی سیستمی دیگر با قدرت پردازشی و سرعت اجرایی نصف سیستم موجود در چه زمانی اجرا خواهد شد (پیچیدگی این الگوریتم از مرتبه ی $n \log(n)$ می باشد)

$$10 \log 10 \rightarrow 8ms$$

وقتی قدرت پردازنده نصف می شود پس زمان پردازش ۲ برابر می شود بنابراین داریم :

$$10 \log 10 \rightarrow 16ms$$

وقتی ورودی را ۱۰۰ می دهیم داریم :

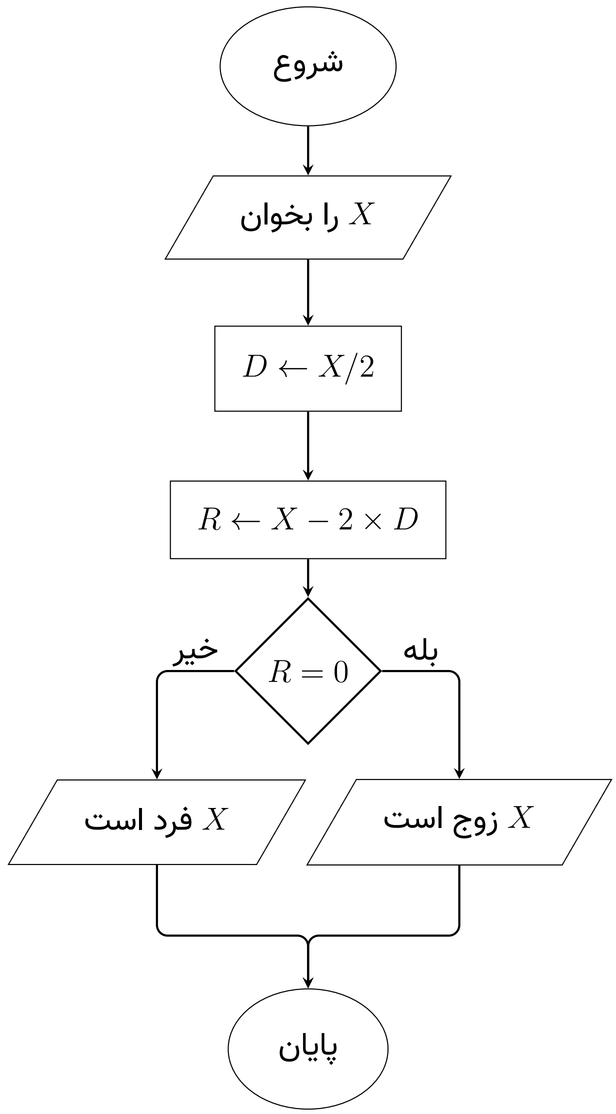
$$\begin{aligned} 100 \log 100 &= 100 \log 10^2 \\ &= 2 \times 100 \log 10 \\ &= 200 \log 10 \\ &= 20 \times 10 \log 10 \\ &= 20 \times \underbrace{10 \log 10}_{\text{16 ms}} \\ &= 320ms \end{aligned}$$

الگوریتم جستجوی دودویی را بیان کنید و مشخص کنید از کدام دسته از الگوریتم هاست و پیچیدگی آن چیست ؟

الگوریتم جستجوی دودویی از دسته ی الگوریتم های تقسیم و غلبه است و مرتبه ی اجرایی آن $O(\log(n))$ می باشد .

```
int BinarySearch(A,n,key) {
   l = 1;
   h = n;
   while(l <= h) {
      mid = (l+h)/2;
      if(key == A[mid]) {
         return mid;
      }
      if(key < A[mid]) {
         h = mid - 1;
      } else {
         l = mid + 1;
      }
   }
   return -1;
}
```

فلوچارتی را رسم کنید که عددی را از ورودی دریافت کند و مشخص کند که زوج است یا خیر ؟



پیچیدگی موارد زیر را مشخص کنید ؟

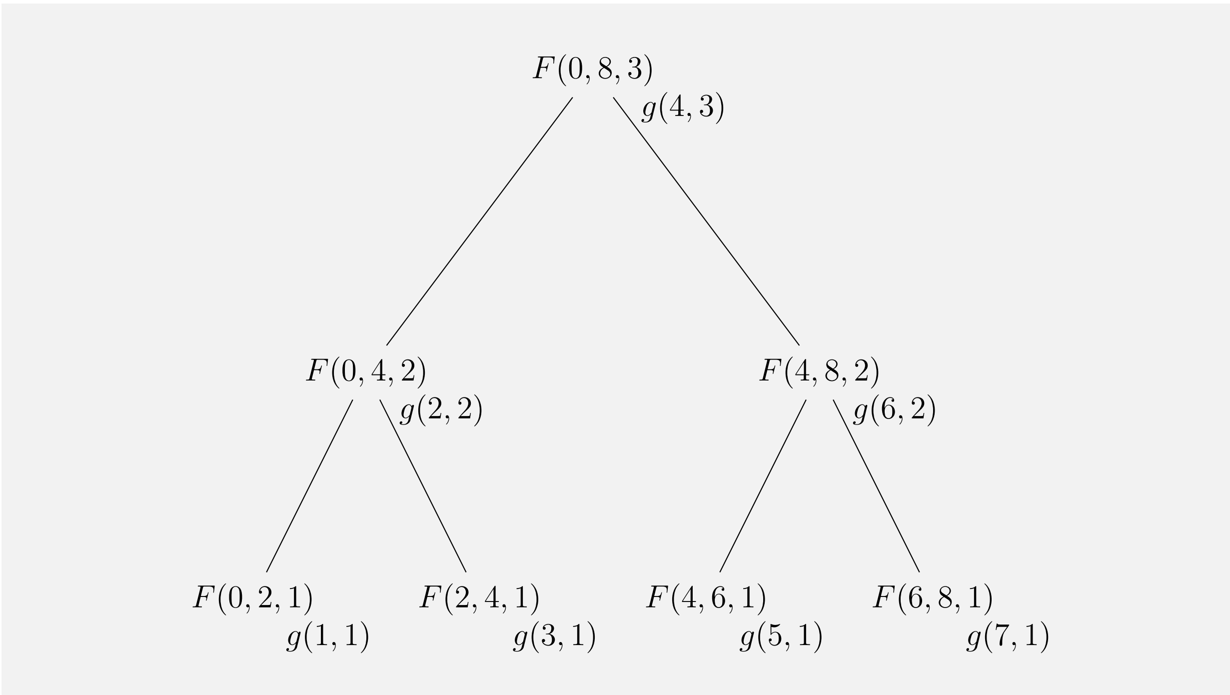
```
z = 0;                                       .
for(i=1;i<=n;i++) {                          n
   for(j=1;j<=n;j+=2) {                      n * n/2
      for(k=1;k<=n^2;n*=3) {                 n * n/2 * log(n)
         z++;                                n * n/2 * log(n)
      }                                      .
   }                                         .
}                                            .
```

$$O(n^2 \log{(n)})$$

```
F(n) {                                                      .
   if( n <= 1 ) {                                           .
      return 1;                                             .
   } else {                                                 .
      return F(n/2) + F(n/3) + F(n-1) + F(n-2);             .
   }                                                        .
}                                                           .
```

$$\left.\begin{array}{ll} F(n/2) & \to \log_2^n \\ F(n/3) & \to \log_3^n \\ F(n-1) & \to n \\ F(n-2) & \to \frac{n}{2} \end{array}\right\} \Rightarrow O(n)$$

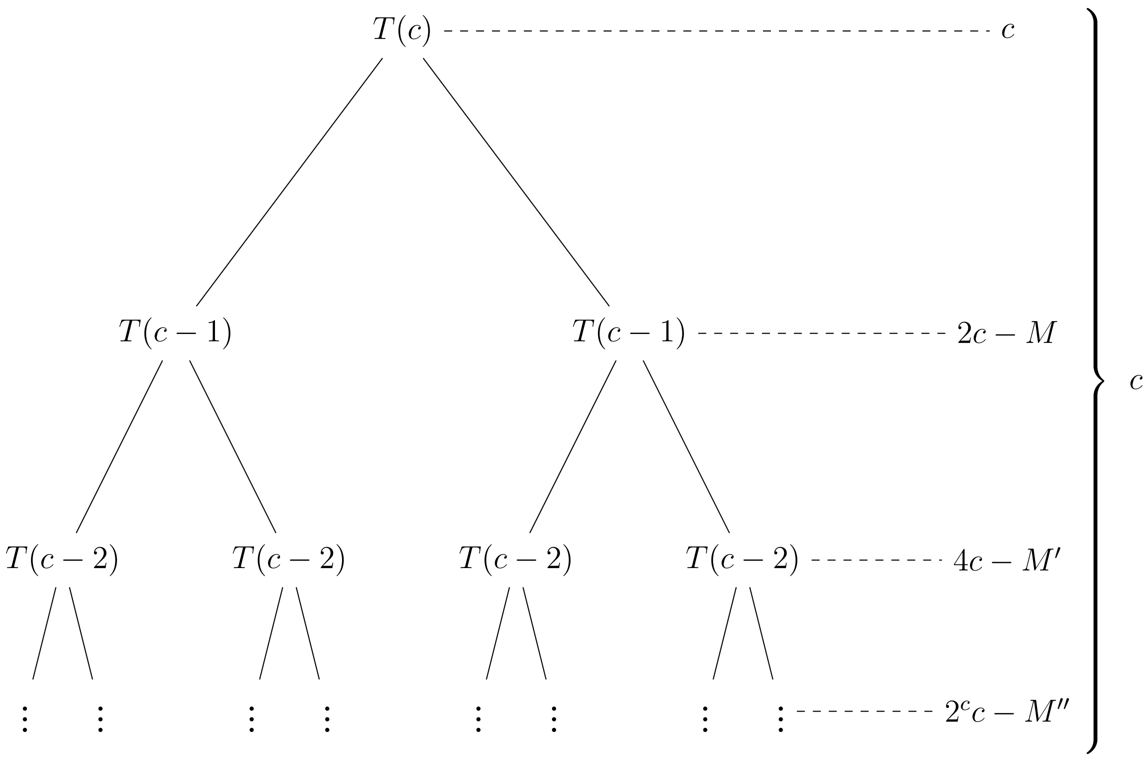
فرض کنید تابع $g(x,y)$ به این صورت تعریف شده باشد که در مکان x به تعداد y علامت $*$ را چاپ کند .
در این صورت :
خروجی الگوریتم `F_Print(0,8,3)` چیست ؟



پیچیدگی زمانی این الگوریتم را محاسبه کنید ؟

```
F_Print(int a,int b,int c) {                    T(c)
   int m = (a+b)/2 ;                            .
   if(c > 0) {                                  .
      g(m,c);                                   c
      F_Print(a,m,c-1);                         T(c-1)
      F_Print(m,b,c-1);                         T(c-1)
   }                                            .
}                                               .
```

$$T(c) = 2\,T(c-1) + c$$



$$\begin{aligned}
c + 2c + 4c + \cdots + 2^c c - M &= c[1 + 2 + 4 + \cdots + 2^c] \\
&= c[1 + 2 + 2^2 + 2^3 + \cdots + 2^c] \\
&= c[2^{c+1} - 1] \\
&= c2^{c+1} - c \\
&\Rightarrow \boxed{f(n) = O(c2^c)}
\end{aligned}$$

$$T(c) = 2\,T(c-1) + c$$

$$T(c-1) = 2T(c-2) + c - 1$$

substitute

$$\Rightarrow T(c) = 2[2T(c-2) + c - 1] + c$$

$$\Rightarrow T(c) = 2^2T(c-2) + 2c - 2 + c$$

$$T(c-2) = 2T(c-3) + c - 2$$

$$\Rightarrow T(c) = 2^2[2T(c-3) + c - 2] + 2c - 2 + c$$

$$\Rightarrow T(c) = 2^3T(c-3) + 2^2c - 2^3 + 2c - 2 + c$$

$$\Rightarrow T(c) = 2^3T(c-3) + 2^2c + 2c + c - 2^3 - 2$$

continue for k times

$$\Rightarrow T(c) = 2^kT(c-k) + \underbrace{2^{k-1}c + \cdots + 2^2c + 2c + c}_{\text{k times}} + M$$

$$\Rightarrow T(c) = 2^kT(c-k) + c[\underbrace{2^{k-1} + \cdots + 2^2 + 2 + 1}_{\text{k times}}] + M$$

$$\Rightarrow T(c) = 2^kT(c-k) + c[2^k - 1]$$

$$\Rightarrow T(c) = 2^kT(c-k) + c2^k - c$$

$$c - k = 0$$

$$T(c) = 2^cT(c-c) + c2^c - c$$

$$T(c) = 2^cT(0) + c2^c - c$$

$$T(c) = 2^c + c2^c - c$$

$$\Rightarrow f(n) = O(c2^c)$$