جلسه اول

- معارفه
- بارم بندی
- کلیات ساختمان داده و الگوریتم ها
- منابع و ماخذ
- الگوریتم مرتب سازی درجی ، ایده‌ی آن حل مثال و نمایش کد
    - الگوریتم مرتب سازی حبابی، سریع و ...

جلسه دوم.

- تکمیل مرتب سازی درجی (نمایش الگوریتم) + تحلیل الگوریتم
- نمادهای مجانبی (بخصوص big O)
- مرتب سازی ادغامی و تحلیل آن
- تحلیل unfolding
- تحلیل بصری یا درختی

نماد مجانبی

یادآوری. در جلسه‌ی اول insersion sort تدریس شد. کلیت کار و یک مثال از آن حل شد.

مساله‌ی زیر را در نظر بگیرید.

مساله. می‌خواهیم در یک آرایه دومین کوچک‌ترین عنصر رو پیدا کنیم. دو روش را می‌توان حداقل در نظر داشت.

الف. آرایه را مرتب کنیم. و دومین عنصر آرایه را به عنوان نتیجه باز گردانیم.

ب. تابع custom دقیقا برای این مساله بنویسیم. استفاده از دو متغییر min و second min هر وقت عددی یافته شد که از minimum کوچکتر بود آن را به second min انتقال بده.

الگوریتم الف یا با کمک مرتب سازی به شکل زیر خواهد بود.

```
def find_second_smallest (data):
  if len(data)< 2:
    return None

  sortedData = sorted(data)
  return sortedData[1]

# Example usage
my_list = [5, 3, 8, 1, 2]
second_smallest = find_second_smallest(my_list)

if second_smallest is not None:
  print(f"Second smallest item: {second_smallest}")
else:
  print("List has less than 2 elements")
```

و الگوریتم دوم کاستوم به شکل زیر خواهد بود

```
def find_second_smallest(data):
  """
  Finds the second smallest item in a list in Python.

  Args:
      data: A list of numbers.

  Returns:
      The second smallest item in the list, or None if the list has less
than 2 elements.
  """

  if len(data) < 2:
    return None

  # Initialize first and second smallest values
  first_smallest = second_smallest = float('inf')

  # Iterate through the list
  for num in data:
    # Update first_smallest if necessary
    if num < first_smallest:
      second_smallest = first_smallest
      first_smallest = num
    # Update second_smallest if necessary, but avoid duplicates
    elif num > first_smallest and num < second_smallest:
      second_smallest = num

  return second_smallest

# Example usage
my_list = [5, 3, 8, 1, 2]
second_smallest = find_second_smallest(my_list)

if second_smallest is not None:
  print(f"Second smallest item: {second_smallest}")
else:
  print("List has less than 2 elements")
```

هر دو اینکار را انجام میدهد. حال کدام بهتر است ؟

1. برای مقایسه به عملگرهای ریاضی نیاز داریم پس باید توابع برنامه نویسی را به دنیای ریاضیات انتقال بدهیم
2. میتوانیم آن ها را به توابع ریاضی تبدیل کنیم .
3. برای اینکار از مدل ram استفاده میکنیم. مدل ram مدلی است که فرض می‌کند الگوریتم با n ورودی بروی یک سیستم کامپیوتری با یک ram و بدون تکنولوژی‌های تسریع الگوریتم همچون موازی سازی اجرا میکنیم و در نهایت تعداد دستورات (میتواند زمان اجرای آن‌ها باشد) به عنوان خروجی این تابع خواهد بود
4. بعبارتی مدل ram تابع fn را بر حسب متغییر n ایجاد میکند. که n تعداد ورودی ما هست و fn تعداد دستوراتی که باید توسط کامپیوتر اجرا شود.

برای مثال بسیار ساده یک دستور for زیر را در نظر بگیرید این تابع به هر تعداد ورودی که دریافت نمایید همان مقادیر را چاپ میکند. پس اگر آرایه‌ای به طول ده بگیرد ۱۰ عدد چاپ میکند و اگر آرایه‌ای به طول ۲۰ بگیرد ۲۰ عدد در خروجی چاپ می‌کند. پس اگر n عدد در آرایه داشته باشیم چند دستور ایجاد می‌کند؟

<table>
<tr>
<td>

```
def print_array_like_struct(data):
    for i in range (len(data)):
        print (data[i])
```

</td>
<td>

```
array = [2,4,5]
print_array_like_struct(array)
```

</td>
</tr>
<tr>
<td></td>
<td>

```
2
4
5
```

</td>
</tr>
</table>

پس من میتوانم تابع فوق را به شکل زیر مدل کنم که نشان دهنده‌ی یک تابع خطی است.

$$f(n) = n$$

N ورودی پس N بار خط `print (data[i])` اجرا شده و nخروجی نمایش داده خواهد شد.

در مدل ram در واقع آن چه که مهم است اجرای تعداد دستورات توسط cpu است آن الگوریتم که بیشتر دستور تولید کند سربار بیشتری خواهد داشت پس در این روش تعداد دستورات را میشماریم.

برای شمارش دستورات

1. خطوط الگوریتم را شماره میزنیم
2. با توجه به نوع دستورات به سراغ دستوراتی میرویم که دستورات زیاد تولید میکنند همانند حلقه‌ها همانند loop و while و ...
3. از بین آن‌ها حلقه‌هایی را در نظر میگیریم که وابسته به طول n هستند
4. فرض میکنیم n ورودی داریم حال برای هر خط از کد تعداد بار اجرای آن خط را میشماریم. برای مثال بالا دو خط کد داریم :

	تعداد دستورات با n ورودی	
`for i in range (len(data)):`	N بار	
`print (data[i])`	N بار	

پس یعنی تعداد کل دستورات برای یک آرایه‌ای با ۱۰ عنصر چند خواهد بود؟ ۲۰ خواهد بود. ده بار خط اول و ده بار خط دوم. و تابع آن

$$f(n) = 2 * n$$

خواهد بود. تابع ۲ کمی دقیق‌تر از تابعی ۱ است که ابتدا با حدس بدست آوردیم.

به همین سادگی تابع الگوریتم کاستوم را بدست آوردیم. حال برای الگوریتم با روش مرتب سازی ادامه میدهیم. برای الگوریتم مدنظر به همین شکل باید تعداد دستورات را بشماریم اما اینبار تابع *Sorted* استفاده شده است. باقی خطوط شامل *loop* و حلقه‌ی خاصی نیست و وابستگی چندانی به ورودی ندارد پس تنها یک خط از کد برای ما جذابیت دارد و آن هم خط

```
sortedData = sorted(data)
```

چرا که باید تعداد کل دستورات را بشماریم و برای اینکار باید به سراغ تابع Sorted برویم و ببینیم خطوط آن شامل چه دستوراتی است و تعداد دستورات آن را برای n ورودی بشماریم.

<table>
<tr><td><pre>def find_second_smallest
(data):
  if len(data)< 2:
    return None

  sortedData = sorted(data)
  return sortedData[1]</pre></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
</table>

معمولا الگوریتم‌های مرتب سازی از دو حلقه‌ی for تو در تو تشکیل شده اند. اینجا نیز یک فرض جالب برای آنکه بتوانیم مفاهیم دیگر چون تحلیل مرتب سازی درجی را داشته باشیم صورت خواهیم داد. فرض میکنیم الگوریتم sorted مرتب سازی درجی است. حال میخواهیم تعداد خطوط الگوریتم مرتب سازی درجی را بشماریم.

بگذارید با الگوریتم مرتب سازی درجی که در کتاب مقدمه‌ای بر الگوریتم‌های clrs آمده است شروع کنیم و آن را به روش این کتاب بشماریم.

Pseudo code	

#	Pseudo code		
0	Insertion-Sort(A)		
1	For j=2 to A.length		
2	Key = A[j]		
3	// insert A[j] into the sorted sequence A[1..j-1]		
4	i = j-1		
5	While i>0 and A[i]>key		
6	A[i+1] = A[i]		
7	i = i - 1		
8	A[i+1] = key		

Code line number	Code line	یا هزینه Cost	دفعات تکرار
0	Insertion-Sort(A)		
1	For j=2 to A.length	c_1	N
2	Key = A[j]	c_2	n-1
3	// insert A[j] into the sorted sequence A[1..j-1]	c_3= 0	n-1
4	i = j-1	c_4	n-1
5	While i>0 and A[i]>key	c_5	$\sum_{j=2}^{n} t_j$
6	A[i+1] = A[i]	c_6	$\sum_{j=2}^{n} t_j - 1$
7	i = i - 1	c_7	$\sum_{j=2}^{n} t_j - 1$
8	A[i+1] = key	c_8	n-1

t_j تعداد اجرای while را نشان میدهد بعبارتی t_j یعنی تعداد دستورات به تعداد t است و این t وابسته به دور j ام هست. به زبان ساده اگر داریم با کمک مرتب سازی درجی اعضای یک صف را مرتب میکنیم. ممکن است j=۷ یا دور محمد باشد و از قضا محمد قدش بلند است پس while هرگز اجرا نشود اما در دور ۱۲ یا j=12 به رضا میرسیم که قدش بسیار کوتاه است و باید به ابتدای صف درج شود در این صورت حلقه‌ی while باید به بیشترین مقدار یعنی j بار اجرا شود. پس در کل tj یک متغییر وابسته به j.

دو درس برای ما دارد:

1. لزومی ندارد دقیقا تعداد دقیق هر خط را نسبت به n بدست بیاورید و میتوانید از متغییرهای کمکی استفاده کنید چرا که در نهایت در بخش تحلیل میتوانید آن را به طور دقیق مشخص کنید.
2. لزومی ندارد تابع بر حسب فقط تعداد باشد و میتوانید با نوشتن هزینه یا cost تابع مد نظر را به واحد زمان انتقال دهید.

بطور کلی الگوریتم مرتب سازی را بار دیگر با مدل ram و شمارش دستورات به شکل سریالی بدنیای ریاضی بردیم و تابع زیر را خلق کردیم:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

بهترین حالت زمانی است که آرایه مرتب باشد و خطوط داخل حلقه‌ی while اصلا اجرا نمی‌شود ... پس در این حالت :

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_8(n-1)$$
$$= c_1 n + c_2 n - c_2 + c_4 n - c_4 + c_5 (n-1) + c_8 n - c_8 =$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

این زمان اجرا را میتوان به صورت $an + b$ نشان داد. که در آن a و b ثابت اند و به هزینه‌های ثابت c_i بستگی دارند. بنابراین تابع بالا یک تابع خطی است.

اما اینکه ما از یک تابع مرتب سازی برای یک آرایه مرتب استفاده کنیم خیلی جالب نیست و تحلیل بهتر مد نظر است و آن هم حالت بدترین حالت و حالت متوسط است.

در بدترین حالت فرض بر آن داریم که بدترین سناریو ممکن رخ داده است و آرایه‌ای که داریم در نامرتب ترین حالت خود است یعنی به طور برعکس از بزرگ به کوچک مرتب شده است و حال الگوریتم مرتب سازی درجی مجبور است برای درج هر عنصر همه‌ی عماصر بخش مرتب را هر بار ببیند و عنصر جدید را در ابتدای آن درج کند. بعبارتی حلقه‌ی while در بدترین حالت برای j=n باید n-1 بار اجرا شود. تابع t_j در حالتی که j = n باشد هم باید n بار در واقع اجرا شود. مثلا اگر سه عنصر مرتب داشته باشیم و عنصر چهارم را بخواهیم وارد این مجموعه درج کنیم j=4 باشد پس $t_j = ۳ + ۱$ خواهد بود چرا که سه عنصر مرتب باید شیفت پیدا کنند و یکبار هم که برای آخر حلقه‌ی while چک خواهد شد. پس $t_j = j$ خواهد شد برای همه‌ی j ها از ۲ تا n.

پس رابطه‌هایی که پیچیده بودند به سادگی بر حسب n در این حالت قابل محاسبه خواهند شد.

$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$	
$\sum_{j=2}^{n} t_j - 1 = \sum_{j=2}^{n} j - 1 = \frac{n(n-1)}{2}$	

حال به فرمول که با شمارش دستورات مرتب سازی درجی بدست آوردیم بر میگردیم و مقادیر را وارد و ساده میکنیم:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

و tj ها را جایگذاری میکنیم:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1\right) + c_6 \frac{n(n-1)}{2} + c_7 \frac{n(n-1)}{2} + c_8(n-1)$$
$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n + (c_2 + c_4 + c_5 + c_8)$$

میتوان بدترین زمان اجرا را به صورت $an^2 + bn + c$ نشان داد. که در آن a،b و c ثابت هایی بر حسب ci هستند. پس تابع یک تابع درجه دوم است.

یادمان نرود داشتیم دو الگوریتم که برای مساله‌ی پیدا کردن دومین مینیمم بود مقایسه بین دو الگوریتم را صورت میدادیم و گفتیم باید برای مقایسه از عملگرهای ریاضی استفاده کنیم پس کدها را به دنیای ریاضیات آوردیم.

$bn = 2n$	**الگوریتم پیدا کردن دومین کمینه به روش custom**
$an^2 + bn + c$	**الگوریتم پیدا کردن دومین کمینه بر حسب مرتب سازی درجی**

پس اگر ۱۰ تا عدد در یک آرایه بود و میخواستیم دومین مینیمم را پیدا کنیم برای الگوریتم cusrom ۲۰ مقایسه و برای الگوریتم درجی حداقل 100مقایسه نیاز داشتیم چرا که در این تابع an^2 داریم و n=10 هست. به سادگی میتوان با همین مثال گفت که کدام الگوریتم برای حل ما مناسب است و آن هم الگوریتم custom هست اما به شکل صحیح تر برای مقابسه‌ی دو تابع ریاضی میتوان از limit گیری استفاده کرد. و اینجا هم همین کار را میکنیم .. اما یکسوال ...

$$\lim_{n\to\infty}\left(\frac{g(n)}{f(n)}\right) = 0$$

حد به سمت بینهایت نسبت دو عدد صفر شده است. این به چه معناست. این بدان معنا است که g(n) سرعت رشد کنتری نسبت به f(n) دارد و یا به عبارتی f(n) بزرگتر از g(n) هستش وقتی که n به سمت بینهایت میل میکند. اما استفاده از تابع حد یک دلیل دیگر هم برای ما خواهد داشت و آن به درک نگاه ما به بی‌نهایت بر می‌گردد.

در واقع مدل‌های ریاضی در بی نهایت هستش که ذات خود را نشان می‌دهند. برای مثال من می‌خواهم مشخص کنم یک سکه سالم است یا غیر سالم اگر سالم باشد توقع دارم اگر تعداد زیادی این سکه را پرتاب کنم ۵۰ درصد شیر و ۵۰ درصد خط بیاید. اگر با پرتاب بسیار بالا متوجه شدیم ۷۰ درصد شیر و ۳۰ درصد خط می‌آید میتوانیم نتیجه بگیریم سکه ناسالم هست.

بعبارتی بحث و مفهوم همگرایی در بینهایت وضعیت مدل‌های ریاضی را نشان می‌دهد. برای مثال برای اینکه به یک شهر بگوییم حاصلخیز و بارانی هست میتوانیم ابتدا ماشین متناهی آن یا FSM آن را ایجاد کنیم و با کمک اصل کولموگروف ببینیم کل مدل به کدام سمت تمایل خواهد داشت.

نماد مجانبی

اما نمادهای مجانبی چیست. حال که کدهای برنامه‌نویسی را به دنیای ریاضی بردیم و توابع ریاضی را برای آن‌ها ساختیم و با کمک مشتق‌گیری (عملگرهای ریاضیاتی) آن‌ها را مقایسه کردیم. حال می‌توانیم آن‌ها را وارد دنیای مجموعه‌های ریاضی (نمودار ون، اشتراک، اجتماع و ...) وارد کنیم و در این دنیا می‌توانیم این توابع ریاضیاتی را گروه بندی کنیم. پس نمادهای مجانبی نمادهایی در دنیای مجموعه‌های ریاضی است که نشان میدهد کدام تابع به کدام مجموعه‌ها تعلق دارد ، کدام مجموعه‌ها از بقیه بزرگتر هست و الی آخر. همانند پرچم‌های محلی، پرچم‌های استانی، شهری، کشوری و ... که سطح و قدرت این مجموعه‌ها را نشان می‌دهد.

به طور واضح تر عملگرهای < بزرگتر، کوچک‌تر و یا = برای مقایسه‌ی اعداد در دنیای محاسباتی هستند حال در دنیای مجموعه‌ها این عملگر ها معادل دارند و آن هم به نمادهای مجانبی مشهور هستند. که در جدول زیر مشاهده میکنیم.

عملگر	نماد مجانبی در دنیای مجموعه‌ها	معنا	برای مثال f(x) بزرگتر از g(x) هستش
$\geq$	Big O	بزرگتر مساوی	$\boldsymbol{g(x) \in O(f(x))}$
$\gg$	Small o	بزرگ‌تر	$\boldsymbol{g(x) \in o(f(x))}$
$=$	$\boldsymbol{\theta}$	برابر	$\boldsymbol{g(x) \in \theta(g(x))}$
$\leq$	Big Omega$\boldsymbol{\Omega}$	کوچکتر مساوی	$\boldsymbol{f(x) \in \Omega(g(x))}$
$\ll$	Small omega $\boldsymbol{\omega}$	کوچکتر	$\boldsymbol{f(x) \in \omega(g(x))}$

از موارد بالا BigO از بقیه بیشتر مورد استفاده قرار میگیرد چرا که برای نمایش در بدترین حالت (کران بالا) کاربرد دارد. میگوییم این الگوریتم با این تابع دیگر از این کران بالا در بدترین حالت خودش خارج نمیشه. برای مثال

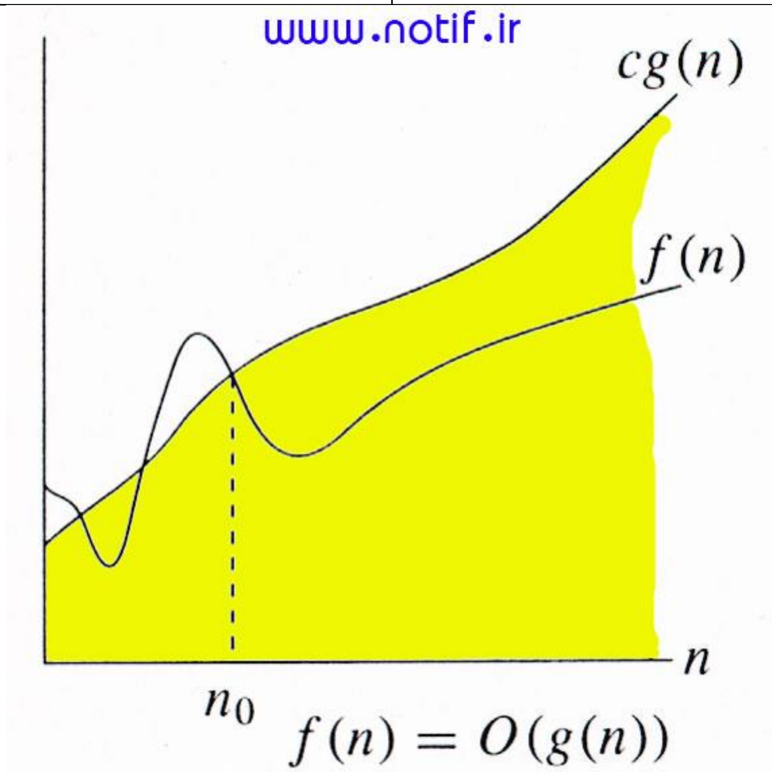
$$\boldsymbol{f(x) = n^2} \text{ و } \boldsymbol{g(x) = n}$$

خوب میدانیم که $f(x) > g(x)$ اگر این دو توابع دو الگوریتم باشند برای مثال f(x) تابع پیدا کردن دومین کمینه به کمک insertion sort و دیگری تابع پیدا کردن دومین کمینه به کمک روش custom باشد آنوقت میتوانیم عبارت زیر را با کمک نمادهای مجانبی به شکل زیر بنویسیم:

$$\boldsymbol{g(x) \in O(f(x))}$$

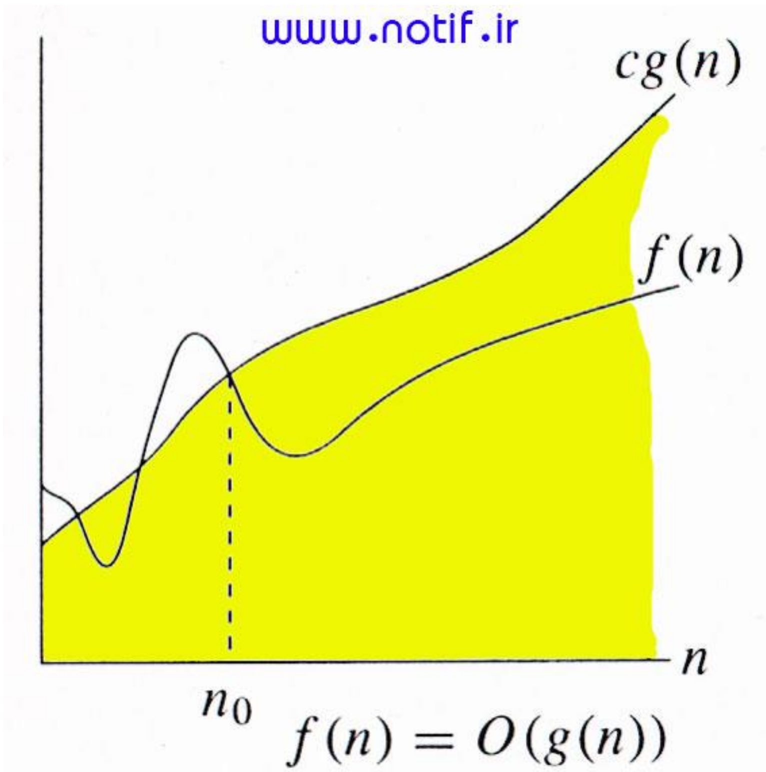
این یعنی آنکه هر چقدر میخواهی n را بزرگ و آشفته کن الگوریتم مبتنی بر custom بهتر نتیجه می‌گیرد و همچنین الگوریتم custom حتی در بدترین سناریوها مرتبه‌ای کمتر از مرتبه f(x) دارد.

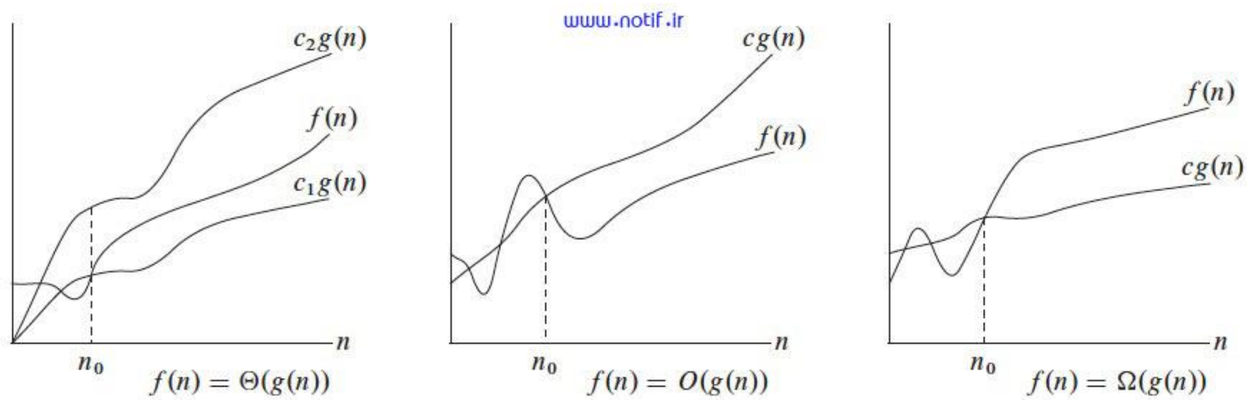
- تعریف ریاضی و دقیق big O

Notif.ir	
$O(g(n)) = f(n): \exists\, C, n_0 > 0 \text{ that } \forall\, n \geq n_0 \text{ we have } 0 \leq f(n) \leq Cg(n)$	
	

تمرین ها و مثالها:

1. روابط ریاضی را برای small o ، small omega و big omega نشان دهید. چرا که شما به آسانی میتوانید دریابید که اگر عبارت a از b کوچکتر است پس عبارت b از a بزرگ‌تر است. بعبارتی اگر برای تابع f(x) و g(x) رابطه‌ی bigO صدق می‌کند به طوریکه f = O(g) پس $g = \Omega(f)$.
2.


www.notif.ir
$cg(n)$
$f(n)$
n
n_0
$f(n) = O(g(n))$


$c_2g(n)$
$f(n)$
$c_1g(n)$
n
n_0
$f(n) = \Theta(g(n))$
www.notif.ir
$cg(n)$
$f(n)$
n
n_0
$f(n) = O(g(n))$
$f(n)$
$cg(n)$
n
n_0
$f(n) = \Omega(g(n))$