به نام او، به یاد او، برای او

Data Structures & Algorithm Design

تجزیه و تحلیل الگوریتم ها

Algorithm Analysis

Dr. *Ali Valinejad*

valinejad.ir
valinejad@umz.ac.ir

اهمیت تجزیه و تحلیل الگوریتم ها

- گاهی اوقات بیش از یک ***الگوریتم*** برای حل یک ***مساله*** وجود دارد. در چنین مواقعی انتخاب ***الگوریتم کارا***(Efficient) بسیار اهمیت دارد. لذا لازم است تا معیاری برای مقایسه عملکرد الگوریتم ها در نظر بگیریم.
- تجزیه و تحلیل الگوریتم‌ها حوزه‌ای از علوم رایانه است که ابزارهایی برای تجزیه و تحلیل کارایی روش های مختلف حل یک مساله ارائه می‌دهد.
- اکثر اوقات ***پیچیدگی زمانی***(***time complexity***) و ***پیچیدگی فضایی***(***space complexity***) الگوریتم ها دو معیار مهم در مقایسه الگوریتم‌ها هستند.
- منظور از ***پیچیدگی زمانی*** یک الگوریتم، حداکثر میزان زمان مصرفی برای اجرای الگوریتم به عنوان تابعی از اندازه مساله می‌باشد.
- منظور از ***پیچیدگی فضایی(مکانی، حافظه‌ای)*** یک الگوریتم، حداکثر مقدار فضا یا حافظه‌ی مورد استفاده در اجرای الگوریتم به عنوان تابعی از اندازه مساله می‌باشد.
- پیچیدگی زمانی و فضایی به موارد زیادی مانند سخت افزار، سیستم عامل، پردازنده‌ها و غیره بستگی دارند.
- به طورکلی، علاقه چندانی به بررسی پیچیدگی زمانی و فضایی الگوریتم‌ها به ازای ورودی‌های با اندازه کوچک نداریم.
- اندازه‌گیری میزان رشد پیچیدگی زمانی و فضایی الگوریتم‌ها با افزایش اندازه ورودی معیاری مناسب برای مقایسه الگوریتم‌ها است.

- از آنجا که تکنیک‌های مورد استفاده برای تعیین میزان حافظه مصرفی، زیر مجموعه‌ای از روش‌های مورد استفاده برای تعیین میزان زمان مصرفی است، در این درس فقط روی روش‌های مورد استفاده برای تعیین ***پیچیدگی زمانی*** الگوریتم تمرکز می‌شود.

پیچیدگی زمانی یک الگوریتم و اندازه ورودی

❑ **پیچیدگی زمانی یک الگوریتم به عنوان تابعی از *اندازه ورودی* اندازه‌گیری می شود.**

- پیچیدگی یک الگوریتم برای مرتب کردن n عنصر را می‌توان به صورت تابعی از n نشان داد.
- پیچیدگی یک الگوریتم برای ضرب یک ماتریس $m \times n$ در یک ماتریس $n \times p$ را می‌توان به صورت تابعی از m، n و p نشان داد.
- پیچیدگی یک الگوریتم برای تعیین اینکه آیا x یک عدد اول است یا خیر را را می‌توان به صورت تابعی از تعداد بیتهای لازم برای نمایش دودویی n نشان داد ($n=[\log_2 (x + 1)]$).

تحلیل پیچیدگی زمانی یک الگوریتم

فرض کنید پیچیدگی زمانی مساله p را به صورت T(p) نشان دهیم، در این صورت:

T(p)=زمان اجرا +زمان کامپایل

زمان کامپایل:

- وابسته به ورودی نیست.
- در مقایسه با زمان اجرا قابل صرفنظر کردن است.

زمان اجرا:

- به دقت قابل اندازه گیری نیست. زیرا وابسته به سخت افزار، سیستم عامل، بار کامپیوتر و ... است.
- به صورت قدم‌های اجرایی مورد نیاز تخمین زده می شود.

با توجه به این که زمان کامپایل در مقایسه با زمان اجرا قابل صرفنظر کردن است، در بررسی پیچیدگی زمانی الگوریتم‌ها فقط زمان اجرا را بررسی می‌کنیم.

تحلیل پیچیدگی زمانی یک الگوریتم

با توجه به این که زمان کامپایل در مقایسه با زمان اجرا قابل صرفنظر کردن است، در بررسی پیچیدگی زمانی الگوریتم‌ها فقط ***زمان اجرا*** را بررسی می‌کنیم.

- وقتی در مورد زمان اجرا یا سرعت یک الگوریتم (شبه کد، قطعه کد) صحبت می‌کنیم باید معیاری برای ارزیابی سرعت داشته باشیم.
- این معیار باید مستقل از سخت افزار کامپیوتر، زبان برنامه نویسی، مهارت برنامه‌نویس و هر چیز مرتبط با کامپیوتر باشد.
- این معیار فقط باید به الگوریتم وابسته باشد.
- زمان اجرای یک الگوریتم به طور معمول با اندازه ورودی رشد می‌کند. معمولا تخمین بدترین زمان اجرا با اهمیت است.

تخمین پیچیدگی زمانی یک الگوریتم

۱- روش مطالعه تجربی،

۲- روش تحلیل تئوری.

تخمین پیچیدگی زمانی یک الگوریتم

۱- روش مطالعه تجربی:

- نوشتن یک برنامه کامپیوتری برای اجرای الگوریتم مورد بررسی،

```
from time import time
start_time=time()
# your code
end_time=time()
Elapsed=end_time-start_time
```

```
#include<time.h>
clock_t  start, stop;
start=clock()
# your code
stop=clock()
cout<<float(stop-start)/CLOCKS_PER_SEC;
```

- اجرای برنامه نوشته شده برای ورودی‌هایی با اندازه متفاوت و ثبت زمان هر اجرا.

(برای هر اندازه ورودی، میانگین زمان چندبار اجرا یادداشت می‌شود)

- نمایش گرافیکی زمان اجرا نسبت به اندازه ورودی

تخمین پیچیدگی زمانی یک الگوریتم

۲- روش تحلیل تئوری:

- نوشتن شبه‌کد یا برنامه کامپیوتری مربوط به الگوریتم مورد بررسی،
- استفاده از توضیحات سطح بالا از الگوریتم به جای اجرای برنامه کامپیوتری مربوط به الگوریتم،
- مدل‌سازی ریاضی زمان اجرای الگوریتم به صورت تابعی از طول یا اندازه ورودی،

- مدل‌سازی ریاضی زمان اجرای یک الگوریتم را می‌توان به دو روش زیر انجام داد:
  - ❖ ***شمارش عمل‌های اصلی***
  - ❖ ***شمارش گام‌های اجرایی***

تخمین پیچیدگی زمانی یک الگوریتم

❑ روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

- ***یک روش برای تخمین پیچیدگی زمانی یک برنامه یا الگوریتم، انتخاب یک یا چند عمل اصلی برای تعیین میزان تکرار انجام هر یک از آن‌هاست. موفقیت این روش بستگی به توانایی ما در شناسایی عملیاتی دارد که بیشترین مشارکت را در پیچیدگی زمانی دارند.***
- تعداد تکرارهای عملیات اصلی در برخی از الگوریتم‌ها(مثل الگوریتم جمع اعضای آرایه) فقط به اندازه ورودی بستگی دارد و در برخی دیگر(مثل جستجوی ترتیبی) هم به اندازه ورودی و هم به خود ورودی بستگی دارد.
- روش سر راستی برای تعیین اعمال اصلی وجود ندارد.
- متناسب با تعداد یا اندازه ورودی یک الگوریتم، معمولا یک یا چند عمل که به دفعات در اجرای الگوریتم تکرار می‌شوند را به عنوان عمل اصلی اجرای الگوریتم انتخاب می‌کنیم.
- چند داوطلب برای عمل اصلی عبارتند از:

  تخصیص مقدار به یک شی، مقایسه دو عدد، انجام یک عمل حسابی، جایگذاری، فراخوانی تابع و ...
- گاهی اوقات دو عمل اصلی که تعداد تکرار یکسانی ندارند، به طور مجزا شمرده می‌شوند.
- همیشه در تحلیل پیچیدگی الگوریتم ها، فرض می‌کنیم اعمال اصلی به بهترین وجه پیاده‌سازی می‌شوند.

تخمین پیچیدگی زمانی یک الگوریتم

- روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

مثال۱: حداکثر تعداد تکرار عمل s++

```
int s = 0;
for (int i = 0; i < N; i++)
   for (int  j = 0; j < i; j++)
      s++;
```

```
s=0
for i in range(N):
    for j in range(i):
        s+=1
```

تخمین پیچیدگی زمانی یک الگوریتم

- روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

مثال۱: حداکثر تعداد تکرار عمل s++

```
int s = 0;
for (int i = 0; i < N; i++)
   for (int  j = 0; j < i; j++)
      s++;
```

```
s=0
for i in range(N):
    for j in range(i):
        s+=1
```

Assume $N=2^k$, how many times s++ will run?

When i=0, it will run 0 times.
When i=1, it will run 1 times.
When i=2, it will run 2 times.
:
When i=N-1, it will run N-1 times.

Total number of times s++ will run is 0+ 1 + … + N-1= N(N-1)/2.
So the time complexity will be **$O(N^2)$**.

تخمین پیچیدگی زمانی یک الگوریتم

- روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

مثال۲: حداکثر تعداد تکرار عمل s++

```
int s = 0;
for (int i = N; i > 0; i /= 2)
   for (int j = 0; j < i; j++)
      s++;
```

```
s=0
i=N
while i>0:
    for j in range(i):
        s+=1
    i//=2
```

تخمین پیچیدگی زمانی یک الگوریتم

- روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

مثال۲: حداکثر تعداد تکرار عمل s++

```
int s = 0;
for (int i = N; i > 0; i /= 2)
   for (int j = 0; j < i; j++)
      s++;
```

```
s=0
i=N
while i>0:
   for j in range(i):
      s+=1
   i//=2
```

Assume $N=2^k$, how many times s++ will run?

When i=N, it will run N times.
When i=N/2, it will run N/2 times.
When i=N/4, it will run N/4 times.
:
When i=1, it will run 1 times.

Total number of times s++ will run is $N+ N/2 +N/4 \dots + 1=$
$N+N/2+N/4+\dots+N/N)=$
$N+N/2+N/4+\dots+N/2^k)=$
$N(1-1/2^{k+1})/(1-1/2)=$
$2N(1-1/2^{k+1})=$
$2N(2^{k+1}-1)/2^{k+1}=$
$2N(2N-1)/(2N)=$
$2N-1$.

So the time complexity will be **O(N)**.

تخمین پیچیدگی زمانی یک الگوریتم

- روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

مثال ۳: الگوریتم محاسبه مجموع درایه‌های یک آرایه

```
def RecursiveSum(arr, n):
    if n<=0:
        return 0
    else:
        return RecursiveSum(arr, n-1)+arr[n-1]
```

عمل اصلی: صدا زدن تابع

حداکثر تعداد تکرار عمل اصلی در الگوریتم بازگشتی برابر **n+1** است

```
def IterativeSum(arr, n):
    s = 0
    for i in range(n):
        s = s + arr[i]
    return s
```

عمل اصلی: جمع دو مقدار

حداکثر تعداد تکرار عمل اصلی در الگوریتم تکراری برابر **n** است.

اندازه آرایه(n)	حداکثر تعداد عمل اصلی	
	الگوریتم تکراری	الگوریتم بازگشتی
۱۰	۱۰	۱۱
۱۵	۱۵	۱۶
۱۰۰	۱۰۰	۱۰۱
۱۰۰۰	۱۰۰۰	۱۰۰۱

برای اندازه ورودی بزرگ، حداکثر تعداد انجام عمل اصلی برای دو الگوریتم تقریبا یکسان است.

تخمین پیچیدگی زمانی یک الگوریتم

- روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

مثال۴: محاسبه جمله nام فیبوناچی با دو الگوریتم *تکراری و بازگشتی*

```
def RecursiveFib(n):
    if n<0:
        print("Incorrect input")
    elif n==0:
        return 0
    elif n==1:
        return 1
    else:
        return RecursiveFib(n-1)+ RecursiveFib(n-2)
```

عمل اصلی: صدا زدن تابع

حداکثر تعداد تکرار عمل اصلی در الگوریتم بازگشتی بزرگتر از $2^{n/2}$ است

```
def IterativeFib(n):
    F=[0]*(n)
    if n < 0:
        print("Incorrect input")
    elif n>0:
        F[0]=1
        F[1]=1
        for i in range(2,n):
            F[i]=F[i-1]+F[i-2]
    return F[-1]
```

عمل اصلی: تخصیص مقدار به متغیر

حداکثر تعداد تکرار عمل اصلی در الگوریتم تکراری برابر **n+1** است.

قضیه: فرض کنید T(n) تعداد فراخوانی‌های تابع RecursiveFib(.) برای محاسبه n امین عدد فیبوناچی باشد. ثابت کنید:

$T(n) > (2^{n/2})$

تخمین پیچیدگی زمانی یک الگوریتم

- روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

مثال۴: محاسبه جمله nام فیبوناچی با دو الگوریتم *تکراری و بازگشتی*

قضیه: فرض کنید T(n) تعداد فراخوانی‌های تابع RecursiveFib(.) برای محاسبه n امین عدد فیبوناچی باشد. ثابت کنید:

$T(n) > (2^{n/2})$

اثبات: به استقراء. داریم:

پایه استقراء:

$T(2)=3>2^{2/2}$

$T(3)=5>2^{3/2}$

فرض استقراء:

$T(m)>2^{m/2}, \quad 2 <= m < n$

گام استقراء:

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

$>2^{(n-1)/2}+ 2^{(n-2)/2}+1$

$>2^{(n-2)/2}+ 2^{(n-2)/2} =2*2^{(n-2)/2}=2^{n/2}$ ■

تخمین پیچیدگی زمانی یک الگوریتم

- روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

مثال۴: محاسبه جمله nام فیبوناچی با دو الگوریتم *تکراری و بازگشتی*

حداکثر تعداد تکرار عمل اصلی در الگوریتم بازگشتی بزرگتر از $2^{n/2}$ است

حداکثر تعداد تکرار عمل اصلی در الگوریتم تکراری برابر **n+1** است.

اندازه ورودی(n)	حداکثر تعداد عمل اصلی	
	الگوریتم فیبوناچی (بازگشتی)	الگوریتم فیبوناچی (تکراری)
۴۰	>۱۰۴۸۵۷۶	۴۱
۶۰	>۱.۱ × ۱۰^۹	۶۱
۸۰	>۱.۱ × ۱۰^۱۲	۸۱
۱۰۰	>۱.۱ × ۱۰^۱۵	۱۰۱
۱۲۰	>۱.۲ × ۱۰^۱۸	۱۲۱
۱۶۰	>۱.۲ × ۱۰^۲۴	۱۶۱
۲۰۰	>۱.۳ × ۱۰^۳۰	۲۰۱

الگوریتم تکراری بسیار سریع‌تر از الگوریتم بازگشتی است.

تخمین پیچیدگی زمانی یک الگوریتم

- روش *شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم

مثال۵: مقایسه الگوریتم هایجستجوی دودویی و جستجوی ترتیبی

```
def IterativeBinarySearch(arr, left, right, x):
    while left <= right:
        mid = left + (right - left)/2;
        if arr[mid] == x:
            return True, mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return False, -1
```

عمل اصلی: مقایسه دو مقدار

حداکثر تعداد مقایسه‌های انجام شده توسط الگوریتم جستجوی دودویی تکراری برابر $\log_2 n+1$ است

```
def SequentialSearch(arr, x):
    pos = 0
    found = False

    while pos < len(arr) and not found:
        if arr[pos] == x:
            found = True
        else:
            pos = pos + 1
    return found, -1
```

عمل اصلی: مقایسه دو مقدار

حداکثر تعداد مقایسه‌های انجام شده توسط الگوریتم جستجوی ترتیبی برابر n است.

اندازه آرایه(n)	حداکثر تعداد مقایسه های انجام شده	
	الگوریتم جستجوی ترتیبی	الگوریتم جستجوی دودویی تکراری
۱۲۸	۱۲۸	۸
۱۰۲۴	۱۰۲۴	۱۱
۱۰۴۸۵۷۶	۱۰۴۸۵۷۶	۲۱
۴۲۹۴۹۶۷۲۹۴	۴۲۹۴۹۶۷۲۹۴	۳۳

معمولا الگوریتم جستجوی دودویی تکراری بسیار سریع‌تر از الگوریتم جستجوی ترتیبی است.

تخمین پیچیدگی زمانی یک الگوریتم

تمرین: حداکثر تعداد مقایسه‌های انجام شده در اجرای الگوریتم جستجوی دودویی بازگشتی با اندازه ورودی n چقدر است؟

```
def RecursiveBinarySearch (arr, left, right, x):
    if right >= left:
        mid = left + (right - left)/2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return RecursiveBinarySearch(arr, left, mid-1, x)
        else:
            return RecursiveBinarySearch(arr, mid+1, right, x)
    else:
        return  -1
```

تخمین پیچیدگی زمانی یک الگوریتم

❑ *شمارش گام‌های اجرایی* برای تخمین پیچیدگی زمانی یک الگوریتم

- *روش شمارش عمل‌های اصلی* برای تخمین پیچیدگی زمانی یک الگوریتم، از زمان صرف شده در انجام همه عمل‌ها غیر از عمل‌های اصلی انتخاب شده صرفنظر می‌کند. اما *روش شمارش گام‌های اجرایی زمان صرف شده در انجام همه قدم‌های اجرایی الگوریتم را در نظر می‌گیرد.*
- *یک گام اجرایی* هر واحد محاسباتی مستقل از اندازه ورودی است (برای مثال، ۱۰ عمل جمع یک گام اجرایی محسوب می‌شود. ۱۰۰ عمل جمع یک گام اجرایی محسوب می‌شود. اما اگر n اندازه ورودی باشد، n عمل جمع یک گام اجرایی محسوب نمی‌شود.).

برای شمارش تعداد گام‌های اجرایی می توان از دو روش زیر استفاده کرد:

- ✓ **روش استفاده از شمارنده count**
- ✓ **روش رسم جدول**

تخمین پیچیدگی زمانی یک الگوریتم

❑ *شمارش گام‌های اجرایی* برای تخمین پیچیدگی زمانی یک الگوریتم

برای شمارش تعداد گام‌های اجرایی می توان از دو روش زیر استفاده کرد:

- ✓**استفاده از شمارنده count**
- ✓**رسم جدول**

قرارداد در شمارش گام‌های اجرایی:

- توضیحاتی که در خطوط برنامه وجود دارند اجرا نمی شوند
- اعلان متغیرها اجرا نمی‌شوند.
- صدا زدن تابع یک گام اجرایی محسوب می شود.
- عبارات محاسباتی مستقل از اندازه ورودی یک گام اجرایی محسوب می‌شوند(هزینه اجرایی یک دارند).
- هزینه اجرایی حلقه‌های for، while برابر دفعات تکرار بعلاوه یک است.
- دستورات شرطی هزینه اجرایی یک دارند بعلاوه هزینه عباراتی که اجرا می‌شوند.
- هزینهreturn ، continue، break برابر یک است.

تخمین پیچیدگی زمانی یک الگوریتم

❑ *شمارش گام‌های اجرایی* برای تخمین پیچیدگی زمانی یک الگوریتم

✓روش استفاده از شمارنده count

- نوشتن یک برنامه کامپیوتری برای اجرای الگوریتم مورد بررسی،
- تعریف متغیر سراسری count از نوع صحیح با مقدار اولیه صفر برای اندازه‌گیری تعداد تکرار گام‌های اجرای الگوریتم،
- هر جایی که یک عبارت اجرا می شود، count را به میزان مناسب افزایش دهید،
- حذف کد مربوط به گام‌های اجرایی،
- اجرای برنامه و چاپ مقدار نهایی count به عنوان تخمینی از پیچیدگی زمانی الگوریتم.

تخمین پیچیدگی زمانی یک الگوریتم

- شمارش گام‌های اجرایی برای تخمین پیچیدگی زمانی یک الگوریتم

✓روش استفاده از شمارنده count

```
def IterativeSum(arr, n):
    s = 0
for i in range(n):
    s = s + arr[i]
return s
```

```
count=0

def IterativeSum(arr, n):
    s = 0
    count+=1 # s=0
    for i in range(n):
        count+=1 # for
        s = s + arr[i]
        count+=1  # s=s+arr[i]
     count+=1  # for
    return s
    count+=1 #return
```

```
count=0

def IterativeSum(arr, n):
    for i in range(n):
        count+=2
    count+=3
```

count = 2n+3

تخمین پیچیدگی زمانی یک الگوریتم

❑ *شمارش گام‌های اجرایی* برای تخمین پیچیدگی زمانی یک الگوریتم

✓روش رسم جدول

- نوشتن یک برنامه کامپیوتری برای اجرای الگوریتم مورد بررسی،
- شماره‌گذاری خط‌های قطعه کد،
- تکرار مراحل زیر برای هر دستور اجرایی:

  مشخص کردن تعداد گام‌های مورد نیاز در یکبار اجرای هر دستور،

  مشخص کردن تعداد تکرار یک دستور

  ترکیب دو گام فوق و محاسبه سهم هر دستور در تعداد کل گام‌ها
- جمع سهم همه دستورات برای محاسبه تعداد کل قدم‌های اجرایی الگوریتم

تخمین پیچیدگی زمانی یک الگوریتم

- شمارش گام‌های اجرایی برای تخمین پیچیدگی زمانی یک الگوریتم

```
def IterativeSum(arr, n):
1)    s = 0
2)    for i in range(n):
3)        s = s + arr[i]
4)    return s
```

✓روش رسم جدول

Line	cost	Freq.	Total
1	c_1	1	c_1
2	c_2	n+1	$c_2(n+1)$
3	c_3	n	c_3n
4	c_4	1	c_4
			$c_1+(c_2+c_3)n+c_2+c_4$

- **ستون اول:** شماره خط،
- **ستون دوم:** هزینه زمانی هر بار اجرای دستور(s/e یا step per execute)،
- **ستون سوم:** تعداد دفعاتی که دستور اجرا میشود(frequency)،
- **ستون چهارم:** مجموع هزینه تکرارهای یک دستور(total).
- **سطر آخر:** مجموع هزینه زمانی کل دستورات اجرایی قطعه کد.

تخمین پیچیدگی زمانی یک الگوریتم

- شمارش گام‌های اجرایی برای تخمین پیچیدگی زمانی یک الگوریتم

```
def IterativeSum(arr, n):
1)    s = 0
2)    for i in range(n):
3)        s = s + arr[i]
4)    return s
```

✓روش رسم جدول

Line	s/e	Freq.	Total
1	c_1	1	c_1
2	c_2	n+1	$c_2(n+1)$
3	c_3	n	c_3n
4	c_4	1	c_4
			$c_1+(c_2+c_3)n+c_2+c_4$

Line	s/e	Freq.	Total
1	1	1	1
2	1	n+1	n+1
3	1	n	n
4	1	1	1
			2n+3

تخمین پیچیدگی زمانی یک الگوریتم

❑ *شمارش گام‌های اجرایی* برای تخمین پیچیدگی زمانی یک الگوریتم

✓روش رسم جدول

```
def RecursiveSum(arr, n):
1)   if n<=0:
2)       return 0
3)   else:
4)       return RecursiveSum(arr, n-1)+arr[n-1]
```

Line	s/e	Freq.		Total	
		n<=0	n>0	n<=0	n>0
1	c_1	1	0	c_1	0
2	c_2	1	0	c_2	0
3	c_3	0	1	0	c_3
4	c_4	0	1+T(n-1)	0	c_4(1+T(n-1))
				c_1+c_2	c_3+c_4+c_4T(n-1)

$$T(n) = \begin{cases} c_1+c_2 & n \le 0 \\ c_3+c_4+c_4T(n-1) & n > 0 \end{cases}$$

$T(n)$: پیچیدگی زمانی تابع RecursiveSum(arr, n)

تخمین پیچیدگی زمانی یک الگوریتم

❑ *شمارش گام‌های اجرایی* برای تخمین پیچیدگی زمانی یک الگوریتم

```
def RecursiveSum(arr, n):
1)   if n<=0:
2)       return 0
3)   else:
4)       return RecursiveSum(arr, n-1)+arr[n-1]
```

✓روش رسم جدول

با توجه به قرارداد:

$c_1=c_2=c_3=c_4=1$

Line	s/e	Freq.		Total	
		n<=0	n>0	n<=0	n>0
1	1	1	0	1	0
2	1	1	0	1	0
3	1	0	1	0	1
4	1	0	1+T(n-1)	0	(1+T(n-1))
				2	2+T(n-1)

$$T(n)=\begin{cases}2 & n\le 0\\ 2+T(n-1) & n>0\end{cases}$$

$T(n)$: پیچیدگی زمانی تابع RecursiveSum(arr, n)

تخمین پیچیدگی زمانی یک الگوریتم

❑ *شمارش گام‌های اجرایی* برای تخمین پیچیدگی زمانی یک الگوریتم

✓روش رسم جدول

```
def RecursiveSum(arr, n):
1)   if n<=0:
2)       return 0
3)   else:
4)       return RecursiveSum(arr, n-1)+arr[n-1]
```

$$T(n) = \begin{cases} 2 & n \leq 0 \\ 2+T(n-1) & n > 0 \end{cases} \Rightarrow T(n) = 2n + 2$$

اثبات:

$$for \quad n > 0 : \quad T(n) = 2 + T(n-1)$$
$$= 2 + 2 + T(n-2) = 2 * 2 + T(n-2)$$
$$= 2 * 2 + 2 + T(n-3) = 2 * 3 + T(n-3)$$
$$= 2 * 4 + T(n-4)$$
$$:$$
$$= 2 * n + T(0) = 2n + 2$$

تخمین پیچیدگی زمانی یک الگوریتم

تمرین۱: با استفاده از روش استفاده از شمارنده count تعداد گام‌های اجرایی قطعه کد زیر را محاسبه کنید.

تمرین۲: با استفاده از روش جدول تعداد گام‌های اجرایی قطعه کد زیر را محاسبه کنید.

```
# C=C+AB
def Mat2Mat(A, B, C, m, p, n,):
    for i in range(m):
        for j in range(n):
            for k in range(p):
                C[i,j]=C[i,j] + C[i,k]* C[k,j]
    return C
```

تخمین پیچیدگی زمانی یک الگوریتم

۱- روش مطالعه تجربی،

۲- روش تحلیل تئوری.

- نوشتن و اجرای کامل برنامه کامپیوتری برای بعضی از الگوریتم‌ها بسیار مشکل و زمانبر است.
- برخلاف روش مطالعه تجربی، روش تحلیل تئوری تاثیر همه ورودی‌های احتمالی را بر زمان اجرا در نظر می‌گیرد.
- برخلاف روش مطالعه تجربی، بررسی سرعت اجرای یک الگوریتم با استفاده از روش تحلیل تئوری مستقل از مشخصات سخت افزاری و نرم افزاری است.

انواع پیچیدگی زمانی یک الگوریتم

- **بهترین حالت(B(n)):**تعداد تکرار گام‌های اجرایی به ازای بهترین ورودی را پیچیدگی زمانی الگوریتم در بهترین حالت گویند.

  مثال: در الگوریتم جستجوی دودویی یک عنصر در آرایه‌ای به طول n، B(n)=1.

- **بدترین حالت(W(n)):** تعداد تکرار گام‌های اجرایی به ازای بدترین ورودی را پیچیدگی زمانی الگوریتم در بدترین حالت گویند.

  مثال: در الگوریتم جستجوی دودویی یک عنصر در آرایه‌ای به طول n، W(n)=lgn+1.

- **حالت میانگین(A(n)):** اگر در الگوریتمی تعداد تکرار گام‌های اجرایی برای ورودی‌های مختلف با اندازه یکسان برابر نباشد، میانگین تعداد تکرار برای همه ورودی‌ها را پیچیدگی زمانی الگوریتم در حالت میانگین گویند.

  مثال: اگر در الگوریتم جستجوی ترتیبی یک عنصر در آرایه‌ای به طولn، بدانیم عنصر مورد نظر با احتمال p در آرایه وجود دارد و احتمال حضور عنصر در هر خانه دلخواه p/n باشد، A(n) از رابطه زیر محاسبه می‌شود:

$$A(n) = \sum_{k=1}^{n} k\frac{p}{n} + n(1-p) = n\left(1-\frac{p}{2}\right) + \frac{p}{2}$$

  - $k\frac{p}{n}$: احتمال وجود در هر خانه
  - n: تعداد تکرار های لازم وقتی کلید در آرایه وجود ندارد.
  - $(1-p)$: احتمال عدم وجود در آرایه

- **حالت معمولی(T(n)):** اگر در الگوریتمی تعداد تکرار گام‌های اجرایی برای همه ورودی‌های با اندازه یکسان برابر باشد، تعداد تکرارها را پیچیدگی زمانی الگوریتم در حالت معمولی گویند.

  مثال: در الگوریتم جمع کردن عناصر آرایه‌ای به طول n، T(n)=n.

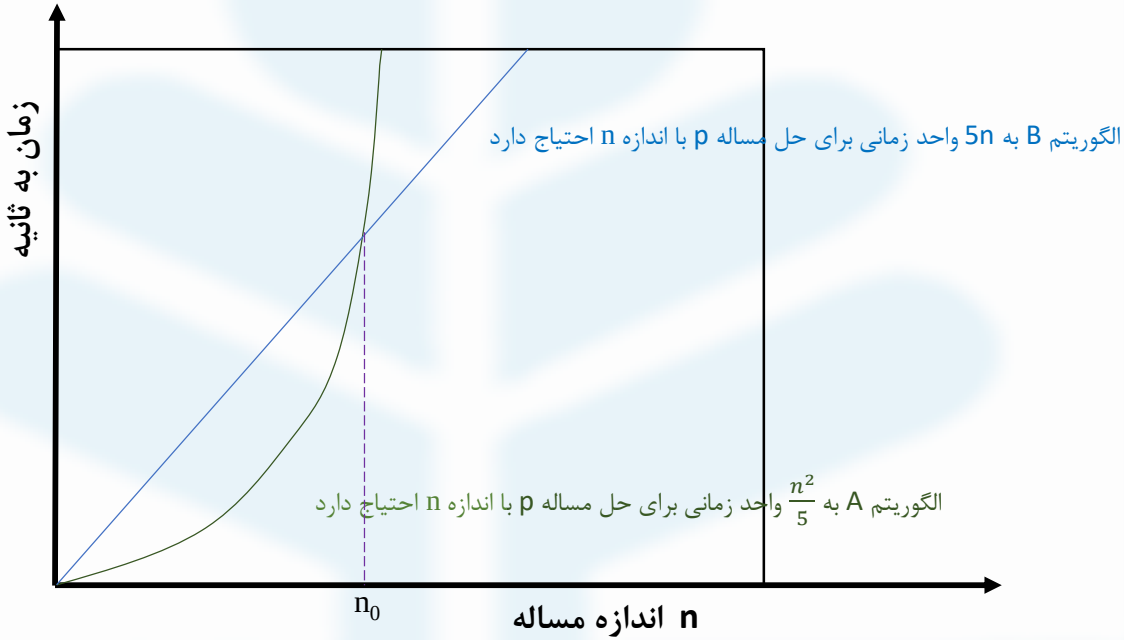
انواع پیچیدگی زمانی یک الگوریتم

- از w(n) زمانی استفاده می‌شود که حتی به ازای برخی از ورودی‌ها نباید زمان اجرای الگوریتم بالا باشد.
- از A(n) زمانی استفاده می شود که الگوریتم به دفعات زیاد و روی انواع ورودی‌ها اجرا می‌شود.
- محاسبه A(n) سخت‌تر از محاسبه w(n) است.
- در مواقعی که T(n) موجود باشد داریم: T(n)=W(n)=A(n)=B(n)

به طور کلی تابع پیچیدگی تابعی است از مجموعه اعداد صحیح مثبت به مجموعه اعداد حقیقی نامنفی.

$$f(n)\colon \mathbb{Z}^{+} \longrightarrow \mathbb{R}^{+}\cup\{0\}$$

نرخ رشد الگوریتم‌ها (Algorithm Growth Rates)



سوال: با توجه به نمودار فوق، کدام الگوریتم مساله p را کاراتر (در این جا سریعتر) حل می کند؟

جواب: برای مقایسه کارایی الگوریتم‌ها می‌توان از نرخ رشد آنها استفاده نمود.

نرخ رشد الگوریتم‌ها (Algorithm Growth Rates)

- زمان اجرای الگوریتم‌ها به عنوان تابعی از اندازه مساله اندازه‌گیری می‌شود. به عنوان مثال می‌گوییم:

  الگوریتم A به $\frac{n^2}{5}$ واحد زمانی برای حل یک مساله با اندازه ورودی n احتیاج دارد.

  الگوریتم B به 3n واحد زمانی برای حل یک مساله با اندازه ورودی n احتیاج دارد.

- سوالی اساسی در تحلیل الگوریتم‌ها، تعیین ***میزان رشد*** زمان اجرای یک الگوریتم به عنوان تابعی از اندازه مساله است.

  الگوریتم A به زمانی متناسب با n^2 برای حل یک مساله با اندازه ورودی n احتیاج دارد.

  الگوریتم B به زمانی متناسب با n برای حل یک مساله با اندازه ورودی n احتیاج دارد.

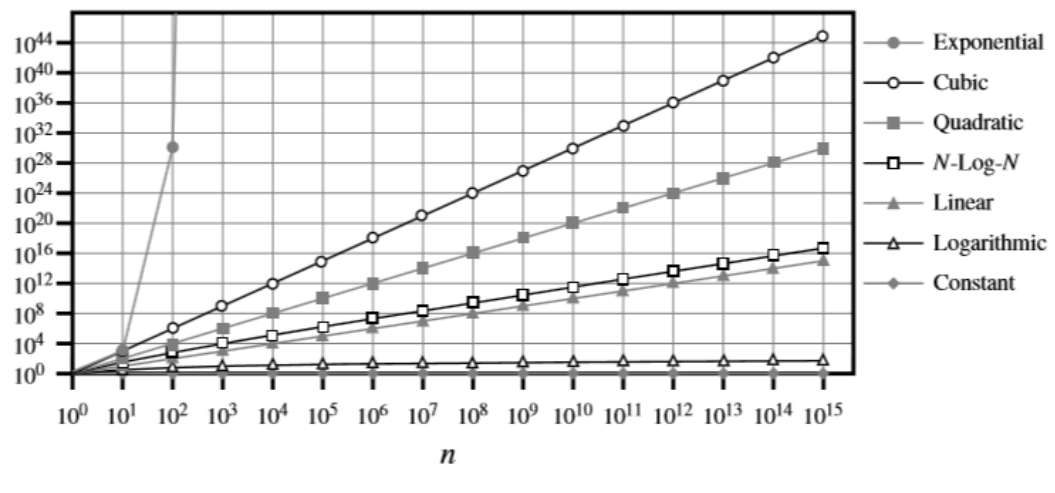
- نیاز زمانی متناسب یک الگوریتم را ***نرخ رشد*** آن الگوریتم نامند.

- بطور خلاصه، در تجزیه و تحلیل الگوریتم‌ها روی ***نرخ رشد*** زمان اجرای الگوریتم‌ها به عنوان تابعی از اندازه ورودی تمرکز می‌شود. به عنوان مثال، اغلب کافیست بدانیم که زمان اجرای یک الگوریتم (با اندازه ورودی n)، متناسب با n افزایش می‌یابد نه اینکه الگوریتم به 3n واحد زمانی برای حل یک مساله با اندازه ورودی n احتیاج دارد.

- با محاسبه نرخ رشد دو الگوریتم می‌توان آن دو الگوریتم را با هم مقایسه کرد.

نرخ رشدهای متداول

تابع	نام نرخ رشد
c	ثابت
$log\ N$	لگاریتمی
N	خطی
$N\ log\ N$	$N\ log\ N$
N^2	مربعی
N^3	مکعبی
a^N	نمایی



نکته: شیب خط متناظر با هرکدام از این توابع در نمودارlog-log، نرخ رشد آن را نشان می‌دهد.

آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

- در بحث تحلیل الگوریتم‌ها بجای درنظر گرفتن رفتار دقیق توابع، رفتار مجانبی آنها مورد بررسی قرار می‌گیرد.
- زمان اجرای الگوریتم‌ها، با استفاده از توابعی انجام نشان داده می‌شود که با حذف عوامل ثابت، اندازه ورودی n را به مقادیری متناظر با عامل‌های اصلی می‌نگارند. این عامل‌های اصلی نرخ رشد زمان اجرای الگوریتم را براساس اندازه ورودی تعیین می‌کنند.
- نمادهای مجانبی، ابزارهایی ریاضی برای نمایش پیچیدگی زمانی الگوریتم ها در تحلیل مجانبی الگوریتم ها هستند.

$$10n^2 + 4n + 2 = O(n^2)$$

آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

✓ نماد اوی بزرگ(Big oh)

- گوییم تابع $f(n)$ از مرتبه(درجه) اوی بزرگ تابع $g(n)$ است، هرگاه:

$$\exists\ c, n_0 > 0 \quad s.t. \quad \forall \quad n > n_0 \qquad f(n) \leq c\ g(n)$$

در این صورت می‌نویسیم: $f(n) = O(g(n))$

مثال۱: تابع $f(n) = 3n + 2$ از مرتبه $O(n)$ است. زیرا

$$if\ c = 4\ \ \&\ \ n_0 = 2 \quad then \quad \forall \quad n > n_0 \quad 3n + 2 \leq 4n$$

مثال۲: تابع $f(n) = 10n^2 + 4n + 2$ از مرتبه $O(n^2)$ است. زیرا

$$if\ c = 11\ \ \&\ \ n_0 = 5 \quad then \quad \forall \quad n > n_0 \quad 10n^2 + 4n + 2 \leq 11n^2$$

مثال۳: تابع $f(n) = 10n^2 + 4n + 2$ از مرتبه $O(n)$ **نیست**. زیرا

$$\nexists\ c, n_0 > 0 \quad s.t. \quad \forall \quad n > n_0 \quad 10n^2 + 4n + 2 \leq cn$$

آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

✓ نماد اوی بزرگ(Big oh)

- گوییم تابع $f(n)$ از مرتبه(درجه) اوی بزرگ تابع $g(n)$ است، هرگاه:

$$\exists\ c, n_0 > 0 \quad s.t. \quad \forall \quad n > n_0 \qquad f(n) \leq c\ g(n)$$

در این صورت می‌نویسیم: $f(n) = O(g(n))$

نکته۱: $f(n) = 3n + 2$ از مرتبه $O(n)$، $O(n^2)$، $O(n^3)$، $O(n^4)$ و ... است. اما از $O(n)$ برای مرتبه اوی بزرگ $f(n)$ استفاده می‌شود.

نکته۲: اگر $f(n)$ تابعی چندجمله‌ای باشد، جمله بزرگتر لحاظ می‌گردد(مثال۳).

- $2n^3 + 3n^2 + n$

$= 2n^3 + 3n^2 + O(n)$

$= 2n^3 + O(3n^2 + n)$

$= 2n^3 + O(n^2)$

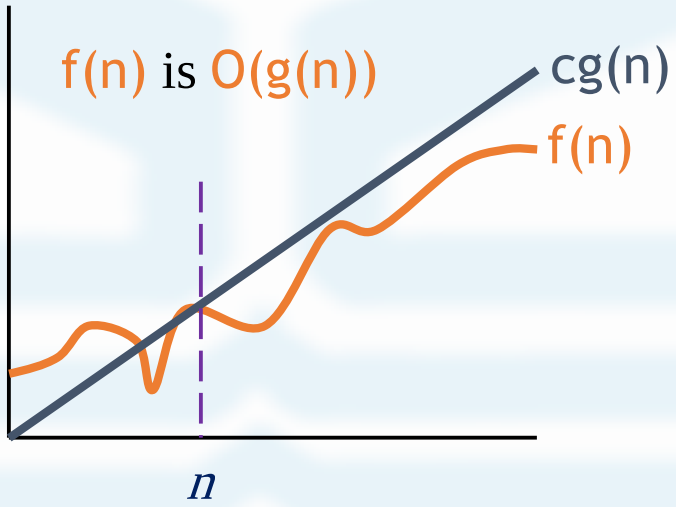
$= O(2n^3 + n^2)$

$= O(n^3) = O(n^4) = O(n^5)$

آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

✓ نماد اوی بزرگ(Big oh)

- ❖ $f(n) = O(g(n))$ به این مفهوم است که الگوریتمی که دارای تابع پیچیدگی $f(n)$ است به ازای nهای بزرگ **حداقل به تندی** $g(n)$ است.
- ❖ $f(n) = O(g(n))$ یک **حد بالای** برای $f(n)$ قرار می‌دهد که لزوما بهترین حد بالا نیست.
- ❖ در آنالیز مجانبی الگوریتم‌ها به دنبال کوچکترین حد بالا هستیم.



آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

✓ نماد اوی بزرگ(Big oh)

- ❖ $f(n) = O(g(n))$ به این مفهوم است که الگوریتمی که دارای تابع پیچیدگی $f(n)$ است به ازای nهای بزرگ **حداقل به تندی** $g(n)$ است.
- ❖ $f(n) = O(g(n))$ یک **حد بالای** برای $f(n)$ قرار می‌دهد که لزوما بهترین حد بالا نیست.
- ❖ در آنالیز مجانبی الگوریتم‌ها به دنبال کوچکترین حد بالا هستیم.

نماد اوی بزرگ	تابع	نام نرخ رشد
$O(1)$	c	ثابت
$O(\log N)$	$\log N$	لگاریتمی
$O(N)$	N	خطی
$O(N \log N)$	$N \log N$	$N \log N$
$O(N^2)$	N^2	مربعی
$O(N^3)$	N^3	مکعبی
$O(a^N)$	a^N	نمایی

آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

✓ نماد امگای بزرگ(Ω)

- گوییم تابع $f(n)$ از مرتبه(درجه) امگای بزرگ تابع $g(n)$ است، هرگاه:

$$\exists\ c, n_0 > 0 \quad s.t. \quad \forall \quad n > n_0 \qquad f(n) \geq c\ g(n)$$

در این صورت می‌نویسیم: $f(n) = \Omega(g(n))$

مثال۱: تابع $f(n) = 3n + 2$ از مرتبه $\Omega(n)$ است. زیرا

$$if\ c = 3\ \ \&\ \ n_0 = 1 \quad then \quad \forall \quad n > n_0 \quad 3n + 2 \geq 3n$$

مثال۲: تابع $f(n) = 10n^2 + 4n + 2$ از مرتبه $\Omega(n^2)$ است. زیرا

$$if\ c = 1\ \ \&\ \ n_0 = 1 \quad then \quad \forall \quad n > n_0 \quad 10n^2 + 4n + 2 \geq n^2$$

مثال۳: تابع $f(n) = 10n^2 + 4n + 2$ از مرتبه $\Omega(1)$ **است**.

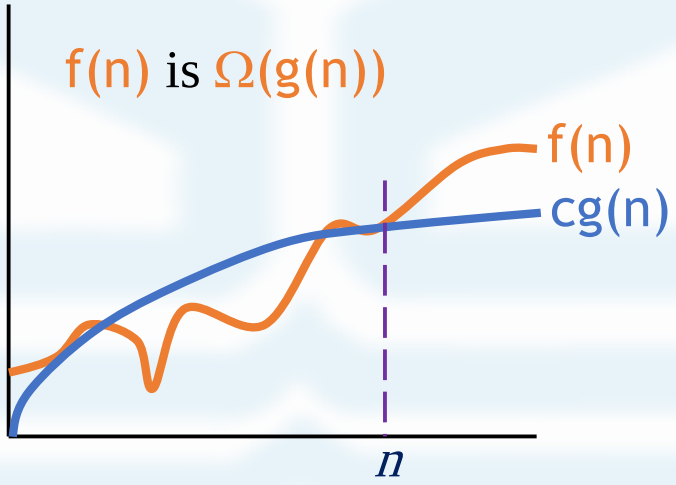
مثال۴: تابع $f(n) = 10n^2 + 4n + 2$ از مرتبه $\Omega(n)$ **است**.

نکته۱: اگر $f(n)$ تابعی چندجمله‌ای باشد، جمله بزرگتر لحاظ می‌گردد(مثال۲).

آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

✓ نماد امگای بزرگ(Ω)

- ❖ $f(n) = \Omega(g(n))$ به این مفهوم است که الگوریتمی که دارای تابع پیچیدگی $f(n)$ است به ازای nهای بزرگ **حداکثر به تندی** $g(n)$ است.
- ❖ $f(n) = \Omega(g(n))$ یک **حد پایین** برای $f(n)$ قرار می‌دهد که لزوما بهترین حد پایین نیست.
- ❖ در آنالیز مجانبی الگوریتم‌ها به دنبال بزرگترین حد پایین هستیم.



آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

✓ نماد تتای بزرگ(Θ)

- گوییم تابع $f(n)$ از مرتبه(درجه) تتای بزرگ تابع $g(n)$ است، هرگاه:

$$\exists\ c_1, c_2, n_0 > 0 \quad s.t. \quad \forall \quad n > n_0 \quad c_2 g(n) \leq f(n) \leq c_1\, g(n)$$

در این صورت می‌نویسیم: $f(n) = \Theta(g(n))$

مثال۱: تابع $f(n) = 3n + 2$ از مرتبه $\Theta(n)$ است. زیرا

$$if \quad c_1 = 3,\ c_2 = 4 \quad \& \quad n_0 = 2 \quad then \quad \forall \quad n > n_0 \qquad 3n \leq 3n + 2 \leq 4n$$

مثال۲: تابع $f(n) = 10n^2 + 4n + 2$ از مرتبه $\Theta(n^2)$ است.

نکته۱: اگر $f(n)$ تابعی چندجمله‌ای باشد، جمله بزرگتر لحاظ می‌گردد(مثال۲).

❖ $f(n) = \Theta(g(n))$ یک **حد بالا و پایین سفت و سخت** روی $f(n)$ قرار می‌دهد.

آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

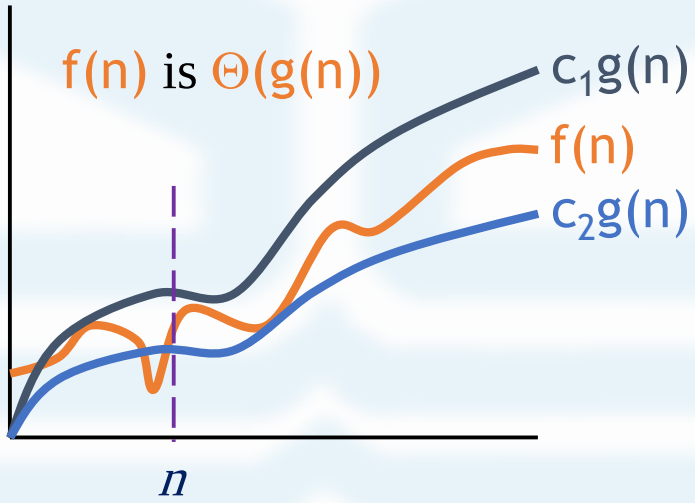
✓ نماد تتای بزرگ(Θ)

- گوییم تابع $f(n)$ از مرتبه(درجه) تتای بزرگ تابع $g(n)$ است، هرگاه:

$$\exists\ c_1, c_2, n_0 > 0 \quad s.t. \quad \forall \quad n > n_0 \quad c_2 g(n) \le f(n) \le c_1\, g(n)$$

در این صورت می‌نویسیم: $f(n) = \Theta(g(n))$

❖ $f(n) = \Theta(g(n))$ یک **حد بالا و پایین سفت و سخت** روی $f(n)$ قرار می‌دهد.



آنالیز مجانبی الگوریتم (Asymptotic Algorithm Analysis)

- **اگر Ω و O برابر نباشند. تفاوت بین این دو نشان می دهد که هنوز جای کار وجود دارد:**

  در مورد O می دانیم که مسئله حداکثر این مقدار کار نیاز دارد. لذا، اگر الگوریتمی از این بزرگتر باشد به درد نمی خورد .

  در مورد Ω می دانیم که مسئله حداقل این مقدار کار نیاز دارد .لذا، باید تلاش کنیم که راه حل خود را به این نقطه برسانیم.

- **اگر Ω و O برابر باشند یعنی الگوریتم بهینه است.**

تجزیه و تحلیل الگوریتم ها- نماد اوی بزرگ (Big-Oh)

Suppose a program P is $\mathbf{O(n^3)}$, and a program **Q** is $\mathbf{O(3^n)}$, and that currently both can solve problems of size **50** in **1** hour. If the programs are run on another system that executes exactly **729** times as fast as the original system, what size problems will they be able to solve (in **1** hour)?

$n^3 = 50^3 * 729$

$n = \sqrt[3]{50^3} * \sqrt[3]{729}$

$n = 50*9$

$n = 50*9 = 450$

$3^n = 3^{50} * 729$

$n = \log_3 (729 * 3^{50})$

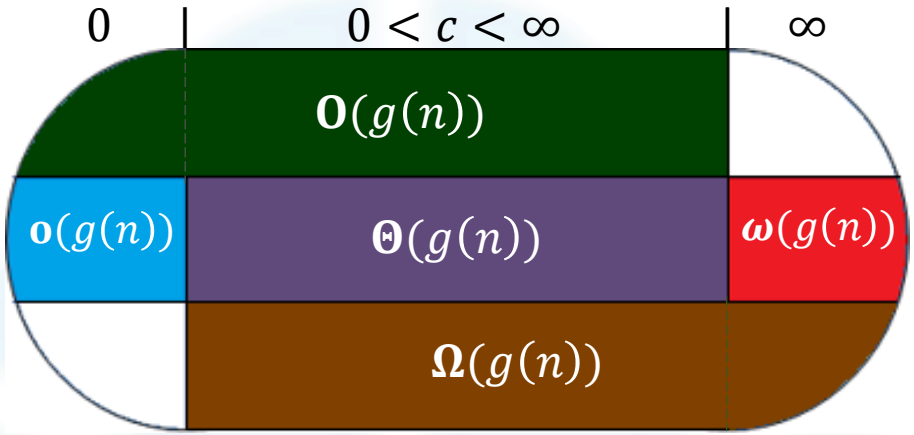
$n = \log_3(729) + \log_3 3^{50}$

$n = 6 + \log_3 3^{50}$

$n = 6 + 50 = 56$

- ***Improvement**: problem size increased by **9** times for **n^3** algorithm but only a slight improvement in problem size (**+6**) for **exponential** algorithm.*

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



$$if \quad 0 \le \lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty \quad \Rightarrow \quad f(n) = \mathbf{O}(g(n))$$

$$if \quad \lim_{n\to\infty}\frac{f(n)}{g(n)} = 0 \quad \Rightarrow \quad f(n) = \mathbf{o}(g(n))$$

$$if \quad 0 < \lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty \quad \Rightarrow \quad f(n) = \mathbf{\Theta}(g(n))$$

$$if \quad \lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty \quad \Rightarrow \quad f(n) = \boldsymbol{\omega}(g(n))$$

$$if \quad 0 < \lim_{n\to\infty}\frac{f(n)}{g(n)} \quad \Rightarrow \quad f(n) = \mathbf{\Omega}(g(n))$$

[1] Data Structures and Algorithms in Python, Goodrich, Tamassia & Goldwasser, 2013
[2] *Foundations of Algorithms Using C++ Pseudocode*, Richard Neapolitan, Kumarss Naimipour, 2009.
[3] *The Desin and Analysis of Algorithms,* Anany Levitin, 2012.
[4] Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Donald L. Rivest, and Clifford Stein
[5] http://www.algorithmha.ir
[6] CENG 213 Data Structures, Algorithm Analysis
[7] ECE 250 Data Structures, Algorithm
[8] www.cis.upenn.edu/~matuszek/cit594-2004/Lectures/49-analysis-2.ppt
[9] https://stackabuse.com/big-o-notation-and-algorithm-analysis-with-python-examples/
[10] https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis
[11] https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/
[12]http://sharif.ir/~shahram.khazaei/files/courses/data-structures/lecture04.pdf
[13]- درس طراحی الگوریتم ها با شبه کد های++ c ، جعفر پورامینی