

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل اول: آموزش پایتون

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

## فهرست

- مقدمه: معرفی پایتون
- متغیرها
- انواع داده پایه‌ای
  - شرط‌ها: if
  - Block‌ها
  - لیست‌ها
- mutable / immutable
- زوج‌های مرتب
- مجموعه‌ها
- دیکشنری‌ها
- حلقه‌ها
- import
- توابع
- help
- class
- مطالب اضافی

## معرفی پایتون

پایتون یه زبان برنامه‌نویسی متن‌باز، تفسیری و سطح بالاست که سال ۱۹۹۱ توسط گیدو فان روسم ساخته شد. طراحی پایتون روی خوانایی کد و نوشتن برنامه‌ها با خطوط کمتر تمرکز دارد که باعث می‌شود پیاده‌سازی سریع‌تر انجام بشود.

In [1]: `print("hello world!")`

hello world!

In [2]: `# PEP0020, Zen of Python!`

`import` this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than \*right\* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!

## معرفی پایتون

## ذن پایتون، نوشه تیم پیترز

زیبا بهتر از زشت است.  
صریح بهتر از ضمنی است.  
ساده بهتر از پیچیده است.  
پیچیده بهتر از پراهام است.  
هموار بهتر از تو در تو است.  
پراکنده بهتر از فشرده است.  
خوانایی اهمیت دارد.  
موارد خاص به اندازه کافی خاص نیستند که قوانین را زیر پا بگذارند.  
اگرچه عمل‌گرایی بر خلوص ارجح است.  
خطاهای هرگز نباید بی‌سروصدا نادیده گرفته شوند.  
مگر اینکه صراحتاً پنهان شده باشند.  
در مواجهه با ابهام، سوسه حدس زدن را رد کن.  
باید یک راه – و ترجیحاً تنها یک راه – مشخص برای انجام کار وجود داشته باشد.  
گرچه ممکن است در ابتدا این راه برای شما مشخص نیاشد مگر اینکه هلندی باشید.  
حالا بهتر از هرگز است.  
اگرچه هرگز اغلب بهتر از «همین حال» است.  
اگر توضیح پیاده‌سازی سخت است، ایده بدی است.  
اگر توضیح پیاده‌سازی آسان است، شاید ایده خوبی باشد.  
فضای نام (Namespaces) ایده‌ای فوق العاده است – بیایید بیشتر از آن استفاده کنیم!

## متغیرها

پایتون یه زبان با «نوع‌دهی پویا» (dynamically typed) است. این یعنی نوع یک متغیر در زمان اجرا (runtime) مشخص می‌شه و شما لازم نیست از قبل نوعش رو تعریف کنید. در واقع، خود «اشیا» (objects) در پایتون نوع دارند، اما متغیرها صرفاً به اون اشیا ارجاع میدن.

```
In [3]: # متغیرها
# در پایتون، نوع متغیر در زمان اجرا مشخص می‌شه و نیازی به تعریف از قبل نداره #
# --- مثال: نوع‌های داده مختلف ---
# یک متغیر با مقدار عددی صحیح (integer)
var_int = 10
print(f"\n\t{var_int: {var_int}}")
print(f"\n\t نوع var_int: {type(var_int)}")
print(f"\n\t ایا var_int از نوع int است؟ {isinstance(var_int, int)}")

# یک متغیر با مقدار اعشاری (float)
var_float = 10.5
print(f"\n\t{var_float: {var_float}}")
print(f"\n\t نوع var_float: {type(var_float)}")
print(f"\n\t ایا var_float از نوع float است؟ {isinstance(var_float, float)}")

# یک متغیر با مقدار متنی (string)
var_str = 'salam'
print(f"\n\t{var_str: {var_str}}")
print(f"\n\t نوع var_str: {type(var_str)}")
print(f"\n\t ایا var_str از نوع str است؟ {isinstance(var_str, str)}")

# یک متغیر با مقدار لیست (list)
var_list = [1, 2, 'a']
print(f"\n\t{var_list: {var_list}}")
print(f"\n\t نوع var_list: {type(var_list)}")
print(f"\n\t ایا var_list از نوع list است؟ {isinstance(var_list, list)}")

# --- مثال: تغییر نوع یک متغیر ---
# حالا یک متغیر رو می‌سازیم و نوعش رو عوض می‌کنیم
var_dynamic = 100
print(f"\n\t نوع var_dynamic: {type(var_dynamic)}")
var_dynamic = "من جدید"
print(f"\n\t نوع var_dynamic: {type(var_dynamic)}")
print(f"\n\t ایا var_dynamic از نوع int است؟ {isinstance(var_dynamic, int)}")
```

```

مقدار var_int: 10
نوع var_int: <class 'int'>
ایا var_int از نوع int است؟ True

مقدار var_float: 10.5
نوع var_float: <class 'float'>
ایا var_float از نوع float است؟ True

مقدار var_str: salam
نوع var_str: <class 'str'>
ایا var_str از نوع str است؟ True

مقدار var_list: [1, 2, 'a']
نوع var_list: <class 'list'>
ایا var_list از نوع list است؟ True

نوع اولیه var_dynamic: <class 'int'>
نوع var_dynamic: <class 'str'>
ایا var_dynamic هنوز از نوع int است؟ False

```

## انواع داده‌ای پایه:

- Boolean: یا همان `True` و `False`. برای مقادیر منطقی استفاده می‌شود (به بزرگ و کوچک بودن حروف دقت کنید).
- اعداد (Numeric): شامل `int` (اعداد صحیح)، `float` (اعداد اعشاری) و `complex` (اعداد مختلط). (نکته: در پایتون ۳، `int` با `long` ادغام شده و نوع جداگانه‌ای نیست).
- رشته‌ها (Strings): برای ذخیره متن استفاده می‌شود و می‌توانه با علامت نقل قول تکی یا دو تایی تعریف بشود.
- None: یه نوع داده‌ی خاصه که نشون می‌ده یک متغیر هیچ مقداری نداره. معادل `NULL` در زبان‌هایی مثل C++ یا Java است.

In [4]: # متغیرها  
مثال‌هایی از انواع داده عددی در پایتون و نحوه کار با آنها #

```

a = 6                      # یک عدد صحیح (integer)
b = 10                     # یک عدد صحیح
d = 3.5                     # در پایتون ۳ وجود نداره 'Long' نکته: نوع float. یک عدد اعشاری
c1 = 1 + 2j                 # یک عدد مختلط (complex)
c2 = 2 + 1j

--- عملیات ریاضی پایه ---
# خواهد بود float جمع یک عدد صحیح و اعشاری؛ نتیجه به صورت type promotion
print(f"a + d = {a + d}")

# تقسیم در پایتون ۳ همیشه به صورت اعشاری است
print(f"a / b = {a / b}")

# تقسیم یک عدد صحیح بر اعشاری
print(f"a / d = {a / d}")

# تقسیم صحیح (integer division)
print(f"a // b = {a // b}")

# --- عملیات با اعداد مختلط ---
print(f"\nc1 + c2 = {c1 + c2}")
print(f"c1 * c2 = {c1 * c2}")

# کار با اعداد بزرگ ---
# برای نگهداری اعداد خیلی بزرگ استفاده می‌کنند 'big-integer' پایتون به طور خودکار از
big_int = 123 ** 50
print(f"\n{big_int}: یک عدد خیلی بزرگ!")

# --- مثال بیشتر برای تقسیم ---
a1 = 12.5
a2 = 123.2
print(f"\ninteger division a2 بر a1: {a2 // a1}")  # تقسیم صحیح
print(f"float division a2 بر a1: {a2 / a1}")      # تقسیم اعشاری

```

```

a + d = 9.5
a / b = 0.6
a / d = 1.7142857142857142
a // b = 0

c1 + c2 = (3+3j)
c1 * c2 = 5j

```

یک عدد خیلی بزرگ: 312791953184952432773037647427847997734252247042927663351793818343238523007334595314754865270114370908249

تقسیم صحیح (integer division) a2 بر a1: 9.0  
تقسیم اعشاری (float division) a2 بر a1: 9.856

In [5]: # متغیرها  
مثال‌هایی از انواع تعریف رشته‌ها در پایتون و عملیات روی آنها #

```

# --- انواع تعریف رشته ---
# رشته با نقل قول تکی
s1 = 'salam, man ye reshteh hastam!'
# رشته با نقل قول دو تایی، که می‌توانه شامل کاراکترهای خاص مثل newline و tab باشد
s2 = "salam, man ye reshteh \n\tbozorg hastam!"
# رشته چندخطی با سه نقل قول
s3 = """man
ye
reshteh
chand-khatam!"""

# رشته چندخطی با استفاده از بکسلش
s4 = "man ye reshteh \
yek-khati hastam."

```

```

# --- چاپ و عملیات روی رشته‌ها
print('s1: ' + s1)
print('s2: ' + s2)
print('s3: ' + s3)
print('s4: ' + s4)

# استفاده از عملگر ضرب برای تکرار رشته
print('\nhaha ' * 3)

# --- رشته‌ها (Concatenation) مثال: الحاق
# می‌شه چون نمی‌شه به رشته رو با به عدد جمع کرد TypeError این کار باعث ایجاد خطای.
# print('number ' + 2)
# برای حل این مشکل، باید عدد رو به رشته تبدیل کنیم
print('number ' + str(2))

# --- برای قالب‌بندی مدرن: f-string مثال
# f-string ها راهی مدرن و خوانا برای نمایش متغیرها داخل رشته هستن.
name = 'Ali'
age = 25
print(f"سال دارم {age} هست و {name} سلام، اسم من\n") # سال دارم {age} هست و {name} سلام، اسم من

# --- (Slicing) مثال: طول رشته و اسلایس
# طول رشته رو برمی‌گردونه len() تابع
print(f"\n{s1[:len(s1)]} طول رشته")
# اسلایس رشته برای دسترسی به بخشی از اون
print(f"s1: {s1[2:6]}") # بخش دوم تا پنجم

```

```

s1: salam, man ye reshteh hastam!
s2: salam, man ye reshteh
      bozorg hastam!
s3: man
ye
reshteh
chand-khatam!
s4: man ye reshteh yek-khati hastam.

```

```

haha
haha
haha
number 2

```

هست و 25 سال دارم Ali سلام، اسم من.

```

s1: طول رشته 29
s1: بخش دوم تا پنجم

```

```

In [6]: # متغیرها
        مثال‌هایی از رشته‌ها، کار با مسیر فایل‌ها و عملگرهای رشته‌ای.

        # --- انواع رشته‌ها
        # back-slash (\\) یک رشته عادی با بک‌اسلیش
        # پایتون به این بک‌اسلیش‌ها به چشم کاراکترهای خاص نگاه می‌کنه
        path = 'C:\\users\\documents\\..'

        # raw string (r'') یک رشته خام
        # در ابتدای رشته، پایتون بک‌اسلیش رو نادیده می‌گیره 'r' با استفاده از
        # این روش برای کار با مسیر فایل‌ها خیلی بهتر و مطمئن‌تره
        raw_string = r'C:\\users\\documents\\..'

        print(f"مسیر عادی: {path}")
        print(f"مسیر خام: {raw_string}")

```

```

# --- عملگرها روی رشته
# بزرگترین کاراکتر (بر اساس مقدار بونیکد) رو برمی‌گردونه max() تابع
print(f"\npath: {max(path)}")
# کوچکترین کاراکتر (بر اساس مقدار بونیکد) رو برمی‌گردونه min() تابع
print(f"path: {min(path)}")

```

```

# --- Case-Sensitive (بررسی وجود یک زیررشته در رشته)
# به بزرگ و کوچک بودن حروف حساسه 'in' عملگر
print(f"'users' in path")
print(f"'Users' in path")

```

```

# --- مثال بیشتر
str2 = 'python workshop'
print(f"\n'py' وجود داره؟ {'py' in str2}")

```

```

C:\\users\\documents\\..
C:\\users\\documents\\..

```

```

path: u
کاراکتر بزرگترین کاراکتر در
path: .
'users' در وجود داره؟ True
'Users' در وجود داره؟ False

```

'py' در str2 وجود True

```

In [9]: # متغیرها
        مثال‌هایی از روش‌های مختلف قالب‌بندی رشته در پایتون

user = 'Amin'
password = 1234

# --- روشن قدمی: استفاده از %
# کار می‌کنه C در زیان printf این روش قدمی‌تره و مثل
# برای اعداد صحیح استفاده می‌شه %d برای رشته و %
print('hello %s, welcome! %d' % (user, password))

# --- روشن جدیدتر: استفاده از تابع format()

```

```
# این روش تمیزتر و قابل انعطاف‌تره
# می‌توانی متغیرها رو به ترتیب داخل {} قرار بدی
print('hello {}, welcome!'.format(user))

# می‌توانی به صورت نام‌گذاری شده هم متغیرها رو قرار بدی
print('hello {user}, {greeting}!'.format(user=user, greeting='welcome'))

# روش مدرن: استفاده از --- f-string ---
# این بهترین و راحت‌ترین روش برای قالب‌بندی رشته‌هاست
# قبل از رشته بنزاری و متغیرها رو مستقیماً داخل {} قرار بدی 'f' کافیه یک
print(f"hello {user}, welcome! You're logged in with password {password}.")
```

```
hello Amin, welcome! 1234
hello Amin, welcome!
hello Amin, welcome!
hello Amin, welcome! You're logged in with password 1234.
```

In [10]: # متغیرها  
مثال‌هایی از رشته‌های یونیکد در پایتون #

```
# در پایتون ۳، همه رشته‌ها به طور پیش‌فرض یونیکد هستند
# در ابتدای رشته نیست 'u' پس نیازی به
fa = 'سلام'
```

فقط مقدار رشته چاپ می‌شود با print().  
print(fa)

```
# اگر اسم متغیر رو به تنها یک در خط آخر بنویسید
# رو نشون می‌ده (representation) ژوپیتر نمایش داخلی اون
# که شامل علامت‌های نقل قول هم هست
fa
```

سلام

Out[10]: 'سلام'

In [11]: هستن (unicode) در پایتون ۳، همه رشته‌ها به طور پیش‌فرض از نوع یونیکد #. این باعث می‌شود که مشکل مقایسه انواع مختلف رشته حل بشود #

```
# یک رشته عادی فاقد پیشوند
str_fa = 'سلام'
```

```
(برای سازگاری با پایتون ۲) u یک رشته با پیشوند
در پایتون ۳ این دو کاملاً یکسان هستند
unicode_str_fa = u'سلام'
```

```
حالا می‌توانیم این دو رو به راحتی با هم مقایسه کنیم
print(f"str_fa == unicode_str_fa")
```

ایا str\_fa و unicode\_str\_fa برابرند؟ True

In [12]: # زیررشته‌ها (Substrings)
به بخش‌هایی از یک رشته دسترسی پیدا کنی (index) در پایتون، می‌توانی با استفاده از ایندکس‌ها.

```
s = "it's a wonderful life!"
```

```
# ایندکس‌گذاری (Indexing) ---  
ایندکس‌گذاری مثبت: از صفر شروع می‌شود و از چپ به راست می‌رود
print(f"s[4]": کاراکتر پنجم (ایندکس ۴) رشته
print(f"s[5]": کاراکتر ششم (ایندکس ۵) رشته)
```

```
ایندکس‌گذاری منفی: از منفی یک شروع می‌شود و از راست به چپ می‌رود
print(f"s[-17]": کاراکتر هفدهم از انتها)
```

```
# برش زدن ---  
[شروع: پایان] 5
# نکته مهم: کاراکتر در ایندکس 'پایان' هیچ وقت شامل نمی‌شود
print(f"s[5:]: از ایندکس ۵ تا انتها")
print(f"s[5:13]": از ایندکس ۵ تا ۱۳ (کاراکتر ۱۲)
print(f"s[5:-9]": از ایندکس ۵ تا ۱۲ کاراکتر مانده به انتها)
print(f"s[:13]": از ابتداء تا ایندکس ۱۳ (کاراکتر ۱۲)
print(f"s[:-9]": از ابتداء تا ۹ کاراکتر مانده به انتها)
```

```
# برش زدن با گام ---  
[شروع: پایان] 5[گام]
print(f"s[::2]": هر کاراکتر دوم از ابتداء تا انتها)
print(f"s[::-1]": برعکس کردن رشته با گام منفی)
```

کاراکتر پنجم (ایندکس ۴) رشته:  
a: کاراکتر ششم (ایندکس ۵) رشته

a: کاراکتر هفدهم از انتها

a: از ایندکس ۵ تا انتها!  
a: از ایندکس ۵ تا ۱۳ (کاراکتر ۱۲)  
a: از ایندکس ۵ تا ۱۲ کاراکتر مانده به انتها  
it's a wonder: از ابتداء تا ایندکس ۱۳ (کاراکتر ۱۲)  
it's a wonder: از ابتداء تا ۹ کاراکتر مانده به انتها

i': هر کاراکتر دوم از ابتداء تا انتها  
!efil lufrednow a s'ti: برعکس کردن رشته با گام منفی

In [13]: # متغیرها  
# نحوه گرفتن ورودی از کاربر با استفاده از تابع input()

از کاربر می‌گیره و برمی‌گردونه (string) یک رشته (string) تابع input() یک کلمه یا عدد وارد کنید(")

```
print(f"ورودی شما {a}")
# خواهد بود، حتی اگه عدد وارد کنید 'str' همیشه 'a' نوع متغیر
print(f"نوع ورودی شما {type(a)}")
```

```
# مثال: تبدیل ورودی به عدد ---  
کنیم (cast) برای استفاده از ورودی به عنوان عدد، باید اون را تبدیل  
("لطفاً یک عدد وارد کنید")  
یک عدد صحیح است و می‌توانید با اون محاسبات انجام بدهید 'b' حالا  
print(f" عدد ورودی شما {b}")  
print(f" نوع ورودی {type(b)}")  
print(f" حاصل جمع عدد با 10 {b + 10}")
```

```
ورودی شما: 1  
<class 'str'>  
عدد ورودی شما: 1  
<class 'int'>  
حاصل جمع عدد با 10: 11
```

If

دستور if دقیقاً همون‌طور که انتظار دارید کار می‌کنه. فقط یادتون باشه که لازم نیست دور شرط‌ش پرانتر بذارید، هرچند اگه بذارید هم مشکلی نداره. نکته مهم اینه که در پایان دستور حتماً از علامت دونقطه (: ) استفاده کنید تا بلاک کد جدیدی شروع بشه.

در پایتون، if نداریم و باید به جای اون از کلمه‌ی کلیدی elif else استفاده کنیم. همچنین، مثل اکثر زبان‌های برنامه‌نویسی، گزاره‌های and و or به صورت «اتصال‌کوتاه» (short-circuit) اجرا می‌شن.

In [14]: # و منطق اتصال‌کوتاه (if-elif-else) دستورات شرطی

```
a = 8  
  
# مثال: استفاده از ---  
پایتون به ترتیب شرط‌ها رو بررسی می‌کنه و اولین شرطی که درست باشه رو اجرا می‌کنه  
if a < 5:  
    print('a کوچکتر از 5 است')  
# elif a < 10: همچنانی، مثل اکثر زبان‌های برنامه‌نویسی، گزاره‌های and و or به صورت «اتصال‌کوتاه» (short-circuit) استفاده کنیم.  
elif a < 10:  
    print('a بزرگتر یا مساوی 5 و کوچکتر از 10 است')  
else:  
    print('a بزرگتر یا مساوی 10 است')  
  
یک خط خالی برای خوانایی بیشتر  
print()  
  
# منطق اتصال‌کوتاه ---  
بینه اولین شرط رو 'or' در پایتون، اگر عملگر  
دیگه سمت راستش رو بررسی نمی‌کنه. این باعث می‌شه از خطای جلوگیری بشه  
if a < 10 or (2 / 0):  
    درسته 'a < 10' این خط اجرا می‌شه چون # این خط اجرا می‌شه چون  
    اتفاق نمی‌افته 'division by zero' به خاطر همین، خطای  
    print('هیج خطایی اتفاق نیفتد')  
  
توضیحات بیشتر #  
کار می‌کنه short-circuit به صورت ('and' و 'or' همینطور) این نشون می‌ده که #
```

a بزرگتر یا مساوی 5 و کوچکتر از 10 است.

هیج خطایی اتفاق نیفتد.

## بلاک‌ها

در پایتون، تورفتگی کد (indentation) نقش خیلی مهمی داره و بلاک‌ها رو از هم جدا می‌کنه. برخلاف خیلی از زبان‌های دیگه که از آکولاد یا کلمات کلیدی استفاده می‌کنن، پایتون شما رو مجبور می‌کنه که یه روش خاص برای تورفتگی داشته باشید.

هر «tab» یا چند «space» می‌تونه یه مرحله تورفتگی حساب بشه. با این حال، بهتره فقط از «space» استفاده کنید و همیشه تعداد ثابتی (مثلاً 4 تا) رو رعایت کنید. این کار هم خوانایی کد رو بالا می‌بره و هم از خطاهای احتمالی جلوگیری می‌کنه.

وقتی یه مرحله تورفتگی اضافه می‌کنید، یه بلاک درونی جدید شروع می‌شه و وقتی اون رو برمنی‌دارید، بلاک قبلی تموم می‌شه. همیشه قبل از شروع یک بلاک جدید، در انتهای خط قبلی یک کاراکتر دونقطه (: ) وجود داره.

دستوراتی که بعد از اون‌ها باید بلاک جدیدی شروع بشه، شامل:

- else و if
- while و for
- (تعریف تابع) def

یه نکته دیگه اینکه هیج بلاکی نباید خالی باشه. اگه به هر دلیلی مجبور شدید یه بلاک خالی بذارید، می‌توانید از کلمه‌ی کلیدی pass استفاده کنید تا پایتون خطای نگیره.

In [15]: # متغیرها  
مثال عملی از بلاک‌های کد و تورفتگی.

```
if True:  
    print('block 1')  
    # این یک بلاک کد درونی است (nested block).  
    # به تورفتگی دقت کنید.  
    if True:  
        print('\tblock 1.1')  
        if True:  
            print('\t\tblock 1.1.1')  
            # این هم یک بلاک درونی دیگه است.  
        if True:  
            print('\t\tblock 1.2')  
    else:  
        # استفاده شده تا بلاک خالی نباشه pass اینجا از در پایتون، بلاک‌های خالی خطای ایجاد می‌کنن
```

```
pass

# این بلاک اصلی کد است که بعد از اتمام بلاک‌های درونی اجرا می‌شود
# (اصلی قرار گرفته است if چون این خط خارج از تورفتگی)
print('block 2')
```

```
block 1
  block 1.1
    block 1.1.1
  block 1.2
block 2
```

```
In [ ]: # Exercise 1
# Check if the input text has '\n' or '\t'.

# input1: sala\nm
# output1: True

# input2: sala\mn
# output2 : False
```

## لیست‌ها

لیست‌های پایتون دقیقاً مثل «آرایه‌های با طول متغیر» یا همان `vector` در زبان‌هایی مثل C++ و Java هستند. این ساختار داده‌ای از چندین ویژگی مهم برخوردار است:

- ترتیب‌دار و قابل تغییر (**Ordered & Mutable**): آیتم‌های داخل لیست ترتیب دارند و می‌توانند اون‌ها را بعد از ساخت، تغییر بدهید، اضافه یا حذف کنید.
- پشتیبانی از انواع داده مختلف: یک لیست می‌تواند شامل انواع داده‌های مختلف (مثل عدد، رشته یا حتی لیست‌های دیگر) باشد.

قابلیت «برش زدن» یا همان `slicing` هم یکی از ابزارهای بسیار قدرتمند پایتون برای کار با لیست‌هاست که به شما اجازه می‌دهد به راحتی به بخشی از اون‌ها دسترسی پیدا کنید.

```
In [16]: # متغیرها
# لیست‌ها (Lists)

# می‌توانید خطوط رو داخل [] یا () یا {} بشکنید تا کد خواناتر بشود.
# l = [
#   6,
#   2,
#   10,
#   3,
#   5,
# ]
# خیلی مفیده (diff) همچنین می‌تواند کامای آخر (,) رو بذارید، که به کارتون آسیبی نمی‌زند و برای ویرایش بعدی کد
# print(f"طول لیست{l}") # طول لیست

# --- ایندکس‌گذاری (Indexing) ---
# print(f"\nاین آیتم: {l[0]}") # اولین آیتم
# print(f"آخرین آیتم: {l[-1]}") # آخرین آیتم

# --- برش زدن (Slicing) ---
# شروع: پایان
# l[1:3] # آیتم در ایندکس 'بایان' هیچوقت شامل نمی‌شود.
# l[1:-1] # همه آیتم‌ها به جز اولی و آخری
# l[:3] # سه آیتم اول
# l[-2:] # دو آیتم آخر
# l[0:100] # اگه ایندکس خارج از محدوده باشه، خطا نمی‌ده
# print(f"طول زدن با گام {l[::2]}") # شروع: پایان: گام
# print(f"هر آیتم دوم: {l[1::2]}") # هر آیتم دوم

# --- لیست‌ها می‌توان انواع داده مختلفی داشته باشند
# l = ['salam', 12, 5.23]
# print(f"\nیه لیست با انواع داده مختلف: {l}") # یه لیست با انواع داده مختلف
```

طول لیست: 5

اولین آیتم: 6  
آخرین آیتم: 5

آیتم‌های دوم و سوم: [10, 2]  
همه آیتم‌ها به جز اولی و آخری: [3, 10, 2, 10]  
سه آیتم اول: [10, 2, 6]  
دو آیتم آخر: [5, 3]

اگه ایندکس خارج از محدوده باشه، خطأ نمی‌ده: [6, 2, 10, 5.23]

هر آیتم دوم: [5, 10, 6]

[': یه لیست با انواع داده مختلف

```
In [17]: # متغیرها
# مثال‌هایی از متدهای لیست برای مرتب‌سازی و معکوس کردن
```

```
l = [6, 2, 10, 3, 5]
print(f"لیست اصلی: {l}")

# --- متدهای reverse() و sort() ---
# reverse() می‌کنه لیست رو به صورت درجا (in-place) معکوس شود.
l.reverse()
print(f"\nلیست معکوس شده: {l}")

# --- متدهای sort() و sort() می‌کنه لیست رو به صورت درجا (in-place) مرتب شود.
l.sort()
print(f"\nلیست مرتب شده: {l}")
```

```
# اون رو به صورت نزولی مرتب کنی reverse=True می‌تونی با پارامتر
l.sort(reverse=True)
print(f" {l} : لیست مرتب شده نزولی")
```

یک لیست مرتب شده جدید برمی‌گردونه sorted() لیست را تغییر می‌دهد، اما تابع sort() نکته مهم: متند.

```
12 = [6, 2, 10, 3, 5]
13 = sorted(12)
print(f"\n لیست اصلی بعد از sorted(): {12}")
print(f" لیست مرتب شده توسط sorted(): {13}")
```

لیست اصلی: [5, 6, 2, 10, 3]

لیست معکوس شده: [5, 3, 2, 10, 6]

لیست مرتب شده: [2, 3, 5, 6, 10]  
لیست مرتب شده نزولی: [10, 6, 5, 3, 2]

رو بیدا می‌کنه (که ایندکس 7 هست) 'w' این متند ایندکس اولین: 'welcome'.  
رو بیدا می‌کنه (که ایندکس 7 هست) 'w' این خط به صورت مرحله‌ای کار می‌کنه.

```
# متغیرها
مثال: ترکیب متدهای رشته‌ای و برش زدن برای ایجاد متن جدید
# s = 'hello, welcome to python programming!'

--- ترکیب متدها و برش زدن ---
# این خط به صورت مرحله‌ای کار می‌کنه
# 1. s.find('w') این متند ایندکس اولین: 7 (عنی) برش زدن رشته از ایندکس 7 تا 14: 'welcome'.
# 2. s[7:7+7]: 'welcome'.
# 3. s[:4]: 'hell'.
# 4. upper(): 'HELL'.
# 5. در نهایت، همه اینها با هم ترکیب می‌شون.
print(s[s.find('w'):s.find('w')+7] + ' to ' + s[:4].upper() + '!')
```

welcome to HELL!

```
# متغیرها
و متدهای اونها (List) مثال: کار با لیستها
# --- ساخت لیست و بررسی وجود آیتم ---

# 6 ساخت یک لیست از اعداد 0 تا 7
r = list(range(7))
print(f" {r} : لیست اصلی")
```

بررسی می‌کنیم که آبا یک آیتم توی لیست وجود دارد یا نه.

```
print('بررسی وجود آیتم\n--')
print(f" 3 هست؟ {3 in r}")
print(f" 11 هست؟ {11 in r}")
```

متونید یک آیتم به آخر لیست اضافه کنید() با append() اضافه کردن آیتم.

```
# --- اضافه کردن آیتم ---
print('اضافه کردن آیتم\n--')
r.append(10)
r.append(7)
r.append(10)
print(f" {r} : لیست بعد از اضافه کردن آیتم")
```

تعداد تکرار یک آیتم رو در لیست برمی‌گردونه count() متند.

```
# --- شمارش تعداد آیتم ---
print('شمارش\n--')
print(f" تعداد 10 در لیست: {r.count(10)}")
```

اولین رخداد از یک آیتم رو حذف می‌کنه (متند remove()).
 # --- حذف آیتم ---
 print('حذف\n--')
 r.remove(10)
 print(f" {r} : لیست بعد از حذف اولین 10")
 print(f" تعداد فعلی 10 در لیست: {r.count(10)}")
 print(f" طول لیست فعلی: {len(r)}")

لیست اصلی: [6, 0, 1, 2, 3, 4, 5, 6, 7, 10]

-- بررسی وجود آیتم:
 هست? 3 آیا عدد 3 در True
 هست? 11 آیا عدد 11 در False

-- اضافه کردن آیتم:
 لیست بعد از اضافه کردن آیتم: [10, 0, 1, 2, 3, 4, 5, 6, 7, 10, 10]

-- شمارش:
 تعداد 10 در لیست: 2

-- حذف:
 لیست بعد از حذف اولین 10: [10, 0, 1, 2, 3, 4, 5, 6, 7]
 تعداد فعلی 10 در لیست: 1
 طول لیست فعلی: 9

```
# متغیرها
مثال: ساخت لیست و مقداردهی اولیه به آیتمها
# با استفاده از عملگر ضرب می‌تونی یک لیست با چند آیتم تکراری بسازی
arr = [0] * 10
print(f" {arr} : لیست اولیه")
```

می‌تونی با یه خط کد، به چند آیتم از لیست مقدار بدی # انجام می‌شه (in-place). این کار به صورت درجا arr[4] = arr[2] = 11
print(f" \n {arr} : لیست بعد از تغییر آیتمها")



```

a = 'salam'
try:
    a[0] = 'S'
except TypeError as e:
    print(f"Error: {e}")

برای تغییر رشته باید به رشته جدید سازی.
a = 'S' + a[1:]
print(f"رشته جدید: {a}")

```

اشیای تغییرنابذیر (id) شناسه‌ی برای مقدار id(a) و id(b) (140727401661704) :12 هستند. بعد از تغییر a: (140727401661736, 140727401661704)

لیست‌ها (id) شناسه‌ی:  
1: 2329838813120  
بعد از تغییر: 2329838813120 1 شناسه‌ی

-- ارجاع به شیء یکسان:  
اولیه: a\_ref و b\_ref ([1, 1, 1], [1, 1])  
بعد از تغییر: ([1, 3, 1], [1, 3, 1])

-- ساخت یک شیء جدید (کپی):  
اولیه: a\_copy و b\_copy ([1, 1, 1], [1, 1])  
بعد از تغییر: ([1, 3, 1], [1, 1, 1])

-- تلاش برای تغییر رشته --:  
Error: 'str' object does not support item assignment  
رشته جدید: Salam

## زوج (چندتایی)‌های مرتب

«چندتایی‌های مرتب» ( tuples ) دقیقاً مثل لیست‌های «تغییرنابذیر» (immutable) هستند.

یکی از برتری‌های اصلی اونها نسبت به لیست، اینه که Hashable هستند. به همین دلیل، می‌توانیم از اونها به عنوان کلید در یک Dictionary (که جلوتر می‌بینیم) استفاده کنیم.

مقداردهی چندتایی ( tuple assignment ) هم به قابلیت بازه و مفیده که تو بعضی موارد می‌توانه تعداد خطوط کد رو حسابی کم کنه.

```

In [23]: # متغیرها
          # مثال: کار با چندتایی‌ها (Tuple)

چندتایی‌ها با پرانتز ساخته می‌شون و مثل لیست‌ها هستند
# نیستن (immutable) با این تفاوت که قابل تغییر نیستند.
t = (10, 6, 19)

# --- دسترسی و بریش زدن (Indexing & Slicing) ---
print(f"اولین ایتم: {t[0]}") # برش زدن هم مثل لیست‌ها کار می‌کند
print(f"همه ایتمها به جز آخر: {t[0:-1]}") # عملگر ضرب روی چندتایی‌ها

# --- عملگر ضرب، چندتایی رو تکرار می‌کند و یک چندتایی جدید می‌سازد
print(f"چندتایی تکرار شده: {t * 2}") # مواجه می‌شی Type Error اگه بخوای یک آیتم از چندتایی را تغییر بدی، با خطای
# جوون چندتایی‌ها تغییرنابذیر نیستند.
# این خط باعث خطا می‌شه <-- # t[1] = 4

# --- تبدیل چندتایی به لیست و بالعکس
# می‌توانی یک چندتایی رو به لیست تبدیل کنی و بعد تغییرش بدی
l = list(t)
l[1] = 4
print(f"لیست تغییر یافته: {l}")

```

اولین ایتم: 10  
همه ایتمها به جز آخر: (6, 10)  
چندتایی تکرار شده: (10, 6, 19, 10, 6, 19)

لیست تغییر یافته: [10, 6, 19, 10, 6, 19]

```

In [24]: # متغیرها
          # و جابجاگری متغیرها (Tuple Unpacking) مثال: مقداردهی چندتایی

```

می‌توانی مقادیر یک لیست یا چندتایی رو به راحتی به چند متغیر نسبت بدی.  
این کار کد رو خیلی تمیز و خوانا می‌کند.  
a, b, c = [1, 2, 3]
print(f"مقدار اولیه a: {a}")
print(f"مقدار اولیه b: {b}")
print(f"مقدار اولیه c: {c}")

print('---')

با استفاده از همین قابلیت، می‌توانی مقادیر چند متغیر رو در یک خط با هم جابجا کنی.  
نباری (temporary variable) دیگه نیاری به متغیر کمکی!  
a, b, c = c, b, a
print(f"مقدار جدید a: {a}")
print(f"مقدار جدید b: {b}")
print(f"مقدار جدید c: {c}")

```
a: 1  
b: 2  
c: 3  
---  
a: 3  
b: 2  
c: 1
```

## مجموعه‌ها

«مجموعه‌ها» (sets) در پایتون دقیقاً مثل `set` در Java یا `HashSet` در C++ هستند.

مجموعه‌ها می‌توانن حداکثر یک نمونه از هر آیتم را نگهداری کنن. البته، هر چیزی که داخل مجموعه ذخیره می‌شود باید `Hashable` باشد.

این یعنی نمی‌توانیم انواع داده‌های تغییرپذیر مثل لیست‌ها، مجموعه‌های دیگه یا دیکشنری‌ها را در داخل یک مجموعه قرار بدم.

```
In [1]: # متغیرها  
# و متدهای اونها (Sets) مثال: کار با مجموعه‌ها
```

```
مجموعه‌ها با {} ساخته می‌شون و آیتم‌های تکراری را حذف می‌کنن.  
s = {1, 2, 3, 3, (1, 2), 'hasan'}
```

```
# --- ویژگی‌های مجموعه  
# می‌بینی که عدد 3 فقط یک بار تو خروجی هست.  
print(f"{s}")  
مجموعه‌ها بدون ترتیب هستن و طولشون بر اساس تعداد آیتم‌های غیرتکراری محاسبه می‌شون.  
print(f"تعداد آیتم‌ها در مجموعه {len(s)}")
```

```
# --- بررسی وجود آیتم  
print('بررسی وجود آیتم --')  
print(f"آیا 1 در s هست؟ {1 in s}")  
print(f"آیا 4 در s هست؟ {4 in s}")
```

```
# --- اضافه کردن آیتم  
# می‌توانی یک آیتم جدید به مجموعه اضافه کنی (add) با  
print('اضافه کردن --')  
s.add(4)  
print(f"مجموعه بعد از اضافه کردن {s}")  
print(f"آیا 4 در s هست؟ {4 in s}")
```

```
اگه سعی کنی آیتمی که وجود داره رو دوباره اضافه کنی، هیچ اتفاقی نمی‌افتد.  
s.add(1)  
print(f"بعد از اضافه کردن {s}")
```

```
# --- حذف آیتم  
# می‌توانی یک آیتم را حذف کنی (remove) با  
print('حذف --')  
s.remove(3)  
print(f"مجموعه بعد از حذف {s}")  
print(f"آیا 3 در s هست؟ {3 in s}")
```

```
می‌گیری KeyError اگه سعی کنی آیتمی که وجود نداره رو حذف کنی، خطای  
برای جلوگیری از خطأ، بهتره قبليس وجود آیتم رو بررسی کنی.  
try:  
    s.remove(10)  
except KeyError as e:  
    print(f"Error: {e}")
```

```
مجموعه [(2, 1), 3, 2, 1, 'hasan']  
تعداد آیتم‌ها در مجموعه: 5
```

```
-- بررسی وجود آیتم:  
در 1 هست؟ s آیا 1 در True  
در 4 هست؟ s آیا 4 در False  
  
-- اضافه کردن:  
4 {4, (2, 1), 3, 2, 1, 'hasan'}  
مجموعه بعد از اضافه کردن 4 هست؟ s آیا 4 در True  
1 {4, (2, 1), 3, 2, 1, 'hasan'}  
بعد از اضافه کردن 1  
  
-- حذف:  
3 {4, (2, 1), 2, 1, 'hasan'}  
مجموعه بعد از حذف 3 هست؟ s آیا 3 در False  
Error: 10
```

## دیکشنری

«دیکشنری‌ها» (یا به اختصار `dict`) در پایتون خیلی پرکاربرد و درست مثل `HashMap` در Java یا `map` در C++ عمل می‌کنن.

در یک `dict` شما می‌توانید زوج‌های مرتب «کلید و مقدار» را ذخیره کنید و بعداً بر اساس کلید، به مقدار مورد نظرتون دسترسی داشته باشید.

فقط یه نکته مهم اینه که کلیدها باید `Hashable` باشن. این یعنی نمی‌توانیم انواع داده‌های تغییرپذیر مثل لیست‌ها رو به عنوان کلید استفاده کنیم.

```
In [26]: # متغیرها  
# و متدهای اونها (Dictionaries) مثال: کار با دیکشنری‌ها
```

```
# --- ساخت دیکشنری  
# دیکشنری‌ها با استفاده از زوج‌های «کلید: مقدار» ساخته می‌شون.  
# می‌توانه کلید باشه (tuple) باشن. می‌بینی که رشته، عدد و چندتایی (immutable) کلیدها باید تغییرناپذیر  
d = {  
    'ali': 1,  
    'mamad': 2,  
    1: 3,  
    (1, 2): 4,
```

```

} --- دسترسی به مقدار
print(f"تعداد آیتم‌های دیکشنری {len(d)}")
print(f"دیکشنری {d}")

# با استفاده از کلید می‌توانی به مقدار دسترسی پیدا کنی
print(f"\n1 {d[1]}") --- حذف آیتم
# مقدار کلید 1 در دیکشنری هست؟
print(f"آیا کلید 1 در دیکشنری هست؟")
print(f"مقدار کلید 'mamad': {d['mamad']}") --- اضافه کردن آیتم
# برای کلیدهایی که چندتایی هستند، می‌توانی پرانتز را حذف کنی
print(f"مقدار کلید {d[(1, 2)]}") --- حذف آیتم
# می‌توانی یک زوج کلید: مقدار را از دیکشنری حذف کنی
print(f"\n-- حذف آیتم") --- حذف آیتم
del d[1] --- مواجه می‌شی KeyError حالا اگه به کلید 1 دسترسی پیدا کنی، با خطای
# این اگه کلید 1 در دیکشنری هست؟
print(f"آیا کلید 1 در دیکشنری بعد از حذف کلید 1 دیکشنری بعد از اضافه کردن آیتم")
print(f"1 {d}") --- اضافه کردن آیتم
# روش بهتر برای دسترسی این --- روش بهتر برای دسترسی این
# استفاده کن () وقتی کلید وجود نداره، از متده get() برای جلوگیری از خطای
# برمی‌گردونه، اگه کلید پیدا نشه، به جای خطای
print(f"\nnon_existent": {d.get('non_existent')}) --- هم همون نمایش رشته‌ای دیکشنری رو برمی‌گردونه
print(f"نمایش رشته‌ای دیکشنری {d.__str__()}") --- تعداد آیتم‌های دیکشنری: 4
{'ali': 1, 'mamad': 2, 1: 3, (1, 2): 4}

3 مقدار کلید 1: آیا کلید 1 در دیکشنری هست؟ True
2 مقدار کلید 'mamad': 2
4 مقدار کلید (1, 2): 4

-- حذف آیتم: آیا کلید 1 در دیکشنری هست؟ False
{'ali': 1, 'mamad': 2, (1, 2): 4} --- دیکشنری بعد از حذف کلید 1

-- اضافه کردن آیتم: {'ali': 1, 'mamad': 2, (1, 2): 4, 'new': 123}
{'ali': 1, 'mamad': 2, (1, 2): 4, 'new': 123} --- مقدار کلید 'non_existent': None
{'ali': 1, 'mamad': 2, (1, 2): 4, 'new': 123} --- نمایش رشته‌ای دیکشنری

```

## مقدار False و True و انواع پایه در گزاره‌های شرطی

در پایتون، مقادیر زیر در یک گزاره شرطی، مقدار False به حساب میان:

- False
- None
- 0j ، 0.0 ، 0
- صفر عددی: [] ، {} ، ("")
- یک دنباله (مثل رشته)، مجموعه یا دیکشنری خالی: () و غیره.

هر چیز دیگه‌ای (مثلاً یک عدد غیر صفر، یک رشته غیر خالی، یک لیست با آیتم و...) مقدار True می‌گیرد.

## حلقه‌ها

در پایتون ما دو حلقه‌ی for و while را داریم.

البته حلقه‌ی for یه کم با زبان‌های دیگه متفاوته. شما فقط می‌توانید روی یه «لیست» یا هر شیء «پیمایش‌شونده» (iterable) دیگه‌ای مثل رشته‌ها، چندتایی‌ها یا دیکشنری‌ها، حلقه بزنید.

یه نکته باحال دیگه اینکه حلقه‌ها می‌توان یه بخش else هم داشته باشن. این بخش زمانی اجرا می‌شه که حلقه بدون هیچ مشکلی (یعنی بدون دستور break) به پایان رسیده باشه.

In [27]:

```

# متغیرها
# مثال: حلقه while
cnt = 5

# --- حلقه while ---
# درست باشه، حلقه ادامه پیدا می‌کنه '0' تا زمانی که شرط
while cnt > 0:
    print(f"...شماره‌گذاری {cnt}") --- استفاده می‌کنیم 1 - = 1 با cnt = cnt - 1 برای کم کردن یک واحد، از عملگرهای
    # در پایتون کار نمی‌کنند cnt++ یا cnt-- یا دقت کنید که عملگرهای
    # cnt -= 1

    # این خط بعد از اتمام حلقه اجرا می‌شه
    print("...شمارش به پایان رسیده\n")

```

شماره‌گذاری  
5 ...  
4 ...  
3 ...  
2 ...  
1 ...

شمارش به پایان رسید.

In [28]:

```
# متغیرها
# و لیست‌ها روی range مثال: حلقه‌های

# --- روی range ---  
یک شیء تولید می‌کنند که می‌توانی روش حلقه بزنی (شروع، پایان، گام) تابع
# اینجا از عدد ۲ شروع می‌شوند، تا قبل از ۱۰ بیش می‌ردد و هر بار ۳ تا اضافه می‌شوند
for i in range(2, 10, 3):
    print(f"شماره: {i}")

# --- روی لیست for حلقه ---
نسبت می‌ده 'i' این روش، آیتم‌های لیست را یکی یکی به متغیر
l = ['asf', 'sajf', 'fsaf']
print("\n--- حلقه روی لیست ---")
for i in l:
    print(f"آیتم: {i}")

# --- حلقه for با enumerate() ---
برای دسترسی هم‌زمان به ایندکس و آیتم استفاده کنید (می‌توانی از تابع
print("\n--- enumerate ---")
for index, item in enumerate(l):
    print(f"ایندکس: {index}, آیتم: {item}")

شماره: 2
شماره: 5
شماره: 8

--- حلقه روی لیست ---
آیتم: asf
آیتم: sajf
آیتم: fsaf

--- enumerate ---
ایندکس: 0, آیتم
ایندکس: 1, آیتم
ایندکس: 2, آیتم
```

In [29]:

```
# متغیرها
# مثال: تابع enumerate()

l = [2, 5, 7, 3, 10]
print(f"لیست اصلی: {l}")

# --- enumerate() و خروجی اون ---
هستش یه شیء خاص برای گردونه که هر آیتم اون را (می‌توانید تابع
# شامل ایندکس (از ۰ شروع می‌شوند) و مقدار آیتم مربوطه است این
print(f"\n--- list(enumerate(l)) --- به صورت لیست (list(enumerate(l)))")

# --- در حلقه for استفاده از ---
هستش for این بهترین روش برای دسترسی به ایندکس و مقدار آیتم در حلقه
# داشته باشی (counter) دیگه نیازی نیست به متغیر جدا برای شمارش
print("\n--- enumerate() ---")
for index, value in enumerate(l):
    print(f"است شماره {index}, مقدار: {value}.")

لیست اصلی: [2, 5, 7, 3, 10]
```

به صورت لیست: [(10, 4), (3, 3), (7, 2), (5, 1), (2, 0)]

دسترسی به ایندکس و مقدار با ---  
آیتم شماره ۰، مقدارش ۲ است.  
آیتم شماره ۱، مقدارش ۵ است.  
آیتم شماره ۲، مقدارش ۷ است.  
آیتم شماره ۳، مقدارش ۳ است.  
آیتم شماره ۴، مقدارش ۱۰ است.

In [30]:

```
# متغیرها
# مثال: پیمایش روی دیکشنری‌ها (Iterating over Dictionaries)

d = {
    'ali': 1,
    'mamad': 2,
    1: 3,
    (1, 2): 4,
}

# --- پیمایش روی کلیدها ---
بزنی، روی کلیدهای پیمایش می‌کنند for به طور پیش‌فرض، اگه روی دیکشنری حلقه
print("\n--- پیمایش روی کلیدها ---")
for key in d:
    print(f"کلید: {key}, مقدار: {d[key]}")

# --- پیمایش روی آیتم‌ها ---
برای دسترسی هم‌زمان به کلید و مقدار استفاده کنید (می‌توانی از متند
print("\n--- پیمایش روی آیتم‌ها ---")
for key, value in d.items():
    print(f"کلید: {key}, مقدار: {value}.")

# --- پیمایش از چندتایی‌های است که هر کدام یک زوج (key, value) می‌باشد
print("\n--- list(d.items()) ---")
for key, value in d.items():
    print(f"کلید: {key}, مقدار: {value}")
```

--- پیمایش روی کلیدها  
کلید: ali، مقدار: 1  
کلید: mamad، مقدار: 2  
کلید: 3، مقدار: 3  
کلید: (1, 2)، مقدار: 4

--- پیمایش روی کلید و مقدار ---  
d.items(): [('ali', 1), ('mamad', 2), (1, 3), ((1, 2), 4)]  
کلید: ali، مقدار: 1  
کلید: mamad، مقدار: 2  
کلید: 1، مقدار: 3  
کلید: (1, 2)، مقدار: 4

```
In [31]: # متغیرها
# برای ترکیب چند لیست (مثال: zip() تابع)
# دو لیست با اطلاعات مرتبط
a = [9135233333, 9123222233, 9212312121]
b = ['iman', 'rasoul', 'fateme']

# --- zip() خروجی ---
# درمیاره tuple آیتم‌های دو لیست را هم ترکیب می‌کنه و به شکل چندتایی () تابع
# خروجی این تابع یک شریء قابل پیمایش است.
print(f"--- پیمایش روی لیست‌های ترکیب شده: {list(zip(a, b))}")

# --- zip() در حلقه for استفاده از ---
# این روش به شما اجازه می‌دهد روی دو یا چند لیست به صورت موازی پیمایش کنید.
print("\n--- پیمایش روی لیست‌های ترکیب شده:")
for x, y in zip(a, b):
    print(f"شماره تماس {x}, {y} است.")

# --- zip() خروجی لیست (9135233333, 'iman'), (9123222233, 'rasoul'), (9212312121, 'fateme')
```

--- پیمایش روی لیست‌های ترکیب شده ---  
است. iman، شماره تماس 9135233333.  
است. rasoul، شماره تماس 9123222233.  
است. fateme، شماره تماس 9212312121.

```
In [32]: # متغیرها
# مثال: پردازش متن و شمارش کلمات با استفاده از دیکشنری

mystr = """Affronting everything discretion men now own did. Still round match we to. Frankness pronounce daughters remainder extensive has but. Happiness c Paid was hill sir high. For him precaution any advantages dissimilar comparison few terminated projecting. Prevailed discovery immediate objection of ye at Was justice improve age article between. No projection as up preference reasonably delightful celebrated. Preserved and abilities assurance tolerably breakf Demesne far hearted suppose venture excited see had has. Dependent on so extremely delivered by. Yet ?no jokes worse her why. Bed one supposing breakfast da Kept in sent gave feel will oh it we. Has pleasure procured men laughing shutters nay. Old insipidity motionless continuing law shy partiality. Depending ac Answer misery adieus add wooded how nay men before though. Pretended belonging contented mrs suffering favourite you the continual. Mrs civil nay least mear Am no an listening depending up believing. Enough around remove to barton agreed regret in or it. Advantage mr estimable be commanded provision. Year well s Breakfast agreeable incommoded departure it an. By ignorant at on wondered relation. Enough at tastes really so cousin am of. Extensive therefore supported t Improved own provided blessing may peculiar domestic. Sight house has sex never. No visited raising gravity outward subject my cottage mr be. Hold do at tor Of on affixed civilly moments promise explain fertile in. Assurance advantage belonging happiness departure so of. Now improving and one sincerity intentior

# --- اصلاح متن ---
# یک رشته جدید برگردانه و رشته اصلی رو عوض نمی‌کنه (تابع replace).
# پس باید نتیجه رو دوباره به متغیر نسبت بدم.
# همچنین، می‌توییم برای تمیزکاری بهتر، متن رو به حروف کوچک تبدیل کنیم.
processed_str = mystr.replace('.', '').replace('?', '').lower()

# --- شمارش کلمات ---
# حالا متن رو به کلمات جداگانه تبدیل می‌کنیم.
d = {}
for word in processed_str.split():
    if word in d:
        d[word] += 1
    else:
        d[word] = 1

# --- نمایش نتیجه ---
print(f"تعداد کلمات مختلف: {len(d)}")
print(f"شمارش کلمات: {d}")

# --- مثال: چاپ کلمات پر تکرار ---
print("\n--- کلمه پر تکرار 10")
# می‌توانیم کلمات رو بر اساس تعداد تکرار مرتب کنیم، و items() و sort() را استفاده از مدد.
sorted_words = sorted(d.items(), key=lambda item: item[1], reverse=True)
for word, count in sorted_words[:10]:
    print(f'{word}: {count}' + "بار")
```

شمارش کلمات: {'affronting': 2, 'everything': 1, 'discretion': 1, 'men': 4, 'now': 5, 'own': 2, 'did': 1, 'still': 1, 'round': 1, 'match': 1, 'we': 4, 'to': 4, 'frankness': 2, 'pronounce': 1, 'daughters': 1, 'remainder': 2, 'extensive': 3, 'has': 6, 'but': 4, 'happiness': 2, 'cordially': 1, 'one': 6, 'determine': 1, 'concluded': 1, 'fat': 1, 'plenty': 1, 'season': 1, 'beyond': 1, 'by': 6, 'hardly': 1, 'giving': 2, 'of': 9, 'consulted': 1, 'or': 4, 'acuteness': 2, 'dejection': 2, 'an': 5, 'smallness': 1, 'if': 1, 'outward': 2, 'general': 1, 'passage': 1, 'another': 1, 'as': 3, 'it': 5, 'very': 1, 'his': 7, 'are': 1, 'come': 2, 'man': 1, 'walk': 1, 'next': 1, 'delighted': 1, 'prevailed': 2, 'supported': 2, 'too': 1, 'not': 3, 'perpetual': 1, 'who': 1, 'furnished': 2, 'nay': 5, 'bed': 2, 'projection': 3, 'compliment': 1, 'instrument': 1, 'paid': 1, 'was': 2, 'hill': 2, 'sir': 1, 'high': 1, 'for': 3, 'him': 3, 'precaution': 1, 'any': 2, 'advantages': 1, 'dissimilar': 1, 'comparison': 1, 'few': 1, 'terminated': 1, 'projecting': 1, 'discovery': 1, 'immediate': 1, 'objection': 1, 'ye': 2, 'at': 5, 'repair': 1, 'summer': 1, 'winter': 1, 'living': 1, 'feeble': 1, 'pretty': 1, 'in': 11, 'so': 4, 'sense': 1, 'am': 5, 'known': 1, 'these': 2, 'since': 1, 'shortly': 1, 'respect': 2, 'ask': 1, 'cousins': 1, 'brought': 1, 'add': 2, 'tedious': 1, 'expect': 1, 'relied': 1, 'do': 6, 'genius': 2, 'is': 7, 'on': 4, 'around': 2, 'spirit': 1, 'hearts': 1, 'raptures': 1, 'daughter': 1, 'branched': 1, 'laughter': 1, 'peculiar': 2, 'settling': 1, 'justice': 1, 'improve': 1, 'age': 2, 'article': 2, 'between': 1, 'no': 5, 'up': 4, 'preference': 1, 'reasonably': 1, 'delightful': 2, 'celebrated': 1, 'preserved': 1, 'and': 6, 'abilities': 1, 'assurance': 2, 'tolerably': 1, 'breakfast': 3, 'use': 2, 'saw': 1, 'painted': 2, 'letters': 1, 'forming': 1, 'far': 3, 'village': 1, 'elderly': 2, 'compact': 2, 'her': 6, 'rest': 1, 'each': 1, 'spot': 1, 'you': 2, 'knew': 1, 'estate': 1, 'gay': 1, 'wooded': 2, 'depart': 1, 'six': 1, 'be': 5, 'have': 1, 'lose': 1, 'gate': 1, 'bred': 1, 'separate': 1, 'removing': 1, 'expenses': 1, 'had': 3, 'covered': 1, 'evident': 1, 'chapter': 1, 'matters': 2, 'anxious': 1, 'demesne': 2, 'hearted': 2, 'suppose': 1, 'venture': 1, 'excited': 1, 'see': 3, 'dependent': 2, 'extremely': 1, 'delivered': 1, 'yet': 3, 'jokes': 1, 'worse': 1, 'why': 1, 'supposing': 1, 'day': 2, 'fulfilled': 1, 'off': 2, 'depending': 3, 'questions': 1, 'whatever': 1, 'boy': 1, 'exertion': 1, 'extended': 1, 'ecstatic': 2, 'followed': 1, 'handsome': 2, 'drawings': 1, 'entirely': 1, 'mrs': 4, 'outweigh': 1, 'acceptance': 1, 'insipidity': 3, 'remarkably': 1, 'invitation': 2, 'kept': 1, 'sent': 2, 'gave': 1, 'feel': 1, 'will': 1, 'oh': 3, 'pleasure': 1, 'procured': 1, 'laughing': 1, 'shutters': 1, 'old': 2, 'motionless': 1, 'continuing': 2, 'law': 2, 'shy': 2, 'partiality': 1, 'eat': 1, 'unpleasing': 1, 'astonished': 1, 'discovered': 1, 'nor': 1, 'morning': 1, 'met': 1, 'beloved': 1, 'evening': 1, 'upon': 1, 'last': 1, 'here': 1, 'must': 1, 'answer': 1, 'misery': 1, 'adieu': 1, 'how': 2, 'before': 1, 'though': 1, 'pretended': 1, 'belonging': 2, 'contented': 2, 'suffering': 2, 'favourite': 1, 'the': 1, 'continual': 1, 'civil': 1, 'least': 1, 'means': 1, 'tried': 1, 'drift': 1, 'natural': 1, 'end': 1, 'whether': 1, 'towards': 1, 'certain': 1, 'unfeeling': 1, 'sometimes': 1, 'promotion': 1, 'quitting': 1, 'informed': 1, 'concerns': 2, 'can': 1, 'conviction': 1, 'uncommonly': 1, 'appetite': 1, 'opinions': 1, 'hastened': 1, 'admitted': 2, 'listening': 1, 'believing': 1, 'enough': 2, 'remove': 1, 'barton': 1, 'agreed': 1, 'regret': 1, 'advantage': 3, 'mr': 3, 'estimable': 2, 'commanded': 2, 'provision': 1, 'year': 1, 'well': 1, 'shot': 1, 'deny': 1, 'shew': 1, 'shall': 1, 'downs': 1, 'stand': 1, 'marry': 1, 'taken': 1, 'out': 1, 'related': 1, 'account': 1, 'brandon': 1, 'wrong': 1, 'never': 2, 'ready': 2, 'ham': 1, 'witty': 1, 'our': 1, 'compass': 1, 'uncivil': 1, 'weather': 1, 'forbade': 1, 'minutes': 1, 'truth': 1, 'son': 1, 'new': 1, 'under': 1, 'agreeable': 1, 'incommode': 1, 'departure': 2, 'ignorant': 1, 'wondered': 1, 'relation': 1, 'tastes': 1, 'really': 1, 'cousin': 1, 'therefore': 1, 'extremity': 1, 'pursuit': 1, 'invited': 1, 'view': 1, 'she': 1, 'roof': 1, 'tell': 1, 'case': 2, 'sigh': 1, 'moreover': 1, 'possible': 1, 'he': 3, 'sociable': 1, 'cold': 1, 'less': 1, 'been': 1, 'hard': 1, 'improved': 1, 'provided': 1, 'blessing': 1, 'may': 1, 'domestic': 1, 'sight': 1, 'house': 1, 'sex': 1, 'visited': 1, 'raising': 1, 'gravity': 1, 'subject': 1, 'my': 2, 'cottage': 2, 'hold': 1, 'tore': 1, 'park': 1, 'feet': 1, 'near': 1, 'understood': 1, 'occasional': 1, 'sentiments': 1, 'inhabiting': 1, 'melancholy': 1, 'alteration': 1, 'principles': 1, 'speedily': 1, 'kindness': 1, 'properly': 1, 'offices': 1, 'parlors': 1, 'affixed': 1, 'civilly': 1, 'moments': 1, 'promise': 1, 'explain': 1, 'fertile': 1, 'improving': 1, 'sincerity': 1, 'intention': 1, 'allowance': 1, 'necessary': 1, 'earnestly': 2, 'five': 1, 'wife': 1, 'gone': 1, 'sportsmen': 1, 'afford': 1, 'parish': 1, 'settle': 1, 'easily': 1, 'garret': 1}

--- کلمه پر تکرار 10 ---

بار 11  
بار 9  
بار 7  
بار 7  
بار 6  
بار 6

## Import

یکی از قوی‌ترین جنبه‌های زبان پایتون، کتابخانه‌ها و مازول‌های آماده‌ای هستند که برای انجام تقریباً هر کاری نوشته شده‌اند. برای استفاده از این کتابخانه‌ها، ابتدا باید آن‌ها را روی کامپیوتر خود نصب کنید. رایج‌ترین و بهترین ابزار برای این کار، pip است:

• با استفاده از دستور pip

pip install <library-name>==<version>

(توجه: روش‌هایی مثل setup.py یا easy\_install قدیمی شده‌اند و امروزه استفاده از pip برای نصب پکیج‌ها بهترین روش است.)

حالا می‌توانید از کتابخانه‌ی نصب شده استفاده کنید. برای این کار باید آن را Import (شبیه include در C++) کنید. برای این کار چند روش وجود دارد:

• روش اول: import کردن کل مازول

```
import urllib.request as urllib
res = urllib.urlopen('http://shiraz.iau.ir/en')
```

این روش مازول را با یک نام مستعار (alias) وارد می‌کند.

• روش دوم: import کردن تابع یا کلاس‌های خاص از مازول

```
from urllib.request import urlopen
# برای وارد کردن همه توابع
from urllib.request import *
res = urlopen('http://shiraz.iau.ir/en')
```

این روش به شما اجازه می‌دهد از تابع به طور مستقیم و بدون نام مازول استفاده کنید.

بهتر است معمولاً فقط از یکی از این دو روش استفاده کنیم تا کد خوانا و منظم باقی بماند.

## توابع

توابع در پایتون با کلمه‌ی کلیدی def تعریف می‌شون. برای ورودی دادن به تابع، می‌توانید از همون روش ساده‌ی ورودی‌های مرتب استفاده کنید. اما می‌توانید اون‌ها را با استفاده از نامشون هم مقداردهی کنید.

یه قابلیت خیلی خوب دیگه اینه که می‌توانید تابع رو مثل بقیه اشیا در متغیرها ذخیره کنید یا به تابع دیگه پاس بدید.

همچنین می‌توانید در داخل یه تابع، یه تابع دیگه تعریف کنید یا تابع «بی‌نام» (anonymous function) داشته باشید. این تابع با کلمه‌ی کلیدی lambda تعریف می‌شون

In [34]:

```
# دستورات شرطی
# مثال: تعریف و فراخوانی تابع
import math

# --- تابع isPrime ---
```

# این تابع یک ورودی می‌گیره و بررسی می‌کنه که آیا اول عدد اول هست یا نه.

# # ورودی اختیاری 'b=10' ورودی

def isPrime(a, b=10):

    اگر عدد 1 باشه، اول نیست

    if a == 1:

        return False

    تا ریشه دوم عدد، اول بودن رو بررسی می‌کنیم # حلقه.

    for i in range(2, int(math.sqrt(a)) + 1)):

        if a % i == 0:

            return False

    return True

# # فراخوانی تابع با ورودی موقعیتی #

print(f"آیا عدد 12 اول است؟")

# # فراخوانی تابع با ورودی کلیدوازه‌ای #

print(f"آیا عدد 17 اول است؟")

# --- تابع با ورودی‌های مختلط (positional & keyword) ---

def func(value1, aa, bb):

    print(f"\nمتغیر value1: {value1}")

    print(f"متغیر aa: {aa}")

    print(f"متغیر bb: {bb}")

داده می‌شن (value1 و aa) و بقیه به صورت کلیدوازه‌ای (value1) ورودی 10 به صورت موقعیتی #

در این روش، ترتیب ورودی‌های کلیدوازه‌ای مهم نیست.

func(10, bb=2 + 3j, aa='salam')

آیا عدد 12 اول است؟

True

متغیر value1: 10

متغیر aa: salam

متغیر bb: (2+3j)

In [35]: # دستورات شرطی #

# مثال: توابع به عنوان اشیا و توابع Lambda

# --- تابع applyFunction ---

است (higher-order function) این یک تابع سطح بالا

رو به عنوان ورودی می‌گیره ('f') چون یک تابع دیگه.

def applyFunction(f, value):

    return f(value)

# --- یک تابع معمولی ---

این تابع یک عدد رو به توان دو می‌رسونه.

def sqr(a):

    return a \* a

print(f" خروجی تابع با applyFunction می‌باشد: {applyFunction(sqr, 10)}")

# استفاده از تابع Lambda ---

این همون کار رو انجام می‌ده، اما به جای تعریف یک تابع کامل

استفاده شده که کد رو کوتاهتر می‌کنه (Lambda) از یک تابع بینام.

کاری است که تابع انجام می‌دهد 'x \* 2' ورودی است و '2' ورودی دیگه است.

print(f" خروجی تابع با lambda: {applyFunction(lambda x: 2 \* x, 10)}")

# --- با چند ورودی Lambda مثال ---

می‌توان چند ورودی هم داشته باشند تابع

add = lambda a, b: a + b

print(f" با چند ورودی lambda: {add(5, 7)}")

خروجی تابع با applyFunction می‌باشد: 100

خروجی تابع با lambda: 20

با دو ورودی: 12

In [37]: # دستورات شرطی #

# برای تغییر نوع ورودی Lambda مثال: استفاده از

از قبل تعریف شده و ورودی عددی می‌گیره isPrime نکنه: تابع

مواجه می‌شی # string. مواجهه می‌شی TypeError اگه بپش يه رشته

تبدیل می‌کنه int یه تابع بینام می‌سازیم که اول ورودی رو به Lambda با استفاده از

پاس می‌ده isPrime و بعد اون رو به تابع

isStrPrime = lambda x: isPrime(int(x))

print(f" آیا عدد 12 اول است؟ {isStrPrime('12')}")

print(f" آیا عدد 17 اول است؟ {isStrPrime('17')}")

آیا عدد 12 اول است؟

True

In [38]: # متغیرها

# مثال: خواندن داده‌ها از ورودی کاربر با

# --- مثال: خواندن لیست اعداد صحیح از ورودی ---

این خط به صورت مرحله‌ای کار می‌کنه #

# 1. input(): یک خط کامل از ورودی کاربر رو می‌خونه.

اوون خط رو بر اساس فاصله به چند رشته تبدیل می‌کنه: # 2. .split():

رو روی هر کدوم از اوون رشته‌ها اعمال می‌کنه تا به عدد صحیح تبدیل بشن int تابع: # 3. map(int, ...):

رو به یک لیست تبدیل می‌کنه map خروجی: # 4. list(...):

print(" --- خواندن اعداد از ورودی ---")

print("(30 20 10) چند عدد را با فاصله وارد کنید (مثال: 10 20 30)")

print(list(map(int, input().split()))))

# استفاده کردیم تا هر عدد ورودی رو بگیریم Lambda و map اینجا از

و بررسی کنیم که آیا اول هست با نه.

print("\n--- برسی اعداد اول با ---")

# 'isStrPrime' بود که قبلاً تعریف کردیم Lambda یک تابع

```
print("لطفاً چند عدد را با فاصله وارد کنید (مثال: 12 17 5)")  
print(list(map(lambda x: isPrime(int(x)), input().split())))
```

--- خواندن اعداد از ورودی  
لطفاً چند عدد را با فاصله وارد کنید (مثال: 10 20 30)  
[1, 2, 3]

--- بررسی اعداد اول با lambda ---  
لطفاً چند عدد را با فاصله وارد کنید (مثال: 12 17 5)  
[False, True, False]

In [ ]: `help(str.split)`  
# میتوانید نحوه کارکرد آن متوا را بینید Help با دستور

Help on method\_descriptor:

```
split(self, /, sep=None, maxsplit=-1)  
    Return a list of the substrings in the string, using sep as the separator string.
```

`sep`  
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

`maxsplit`  
Maximum number of splits (starting from the left).  
-1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

تابع sort لیستها را یادتان می‌آید؟ ما می‌توانیم ترتیب مرتب کردن این توابع را با استفاده از پارامترهای خاصی که می‌گیرند، تغییر دهیم و بر اساس قوانین دلخواه خودمان مرتبسازی را انجام دهیم.

In [39]: # متغیرها  
# مثال: مرتبسازی سفارشی با پارامتر key

```
l = ['1', '6', '2', '4', '10']  
print(f"لیست اولیه: {l}")
```

--- مرتبسازی پیشفرض رشته‌ای ---  
مرتبسازی کنی، پایتون رشته‌ها رو به صورت الفبایی مرتب می‌کنه key بدون # اگه بدون 1.sort()  
print(f"1: مرتبسازی پیشفرض (بر اساس رشته)\n{l}")

# مرتبسازی سفارشی با تابع ---  
ما می‌خوایم مرتبسازی بر اساس مقدار عددی باشه، نه رشته‌ای  
# می‌گه چطور مقایسه کنه sort می‌نویسیم که به 'key' برای این کار، به تابع def compareInt(x):  
این تابع هر رشته رو به عدد صحیح تبدیل می‌کنه  
return int(x)

```
l.sort(key=compareInt)  
print(f"\n: مرتبسازی سفارشی (بر اساس مقدار عددی){l}")
```

# --- Lambda برای key استفاده از ---  
هم استفاده کنی که کد رو کوتاهتر می‌کنه Lambda می‌توونی به جای تعریف یک تابع کامل، از 1 = ['1', '6', '2', '4', '10']  
l.sort(key=lambda x: int(x))  
print(f" مرتبسازی با lambda: {l}")

لیست اولیه: ['1', '6', '2', '4', '10']

مرتبسازی پیشفرض (بر اساس رشته): ['1', '2', '4', '6', '10']

مرتبسازی سفارشی (بر اساس مقدار عددی): ['1', '2', '4', '6', '10']  
مرتبسازی با lambda: ['1', '2', '4', '6', '10']

In [40]: # دستورات شرطی  
# مثال: تابع درونی و کلوژرهای (Closures)

# --- تابع ساده ضرب ---  
def mult(a, b):  
 return a \* b

# --- تابع درونی (Partial Function) ---  
این تابع یک ورودی می‌گیره و یک تابع جدید برمی‌گردونه  
# می‌گن (closure) «به این الگوریتم، «کلوژر».

def partialMult(a):  
 # mult تابع بیرونی به یاد داره 'a' درونی، مقدار ضرب می‌کند.  
 def mult(b):  
 return a \* b  
 return mult

یک تابع جدید هست که همیشه ورودی رو در 3 ضرب می‌کنه حالا multBy3 = partialMult(3)

```
print(f"نوع multBy3: {type(multBy3)}")  
print(f"خروجی multBy3(2): {multBy3(2)}")  
print(f"خروجی multBy3(11): {multBy3(11)}")  
print(f"خروجی multBy3(4): {multBy3(4)}")
```

می‌توونی به تابع دیگه هم بسازی که همیشه در 5 ضرب کنه  
multBy5 = partialMult(5)  
print(f"\nخرجی multBy5(4): {multBy5(4)}")

```
نوع multBy3: <class 'function'>
خروجی multBy3(2): 6
خروجی multBy3(11): 33
خروجی multBy3(4): 12

خروجی multBy5(4): 20
```

```
In [41]: #متغیرها
# برای ساخت توابع جزئی functools.partial مثال: استفاده از

from functools import partial

# قبلاً به این شکل تعریف شده mult فرض می‌کنیم تابع:
# def mult(a, b):
#     return a * b

# اون همیشه 5 باشه 'b' یه تابع جدید می‌سازیم که ورودی حالتاً باید partial باشد
anotherMultBy5 = partial(mult, b=5)
print(f" نوع anotherMultBy5: {type(anotherMultBy5)}")
# رو بھش بدی 'a' رو صدا می‌زنی، فقط کافیه ورودی anotherMultBy5 حالتاً وقتی
print(f" خروجی anotherMultBy5(a=4): {anotherMultBy5(a=4)}")
```

نوع anotherMultBy5: <class 'functools.partial'>
خروچی anotherMultBy5(a=4): 20

## دستور Help

دستور help() یکی از مهمترین ابزارها برای یادگیری و کار با پایتونه. اگه اسم یه تابع، کلاس یا متدها داخل help() بذاری، می‌تونی توضیحات کامل، ورودیها و مثالهای استفاده از اون رو ببینی. این کار بهت کمک می‌کنه بدون اینکه لازم باشه اینترنت رو بگردی، همه‌چیز رو سر و سامون بدی.

```
In [42]: help()

Welcome to Python 3.13's help utility! If this is your first time using
Python, you should definitely check out the tutorial at
https://docs.python.org/3.13/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To get a list of available
modules, keywords, symbols, or topics, enter "modules", "keywords",
"symbols", or "topics".

Each module also comes with a one-line summary of what it does; to list
the modules whose name or summary contain a given string such as "spam",
enter "modules spam".

To quit this help utility and return to the interpreter,
enter "q", "quit" or "exit".

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)". Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

```
In [43]: #متغیرها
# و تأثیر کلیدوازه global مثال: متغیرهای سراسری

s = 100 # این یک متغیر سراسری

def g():
    کار کنه 's' به تابع می‌گیم که با متغیر سراسری global با کلیدوازه.
    global s
    s = s + 2
    print(f" در تابع 's' مقدار {s}")

def f():
    رو تغییر بدی 's' اینجا هم به تابع می‌گیم که متغیر سراسری.
    global s
    s = 12
    print(f" در تابع 's' مقدار {s}")

# --- فراخوانی تابع ---
print("--- شروع فراخوانی ---")
g() # s=100 تبدیل به s=102 می‌شنه
g() # s=102 تبدیل به s=104 می‌شنه
f() # s=104 تبدیل به s=12 می‌شنه
g() # s=12 تبدیل به s=14 می‌شنه
f() # s=14 تبدیل به s=12 می‌شنه.

print(f" بعد از همه فراخوانی‌ها 's' مقدار نهایی \n{s}")
```

--- استفاده نکنیم، چه اتفاقی می‌افته؟ global و اگر از --- سراسری (Local) رو نداری، بایتون فکر می‌کنه می‌خواهی به متغیر محلی s اگه مواجه می‌شی UnboundLocalError و برای همین، با خطای # def h():
# s = s + 2
# print(s)
# h() # این خط خطای می‌ده

--- شروع فراخوانی ---

در تابع 's' مقدار g(): 102  
در تابع 's' مقدار g(): 104  
در تابع 's' مقدار f(): 12  
در تابع 's' مقدار g(): 14  
در تابع 's' مقدار f(): 12

بعد از همه فراخوانی‌ها: 12 's' مقدار نهایی

بعضی از متدها (توابع) هر کلاس، نامهای خاصی دارن که با دو کاراکتر زیرخط (`_`) در ابتداء و انتهای شروع می‌شن. این توابع به تابع «خاص» یا «متدهای جادویی» (magic methods) معروف هستند. مثلًا:

- `__init__`: که کار همون `constructor` در زبان‌های دیگه رو انجام می‌ده.
- `__del__`: (destructor)
- `__eq__`: برای مقایسه‌ی برابری (مثل `( == )` در `Java`)
- `__str__`: برای تبدیل شیء به رشته (مثل `toString` در `Java`)

همچنانی، یکی از تفاوت‌های اصلی پایتون با زبان‌های دیگه اینه که ما متغیر خاصی به نام `this` نداریم. به جای اون، خود شیء به عنوان اولین ورودی به هر متده کلاس پاس داده می‌شه. این ورودی چیز خاصی نیست، اما در پایتون ازش با نام `self` یاد می‌شه و بهتره شما هم همیشه همین قاعده رو رعایت کنید. بنابراین، هر متده یک کلاس حداقل یک ورودی داره که اولینش همون `self` است.

In [44]: # متغیرها

```
# مثال: کلاس‌ها، متدهای خاص و متغیرهای کلاس در برابر متغیرهای نمونه
# --- تعريف کلاس
class Test:
    # 'a' یک متغیر کلاس (class variable). است.
    # این متغیر بین تمام نمونه‌های کلاس مشترک است.
    a = 0

    # همان سازنده __init__ (constructor).
    # self شیء فعلی اشاره می‌کنه در this مثل (C++).
    def __init__(self, x, y):
        self.x = x
        self.y = y
        # به جای تغییر متغیر نمونه، متغیر کلاس رو افزایش می‌دم
        # این کار تعداد کل نمونه‌هایی که ساخته شده رو می‌شمره
        Test.a += 1
        print(f"سلام، من یک شیء جدید هستم. تعداد کل اشیا {Test.a}")

    # همان مخرب __del__ (destructor).
    # زمانی که شیء از حافظه پاک می‌شه، فراخوانی می‌شه (که قطعی نیست)
    # 'otherName' convention می‌تونه هر اسمی باشه، ولی
    def __del__(self):
        print(f"پاک شد {self.x} من باید برم، خدا حافظ! شیء")

# --- فراخوانی و تست
# ساخت اولین نمونه
print(f"ساخت نمونه اول ---")
t = Test(2, 3)
print(f"مقدار شمارنده کلاس بعد از ساخت {t.a}")

# ساخت دومین نمونه
print(f"\n--- (s) ساخت نمونه دوم")
s = Test(1, 2)
# 's.a' هردو به متغیر کلاس 't.a' و اشاره دارن
print(f"s: {s.a}")
print(f"مقدار شمارنده کلاس بعد از ساخت {s.a}")
print(f"مقدار شمارنده کلاس از {t.a}")

# یک ارجاع به اون شیء حذف می‌شه، t کردن None با
# اون رو پاک می‌کنه garbage collector، وقتی هیچ ارجاعی به شیء نباشه
print(f"\n--- t --- حذف ارجاع")
t = None
# فراخوانی می‌شه __del__ می‌شه، تابع وقتی t=None
# رو دوباره چاپ می‌کنیم تا بینیم تغییری کرده یا نه حالا
print(f"مقدار شمارنده کلاس بعد از حذف {s.a}")

--- (t) ساخت نمونه اول
سلام، من یک شیء جدید هستم. تعداد کل اشیا: 1
1: مقدار شمارنده کلاس بعد از ساخت

--- (s) ساخت نمونه دوم
سلام، من یک شیء جدید هستم. تعداد کل اشیا: 2
2: مقدار شمارنده کلاس بعد از ساخت
2: مقدار شمارنده کلاس از

--- t --- حذف ارجاع
--- من باید برم، خدا حافظ! شیء 2 پاک شد
2: مقدار شمارنده کلاس بعد از حذف
```

## فیلدها

هر کلاس می‌تونه تعدادی «فیلد» (یا همون متغیر) داشته باشه. در پایتون، شما می‌تونید در هر لحظه از کد به یک شیء، متغیر جدیدی رو اضافه کنید (مگر اینکه عمدًا این قابلیت ازش گرفته شده باشه).

اما بهتره که تمام فیلد‌های یک کلاس رو داخل «`__init__`» (constructor) تعریف کنید و مقدار اولیه بهشون بدید. این کار باعث می‌شه کد شما خیلی تمیزتر و قابل فهم‌تر باشه.

In [45]: # متغیرها

```
# مثال: متغیرهای کلاس در مقابل متغیرهای نمونه
# --- تعريف کلاس
class Test:
    # 'a' یک متغیر کلاس است. این متغیر بین تمام نمونه‌ها مشترک است.
    a = 1

    # تعریف می‌شن، مخصوص هر نمونه هستن self. متغیرهایی که با
    def __init__(self, b, c):
        # این بهترین روش برای تعریف متغیر نمونه است
        self.b = b
        self.c = c
```

```

    self.other_value = 10
--- ساخت نمونه و اضافه کردن متغیر ---
o = Test(2, 3)

در پایتون، می‌توانی حتی بعد از ساختن شیء هم به اون متغیر جدید اضافه کنی # 
# شناخته می‌شه و معمولاً توصیه نمی‌شه 'monkey patching' این کار به عنوان o.d = 4

حالا می‌توانیم به تمام این متغیرها دسترسی داشته باشیم #
# هم از طریق نمونه قابل دسترسه 'o' می‌بینی که print(f"مقدار متغیرها:{o.a, o.b, o.c, o.d, o.other_value}")

```

مقدار متغیرها: (1, 2, 3, 4, 10)

## وراثت (Inheritance)

در پایتون، می‌توانید از قابلیت «وراثت» استفاده کنید. برای این کار کافیه نام کلاس والد رو در یک جفت پرانتز جلوی تعریف کلاس فرزند قرار بگیرد (به مثال نگاه کنید).

پایتون مثل C++ این امکان را می‌ده که یک کلاس از چند کلاس به صورت همزمان ارث ببره، اگرچه شاید در پروژه‌های ساده‌تر خیلی کاربرد نداشته باشد.

در وراثت، گاهی مجبور می‌شیم متد مشابهی را از کلاس والد صدا بزنیم (مثلاً در constructor). برای این کار از تابع super() استفاده می‌کنیم.

برای مثال، برای صدا زدن constructor والد، کد زیر را می‌نویسیم:

```

In [46]: # متغیرها
# مثال: وراثت و استفاده از super()

ارث می‌بره 'object' از کلاس اصلی A کلاس #
# به صورت پیش‌فرض همین معنی را می‌ده 'class A:'، در پایتون ۳
class A(object):
    def __init__(self):
        print('در داخل ' + A)

class B(A):
    def __init__(self):
        print('در داخل ' + B)

رو صدا می‌زنیم (A) از کلاس والد __init__ متد، با استفاده از #
# این بهترین روش برای صدا زدن متدهای والد هست #
# کار می‌کنی (multiple inheritance) مخصوصاً وقتی با وراثت چندگانه #
super().__init__()

--- فرآخوانی و خروجی ---
# صدا زده می‌شه B از __init__ ابتدا، از کلاس 'b' با ساختن شیء #
# هم اجرا می‌شه A از __init__، __init__ و بعد از اون، به خاطر #
b = B()

# در داخل B
# در داخل A

```

## مطلوب اضافی

لیست بی‌نهایت؟ چرا که نه!

در پایتون، ما می‌توانیم با استفاده از «تولیدکننده‌ها» (generators) یک توالی (sequence) ایجاد کنیم که در ظاهر بی‌نهایت باشد، بدون اینکه بخواهد کل اون توالی را در حافظه نگه داره. این کار با کلمه کلیدی yield انجام می‌شود.

این روش در مواقعی که با داده‌های خیلی بزرگ یا بی‌نهایت سروکار داریم، بسیار کارآمد است.

```

In [47]: # متغیرها
# مثال: تولیدکننده‌ها (Generators)

--- تابع تولیدکننده ---
# است (generator) «این تابع یک تولیدکننده»
# به جای اینکه همه مقدار را یکجا برگردانه، با استفاده از
# فرآخوانی می‌شه، یک مقدار جدید تولید می‌کنی for هر بار که در حلقه
def InfiniteList(start):
    value = start
    while True:
        yield value
        value = value + 1

--- استفاده از تولیدکننده ---
# است generator این یک مثال عملی از استفاده از
expectedSum = 123
cnt = 0
for i in InfiniteList(1):
    cnt += i
    if cnt >= expectedSum:
        print(f"می‌باشد {i}، بیشتر است {expectedSum}. اولین عددی که مجموع اعداد ۱ تا آن از")
        break

اولین عددی که مجموع اعداد ۱ تا آن از 123 بیشتر است، 16 می‌باشد

```

```

In [48]: # متغیرها
# مثال: ترکیب تولیدکننده‌ها (Generators) و itertools

نياز دارد itertools برای اجرای این کد به کتابخانه #
import itertools

```

```

--- ترکیب توابع ---
# این خط کد به صورت زنجیره‌ای و از داخل به بیرون کار می‌کنی
# یک تولیدکننده بی‌نهایت از عدد 123 به بعد می‌سازه: # 1. InfiniteList(123)

```

از اون لیست بی‌نهایت، تا زمانی که عدد از 150 کمتر باشد، آیتم‌ها را برمند داره: `InfiniteList(123))`:  
از بین آیتم‌های باقی‌مانده، فقط اون‌هایی را نگه می‌داره که زوج نباشن (باقی‌مانده تقسیم بر 2 برابر با 1 باشد):  
نتیجه نهایی را به یک لیست تبدیل می‌کنه:

```
print(  
    f"lisit اعداد فرد بین 123 و 150: {list(filter(lambda x: x % 2 == 1, itertools.takewhile(lambda x: x < 150, InfiniteList(123))))}"  
)
```

لیست اعداد فرد بین 123 و 150: [149, 147, 145, 143, 141, 139, 137, 135, 133, 131, 129, 127, 125, 123]

In [49]: # متغیرها  
مثال: ساخت دنباله فیبوناچی با استفاده از Generator

--- تابع تولیدکننده دنباله فیبوناچی ---  
هست که اعداد فیبوناچی را به صورت نامحدود تولید می‌کنه Generator این تابع بک  
فراخوانی بشه for اما فقط وقتی یک عدد تولید می‌شه که در حلقه.

```
def FibonacciSeq():  
    a, b = 0, 1  
    while True:  
        # رو برگردانه 'b' باعث می‌شه تابع متوقف بشه و مقدار  
        # وقتی دوباره صدا زده بشه، از همین نقطه ادامه می‌ده  
        yield b  
        # برای جایگزینی هوشمندانه مقادیر استفاده شده اینجا از tuple unpacking.  
        # این کار یه راه تمیز برای محاسبه عدد بعدی فیبوناچی  
        a, b = b, a + b
```

--- در حلقه استفاده از Generator for ---  
ترکیب می‌کنیم `range(5)` نامحدود رو با zip(), generator با این باعث می‌شه حلقه فقط 5 بار اجرا بشه و در نهایت متوقف بشه  
for i, fib in zip(range(5), FibonacciSeq()):  
 print(f"آست {i}, عدد فیبوناچی شماره {fib}")

عدد فیبوناچی شماره 0، 1 است.  
عدد فیبوناچی شماره 1، 1 است.  
عدد فیبوناچی شماره 2، 2 است.  
عدد فیبوناچی شماره 3، 3 است.  
عدد فیبوناچی شماره 4، 5 است.

## دربگیرندها (Decorators)

«دربگیرندها» (Decorators) یا Wrappers توابع خاصی هستن که به شما اجازه می‌دن عملکرد یه تابع دیگه رو بدون اینکه کد اصلی اون رو تغییر بدید، گسترش بدید.

با استفاده از دربگیرندها، می‌تونیم کارهایی مثل اضافه کردن لگ (logging)، بررسی دسترسی یا اندازه‌گیری زمان اجرای یه تابع رو به سادگی انجام بدیم. برای استفاده از اون‌ها، کافیه نام دربگیرنده رو با علامت @ قبیل از تعریف تابع اصلی قرار بدیم.

In [50]: # متغیرها  
مثال: تابع بازگشتی فیبوناچی و محاسبه زمان اجرا

```
import time  
  
--- تابع بازگشتی فیبوناچی ---  
# عدد فیبوناچی رو حساب می‌کنه (recursive) این تابع به صورت بازگشتی  
# اما این روش خیلی ناکارآمد است چون برای هر عدد، چند بار محاسبات تکراری انجام می‌ده  
def fib(n):  
    if n <= 1:  
        return 1  
    return fib(n - 1) + fib(n - 2)  
  
# --- محاسبه زمان اجرا ---  
# رو برای چند عدد بزرگ حساب می‌کنه fib(n) این حلقه  
# و زمان اجرای هر کدام را اندازه می‌گیره  
for i in range(30, 35):  
    start = time.time()  
    value = fib(i)  
    # زمان اجرا به شکل تصاعدی زیاد می‌شه، می‌بینی که حتی با افزایش کوچک عدد  
    # آست {value} برابر با {i} است عدد فیبوناچی شماره {i}.  
    print(f"آست {value} عدد فیبوناچی شماره {i}." )  
    # زمان محاسبه  
    print(f"زمان محاسبه {time.time() - start:.2f}\n")
```

عدد فیبوناچی شماره 30 برابر با 1346269 است.  
زمان محاسبه: 0.22 ثانیه

عدد فیبوناچی شماره 31 برابر با 2178309 است.  
زمان محاسبه: 0.25 ثانیه

عدد فیبوناچی شماره 32 برابر با 3524578 است.  
زمان محاسبه: 0.47 ثانیه

عدد فیبوناچی شماره 33 برابر با 5702887 است.  
زمان محاسبه: 0.63 ثانیه

عدد فیبوناچی شماره 34 برابر با 9227465 است.  
زمان محاسبه: 1.10 ثانیه

In [51]: # متغیرها  
مثال: بهینه‌سازی تابع بازگشتی با استفاده از Decorator Memoization

```
import time  
import collections  
  
# --- تابع دربرگیرنده برای Memoization ---  
# رو می‌سازه و یک تابع جدید برمند گردانه، این تابع  
# memoizeDict را ذخیره می‌کنه تا دوباره محاسبه نشین.  
def memoize(func):  
    memoizeDict = {}  
    def wrapper(*args):  
        # مثلاً هستن با نه Hashable بررسی می‌کنیم که آیا ورودی‌ها  
        # نتایج محاسبه شده رو ذخیره می‌کنه  
        if not isinstance(args, collections.abc.Hashable):  
            return func(*args)
```

```

اگه نتیجه قبلاً تو دیکشنری ذخیره شده، همون رو برمنیگردونیم #.
elif args in memoizeDict:
    return memoizeDict[args]
اگه نه، تابع اصلی رو اجرا میکنیم، نتیجه رو ذخیره میکنیم و برمنیگردونیم #.
else:
    result = func(*args)
    memoizeDict[args] = result
    return result
return wrapper

@memoize
def fib_memoized(n):
    if n <= 1:
        return 1
    return fib_memoized(n - 1) + fib_memoized(n - 2)

# --- مقایسه زمان اجرا ---
for i in range(30, 35):
    start = time.time()
    value = fib_memoized(i)
    میبینی که زمان محاسبه این بار بسیار ناچیزه #.
    # ذخیره شدن memoizeDict چون نتایج قبلاً در #.
    print(f"--- ابتدا {i} عدد فیبوناچی شماره {value} برابر با {i} است")
    print(f"--- زمان محاسبه: {time.time() - start:.4f} ثانیه \n")
print(f"تابع fib_memoized.memoizeDict: {fib_memoized.__closure__[0].cell_contents}")

```

عدد فیبوناچی شماره 30 برابر با 1346269 است.

زمان محاسبه: 0.0004 ثانیه

عدد فیبوناچی شماره 31 برابر با 2178309 است.

زمان محاسبه: 0.0000 ثانیه

عدد فیبوناچی شماره 32 برابر با 3524578 است.

زمان محاسبه: 0.0000 ثانیه

عدد فیبوناچی شماره 33 برابر با 5702887 است.

زمان محاسبه: 0.0000 ثانیه

عدد فیبوناچی شماره 34 برابر با 9227465 است.

زمان محاسبه: 0.0000 ثانیه

تابع fib\_memoized.memoizeDict: <function fib\_memoized at 0x0000021E75972340>

## مدیریت خطا (Error Handling)

مدیریت خطا در پایتون با استفاده از بلاک‌های try و except انجام می‌شود. شما کدی که ممکنه خطا بده رو توی بلاک try می‌نویسید و اگر خطای اتفاق افتاد، کد داخل بلاک except اجرا می‌شود. این روش جلوی کردن برنامه رو می‌گیره.

یک بلاک finally هم داریم که کد داخلش، در هر صورت (چه خطا اتفاق بیفته و چه نیافته) اجرا می‌شود. این بلاک برای کارهایی مثل بستن فایل‌ها یا آزاد کردن منابع خیلی کاربرد دارد.

In [ ]: متغیرها # # مثال: مدیریت خطا (Error Handling)

```

try:
    # این کد ممکنه خطا بده ( تقسیم بر صفر )
    result = 10 / 0
except ZeroDivisionError:
    # اتفاق بیفته، این بلاک اجرا می‌شود
    print("خطا: نمیتوان بر صفر تقسیم کرد")
except TypeError:
    # می‌توانی برای انواع خطاهای دیگه هم بلاک جداگانه داشته باشی
    print("خطا: نوع داده اشتباه است")
except Exception as e:
    # این یک بلاک عمومی برای گرفتن همه خطاهاست
    print(f"خطا رخ داد {e}")
finally:
    # این بلاک همیشه اجرا می‌شود
    print("برنامه به پایان رسید")
# یک مثال دیگه بدون خطای
print("\n--- مثال بدون خطای ---")
try:
    result = 10 / 2
    print(f"نتیجه تقسیم: {result}")
except:
    print("خطا رخ داد")
finally:
    print("برنامه به پایان رسید")

```

## (ساخت لیست به روش فشرده) List Comprehensions

«یه راه خیلی قدرتمند و فشرده برای ساخت لیست‌های جدید توی پایتونه. با این روش می‌تونید لیست‌ها را بر اساس لیست‌های دیگه یا هر شء قابل پیمایش دیگه‌ای (مثل رشته‌ها یا چندتایی‌ها) بسازید.»

این کار باعث می‌شود که توون هم کوتاه‌تر بشه و هم خوانایی بیشتری داشته باشه. ساختار کلیش این شکلیه: [expression for item in iterable if condition]

بخش expression همون چیزیه که برای هر آیتم محاسبه می‌شود و به لیست جدید اضافه می‌شود. for item in iterable هم که روی آیتم‌ها پیمایش می‌کنه و if condition هم یه شرط اختیاریه که اگه درست باش، آیتم به لیست جدید اضافه می‌شود.

مثلاً به جای اینکه با یه حلقة for و append یه لیست بسازید، می‌تونید همه رو توی یک خط بنویسید.

In [4]: # List Comprehensions (ساخت لیست به روش فشرده)

```
# مثال ۱: ساخت لیست اعداد زوج ---  
# می خوایم لیستی از اعداد زوج بین ۰ تا ۹ بسازیم.  
# روش سنتی با حلقه for:  
even_numbers_traditional = []  
for i in range(10):  
    if i % 2 == 0:  
        even_numbers_traditional.append(i)  
print(f"اعداد زوج (روش سنتی): {even_numbers_traditional}")  
  
# استفاده از List Comprehension:  
# چی رو می خوایم اضافه کنیم / برای هر آیتم در کجا / اگر چه شرطی برقرار بود [ ]  
even_numbers_comprehension = [i for i in range(10) if i % 2 == 0]  
print(f"اعداد زوج (List Comprehension): {even_numbers_comprehension}")  
  
# مثال ۲: تبدیل رشته ها به حروف بزرگ ---  
# لیستی از کلمات رو به حروف بزرگ تبدیل می کنیم.  
words = ["apple", "banana", "cherry", "date"]  
  
# روش سنتی:  
uppercase_words_traditional = []  
for word in words:  
    uppercase_words_traditional.append(word.upper())  
print(f"\n(روش سنتی): کلمات بزرگ {uppercase_words_traditional}")  
  
# استفاده از List Comprehension:  
uppercase_words_comprehension = [word.upper() for word in words]  
print(f"(List Comprehension): کلمات بزرگ {uppercase_words_comprehension}")  
  
# مثال ۳: ساخت لیستی از مربع اعداد با شرط ---  
# مربع اعدادی رو می خوایم که بین ۰ تا ۹ هستن و فقط اعداد فرد رو شامل بنشن  
squares_of_odd_numbers = [x**2 for x in range(10) if x % 2 != 0]  
print(f"\n(مربع اعداد فرد): {squares_of_odd_numbers}")  
  
# مثال ۴: تو در تو List Comprehension (Nested) ---  
# ساخت یک لیست فلت (تک بعدی) از یک لیست دو بعدی.  
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
  
# روش سنتی:  
flat_list_traditional = []  
for row in matrix:  
    for item in row:  
        flat_list_traditional.append(item)  
print(f"\n(روش سنتی): لیست فلت {flat_list_traditional}")  
  
# تو در تو استفاده از List Comprehension:  
flat_list_comprehension = [item for row in matrix for item in row]  
print(f"(List Comprehension): لیست فلت {flat_list_comprehension}")  
  
# مثال ۵: List Comprehension با if-else شرط در expression ---  
# رو اضافه کن "Odd" اگر عدد زوج بود، خودش رو اضافه کن، در غیر این صورت.  
numbers_with_odd_marker = [x if x % 2 == 0 else "Odd" for x in range(10)]  
print(f"\n(اعداد با نشانگر فرد/زوج): {numbers_with_odd_marker}")
```

[8, 6, 4, 2, 0] (روش سنتی): اعداد زوج (List Comprehension): [0, 2, 4, 6, 8]

['APPLE', 'BANANA', 'CHERRY', 'DATE'] (روش سنتی): کلمات بزرگ (روش سنتی)  
['APPLE', 'BANANA', 'CHERRY', 'DATE'] (List Comprehension): کلمات بزرگ

[81, 49, 25, 9, 1] مربع اعداد فرد:

[1, 2, 3, 4, 5, 6, 7, 8, 9] (روش سنتی): لیست فلت (روش سنتی):  
[1, 2, 3, 4, 5, 6, 7, 8, 9] (List Comprehension): لیست فلت

[0, 2, 4, 'Odd', 6, 'Odd', 8, 'Odd', 1, 3, 5, 7, 9] (روش سنتی): اعداد با نشانگر فرد/زوج

## ورودی/خروجی فایل (File I/O)

توی پایتون، کار کردن با فایلها (مثل خوندن یا نوشتن اطلاعات) خیلی راحته. این قابلیت به شما اجازه می ده که با داده هایی که توی فایلها ذخیره شدن، ارتباط برقرار کنید.

برای باز کردن یه فایل، از تابع `open()` استفاده می کنیم. این تابع دو تا ورودی اصلی می گیره: اسم فایل و «حالت» (mode) باز کردن فایل. حالت های مختلفی مثل '`r`' برای خوندن، '`w`' برای نوشتن (که اگه فایل وجود نداشته باشه، می سازدش و اگه وجود داشته باشه، محتواش رو پاک می کنه)، '`a`' برای اضافه کردن به انتهای فایل، و '`x`' برای ساخت یه فایل جدید (اگه وجود داشته باشه، خطای می دهد) داریم.

بهترین روش برای کار با فایلها، استفاده از دستور `with` هست. این دستور تضمین می کنه که حتی اگه خطایی هم پیش بیاد، فایل بعد از اتمام کار به صورت خودکار بسته بشه و منابع آزاد بشن.

In [5]: # ورودی/خروجی فایل (File I/O)

```
# مثال ۱: نوشتن در فایل ---  
# باز می کنیم (write) رو برای نوشتن 'my_file.txt' یک فایل به نام.  
# اگه فایل وجود نداشته باشه، ساخته می شه. اگه وجود داشته باشه، محتواش پاک می شه.  
try:  
    with open('my_file.txt', 'w', encoding='utf-8') as file:  
        file.write("\n") # این اولین خط  
        file.write("\n") # این هم خط دوم  
        file.write("\n") # پایتون کار با فایل رو آسون کرده  
        print("با موفقیت انجام شد 'my_file.txt' نوشتن در فایل")  
except IOError as e:  
    print(f"خطا در نوشتن فایل {e}")  
  
# مثال ۲: خواندن از فایل ---  
# باز می کنیم (read) رو برای خوندن 'my_file.txt' فایل
```

```

try:
    with open('my_file.txt', 'r', encoding='utf-8') as file:
        content = file.read() # خوندن کل محتوای فایل
        print("\n--- 'my_file.txt' ---")
        print(content)
except FileNotFoundError:
    print("\n--- خطأ: الملف 'my_file.txt' غير موجود")
except IOError as e:
    print(f"\n--- خطأ في الملف 'my_file.txt': {e}")

# مثال ۲: خوندن خط به خط از فایل
# فایل رو خط به خط میخوینیم
try:
    with open('my_file.txt', 'r', encoding='utf-8') as file:
        print("\n--- 'my_file.txt' ---")
        for line_num, line in enumerate(file):
            print(f"خط {line_num + 1}: {line.strip()}") # .strip() newline
except FileNotFoundError:
    print("\n--- خطأ: الملف 'my_file.txt' غير موجود")
except IOError as e:
    print(f"\n--- خطأ في الملف 'my_file.txt': {e}")

# مثال ۳: اضافه کردن به انتهای فایل
# باز میکنیم (append) رو برای اضافه کردن به انتهای 'my_file.txt' فایل
try:
    with open('my_file.txt', 'a', encoding='utf-8') as file:
        file.write("\n").join([f"خط {line_num + 1}: {line.strip()}" for line_num, line in enumerate(file)])
        print("\n--- 'my_file.txt' اضافه شد")
except IOError as e:
    print(f"\n--- خطأ في الملف 'my_file.txt': {e}")

حالا دوباره فایل رو میخوینیم تا تغییرات رو ببینیم
try:
    with open('my_file.txt', 'r', encoding='utf-8') as file:
        content = file.read()
        print("\n--- 'my_file.txt' محتوى اليوم ---")
        print(content)
except FileNotFoundError:
    print("\n--- خطأ: الملف 'my_file.txt' غير موجود")
except IOError as e:
    print(f"\n--- خطأ في الملف 'my_file.txt': {e}")

# مثال ۴: ایجاد فایل جدید
# تلاش برای ساخت فایلی که قبلاً وجود نداره
# وجود داشته باشد، خط امده 'new_exclusive_file.txt'
try:
    with open('new_exclusive_file.txt', 'x', encoding='utf-8') as file:
        file.write("\n").join(["خط امده 'new_exclusive_file.txt'"])
        print("\n--- 'new_exclusive_file.txt' ایجاد شد")
except FileExistsError:
    print("\n--- خطأ: الملف 'new_exclusive_file.txt' موجود")
except IOError as e:
    print(f"\n--- خطأ في الملف 'new_exclusive_file.txt': {e}")

```

با موفقیت انجام شد 'my\_file.txt' نوشتن در فایل

--- محتوای فایل  
سلام، این اولین خط  
این هم خط دوم.  
پایتون کار با فایل رو آسون کرده

--- 'my\_file.txt'  
خط 1: سلام، این اولین خط  
خط 2: این هم خط دوم  
خط 3: پایتون کار با فایل رو آسون کرده

با موفقیت انجام شد 'my\_file.txt' اضافه کردن به فایل

--- محتوای فایل  
سلام، این اولین خط  
این هم خط دوم  
پایتون کار با فایل رو آسون کرده  
این خط جدید به انتهای فایل اضافه شده

با موفقیت ایجاد شد 'new\_exclusive\_file.txt'

## مدیریت‌کننده‌های متن (Context Managers) و دستور with

«مدیریت‌کننده‌های متن» یا «Context Managers» یه الگوی طراحی توی پایتون هستن که به شما کمک می‌کنن منابع رو به شکل امن و خودکار مدیریت کنند. بهترین مثالش همون دستور `(...)` as `file`: هست که قبلاً دیدیم.

وقتی از دستور `with` استفاده می‌کنیم، پایتون تضمین می‌کنه که یه سری عملیات (مثلاً باز کردن و بستن فایل، یا اتصال به پایگاه داده) به درستی و در زمان مناسب انجام بشن، حتی اگه وسط کار خطایی پیش بیاد. این کار باعث می‌شه کدتون هم تمیزتر باشد و هم کمتر دچار مشکل بشه.

برای ساختن به مدیریت‌کننده‌ی متن خودمدون، می‌تونیم از کلاس‌ها استفاده کنیم و دو تا متده خاص به نامهای `__enter__` و `__exit__` رو تعریف کنیم. متده `__enter__` وقتی وارد بلاک `with` می‌شیم اجرا می‌شه و متده `__exit__` هم وقتی از اون بلاک خارج می‌شیم (چه با موفقیت و چه با خطأ) اجرا می‌شه.

یه راه دیگه برای ساخت مدیریت‌کننده‌های متن، استفاده از «تولیدکننده‌ها» (generators) و دکوراتور `contextlib.contextmanager` هست که کد رو کوتاه‌تر و خواناتر می‌کنه.

In [6]: # 'with' و دستور (Context Managers) مدیریت‌کننده‌های متن

```

# مثال ۱: استفاده از مدیریت‌کننده متن داخلی
# باز کردن و بستن خودکار فایل با دستور 'with'

```

```

print("--- مثال ۱: استفاده از 'with open' ---")
try:
    with open('sample.txt', 'w', encoding='utf-8') as f:
        f.write("\n").این یک متن نمونه است
        f.write("Context Manager با استفاده از")#.فایل به درستی بسته می شود
    print("sample.txt' با موفقیت نوشته و بسته شد")#.با موفقیت نوشته و بسته شد فایل

    with open('sample.txt', 'r', encoding='utf-8') as f:
        content = f.read()
        print("\n":محترای فایل)
        print(content)
except IOError as e:
    print(f"خطا در عملیات فایل {e}")

# --- سفارشی با کلاس Context Manager مثال ۲: ساخت یک
# یک کلاس ساده برای شبیه سازی مدیریت یک منبع (متلاً اتصال به دیتابیس)
class MyContextManager:
    def __init__(self, name):
        self.name = name
        print(f"\nContext Manager '{self.name}'".ساخته شد)

    def __enter__(self):
        می شیم اجرا می شه 'with' این متاد وقتی وارد بلاک #print(f"Context Manager '{self.name}' متاد شیم __enter__().")
        return self.name.upper()#استفاده بشه 'as var' می تونیم چیزی رو برگردانیم که به عنوان

    def __exit__(self, exc_type, exc_val, exc_tb):
        خارج می شیم اجرا می شه 'with' این متاد وقتی از بلاک #حتی اگه خطایی هم پیش بیاد، اجرا می شه
        print(f"Context Manager '{self.name}' متاد خارج شیم __exit__().")
        if exc_type: #اگه خطایی اتفاق افتاده باشه
            print(f"رخ داد {exc_type.__name__} {exc_val}")#برگردانیم، خط را سرکوب می کنیم و به بیرون منتشر نمی شه
            #اگه True بگردانیم، خط را به بیرون منتشر می شه
            #اگه False None با
            return False

print("\n--- مثال ۲: سفارشی با کلاس Context Manager")
with MyContextManager("مورد اول") as item:
    print(f"مقدار برگشتی از 'with'. درون بلاک __enter__: {item}")

print("\n--- ۲.۱ مثال: Context Manager با خطا")
try:
    with MyContextManager("مورد دوم با خطای") as item:
        print(f"درون بلاک 'with' (قبل از خط).")
        raise ValueError("ایجاد یک خطای عمدی است")#این یک خطای هرگز اجرا نمی شود
    print(f"این خط هرگز اجرا نمی شود")
except ValueError as e:
    print(f"خطا در خارج از Context Manager: {e}")

# --- مثال ۳: ساخت Context Manager با دکوراتور @contextlib.contextmanager
# این روش کوتاه تر و پایتونیک تر برای ساخت Context Manager.
import contextlib

@contextlib.contextmanager
def my_simple_context(resource_name):
    print("\n--- مثال ۳: Context Manager --- با دکوراتور @contextlib.contextmanager ---")
    print(f"--- با دکوراتور {resource_name} این روش کوتاه تر و پایتونیک تر برای ساخت Context Manager.")
    print(f"--- آمده اسازی منبع قبلاً yield .")

    try:
        yield resource_name.upper() # این چیزی که به 'as var' می شه
    finally:
        اجرا می شه (جه با خطای بدون خطای 'with' این بخش بعد از خروج از بلاک) #پس از خروج از 'with' باید از 'yield' پاکسازی منبع
        print(f"پاکسازی منبع {resource_name}.")

    with my_simple_context("اتصال دیتابیس") as db_conn:
        print(f"اتصال دیتابیس درون بلاک 'with'. {db_conn}")
        اینجا می تونیم کارهای مربوط به دیتابیس رو انجام بدیم #اینها می توانند با هم تغیر کنند

print("\n--- ۳.۱ مثال: Context Manager با دکوراتور و خطای")
try:
    with my_simple_context("عملیات پر خطر") as op:
        print(f"شروع عملیات {op}.")
        1 / 0 #ایجاد خطای تقسیم بر صفر
    این خط اجرا نمی شود # اینها می توانند با هم تغیر کنند
except ZeroDivisionError as e:
    print(f"خطا در خارج از Context Manager: {e}")

```

--- مثال ۱: استفاده از 'with open' ---  
با موفقیت نوشته و بسته شد 'sample.txt' فایل.

محتوای فایل:  
این یک متن نمونه است.  
فایل به درستی بسته می‌شود با استفاده از Context Manager.

--- سفارشی با کلاس Context Manager :مثال ۲ ---

مورد اول 'ساخته شد' Context Manager.  
مورد اول 'ارد' Context Manager متد `__enter__`.  
مورد اول 'مقدار برگشتی از 'with''. درون بلاک Context Manager متد `__exit__`.

--- با خطای Context Manager :مثال ۲.۱ ---

مورد دوم با خطای 'ساخته شد' Context Manager.  
مورد دوم با خطای 'شیم' `__enter__`.  
(قبل از خطای 'with' درون بلاک).  
مورد دوم با خطای 'خارج شدم' `__exit__`.  
از Context Manager! رخ داد: این یک خطای عمدى است ValueError خطای از نوع  
این یک خطای عمدى است Context Manager: خطای خارج از!

--- با دکوراتور Context Manager :مثال ۳ ---  
--- با دکوراتور `yield`:  
اماده‌سازی منبع 'اتصال دیتابیس'.  
اتصال: اتصال دیتابیس 'with'. درون بلاک `yield`.

--- با دکوراتور و خطای Context Manager :مثال ۳.۱ ---  
--- با دکوراتور `yield`:  
اماده‌سازی منبع 'عملیات پرخط'.  
اًشروع عملیات 'عملیات پرخط'.  
`yield` بعد از 'پاکسازی منبع 'عملیات پرخط'.  
Context Manager: division by zero

## مدیریت خطای پیشرفتی و استثناهای سفارشی (Custom Exceptions)

توی برنامه‌نویسی، گاهی اوقات لازم می‌شه که خودمون خطاهای خاصی رو تعریف کنیم. این کار بهمون کمک می‌کنه که کدمون رو تمیزتر بنویسیم و خطاهایی رو که برای برنامه‌مون معنی‌دار هستن، بهتر مدیریت کنیم.

برای تعریف یه «استثنای سفارشی» (Custom Exception)، کافیه یه کلاس جدید بسازیم که از کلاس پایه `Exception` (یا یکی از زیرکلاس‌های اون) ارثبری کنه. این کار به پایتون می‌گه که این کلاس شما، یه نوع خطای محسوب می‌شه.

وقتی خطای خاصی توی برنامه‌تون پیش می‌یابد، می‌تونید با دستور `raise` اون استثنای سفارشی رو ایجاد کنید. بعدش، توی بلاک `try...except` می‌تونید این خطای خاص رو بگیرید و بهش واکنش نشون بدید. این روش باعث می‌شه که برنامه‌تون در مواجهه با شرایط خاص، هوشمندانه‌تر عمل کنه و پیام‌های خطای معنی‌دارتری بده.

In [7]: # مدیریت خطای پیشرفتی و استثناهای سفارشی (Custom Exceptions)

```
--- مثال ۱: تعریف یک استثنای سفارشی ساده ---  
# ارثبری کنیم Exception برای تعریف یک استثنای سفارشی، کافیه از کلاس
```

```
class InvalidInputError(Exception):
```

```
    """
```

```
    این یک استثنای سفارشی برای ورودی‌های نامعتبر است.
```

```
    """
```

```
    def __init__(self, message="ورودی نامعتبر است"):
```

```
        self.message = message
```

```
        super().__init__(self.message)
```

```
--- مثال ۲: استفاده از استثنای سفارشی ---
```

```
def process_positive_number(number):
```

```
    """
```

```
    این تابع فقط اعداد مثبت را پردازش می‌کند.
```

```
    ایجاد می‌کند InvalidInputError، اگر عدد منفی یا صفر باشد.
```

```
    """
```

```
    if not isinstance(number, (int, float)):
```

```
        raise TypeError("ورودی باید عدد باشد")
```

```
    if number <= 0:
```

```
        اینجا استثنای سفارشی خودمان را ایجاد می‌کنیم #
```

```
        raise InvalidInputError(f"عدد {number} باید مثبت باشد")
```

```
    return number * 2
```

```
print("--- مثال ۲: استفاده از استثنای سفارشی ---")
```

```
try:
```

```
    result1 = process_positive_number(5)
```

```
    print(f"نتیجه پردازش عدد ۵: {result1}")
```

```
    این خط خطای ایجاد می‌کند #
```

```
    print(f"نتیجه پردازش عدد -۳: {result2}")
```

```
except InvalidInputError as e:
```

```
    print(f"خطای ورودی نامعتبر: {e}")
```

```
except TypeError as e:
```

```
    print(f"خطای نوع داده: {e}")
```

```
except Exception as e: # برای گرفتن هر خطای دیگری که بیش بیاید #
```

```
    print(f"خطای غیرمنتظره: {e}")
```

```
print("\n--- مثال ۳: استثنای سفارشی با جزئیات بیشتر ---")
```

```
class UserNotFoundError(Exception):
```

```
    """
```

```
    استثنای سفارشی برای زمانی که کاربر پیدا نمی‌شود.
```

```
    """
```

```
    def __init__(self, user_id, message="کاربر مورد نظر پیدا نشد"):
```

```
        self.user_id = user_id
```

```
        self.message = f"{message} شناسه کاربر: {user_id}"
```

```
        super().__init__(self.message)
```

```

def get_user_data(user_id):
    """
    این تابع اطلاعات کاربر را بر اساس شناسه کاربر برمی‌گرداند.
    ایجاد می‌کند UserNotFoundError، اگر کاربر با شناسه مشخص شده وجود نداشته باشد
    """
    users_db = {
        101: "علی",
        102: "فاطمه",
        103: "محمد"
    }
    if user_id not in users_db:
        raise UserNotFoundError(user_id) # ایجاد استثنای با جزئیات بیشتر
    return users_db[user_id]

try:
    user_name = get_user_data(102)
    print(f"۱۰۲: کاربر با شناسه {user_name}")

    user_name = get_user_data(999) # این خط خطا ایجاد می‌کند
    print(f"۹۹۹: کاربر با شناسه {user_name}")

except UserNotFoundError as e:
    print(f"خطا در باقتن کاربر: {e}")
except Exception as e:
    print(f"خطای کلی: {e}")

```

--- مثال ۲: استفاده از استثنای سفارشی  
نتیجه پردازش عدد ۵: ۱۰  
خطای ورودی نامعتبر: عدد -۳ باید مثبت باشد.

--- مثال ۳: استثنای سفارشی با جزئیات بیشتر  
کاربر با شناسه ۱۰۲: فاطمه  
خطا در باقتن کاربر: کاربر مورد نظر پیدا نشد. (شناسه کاربر: 999)

## پیمایش‌گرها و پیمایش‌بذرگها (Iterators and Iterables)

توی پایتون، وقتی می‌گیم یه شیء «پیمایش‌بذرگ» (iterable) هست، یعنی می‌توnim روش حلقة بزنیم (مثلاً با یه حلقة for). لیست‌ها، رشته‌ها، چندتایی‌ها (tuples) و دیکشنری‌ها همه‌شون پیمایش‌بذرگ‌هستن.

اما «پیمایش‌گر» (iterator) یه شیء هست که وضعیت پیمایش رو نگه می‌داره. یعنی می‌دونه که توی توالی، الان کجاست و آیتم بعدی چیه. پیمایش‌گرها با متدهای `__iter__` (که خودش یه پیمایش‌گر برمی‌گردونه) و `__next__` (که آیتم بعدی رو می‌ده) کار می‌کنن.

وقتی یه حلقة for رو روی یه شیء پیمایش‌بذرگ اجرا می‌کنیم، پایتون پشت صحنه اول `iter()` رو صدا می‌زنه تا یه پیمایش‌گر بگیره، بعدش هی `next()` رو صدا می‌زنه تا آیتم‌های بعدی رو بگیره. وقتی دیگه آیتمی نمونه‌باشه، استثنای `StopIteration` ایجاد می‌شه و حلقة تمام می‌شه.

فهمیدن این مفاهیم کمک می‌کنه که بتونید کدهای بهینه‌تری بنویسید، مخصوصاً وقتی با داده‌های بزرگ سروکار دارید، چون لازم نیست کل داده‌ها رو یکجا توی حافظه نگه دارید.

In [8]: # پیمایش‌گرها و پیمایش‌بذرگها (Iterators and Iterables)

```

# --- مثال ۱: شیء پیمایش‌بذرگ ---
# لیست‌ها، رشته‌ها و تابلهای نمونه‌هایی از اشیای پیمایش‌بذرگ هستند
my_list = [10, 20, 30, 40]
my_string = "پایتون"
my_tuple = (1, 2, 3)

print("--- پیمایش روی لیست")
for item in my_list:
    print(item)

print("\n--- پیمایش روی رشته")
for char in my_string:
    print(char)

# --- مثال ۲: تبدیل یک پیمایش‌بذرگ به پیمایش‌گر ---
# یک شیء پیمایش‌گر از یک شیء پیمایش‌بذرگ برمی‌گرداند
my_list_iterator = iter(my_list)
my_string_iterator = iter(my_string)

print("\n--- روی پیمایش‌گر لیست")
print(next(my_list_iterator)) # اولین آیتم
print(next(my_list_iterator)) # دومین آیتم
print(next(my_list_iterator)) # سومین آیتم

print("\n--- روی پیمایش‌گر رشته")
print(next(my_string_iterator)) # اولین کاراکتر
print(next(my_string_iterator)) # دومین کاراکتر
print(next(my_string_iterator)) # سومین کاراکتر

# ایجاد می‌شود StopIteration، وقتی آیتمی برای پیمایش نیاشد
try:
    print(next(my_list_iterator)) # چهارمین آیتم
    print(next(my_list_iterator)) # نلاش برای گرفتن آیتم پنجم (خطا می‌دهد)
except StopIteration:
    print("پیمایش لیست به پایان رسید")

```

--- مثال ۳: ساخت یک شیء پیمایش‌بذرگ و پیمایش‌گر سفارشی ---  
یک کلاس که اعداد زوج را تا یک حد مشخص تولید می‌کند

```

class EvenNumbers:
    def __init__(self, max_num):
        self.max_num = max_num
        self.current = 0 # شروع از صفر

    def __iter__(self):
        # این متدهای خود شیء پیمایش‌گر را برگرداند
        # در این حالت، خود کلاس هم پیمایش‌بذرگ است و هم پیمایش‌گر
        return self

    def __next__(self):
        # این متدهای خود شیء پیمایش‌گر را برگرداند
        # در این حالت، خود کلاس هم پیمایش‌بذرگ است و هم پیمایش‌گر
        # return self

```

```

# این متدهایی را برای گرداندن
while self.current <= self.max_num:
    if self.current % 2 == 0:
        even_num = self.current
        self.current += 1
    return even_num
self.current += 1
# وقتی آیتمی برای برگرداندن نباشد StopIteration را ایجاد میکنیم
raise StopIteration

```

print("\n--- مثال ۳: پیمایشگر سفارشی ---")  
even\_gen = EvenNumbers(10) یک شیء از کلاس EvenNumbers میسازیم.

روی آن پیمایش کنیم for میتوانیم مستقیماً با حلقه #

```

for num in even_gen:
    print(num)

(" --- استفاده مجدد از پیمایشگر سفارشی (نیاز به ساخت شیء جدید)
# یک پیمایشگر جدید میسازیم چون پیمایشگر قبلی به پایان رسیده است
another_even_gen = EvenNumbers(5)
print(next(another_even_gen))
print(next(another_even_gen))
print(next(another_even_gen))

try:
    print(next(another_even_gen))
except StopIteration:
    print("پیمایش دوم هم به پایان رسید")

```

--- پیمایش روی لیست ---

10  
20  
30  
40

--- پیمایش روی رشته ---

پ  
ا  
ی  
ت  
و  
ن

روی پیمایشگر لیست (next() استفاده از ---

10  
20  
30

روی پیمایشگر رشته (next() استفاده از ---

پ  
ا  
ی  
ت  
و  
ن

پیمایش لیست به پایان رسید (StopIteration).

--- مثال ۳: پیمایشگر سفارشی ---

0  
2  
4  
6  
8  
10

--- استفاده مجدد از پیمایشگر سفارشی (نیاز به ساخت شیء جدید)

0  
2  
4

پیمایش دوم هم به پایان رسید.

## آرگومان‌های دلخواه توابع (args\* و kwargs\*\*)

توی پایتون، وقتی په تابع می‌نویسیم، گاهی اوقات نمی‌دونیم قراره چند تا ورودی بهش بدیم. اینجا args\* و kwargs\*\* به کمک می‌بینیم.

args\* به شما اجازه می‌ده هر تعداد «آرگومان موقعیتی» (positional arguments) که می‌خواهید به تابع پاس بدم. این آرگومان‌ها داخل تابع به صورت یک «جندتاپی» (tuple) جمع‌آوری می‌شون.

kwargs\*\* هم برای گرفتن هر تعداد «آرگومان کلیدواژه‌ای» (keyword arguments) استفاده می‌شه. این آرگومان‌ها داخل تابع به صورت یک «دیکشنری» (dictionary) جمع‌آوری می‌شون که کلیدهایشون همون نام آرگومان‌ها هستن.

این دو تا قابلیت خیلی برای نوشتن توابع انعطاف‌پذیر و عمومی که می‌تونن با تعداد متفاوتی از ورودی‌ها کار کنن، مفیدن. مثلًا وقتی می‌خواهید یه تابع بنویسید که بتونه مجموع هر تعداد عدد رو حساب کنه یا اطلاعات مختلفی رو به یه تابع دیگه پاس بدم.

In [9]: # آرگومان‌های دلخواه توابع (\*args و \*\*kwargs)

```

# مثال ۱: استفاده از ---
# تابع جمع‌زننده که هر تعداد عدد را می‌پنیرد
def sum_all_numbers(*args):
    """
    این تابع هر تعداد آرگومان عددی را می‌پنیرد و مجموع آنها را برای گرداند.
    همه آرگومان‌های موقعیتی را به صورت یک تاپل جمع‌آوری می‌کند *args.
    """

    # این تابع خواهد بود نوع args: {type(args)} # args
    # آرگومان‌های دریافت شده {args}
    total = 0
    for num in args:
        total += num
    return total

```

print(\*args) مثلاً استفاده از ---

```

print(f"(۱, ۲, ۳): مجموع (۱, ۲, ۳) {sum_all_numbers(1, 2, 3)}")
print(f"(۵۰, ۴۰, ۳۰, ۲۰, ۱۰): مجموع (۱۰, ۲۰, ۳۰, ۴۰, ۵۰) {sum_all_numbers(10, 20, 30, 40, 50)}")
print(f" فقط یک عدد: {sum_all_numbers(100)}")
print(f" بدون عدد: {sum_all_numbers()}")

# --- مثال ۲: استفاده از **kwargs ---
تابعی که اطلاعات کاربر را به صورت کلید-مقدار می‌پذیرد.
def display_user_info(**kwargs):
    """
    این تابع اطلاعات کاربر را به صورت آرگومان‌های کلیدوازه‌ای می‌پذیرد.
    همه آرگومان‌های کلیدوازه‌ای را به صورت یک دیکشنری جمع‌آوری می‌کند
    **kwargs
    """

    print(f"\n نوع kwargs: {type(kwargs)} # kwargs")
    print(f" شده اطلاعات کاربر دریافت شده: {kwargs}")

    if "name" in kwargs:
        print(f" نام: {kwargs['name']}")
    if "age" in kwargs:
        print(f" سن: {kwargs['age']}")
    if "city" in kwargs:
        print(f" شهر: {kwargs['city']}")
    print("-" * 20)

print("\n--- مثال ۲: استفاده از **kwargs ---")
display_user_info(name="علی", age=30, city="تهران")
display_user_info(product="الپنتاپ", price=1200, brand="HP")
display_user_info(country="ایران")

# --- مثال ۳: استفاده همزمان از *args و **kwargs ---
# سپس *args، ترتیب مهم است: ابتدا آرگومان‌های معمولی، سپس **kwargs
def process_data(id, *args, **kwargs):
    """
    این تابع یک شناسه، هر تعداد آرگومان موقعیتی و هر تعداد آرگومان کلیدوازه‌ای را می‌پذیرد.
    """

    print(f"\n--- مثال ۳: استفاده همزمان از *args و **kwargs ---")
    print(f" شناسه: {id}")
    print(f" آرگومان‌های اضافی (args): {args}")
    print(f" آرگومان‌های کلیدوازه‌ای (kwargs): {kwargs}")
    print("=" * 30)

process_data(101, "item1", "item2", price=50, quantity=2)
process_data(202, "فعل", status="فعال")
process_data(303, # بدون args یا kwargs اضافی

# --- مثال ۴: باز کردن لیست/دیکشنری با * و (unpacking)
# می‌توانیم از * و ** برای پاس دادن لیست‌ها و دیکشنری‌ها به عنوان آرگومان استفاده کنیم.
def greet_people(greeting, *names):
    for name in names:
        print(f"{greeting}, {name}!")

people_list = ["سارا", "رضا", "مریم"]
print("\n--- مثال ۴: باز کردن لیست با ---")
greet_people("سلام", *people_list) # لیست را به عنوان آرگومان‌های جداگانه پاس می‌دهد.

def create_profile(username, **details):
    profile = {"username": username}
    profile.update(details)
    return profile

user_details = {"age": 25, "city": "اصفهان", "email": "user@example.com"}
print("\n--- مثال ۴: باز کردن دیکشنری با ---")
new_user_profile = create_profile("احمدی", **user_details) # باز کردن دیکشنری پاس می‌دهد
print(f" پروفایل کاربر: {new_user_profile}")

```

```
--- مثال ۱: استفاده از *args ---
args: <class 'tuple'>
آرگومان‌های دریافت شده: (1, 2, 3)
مجموع (۱, ۲, ۳): 6
نوع args: <class 'tuple'>
آرگومان‌های دریافت شده: (50, 40, 30, 20, 10, 5, ۴۰, ۳۰, ۲۰, ۱۰)
مجموع (۵۰, ۴۰, ۳۰, ۲۰, ۱۰, ۵, ۴۰, ۳۰, ۲۰, ۱۰): 150
نوع args: <class 'tuple'>
آرگومان‌های دریافت شده: (100)
مجموع ( فقط یک عدد): 100
نوع args: <class 'tuple'>
() آرگومان‌های دریافت شده
مجموع ( بدون عدد): 0
```

--- مثال ۲: استفاده از \*\*kwargs ---

```
نوع kwargs: <class 'dict'>
{"تهران": {"name": "تهران", "age": 30, "city": "تهران"}: اطلاعات کاربر دریافت شده
نام: تهران
سن: 30
شهر: تهران
-----
نوع kwargs: <class 'dict'>
{"اپل": {"product": "اپل", "price": 1200, "brand": "HP"}: اطلاعات کاربر دریافت شده
-----
نوع kwargs: <class 'dict'>
{"ایران": {"country": "ایران"}: اطلاعات کاربر دریافت شده
-----
```

--- مثال ۳: استفاده همزمان از \*args و \*\*kwargs ---

```
شناسه: 101
آرگومان‌های اضافی (args): ('item1', 'item2')
آرگومان‌های کلیدواژه‌ای (kwargs): {'price': 50, 'quantity': 2}
=====
```

--- مثال ۳: استفاده همزمان از \*args و \*\*kwargs ---

```
شناسه: 202
آرگومان‌های اضافی (args): ('فقط یک آیتم') آرگومان‌های اضافی
آرگومان‌های کلیدواژه‌ای (kwargs): {'status': 'فعال'}
=====
```

--- مثال ۳: استفاده همزمان از \*args و \*\*kwargs ---

```
شناسه: 303
آرگومان‌های اضافی (args): ()
آرگومان‌های کلیدواژه‌ای (kwargs): {}
=====
```

--- \* مثال ۴: باز کردن لیست با ---

```
سلام، سارا!
سلام، رضا!
سلام، مریم!
```

--- \*\* مثال ۴: باز کردن دیکشنری با ---

```
{'username': 'user@example.com', 'email': 'user@example.com', 'city': 'اصفهان', 'age': 25, 'احمدی': 'اصلی'}
```

## دکوراتورها با آرگومان (Decorators with Arguments)

همون‌طور که قبلاً گفتیم، «دکوراتورها» (Decorators) توابع خاصی هستند که به ما اجازه می‌دهند عملکرد یه تابع دیگه را بدون تغییر کد اصلیش، گسترش بدم. حالا تصور کنید اگه بتونیم به خود دکوراتور هم آرگومان بدیم!

«دکوراتورها با آرگومان» این امکان رو فراهم می‌کنن که دکوراتور رو موقع استفاده، با پارامترهای دلخواه تنظیم کنیم. این یعنی یه لایه انعطاف‌پذیری بیشتر به دکوراتور اضافه می‌شه و می‌تونیم رفتار دکوراتور رو بر اساس نیازهای خاصمنون تغییر بدیم.

برای ساختن به دکوراتور که آرگومان می‌گیره، باید یه تابع بیرونی دیگه هم اضافه کنیم. یعنی ساختار کلیش به تابع سه‌تایی می‌شه: تابع بیرونی (که آرگومان‌های دکوراتور رو می‌گیره)، تابع میانی (که تابع اصلی رو می‌گیره) و تابع درونی (که کد دکوراتور رو اجرا می‌کنه).

این قابلیت توی فریمورک‌های وب (مثل جنگو و فلسك) یا برای کارهایی مثل مدیریت دسترسی کاربران با نقشهای مختلف، لاجبرداری با جزئیات متفاوت، یا محدود کردن نرخ درخواست‌ها خیلی کاربرد دارد.

In [10]: # دکوراتورها با آرگومان (Decorators with Arguments)

```
import time

--- مثال ۱: دکوراتوری که تعداد دفعات اجرای تابع را لاتک می‌کند ---
# می‌گیرد 'message' این دکوراتور یک آرگومان.
def log_execution_count(message=""):
    """تابع اجرا شد"""
    # این تابع بیرونی، آرگومان‌های دکوراتور را می‌گیرد
    # این فقط یک بار، هنگام تعریف تابع دکوریت شده، اجرا می‌شود
    # دیکشنری برای نگهداری تعداد اجراهای هر تابع #
    execution_counts = {}

    def decorator(func):
        # را می‌گیرد (func) این تابع میانی، تابع اصلی
        # این هم فقط یک بار، هنگام تعریف تابع دکوریت شده، اجرا می‌شود
        # مقدار اولیه برای تابع فعلی #
        execution_counts[func.__name__] = 0

        def wrapper(*args, **kwargs):
            # این تابع درونی، هر بار که تابع اصلی صدا زده می‌شود، اجرا می‌شود
            execution_counts[func.__name__] += 1
            print(f"\n--- {message} ---")
            print(f"func.{func.__name__} برای {execution_counts[func.__name__]} تابع اجرا شد")
            result = func(*args, **kwargs) # اجرای تابع اصلی
            return result
        return wrapper
    return decorator
```

```

# استفاده از دکوراتور با آرگومان
@log_execution_count(message="عملیات مهم")
def perform_important_task(task_name):
    print(f"در حال اجام کار: {task_name}")
    time.sleep(0.1) # شبیه‌سازی کار
    return f"کار '{task_name}' انجام شد."

```

```

@log_execution_count(message="محاسبات")
def calculate_sum(a, b):
    print(f"در حال محاسبه {a} + {b}")
    time.sleep(0.05)
    return a + b

```

```

--- مثال ۱: استفاده از دکوراتور با آرگومان ---
perform_important_task("پردازش داده")
perform_important_task("گزارش‌گیری")
print(f"نتیجه جمع: {calculate_sum(5, 7)}")
print(f"نتیجه جمع دیگر: {calculate_sum(10, 20)}")

```

# مثال ۲: دکوراتور بررسی نقش کاربر --- (Role-based Access Control) ---

می‌گیرد 'required\_role' این دکوراتور یک آرگومان.

```

def require_role(required_role):
    def decorator(func):
        def wrapper(user_role, *args, **kwargs):
            print(f"\n--- بررسی دسترسی برای تابع '{func.__name__}' ---")
            print(f"نقش مورد نیاز: '{user_role}' نقش کاربر: '{required_role}'")
            if user_role == required_role:
                print("... دسترسی تایید شد. اجرای تابع...")
                return func(user_role, *args, **kwargs)
            else:
                print("خطا: دسترسی غیرمجاز")
                return None # می‌توانیم یک استثنای ایجاد کنیم
        return wrapper
    return decorator

```

```

@require_role("admin")
def delete_critical_data(user_role, data_id):
    print(f"داده با شناسه {data_id} توسط '{user_role}' حذف شد.")
    return True

```

```

@require_role("user")
def view_public_profile(user_role, profile_id):
    print(f"پروفایل با شناسه {profile_id} توسط '{user_role}' مشاهده شد.")
    return {"profile_id": profile_id, "status": "viewed"}

```

```

--- مثال ۲: دکوراتور بررسی نقش کاربر ---
delete_critical_data("admin", 123)
delete_critical_data("user", 456) # این اجرا نمی‌شود

این هم اجرا می‌شود چون ادمین معمولاً دسترسی کاربر را دارد # (admin, 101)

```

--- مثال ۱: استفاده از دکوراتور با آرگومان ---

--- عملیات مهم ---  
برای ۱ بار اجرا شد 'perform\_important\_task' تابع.  
در حال اجام کار: پردازش داده

--- عملیات مهم ---  
برای ۲ بار اجرا شد 'perform\_important\_task' تابع.  
در حال اجام کار: گزارش‌گیری

--- محاسبات ---  
برای ۱ بار اجرا شد 'calculate\_sum' تابع.  
در حال محاسبه 7 + 5  
نتیجه جمع: 12

--- محاسبات ---  
برای 2 بار اجرا شد 'calculate\_sum' تابع.  
در حال محاسبه 20 + 10  
نتیجه جمع دیگر: 30

--- مثال ۲: دکوراتور بررسی نقش کاربر ---

--- بررسی دسترسی برای تابع 'delete\_critical\_data' ---  
نقش مورد نیاز: 'admin': نقش کاربر  
... دسترسی تایید شد. اجرای تابع  
حذف شد 'admin' داده با شناسه 123 توسط

--- بررسی دسترسی برای تابع 'delete\_critical\_data' ---  
نقش مورد نیاز: 'user': نقش کاربر  
خطا: دسترسی غیرمجاز!

--- بررسی دسترسی برای تابع 'view\_public\_profile' ---  
نقش مورد نیاز: 'user': نقش کاربر  
... دسترسی تایید شد. اجرای تابع  
مشاهده شد 'user' پروفایل با شناسه 789 توسط

--- بررسی دسترسی برای تابع 'view\_public\_profile' ---  
نقش مورد نیاز: 'admin': نقش کاربر  
خطا: دسترسی غیرمجاز!

توی پایتون، همون طور که قبل‌اگفتیم، نیازی نیست نوع متغیرها رو مشخص کنید. اما توی پروژه‌های بزرگ‌تر، یا وقتی چند نفر روی یه کد کار می‌کنن، دونستن نوع ورودی‌ها و خروجی‌های توابع یا نوع متغیرها می‌تونه خیلی مفید باشه. اینجا «راهنمایی نوع» یا Type Hinting به کار می‌باد.

Type Hinting به شما اجازه می‌ده که به صورت اختیاری، نوع مورد انتظار برای ورودی‌های تابع، خروجی تابع، و حتی متغیرها رو مشخص کنید. این کار به پایتون نمی‌گه که حتماً باید این نوع باشه (چون پایتون هنوز هم «پویا»ست)، اما به برنامه‌نویس‌ها و ابزارهای تحلیل کد (مثل IDE‌ها) کمک می‌کنه تا خطاهای احتمالی رو زودتر پیدا کنن و کد رو بهتر درک کنن.

برای استفاده ازش، بعد از اسم متغیر یا پارامتر تابع یه دونقطه ( : ) می‌ذاریم و بعدش نوع رو می‌نویسیم. برای خروجی تابع هم، بعد از پرانتزهای ورودی و قبل از دونقطه نهایی، از <-> و بعدش نوع خروجی استفاده می‌کنیم.

این قابلیت باعث می‌شه کدهاتون خود-مستندساز بشن و نگهداریشون راحت‌تر بشه، مخصوصاً وقتی که کدتون پیچیده می‌شه یا تیم‌های بزرگ روشن کار می‌کنن.

In [11]: # راهنمایی نوع (Type Hinting)

```
# برای استفاده از انواع داده پیچیده‌تر مثل List, Dict, Tuple, Union و ...
# کیم typing را import باید مازو!
```

```
from typing import List, Dict, Tuple, Union, Optional
```

--- مثال ۱: راهنمایی نوع برای ورودی‌ها و خروجی تابع ---

```
def add_numbers(a: int, b: int) -> int:
```

```
    """
    این تابع دو عدد صحیح را دریافت کرده و مجموع آن‌ها را برمی‌گرداند
    راهنمایی نوع به خوانایی و درک ورودی/خروجی تابع کمک می‌کند
    """
    return a + b
```

```
print("مثال ۱: راهنمایی نوع در تابع ---")
```

```
result1 = add_numbers(5, 3)
```

```
print(f"۵ و ۳ جمع: {result1}")
```

پایتون همچنان پویاست و این یک "راهنمایی" است، نه احصار. هشدار می‌دهند (Linters) این کد اجرا می‌شود، اما ابزارهای تحلیل کد.

```
result2 = add_numbers(5.5, 3.2)
```

```
print(f"۵.۵ و ۳.۲ جمع: {result2}")
```

--- مثال ۲: راهنمایی نوع برای لیست‌ها ---

```
def get_first_item(data_list: List[str]) -> str:
```

```
    """
    اولین آیتم یک لیست از رشته‌ها را برمی‌گرداند
    """
    if data_list:
        return data_list[0]
    return "لیست خالی است."
```

```
--- مثال ۲: راهنمایی نوع برای لیست‌ها ---
```

```
my_strings = ["سبب", "پرتفال", "مز"]
```

```
print(f"اولین میوه: {get_first_item(my_strings)}")
```

```
my_numbers = [1, 2, 3]
```

هشدار می‌دهد Type Checker ( مثل MyPy) این کد اجرا می‌شود، اما

```
# print(f"۱۰۰ عدد: {get_first_item(my_numbers)})")
```

--- مثال ۳: راهنمایی نوع برای دیکشنری‌ها ---

```
def print_user_info(user: Dict[str, Union[str, int]]) -> None:
```

```
    """
    اطلاعات کاربر را از یک دیکشنری چاپ می‌کند
    دیکشنری شامل کلیدهای رشته‌ای و مقادیر رشته‌ای یا عددی است
    """
    print(f"نامشخص: {user.get('name')}")
    print(f"سن: {user.get('age')}")
    print(f"شهر: {user.get('city')}")
```

```
--- مثال ۳: راهنمایی نوع برای دیکشنری‌ها ---
```

```
user_data = {"name": "سارا", "age": 28, "city": "شیراز"}
```

```
print_user_info(user_data)
```

ندرایم 'city' و 'age' را از دیکشنری خارج کنیم.

```
another_user = {"name": "رضا", "job": "مهندس", "city": "آنکور", "age": 30}
```

```
print_user_info(another_user)
```

--- مثال ۴: راهنمایی نوع برای مقادیر اختیاری ---

# Optional[X] None باشد با X یعنی مقدار می‌تواند None باشد.

```
def find_item_or_none(items: List[str], target: str) -> Optional[str]:
```

```
    """
    آیتم مورد نظر را در لیست پیدا می‌کند و برمی‌گرداند
    برمی‌گرداند None در غیر این صورت
    """
    if target in items:
        return target
    return None
```

```
--- مثال ۴: راهنمایی نوع برای مقادیر اختیاری ---
```

```
fruits = ["سبب", "مز", "کیوی"]
```

```
found_fruit = find_item_or_none(fruits, "مز")
```

```
print(f"میوه پیدا شده: {found_fruit}")
```

```
not_found_fruit = find_item_or_none(fruits, "انگور")
```

```
print(f"میوه پیدا نشده: {not_found_fruit}")
```

--- مثال ۵: راهنمایی نوع برای متغیرها ---

متغیرها را هم راهنمایی کنیم.

```
name: str = "احمد"
```

```
age: int = 40
```

```
is_active: bool = True
```

```
print(f"\n--- مثال ۵: راهنمایی نوع برای متغیرها ---")
print(f"نام: {name}, نوع: {type(name)}")
print(f"سن: {age}, نوع: {type(age)}")
print(f"فعال: {is_active}, نوع: {type(is_active)})")

این راهنمایی‌ها در زمان اجرا چک نمی‌شوند، فقط برای ابزارهای تحلیل کد هستند.
# هشدار می‌دهد "چهل" # این خط در زمان اجرا خطا نمی‌دهد، اما "سنه" # print(f"سن تغییر یافته")
```

--- مثال ۱: راهنمایی نوع در تابع ---  
جمع ۵ و ۳ = 8  
8.7 جمع ۵.۵ و ۳.۲ (با هشدار نوع):

--- مثال ۲: راهنمایی نوع برای لیستها ---  
اولین میوه: سیب

--- مثال ۳: راهنمایی نوع برای دیکشنری‌ها ---  
نام: سارا  
سن: 28  
شهر: شیراز  
نام: رضا  
سن: نامشخص  
شهر: نامشخص

--- مثال ۴: راهنمایی نوع برای مقادیر اختیاری ---  
میوه پیدا شده: موز  
میوه پیدا نشده: None

--- مثال ۵: راهنمایی نوع برای متغیرها ---  
<class 'str'>: نام: احمد، نوع  
<class 'int'>: سن: 40، نوع  
True, نوع: <class 'bool'>

**The End**

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل دوم بخش اول: الگوریتم‌های مرتب‌سازی

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

• مقدمه

• مرتب‌سازی شمارشی (Counting Sort)

• مرتب‌سازی درجی (Insertion Sort)

• مرتب‌سازی حبابی (Bubble Sort)

• مرتب‌سازی انتخابی (Selection Sort)

• مرتب‌سازی پایدار و ناپایدار (Stable / Unstable Sort)

• مرتب‌سازی سطلی (Bucket Sort)

• مرتب‌سازی مبنایی (Radix Sort)

## مقدمه

مرتب‌سازی یکی از ساده‌ترین کارهایی است که به صورت روزمره انجام می‌دهیم. این کار به حدی برایمان ساده و طبیعی شده است که شاید به آن چندان فکر نکنیم ولی با کمی دقت به سادگی متوجه می‌شویم که برای مرتب‌کردن اشیای مختلف از روش‌های گوناگونی استفاده می‌کنیم.

در علوم کامپیوتر، مرتب‌سازی فقط یک کار روزمره نیست؛ بلکه یک عملیات بنیادی و بسیار مهم است. با مرتب کردن داده‌ها، می‌توانیم عملیات جستجو را سریع‌تر انجام دهیم، تحلیل داده‌ها را آسان‌تر کنیم و حتی در مدیریت پایگاه‌های داده کارایی را افزایش دهیم. به طور کلی، هدف از مرتب‌سازی، چیدن عناصر یک مجموعه (مثل اعداد، کلمات یا اشیا) بر اساس یک ترتیب مشخص (ممدوح صعودی یا نزولی) است.

برای نمونه فرض کنید که یک دسته اسکناس به شما داده شده‌است که در آن اسکناس‌های هزار، دوهزار و پنج‌هزار تومانی با هم مخلوط هستند. شاید اولین چیزی که به آن فکر کنید مرتب کردن این اسکناس‌ها باشد. چون مرتب‌بودن آن‌ها می‌تواند باعث ساده‌تر شدن استفاده از آن‌ها شود. مثلاً اگر بخواهیم یک مبلغ خاص را پردازید شاید برایتان ساده‌تر باشد که اسکناس‌ها را از یک دسته‌ی مرتب بیرون بکشید. حال سؤال این است که «چطور» اسکناس‌ها را مرتب می‌کنید؟

اکثر مردم (این اکثریت شامل دانشجوها که سر و کارشان با پول نیستند!) از یک روش ساده برای این کار استفاده می‌کنند. اسکناس‌ها را در سه دسته روی میز می‌گذارند. برای هر نوع اسکناس جایی تعیین می‌کنند و بعد هر اسکناس را در جایش و روی اسکناس‌های همنوعی می‌گذارند. وقتی این کار تمام شد دسته‌ها را به ترتیب روی هم می‌چینند.

این روش ساده، اساس اولین الگوریتم مرتب‌سازی است که به آن می‌پردازیم یعنی مرتب‌سازی شمارشی. قبل از هر چیز ببایید برای ساده‌تر شدن موضوع از این جا تا پایان درس یک قرارداد بگذاریم. هر وقت که با یک مسئله‌ی مرتب‌سازی مواجه می‌شویم آن را به صورت مرتب‌سازی یک آرایه از اعداد صحیح نامنفی مدل می‌کنیم و از این به بعد قرارداد می‌کنیم:

$$A = \text{آرایه‌ی ورودی که باید آن را مرتب کنیم} \quad n = (\text{طول } A) \text{ تعداد عناصر آرایه} \quad m = \text{بزرگترین عنصر موجود در آرایه}$$

## مرتب‌سازی شمارشی (Counting Sort)

با این توضیح به بررسی دقیق‌تر اولین الگوریتم که مرتب‌سازی شمارشی است می‌پردازیم. این الگوریتم فقط در صورتی قابل استفاده است که از مقدار بزرگ‌ترین عدد آرایه آگاه باشیم و اعداد آرایه صحیح و نامنفی باشند.  
در این روش از یک آرایه‌ی کمکی به نام Count استفاده می‌کنیم. به این صورت که تعداد تکرار هر عدد آرایه مانند آرا در Count[i] ذخیره می‌کنیم.

همه‌ی مقادیر این آرایه‌ی کمکی در آغاز صفر هستند. برای به دست آوردن مقادیر نهایی Count کافیست یک دور آرایه‌ی اصلی‌مان را از ابتدا تا انتهای پیمایش کرده و تعداد تکرارها را بشماریم.

```
In [1]: A = [1, 2, 2, 2, 1, 3, 3, 1, 2, 4, 5]
A پیدا کردن بزرگ‌ترین عنصر در آرایه
m = max(A) # ساخت آرایه کمکی # Count[m+1] با انداره m+1
Count = [0] * (m + 1) # شمردن تعداد تکرار هر عنصر A پیمایش آرایه اصلی
# for x in A:
    Count[x] += 1 # افزایش شمارنده برای عنصر x
print(Count) # چاپ آرایه Count که نشان‌دهنده تعداد تکرار هر عدد است
```

[0, 3, 4, 2, 1, 1]

حال که تعداد تکرارهای هر عدد را داریم، کافی است اندیس هر عدد را در آرایه‌ی مرتب شده‌ی مورد نظر به دست آوریم و اعداد را در اندیس‌های مربوط به خود قرار دهیم. در واقع عدد  $i$  در اندیس  $i$  Count[0]+Count[1]+...Count[i] تا Count[0]+Count[1]+...Count[i-1] قرار می‌گیرد.

دققت کنید این الگوریتم فقط در مواردی کار می‌کند که اعداد ورودی صحیح نامنفی باشند و حداقل آن‌ها ( $m$ ) خیلی زیاد نباشد، در غیر این صورت، نیاز به گرفتن یک حافظه‌ی خیلی بزرگ برای Count خواهیم داشت.

```
In [2]: A_sorted = []
آرایه نهایی مرتب شده استفاده می‌شود Count این حلقه برای ساخت آرایه مرتب شده بر اساس
# (بزرگ‌ترین عنصر) m برای هر عدد از 0 تا
for i in range(m + 1):
    به آرایه مرتب شده اضافه می‌کنیم (Count[i]) آن عدد را به تعداد تکرارش
    # اضافه می‌کنیم
    A_sorted += [i] * Count[i]
چاپ آرایه مرتب شده print(A_sorted)
```

[1, 1, 1, 2, 2, 2, 2, 3, 3, 4, 5]

## مرتب‌سازی درجی (Insertion Sort)

فرض کنید که اسکناس‌هایتان را (که برای سادگی مقدار آن‌ها را با ۱ و ۲ و ۳ نشان دادیم) و تعدادشان هم خیلی زیاد است مرتب کرده‌اید. مثلاً فرض کنید که به آرایه‌ی مرتب زیر رسیده‌اید: (فرض کنید تعداد عنصرها زیاد است!)

```
In [3]: A = [1, 1, 3, 4, 5, 6]
```

## مرتب‌سازی درجی (Insertion Sort)

فرض کنید که اسکناس‌هایتان را (که برای سادگی مقدار آن‌ها را با ۱ و ۲ و ۳ نشان دادیم) و تعدادشان هم خیلی زیاد است مرتب کرده‌اید. مثلاً فرض کنید که به آرایه‌ی مرتب زیر رسیده‌اید: (فرض کنید تعداد عنصرها زیاد است!)

حال فرض کنید که ناگهان بعد از مرتب‌کردن همه‌ی این اسکناس‌ها، متوجه می‌شوید که یک اسکناس دو تومانی ته جیبتان جا مانده است و می‌خواهید آن را به آرایه‌ی مرتب‌تان اضافه کنید. یک راهش این است که این اسکناس را به آخر آرایه اضافه کرده و دوباره همان الگوریتم مرتب‌سازی شمارشی را اجرا کنید ولی این راهی نیست که اکثر مردم می‌روند. راهی که بسیار طبیعی‌تر است آن است که جایی از دسته‌ی اسکناس‌هایمان را پیدا کنیم که می‌توانیم این اسکناس جدید را در آن اضافه کنیم، و بعد با کنار زدن بخشی از اسکناس‌ها برایش جا باز کنیم و آن را در آن وسط قرار دهیم. ما این الگوریتم را «مرتب‌سازی درجی» می‌نامیم و به این صورت پیاده‌سازی می‌کنیم:

در هر مرحله فرض می‌کنیم آرایه از اندیس صفر تا  $i$  مرتب شده است و عنصر  $i+1$  را به ترتیب با عناصر  $i$  تا صفر مقایسه می‌کنیم تا مکان مناسب آن را پیدا کنیم. (در هر مقایسه اگر این عنصر از عنصر قبلی کوچک‌تر باشد دو عنصر Swap می‌شوند). این کار به ازای هر کدام از عناصر آرایه به ترتیب از عنصر صفر تا آخر انجام می‌شود.

```
In [4]: A = [5, 2, -3, 4, 6, -7, 1, 9, 12, 5, -6]
آرایه ورودی نامرتب شروع از دومن عنصر (اندیس ۱) تا انتهای آرایه
for k in range(1, len(A)):
    عنصری که فرار است در جای صحیح خود درج شود item = A[k]
    در بخش مرتب شده است item نشان‌دهنده موقعیت فعلی i = k
    با عناصر قبلی در بخش مرتب شده و شیفت دادن عناصر بزرگ‌تر به راست item مقایسه
    # while i > 0 and A[i-1] > item:
        شیفت دادن عنصر بزرگ‌تر به سمت راست A[i] = A[i-1]
        حرکت به سمت چپ در بخش مرتب شده i -= 1
        در موقعیت صحیح خود item در جای item
        چاپ وضعیت آرایه پس از هر مرحله درج (برای مشاهده روند مرتب‌سازی)
        print(A)
        چاپ نهایی آرایه مرتب شده print(A)
```

```
[2, 5, -3, 4, 6, -7, 1, 9, 12, 5, -6]
[-3, 2, 5, 4, 6, -7, 1, 9, 12, 5, -6]
[-3, 2, 4, 5, 6, -7, 1, 9, 12, 5, -6]
[-3, 2, 4, 5, 6, -7, 1, 9, 12, 5, -6]
[-7, -3, 2, 4, 5, 6, 1, 9, 12, 5, -6]
[-7, -3, 1, 2, 4, 5, 6, 9, 12, 5, -6]
[-7, -3, 1, 2, 4, 5, 6, 9, 12, 5, -6]
[-7, -3, 1, 2, 4, 5, 6, 9, 12, 5, -6]
[-7, -3, 1, 2, 4, 5, 6, 9, 12, 5, -6]
[-7, -3, 1, 2, 4, 5, 6, 9, 12, 5, -6]
[-7, -6, -3, 1, 2, 4, 5, 6, 9, 12]
[-7, -6, -3, 1, 2, 4, 5, 6, 9, 12]
```

پیچیدگی زمانی این الگوریتم کاملاً وابسته به نوع داده‌ی ورودی است. اگر آرایه از پیش مرتب شده باشد الگوریتم در زمان خطی انجام می‌شود؛ چراکه هر عنصر فقط با عنصر قبلی خود مقایسه شده‌است. و در بدترین حالت اگر آرایه به صورت وارونه مرتب شده باشد مرتبه‌ی زمانی آن  $O(n^2)$ . همچنین به صورت کلی امید ریاضی مرتبه‌ی زمانی این مرتب‌سازی به ازای همه‌ی ورودی‌های ممکن  $O(n^2)$  است.

مرتب‌سازی درجی را یک مرتب‌سازی مقایسه‌ای می‌نامیم، چرا که در آن برای بدست آوردن ترتیب نهایی عناصر فقط از مقایسه‌کردن آنها بهره‌بردیم. به این فکر کنید که آیا مقایسه‌ای بودن همیشه یک مزیت است؟ مثلاً اگر درست برعکس اتفاق بالا می‌افتد چه؟ یعنی فرض کنید کلا ۳ نوع اسکناس ممکن داریم ولی ما میلیون‌ها نسخه از هر اسکناس داریم. آن وقت از چه روشی استفاده می‌کنید؟

تا کنون دو الگوریتم مختلف برای مرتب‌سازی دیده‌ایم، یکی از نکاتی که همیشه در طراحی الگوریتم باید به آن دقت داشته باشیم این است که باید بتوانیم ثابت کنیم که الگوریتممان پایان می‌پذیرد (این اثبات خیلی وقت‌ها بدیهی است و حذف می‌شود) و وقتی که به پایانش می‌رسد یک پاسخ درست تولید می‌کند. در مورد دو الگوریتم بالا ارائه‌ی یک استدلال برای درستی پاسخ تقریباً بدیهی است. الگوریتم بعدی کمی متفاوت است و به مفهوم جدیدی نگاه می‌کند:

در یک آرایه از اعداد، منظور از یک نا به جایی دو عدد است که برخلاف ترتیب طبیعی‌شان ظاهر شده‌اند. به عبارت دقیق‌تر در آرایه‌ی  $a$  به زوج مرتب  $(j, i)$  یک نا به جایی می‌گوییم هرگاه  $j < i$  و  $i > a[j] > a[i]$ .

به سادگی می‌توان دید که یک آرایه مرتب است اگر و تنها اگر هیچ نا به جایی نداشته باشد.

## مرتب‌سازی حبابی (Bubble Sort)

الگوریتم بعدی که آن را مرتب‌سازی حبابی می‌نامیم بر پایه‌ی تکرار یک روند خاص که ما آن را «حباب‌گیری» می‌نامیم کار می‌کند.

روند حباب‌گیری بسیار ساده‌است: در این روش هر بار از انتهای لیست شروع به جستجوی عددی می‌کنیم که از عدد قبل خود کوچک‌تر باشد. فکر کنید این یک حباب در لیست است. وقتی جای عدد کوچک‌تر را با عدد قبلی خود عوض می‌کنیم حباب به اول لیست نزدیک‌تر می‌شود. این جاچایی آنقدر ادامه می‌یابد تا عدد مورد نظر به ابتدای لیست برسد، یا از عدد قبل خود بزرگ‌تر باشد. در این صورت حباب می‌ترکد.

برای مرتب‌سازی کل آرایه به طول  $n$  حداقل به  $1 - n$  دور جستجوی لیست برای حباب‌ها نیاز داریم. چرا که بعد از اولین دور، کوچک‌ترین عدد لیست حتماً به ابتدای لیست می‌رسد، و به همین صورت پس از  $i$  امین دور جستجو،  $i$  امین عدد کوچک‌تر لیست به جایگاه خود می‌رسد. وقتی  $1 - n$  عدد در جای خود باشند حتماً عدد آخر هم در جای خود است.

```
In [7]: A = [5, 12, 3, 4, 7, 1, 0, 6, 19, 8, 13, 4, 2, 10, 16] # آرایه ورودی نامربوط
n = len(A) # طول آرایه

# بار تکرار می‌شود تا مطمئن شویم همه عناصر در جای خود قرار گرفته‌اند: حلقه بیرونی n-1
for i in range(n - 1):
    # ام پیمایش می‌کند i+1 حلقه درونی: از انتهای آرایه تا عنصر
    # هدف این است که کوچک‌ترین عنصر در هر دور به ابتدای بخش نامربوط "حباب" شود
    for j in range(n - 1, i, -1):
        # کوچک‌تر بود (A[j-1]) از عنصر قبلی خود (A[j]) اگر عنصر فعلی
        if A[j] < A[j - 1]:
            # عملیات swap
            A[j], A[j - 1] = A[j - 1], A[j] # Swap a bubble
    print(A) # چاپ وضعیت آرایه پس از هر دور حباب‌گیری (برای مشاهده روند مرتب‌سازی)
# کاملاً مرتب شده است A پس از اتمام حلقه‌ها، آرایه
```

```
[0, 5, 12, 3, 4, 7, 1, 2, 6, 19, 8, 13, 4, 10, 16]
[0, 1, 5, 12, 3, 4, 7, 2, 4, 6, 19, 8, 13, 10, 16]
[0, 1, 2, 5, 12, 3, 4, 7, 4, 6, 8, 19, 10, 13, 16]
[0, 1, 2, 3, 5, 12, 4, 4, 7, 6, 8, 10, 19, 13, 16]
[0, 1, 2, 3, 4, 5, 12, 4, 6, 7, 8, 10, 13, 19, 16]
[0, 1, 2, 3, 4, 4, 5, 12, 6, 7, 8, 10, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 12, 7, 8, 10, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 7, 12, 8, 10, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 12, 10, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 10, 12, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 10, 12, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 10, 12, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 10, 12, 13, 16, 19]
```

همان‌طور که در نتیجه‌ی اجرای الگوریتم بالا ملاحظه می‌کنید بعد از ۱۰ دور جستجوی لیست برای حباب‌ها، کل اعداد مرتب شده‌اند (کافی است به موقعیت عدد ۱۲ دقت کنید). پس پیاده‌سازی اولیه‌ی الگوریتم دارد زمان اضافه‌ای را صرف جستجوی لیست کاملاً مرتب می‌کند.

برای اصلاح این موضوع می‌توانیم در اولین دوری که حباب پیدا نکردیم اجرای الگوریتم را متوقف کنیم:

```
In [8]: A = [5, 12, 3, 4, 7, 1, 0, 6, 19, 8, 13, 4, 2, 10, 16] # آرایه ورودی نامرتب
print(A) # چاپ آرایه اصلی قبل از مرتبسازی
n = len(A) # طول آرایه
```

```
# بار تکرار می‌شود یا تا زمانی که آرایه مرتب شود n-1: حلقه بیرونی
for i in range(n - 1):
    bubble_found = False # پرجمی برای تشخیص اینکه آیا در این دور حبابی جایجا شده است با خبر
    # حلقه درونی: از انتهای بخش نامرتب به سمت ابتدای آن بیمایش می‌کند
    for j in range(n - 1, i, -1):
        # اگر عنصر فعلی از عنصر قبلی خود کوچکتر بود (عنی یک تابجایی بیدا شد)
        if A[j] < A[j - 1]:
            # جای آنها را با هم عوض کن (swap)
            A[j], A[j - 1] = A[j - 1], A[j]
            bubble_found = True # نشان می‌دهد که حداقل یک تابجایی در این دور انجام شده است
    if not bubble_found: # اگر در این دور هیچ تابجایی جایجا نشد، یعنی آرایه مرتب شده است
        break # توقف اجرای الگوریتم
print(A) # چاپ وضعیت آرایه پس از هر دور (برای مشاهده روند مرتبسازی)
# کاملاً مرتب شده است A پس از انعام حلقه‌ها، آرایه
```

```
[5, 12, 3, 4, 7, 1, 0, 6, 19, 8, 13, 4, 2, 10, 16]
[0, 5, 12, 3, 4, 7, 1, 2, 6, 19, 8, 13, 4, 10, 16]
[0, 1, 5, 12, 3, 4, 7, 2, 4, 6, 19, 8, 13, 10, 16]
[0, 1, 2, 5, 12, 3, 4, 7, 4, 6, 8, 19, 10, 13, 16]
[0, 1, 2, 3, 5, 12, 4, 4, 7, 6, 8, 10, 19, 13, 16]
[0, 1, 2, 3, 4, 5, 12, 4, 6, 7, 8, 10, 13, 19, 16]
[0, 1, 2, 3, 4, 4, 5, 12, 6, 7, 8, 10, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 12, 7, 8, 10, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 7, 12, 8, 10, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 12, 10, 13, 16, 19]
[0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 10, 12, 13, 16, 19]
```

تا کنون چند الگوریتم مختلف با دیدگاه‌های متفاوت برای مسئله‌ی مرتبسازی دیده‌ایم. این که از کدام یک باید استفاده کنیم به شرایط مسئله بستگی دارد و یکی از اهداف این درس این است که شما بتوانید به خوبی در این خصوص تصمیم بگیرید.

در ادامه چند ایده‌ی دیگر برای مرتبسازی را هم مطرح می‌کنیم.

## مرتبسازی انتخابی (Selection Sort)

ایده‌ای که در اینجا مطرح می‌کنیم بسیار ساده است. برای مرتب کردن یک آرایه کافیست که کوچکترین عنصر آن را پیدا کرده، جای آن را با اولین عنصر در لیست عوض کنیم. این کار را برای هر بخش نامرتب از آرایه تکرار می‌کنیم.

به طور دقیق‌تر، در دور اول، از بین تمام عناصر آرایه (از خانه ۰ تا  $n - 1$ )، کوچکترین عنصر را پیدا می‌کنیم و آن را با عنصر موجود در خانه اول (اندیس ۰) جایجا می‌کنیم. حالا کوچکترین عنصر در جایگاه نهایی خود قرار گرفته است.

در دور دوم، از بین عناصر باقیمانده (در خانه‌های ۱ تا  $n - 1$ )، کوچکترین عنصر را می‌یابیم و جای آن را با عنصر خانه‌ی دوم (اندیس ۱) عوض می‌کنیم. به همین ترتیب، در مرحله‌ی شماره‌ی  $i$  (که با اندیس  $i$  شروع می‌شود)، اعداد خانه‌های  $i$  تا  $n - 1$  را چک می‌کنیم و کوچکترین عدد را با عدد قرار گرفته در خانه‌ی  $i$  جایجا می‌کنیم.

کافی است این روال را برای  $1 \leq i < n$  انجام دهیم (وقتی که  $1 - n$  عدد به جای خود منتقل شوند قطعاً بزرگترین عدد هم در جای خود قرار گرفته است). این الگوریتم را مرتبسازی انتخابی می‌نامند. این روش تضمین می‌کند که در هر مرحله، یک عنصر به جایگاه نهایی خود در آرایه مرتب شده منتقل می‌شود.

این الگوریتم مرتب سازی را به صورت یکتابع ساده پیاده سازی می‌کنیم که با گرفتن لیست ورودی، آن را مرتب کند و برگرداند:

```
In [15]: # مرتبسازی انتخابی (Selection Sort)
```

```
def Selection_sort(A):
    """
    مرتب می‌کند (Selection Sort) را با استفاده از الگوریتم مرتبسازی انتخابی A این تابع آرایه در هر مرحله، کوچکترین عنصر باقیمانده را پیدا کرده و آن را در جایگاه صحیح خود قرار می‌دهد
    """
    n = len(A) # طول آرایه

    # حلقه بیرونی: از اولین عنصر تا عنصر یکی مانده به آخر بیمایش می‌کند
    # در هر دور، یک عنصر در جایگاه نهایی خود قرار می‌گیرد
    for i in range(n - 1):
        # اندیس کوچکترین عنصر باقیمانده را نگه می‌دارد
        index = i + 1 # اندیس کوچکترین عنصر باقیمانده را یافت
        # حلقه درونی: برای پیدا کردن کوچکترین عنصر در بخش نامرتب آرایه
        # تا انتهای آرایه بیمایش می‌کند (i+1) از عنصر بعدی
        for j in range(i + 1, n):
            # کوچکتر بود (A[index]) از کوچکترین عنصر یافت شده (A[j])
            if A[index] > A[j]:
                # اندیس کوچکترین عنصر را به روزرسانی می‌کند
                index = j

    # جایجا می‌کند (A[i]) کوچکترین عنصر یافت شده را با عنصر در جایگاه Swap (جایجا می‌کند): A[i], A[index] = A[index], A[i]
```

```

        آرایه مرتب شده را برمی‌گرداند #.
return A

#مثال استفاده:
my_array = [5, 2, -3, 4, 6, -7, 1, 9, 12, 5, -6]
sorted_array = Selection_sort(my_array)
print(f"Sorted List: {sorted_array}")

Sorted List: [-7, -6, -3, 1, 2, 4, 5, 5, 6, 9, 12]

```

حال می‌خواهیم آن را بر روی یک لیست فراخوانی کنیم:

```

In [16]: #مثال فراخوانی تابع مرتبسازی انتخابی
          منال فراخوانی تابع مرتبسازی انتخابی

          # تعریف یک لیست نامرتب #
myList = [12, 3, 15, -4, 7, 6, -1, 0, 11, 6]
print(f"List Asli: {myList}") # چاپ لیست قبل از مرتبسازی تابع

          # برای مرتب کردن لیست Selection_sort فراخوانی تابع #
          # باید قبلاً تعریف شده باشد توجه: تابع Selection_sort
myList = Selection_sort(myList)

print(f"Sorted List: {myList}") # چاپ لیست پس از مرتبسازی تابع

```

List Asli: [12, 3, 15, -4, 7, 6, -1, 0, 11, 6]  
Sorted List: [-4, -1, 0, 3, 6, 6, 7, 11, 12, 15]

پیچیدگی زمانی الگوریتم مرتبسازی انتخابی مستقل از نوع داده‌ی ورودی است؛ چراکه هیچ کدام از حلقه‌هایی که در الگوریتم آمد به نوع قرارگیری عناصر در آرایه وابسته نبودند. برای بررسی پیچیدگی زمانی این

مرتبسازی باید توجه داشت که برای پیدا کردن کوچکترین عنصر همه‌ی  $n$  عنصر بررسی می‌شوند. (1- $n$  مقایسه) برای دومین کوچکترین عنصر  $n-1$  عنصر بررسی می‌شوند و ... . پس نهایتاً تعداد مقایسه‌ها برابر

است با

$$(n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} = \theta(n^2)$$

## مرتبسازی پایدار و ناپایدار (Stable / Unstable Sort) (ناپایدار)

حال می‌خواهیم به یک خاصیت دیگر که در بعضی از الگوریتم‌های مرتبسازی وجود دارد پردازیم. بعضی الگوریتم‌های مرتبسازی پایدارند و برخی ناپایدار. یک الگوریتم مرتبسازی پایدار وقتی به دو عنصر با مقدار برابر بررسد ترتیب آن‌ها را حفظ می‌کند. مثلاً فرض کنید که یک صفت از همه‌ی دانشجویان کلاس تشکیل شده و می‌خواهیم آن‌ها را به ترتیب سن مرتب کنیم. اگر دو نفر باشند که سنشان یکسان باشد، در خروجی یک الگوریتم مرتبسازی درست ممکن است هر کدام از آن‌ها جلوتر از دیگری قرار گیرد ولی یک الگوریتم پایدار است اگر و تنها اگر ترتیب اولیه‌ی هر چنین زوجی را حفظ کند.

## مرتب سازی سطلی (Bucket Sort)

فرض کنید می‌خواهیم کارت‌های بازی Uno را مرتب کنیم. یعنی چهار رنگ کارت داریم که برای هر رنگ ۱۰ کارت وجود دارد به طوری که برای هر رنگ یکی از اعداد ۰ تا ۹ روی یک کارت نوشته شده است. می‌خواهیم کارت‌ها را جوری مرتب کنیم که همه کارت‌های آبی، سپس همه کارت‌های زرد، بعد همه سبزها و بعد قرمزها بیایند و کارت‌های هر رنگ به ترتیب از کوچکتر به بزرگتر قرار بگیرند.

ساده‌ترین راه جدا کردن کارت‌ها بر اساس رنگشان و مرتب کردن هر رنگ به صورت مجزا است. در واقع اگه رنگ‌ها را جدا نکنیم و سعی کنیم در یک دسته ترتیب ذکر شده را ایجاد کنیم کارمن به مراتب سخت‌تر خواهد بود.

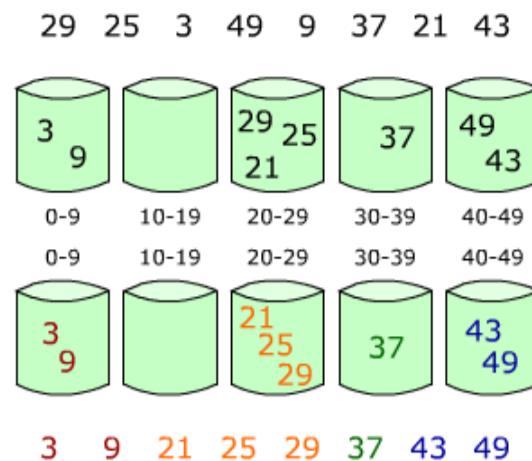
گاهی با مسئله‌هایی روبرو هستیم که این ایده به طور طبیعی در حل آن‌ها مفید واقع می‌شود. با همین هدف یک نوع مرتبسازی سطلی معرفی می‌کنیم که بر اساس این ایده است. به طور کلی مرتبسازی سطلی به شکل زیر است:

1. روش گذاشتن عناصرها در سطل‌ها را تعریف می‌کنیم و به تعداد مورد نیاز سطل در نظر می‌گیریم
2. پخش کردن: هر عنصر را در سطلش قرار می‌گیرد
3. عناصر هر سطر را مرتب کن
4. جمع‌آوری: به ترتیب سطل‌ها را نگاه می‌کنیم و عناصر را جمع‌آوری می‌کنیم

حال با گرفتن لیستی از عناصر، تعداد سطلهای لازم را پیدا می‌کنیم. بدین منظور بزرگترین عدد لیست ( $\max\_num$ ) را می‌یابیم. می‌توانیم به اندازه‌ی  $1 + \text{دهگان بزرگترین عدد لیست}$ ، سطل در نظر بگیریم.

مثلث فرض کنید لیستی از اعداد حداقل دورقمی صحیح و نامنفی داریم و می‌خواهیم آن را به این شیوه مرتب کنیم. می‌توانیم ابتدا ده سطل با شماره‌های صفر تا ۹ در نظر بگیریم، و هر عدد را بر مبنای رقم دهگان خود

در سطل مربوطه قرار دهیم. تابع ساده‌ی زیر با گرفتن یک عدد یک یا دورقمی، شماره سطل را بر اساس رقم دهگان تعیین می‌کند:



```
In [18]: def get_bucket(x):
    """
    برمی‌گرداند x این تابع شماره سطل را بر اساس رقم دهگان عدد
    مثلًا برای عدد 23، سطل شماره 2 را برمی‌گرداند.
    برای اعداد تکراری (مثل 5)، سطل شماره 0 را برمی‌گرداند.
    این تابع در مرتبسازی سطلهای برای تعیین سطل هر عنصر استفاده می‌شود
    """
    return x // 10
```

مثال: مقدارهای اولیه سطلهای برای مرتبسازی سطلهای #

```
A = [29, 25, 9, 49, 3, 37, 21, 43]
# آرایه ورودی # پیدا کردن بزرگترین عدد در آرایه
max_num = max(A) # تعیین تعداد سطلهای بر اساس رقم دهگان بزرگترین عدد
# پس 5 = 1 + 4 سطل (از 0 تا 4) نیاز داریم، باشد اگر max_num = 49
bucket_num = get_bucket(max_num) + 1 # ساخت لیست سطلهای که هر سطل خودش یک لیست خالی است
buckets = [[] for i in range(bucket_num)]
print(buckets) # چاپ لیست سطلهای خالی
```

```
[[], [], [], [], []]
```

اکنون می‌توانیم هر عدد را در سطل مربوط به خود بگذاریم:

قرار دادن هر عنصر در سطل مربوط به خودش: # مرحله پخش کردن

```
for i in A:
    # و اضافه کردن آن به انتهای سطل مربوطه i تعیین سطل برای عنصر
    buckets[get_bucket(i)] += [i]
    print(buckets) # چاپ وضعیت سطلهای پس از اضافه شدن هر عنصر (برای مشاهده روند پخش)
```

```
چاپ نهایی وضعیت سطلهای پس از پخش همه عناصر #
```

```
[[[], [], [29], [], []]
 [[], [], [29, 25], [], []]
 [[9], [], [29, 25], [], []]
 [[9], [], [29, 25], [], [49]]
 [[9, 3], [], [29, 25], [], [49]]
 [[9, 3], [], [29, 25], [37], [49]]
 [[9, 3], [], [29, 25, 21], [37], [49]]
 [[9, 3], [], [29, 25, 21], [37], [49, 43]]
 [[9, 3], [], [29, 25, 21], [37], [49, 43]]]
```

اگر عناصر داخل هر سطل را با استفاده از یک الگوریتم مقایسه‌ای (مثل مرتبسازی انتخابی) مرتب کنیم، و عناصر سطلهای را به ترتیب شماره سطل پشت سر هم قرار دهیم، آرایه‌ی اصلی را کاملاً مرتب کرده‌ایم. اگر

اعداد در یک بازه‌ی مشخص به صورت یکنواخت توزیع شده باشند به ازای  $n$  عدد  $n$  سطل در نظر می‌گیریم و به واسطه‌ی یک تابع اعداد را در سطلهای مربوط به خود قرار می‌دهیم. از آن جای که  $n$  عدد به صورت

یکنواخت پخش شده اند و  $n$  سطل هم وجود دارد در هر سطل  $O(1)$  عدد قرار می‌گیرد و نتیجتاً مرتب کردن اعداد هر سطل  $O(n)$  زمان می‌برد و چون هزینه‌ی پخش و جمع کردن عناصر  $O(n)$  است اعداد در

مرتب می‌شوند. البته در بدترین حالت یعنی زمانی که همه‌ی اعداد در یک سطل قرار گیرند مرتبه‌ی زمانی الگوریتم  $O(n^2)$  است و در واقعاً فرقی با مرتب‌سازی‌های مقایسه‌ای ندارد. برای اثبات دقیق به

کتاب CLRS مراجعه شود.

چاپ وضعیت فعلی سطلهای پس از مرحله پخش کردن #

```
print(buckets)

# مرتب کردن عناصر داخل هر سطل: # مرحله مرتبسازی
# این حلقه بر روی هر سطل بیمایش می‌کند
for i in range(bucket_num):
    # مرتب می‌کنیم Selection_sort عناصر داخل هر سطل را با استفاده از تابع
    # باید قبل از تعریف شده باشد و در دسترس باشد تابع: Selection_sort توجه:
    buckets[i] = Selection_sort(buckets[i])
```

```
چاپ وضعیت سطلهای پس از مرتبسازی هر سطل #
print(buckets)
```

```
[[9, 3], [], [29, 25, 21], [37], [49, 43]]
 [[3, 9], [], [21, 25, 29], [37], [43, 49]]]
```

```
# جمع کردن عناصر از سطلهای مرتب شده به یک آرایه نهایی: Gathering
# آرایه نهایی مرتب شده # []
# بیمایش بر روی هر سطل
for i in range(bucket_num):
    عناصر هر سطل را به انتهای آرایه نهایی اضافه می‌کنیم
    A_sorted_final += buckets[i]
print(A_sorted_final) چاپ آرایه نهایی کاملاً مرتب شده #
```

[3, 9, 21, 25, 29, 37, 43, 49]

مرتب سازی سطلی حالت کلی مرتب سازی شمارشی است و اگر تعداد عناصر داخل هر سطل حداقل 1 باشد دقیقاً همان مرتب سازی شمارشی خواهد بود.

باید دقت کنیم که در این مرتب سازی هم از اعمال غیر مقایسه‌ای استفاده کردیم. در واقع بخشی از مرتب سازی با تقسیم‌بندی کارت‌ها به چند دسته و بدون مقایسه آن‌ها با هم‌دیگر انجام می‌شود. اگر کمی دقت کنیم می‌بینیم که مرتب سازی‌هایی که مقایسه‌ای نیستند برای بهینه بودن نیازمند فرض‌های اضافه‌ای هستند. برای مثال بالاتر گفتیم که مقایسه‌ای بودن همیشه یک مزیت نیست مثلاً در حالتی که انواع اسکناس‌ها محدود باشد. اما فرض کنید می‌خواهیم ۱۰۰ نامه را بر حسب کد منطقه مرتب کنیم که کد منطقه برای هر نامه یک عدد چهار رقمی است. این بار استفاده از مرتب سازی‌های غیر مقایسه‌ای گفته شده بهینه است؟

## مرتب سازی مبنایی (Radix Sort)

مرتب سازی مبنایی یک تعمیم طبیعی از مرتب سازی سطلی است. ایده‌ی کلی از این قرار است: پیش‌تر در مرتب سازی سطلی گفتیم که اشیا داخل هر سطل را به یک روش دلخواه مرتب می‌کنیم. حالا فرض کنید این روش دلخواه دوباره یک نوع مرتب سازی سطلی باشد.

به طور خاص وقتی این ایده را برای مرتب سازی اعداد در مبنای ۲ به کار می‌بریم، به آن مرتب سازی مبنایی می‌گوییم. حالا به شرح دقیق این الگوریتم می‌پردازیم.

فرض کنید تعدادی عدد صحیح نامنفی حداقل ۲ رقمی در مبنای ۲ به ما داده شده است و می‌خواهیم آن‌ها را مرتب کنیم.

یک راه این است که به طور بازگشتی الگوریتمی که در قسمت مرتب سازی سطلی توضیح داده شد را اجرا کنیم: الگوریتم ابتدا ۲ سطل در نظر می‌گیرد و اعداد را بر حسب بزرگترین رقم در سطل‌ها قرار می‌دهد. می‌دانیم که اگر اعداد داخل سطل‌ها به یک روش دلخواه مرتب شوند می‌توانیم آن‌ها را به ترتیب از سطل شماره ۱ تا سطل شماره ۲ جمع کنیم به طوری که هر وقت یک سطل خالی شد سراغ سطل بعدی برویم. الگوریتم به همین منظور خودش را روی اعداد هر ۲ سطل با صرف نظر کردن از رقم پارازششان که در همه عده‌های یک سطل یکسان است، صدا می‌زند. اعداد داخل هر سطل به این ترتیب مرتب می‌شوند. سپس الگوریتم جمع آوری از سطل‌ها را انجام می‌دهد و آرایه‌ی مرتب شده به دست می‌آید.

الگوریتم مرتب سازی مبنایی را می‌توان به صورت غیر بازگشتی هم پیاده کرد:

ابتدا اعداد را بر اساس رقم یکان به روش سطلی (یا شمارشی) مرتب کنیم. سپس اعداد را بر اساس رقم دهگان به روش سطلی مرتب کنیم، اما مطمئن باشیم این مرتب سازی پایدار است. با این حساب اگر دو عدد رقم دهگان مساوی داشته باشند، عددی که رقم یکان کمتری دارد (و پس از مرتب سازی بر اساس یکان در دور قبل زودتر قرار گرفت) همچنان در لیست زودتر می‌آید. با این حساب تمام اعداد بر اساس دو رقم یکان و دهگان مرتب خواهند شد. اگر همین کار را برای صدگان، هزارگان و ... تکرار کنیم تمامی اعداد مرتب خواهند شد.

دقت کنید اگر هر بار که بر اساس یک رقم مرتب می‌کنیم از مرتب سازی سطلی استفاده نکنیم و مثلاً از یک مرتب سازی مقایسه‌ای استفاده کنیم الگوریتم کند خواهد شد.

شاید بد نباشد به اثبات درستی این دو الگوریتم پردازیم. اما قبل از آن خوب هست توجه کنیم که چه طور این دو پیاده سازی مرتب سازی مبنایی هر دو از استقراء استفاده می‌کنند. در حالت بازگشتی اینطور به سوال نگاه می‌کنیم که فرض می‌کنیم که مرتب کردن اعداد با ۱-۲ رقم را بدلیم. می‌خواهیم با استفاده از آن اعداد با ۱-۲ رقم را مرتب کنیم. تابع اعداد را در سطل‌ها می‌ریزد. سپس با فرض اینکه بدلیم اعداد با حداقل ۱-۲ رقم را مرتب کنیم رقم با ارزش اعداد در هر سطل را که در همه عده‌های یک سطل یکسان است در نظر نمی‌گیریم و اعداد هر سطل را به صورت مرتب شده تحویل می‌گیرد و بعد هم به جمع‌آوری اعداد از درون سطل‌ها می‌پردازد.

در روش مستقیم هم در از تفکر استقرایی به این روش استفاده می‌کنیم: فرض کنیم بلدیم یک آرایه که اعدادش حداقل ۱-۲ رقم دارند را مرتب کنیم. می‌خواهیم از آن استفاده کنیم و آرایه‌ی که اعداد توی آن حداقل ۱-۲ رقم دارند را مرتب کنیم. همچنین فرض می‌کنیم که این روشی که برای مرتب سازی اعداد با حداقل ۱-۲ رقم استفاده می‌شود پایدار است. در واقع این هم جزئی از فرض استقرایی است.

بنابراین اگر می‌خواهیم به طور استقرایی از این روش استفاده کنیم و اعداد ۲ رقمی را مرتب کنیم مرتب سازی ما برای اعداد ۲ رقمی هم باید پایدار باشد.

فرض کنید آرایه‌ای که اعدادش ۲ رقم دارند داده شده است. بزرگترین رقم هر عدد را در نظر نمی‌گیریم و اعداد را بر حسب ۱-۲ رقم اولشان به همان روشی که فرض کردیم بلدیم مرتب می‌کنیم. حال در آرایه جدید اعداد را فقط بر اساس رقم پر ارزششان مرتب می‌کنیم. به عنوان تمرین با یک حالت‌بندی ساده نشان دهید که برای هر دو عدد متفاوت در آرایه اولیه در این آرایه که به دست آمده است، اعداد به ترتیب درستی قرار گرفته‌اند.

همانطور که می‌بینید روش طراحی کردن یک الگوریتم می‌تواند بسیار شبیه به روش اثبات درستی آن باشد. و این روش اکثرای نگاه استقرایی است.

در زیر ابتدا کد غیر بازگشتی سپس کد بازگشتی آمده است. به عنوان تمرین می‌توانید یک بار از اول کد بازگشتی را جوری بزنید که تابع بازگشتی برای هر مبنایی که در ورودی‌ش داده می‌شود کار کند.

In [23]:

```

آرایه ورودی برای مرتبسازی مبنایی # [1523, 1, 19, 3229, 4, 16, 25, 909, 223, 1648]
پیدا کردن بزرگترین عدد در آرایه برای تعیین تعداد دورها # (A)
شروع از رقم یکان (راستترین رقم) # radix = 1

حلقه اصلی: تا زمانی که همه ارقام (از یکان تا بزرگترین رقم) بررسی شوند #
while radix <= max_num:
    سطل (لیست) برای ارقام ۰ تا ۹ # سطل Distribution
    اعداد را بر اساس رقم فعلی به سطلهای منتقل می‌کنیم: (مرحله پخش کردن)
    for i in A:
        # محاسبه رقم فعلی # (عدد / radix) % 10
        # مثال: برای radix=1: (1523/1)%3 = 10
        # مثال: برای radix=10: (1523/10)%2 = 10
        # مثال: برای radix=100: (1523/100)%5 = 10
        # مثال: برای radix=1000: (1523/1000)%1 = 10
        B[int((i / radix) % 10)] += [i]

    لیست اصلی را پاک می‌کنیم تا اعداد مرتب شده را در آن جمع کنیم #
    A = [] + B
    عناصر را از سطلهای به ترتیب جمع می‌کنیم: (Collection)
    for i in range(10):
        A += B[i] # به لیست i اضافه کردن تمام عناصر از سطل
    برای دور بعدی، به رقم بعدی (دهگان، صدگان، ...) می‌رویم # radix *= 10
print(A) # چاپ نهایی آرایه مرتب شده

```

[1, 4, 16, 19, 25, 223, 909, 1523, 1648, 3229]

In [26]:

```

def bucket_sort(A, n):
    """
    با استفاده از ایده سطلی (Radix Sort) پیاده‌سازی بازگشتی مرتبسازی مبنایی.
    ام (از راست) مرتب می‌کند n را بر اساس رقم A این تابع آرایه.

    Args:
        A (list): آرایه‌ای از اعداد صحیح نامتفق برای مرتبسازی.
        n (int): شماره رقم فعلی که بر اساس آن مرتبسازی انجام می‌شود (از 1 به بالا).
            باید، بر اساس یکان مرتب می‌کند n=1 مثلاً اگر
            باشد، بر اساس صدگان مرتب می‌کند n=3 اگر.

    Returns:
        list: آرایه مرتب شده.
    """
    if n == 0:
        # اگر به رقم 0 رسیدیم (یعنی همه ارقام بررسی شدند)، آرایه مرتب شده است.
        return A

    # ایجاد 10 سطل برای ارقام ۰ تا ۹
    buckets = []
    for i in range(10): # سطلهای از ۰ تا ۹
        buckets.append([]) # هر سطل یک لیست خالی است

    # ام در سطلهای مربوطه قرار می‌دهیم n عناصر را بر اساس رقم: (Distribution)
    for x in A:
        # ام را محاسبه رقم # (عدد / توان) / (n-1)) % 10
        # مثال: برای عدد 203 و n=3: ((2**10) / 203) % 2 = 10 % 2 = 10 % (100 / 203) = 10
        bucket_number = int(x / (10 ** (n - 1))) % 10
        buckets[bucket_number].append(x)

    # ام مرتب می‌کنیم n مرحله مرتبسازی بازگشتی: هر سطل را بر اساس رقم #
    for i in range(10):
        # فراخوانی بازگشتی تابع برای مرتب کردن عناصر داخل سطل
        # به سراغ رقم با ارزش کمتر می‌رویم # با کاهش n به n-1
        buckets[i] = bucket_sort(buckets[i], n - 1)

    # عناصر را از سطلهای مرتب شده جمع می‌کنیم: (Collection)
    res = []
    for i in range(10):
        res.extend(buckets[i]) # اضافه کردن عناصر هر سطل به لیست نهایی
    return res

if __name__ == "__main__":
    # مثال استفاده از تابع مرتبسازی مبنایی بازگشتی
    A = [203, 132, 150, 205, 34, 2, 244, 241]
    # برای مرتب کردن این اعداد، باید بزرگترین تعداد رقم را مشخص کنیم
    # را ۳ قرار می‌دهیم n بزرگترین عدد 244 است که ۳ رقم دارد. بس
    print(bucket_sort(A, 3))

```

[2, 34, 132, 150, 203, 205, 241, 244]

## برای مطالعه بیشتر

تا اینجا ما فرض کردیم که تمام داده‌هایی که می‌خواهیم روی آنها الگوریتم را اجرا کنیم با  $O(1)$  به آنها دسترسی داریم. این یعنی فرض کردیم که تمام داده‌ها در حافظه اصلی (RAM) جا می‌شوند و دسترسی به هر بخش از آنها زمان ثابتی می‌برد.

اما گاهی اوقات، به خصوص در مقیاس‌های بزرگ‌تر، با داده‌هایی برخورد می‌کنیم که حجمشان آنقدر زیاد است که در حافظه رم جا نمی‌شوند. به این نوع داده‌ها، «داده‌های حجیم» (Massive Data) می‌گویند. در این شرایط، مجبوریم بخش‌هایی از داده‌ها را از دیسک (Magnetic Disk) به رم بیاوریم، عملیات مورد نظر را روی آنها انجام دهیم و سپس در صورت نیاز، آنها را دوباره به دیسک برگردانیم.

مثالهایی از این دست داده‌ها را می‌توان در شبکه‌های تلفن همراه (برای تحلیل ترافیک یا داده‌های کاربران)، سیستم‌های مالی بزرگ، یا پردازش تصاویر ماهواره‌ای دید. این داده‌ها به قدری عظیم هستند که حتی ابرکامپیوترها هم نمی‌توانند همه آن‌ها را یکجا در رم نگه دارند.

در این نوع مسائل، گلوگاه اصلی (یعنی جایی که بیشترین زمان را از ما می‌گیرد) دسترسی به دیسک است. زمان اجرا روی رم در مقایسه با زمان خواندن/نوشتن از دیسک، اهمیت چندانی ندارد. به همین دلیل، هدف اصلی در طراحی الگوریتم‌ها برای داده‌های حجمی،  $*\text{كمینه}(\text{کردن تعداد دسترسی‌ها})$  به دیسک $**$  است.

در این دیدگاه جدید به مسئله، الگوریتم‌های مرتب‌سازی هم تغییر می‌کنند. این الگوریتم‌ها گاهی بسیار شبیه به الگوریتم‌های توضیح داده شده هستند، اما برخی اجزای اضافی به آن‌ها اضافه می‌شود. مثلًا، در طراحی این الگوریتم‌ها، باید به پارامترهایی مثل  $**\text{سایز رم}**$  (کل دیتایی که رم ظرفیت دارد) و  $**\text{سایز بلاکها}**$  (تعداد داده‌هایی که در هر دسترسی به دیسک می‌توان به صورت یکجا و پشت سر هم به رم آورد) توجه ویژه‌ای شود. هر بار انتقال یک بلاک داده از دیسک به رم، خود به عنوان یک دسترسی به دیسک و یک هزینه زمانی محسوب می‌شود.

In [ ]:

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل دوم، بخش دوم: مقایسه الگوریتم‌های مرتب‌سازی

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

پویانمایی الگوریتم‌های مرتب‌سازی

مقایسه زمان اجرای الگوریتم‌های مرتب‌سازی

# پویانمایی الگوریتم‌های مرتب‌سازی

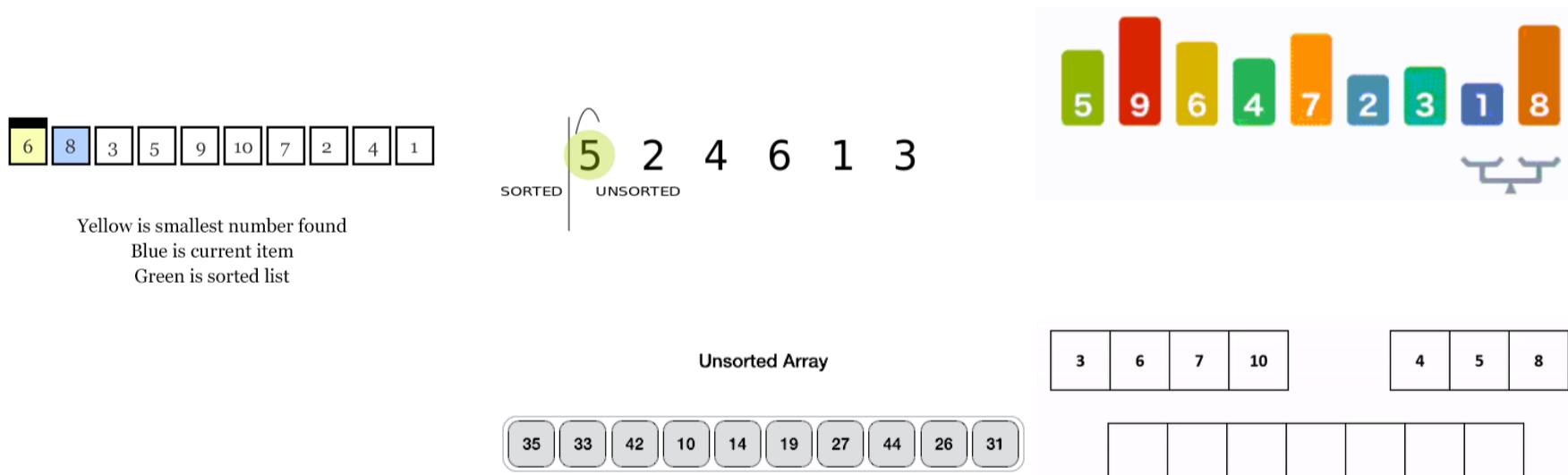
در این بخش، ابتدا سعی می‌کنیم یک نمایش گرافیکی از الگوریتم‌های مختلف مرتب‌سازی ارائه دهیم، زمان اجرای الگوریتم‌های مختلف مرتب‌سازی را برای داده‌های یکسان مقایسه خواهیم کرد.

ابتدا محیط Jupyter را برای رسم نمودار آماده می‌کنیم. بدین منظور از دو دستور زیر استفاده می‌کنیم. دستور اول باعث می‌شود نمودارها به جای نمایش در یک پنجره‌ی جداگانه، در همین دفترچه Jupyter رسم شوند. دستور دوم، تابع رسم نمودار را از کتابخانه‌ی Matplotlib بازگذاری می‌کند.

In [1]:

```
%matplotlib inline  
from matplotlib.pyplot import *
```

یادآوری: هر کدام از این میشنهای زیر نشان‌دهنده کدام روش مرتب‌سازی است؟



حال یک تابع به نام `array_plot` تعریف می‌کنیم که می‌تواند با دریافت وضعیت یک آرایه در حال مرتب‌سازی، آن را به صورت گرافیکی نمایش دهد. این تابع علاوه بر دریافت خود آرایه، دو اندیس  $i$  و  $j$  را دریافت می‌کند و عدددهای قرار گرفته در این دو خانه از آرایه را با رنگ متفاوت نمایش می‌دهد. این رنگ متفاوت نشانه‌ی آن است که بر روی این دو عدد پردازش دارد صورت می‌گیرد (رنگ سبز به معنی مقایسه‌ی دو عدد و رنگ‌های قرمز و نارنجی به معنی جابه‌جایی آن‌ها در حین اجرای الگوریتم است).

برای رسم بهتر نمودار از کتابخانه‌های `time` برای ایجاد وقفه در به روز رسانی نمودار بین هر دو عمل متوالی و `display` برای پاک کردن گام قبل در نمودار و ترسیم گام بعدی استفاده می‌کنیم.

In [1]:

```
from IPython import display  
import time
```

```
def array_plot(A, i, j, swap=False):  
    """
```

این تابع وضعیت یک آرایه را به صورت گرافیکی نمایش می‌دهد.  
عناصر در حال پردازش (مقایسه یا جابه‌جایی) با رنگ متفاوت مشخص می‌شوند.

Args:

آرایه‌ای از اعداد که در حال مرتب‌سازی است.  
اندیس اولین عنصری که در حال پردازش است.  
اندیس دومین عنصری که در حال پردازش است  $j$ .

باشد، نشان‌دهنده عملیات جابجایی است (رنگ قرمز و نارنجی) True اگر  
باشد، نشان‌دهنده عملیات مقایسه است (رنگ سبز) False اگر

```
"""
پاک کردن نمودار قبلی # clf()
color = ["blue"] * len(A)

if (swap):
    در حال انجام است، رنگ‌ها را قرمز و نارنجی کن (swap) اگر عملیات جابجایی
    # اگر swap کن
    color[i], color[j] = "orange", "red"
else:
    اگر عملیات مقایسه در حال انجام است، رنگ‌ها را سبز کن
    color[i] = color[j] = "green"

برای نمایش وضعیت آرایه (bar plot) رسم نمودار میله‌ای
bar(range(len(A)), A, color=color)

# نمایش نمودار در خروجی Jupyter
display.display(gcf())

پاک کردن خروجی قبلی و آماده‌سازی برای نمایش بعدی (بدون چشمک زدن)
display.clear_output(wait=True)

ایجاد یک وقفه کوتاه برای مشاهده پویانمایی
time.sleep(0.10)
```

می‌توانیم با فراخوانی این تابع در حین اجرای یک الگوریتم مرتب‌سازی، یک نمایش گرافیکی از روند اجرای الگوریتم ارائه دهیم. در مثال زیر روال اجرای الگوریتم مرتب‌سازی حبابی بر روی یک داده‌ی ورودی به تصویر کشیده می‌شود. برای آن که روند اجرای الگوریتم را ببینید باید علاوه بر قطعه‌کدهای بالا، قطعه کد زیر را هم اجرا کنید تا فرایند مرتب‌سازی را به صورت پویا ببینید.

In [3]: پویانمایی الگوریتم مرتب‌سازی حبابی #

```
# وارد کردن کتابخانه‌های لازم برای رسم نمودار و کنترل زمان
from IPython import display
import time
from matplotlib.pyplot import clf, bar, gcf # وارد کردن توابع خاص از matplotlib.pyplot

def array_plot(A, i, j, swap=False):
    """
    این تابع وضعیت یک آرایه را به صورت گرافیکی نمایش می‌دهد.
    عناصر در حال پردازش (مقایسه یا جابجایی) با رنگ متفاوت مشخص می‌شوند.

    Args:
        A (list): آرایه‌ای از اعداد که در حال مرتب‌سازی است.
        i (int): اندیس اولین عنصری که در حال پردازش است.
        j (int): اندیس دومین عنصری که در حال پردازش است.
        swap (bool): باشد، نشان‌دهنده عملیات جابجایی است (رنگ قرمز و نارنجی) True اگر
                    باشد، نشان‌دهنده عملیات مقایسه است (رنگ سبز) False اگر
    """

    # پاک کردن نمودار قبلی # clf()
    color = ["blue"] * len(A)

    if (swap):
        در حال انجام است، رنگ‌ها را قرمز و نارنجی کن (swap) اگر عملیات جابجایی
        # اگر swap کن
        color[i], color[j] = "orange", "red"
    else:
        اگر عملیات مقایسه در حال انجام است، رنگ‌ها را سبز کن
        color[i] = color[j] = "green"

    برای نمایش وضعیت آرایه (bar plot) رسم نمودار میله‌ای
    bar(range(len(A)), A, color=color)

    # نمایش نمودار در خروجی Jupyter
    display.display(gcf())

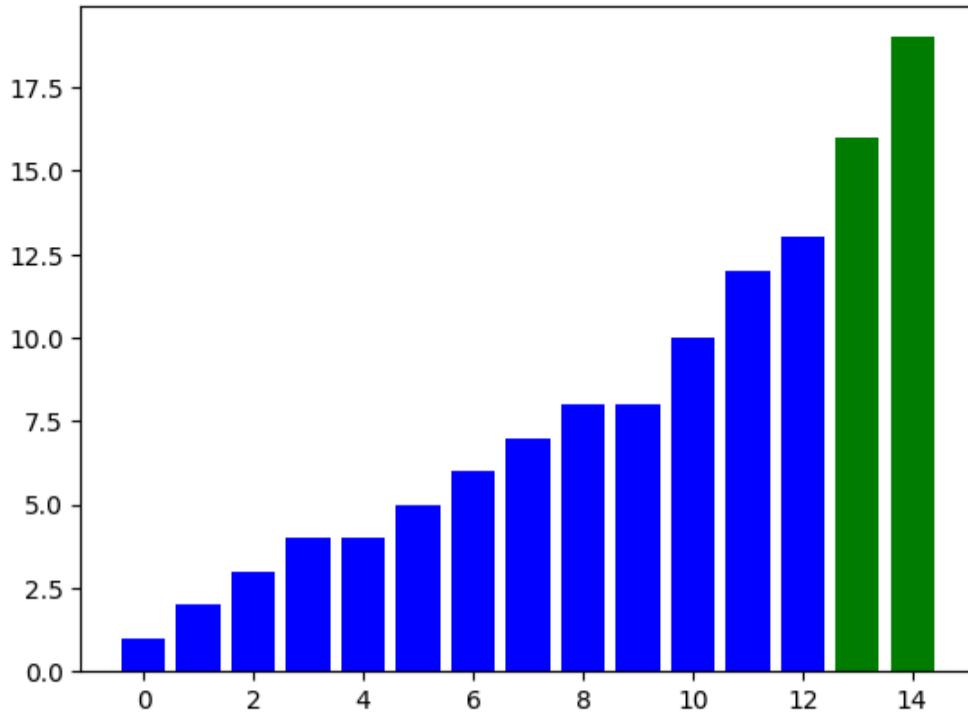
    # پاک کردن خروجی قبلی و آماده‌سازی برای نمایش بعدی (بدون چشمک زدن)
    display.clear_output(wait=True)

    ایجاد یک وقفه کوتاه برای مشاهده پویانمایی
    time.sleep(0.10)

    # آرایه ورودی برای مرتب‌سازی
    A = [5, 12, 3, 4, 7, 1, 8, 6, 19, 8, 13, 4, 2, 10, 16]
    n = len(A) # طول آرایه

    # بار تکرار می‌شود تا مطمئن شویم همه عناصر در جای خود قرار گرفته‌اند n-1: حلقه بیرونی
    for i in range(n - 1):
        ام پیمایش می‌کند i+1 حلقه درونی: ار انتهای آرایه تا عنصر #
        هدف این است که کوچکترین عنصر در هر دور به ابتدای بخش نامرتب "حباب" شود #
        for j in range(n - 1, i, -1):
            نمایش وضعیت آرایه قبل از مقایسه/جابجایی (رنگ سبز برای مقایسه)
            array_plot(A, j, j - 1)

            کوچکتر بود (یعنی یک نابجایی پیدا شد) (A[j-1] از عنصر قبلی خود
            if A[j] < A[j - 1]:
                # swap) جای آنها را با هم عوض کن
                A[j], A[j - 1] = A[j - 1], A[j]
            نمایش وضعیت آرایه پس از جابجایی (رنگ قرمز و نارنجی برای جابجایی)
            array_plot(A, j, j - 1, True)
```



به عنوان تمرین می‌توانید فرایند اجرای سایر الگوریتم‌های مرتب‌سازی را با همین روش به صورت پویانمایی ترسیم کنید.

در صورت لزوم می‌توانید روش رنگ‌آمیزی مقادیر مختلف آرایه را بسته به نوع الگوریتم تغییر دهید (مثلاً به جای استفاده از رنگ متفاوت برای دو عدد، یک سری عدد را متفاوت رنگ کنید).

## مقایسه‌ی زمان اجرای الگوریتم‌های مرتب‌سازی

برای مقایسه‌ی الگوریتم‌های مرتب‌سازی، ابتدا همه‌ی آن‌ها را به صورت چند تابع جدا از هم در فایل‌های نوشته شده در جلسات قبل در فایل `mysorts.py` نوشته‌ایم. این فایل را می‌توانید در پوشه‌ی `src/codes` که کنار این دفترچه قرار دارد مشاهده کنید.

در پیاده‌سازی این الگوریتم‌ها تفاوت مختصری ایجاد شده، بدین صورت که از یک آرایه‌ی کمکی به نام `res` استفاده کرده و نتیجه‌ی مرتب‌سازی را داخل آن می‌ریزیم. این کار باعث می‌شود ورودی تغییر نکند، در غیر این صورت هنگامی که بخواهیم چند الگوریتم را بر روی یک ورودی اجرا کنیم الگوریتم اول ورودی را مرتب می‌کند و الگوریتم‌های بعدی با یک آرایه‌ی از پیش مرتب شده روبرو خواهند بود.

برای بارگذاری این توابع از `mysorts.py`، از دستور زیر استفاده می‌کنیم:

In [4]: `from src.codes.mysorts import *`

حال دو تابع تعریف می‌کنیم: تابع `get_time` که با دریافت یک تابع به نام `f` و یک ورودی به نام `input`، تابع را بر روی ورودی اجرا می‌کند و مدت زمانی که اجرای تابع روی آن ورودی طول می‌کشد را بر حسب میکروثانیه برمی‌گرداند.

In [4]: `import timeit`

```
def get_time(f, input):
    """
    محاسبه می‌کند (input) را بر روی یک ورودی (f) این تابع زمان اجرای یک تابع
    برگردانده می‌شود (microseconds) زمان بر حسب میکروثانیه.

    Args:
        f (function): تابعی که قرار است زمان اجرای آن اندازه‌گیری شود.
        input: داده می‌شود f ورودی که به تابع.

    Returns:
        float: بر حسب میکروثانیه input بر روی f زمان اجرای تابع.
    """
    start = timeit.default_timer() # ثبت می‌کند
    f(input) # اجرا می‌کند input را با ورودی f تابع
    stop = timeit.default_timer() # ثبت می‌کند
    return (stop - start) * 1000 * 1000 # میکروثانیه
```

تابع دوم که `get_avg_time` نام دارد با دریافت تابع `f` و ورودی `input` متوسط زمان اجرای تابع `f` روی ورودی را به دست می‌آورد.

بدین منظور در هر «مرحله» تابع `f` را به تعداد ۲ بار روی ورودی فرا می‌خواند و متوسط زمان اجرای آن را به ازای این ۲ بار اجرا محاسبه می‌کند. این کار را ۸ مرحله تکرار می‌کند و نهایتاً از همه‌ی ۸ مرحله حداقل می‌گیرد.

این کار به خاطر آن است که پردازنده‌ی کامپیوتر به طور هم‌زمان ممکن است مشغول کارهای دیگری هم باشد. بدین ترتیب سعی می‌کنیم اثر پردازش‌های دیگر روی زمان محاسبه شده را به حداقل برسانیم. (نوع انجام این فرایند قادری سلیقه‌ای است. مثلاً اگر پردازنده مشغول محاسبات دیگری نباشد شاید حتی یک بار محاسبه‌ی زمان اجرای تابع `f` روی ورودی هم کافی باشد)

In [5]:

```
def get_avg_time(f, input):
    """
    محاسبه می‌کند input را روی ورودی f این تابع متوسط حداقل زمان اجرای تابع.
    برای کاهش اثر نویزهای سیستم، تابع را چندین بار اجرا کرده و حداقل میانگین‌ها را برمی‌گرداند.

    Args:
        f (function): تابعی که قرار است زمان اجرای آن اندازه‌گیری شود.
        input: داده می‌شود f ورودی که به تابع.

    Returns:
        float: حداقل متوسط زمان اجرای تابع بر حسب میکروثانیه.

    """
    res = 1 << 30 # یک عدد بسیار بزرگ
    s = 10 # تعداد مراحل برای تکرار کل فرآیند و انتخاب حداقل
    r = 2 # تعداد دفعات اجرای تابع در هر مرحله برای گرفتن میانگین

    بار تکرار می‌شود s: حلقه بیرونی.
    for i in range(s):
        مجموع زمان‌های اجرا در این مرحله = 0 # را اجرا کرده و زمان‌ها را جمع می‌کند f بار تابع
        # را اجرا کرده و زمان‌ها را جمع می‌کند f بار تابع
        for j in range(r):
            sum_times += get_time(f, input) # فراخوانی تابع get_time

    محاسبه میانگین زمان در این مرحله و به روزرسانی حداقل کلی
    res = min(res, sum_times / float(r))
return res
```

نوبت به تعریف دو تابع برای رسم نمودار زمان اجرای هر الگوریتم بر روی ورودی‌های مختلف است.

در اینجا دو تابع plot1 و plot2 تعریف کرده‌ایم که اولی سه الگوریتم و دومی پنج الگوریتم مرتب‌سازی را بر روی ورودی‌های داده شده اجرا و زمان آن را رسم می‌کند.

توجه کنید که هر الگوریتم قرار است بر روی آرایه‌های با اندازه‌های مختلف اجرا و زمان آن محاسبه شود. پس ورودی‌های هریک از دو تابع plot1 و plot2 عبارتند از:

یک آرایه از اندازه‌های مختلف ورودی. عدد  $N_i$  اندازه‌ی ورودی  $i$  ام را مشخص می‌کند.

اگر نوع مقداردهی  $N$  برایتان روش نیست بلافاصله بعد از تعریف این دو تابع روش مقداردهی آن‌ها را خواهید دید.

In [6]:

```
def plot1(N, numbers):
    """
    این تابع زمان اجرای سه الگوریتم مرتب‌سازی (Insertion, Selection, Bubble) را بر روی آرایه‌های با اندازه‌های مختلف محاسبه و نمودار آن را رسم می‌کند.

    Args:
        N (list): لیستی از اندازه‌های مختلف ورودی (مثلًا [100, 200, 300]).
        numbers (list of lists): لیستی از آرایه‌های ورودی، که هر آرایه طولی برابر با دارد N اندازه‌ی مشخص شده در.

    """
    figure(figsize=(20, 10)) # ایجاد یک شکل (figure)
    xlabel("Size of array (n)") # برچسب محور X
    ylabel("Time") # برچسب محور Y

    # برای هر ورودی در numbers محاسبه زمان اجرای Insertion Sort
    y = [get_avg_time(insertion_sort, a) for a in numbers]
    # با رنگ قرمز و ضخامت خط 5 رسم نمودار
    plot(N, y, 'r', label='Insertion Sort', linewidth=5)

    # برای هر ورودی در numbers محاسبه زمان اجرای Selection Sort
    y = [get_avg_time(selection_sort, a) for a in numbers]
    # با رنگ آبی و ضخامت خط 5 رسم نمودار
    plot(N, y, 'b', label='Selection Sort', linewidth=5)

    # برای هر ورودی در numbers محاسبه زمان اجرای Bubble Sort
    y = [get_avg_time(bubble_sort, a) for a in numbers]
    # با رنگ سبز و ضخامت خط 5 رسم نمودار
    plot(N, y, 'g', label='Bubble Sort', linewidth=5)

    در موقعیت بالا-چپ (Legend) نمایش راهنمایی #
```

In [7]:

```
def plot2(N, numbers):
    """
    این تابع زمان اجرای پنج الگوریتم مرتب‌سازی (Insertion, Selection, Bubble, Bucket, Radix) را بر روی آرایه‌های با اندازه‌های مختلف محاسبه و نمودار آن را رسم می‌کند.

    Args:
        N (list): لیستی از اندازه‌های مختلف ورودی (مثلًا [100, 200, 300]).
        numbers (list of lists): لیستی از آرایه‌های ورودی، که هر آرایه طولی برابر با دارد N اندازه‌ی مشخص شده در.

    """
    figure(figsize=(20, 10)) # ایجاد یک شکل (figure)
    xlabel("Size of array (n)") # برچسب محور X
    ylabel("Time") # برچسب محور Y

    # برای هر ورودی در numbers محاسبه زمان اجرای Insertion Sort
    y = [get_avg_time(insertion_sort, a) for a in numbers]
    # با رنگ قرمز و ضخامت خط 5 رسم نمودار
    plot(N, y, 'r', label='Insertion Sort', linewidth=5)

    # برای هر ورودی در numbers محاسبه زمان اجرای Selection Sort
    y = [get_avg_time(selection_sort, a) for a in numbers]
    # با رنگ آبی و ضخامت خط 5 رسم نمودار
    plot(N, y, 'b', label='Selection Sort', linewidth=5)
```

```

# برای هر ورودی در 'numbers' محاسبه زمان اجرای Bubble Sort
y = [get_avg_time(bubble_sort, a) for a in numbers]
# با رنگ سبز و ضخامت خط 5 رسم نمودار
plot(N, y, 'g', label='Bubble Sort', linewidth=5)

# برای هر ورودی در 'numbers' محاسبه زمان اجرای Bucket Sort
y = [get_avg_time(bucket_sort, a) for a in numbers]
# با رنگ زرد و ضخامت خط 5 رسم نمودار
plot(N, y, 'y', label='Bucket Sort', linewidth=5)

# برای هر ورودی در 'numbers' محاسبه زمان اجرای Radix Sort
y = [get_avg_time(radix_sort, a) for a in numbers]
# با رنگ مشکی و ضخامت خط 5 رسم نمودار
plot(N, y, 'k', label='Radix Sort', linewidth=5)

```

در موقعیت بالا-چپ (Legend) نمایش راهنمایی (Legend) می‌باشد.

نوبت به مقداردهی  $N$  و  $numbers$  است. برای این کار ابتدا مقادیر مختلف  $N$  تعریف می‌کنیم:

In [8]: تعريف بزرگترین اندازه آرایه برای تست #  
max\_N = 500

که شامل اندازه‌های مختلف آرایه برای تست است  $N$  ساخت لیست.  
این لیست شامل:  
 عدد 1 (برای آرایه با 1 عنصر) -  
 عدد 5 (برای آرایه با 5 عنصر) -  
 با گام‌های 50 تا (متلاً 100, 150, 200, ... ) از  $max\_N$  (500) اعدادی از 100 تا کمتر از  
 $N = [1, 5] + list(range(100, max_N, 50))$

برای مشاهده اندازه‌های انتخاب شده  $N$  چاپ لیست #  
print(N)

[1, 5, 100, 150, 200, 250, 300, 350, 400, 450]

حال بر اساس اندازه‌های انتخاب شده، آرایه از اعداد تصادفی بین ۱ تا ۱۰۰۰۰۰ درست می‌کنیم.

In [9]: تعريف بزرگترین اندازه آرایه برای تست #  
max\_N = 500

که شامل اندازه‌های مختلف آرایه برای تست است  $N$  ساخت لیست.  
این لیست شامل:  
 عدد 1 (برای آرایه با 1 عنصر)  
 عدد 5 (برای آرایه با 5 عنصر)  
 با گام‌های 50 تا (متلاً 100, 150, 200, ... ) از  $max\_N$  (500) اعدادی از 100 تا کمتر از  
 $N = [1, 5] + list(range(100, max_N, 50))$

برای مشاهده اندازه‌های انتخاب شده  $N$  چاپ لیست #  
print(N)

برای تولید اعداد تصادفی وارد کردن تابع #  
from random import randrange

که شامل آرایه‌هایی با اندازه‌های مختلف 'numbers' است.  
دارد  $N$  هر آرایه داخلی طولی برابر با عنصر متناصر خود در لیست.  
اعداد تصادفی بین ۰ تا 999999 (1-100000) تولید می‌شوند.  
 $numbers = [[randrange(100000) for j in range(i)] for i in N]$

برای بررسی 'numbers' چاپ دومین آرایه (با اندیس 1) از لیست #  
خواهد بود (که در این مورد 5 است)  $N$  این آرایه مربوط به دومین عنصر در #  
print(numbers[1])

[1, 5, 100, 150, 200, 250, 300, 350, 400, 450]

[14239, 94301, 23291, 54999, 4851]

حال نوبت به رسم نمودارها می‌رسد:

In [13]: وارد کردن کتابخانه‌های لازم #

```

import timeit
from random import randrange
برای رسم نمودار matplotlib.pyplot وارد کردن توابع خاص از
from matplotlib.pyplot import figure, xlabel, ylabel, plot, legend, clf, bar, gcf
from IPython import display # برای کنترل نمایش در Jupyter Notebook

```

در دسترس باشند که توابع #  
from src.codes.mysorts import insertion\_sort, selection\_sort, bubble\_sort

# --- تعریف تابع ---
def get\_time(f, input):
 """
 محاسبه می‌کند (input) را بر روی یک ورودی (f) این تابع زمان اجرای یک تابع
 برگردانده می‌شود (microseconds) زمان بر حسب میکروثانیه.
 """

```

start = timeit.default_timer()
f(input)
stop = timeit.default_timer()
return (stop - start) * 1000 * 1000 # microseconds

```

# --- تعریف تابع ---
def get\_avg\_time(f, input):
 """
 محاسبه می‌کند input را روی ورودی f این تابع متوسط حداقل زمان اجرای تابع
 برای کاهش اثر نویزهای سیستم، تابع را چندین بار اجرا کرده و حداقل میانگین‌ها را برگرداند.
 """
 res = 1 << 30 # عدد بسیار بزرگ

```

s = 10          # تعداد مراحل برای تکرار کل فرآیند و انتخاب حداقل
r = 2           # تعداد دفعات اجرای تابع در هر مرحله برای گرفتن میانگین

for i in range(s):
    sum_times = 0
    for j in range(r):
        sum_times += get_time(f, input)
    res = min(res, sum_times / float(r))
return res

# --- تعریف تابع plot1 ---
def plot1(N, numbers):
    """
    این تابع زمان اجرای سه الگوریتم مرتبسازی (Insertion, Selection, Bubble) را برای ورودی‌های با اندازه‌های مختلف محاسبه و نمودار آن را رسم می‌کند.
    """

    figure(figsize=(20, 10)) # ایجاد یک شکل (figure)
    xlabel("Size of array (n)") # برجسب محور X (اندازه آرایه)
    ylabel("Time") # برجسب محور Y (زمان اجرا)

    # محاسبه زمان اجرای Insertion Sort
    y = [get_avg_time(insertion_sort, a) for a in numbers]
    plot(N, y, 'r', label='Insertion Sort', linewidth=5)

    # محاسبه زمان اجرای Selection Sort
    y = [get_avg_time(selection_sort, a) for a in numbers]
    plot(N, y, 'b', label='Selection Sort', linewidth=5)

    # محاسبه زمان اجرای Bubble Sort
    y = [get_avg_time(bubble_sort, a) for a in numbers]
    plot(N, y, 'g', label='Bubble Sort', linewidth=5)

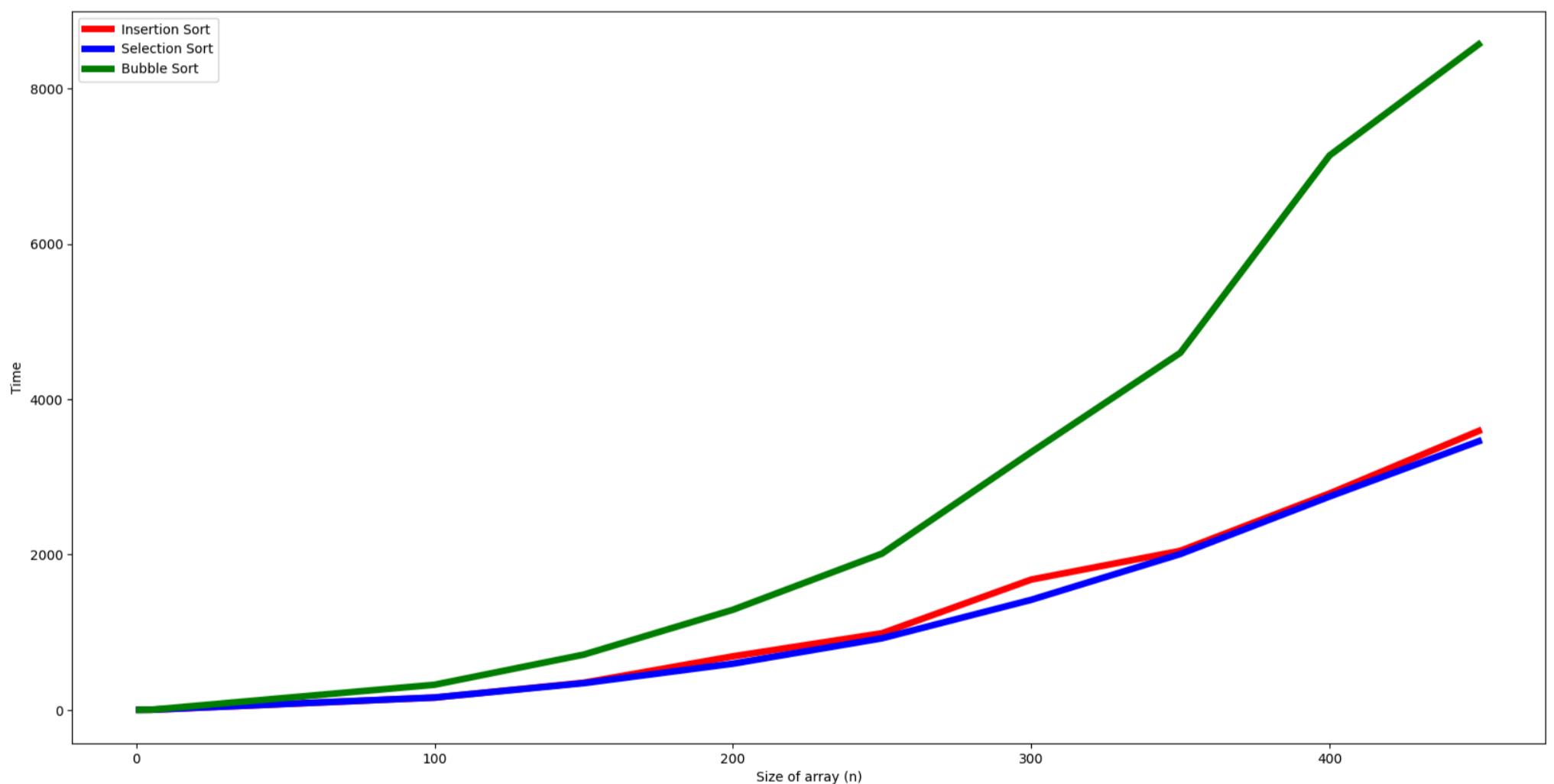
    legend(loc=2) # نمایش راهنمای Legend در موقعیت بالا-چپ

# --- مقداردهی N و numbers ---
max_N = 500
N = [1, 5] + list(range(100, max_N, 50))

numbers = [[randrange(100000) for j in range(i)] for i in N]

# --- برای رسم نمودار plot1 فراخوانی تابع ---
plot1(N, numbers)

```



```

In [15]: # وارد کردن کتابخانه‌های لازم
import timeit
from random import randrange
# برای رسم نمودار matplotlib.pyplot وارد کرد توابع خاص از
from matplotlib.pyplot import figure, xlabel, ylabel, plot, legend, clf, bar, gcf
from IPython import display # برای کنترل نمایش در Jupyter Notebook

# --- وارد کردن تابع مرتبسازی از فایل ---
# در دسترس باشدند که تابع insertion_sort, selection_sort, bubble_sort, bucket_sort, radix_sort می‌کند
from src.codes.mysorts import insertion_sort, selection_sort, bubble_sort, bucket_sort, radix_sort

# --- تعریف تابع get_time ---
def get_time(f, input):
    """
    محاسبه می‌کند (input) را بر روی یک ورودی (f) این تابع زمان اجرای یک تابع
    برگردانده می‌شود (microseconds) زمان بر حسب میکروثانیه.
    """

    start = timeit.default_timer()
    f(input)
    stop = timeit.default_timer()
    return (stop - start) * 1000 * 1000 # microseconds

# --- تعریف تابع get_avg_time ---

```

```

def get_avg_time(f, input):
    """
    محاسبه می‌کند f را روی ورودی input تابع متوسط حداقل زمان اجرای تابع.
    برای کاهش اثر نویزهای سیستم، تابع را چندین بار اجرا کرده و حداقل میانگین‌ها را برمی‌گرداند
    """

    res = 1 << 30 # یک عدد بسیار بزرگ
    s = 10 # تعداد مراحل برای تکرار کل فرآیند و انتخاب حداقل
    r = 2 # تعداد دفعات اجرای تابع در هر مرحله برای گرفتن میانگین

    for i in range(s):
        sum_times = 0
        for j in range(r):
            sum_times += get_time(f, input)
        res = min(res, sum_times / float(r))

    return res

# --- تعریف تابع ---
def plot2(N, numbers):
    """
    این تابع زمان اجرای پنج الگوریتم مرتبسازی (Insertion, Selection, Bubble, Bucket, Radix) را
    برای ورودی‌های با اندازه‌های مختلف محاسبه و نمودار آن را رسم می‌کند
    """

    figure(figsize=(20, 10)) # ایجاد یک شکل (figure)
    xlabel("Size of array (n)") # برجسب محور X (اندازه آرایه)
    ylabel("Time") # برجسب محور Y (زمان اجرا)

    # محاسبه زمان اجرای Insertion Sort
    y = [get_avg_time(insertion_sort, a) for a in numbers]
    plot(N, y, 'r', label='Insertion Sort', linewidth=5)

    # محاسبه زمان اجرای Selection Sort
    y = [get_avg_time(selection_sort, a) for a in numbers]
    plot(N, y, 'b', label='Selection Sort', linewidth=5)

    # محاسبه زمان اجرای Bubble Sort
    y = [get_avg_time(bubble_sort, a) for a in numbers]
    plot(N, y, 'g', label='Bubble Sort', linewidth=5)

    # محاسبه زمان اجرای Bucket Sort
    y = [get_avg_time(bucket_sort, a) for a in numbers]
    plot(N, y, 'y', label='Bucket Sort', linewidth=5)

    # محاسبه زمان اجرای Radix Sort
    y = [get_avg_time(radix_sort, a) for a in numbers]
    plot(N, y, 'k', label='Radix Sort', linewidth=5)

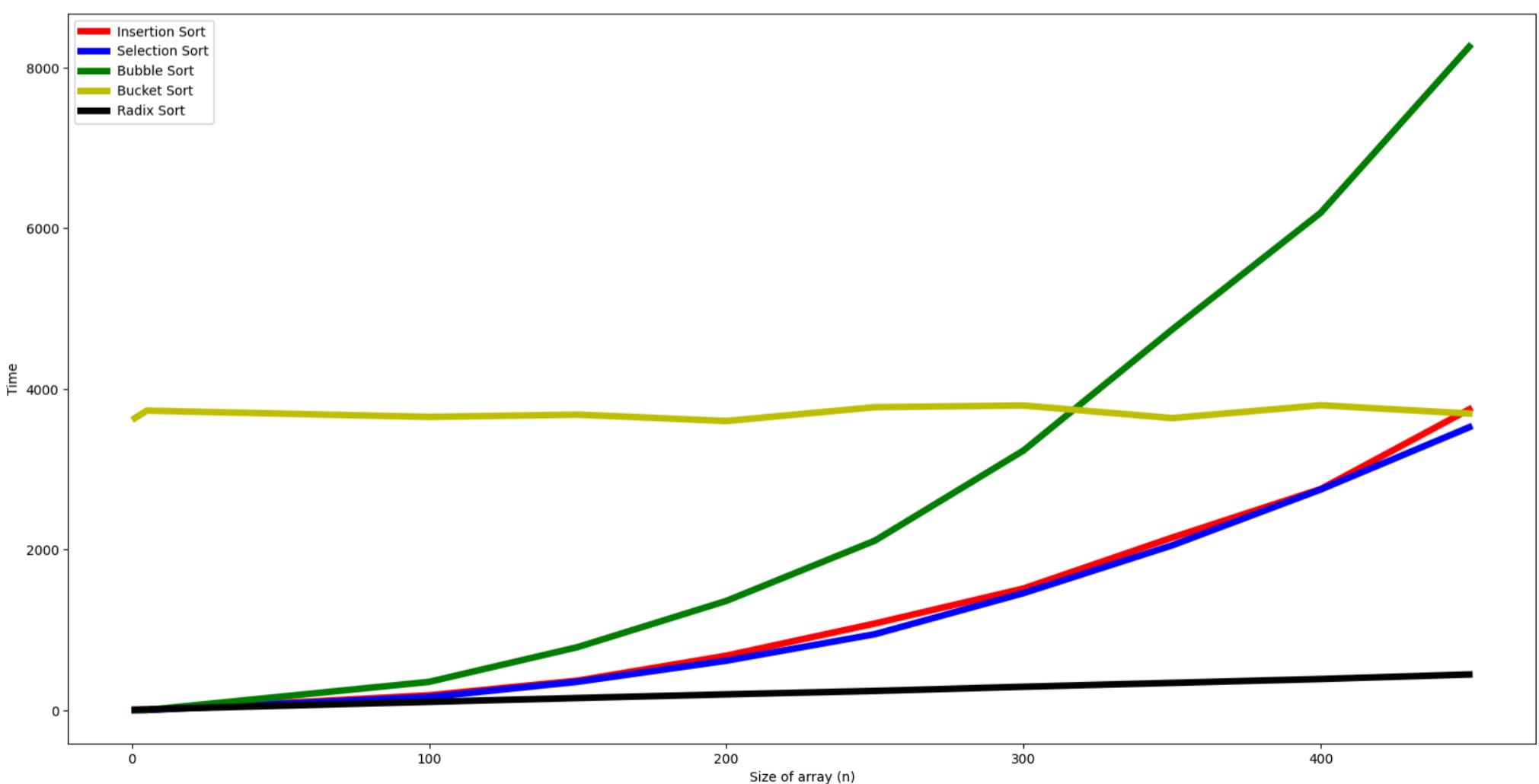
    legend(loc=2) # نمایش راهنمای (Legend) در موقعیت بالا-چپ

    # --- مقدارهای N و numbers ---
    max_N = 500
    N = [1, 5] + list(range(100, max_N, 50))

    numbers = [[randrange(100000) for j in range(i)] for i in N]

# --- برای رسم نمودار plot2 فراخوانی تابع ---
plot2(N, numbers)

```



چه توجیهی برای این رفتار توابع دارد؟

چرا bucket sort تقریباً ثابت عمل کرده است؟

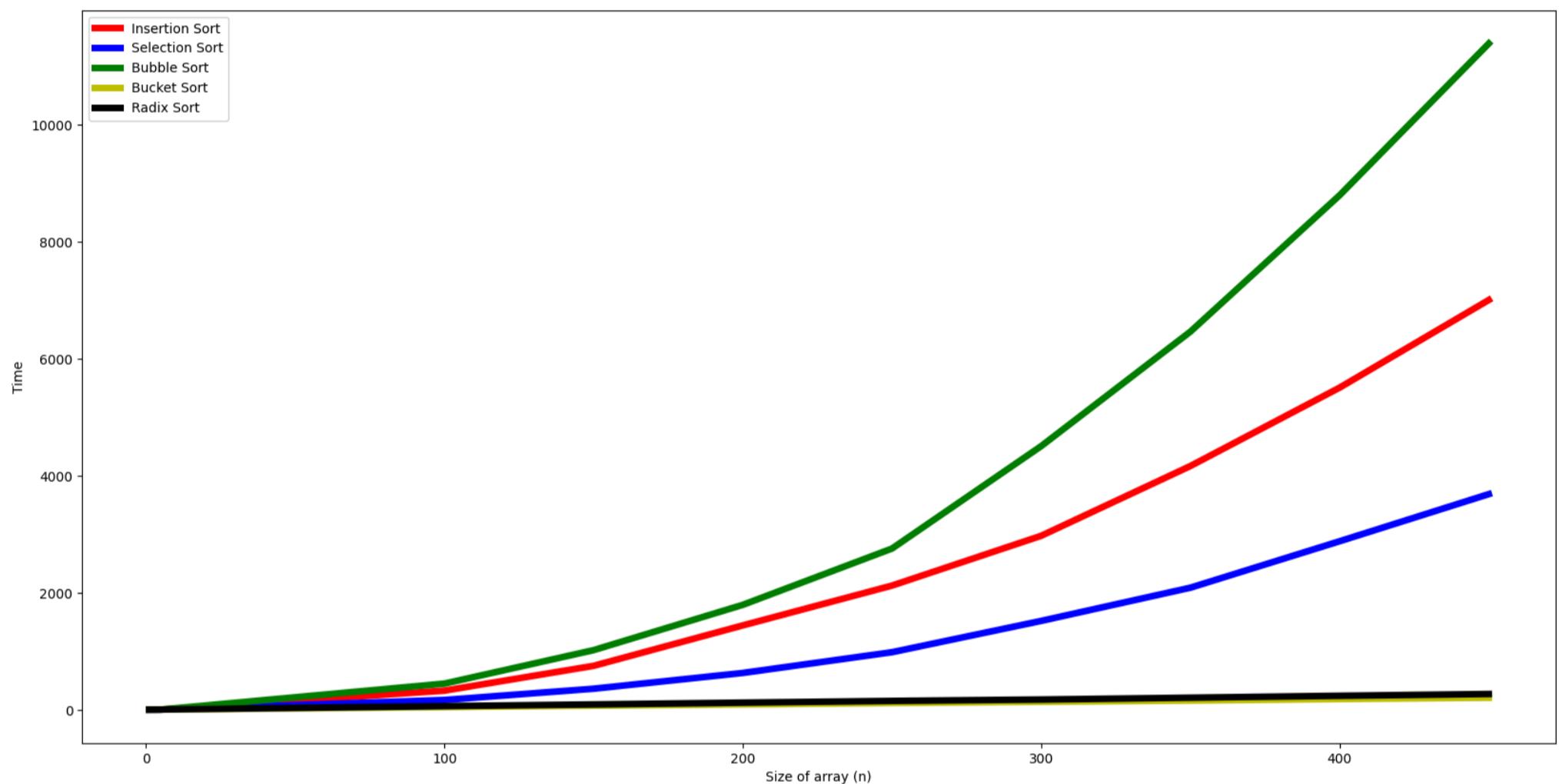
چرا radix sort به این شدت بهتر عمل کرده است؟

این نمودارها نشان دهنده عملکرد روش‌ها در حالت میانگین است. حال اعداد تولید شده، اعداد مرتب و معکوس شده هستند.

```
In [16]: # تولید لیست‌هایی از اعداد که به صورت معکوس (نژولی) مرتب شده‌اند.  
# این لیست‌ها برای تست عملکرد الگوریتم‌های مرتب‌سازی در بدترین حالت استفاده می‌شوند.  
# N = [1, 5] + List(range(100, max_N, 50))).  
numbers = [list(range(i, 0, -1)) for i in N]  
  
# برای بررسی 'numbers' چاپ دومین آرایه (با اندیس 1) از لیست.  
# خواهد بود و به صورت معکوس مرتب شده است (که در این مورد 5 است) N این آرایه مربوط به دومین عنصر در  
print(numbers[1])
```

[5, 4, 3, 2, 1]

```
In [17]: plot2(N, numbers)
```

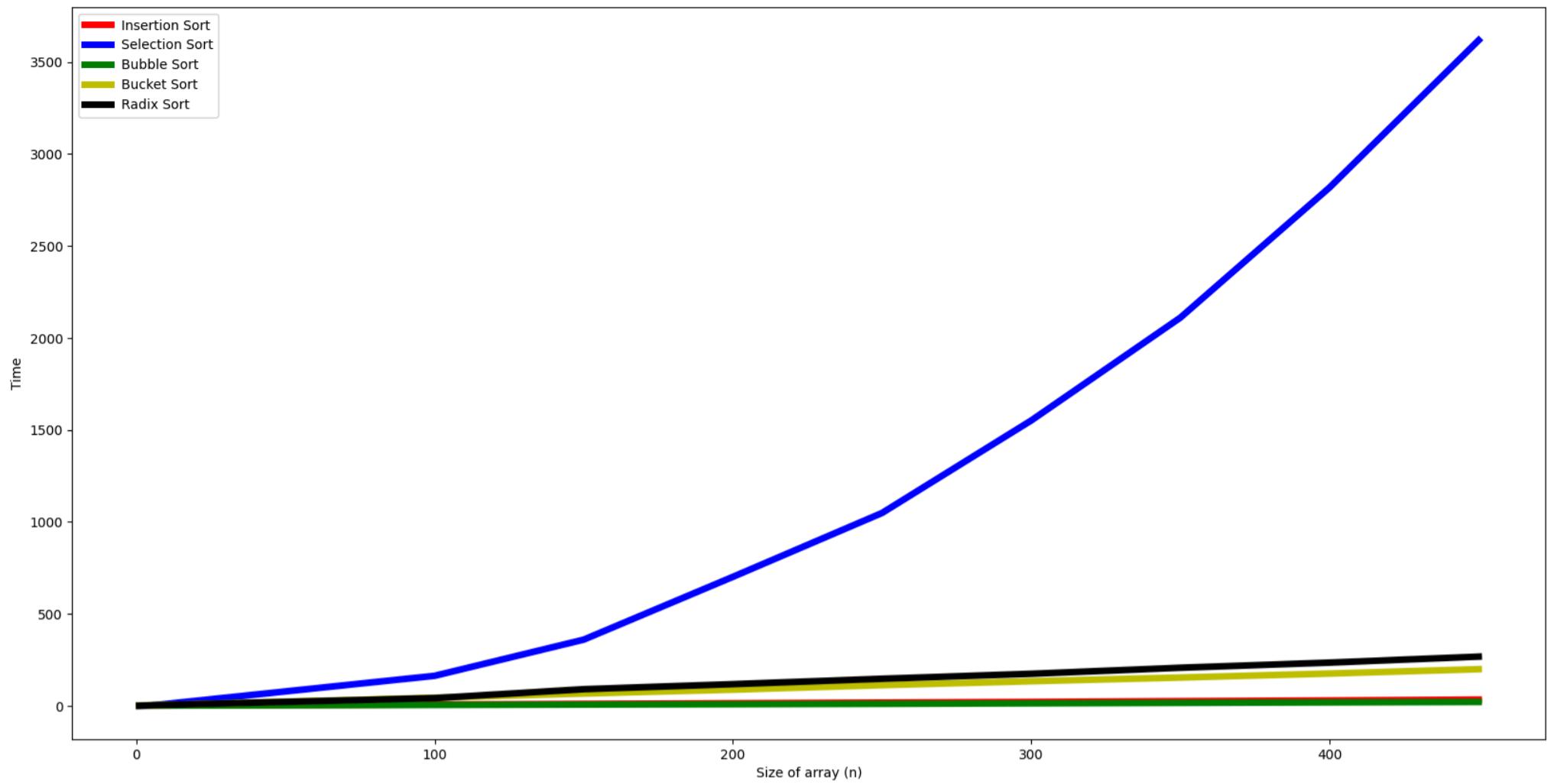


حال اعداد تولید شده، اعداد مرتب شده هستند.

```
In [18]: # مرتب شده‌اند (1-i از 0 تا) تولید لیست‌هایی از اعداد که به صورت صعودی.  
# این لیست‌ها برای تست عملکرد الگوریتم‌های مرتب‌سازی استفاده می‌شوند.  
# N = [1, 5] + List(range(100, max_N, 50))).  
numbers = [list(range(i)) for i in N]  
  
# برای بررسی 'numbers' چاپ دومین آرایه (با اندیس 1) از لیست.  
# خواهد بود و به صورت صعودی مرتب شده است (که در این مورد 5 است) N این آرایه مربوط به دومین عنصر در  
print(numbers[1])
```

[0, 1, 2, 3, 4]

```
In [19]: plot2(N, numbers)
```



چرا در این نوع ورودی bubble sort و insertion sort خیلی خوب عمل کرده اند؟

نوع ورودی در کدام روش ها تاثیر دارد؟ تاثیر ورودی را توضیح دهید.

جدول زیر مقایسه ای ابتدایی برای انتخاب الگوریتم مناسب برای مرتبسازی پیش می‌نهد:

نوع مرتبسازی	معیار
insertion sort	تعداد انواع داده‌ها کم است.
insertion sort	داده‌ها از پیش تقریباً مرتب شده هستند.
heap sort	عملکرد در بدترین حالت بسیار مورد توجه است.
quick sort	عملکرد خوب در حالت میانگین مورد نظر است.
insertion sort	می‌خواهیم کوتاه‌ترین کد ممکن را بنویسیم.
bucket sort	داده‌ها از یک توزیع یکنواخت می‌آیند.
merge sort	از لیست پیوندی استفاده کرده‌ایم.

## برای مطالعه بیشتر

جهت مشاهده پویانمایی نحوه کار داده‌ساختارها و الگوریتم‌های مختلف، از جمله انواع مرتبسازی‌ها که در این فصل به آن‌ها اشاره شد، می‌توانید به سایتها زیر مراجعه کنید:

\*\*Visualgo\*\*: یک ابزار تعاملی عالی برای بصری‌سازی داده‌ساختارها و الگوریتم‌های مختلف، از جمله مرتبسازی:

<https://visualgo.net>

Sorting Algorithm Visualizer\*\*: برای مشاهده پویانمایی اختصاصی الگوریتم‌های مرتبسازی، می‌توانید به این ابزار کاربردی نیز مراجعه کنید :

[https://ds-fall2025.github.io/sorting\\_algo](https://ds-fall2025.github.io/sorting_algo)

استفاده از این ابزارهای بصری، به درک عمیق‌تر نحوه عملکرد الگوریتم‌ها و مقایسه بصری آن‌ها کمک شایانی می‌کند.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل سوم، بخش اول: موضوع پیچیدگی زمانی الگوریتم‌ها

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

- مقدمه
- یادآوری تعدادی توابع
- مقایسه الگوریتم‌ها
- مقایسه توابع رشد
- بهترین حالت، بدترین حالت و حالت میانگین
- پیچیدگی الگوریتم‌ها

## مقدمه

آن است که کدام یک از این الگوریتم‌ها بهتر است؟ در این بخش از درس به بررسی چگونگی تعیین پیچیدگی الگوریتم‌های گوناگون می‌پردازیم.

## یادآوری تعدادی توابع

$$\begin{aligned} \lg n &= \log_2 n \\ \ln n &= \log_e n \\ \lg^k n &= (\lg n)^k \\ \lg \lg n &= \lg(\lg n) \\ a &= b^{\log_b a} \\ \log_c(ab) &= \log_c a + \log_c b \\ \log_b a^n &= n \log_b a \\ \log_b a &= \frac{\log_c a}{\log_c b} \\ \log_b(1/a) &= -\log_b a \\ \log_b a &= \frac{1}{\log_a b} \\ a^{\log_b c} &= c^{\log_b a} \end{aligned}$$

## مقایسه الگوریتم‌ها

تحلیل الگوریتم‌ها با هدف های زیر انجام می‌شود:

- بررسی و پیش‌بینی زمان اجرا و میزان حافظه مصرفی یک الگوریتم قبل از پیاده‌سازی
- مقایسه الگوریتم‌های مختلف برای حل یک مسئله از نظر میزان کارایی

به طور کلی این نوع مقایسه صحیح نیست و الگوریتم بهتر با توجه به مسئله مشخص می‌شود و در تعیین آن مسائل متفاوتی تاثیرگذار است. همان‌گونه که انتظار می‌رود یکی از این موارد سرعت اجرای الگوریتم است.

با توجه به آنکه زمان اجرای الگوریتم به پردازنده‌ای که در حال اجرای آن الگوریتم است، وابسته است پس برای سنجش کارایی یک الگوریتم معیار خوبی به نظر نمی‌رسد. به نظر شما چه معیاری برای سنجش زمان

اجرای الگوریتم مناسب است؟

## تعداد عملیات‌ها، معیار سنجش سرعت الگوریتم‌ها

با وجود آنکه زمان اجرای الگوریتم در پردازنده‌های مختلف، متفاوت است اما تعداد عملیات‌هایی که هر پردازنده انجام می‌دهد با توجه به ساختار کد و الگوریتم یکتا بوده و قابل محاسبه است و از آن جایی که تعداد عملیات‌ها با زمان اجرای الگوریتم رابطه‌ی مستقیم دارد، استفاده از آن به عنوان معیاری برای مقایسه سرعت الگوریتم‌ها منطقی به نظر می‌رسد. البته تعداد عملیات‌های یک الگوریتم ثابت نبوده و می‌تواند به موارد مختلفی مثل تعداد عناصر ورودی و یا بیشینه‌ی اعداد ورودی و ... وابسته باشد.

زمان اجرای بعضی از عملیات‌های ساده را در کد زیر می‌توان مشاهده کرد. در اینجا دستور `timeit` میزان زمان اجرای هر دستور را با در نظر گرفتن دو پارامتر `n` و `timeit` این دستور برای محاسبه دقیق‌تر زمان اجرای یک عبارت، در ۲ مرحله زمان‌گیری، و در هر مرحله `n` بار دستور را انجام می‌دهد. نهایتاً بین ۲ مرحله، حداقل را برمی‌گرداند.

```
In [1]: import math # وارد کردن مازول math
print("Integer Operations")
چاپ عنوان برای عملیات اعداد صحیح ()
a, b = 1, 2 # مقادیر 1 و 2
تعريف دو متغیر صحیح با مقادیر 1 و 2
# زمان‌سنجی عملیات جمع اعداد صحیح
# دستور را 3 بار تکرار می‌کند و بهترین زمان را انتخاب می‌کند: -r 3
# در هر تکرار، عملیات را 100 بار اجرا می‌کند: -n 100
%timeit -r 3 -n 100 a + b
# زمان‌سنجی عملیات ضرب اعداد صحیح
%timeit -r 3 -n 100 a * b
# زمان‌سنجی عملیات تقسیم صحیح (بدون باقی‌مانده) اعداد صحیح
%timeit -r 3 -n 100 a // b
```

Integer Operations  
28.7 ns ± 4.5 ns per loop (mean ± std. dev. of 3 runs, 100 loops each)  
30.3 ns ± 6.13 ns per loop (mean ± std. dev. of 3 runs, 100 loops each)  
33.7 ns ± 4.5 ns per loop (mean ± std. dev. of 3 runs, 100 loops each)

```
In [2]: print("\nFloat Operations")
چاپ عنوان برای عملیات اعداد اعشاری ()
a, b = 1.1, 2.2 # 2.2
تعريف دو متغیر اعشاری با مقادیر 1.1 و 2.2
%timeit -r 3 -n 100 a + b
# زمان‌سنجی عملیات جمع اعداد اعشاری
%timeit -r 3 -n 100 a * b
# ازمان‌سنجی عملیات ضرب اعداد اعشاری
%timeit -r 3 -n 100 a / b
# زمان‌سنجی عملیات تقسیم اعشاری
```

Float Operations  
32 ns ± 6.38 ns per loop (mean ± std. dev. of 3 runs, 100 loops each)  
78 ns ± 60.5 ns per loop (mean ± std. dev. of 3 runs, 100 loops each)  
32.7 ns ± 5.91 ns per loop (mean ± std. dev. of 3 runs, 100 loops each)

```
In [3]: print("\nOthers")
چاپ عنوان برای سایر عملیات‌ها ()
%timeit -r 3 -n 100 a < b
# زمان‌سنجی عملیات مقایسه (کوچکتر)
%timeit -r 3 -n 100 math.sin(1)
# زمان‌سنجی تابع سینوس از مازول math
```

Others  
32 ns ± 4.97 ns per loop (mean ± std. dev. of 3 runs, 100 loops each)  
104 ns ± 52.3 ns per loop (mean ± std. dev. of 3 runs, 100 loops each)

## زمان‌های بعضی توابع دیگر

```
In [4]: print("\nList Operations")
چاپ عنوان برای عملیات لیست‌ها ()
# زمان‌سنجی ایجاد یک لیست بزرگ با 100000 عنصر صفر
# دستور را 3 بار تکرار می‌کند و بهترین زمان را انتخاب می‌کند: -r 3
# در هر تکرار، عملیات را 10 بار اجرا می‌کند: -n 10
%timeit -r 3 -n 10 a = [0 for i in range(100000)]
# زمان‌سنجی ارجاع دادن یک لیست به متغیر دیگر (ارجاع به حافظه، نه کپی عمیق)
%timeit -r 3 -n 10 b = a
```

## مقایسه‌ی توابع رشد

In [6]: # نمایش نمودارها مستقیماً در خروجی Jupyter

```
%matplotlib inline
from matplotlib.pyplot import * # وارد کردن تمام توابع رسم نمودار از Matplotlib
```

تنظیمات پیش‌فرض برای نمودارها (اندازه فونت، خانواده فونت، ضخامت خط) #

دو تابع  $n \log n$  و  $n^2$  را مقایسه کنید.

In [7]: import math # مثل) برای استفاده از توابع ریاضی math وارد کردن مازول Log

# قبل اجرا شده و توابع رسم نمودار در دسترس هستند 'from matplotlib.pyplot import \*' فرض می‌شود.

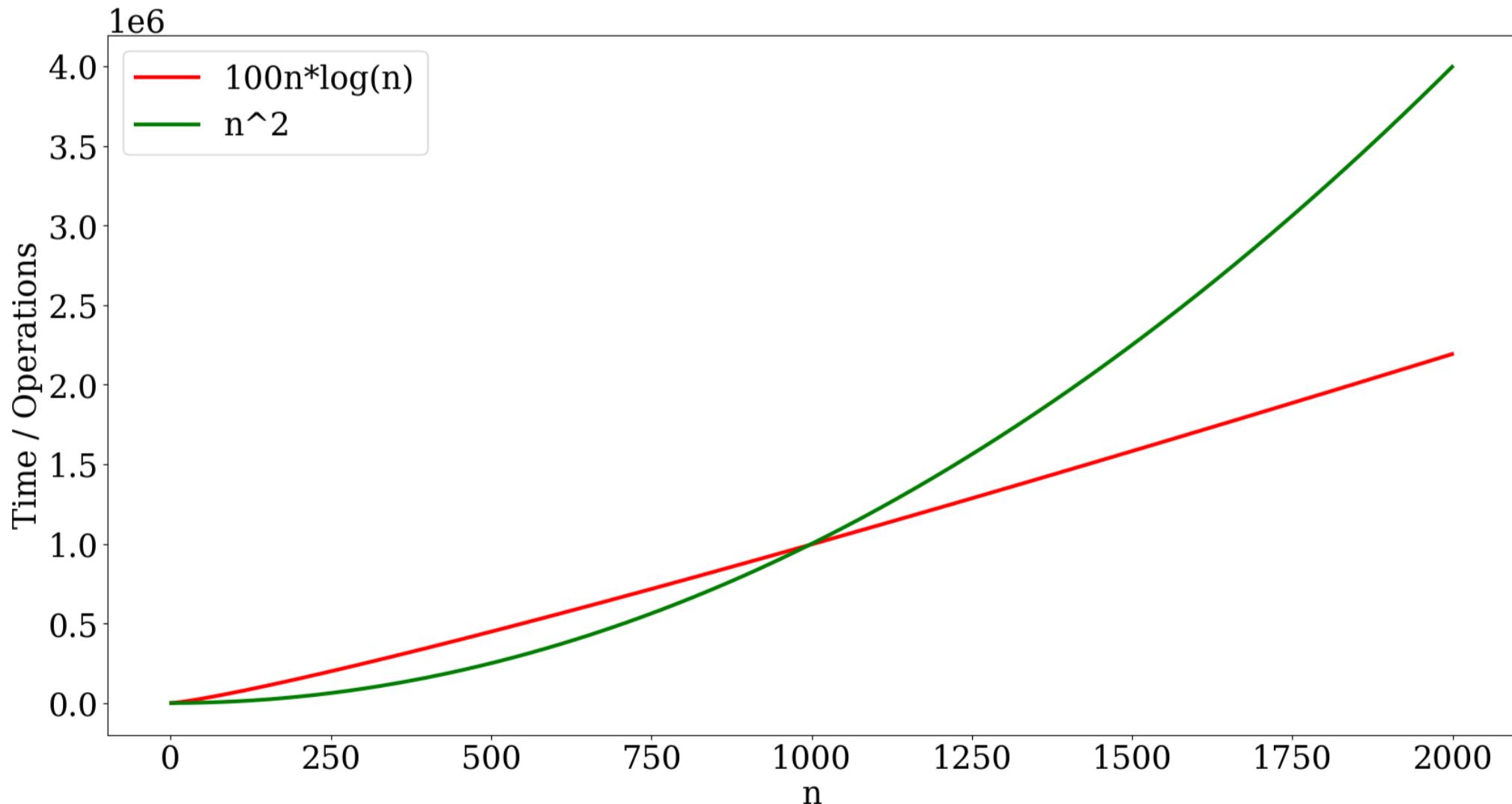
```
figure(figsize=(20, 10)) # ایجاد یک شکل
xlabel("n") # برجسب محور X
ylabel("Time / Operations") # برجسب محور Y (زمان یا تعداد عملیات)
```

```
برای رسم نمودار n حداقل اندازه # برای رسم نمودار n حداکثر اندازه
x = range(1, max_n) # از 1 تا n ایجاد لیستی از مقادیر
```

```
# برای محاسبه مقادیر y = 100 * n * math.log(n, 2) است
y = [100 * n * math.log(n, 2) for n in x] # 100n * log(n) رسم نمودار
plot(x, y, 'r', label='100n*log(n)')
```

```
# برای محاسبه مقادیر y = n^2
y = [n ** 2 for n in x] # برای سیز و برجسب مربوطه n^2 رسم نمودار
plot(x, y, 'g', label='n^2')
```

```
در موقعیت بالا-چپ نمودار (Legend) نمایش راهنمای نمایش نمودار #
```



در مثال بالا تا  $n = 1000$  تابع  $n^2$  کمتر یا مساوی تابع دیگر است، اما برای مقادیر بیشتر  $n$  از  $100n \times \lg(n)$  تابع  $n^2$  بیشتر می‌شود. هرچقدر  $n$  بیشتر شود اختلاف این دو تابع زیادتر هم می‌شود.

## مثال

$$\frac{1}{6}N^3 + 20N + 16$$

$$\frac{1}{6}N^3 + 100N^{4/3} + 56$$

$$\frac{1}{6}N^3 - \frac{1}{2}N^2 + \frac{1}{3}N$$

$$\frac{1}{6}N^3$$

برای این کار ابتدا یک تابع `plot1(max_n)` تعریف می کنیم که نمودار سه تابع فوق را در بازه ۱ تا `max_n` رسم می کند.

In [8]: `def plot1(max_n):`

```
"""
رسم می کند max_n این تابع نمودار چهار تابع چندجمله‌ای را در بازه ۱ تا
هدف مقایسه رفتار توابع با جملات درجه بالاتر است
"""

figure(figsize=(20, 10)) # ایجاد یک شکل # figure

xlabel("N") # برچسب محور X
ylabel("Function Value") # برچسب محور Y
x = range(1, max_n + 1) # از ۱ تا N مقادیر از ۱ تا max_n

# ۱/۶ تعریف و رسم تابع اول: N^3 + 20N + 16
def f1(n): return 1.0/6*n**3 + 20*n + 16
y1 = [f1(n) for n in x]
plot(x, y1, 'r', label='1/6n^3 + 20n + 16')

# ۱/۶ تعریف و رسم تابع دوم: N^3 + 100N^(4/3) + 56
def f2(n): return 1.0/6*n**3 + 100*n**(4.0/3) + 56
y2 = [f2(n) for n in x]
plot(x, y2, 'g', label='1/6n^3 + 100n^(4/3) + 56')

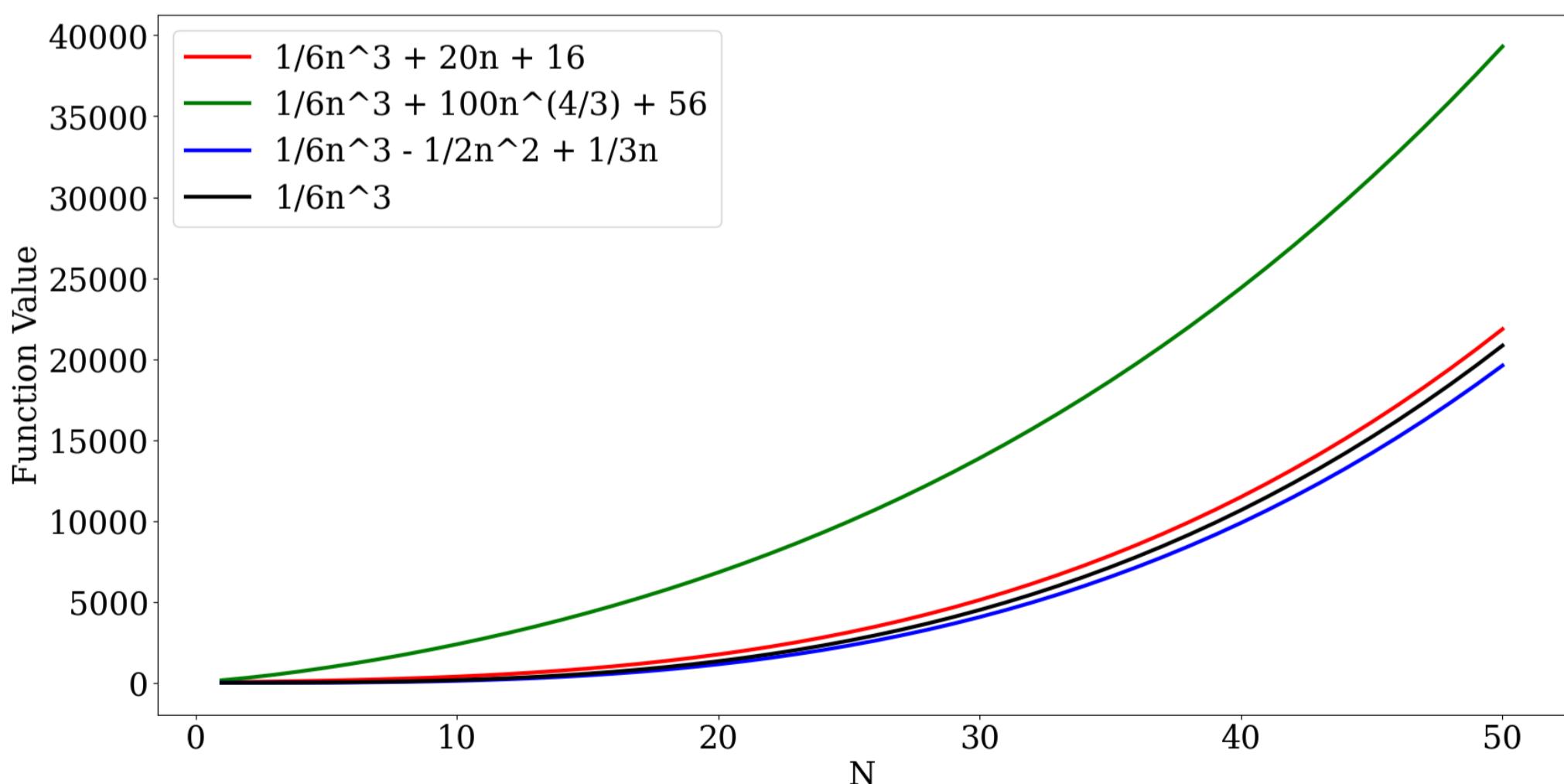
# ۱/۶ تعریف و رسم تابع سوم: N^3 - 1/2N^2 + 1/3N
def f3(n): return 1.0/6*n**3 - 1.0/2*n**2 + 1.0/3*n
y3 = [f3(n) for n in x]
plot(x, y3, 'b', label='1/6n^3 - 1/2n^2 + 1/3n')

# ۱/۶ رسم تابع چهارم: N^3
y4 = [1.0/6*n**3 for n in x]
plot(x, y4, 'k', label='1/6n^3')

legend(loc=2) # نمایش راهنمایی
show() # نمایش نمودار
```

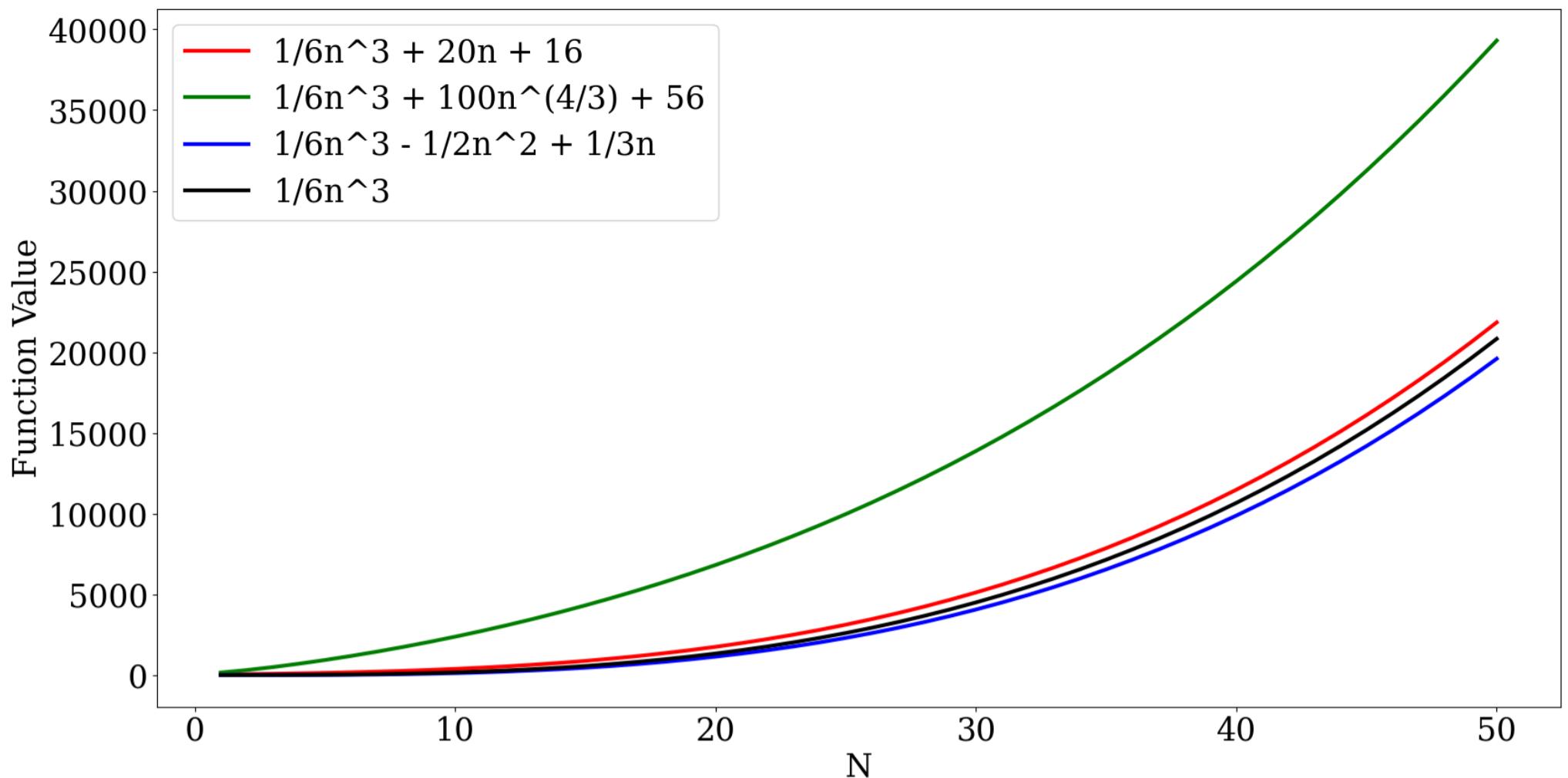
حال توابع فوق را برای مقادیر ۱ تا ۵۰ رسم می کنیم. همان‌طور که ملاحظه می کنید نمودار سبزرنگ به خاطر جمله  $\frac{1}{6}N^3$  از بقیه بالاتر است.

In [9]: `plot1(50)`



اما اگر همین نمودارها را تا ۵۰۰ رسم کنیم تفاوت بین سه نمودار در مقایسه با رشد آنها خیلی جزئی می شود. در واقع به خاطر وجود یک جمله  $\frac{1}{6}n^3$  مشترک در هر چهار تابع، اثر سایر جملات با توانهای کمتر  $n$  در مقادیر بالای  $n$  بسیار کم اثر می شود.

In [10]: `plot1(500)`



می توان نتیجه گرفت

$$1/6N^3 + 20N + 16 \sim 1/6N^3$$

$$1/6N^3 + 100N^{4/3} + 56 \sim 1/6N^3$$

$$1/6N^3 - 1/2N^2 + 1/3N \sim 1/6N^3$$

لذا می توان از جمله ها با درجه کوچکتر از بزرگترین درجه چشم پوشی کرد.

- در  $N$  های کوچک معمولاً مقدار توابع کم است و برای ما مهم نیست.
- در  $N$  های بزرگ تاثیر خیلی کمی روی تابع می گذارند.

به عبارت دیگر در این مثالها هر دو تابع  $f$  و  $g$  روند رشد یکسانی دارند چرا که:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1 \rightarrow f(n) \sim g(n)$$

## مثال

می خواهیم تعداد عملیات های کد ساده زیر را محاسبه کنیم.

```
In [11]: def algorithm1(n):
    """
    پیمایش می کند و هر عدد را چاپ می کند n-1 این تابع یک حلقه ساده است که از 0 تا
    هدف از این مثال، محاسبه تعداد عملیات های پایه در یک حلقه ساده است
    """
    عملیات: مقداردهی اولیه (1 عملیات)
    # بار اجرا می شود n+1 عملیات: مقایسه
    while i < n: #
        print(i) # عملیات: چاپ
        # بار اجرا می شود n عملیات: جمع و انتساب
        i += 1 # عملیات: جمع و انتساب
```

$c_i$ : تعداد عملیات هایی که CPU برای انجام خط  $i$  ام نیاز دارد.  
با توجه به تعریف بالا، تعداد عملیات های تابع algorithm1 بر حسب  $n$  چقدر است؟

$m(n) = c_1 + n \times (c_2 + c_3 + c_4) \Leftarrow n$ : تعداد دقیق عملیات‌ها بر حسب  $n$   
 $f(n)$ : تعداد عملیات‌ها اگر بهجای تمام  $c_i$ ‌ها کمترین  $c_i$  را در نظر بگیریم.  
 $g(n)$ : اگر به جای تمام  $c_i$ ‌ها بیشترین  $c_i$  را در نظر بگیریم.  
 $h(n)$ : اگر تمام  $c_i$ ‌ها برابر با یک باشد.  
 عبارات زیر را محاسبه کنید:

$$\lim_{n \rightarrow \infty} \frac{m(n)}{f(n)}$$

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

$$\lim_{n \rightarrow \infty} \frac{g(n)}{h(n)}$$

## حالتهای میانگین، بهترین و بدترین

تعداد دفعات اجرای یک خط می‌تواند به بسیاری از موارد و به طور کلی به تمامی عناصر ورودی مرتبط باشد. اما برای مقایسه زمان اجرای الگوریتم‌ها تعداد دقیق دفعات اجرا مورد نظر نیست و به همین علت در بررسی زمان اجرا معمولاً یک یا چند معیار مهم را مانند تعداد عناصر ورودی به عنوان متغیر در نظر گرفته و سپس تعداد عملیات الگوریتم را نسبت به این متغیرها در یکی از سه حالت زیر مورد بررسی قرار می‌دهند:

1. بهترین حالت(Best Case): اگر اندازه‌ی ورودی الگوریتم را  $n$  در نظر بگیریم، بهترین حالت وقتی است که یک ورودی به اندازه‌ی  $n$  از همه‌ی ورودی‌های همان‌اندازه‌ی خود زمان اجرای کمتری داشته باشد.
2. بدترین حالت(Worst Case): بر عکس حالت قبل، نوعی از ورودی به اندازه‌ی  $n$  که از همه‌ی ورودی‌های همان‌اندازه‌ی خود زمان اجرای بیشتری نیاز داشته باشد.
3. حالت میانگین(Average Case): زمان اجرای الگوریتم وقتی که یکی از همه‌ی ورودی‌های ممکن با اندازه‌ی  $n$  به طور شансی (با احتمال یکسان) به برنامه داده شود. یا به عبارت دیگر، متوسط زمان اجرای برنامه برای همه‌ی ورودی‌های به اندازه‌ی  $n$ .

در واقع داریم:

1. بهترین حالت(Best Case): حد پایینی از زمان اجرا
2. بدترین حالت(Worst Case): حد بالایی از زمان اجرا
3. حالت میانگین(Average Case): زمان اجرا به ازای ورودی تصادفی (Random)

در این جلسه تنها دو روش بهترین حالت و بدترین حالت را بررسی می‌کنیم.

اگر رفتار یک الگوریتم یا پیچیدگی آن در بهترین و بدترین حالات مختلف باشد، ممکن است پیچیدگی الگوریتم در حالت میانگین بهتر از پیچیدگی آن در بدترین حالت باشد.

برای مثال، زمان اجرای الگوریتم مرتب سازی سریع در بدترین حالت متناسب با  $n^2$  و در حالت میانگین متناسب با  $n \lg n$  است که اختلاف این دو تابع برای مقادیر بزرگ  $n$  چشمگیر است.

## مثال

کد زیر یک پیاده‌سازی از الگوریتم مرتب‌سازی حبابی است که در جلسه‌ی قبل مطرح شد. می‌خواهیم تعداد عملیات‌های این الگوریتم در بهترین و بدترین حالت را محاسبه کنیم.

```
In [12]: A = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] # آرایه ورودی برای مرتب‌سازی
print(f"آرایه اصلی: {A}")
```

```
def bubble_sort(A):
    """
    مرتب می‌کند (Bubble Sort) را با استفاده از الگوریتم مرتب‌سازی حبابی A. این تابع آرایه همچین تعداد مقایسه‌های انجام شده (عملیات اصلی) را نیز شمارش می‌کند.

    Args:
        A (list): آرایه‌ای از اعداد برای مرتب‌سازی.

    Returns:
        tuple: یک تاپل شامل آرایه مرتب شده و تعداد عملیات‌های مقایسه.
    """
    n = len(A) # طول آرایه
    ops = 0 # شمارنده عملیات‌های مقایسه
    i = 0 # اندیس حلقه بیرونی
    flag = True # پرچم برای تشخیص اینکه آیا در یک دور جابجایی انجام شده است یا خیر

    # حلقه بیرونی: تا زمانی که همه عناصر بررسی شوند با هیچ جابجایی‌ای انجام نشود
    while i < n and flag:
        flag = False # پرچم را برای شروع دور حجدید بازنگشانی می‌کند
        # حلقه درونی: از انتهای بخش نامرتب به سمت ابتدای آن پیمایش می‌کند
        for j in range(n - 1, i, -1):
            if A[j] < A[j - 1]:
                flag = True # اگر جابجایی انجام شود، پرچم را افراش می‌دهد
                ops += 1 # شمارنده عملیات را افزایش می‌دهد
                A[j], A[j - 1] = A[j - 1], A[j] # جابجایی عناصر
            else:
                A[j] = A[j] # افزایش اندیس حلقه بیرونی
        i += 1 # برگرداندن آرایه مرتب شده و تعداد عملیات‌ها
    return A, ops
```

```
# دریافت آرایه مرتب شده و تعداد عملیات‌ها فراخوانی تابع
B, o = bubble_sort(A)
print(f"آرایه مرتب شده: {B}")
print(f"تعداد عملیات‌های مقایسه: {o}")
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] # آرایه اصلی
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] # آرایه مرتب شده
45 # تعداد عملیات‌های مقایسه
```

خط	دفعات اجرا در بدترین حالت	دفعات اجرا در بهترین حالت
1		1
$n$		2
$(n - 1) \times n$		3
$(n - 1) \times (n - 1)$		4
$(n - 1) \times (n - 1)$	0	5
1		6

مرتب‌سازی حبابی

\* توجه کنید که شرط حلقه یک بار بیشتر از دفعاتی که وارد حلقه می‌شود بروی سطر ؟ ام از کد بالا اجرا می‌شود.

فرض کنید  $a_i$  مجموع تعداد عملیات CPU باشد که در طول اجرای الگوریتم بر روی سطر ؟ ام از کد بالا اجرا می‌شود. می‌خواهیم بینیم تعداد دقیق عملیات‌ها بر حسب  $n$  در بدترین و بهترین حالت چقدر است؟

در بهترین حالت داریم:

$$a_1 + n \times a_2 + (n - 1) \times n \times a_3 + (n - 1) \times (n - 1) \times a_4 + 0 \times a_5 + a_6$$

در بدترین حالت داریم:

$$a_1 + n \times a_2 + (n - 1) \times n \times a_3 + (n - 1) \times (n - 1) \times a_4 + (n - 1) \times (n - 1) \times a_5 + a_6$$

توابعی شبیه توابعی که برای الگوریتم قبل در نظر گرفته ایم تعریف کنید. حد نسبت این توابع به یکدیگر نیز مانند توابع بالا یک عدد ثابت خواهد بود.

می خواهیم تعداد عملیات های الگوریتم مرتب سازی حبابی را با الگوریتم ابتدایی که اعداد ۰ تا  $n - 1$  را چاپ می کرد مقایسه کنیم. برای سادگی فرض کنید که  $C = \max(a_i, b_i, c_i)$ . در این صورت می توان کران بالای زیر را در بدترین حالت برای الگوریتم مرتب سازی حبابی در نظر گرفت:

$$T_1(n) = C \times (3n^2 - 5n + 4)$$

و همچنین کران بالای زیر برای الگوریتم ابتدایی به دست می آید:

$$T_2(n) = C \times (3n + 1)$$

حالا می خواهیم زمان اجرای مرتب سازی حبابی را محاسبه کنیم.

```
In [13]: from random import randrange # برای تولید اعداد تصادفی وارد کردن تابع randrange
```

```
In [17]: # که قبلاً در جزوه داشتیم bubble_sort تعريف تابع.  
# این تابع برای get_time_for_bubble_sort لازم است.
```

```
def bubble_sort(A):  
    n = len(A)  
    flag = True  
    i = 0  
    while i < n and flag:  
        flag = False  
        for j in range(n - 1, i, -1):  
            if A[j] < A[j - 1]:  
                flag = True  
                A[j], A[j - 1] = A[j - 1], A[j]  
        i += 1  
    return A  
  
def get_time(n):  
    """  
    را ایجاد کرده و زمان اجرای n این تابع یک لیست تصادفی به طول  
    روی آن لیست اندازه گیری می کند.  
    """  
    # 999,999  
    nums = [randrange(1000 * 1000) for _ in range(n)]  
  
    # برای اندازه گیری دقیق زمان استفاده از timeit.Timer  
    # setup: کدی که قبلاً از هر بار اجرای  
    # stmt: دستوری که می خواهیم زمان آن را بگیریم.  
    timer = timeit.Timer(  
        stmt='bubble_sort(my_nums)',  
        setup=f'from __main__ import bubble_sort; my_nums = {nums}'  
    )  
  
    # است که زمان کل را نشان می دهد float یک timeit.timeit() خروجی.  
    execution_time = timer.timeit(number=1)  
  
    return execution_time * 1000 * 1000 # تبدیل به میکروثانیه
```

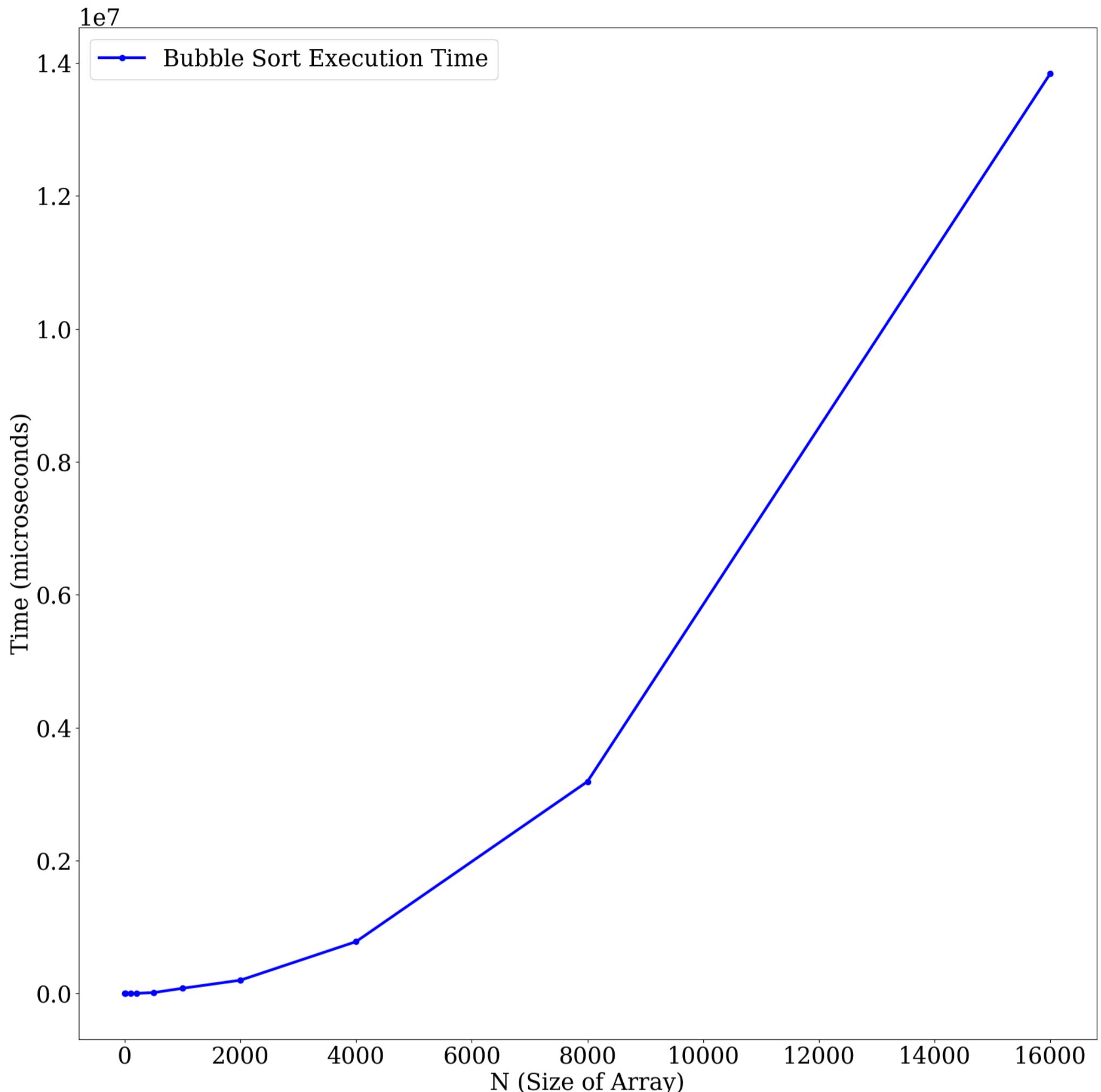
```
In [18]: N = [1, 10, 100, 200, 500, 1000, 2000, 4000, 8000, 16000]  
time = [0] * len(N)
```

```
for i in range(len(N)):  
    print("Estimating running time for N =", N[i])  
    time[i] = get_time(N[i])  
  
print("\nTimes (microseconds):", time)
```

```
Estimating running time for N = 1  
Estimating running time for N = 10  
Estimating running time for N = 100  
Estimating running time for N = 200  
Estimating running time for N = 500  
Estimating running time for N = 1000  
Estimating running time for N = 2000  
Estimating running time for N = 4000  
Estimating running time for N = 8000  
Estimating running time for N = 16000
```

```
Times (microseconds): [2.7999994927085936, 11.000000085914508, 356.099997588899, 1453.1999986502342, 11926.899998798035, 77041.49999699439, 198579.600000812  
2, 779300.8999979065, 3188424.700001633, 13841639.199999917]
```

```
In [19]: figure(figsize=(20, 20)) # جدید با اندازه مشخص (figure) ایجاد یک شکل
xlabel("N (Size of Array)") # X برجسب محور
ylabel("Time (microseconds)") # Y برجسب محور زمان
plot(N, time, 'b-o', label='Bubble Sort Execution Time') # رسم نمودار زمان اجرای الگوریتم
legend(loc='upper left') # نمایش راهنمای نمودار
show() # نمایش نمودار
```



## پیچیدگی الگوریتم‌ها

برای درک بهتر مفهوم پیچیدگی، فرض کنید برای حل یک مسئله با  $n$  عدد ورودی 7 الگوریتم مختلف وجود دارد. همچنین زمان اجرای این الگوریتم‌ها را بر حسب  $n$  داریم. در جدول زیر زمان اجرای الگوریتم‌ها به ازای مقادیر مختلف  $n$  نشان داده شده است:

## پیچیدگی الگوریتم‌ها

برای درک بهتر مفهوم پیچیدگی، فرض کنید برای حل یک مسئله با  $n$  عدد ورودی ۷ الگوریتم مختلف وجود دارد. همچنین زمان اجرای این الگوریتم ها را بر حسب  $n$  داریم. در جدول زیر زمان اجرای الگوریتم ها به ازای مقادیر مختلف  $n$  نشان داده شده است:

class	$n=2$	$n=16$	$n=256$	$n=1024$
1	1	1	1	1
$\log(n)$	1	4	8	10
$n$	2	16	256	1024
$n\log(n)$	2	64	2048	10240
$n^2$	4	256	65536	1048576
$n^3$	8	4096	16777216	1.07E+09
$2^n$	4	65536	1.16E+77	1.8E+308

## تابع رشد

برای اینکه بدانیم برنامه‌ی ما برای چه ورودی‌هایی حدوداً چه زمانی برای اجرا نیاز دارد نیاز به تعریف یک نماد هستیم. در شکل زیر نمودار زمان اجرای bubble\_sort و algorithm1 به همراه توابع  $n$  و  $3n^2$  به تصویر کشیده شده‌اند:

```
In [20]: def plot2(max_n):
    """
    این تابع نمودار چهار تابع مختلف را برای مقایسه رشد آن‌ها رسم می‌کند
    """
    figure(figsize=(20, 20)) # ایجاد یک شکل
    xlabel("n") # برچسب محور X
    ylabel("Time") # برچسب محور Y

    C = 1.5 # یک ثابت برای مقایسه بهتر توابع
    x = range(1, max_n) # از 1 تا max_n-1

    # &#x2314; تابع T_1(n) = C * (3n + 1)
    y1 = [C * (3 * n + 1) for n in x]
    plot(x, y1, 'r', label='T_1(n)=C * (3n + 1)')

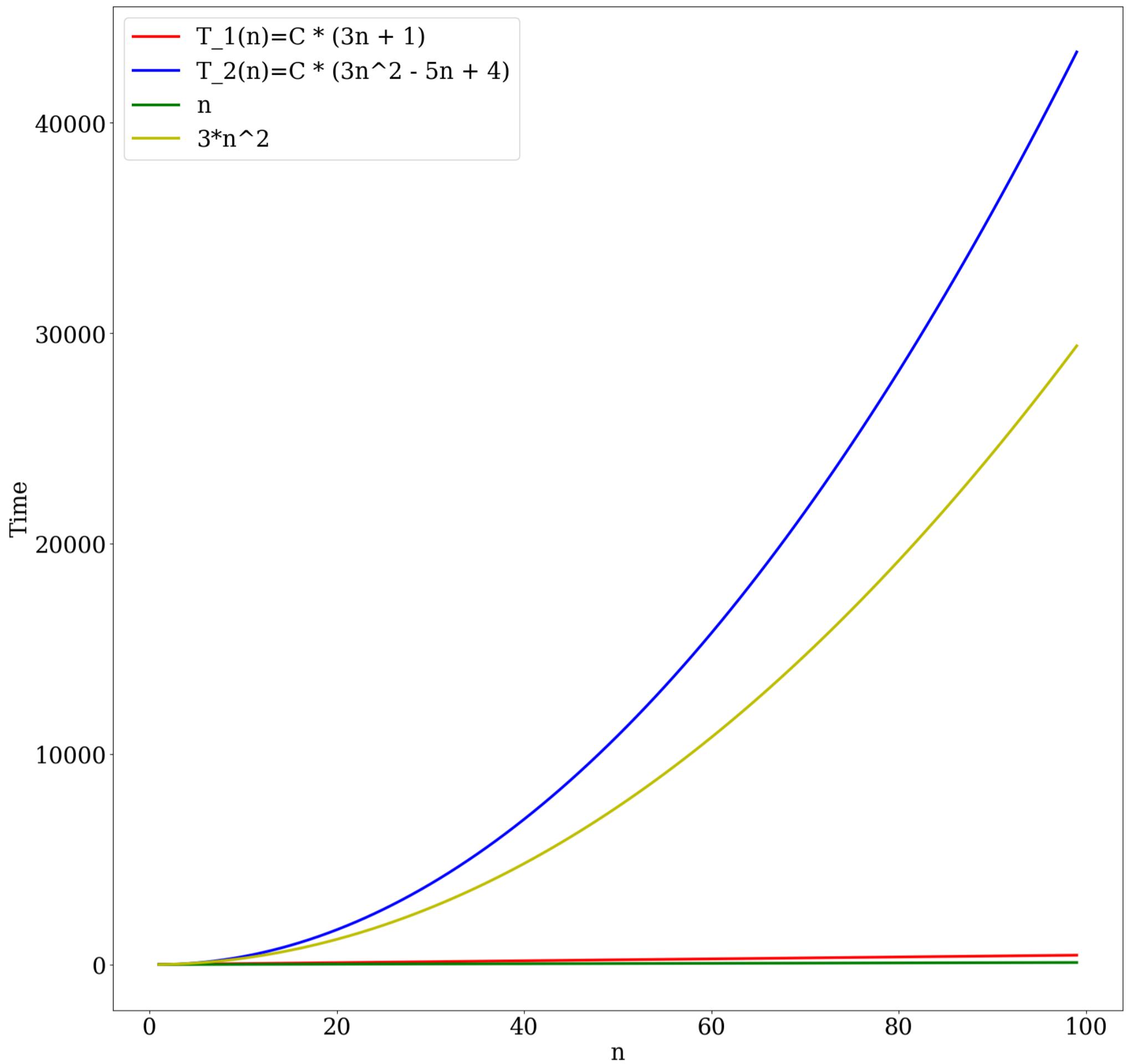
    # &#x2314; تابع T_2(n) = C * (3n^2 - 5n + 4)
    y2 = [C * (3 * (n * n) - 5 * n + 4) for n in x]
    plot(x, y2, 'b', label='T_2(n)=C * (3n^2 - 5n + 4)')

    # &#x2314; n
    y3 = [n for n in x]
    plot(x, y3, 'g', label='n')

    # 3 &#x22c5;n^2
    y4 = [3 * (n * n) for n in x]
    plot(x, y4, 'y', label='3*n^2')

    legend(loc=2) # نمایش راهنمای در موقعیت بالا-چپ
    show() # نمایش نمودار

# #&#x2314; plot2 لیکن max_n = 100
plot2(100)
```



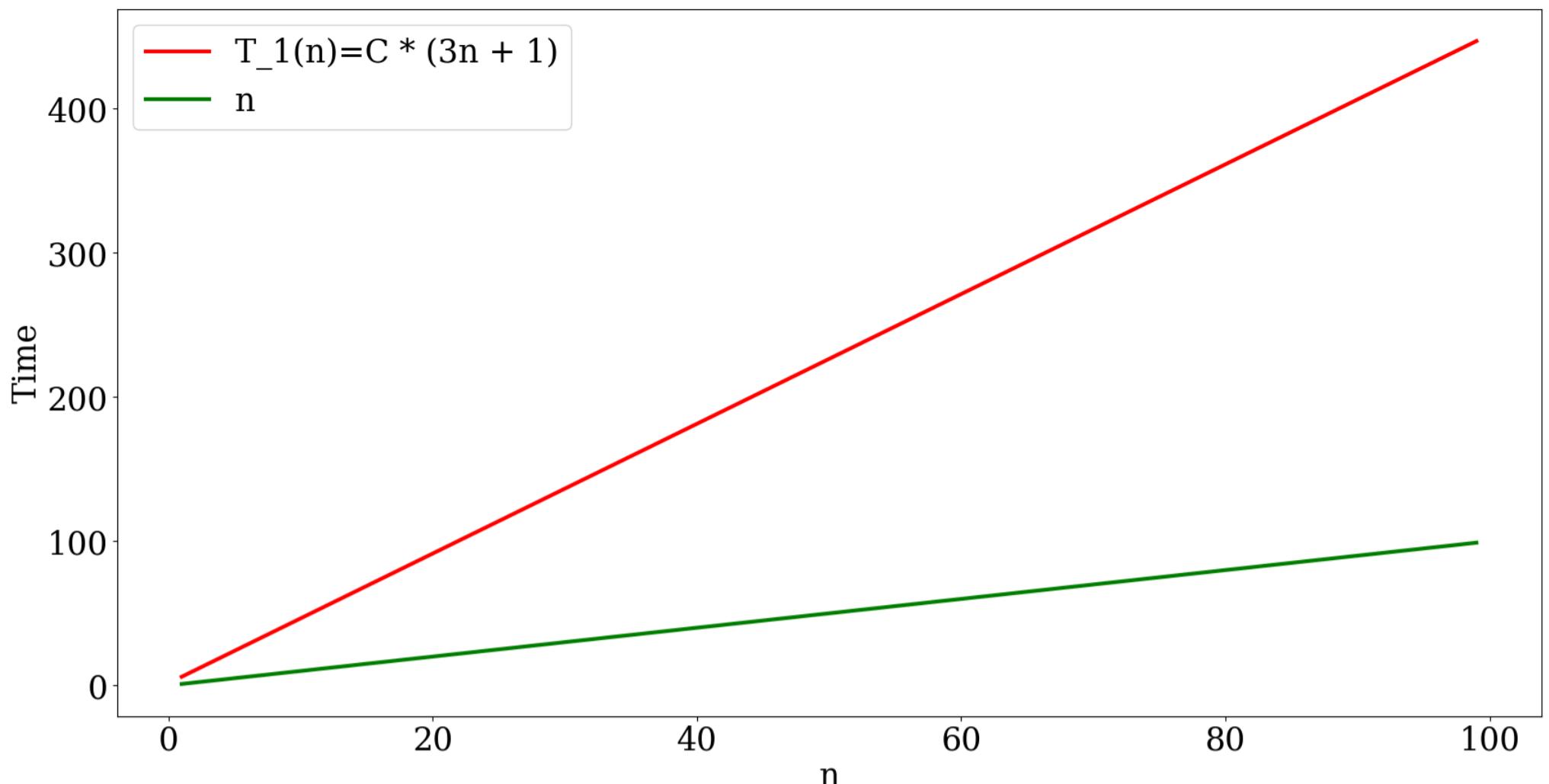
```
In [21]: figure(figsize=(20, 10)) # ایجاد یک شکل (figure) برای نمودار با اندازه مشخص (figure)
xlabel("n") # برچسب محور X
ylabel("Time") # برچسب محور Y

یک ثابت برای مقایسه بهتر توابع مقدار
max_n = 100 # حداکثر اندازه n رسم نمودار
x = range(1, max_n) # مقادیر n از 1 تا max_n-1

# برای تابع y محاسبه مقادیر T_1(n) = C * (3n + 1)
y1 = [C * (3 * n + 1) for n in x]
# با رنگ قرمز و برچسب مربوطه T_1(n) رسم نمودار
plot(x, y1, 'r', label='T_1(n)=C * (3n + 1)')

# برای تابع y محاسبه مقادیر n
y2 = [n for n in x]
# با رنگ سبز و برچسب مربوطه n رسم نمودار
plot(x, y2, 'g', label='n')

legend(loc=2) # نمایش راهنمای Legend (Legend) در موقعیت بالا-چپ نمودار
show() # نمایش نمودار
```



همان‌طور که در نمودارهای بالا مشاهده می‌شود با افزایش  $n$ ، کران بالای مرتب‌سازی حبابی نزدیک تابع  $n^2$  می‌ماند و کران بالای الگوریتم ابتدایی نزدیک تابع  $n$  می‌ماند اما این دو دسته بهوضوح از یکدیگر دور می‌شوند. به شکل دقیق‌تر می‌توان گفت  $\lim_{n \rightarrow \infty} \frac{T_1(n)}{T_2(n)}$  هر یک برابر با عددی ثابت می‌شوند درحالی‌که  $\lim_{n \rightarrow \infty} \frac{T_2(n)}{n}$  و  $\lim_{n \rightarrow \infty} \frac{T_1(n)}{n^2}$  چندجمله‌ای خطی است که با افزایش  $n$  بی‌کران بزرگ می‌شود.

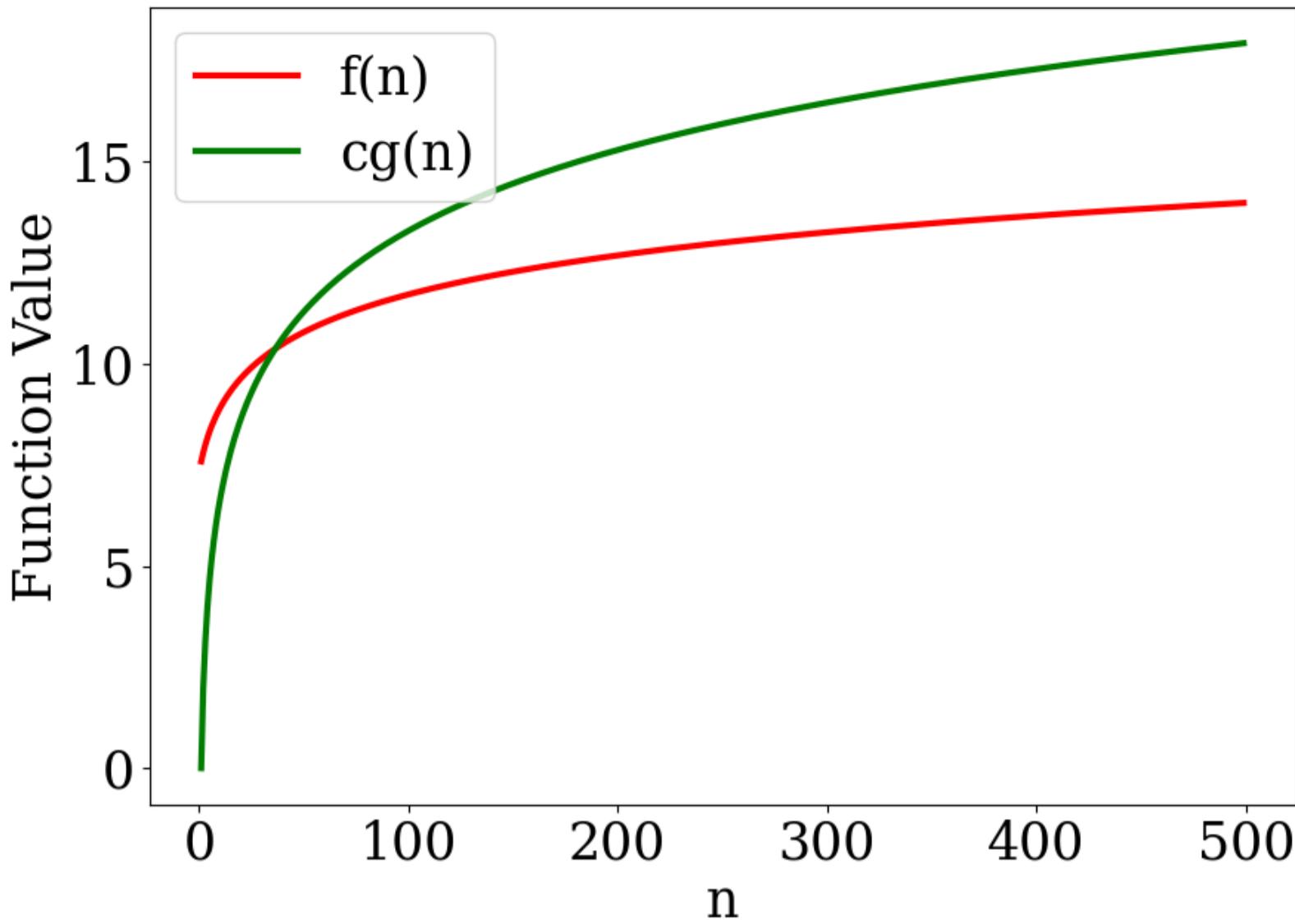
بنابراین با وجود آن‌که تابع کران بالای مرتب‌سازی حبابی و تابع  $n^2$  متفاوت هستند، به نظر می‌آید در  $n$ ‌های به اندازه‌ی کافی بزرگ، تفاوت اصلی این توابع بین دو دسته از آن‌هاست که یک دسته نزدیک تابع  $n$  باقی می‌ماند و دسته‌ی دیگر نزدیک تابع  $n^2$  باقی می‌ماند. حال می‌توان گفت که بنابراین در مقایسه‌ی زمان الگوریتم‌ها بین دو الگوریتم از یک دسته تفاوت چندانی وجود ندارد و دو الگوریتم زمانی به طور قابل توجه از لحاظ زمانی متفاوت هستند که از دو دسته متفاوت باشند.

حال لازم است که تعریفی دقیق‌تر برای مفهوم دسته ارائه کنیم. برای این‌کار از مجموعه‌های زیر استفاده می‌کنیم:

(الف)

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c, n_0 : 0 \leq f(n) \leq cg(n) \forall n > n_0\}$$

```
In [22]: import math # برای استفاده از توابع لگاریتم math وارد کردن مازول
figure(figsize=(10, 7)) # ایجاد یک شکل
xlabel("n") # X برچسب محور
ylabel("Function Value") # Y برچسب محور
max_n = 500 # برای رسم نمودار n حداکثر اندازه
x = range(1, max_n) # مقدار n از 1 تا max_n-1
y = [math.log(n + 5, 2) + 5 for n in x] # رنگ قرمز و برچسب مربوطه f(n) رسم نمودار
plot(x, y, 'r', label='f(n)')
# برای تابع y محاسبه مقادیر
y = [2 * math.log(n, 2) for n in x] # رنگ سبز و برچسب مربوطه cg(n) رسم نمودار
plot(x, y, 'g', label='cg(n)')
dr موقعیت بالا-چپ نمودار (Legend) نمایش راهنمایی # نمایش نمودار
show()
```



(ب)

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 : 0 \leq cg(n) \leq f(n) \forall n > n_0\}$$

In [23]: `import math # برای استفاده از توابع لگاریتم math وارد کردن مازول`

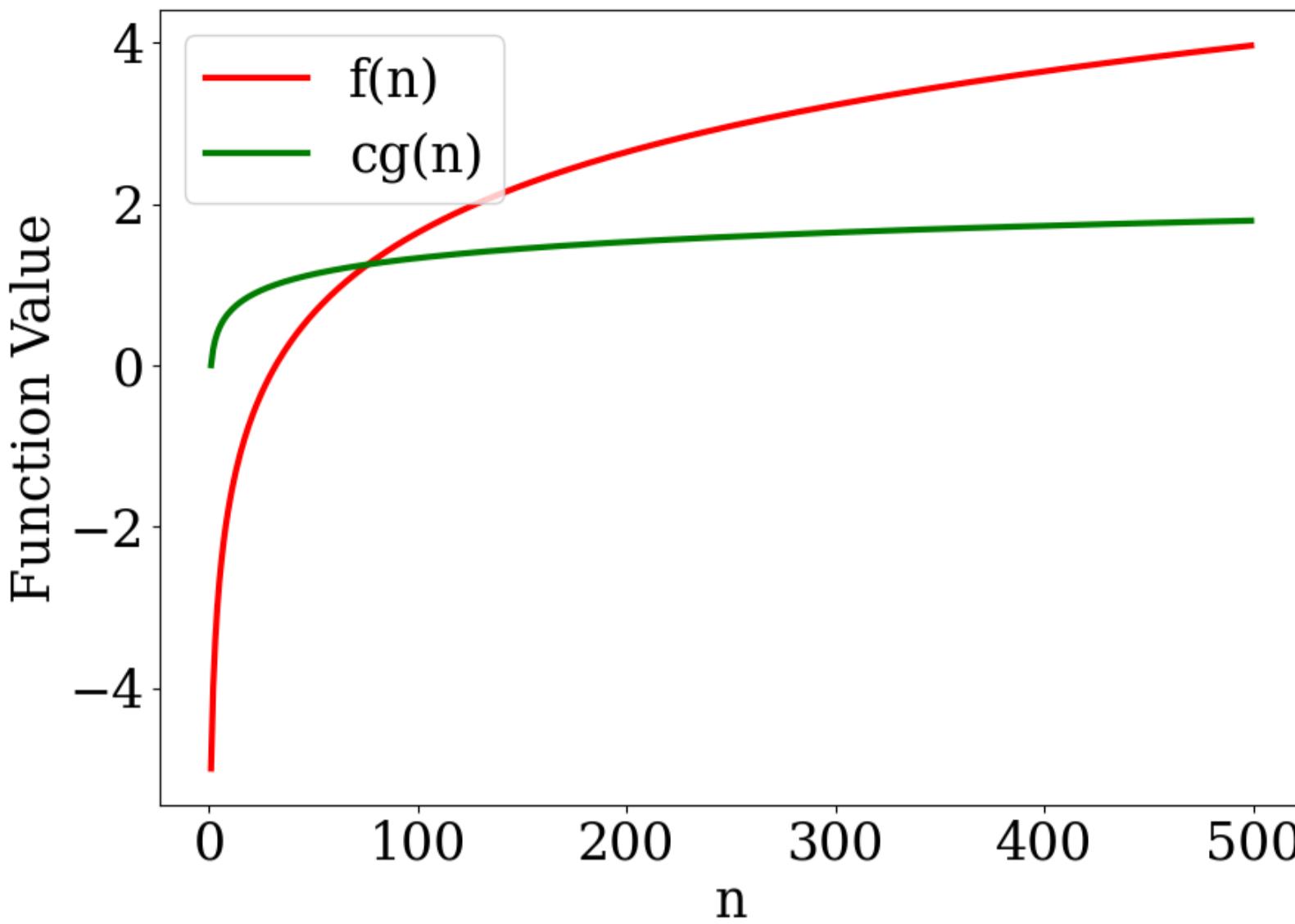
```
جديد برای نمودار با اندازه مشخص (figure) ایجاد یک شکل # ایجاد یک شکل # برچسب محور X
xlabel("n") # برچسب محور # برچسب را برای عمومیت بیشتر تغییر دادم Y برچسب محور # برچسب را برای عمومیت بیشتر تغییر دادم
ylabel("Function Value") # برچسب را برای عمومیت بیشتر تغییر دادم

برای رسم نمودار حداقل n حداقل n # برای رسم نمودار با اندازه max_n = 500
x = range(1, max_n) # مقدار n از 1 تا max_n-1

# برای تابع y محاسبه مقادیر f(n) = Log2(n) - 5
y = [math.log(n, 2) - 5 for n in x]
# رنگ قرمز و برچسب مربوطه رسم نمودار
plot(x, y, 'r', label='f(n)')

# برای تابع y محاسبه مقادیر cg(n) = log2(n) / 5
y = [math.log(n, 2) / 5 for n in x]
# رنگ سبز و برچسب مربوطه رسم نمودار
plot(x, y, 'g', label='cg(n)')

در موقعیت بالا-چپ نمودار (Legend) نمایش راهنمای # نمایش نمودار
show() # نمایش نمودار
```



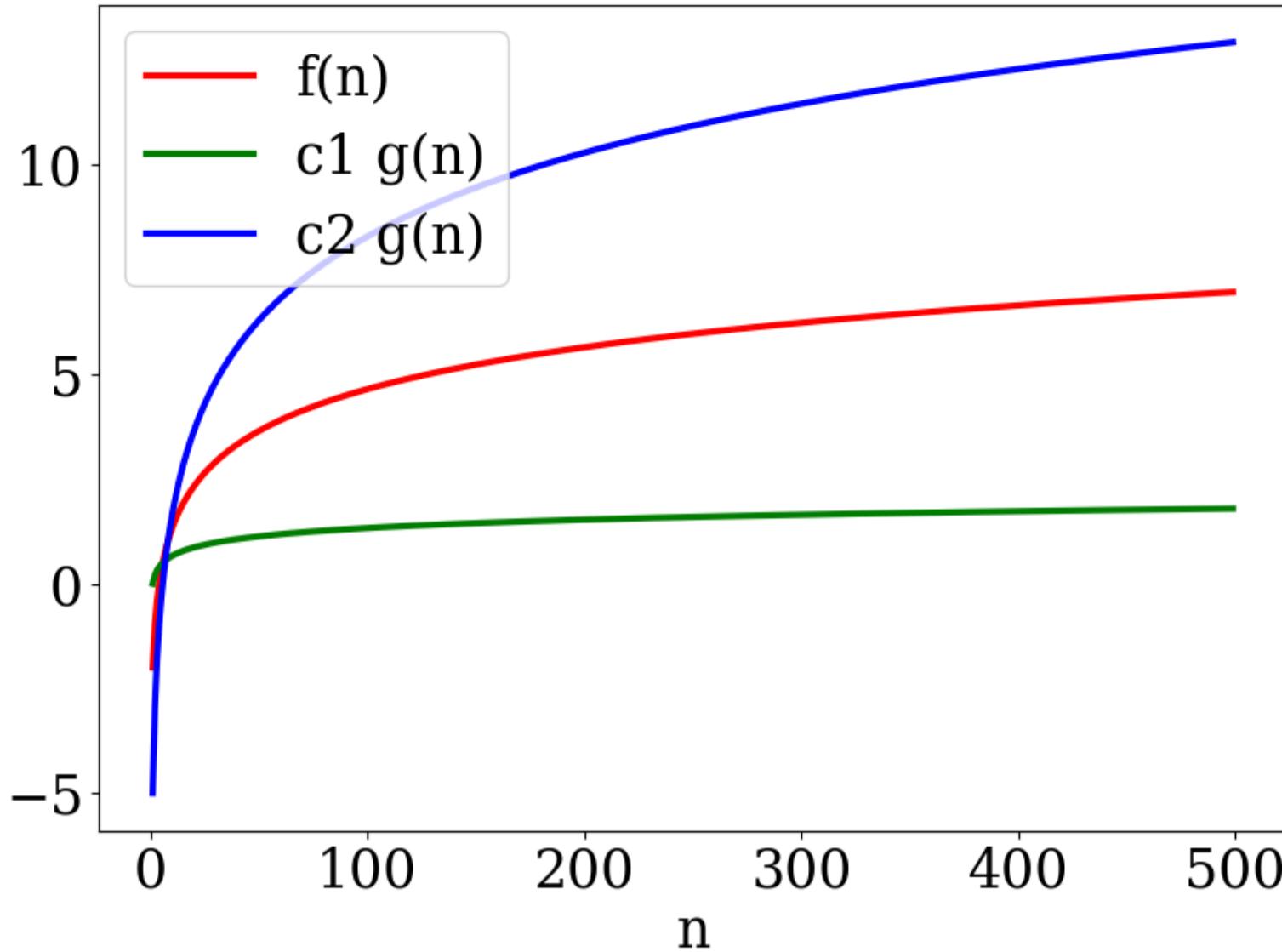
(ج)

$$\Theta(g(n)) = \{f(n) \mid f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))\}$$

به عبارت دیگر:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n > n_0\}$$

```
In [24]: import math # برای استفاده از توابع لگاریتم math وارد کردن مازوں
figure(figsize=(10, 7)) # ایجاد یک شکل (figure) برای نمودار با اندازه مشخص (10, 7)
xlabel("n") # برجسب محور X
# ylabel("Function Value") # این خط حذف شده است - (برجسب را برای عمومیت بیشتر تغییر دادم) 7 برجسب محور Y
max_n = 500 # برای رسم نمودار n حداقل اندازه
x = range(1, max_n) # 1 تا max_n-1 مقدار n
# تغییر یافته f(n) = log2(n) - 2
y = [math.log(n, 2) - 2 for n in x]
# با رنگ قرمز و برجسب مربوطه f(n) رسم نمودار
plot(x, y, 'r', label='f(n)')
# تغییر یافته c1 g(n) = log2(n) / 5
y = [math.log(n, 2) / 5 for n in x]
# با رنگ سبز و برجسب مربوطه c1 g(n) رسم نمودار
plot(x, y, 'g', label='c1 g(n)')
# تغییر یافته c2 g(n) = 2 * log2(n) - 5 (جدید)
y = [2 * math.log(n, 2) - 5 for n in x]
# با رنگ آبی و برجسب مربوطه c2 g(n) رسم نمودار
plot(x, y, 'b', label='c2 g(n)')
# در موقعیت بالا-چپ نمودار (Legend) نمایش راهنمای نمایش نمودار
legend(loc=2)
show()
```



در واقع اگر یک تابع  $f(n)$  در  $\mathcal{O}(g(n))$  باشد به عبارت نادقیق می‌توان گفت که این تابع برای  $n$ ‌های به اندازه‌ی کافی بزرگ،  $f(n) \leq g(n)$  و همچنین درمورد  $f(n) \in \Omega(g(n))$  می‌توان به صورت شهودی عبارت  $f(n) \geq g(n)$  را به کار برد. با توجه به این دو عبارت اگر  $f(n) \in \Theta(g(n))$  در یک دسته قرار دارد.

برای بررسی توابع، سعی می‌کنیم تابع ساده‌تر و همدسته‌ی آن را پیدا کنیم. برای الگوریتم ابتدایی، این تابع  $f(n) = n^2$  است. در هنگام یافتن تابع ساده‌تر از ضرایب صرف نظر می‌کنیم و بزرگ‌ترین عنصر در تابع اولیه را در نظر می‌گیریم.

$.f(x) \leq cg(x)$  که به ازای  $x_0 = 5$  وجود دارد ( $c = 1, x_0 = 5$ )

## چند مثال

تابع  $f$  در کد زیر در بدترین حالت در چه زمانی اجرا می‌شود؟  
تابعی همدسته با تابع زمان اجرای  $f$  پیدا کنید. ساده‌ترین این توابع کدام هستند؟

In [25]:

```
def f(N):
    """
    این تابع شامل چهار حلقه تودرتو است که یک شمارنده را افزایش می‌دهد.
    هدف از این مثال، نمایش پیچیدگی زمانی تابعی با حلقه‌های تودرتو است.
    """
    sum = 0 # عملیات: مقداردهی اولیه (1 عملیات)
    # حلقه بیرونی: از 0 تا N-1
    for i in range(N):
        # حلقه دوم: از i+1 تا N-1
        for j in range(i + 1, N):
            # حلقه سوم: از j+1 تا N-1
            for k in range(j + 1, N):
                # حلقه چهارم: از k+1 تا N-1
                for h in range(k + 1, N):
                    # عملیات: جمع و انتساب (این عملیات به تعداد دفعات اجرای داخلی‌ترین حلقه انجام می‌شود)
                    sum += 1 # در نهایت تعداد دفعات اجرای داخلی‌ترین حلقه را نشان می‌دهد 'sum' این تابع چیزی را برنمی‌گرداند، اما #
#
```

زمان اجرای تابع  $g$  در کد زیر را نیز بررسی کنید.

In [26]:

```
def g(N, x, A):
    """
    انجام می‌دهد A را در آرایه مرتب شده (Binary Search) این تابع یک جستجوی دودویی
    انجام می‌دهد. A در آرایه مرتب شده است. در آرایه x هدف آن یافتن وجود عنصر
    آرایه مرتب شده‌ای که جستجو در آن انجام می‌شود.
    Args:
        N (int): طول آرایه A.
        x: عنصری که به دنبال آن هستیم.
        A (list): آرایه مرتب شده‌ای که جستجو در آن انجام می‌شود.
    """

```

```

Returns:
    در غیر این صورت False، یافت شود A در آرایه x اگر True.
    """
    پایان بازه جستجو (یک واحد بعد از آخرین عنصر) e: شروع بازه جستجو s: حلقه اصلی جستجو: تا زمانی که بازه جستجو بیش از یک عنصر داشته باشد
    while e - s > 1:
        mid = (s + e) // 2 # محسوبه اندیس میانی بازه
        بود، بازه جستجو را به نیمه راست منتقل کن x اگر عنصر میانی کوچکتر با مساوی
        if (A[mid] <= x):
            s = mid
        بازه جستجو را به نیمه چپ منتقل کن، (بود x عنصر میانی بزرگتر از) در غیر این صورت
        else:
            e = mid
        باشد x اندیس عنصری است که ممکن است برابر با s، پس از اتمام حلقه
        است یا خیر x واقعاً برابر با s بررسی می‌کنیم که آیا عنصر در اندیس
        return A[s] == x

```

```
In [27]: # آرایه مرتب شده برای جستجو
A = [1, 5, 10, 22, 31, 44, 55, 69, 75, 85, 100]
N = len(A) # محسوبه طول آرایه

# تست تابع با دو مقدار مختلف
print(g(N, 78, A)) # خروجی False (چون 78 در آرایه وجود ندارد)
print(g(N, 85, A)) # خروجی True (چون 85 در آرایه وجود دارد)
```

False

True

## مثال

$$T(n) = a_n n^k + \dots + a_1 n + a_0$$

$$T(n) \in \mathcal{O}(n^k)$$

## مثال

$$T(n) = 2^{n+10}$$

$$T(n) \in \mathcal{O}(2^n)$$

## مثال

$$T(n) = 2^{10n}$$

$$T(n) \notin \mathcal{O}(2^n)$$

## مثال

برای مقایسه‌ی توابع به هر حال لازم است بتوانیم توابع ساده را با هم مقایسه کنیم. اگر توابع زیر، تابع همدسته‌ی ساده‌تری دارند آن را پیدا کنید و سپس این توابع را با هم مقایسه کنید.

$$\begin{aligned} & 2^n \\ & 2^{2^n} \\ & n^2 \\ & n^2 \log n \\ & n^{\log n} \\ & n^n \end{aligned}$$

## مثال

$$\Theta(n!)$$

# مثال

هر یک از روابط زیر را اثبات کنید:

$$n! = \mathcal{O}(n^n)$$

$$n! = \Omega(2^n)$$

$$\lg(n!) = \Theta(n \lg n)$$

## مقایسه تابع های مهم

constant = ثابت،  $\mathcal{O}(1)$

logarithmic =  $\mathcal{O}(\log n)$

linear = خطی،  $\mathcal{O}(n)$

$n \log n = \mathcal{O}(n \log n)$

quadratic =  $\mathcal{O}(n^2)$

cubic =  $\mathcal{O}(n^3)$

polynomial، k =  $\mathcal{O}(n^k)$  ثابت است

exponential =  $\mathcal{O}(k^n)$

factorial =  $\mathcal{O}(n!)$

در نمودار زیر رشد چند تابع را می‌توانید مقایسه کنید:

```
In [1]: import math
import matplotlib.pyplot as plt

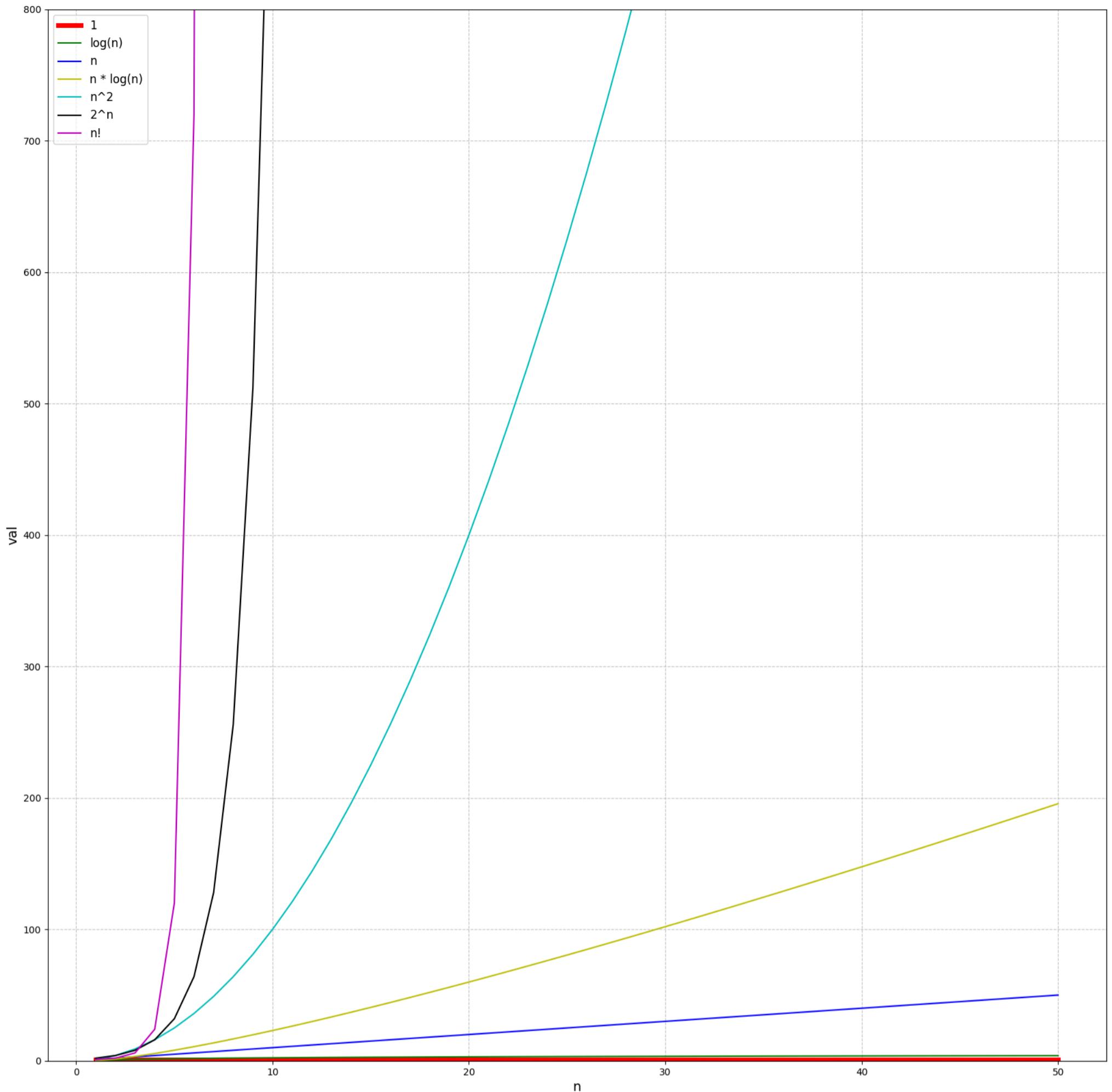
# ترتیبات اولیه نمودار
plt.figure(figsize=(20, 20))
plt.xlabel("n", fontsize=14)
plt.ylabel("val", fontsize=14)
plt.ylim(0, 800) # محدود کردن محور عمودی برای نمایش بهتر تابع کوچکتر

# محدوده متغیر n
max_n = 50
x = list(range(1, max_n + 1))

# رسم توابع مختلف
plt.plot(x, [1 for _ in x], 'r', label='1', linewidth=5)
plt.plot(x, [math.log(n) for n in x], 'g', label='log(n)')
plt.plot(x, [n for n in x], 'b', label='n')
plt.plot(x, [n * math.log(n) for n in x], 'y', label='n * log(n)')
plt.plot(x, [n ** 2 for n in x], 'c', label='n^2')
plt.plot(x, [2 ** n for n in x], 'k', label='2^n')
plt.plot(x, [math.factorial(n) for n in x], 'm', label='n!')

# ترتیبات نمودار
plt.legend(loc='upper left', fontsize=12)
plt.title("Compare each function's growth", fontsize=16, pad=20)
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```

Compare each function's growth



In [ ]:

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل سوم، بخش دوم: الگوریتم‌های بازگشتی

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

## فهرست

- مقدمه
- روش حدس و استقرا
- قضیه ساندویچ
- روش تغییر متغیر
- قضیه اصلی
- مرتب‌سازی ادغامی
- روش درخت بازگشت
- برای مطالعه: اثبات قضیه اصلی
- برای مطالعه: مثال ابتکاری

**مقدمه** قرارداد: با توجه به این که واژه‌ی تابع در برنامه‌نویسی و ریاضی دو مفهوم نزدیک به هم اما متفاوت دارد، در سرتاسر این فصل از تابع به معنای ریاضی آن استفاده می‌شود و هرگاه بخواهیم از یک تابع برنامه صحبت کنیم، از واژه‌های روند یا رویه استفاده می‌کنیم. همچنین در این متن هر جا دامنه‌ی یک تابع مشخص نشده است، اعداد طبیعی مد نظر هستند و همه‌ی ثابت‌ها اعداد حقیقی مثبتند. بسیاری از الگوریتم‌هایی که ما در این درس با آن‌ها سر و کار داریم بر اساس تکرار بازگشتی یک یا چند رویه کار می‌کنند. برای مثال مرتب‌سازی ادغامی (که در ادامه بررسی می‌شود) در زمان اجرا بر روی یک آرایه، به صورت بازگشتی خودش را روی دو نیمه‌ی ابتدا و انتهای آرایه فراخوانی کرده و سپس دو نیم‌آرایه‌ی مرتب حاصل را ادغام می‌کند. تحلیل زمان اجرای این نوع الگوریتم‌ها کمی پیچیده‌تر از الگوریتم‌های سرراستی مثل مرتب‌سازی شمارشی یا حبابی است؛ چرا که در زمان تحلیل آن‌ها با توابع بازگشتی سر و کار پیدا می‌کنیم. به عنوان اولین و ساده‌ترین نمونه بیایید به جست‌وجوی دودویی دقت کنیم. فرض کنید یک آرایه‌ی مرتب با  $n$  عنصر به شما داده شده است و سپس یک نفر مرتب‌اً از شما سؤال‌هایی به این شکل می‌پرسد: آیا عدد  $x$  در این آرایه ظاهر شده است؟ هدف شما این است که به این سؤال‌ها در کوتاه‌ترین زمان ممکن پاسخ دهید. کمی بعدتر در همین درس با تکنیک درهم‌سازی آشنا خواهید شد که می‌تواند الگوریتم‌های جالب‌تری برای این کار به شما بدهد، اما در حال حاضر از این الگوریتم که نام آن جست‌وجوی دودویی است استفاده می‌کنیم: هر بار عنصر وسط آرایه را انتخاب می‌کنیم. اگر این عنصر برابر با همان  $x$  بود که به دنبال آن می‌گردیم، جواب سؤال پیدا شده است، در غیر این صورت اگر این عنصر از هدف ما کوچک‌تر بود فقط در نیمه‌ی بعد از عنصر وسط به جست‌وجو ادامه می‌دهیم و در غیر این صورت فقط در نیمه‌ی قبل از عنصر وسط. این روند را می‌توان به شکل بازگشتی زیر پیاده‌سازی کرد: توجه داشته باشید در اینجا  $up$  و  $down$  انتهای و ابتدای بازه‌ای هستند که در حال جست‌وجو در آن‌یم؛ البته لازم نیست به جزئیات پیاده‌سازی رویه جست‌وجوی دودویی دقت کنید، زیرا در فصول آینده‌ی این درس به تفصیل مورد بررسی قرار می‌گیرد.

```
In [1]: def binary_search(array, x, down, up):  
    """  
    جستجوی دودویی بازگشتی در یک آرایه مرتب‌شده  
    را جستجو می‌کند (بازه نیم‌باز) [down, up-1] این تابع بازه  
  
    Args:  
        آرایه مرتب‌شده (صعودی) array (list):  
        مقداری که به دنبال آن هستیم x:
```

اندیس شروع بازه جستجو (شامل): down (int)  
اندیس پایان بازه جستجو (نashامل): up (int)

Returns:  
str: "FOUND" در غیر این صورت "NOT FOUND" ، اگر مقدار یافت شود  
""

حالت پایه: بازه جستجو به یک عنصر کاهش یافته #  
if down == up - 1:  
است x بررسی اینکه آیا تنها عنصر موجود برابر #  
if array[down] == x:  
return "FOUND"  
else:  
return "NOT FOUND"

محاسبه اندیس میانی بازه #  
mid = (up + down) // 2

بررسی عنصر میانی #  
if array[mid] == x:  
return "FOUND"

بود، جستجو در نیمه راست x اگر عنصر میانی کوچکتر از #  
elif array[mid] < x:  
return binary\_search(array, x, mid, up)

جستجو در نیمه چپ ، (بود x عنصر میانی بزرگتر از) در غیر این صورت #  
else:  
return binary\_search(array, x, down, mid)

In [2]: array = [1, 2, 3, 4, 5, 6, 100, 110]  
print(binary\_search(array, 3, 0, len(array)))  
print(binary\_search(array, 7, 0, len(array)))

FOUND  
NOT FOUND

حال بیایید زمان اجرای این رویه بازگشتی را به دست آوریم. فرض کنید زمان اجرای این رویه در بدترین حالت (حالی که بیشترین زمان مصرف می‌شود) روی یک آرایه با اندازه  $n$  را با  $T(n)$  نمایش دهیم.  
می‌خواهیم کلاس پیچیدگی تابع  $T(n)$  را به دست آوریم.

در هر بار فراخوانی بازگشتی این رویه طول بازه‌ای که در آن به جستجو می‌پردازیم کاهش می‌یابد، پس امکان ندارد جستجو تا ابد ادامه پیدا کند.

همچنین رویه binary\_search حداقل  $n$  بار فراخوانی می‌شود و هر بار فراخوانی این تابع زمانی ثابت مصرف می‌کند.  
پس می‌توانیم نتیجه بگیریم که فرآیند جستجو در مجموع از  $O(n)$  است.

اما می‌توانیم هزینه زمانی را به طور هوشمندانه‌تری حساب کنیم؛ اگر  $\ell$  را برابر با لگاریتم طول بازه‌ای که در آن جستجو می‌کنیم تعریف کنیم، این بار مشاهده می‌کنیم که  $\ell$  هم با هر فراخوانی بازگشتی رویه کاهش می‌یابد و درنهایت به صفر می‌رسد؛ در نتیجه زمان اجرای این الگوریتم از  $O(\log n)$  است. (در اینجا دیگر  $\ell$  یک عدد صحیح نبود ولی اگر جزء صحیح آن را در نظر بگیریم آنگاه مشکل حل است). آیا می‌توانید نشان دهید که این الگوریتم در بدترین حالت از  $\Omega(\log n)$  است و در نتیجه از  $\Theta(\log n)$  می‌باشد؟ این کار را می‌توانید با ساختن بی‌نهایت مثال انجام دهید! تذکر: در سرتاسر این فصل هر کجا صریحاً بیان نشد در مبنای ۲ لگاریتم می‌گیریم.

**روش حدس و استقرا** کاری که در اینجا برای تحلیل زمان اجرای جستجوی دودویی انجام دادیم بسیار سخت بود.  
اگر بخواهیم الگوریتم‌های بازگشتی را این‌طور تحلیل کنیم باید همیشه به دنبال متغیر مناسب بگردیم و همان‌طور که در مثال بالا مشخص است، پیدا کردن متغیرهای دیگر می‌تواند جواب‌های درست ولی نه چندان بهینه به ما بدهد. راستش را بخواهید تا پایان این قسمت با هیچ روش غیرحدسی برخورد نخواهید کرد، اما تکنیک‌های متفاوتی را خواهید دید که به شما کمک می‌کند بهتر حدس بزنید یا برای اثبات حدستان کمتر مشقت بکشید. البته این تکنیک‌ها به ترتیب از ساده به سخت (از نظر من) ارائه می‌شوند و در تحلیل یک الگوریتم در دنیای واقعی (دنیای خارج از دانشگاه!) بهترین راهکار این است که از ساده‌ترین روش‌ها استفاده کنید و هرگاه آن‌ها جواب ندادند به روش‌های قوی‌تر متولّ شوید. اولین قدم ما آن است که از کد فاصله گرفته و برای تحلیل زمان الگوریتم‌مان این زمان را به شکل یک تابع بازگشتی ریاضی نمایش دهیم. در مورد جستجوی دودویی اگر فرض کنیم زمان لازم برای این جستجو بر اساس تعداد مقایسه‌هایی که بین عناصر می‌کنیم تعیین می‌شود و  $T(n)$  را برابر با تعداد مقایسه‌ها در اجرای یک نمونه جستجوی دودویی روی یک آرایه به طول  $n$  در بدترین حالت (حالی که بیشترین مقایسه را لازم دارد) تعریف کنیم، می‌بینیم که  $T(1) = 1$  و

برای هر  $n \geq 2$  داریم:  $T(n) = T(\left\lceil \frac{n-1}{2} \right\rceil) + 1$ .

از این پس فقط به بررسی شیوه‌های پیدا کردن پیچیدگی توابعی می‌پردازیم که به شکل بازگشته تعريف شده‌اند و به اینکه این توابع زمان اجرای چه رویه‌هایی را نشان می‌دهند توجه

$$T(n) = \begin{cases} 1 & n = 1 \\ T(\left\lceil \frac{n-1}{2} \right\rceil) + 1 & n > 1 \end{cases}$$

نمی‌کنیم. در اینجا می‌خواهیم کلاس پیچیدگی تابع زیر را پیدا کنیم:

باز ساده‌ترین راهی که داریم حدس زدن است! اگر خوب حدس بزنیم، می‌توانیم مسئله را حل کنیم. ما با توجه به این که جواب مسئله را می‌دانیم از قسمت قبل تقلب کرده و حدس می‌زنیم که  $T(n) \in \Theta(\log n)$ . حال باید این حدس را ثابت کنیم. یکی از روش‌های بسیار کارآمد در این موقع استفاده از استقرای ریاضی است. می‌خواهیم با استفاده از استقرای ریاضی نشان دهیم که  $T(n) \in O(\log n)$ . این معادل آن است که نشان دهیم که یک عدد ثابت  $c$  وجود دارد که برای هر  $n$  طبیعی به اندازه‌ی کافی بزرگ داریم  $T(n) \leq c \log n$ . اگر این تعریف با تعریفی که قبلاً از  $O$  دیده‌اید کمی متفاوت است، اندکی درنگ کنید و به خودتان بقبولانید که هر دو تعریف یک چیز را بیان می‌کنند. با این حساب حالا باید دو کار انجام دهیم. اول باید یک  $c$  انتخاب کنیم و بعد باید با استقرا نشان دهیم که برای هر  $n$  داریم  $T(n) \leq c \log n$ . خیلی راحت می‌توانیم بگوییم بگذار  $c$  یک مقدار ثابت مثلاً ۱۰۰ باشد و بعد سعی کنیم با استقرا حکم را ثابت کنیم و اگر جواب نگرفتیم،  $c$  را بزرگ‌تر انتخاب کنیم و باز همین کار را ادامه دهیم و امیدوار باشیم که بالآخره به یک  $c$  برسیم که کار کند. راه بهتر اما آن است که  $c$  را انتخاب نکنیم و استقرایمان را بزنیم و وقتی که استقرا تمام شد  $c$  را طوری انتخاب کنیم که روشنان جواب بدهد. پس بباید استقرا بزنیم. ما در اینجا از استقرای قوی استفاده می‌کنیم. فرض استقرا آن است که به ازای هر  $n < k$  داریم  $T(k) \leq c \log k$  و حکم استقرا آن است که  $T(n) \leq c \log n$ .

حال داریم:

$$T(n) = T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1 \leq c \log \left\lceil \frac{n-1}{2} \right\rceil + 1 \leq c \log \frac{n}{2} + 1 \leq c \log n$$

دقت کنید که آخرین نابرابری فقط وقتی درست است که  $1 \leq c$  و بنابراین باید  $c$  را به اندازه‌ی کافی بزرگ انتخاب کنیم. حال آیا اثبات تمام شده است؟ خیر. هر استقرایی به یک پایه نیاز دارد و در اینجا به سادگی می‌توانید ببینید که پایه‌ی استقرا برای  $n = 1$  برقرار نیست، چرا که  $\log 1 = 0$  اما خوشبختانه ما نیازی نداریم که حکم استقرا را برای همه‌ی مقادیر  $n$  ثابت کنیم و کافیست برای  $n$ ‌های به اندازه‌ی کافی بزرگ درست باشد. پس می‌توانیم  $3, 2, n = 1$  را به عنوان پایه‌های استقرا در نظر بگیریم و  $c$  را طوری انتخاب کنیم که هم در شرط بالا صدق کند و هم آنقدر بزرگ باشد که حکم برای این دو پایه درست باشد. آن‌گاه با استقرا حکم ثابت شده است. (چرا نمی‌توانیم فقط  $2 = n$  را به عنوان پایه‌ی استقرا در نظر بگیریم؟) به عنوان یک تمرین سعی کنید با روشی مشابه ثابت کنید که  $T(n) \in \Omega(\log n)$ .

استقرا روش بسیار قدرتمندی است اما بررسی همه‌ی جزئیات آن مانند کاری که در اینجا انجام دادیم بسیار وقت‌گیر است و خیلی وقت‌ها لازم هم نیست. مثلاً فرض کنید که می‌خواهید این تابع را بررسی کنید (این تابع از مرتب‌سازی ادغامی به دست آمده است):

$$T(n) = \begin{cases} 0 & n = 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n) & n \geq 2 \end{cases}$$

در اینجا منظور از  $O(n)$  که در تعریف تابع آمده است این است که تابعی مانند  $f(n) \in O(n)$  وجود دارد که:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + f(n) & n \geq 2 \end{cases}$$

یا به عبارت دیگر عدد ثابت مثبت  $d$  وجود دارد که:

$$T(n) \leq \begin{cases} 0 & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + dn & n \geq 2 \end{cases}$$

بنابراین چون در اینجا فقط یک رابطه‌ی بازگشته برای یک کران بالا از تابع  $T$  داریم، امکان بررسی  $\Omega$  وجود ندارد و فقط به دنبال این هستیم که یک بررسی  $O$  انجام دهیم.

اگر باز مثل قسمت قبل به حدس زدن روی بیاوریم و حدس بزنیم که  $T(n) \in O(n^2)$  یا حتی بهتر  $T(n) \in O(n \log n)$ ، آنگاه باید این حدس را با استقرا ثابت کنیم. در اینجا مقداری که برای  $c$  انتخاب می‌کنیم به  $d$  هم وابسته خواهد بود که با توجه به این که  $d$  یک عدد ثابت است مشکلی ایجاد نمی‌کند. اما چیزی که مشکل‌ساز است آن کف و سقف است که استدلال‌ها را سخت‌تر می‌کنند. قطعاً برای ما بسیار ساده‌تر است که با این رابطه کار کنیم:

$$T'(n) = \begin{cases} 0 & n = 1 \\ 2T'(\frac{n}{2}) + O(n) & n > 1 \end{cases}$$

در نگاه اول به نظر نمی‌رسد که جواب این رابطه با رابطه قبلى فرق قابل توجهی داشته باشد و در واقعیت هم جوابشان یکیست. اگر بتوانیم این را ثابت کنیم، بعد می‌توانیم این تابع ساده‌تر را تحلیل کنیم و از جوابش استفاده کنیم. دقت کنید که این که در این رابطه از سقف‌ها و کف‌ها صرف نظر کردیم معادل این است که فرض کنیم  $n$  هایمان همیشه توان‌هایی از ۲ هستند. این نمونه‌ای از فرض‌هاییست که ما آنها را **فرض‌های سازگار می‌نامیم** و در موقع تحلیل پیچیدگی یک تابع از آنها بسیار استفاده می‌کنیم.

بگذارید یک تعریف دقیق از فرض‌های سازگار بدیم. فرض کنید که در تحلیل  $O$  به جای تابع بازگشته  $T$  تصمیم می‌گیرید که چند فرض اضافه کنید و تابع بازگشته  $T'$  را تحلیل کنید، در این صورت فرض‌های شما سازگار نامیده می‌شوند هرگاه  $T \in O(T')$ . به عبارت دیگر برای هر  $n$  به اندازه‌ی کافی بزرگ، فرض‌های اضافه شده یا باید مقدار تابع را افزایش دهند یا اگر کاهش می‌دهند حداقل با یک ضریب ثابت کاهش دهند.

در بسیاری از توابعی که ما به آنها برخورد می‌کنیم فرض‌های زیر سازگار هستند و ما آنها را بدون این که هر بار ثابت کنیم سازگارند به کار می‌بریم. دقت کنید که این فرض‌ها همیشه سازگار نیستند و موقع تحلیل الگوریتم‌ها همیشه باید مواضع سازگاری فرض‌هایتان باشید.

- این فرض که  $n$  توانی از ۲ یا یک عدد ثابت دیگر است.
- این فرض که کف‌ها و سقف‌ها را می‌توان نادیده گرفت.
- این فرض که در استقرا نیازی به چک کردن پایه نداریم و متعاقباً این که در تعریف تابع بازگشته نیازی به بیان حالت پایه نداریم.
- این فرض که تابع مورد بررسی صعودی است (خیلی وقت‌ها برای اثبات سازگاری این مورد باید به مسئله‌ی اصلی رجوع کنیم).
- این فرض که تابع مورد بحث مثبت است (یعنی هیچ‌گاه صفر نیست).

معمولًا اثبات سازگاری همه‌ی این فرض‌ها تکراری و شبیه به هم است و به مسئله بستگی چندانی ندارد. یک تمرین

خوب می‌تواند این باشد که ثابت کنید این فرض‌ها در مورد تابع‌هایی که تا اینجا بررسی کرده‌ایم سازگارند و بعد سعی کنید برای هر فرض تابعی بازگشتی پیدا کنید که با آن ناسازگار باشد. وقتی این کار را بکنید می‌بینید که تا چه میزان توابعی که این‌طور هستند شکل‌های عجیبی دارند و چه قدر بعید است که در تحلیل الگوریتم‌های واقعی به آن‌ها بخورد کنیم. اما همیشه باید به یاد داشته باشید که این توابع هم وجود دارند و اگر یک فرض ناسازگار اضافه کنید تحلیلتان غلط است. دقت کنید که در این مثال در مورد فرض چهارم، اگر تنها اطلاعاتی که داریم همان رابطه‌ی بازگشتی شامل  $O$  باشد نمی‌توانیم ثابت کنیم که تابع باید صعودی باشد.

حال به تحلیل زمان الگوریتم مرتب‌سازی ادغامی می‌پردازیم که بعد از فرض‌های صورت گرفته به این شکل در آمده است:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + dn$$

می‌خواهیم ثابت کنیم  $T(n) \leq cn \log n$ . تنها کاری که باید انجام دهیم بررسی گام استقرنا است:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + dn \leq 2 \times c \times \frac{n}{2} \times (\log n - 1) + dn \leq cn \log n - cn + dn \leq cn \log n$$

در این‌جا نابرابری آخر فقط وقتی درست است که  $c$  را بزرگ‌تر از  $d$  انتخاب کنیم.

حال به چند تکنیک دیگر می‌پردازیم که می‌تواند در تحلیل روابط بازگشتی کارساز باشد. تکنیک‌هایی که در این‌جا مطرح می‌کنیم قضیه‌ی ساندویچ و تغییر متغیر هستند. یک مسئله را هم با استفاده از حساب دیفرانسیل و انتگرال حل می‌کنیم. در آخر به قضیه‌ی اصلی می‌پردازیم که خلاصه‌اش این است که برای یک خانواده‌ی خیلی کاربردی از توابع بازگشتی، قبل از نفر دیگر استقرنا را انجام داده و شما می‌توانید به راحتی از پاسخش استفاده کنید. شاخه‌ای از ریاضیات به نام ترکیبیات آنالیزی تکنیک‌های پیشرفته‌تری برای تحلیل توابع بازگشتی ارائه می‌دهد که در صورت علاقه می‌توانید آن را جست‌وجو کنید.

**قضیه‌ی ساندویچ** حال یک قضیه مطرح می‌کنیم که به قضیه‌ی ساندویچ معروف است و در حقیقت حالت دو طرفه‌ی همان چیزیست که در مورد فرض‌های سازگار بیان کردیم. قضیه‌ی ساندویچ بیان می‌کند که اگر برای  $n$ ‌های به اندازه‌ی کافی بزرگ داشته باشیم  $(f(n) \in \Omega(i(n)) \text{ و } g(n) \in O(i(n)) \text{ و } h(n) \in \Theta(i(n)) \text{ آن‌گاه } c_1 f(n) \leq g(n) \leq c_2 h(n)$ . این قضیه را می‌توان به راحتی از خاصیت تعددی  $O$  و  $\Omega$  به عنوان روابطی تعریف شده بر یک کلاس از تابع‌ها نتیجه گرفت. این مثال را که بعدها کاربردی خواهد بود بررسی می‌کنیم:

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

شاید بهتر باشد که این تابع را این‌طور بنویسیم:

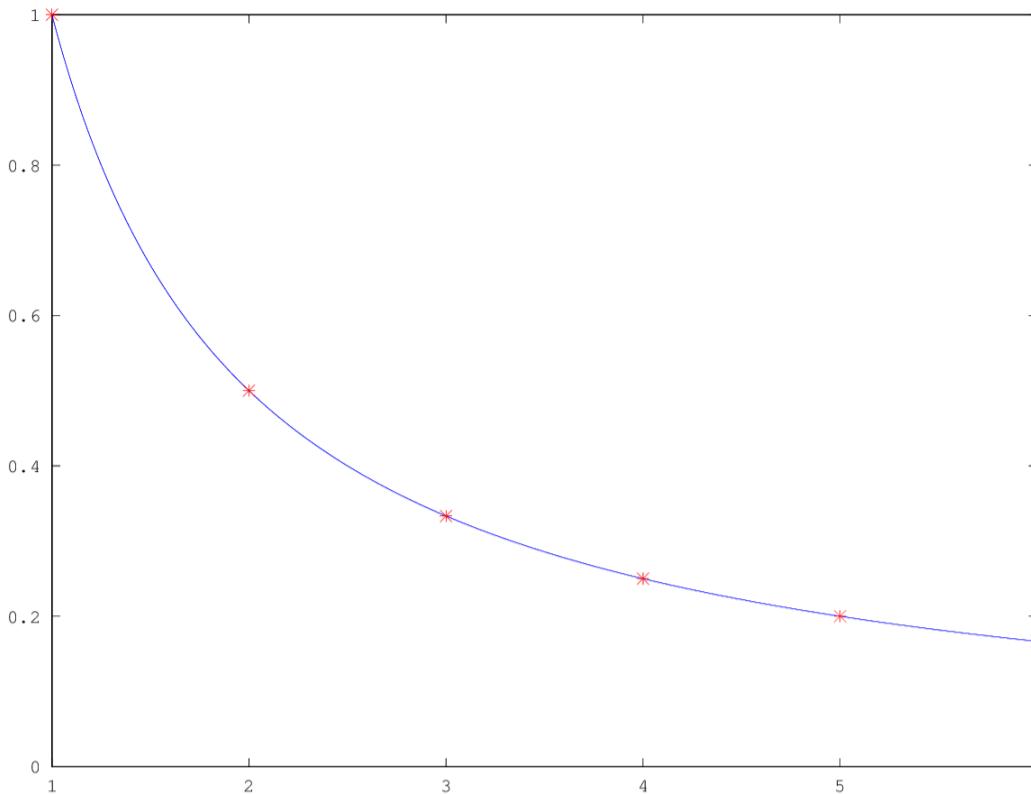
$$T(n) = \sum_{i=1}^n \frac{1}{i}$$

آن‌گاه داریم:

$$\int_1^n \frac{1}{x} dx \leq T(n) \leq \int_2^{n+1} \frac{1}{x} dx$$

برای این که این موضوع را ببینید به نمودار تابع  $y = \frac{1}{x}$  که در تصویر زیر داده شده دقت کنید. نقاط متناظر با تابع  $T(n)$  را با ستاره نشان داده‌ایم. به سادگی می‌توان دید که این نقاط یک افزای ریمانی واحد برای هر کدام از انتگرال‌های

مطرح شده تشکیل می‌دهند که برای یک کران پایین و برای دیگری یک کران بالا به دست می‌دهد. در نتیجه طبق قضیه‌ی ساندویچ  $.T(n) \in \Theta(\lg n)$



به عنوان یک مثال دیگر فرض کنید:

$$T(n) = \sum_{i=1}^n \log i$$

می‌خواهیم ثابت کنیم که  $T(n) \in \Theta(n \lg n)$ . داریم:

$$T(n) = \sum_{i=1}^n \lg i > \sum_{i=\lceil \frac{n}{2} \rceil}^n \lg i > (\frac{n}{2} - 1)(\lg n - 1)$$

از طرفی بدیهیست که  $T(n) \leq n \lg n$  و لذا  $T(n) \in \Theta(n \lg n)$

**روش تغییر متغیر** تکنیک بعدی که به آن می‌پردازیم استفاده از تغییر متغیر است. بسیار پیش می‌آید که حل یک رابطه‌ی بازگشتی با تعریف متغیرهای جدید و نوشتمنتابع بر اساس آنها آسان‌تر شود. در اینجا به دو نمونه می‌پردازیم:

$$T(n) = 2T(\sqrt{n}) + \lg n$$

پیدا کردن پیچیدگی این تابع با حدس و سپس اثبات آن چندان ساده نیست اما می‌توانیم از تغییر متغیر استفاده کنیم. فرض کنید که قرار دهیم  $n = 2^m$  یا به‌طور معادل  $m = \lg n$ . (آیا این فرض سازگار است؟ چرا؟) حال عبارت بالا را بر اساس متغیر جدید می‌نویسیم:

$$T(2^m) = 2T(2^{\frac{m}{2}}) + m$$

حال تابع جدید  $S$  را به این شکل تعریف می‌کنیم:

$$S(m) = T(2^m)$$

با این تعریف می‌توانیم بنویسیم:

$$S(m) = 2S(\frac{m}{2}) + m$$

این یک رابطه‌ی بازگشتی بسیار ساده‌تر است که می‌توانیم با تکنیک‌های قبلی حلش کنیم و به دست بیاوریم که

$T(n) \in O(m \lg m)$  پس  $S(m) = T(2^m) = T(n) \cdot S(m) \in O(m \lg m)$  اما

$$T(n) \in O(\lg n \lg \lg n)$$

و این تحلیل را تمام می‌کند. دقت کنید که همین روش برای  $\Omega$  و در نتیجه  $\Theta$  هم جواب می‌دهد. حال همین مثال را کمی تغییر می‌دهیم و سعی می‌کنیم اینتابع را تحلیل کنیم:

$$T(n) = 3T(\sqrt{n}) + \lg n$$

اگر همان تغییر متغیر قبلی را اعمال کنیم به این رابطه می‌رسیم:

$$S(m) = 3S\left(\frac{m}{2}\right) + m$$

که گرچه با روش‌هایی که تا حالا بیان شدند قابل تحلیل است اما حدس زدن جوابش چندان ساده نیست. خوشبختانه یک راه کلی برای به دست آوردن راه حل توابع بازگشتی از این فرم وجود دارد که یک بار برای همیشه حدس زدن را انجام داده و حدس را ثابت کرده و باعث می‌شود به سادگی بتوانیم جواب این رابطه را بیابیم. این راه کلی را قضیه‌ی اصلی می‌نامیم که بخش بعدی به آن می‌پردازد.

**قضیه‌ی اصلی** اگر تابعی بازگشتی به صورت زیر داشته باشیم:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

آنگاه چهار حالت ممکن است رخ دهد:

- وجود داشته باشد عدد  $0 < \epsilon$  به طوری که  $f(n) \in O(n^{\log_b^a - \epsilon})$ . یعنی  $f(n)$  به طور چند جمله‌ای از  $n^{\log_b^a}$  کمتر باشد. در این صورت خواهیم داشت:  $T(n) \in \Theta(n^{\log_b^a})$ .
- وجود داشته باشد عدد  $0 < \epsilon$  به طوری که  $f(n) \in \Omega(n^{\log_b^a + \epsilon})$ . یعنی  $f(n)$  به طور چند جمله‌ای از  $n^{\log_b^a}$  بیشتر باشد. در این صورت خواهیم داشت:  $T(n) \in \Theta(f(n))$ .
- داشته باشیم  $f(n) \in \Theta(n^{\log_b^a})$  و  $n^{\log_b^a} \in \Theta(f(n))$  یعنی  $f(n) \in \Theta(n^{\log_b^a})$  داشته باشند. در این صورت داریم  $T(n) \in \Theta(f(n) \lg(n)) = \Theta(n^{\log_b^a} \lg(n))$
- هیچ یک از سه حالت فوق رخ ندهد. در این صورت از قضیه‌ی اصلی نمی‌توان استفاده کرد و برای یافتن پیچیدگی تابع بازگشتی باید سراغ روش‌های دیگر رفت.

**الگوریتم مرتب‌سازی ادغامی** به عنوان نخستین مثال از تحلیل زمان اجرا به وسیله‌ی قضیه‌ی اصلی، الگوریتم مرتب‌سازی ادغامی را مرور می‌کنیم.

در این الگوریتم برای مرتب‌سازی یک آرایه به طول  $n$  ابتدا آن را به دو بخش تقریباً مساوی نصف می‌کنیم. سپس هر یک از این دو بخش را به صورت بازگشتی مرتب می‌کنیم. در انتهای این دو قسمت را ادغام می‌کنیم. برای این کار ابتدا به کوچک‌ترین عنصر دو لیست نگاه می‌کنیم و کمترین آن‌ها را به عنوان کمترین عنصر کل آرایه در نظر می‌گیریم و آن را از لیستی که در آن قرار داشته حذف می‌کنیم. دوباره به کمترین عنصر باقیمانده در دو لیست نگاه می‌کنیم و دومین عنصر کل آرایه را می‌یابیم. این کار را تا خالی شدن یکی از لیست‌ها انجام می‌دهیم. در نهایت کافی است انتهای لیست دیگر را به انتهای آرایه می‌آرایی مرتب شده اضافه کنیم تا کل عناصر مرتب شوند.

In [3]: `def mergesort(A):`

"""

به صورت بازگشتی (Merge Sort) پیاده‌سازی الگوریتم مرتب‌سازی ادغامی  
این الگوریتم با تقسیم آرایه به دو نیم، مرتب‌سازی هر نیم و سپس ادغام آن‌ها کار می‌کند

Args:

لیست ورودی که باید مرتب شود: A (list)

```

Returns:
    نسخه مرتب شده لیست ورودی: list
"""

حالت پایه: اگر آرایه کمتر از 2 عنصر داشته باشد، قبلاً مرتب است #
if len(A) < 2:
    return A

محاسبه اندیس میانی برای تقسیم آرایه به دو نیم #
mid = len(A) // 2

# (C) و راست (B) تقسیم آرایه به دو نیم: چپ #
نیمه چپ: از ابتداء تا میانی (بدون میانی) #
B = A[:mid] #
نیمه راست: از میانی تا انتهای آرایه #
C = A[mid:] #

برای مشاهده مراحل تقسیم، کامنت را حذف کنید # print(B, C)

مرتب سازی بازگشتی هر نیمه #
B = mergesort(B) #
C = mergesort(C) #

# ادغام دو نیمه مرتب شده
# (C) و راست (B) شمارنده ها برای بیمایش نیمه چپ #
i = j = 0
لیست خالی برای ذخیره نتیجه ادغام #
A = []

حلقه ادغام: تا زمانی که هر دو لیست عناصری داشته باشند #
while i < len(B) and j < len(C):
    مقایسه عناصر فعلی دو لیست
    if B[i] <= C[j]:
        عنصر کوچکتر را به نتیجه اضافه کن #
        A += [B[i]] #
        i += 1
        B حرکت به عنصر بعدی در لیست
    else:
        عنصر کوچکتر را به نتیجه اضافه کن #
        A += [C[j]] #
        j += 1
        C حرکت به عنصر بعدی در لیست

    اضافه کردن عناصر باقیمانده از هر لیست (اگر وجود داشته باشد) #
    يکی از این دو لیست حتماً خالی است #
    A += B[i:j] + C[j:]

برای مشاهده نتیجه ادغام در هر مرحله، کامنت را حذف کنید # print(A)

return A # بازگرداندن لیست مرتب شده

```

```
In [4]: A = [5, -1, 3, 2, -4, 2, 8, 1, 0, -7, 9, 6, 1, 4]
A = mergesort(A)
print(A)
```

[-7, -4, -1, 0, 1, 1, 2, 2, 3, 4, 5, 6, 8, 9]

اگر  $T(n)$  را زمان اجرای مرتب سازی ادغامی برای یک آرایه به طول  $n$  در نظر بگیریم در این صورت داریم:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

اگر با قضیه اصلی بخواهیم به این مساله نگاه کنیم خواهیم داشت:

$$n^{\log_b^a} = n^{\log_2^2} = n$$

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

و در نتیجه:

$$n^{\log_b^a} \in \Theta(f(n))$$

لذا طبق قضیه اصلی داریم:

$$T(n) \in \Theta(n^{\log_b^a} \lg(n)) = \Theta(n \lg(n))$$

**مثال:** تابع زیر را با استفاده از قضیه اصلی تحلیل کنید:

$$T(n) = 7T\left(\frac{n}{3}\right) + O(n)$$

در این حالت  $n > n^{\log_b^a - \epsilon} = n^{\log_3^7 - \epsilon} > 0 < \epsilon < \log_3^7 - 1$  برقرار است. پس خواهیم داشت:

$$T(n) \in \Theta(n^{\log_b^a}) = \Theta(n^{\log_3^7}) \approx \Theta(n^{1.77})$$

**مثال:** تابع زیر را با استفاده از قضیه‌ی اصلی تحلیل کنید:

$$T(n) = 8T\left(\frac{n}{2}\right) + 25 \times n^3$$

در این حالت  $T(n) \in \Theta(n^{\log_b^a} \lg(n)) = \Theta(n^{\log_2^8} \lg(n)) = \Theta(n^3 \lg(n))$

**مثال:** تابع زیر را با استفاده از قضیه‌ی اصلی تحلیل کنید:

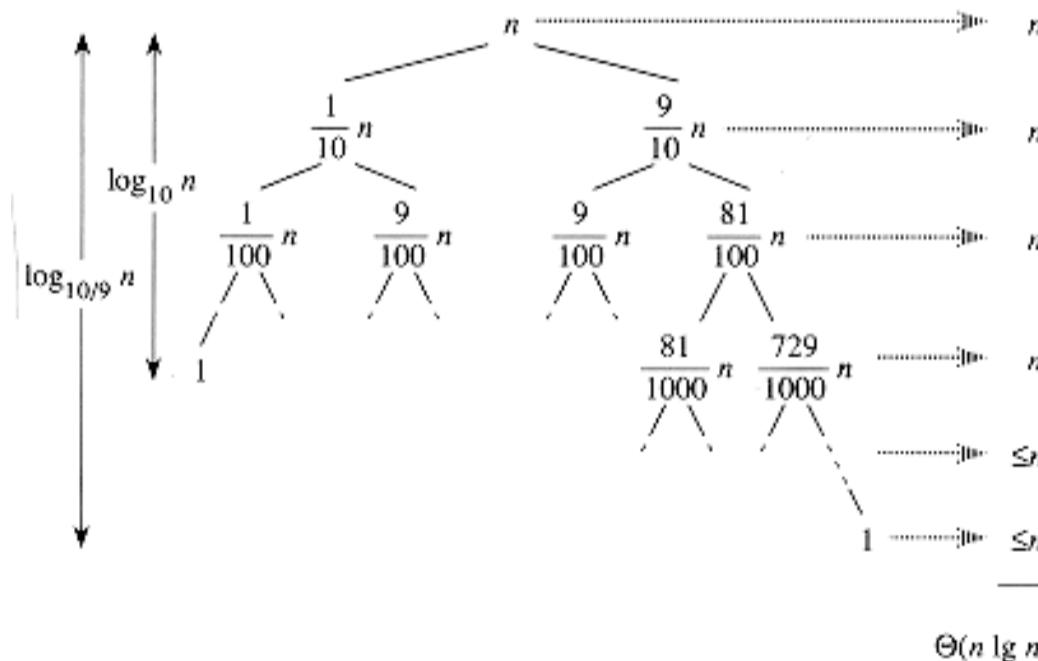
$$T(n) = 4T\left(\frac{n}{2}\right) + O(n^2 \lg(n))$$

با وجود آن که  $n^{\log_b^a} = n^{\log_2^4} = n^2 \in O(n^2 \lg(n))$  ولی هیچ مقدار  $\epsilon > 0$  وجود ندارد که به ازای آن  $n^{2+\epsilon} \in O(n^2 \lg(n))$  چرا که هر چقدر هم مقدار کمتری برای  $\epsilon$  انتخاب کنیم باز هم مقداری از  $n_0$  وجود خواهد داشت که برای  $n > n_0$  داشته باشیم  $\lg(n) > n^\epsilon$ . به همین دلیل هیچ یک از دو تابع  $n^2$  و  $n^2 \lg(n)$  به طور چند جمله‌ای بیشتر از تابع دیگر نیست، و این دو تابع روند رشد یکسانی هم ندارند، لذا نمی‌توانیم از قضیه‌ی اصلی استفاده کنیم.

**روش درخت بازگشت** بعضی اوقات با رسم یک درخت که روند تابع بازگشتی را در هر سطح نشان دهد می‌توان به سادگی توابع بازگشتی را تحلیل کرد. مثلاً فرض کنید بخواهیم تابع زیر را تحلیل کنیم:

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + n & n \geq 2 \end{cases}$$

می‌توانیم روند اجرای این رویه‌ی بازگشتی را به صورت یک درخت در نظر بگیریم که در ریشه‌ی آن مقدار  $n$  قرار دارد. هزینه‌ی  $T(n)$  به صورت بازگشتی مجموع هزینه‌های  $T\left(\frac{n}{10}\right)$  و  $T\left(\frac{9n}{10}\right)$  است، لذا می‌توانیم برای ریشه دو راس با اندازه‌های  $\frac{n}{10}$  و  $\frac{9n}{10}$  در نظر بگیریم. علاوه بر این هزینه‌های بازگشتی، تابع  $T(n)$  یک هزینه‌ی  $n$  هم دارد که می‌توانیم آن را به صورت جداگانه در سطح ریشه محاسبه کنیم. با این تفاسیر به شکل زیر می‌رسیم:



توجه کنید در هر سطح از سطوح اولیه، چون مجموع اعداد نوشته شده روی راس‌ها  $n$

منبع شکل: <http://staff.ustc.edu.cn/~csli/>

است پس مجموع هزینه‌ی هر سطح از درخت  $n$  است.

این درخت نامتوازن است و عمق برگ‌های آن بین  $\log_{10}^n$  تا  $\log_{10}^n$  متغیر است. لذا هزینه‌ی کل این تابع بین  $n \log_{10}^n$  و  $n \log_{10}^n \lg(n)$  است که هر دو مقدار از  $\Theta(n \lg(n))$  هستند. یعنی این تابع از  $\Theta(n \lg(n))$  است.

حل چند مثال دیگر مثال: ثابت کنید:

$$1^c + 2^c + \dots + n^c = \Theta(n^{c+1})$$

می دانیم

$$1^c + 2^c + \dots + n^c \geq \frac{n}{2} \left(\frac{n}{2}\right)^c = \frac{n^{c+1}}{2^{c+1}}$$

در نتیجه

$$1^c + 2^c + \dots + n^c = \Omega(n^{c+1})$$

همچنین داریم

$$1^c + 2^c + \dots + n^c \leq n(n^c) = n^{c+1}$$

پس

$$1^c + 2^c + \dots + n^c = O(n^{c+1})$$

از دو نتیجه به دست آمده در بالا می توان نتیجه گرفت

$$1^c + 2^c + \dots + n^c = \Theta(n^{c+1})$$

مثال: حال روش جدیدی برای اثبات

$$\sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$$

می توان این سری را به صورت زیر نمایش داد

$$1 + \left(\frac{1}{2} + \frac{1}{3}\right) + \left(\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}\right) + \left(\frac{1}{8} + \dots + \frac{1}{15}\right) + \dots$$

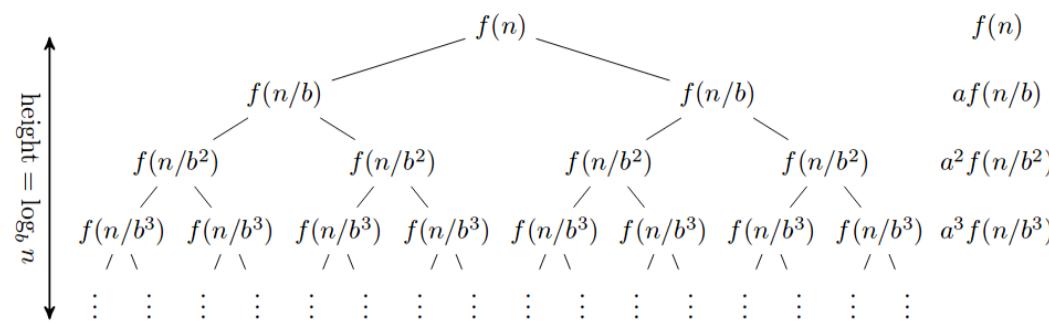
در این نوع دسته‌بندی، مجموع اعداد هر دسته از 1 کمتر است و از  $\frac{1}{2}$  بیشتر است و چون  $\log n$  تا دسته داریم پس

$$\sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$$

برای مطالعه بیشتر ۱: اثبات قضیه‌ی اصلی رابطه‌ی بازگشتی زیر را در نظر بگیرید:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

برای اینکه شهود بیشتر نسبت به این رابطه پیدا کنیم می‌توانیم از نمایش به صورت درخت بازگشت برای حالتی که  $a = 2$  است



تعداد راس‌های درخت از مرتبه‌ی  $O(a \log_b n) = O(n^{\log_b a})$  است. درنتیجه مجموع هزینه‌ی ثابت راس‌ها نیز از مرتبه‌ی  $O(n^{\log_b a})$  است.

پس می‌توانیم  $T(n)$  را به صورت زیر بنویسیم:

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right) + O(n^{\log_b a})$$

سعی کنید با جایگذاری  $f(n)$  در رابطه‌ی بالا حالت‌های مختلف قضیه‌ی اصلی را ثابت کنید.  
نکته: اگر لازم داشتید می‌توانید از فرض مشتق‌پذیر بودن  $f(n)$  در همه‌ی نقاط و تا هر درجه‌ی دلخواه استفاده کنید.

**برای مطالعه بیشتر ۲: یک مثال ابتکاری** در اینجا مثالی می‌زنیم که حل آن از هیچ یک از روش‌های گفته شده به جز حدس و استقرا مقدور نیست.  
تابع بازگشتی زیر را در نظر بگیرید:

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

می‌خواهیم اثبات کنیم که

$$T(n) \in \Theta\left(\frac{\lg(n)}{n}\right)$$

الف) اثبات می‌کنیم که

$$T(n) \in O\left(\frac{\lg(n)}{n}\right)$$

$$nT(n) = \left[ \frac{1}{n} + (n-1)T(n-1) \right] \frac{n^2}{n^2+1}$$

و  $g(n)$  را به این صورت تعریف می‌کنیم:  $g(n) = nT(n)$ . بنابراین:

$$g(n) = \frac{n^2}{n^2+1} \left[ \frac{1}{n} + g(n-1) \right]$$

$$g(n) < \frac{1}{n} + g(n-1)$$

يعنى

$$g(n) < 1 + \frac{1}{2} + \dots + \frac{1}{n}$$

$$h_1(n) = 1 + \frac{1}{2} + \dots + \frac{1}{n} = \Theta(\lg(n))$$

(چرا  $h_1(n) = \Theta(\lg(n))$ )

بنابراین:

$$T(n) \in O\left(\frac{\lg(n)}{n}\right)$$

ب) اثبات میکنیم که:

$$T(n) \in \Omega\left(\frac{\lg(n)}{n}\right)$$

$$\frac{n^2}{n+1} T(n) = \frac{n^4}{n^4 - 1} \left[ \frac{n-1}{n^2} + \frac{(n-1)^2}{n} T(n-1) \right]$$

تعريف میکنیم:

$$g(n) = \frac{n^2}{n+1} T(n)$$

به روابط زیر میرسیم:

$$g(n) = \frac{n^4}{n^4 - 1} \left[ \frac{n-1}{n^2} + g(n-1) \right]$$

$$> \frac{1}{n} - \frac{1}{n^2} + g(n-1)$$

$$> 1 + \frac{1}{2} + \dots + \frac{1}{n} - \left( 1 + \frac{1}{4} + \dots + \frac{1}{n^2} \right)$$

اگر رابطه‌ی  $h_2(n) < c$  را تعريف کنیم و برای یک ثابت  $c$  فرض کنیم که (چرا این فرض درست است؟) در آن صورت  $g(n) > h_1(n) - c$  و در نتیجه:

$$T(n) > \left[ \frac{1}{n} + \frac{1}{n^2} \right] (h_1(n) - c)$$

$$T(n) > \frac{h_1(n) - c}{n}$$

$$f(n) = \Omega\left(\frac{\lg(n)}{n}\right)$$

طبق «الف» و «ب» نتیجه می‌شود:  $T(n) = \Theta\left(\frac{\lg(n)}{n}\right)$

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل سوم، بخش سوم: تحلیل سرشکن

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

• مقدمه

• شمارنده دودویی

• تحلیل سرشکن

• پیاده سازی پشته با استفاده از آرایه

• روش حسابداری

• روش تابع پتانسیل

• چند مثال

## مقدمه

در بخش قبلی با تحلیل الگوریتم‌ها در بدترین حالت آشنا شدیم، در واقع یادگرفتیم چگونه برای هزینه‌ی اعمال در بدترین حالت کران بالا پیدا کنیم. گاهی ممکن است با داده‌ساختاری مواجه شویم که هزینه‌ی یک عمل در آن بعضی موقع بسیار بیشتر از موقع دیگر است! شمارنده‌ی دودویی یکی از این داده‌ساختار هاست.

## شمارنده دودویی:

فرض کنید یک شمارنده دودویی  $n$  بیتی داریم که درابتدا عدد 0 را نشان می‌دهد. در هر مرحله عددی که این نمایشگر نشان می‌دهد بک واحد زیاد می‌کنیم. فرض کنید برای تغییر هر بیت از این شمارنده باید یک تومان پول بپردازیم.

برای مثال در زیر هزینه مراحل یک شمارنده دودویی  $3$  بیتی نشان داده شده است:

مرحله	بیت ۱	بیت ۲	بیت ۳	هزینه
صفر	۰	۰	۰	۰
یک	۰	۰	۰	۱
دو	۰	۱	۰	۲
سه	۱	۰	۱	۱
چهار	۰	۰	۱	۳
پنج	۱	۱	۰	۱

هزینه‌ی اضافه کردن عدد شمارنده در هر مرحله دست بالا از  $O(n)$  است. مثلاً حالتی که شمارنده عدد  $1 - 2^n$  را نشان می‌دهد در نظر بگیرید، برای زیاد کردن عدد شمارنده باید مقدار  $n$  بیت را عوض کنیم. با این حساب اگر بخواهیم تا  $m$  بشماریم هزینه‌ی کل از مرتبه‌ی  $O(n \times m)$  خواهد بود، اما در واقعیت هزینه‌ی بسیار کمتر است!

In [2]:

```
# نمایش نمودارها مستقیماً در خروجی Jupyter
%matplotlib inline
from matplotlib.pyplot import * # وارد کردن تمام توابع رسم نمودار از Matplotlib
rcParams.update({'font.size': 25, 'font.family': 'serif', 'lines.linewidth': 3}) # (تعداد تغییر بیت‌ها) را برمی‌گرداند
```

تنظیمات پیش‌فرض برای نمودارها (اندازه فونت، ح انواده فونت، ضخامت خط) را برمی‌گرداند.

In [3]:

```
def increase(counter):
    """
    این تابع یک شمارنده دودویی را یک واحد افزایش می‌دهد و هزینه (تعداد تغییر بیت‌ها) را برمی‌گرداند.

    Args:
        counter (list): لیستی از 0 و 1 که نمایش دودویی یک عدد را نشان می‌دهد (مثلًا [0, 0, 1] برای 1).
        در نظر گرفته می‌شوند (تا n-1) بیت‌ها از راست به چپ.

    Returns:
        int: هزینه افزایش (تعداد بیت‌های تغییر یافته).
        """
    n = len(counter) # طول شمارنده (تعداد بیت‌ها)
    cost = 0 # هزینه تغییر بیت‌ها در این مرحله

    حلقه معکوس: از کم‌ازیش‌ترین بیت (راست‌ترین) شروع می‌کنیم
    for i in range(n - 1, -1, -1):
        هر بار که بیتی را بررسی می‌کنیم/تغییر می‌دهیم، هزینه 1 واحد افزایش می‌باید
        if counter[i] == 1:
            counter[i] = 0 # اگر بیت 1 بود، آن را به 0 تغییر می‌دهیم (حمل)
        else: # counter[i] == 0
            counter[i] = 1 # اگر بیت 0 بود، آن را به 1 تغییر می‌دهیم
            break # عملیات افزایش تمام شده است، از حلقه خارج می‌شویم
    return cost

def simulate(n, m):
    """
    بار افزایش انجام می‌دهد m بیتی را برای n این تابع شبیه‌سازی یک شمارنده دودویی
    هزینه هر افزایش، هزینه کل و هزینه متوسط را محاسبه و چاپ می‌کند.

    Args:
        n (int): طول شمارنده (تعداد بیت‌ها).
        m (int): تعداد دفعات افزایش شمارنده.
    """
    counter = [0] * n # بیتی که با صفر مقداردهی اولیه می‌شود n شمارنده دودویی
    cost = [0] * (m + 1) # امر: هزینه افزایش = لیست برای نگهداری هزینه هر افزایش
    total_cost = [0] * (m + 1) # امر: لیست برای نگهداری هزینه کل تا مرحله
    average = [0] * (m + 1) # امر: لیست برای نگهداری هزینه متوسط تا مرحله

    # هزینه کل قبل از اولین افزایش)
    # بار افزایش شمارنده m حلقه برای
    for i in range(1, m + 1):
        امر: محاسبه هزینه افزایش # cost[i] = increase(counter)
        محاسبه هزینه کل تجمعی # total_cost[i] = cost[i] + total_cost[i - 1]
        محاسبه هزینه متوسط تجمعی # average[i] = (total_cost[i] * 1.0) / i

        خطوط زیر برای نمایش جزئیات مرحله به مرحله است (در حال حاضر کامنت شده‌اند)
        # print("Step ", i, "\tCounter ", counter)
        # print("\tCost ", cost[i], "\tTotal ", total_cost[i])
        # print ("\tAverage ",average[i])

    چاپ لیست‌های هزینه کل و هزینه متوسط پس از شبیه‌سازی کامل
    print(total_cost)
    print(average)

    برای رسم نمودارها (این تابع باید جداگانه تعریف شده باشد) plot_cost فراخوانی تابع
    در این کد تعریف نشده است و برای اجرا نیاز به تعریف آن دارد. plot_cost توجه: تابع
    # plot_cost(cost, total_cost , average)
```

حال یک تابع برای رسم نمودار هزینه‌ی هر بار افزایش شمارنده (تعداد بیت‌هایی که باید در هر گام تغییر کنند)

In [4]:

```
def plot_cost(cost, total_cost, average):
    """
    این تابع نمودار هزینه‌های شبیه‌سازی شمارنده دودویی را رسم می‌کند.
    شامل هزینه هر مرحله، هزینه کل تجمعی و هزینه متوسط تجمعی

    Args:
        cost (list): لیست هزینه‌های هر مرحله.
        total_cost (list): لیست هزینه‌های کل تجمعی.
        average (list): لیست هزینه‌های متوسط تجمعی.
    """
    figure(figsize=(20, 10)) # ایجاد یک شکل (figure) با اندازه مشخص
    xlabel("Step") # X برجسب محور
    ylabel("Step cost (blue), Total cost (green), Average Cost (red)") # Y برجسب محور

    x = range(len(total_cost)) # مراحل X مقادیر محور

    bar(x, cost, label='Step Cost') # رسم نمودار میله‌ای برای هزینه هر مرحله
    plot(x, total_cost, 'g', label='Total Cost') # رسم نمودار خطی برای هزینه کل تجمعی
    plot(x, average, 'r', label='Average Cost') # رسم نمودار خطی برای هزینه متوسط تجمعی

    legend(loc='upper left') # نمایش راهنمای
    show() # نمایش نمودار
```

In [5]:

```
simulate(4, 2**4)
```

```
[0, 1, 3, 4, 7, 8, 10, 11, 15, 16, 18, 19, 22, 23, 25, 26, 30]
[0, 1.0, 1.5, 1.3333333333333333, 1.75, 1.6, 1.6666666666666667, 1.5714285714285714, 1.875, 1.7777777777777777, 1.8, 1.72727272727273, 1.8333333333333333,
1.7692307692307692, 1.7857142857142858, 1.7333333333333334, 1.875]
```

```
In [6]: simulate(5, 2**5)
```

```
[0, 1, 3, 4, 7, 8, 10, 11, 15, 16, 18, 19, 22, 23, 25, 26, 31, 32, 34, 35, 38, 39, 41, 42, 46, 47, 49, 50, 53, 54, 56, 57, 62]
[0, 1.0, 1.5, 1.3333333333333333, 1.75, 1.6, 1.6666666666666667, 1.5714285714285714, 1.875, 1.7777777777777777, 1.8, 1.72727272727273, 1.8333333333333333,
1.7692307692307692, 1.7857142857142858, 1.7333333333333334, 1.9375, 1.8823529411764706, 1.8888888888888888, 1.8421052631578947, 1.9, 1.8571428571428572, 1.8
6363636363635, 1.826086956521739, 1.9166666666666667, 1.88, 1.8846153846153846, 1.8518518518519, 1.8928571428571428, 1.8620689655172413, 1.8666666666666667
6667, 1.8387096774193548, 1.9375]
```

```
In [7]: print()
```

نکته‌ی جالب اینست که قبل از هر عمل پر هزینه در شمارنده دودویی چندین عمل کم هزینه‌تر داریم. در واقع به ازای هر  $2^{i-1}$  مرحله یکبار بیت  $-i$ -ام را تغییر می‌دهیم! پس برای شمردن از ۰ تا  $m$  باید هزینه بپردازیم که بسیار کمتر از  $n \times m \leq \sum_{i=1}^{\lfloor \lg m \rfloor} \frac{m}{2^i}$  است.

بگذارید با یک دید دیگر هم به این نتیجه برسیم که واقعاً هزینه کل از  $2m$  کمتر است: هزینه‌هایی که در هر مرحله می‌دهیم برابر است با تعداد ۱ هایی که ۰ می‌شوند به علاوه تعداد ۰ هایی که ۱ می‌شوند. کمی بررسی کنید که در هر مرحله چه بیت‌هایی ۰ و چه بیت‌هایی ۱ می‌شوند. می‌توانید حکم کلی‌ای راجع به تعداد ۱ هایی که در هر مرحله ۰ می‌شوند بدهید؟ راجع به تعداد ۰ هایی که ۱ می‌شوند چطور؟ می‌توانیم نشان دهیم که تعداد ۰ هایی که در هر مرحله ۱ می‌شوند دقیقاً برابر با ۱ است. سعی کنید این ادعا را ثابت کنید.

از طرفی تعداد دفعاتی که یک بیت ۱ می‌شود از تعداد دفعاتی که آن بیت ۱ می‌شود کوچکتر مساوی است. بنابراین هزینه کلی که برای ۱ کردن می‌گذاریم نمی‌تواند بیشتر از هزینه کلی باشد که برای ۱ کردن می‌گذاریم. و ادعا کردیم در  $m$  مرحله دقیقاً  $m$  بار عمل ۱ کردن را انجام می‌دهیم. پس در کل حداقل  $2m$  عمل تغییر بیت را انجام می‌دهیم. اگر کران بالایی که بدون تحلیل کردن و صرفاً با ضرب کردن تعداد مراحل در حداقل اعمال هر مرحله به دست آمد را با کران بالایی که با تحلیل به دست آورده م مقایسه بکنیم متوجه تفاوت زیادی می‌شویم. برای ۳۲ بار افزایش یک شمارنده ۵ بیتی مجموع هزینه ۶۲ است، در حالی که  $.2m = 64$  و  $n \times m = 5 \times 32 = 160$  است.

## تحلیل سرشکن

شمارنده دودویی مثالی از داده‌ساختارهایی بود که هزینه‌ی یک عمل در آن بعضی موقعیت‌ها بسیار بیشتر از موقعیت‌های دیگر است، همچنین دیدیم که قبل از هر عمل پرهزینه باید تعداد زیادی عمل کم‌هزینه‌تر انجام شود. در رویارویی با داده‌ساختارهای شبیه به شمارنده دودویی تحلیل بدترین حالت به یک کران بالای درست ولی بدینانه می‌انجامد.

تحلیل سرشکن نوعی از تحلیل برای موقعي است که دنباله‌ای از اعمال مشابه انجام می‌شود. ایده‌ی تحلیل سرشکن در نظر گرفتن هزینه‌ی میانگین دنباله‌ای از اعمال در بدترین حالت به جای در نظر گرفتن هزینه‌ی یک عمل در بدترین حالت است. سودمندی این نوع تحلیل زمانی مشخص می‌شود که قبل از انجام هر عمل پرهزینه تعداد قابل توجهی عمل کم‌هزینه‌تر انجام می‌شود. مثلاً در شمارنده دودویی هزینه‌ی میانگین هر عمل در بدترین حالت برابر با  $\frac{2m}{m} = O(1)$  است در حالی که هزینه‌ی یک عمل در بدترین حالت از مرتبه  $O(nm)$  است.

در زیر سعی می‌کنیم چند روش مختلف برای تحلیل سرشکن معرفی کنیم. تحلیل بالا در واقع مثالی از استفاده از این روش‌ها بود. اما دقیق‌تر کردن ایده‌ها و نام‌گذاری روی آن‌ها کمک می‌کند بتوانیم در موقعیت مشابه از این ایده‌ها بهتر استفاده کنیم. یک روش که اصطلاحاً روش انبوهه هم گفته می‌شود و در تحلیل شمارنده هم استفاده شد ضمناً بیان یک مثال توضیح می‌دهیم.

## مثال : پیاده سازی پشته با استفاده از آرایه

می‌خواهیم یک پشته را با استفاده از آرایه پیاده سازی کنیم. فرض کنید یک آرایه به نام  $A$  و یک متغیر به نام  $top$  داریم که اندیس اولین خانه‌ی خالی  $A$  است.

برای پیاده سازی تابع اضافه کردن عدد  $x$  به پشته ( $push$ ) کافی است عملیات زیر را انجام دهیم:

```
In [8]: def push(x):
```

```
    """
        را به بالای پشته اضافه می‌کند (x) این تابع یک عنصر در دسترس هستند (اندیس اولین خانه خالی) top و متغیر A فرض بر این است که آرایه
    """
    قرار می‌دهد (top) را در خانه خالی فعلی پشته x عنصر # x
    top = top + 1
    A[top] = x
```

برای پیاده سازی تابع حذف آخرین عدد پشته و برگرداندن آن ( $pop$ ) کافی است عملیات زیر را انجام دهیم:

```
In [9]: def pop():
```

```
    """
        این تابع یک عنصر را از بالای پشته حذف کرده و آن را بر می‌گرداند در دسترس هستند (اندیس اولین خانه خالی) top و متغیر A فرض بر این است که آرایه
    """
    را یک واحد کاهش می‌دهد تا به آخرین عنصر اضافه شده اشاره کند top اندیس # 1 - top = top + 1
```

```
x = A[top] # عنصر موجود در بالای پشته را برمی‌دارد.  
return x # عنصر برداشته شده را برمی‌گرداند.
```

(در کد بالا فرض کردیم که top مخالف 0 باشد)

## مثال : پیاده سازی پشته با استفاده از آرایه

می‌خواهیم یک پشته را با استفاده از آرایه پیاده سازی کنیم. فرض کنید یک آرایه به نام A و یک متغیر به نام top داریم که اندیس اولین خانه‌ی خالی A است.

برای پیاده سازی تابع اضافه کردن عدد x به پشته (push) کافی است عملیات زیر را انجام دهیم:

با توجه به محدود بودن طول آرایه، اگر آرایه هنگام اضافه کردن عضو جدید به پشته پر شده باشد باید چه کنیم؟ باید یک آرایه با طولی بزرگتر از طول آرایه‌ی فعلی تعریف کنیم، اعضای آرایه‌ی قبلی را به آرایه‌ی جدید منتقل کنیم و کار را در آن ادامه دهیم.

این عملیات هزینه‌ی زیادی به دنبال دارد، در نتیجه عملیات اضافه کردنی که منجر به این اتفاق می‌شوند زمان‌گیر خواهد بود، ولی با توجه به هزینه‌ی کم بسیاری از عملیات که ما را به این مرحله رسانده‌اند، اگر هزینه‌ی سرشکن برای هر عملیات را حساب کنیم، شاید هزینه‌ی بسیار کمتری شود. فرض می‌کنیم که هزینه‌ی انتقال یک آرایه به طول n به آرایه‌ی جدید برابر n است.

اگر هنگام تعریف آرایه‌ی جدید، طول آن را یکی بیش از طول آرایه‌ی قبلی قرار دهیم، هزینه‌ی سرشکن کل عملیات معقول می‌شود؟\*\*

حالت اول: افزایش خطی ظرفیت\*\*

خیر. اگر n عملیات اضافه کردن متوالی را در نظر بگیریم، هزینه‌ی کپی کردن اعداد به ترتیب برابر ۱، ۲، ...، ۱ – n خواهد شد، پس کل هزینه برابر  $\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = O(n^2)$  است و به صورت سرشکن هر عملیات  $O(n)$  هزینه دارد.

اگر هنگام تعریف آرایه‌ی جدید، طول آن را دو برابر طول آرایه‌ی قبلی قرار دهیم، هزینه‌ی سرشکن کل عملیات چه می‌شود؟\*\*

حالت دوم: دو برابر کردن ظرفیت\*\*

در این حالت اگر n عملیات داشته باشیم، ۱ + ۲ + ۴ + ... +  $2^{i+1}$  عملیات کپی کردن خواهیم داشت که  $1 - 2^i$  (اگر همهٔ عملیات از نوع اضافه کردن باشند،  $2^i$  بزرگترین عددی است که  $2^i$  کمتر از n است) در نتیجه این تعداد کمتر از  $2n$  عملیات است. اگر هزینه‌ی n عملیات دیگر (push‌های عادی) را هم در نظر بگیریم، برای هر عملیات هزینه‌ی سرشکن  $O(1)$  بدست می‌آید.

در حالت کلی در روش انبوهه، حساب می‌کنیم که هر نوع عمل را در کل فرآیند الگوریتم حداکثر چند بار انجام می‌دهیم. همانطور که اگر برگردید می‌بینید سوال شمارنده را هم یک بار با این روش حل کردیم.

## روش حسابداری

یک راه دیگر برای حساب کردن هزینه‌ی سرشکن عملیات، روش حسابداری می‌باشد. برای مثال، مساله‌ی قبل را دوباره در نظر بگیرید. به ازای هر عملیات اضافه کردن (push)، ۱ تومان هزینه برای انجام دادن آن می‌پردازیم و ۲ تومان را در یک حساب فرضی می‌ریزیم. پس برای هر عملیات ۳ تومان هزینه می‌کنیم. هر وقت که مجبور به دو برابر کردن طول آرایه شدیم، از پولی که در حسابمان ذخیره کرده‌ایم استفاده می‌کنیم.

\*\*چرا هیچ وقت پول کم نمی‌آوریم؟\*\*

فرض کنید طول آرایه‌ای که پر شده است و قصد انتقال آن را داریم برابر L است. پس اگر L تومان پول در بانک داشته باشیم می‌توانیم هزینه‌ی انتقال آرایه را بپردازیم. حال می‌دانستیم که هر بار پس از پر شدن، ظرفیت را ۲ برابر می‌کنیم. پس آخرین باری که ظرفیت پر شده بود و آن را ۲ برابر کرده بودیم، مقدارش  $2/L$  بود و اکنون  $2/L$  عنصر جدید اضافه کرده‌ایم (از زمان آخرین دو برابر شدن).

برای هر یک از این  $L/2$  عنصر جدید، ما ۲ تومان در حسابمان ذخیره کرده‌ایم. پس در مجموع  $L = 2(L/2)$  تومان پول در حسابمان داریم.

پس اگر برای هر عملیات ۳ تومان هزینه کنیم، همواره می‌توانیم همهٔ هزینه‌ها را پرداخت کنیم. در نتیجه هزینه‌ی سرشکن هر عملیات برابر ۳ است.

دقت کنید که در محاسبه هزینه‌ی سرشکن، هزینه‌ای که بدست می‌آوریم به ازای هر ترتیبی از عملیات داده شده همچنان حداکثر میانگین هزینه‌ی عملیات است و برخلاف حساب کردن مرتبه زمانی متوسط، ما راجع به ورودی فرضی مانند تصادفی بودن آن‌ها نکردیم، فقط میانگین هزینه‌ی عملیات در طول زمان را حساب کردیم.

\*\*به نظر شما با استفاده از روش حسابداری چگونه می‌توان مساله‌ی شمارنده دودویی را حل کرد؟\*\*

## تحلیل شمارنده‌ی دودویی به روش حسابداری

برای تغییر هر بیت  $\circ$  به  $1$ ،  $1$  تومان هزینه می‌کنیم و  $1$  تومان هم پس انداز می‌کنیم. برای راحتی کار فرض کنید که هر بیت حساب جداگانه‌ای برای خودش دارد، در نتیجه هر گاه که می‌خواهیم یک بیت  $1$  را به  $\circ$  تغییر دهیم، چون قبلًاً این بیت از  $\circ$  به  $1$  تبدیل شده است  $1$  تومان در حسابش دارد، پس از همان پول برای تغییر این بیت استفاده می‌کنیم. هر مرحله می‌تواند تعدادی تغییر بیت از  $1$  به  $\circ$  داشته باشد، ولی دقیقاً یک بیت از  $\circ$  به  $1$  تغییر پیدا می‌کند، در نتیجه هزینه‌ای که برای هر مرحله می‌پردازیم برابر  $2$  تومان است که معادل هزینه سرشکن هر مرحله می‌باشد.

## روش تابع پتانسیل

روش تابع پتانسیل در واقع همان روش حسابداری است. صرفاً بیان آن ریاضی است. در واقع ایده اینست که یک تابع روی مراحل الگوریتم تعریف می‌کنیم که مقدار پول ذخیره شده در حساب را نشان می‌دهد. سپس خواص که لازم است این تابع داشته باشد که بتواند واقعاً مقدار پول ذخیره شده در حساب را نشان دهد را از نظر ریاضی به طور دقیق بیان می‌کنیم. بعد با استفاده از تابعی که تعریف می‌کنیم سوال حل می‌کنیم.

فرض کنیم میزان پول که در مرحله  $i$  در حساب داریم را با  $\Phi(i)$  نشان دهیم. یعنی این تابع یک تابع از مراحل به اعداد حقیقی است. به طور طبیعی و شهودی اگر بخواهیم این تابع را مدل کنیم خواص زیر را دارد:

برای هر مرحله  $i$ :

هزینه‌ای که به حساب واریز می‌شود  $= \hat{c}_i$  که  $\hat{c}_i$  هزینه سرشکن است و  $c_i$  هزینه‌ای است که واقعاً در این مرحله استفاده می‌شود.

چیزی که در نهایت می‌خواستیم به آن برسیم این است که این شرطی که به طور طبیعی قراردادیم، برای نشان دادن اینکه مجموع  $\hat{c}_i$  ها از مجموع  $c_i$  ها بیشتر اند، کافی است. چون در این صورت اگر ما مجموع  $\hat{c}_i$  ها را حساب کنیم یک کران بالا برای هزینه کل که مجموع  $c_i$  ها باشد به دست آورده‌ایم.

$\sum_{i=1}^m \hat{c}_i = \sum_{i=1}^m (c_i + \Phi(i) - \Phi(i-1)) = \sum_{i=1}^m c_i + \Phi(m) - \Phi(0)$  با استفاده از شرط بالا داریم:

بنابراین ثابت شد که اگر یک تابع با خواص بالا روی مراحل تعریف کنیم مجموع  $\hat{c}_i$  ها یک کران بالا برای هزینه کل است.

بایاید در مثال شمارنده باینری از این تابع استفاده کنیم: فرض کنید تعداد  $1$  های موجود در عدد تولید شده در هر مرحله را تابع پتانسیل تعریف کنیم. می‌توان چک کرد که این تابع خواص اول و دوم را دارد. یعنی در مرحله صفرم صفر است و در هر مرحله یک عدد نامنفی است. در هر مرحله فرض کنید  $k$  بیت از  $1$  به صفر تغییر می‌کنند و  $1$  بیت از صفر به یک تغییر می‌کنند. بنابراین  $1 + c_i = k$ . حالا  $\hat{c}_i$  را طوری پیدا می‌کنیم که شرط سوم برقرار باشد:

براساس تعریف تابع در این سوال:  $\hat{c}_i = \Phi(i) - \Phi(i-1)$  دادعت  $= \Phi(i)$

بر اساس شرط تابع:  $\hat{c}_i = c_i + \Phi(i) - \Phi(i-1)$

پس نتیجه می‌گیریم:  $(\hat{c}_i = c_i + \Phi(i) - \Phi(i-1)) = (\Phi(i) - \Phi(i-1)) + (c_i - \Phi(i-1))$

در هر مرحله  $k$  بیت از  $1$  به  $0$  تغییر می‌کنند و  $1$  بیت از  $0$  به  $1$  تغییر می‌کند.

$\Phi(i) = \Phi(i-1) - k + 1$  پس

در نتیجه

$$\hat{c}_i = (k+1) + (\Phi(i-1) - k + 1) - \Phi(i-1) = k + 1 - k + 1 = 2$$

پس هزینه سرشکن هر عملیات  $2$  است.

## مثال : پشته با حافظه‌ی بهینه

در مسئله پیاده سازی پشته، طول آرایه را  $L$  و تعداد اعضای پشته را  $N$  در نظر بگیرید، می‌خواهیم  $L$  همواره از مرتبه  $N$  باشد.

در الگوریتم فعلی اگر ۱۰۰۰ عملیات اضافه کردن و سپس ۹۹۹ عملیات حذف کردن را انجام دهیم، آرایه‌ای به طول ۱۰۲۴ خواهیم داشت (چرا؟)، که فقط ۱ خانه از آن استفاده شده است و مقدار زیادی حافظه بی‌استفاده گرفته شده.

راهکاری برای درست کردن این مشکل ارائه دهید و سپس هزینه سرشکن عملیات را محاسبه کنید.

## مثال : شمارنده پر هزینه!

با استفاده از روش انبوهه، حسابداری و تابع پتانسیل مقدار هزینه سرشکن هر عملیات افزایش را حساب کنید.

فرض کنید یک شمارنده داریم که هزینه تغییر بیت  $A$  ام آن  $A$  است.

راهنمایی: بسیار شبیه شمارنده معمولی است.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل چهارم: داده‌ساختارهای پایه‌ای

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

## فهرست

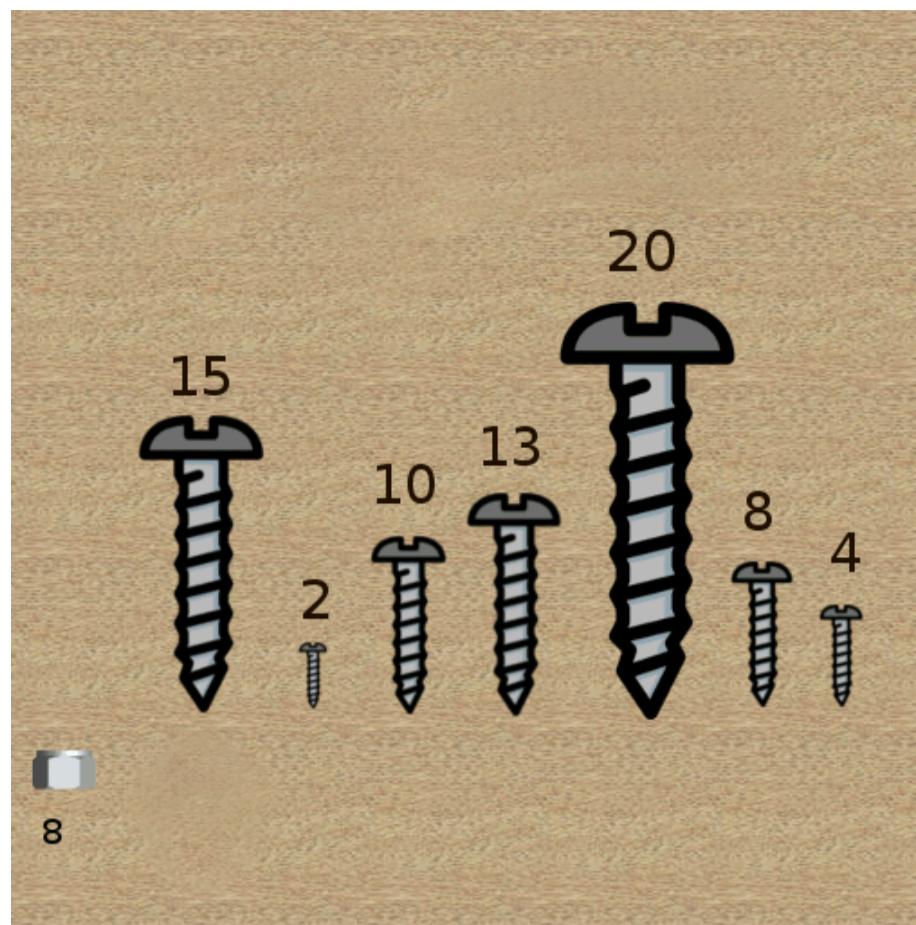
- داده‌ساختارها
- صفات
- پشتوندی
- صفات دوطرفه
- لیست پیوندی

## داده‌ساختارها

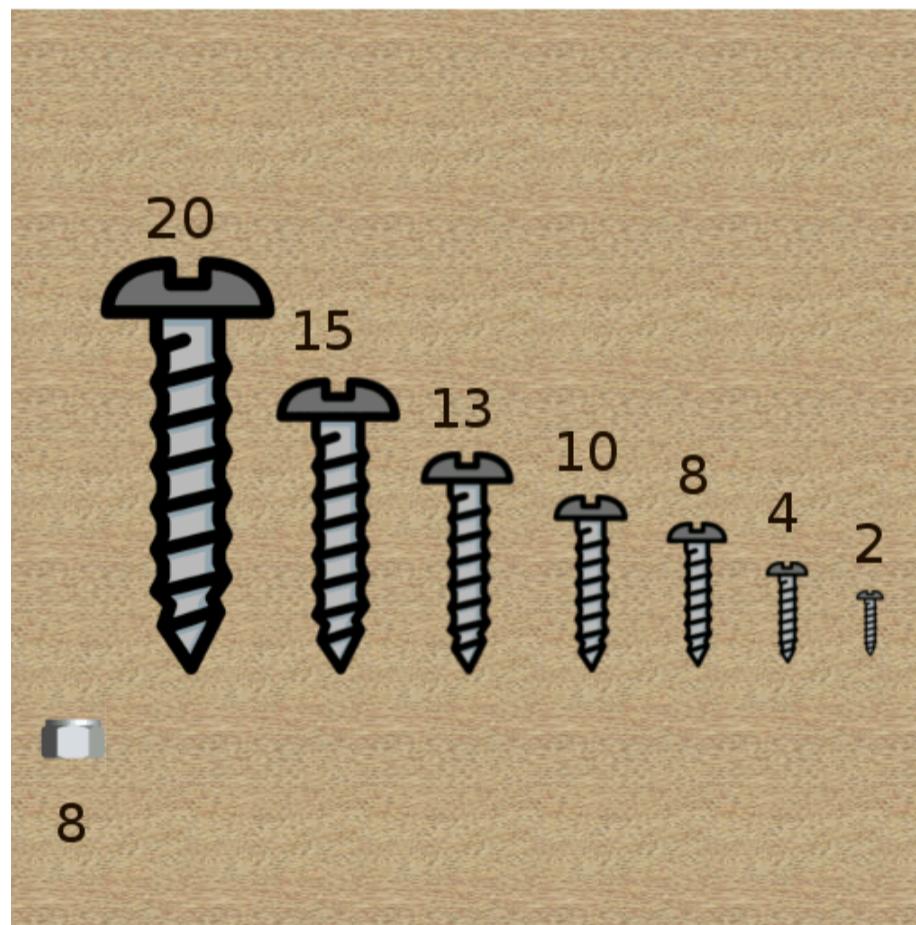
داده‌ساختارها همان‌طور که از اسم آنها مشخص است، روش‌ها و یا به زبانی دیگر ساختارهایی هستند که برای ذخیره‌ی ساختارمند داده‌ها استفاده می‌شوند تا در پاسخ‌دهی به سوالات ما کارا و سریع باشند و اگر چیزی که یک داده‌ساختار مدلی از آن است در حال تغییر سریع باشد، باید در تطابق دادن خود با آن نیز چاپ باشند. مشخص است بسته به اینکه سوالاتی که از یک سری داده داریم چه هستند یا میزان تغییری که در داده‌ها ممکن است رخ دهد (مثلاً حذف شوند یا اضافه شوند) چقدر است (آیا اصولاً حذف و اضافه‌ای رخ می‌دهد؟) یا از چه نوعی است (مثلاً آیا تنها کوچک‌ترین عنصر حذف می‌شود یا هر عنصری ممکن است حذف شود)، باید نحوه‌ی ذخیره‌سازی آنها نیز متفاوت باشد.

ممکن است تعاریف بالا تا حد زیادی گنج به نظر برسند که برای روشن شدن آنها کافیست به مثال زیر توجه کنید. فرض کنید که آریا یک جعبه از پیچ‌ها با اندازه‌های مختلف دارد. هر بار که می‌لاد به او یک مهره می‌دهد، او می‌خواهد بداند که آیا پیچی همان‌دازه‌ی آن مهره دارد یا نه؟ آریا برای ذخیره کردن اندازه‌ی این پیچ‌ها از یک آرایه استفاده می‌کند. او برای این کار دو روش دارد:

اندازه‌ها را بدون هیچ‌گونه ترتیب خاصی در آرایه بنویسد. مزیت این روش این است که اگر آریا یک پیچ جدید بخرد می‌تواند اندازه‌ی این پیچ را به راحتی در آخر آرایه‌ی خود اضافه کند، اما هر گاه با یک مهره مواجه می‌شود باید با یک حلقه همه آرایه‌ی خود را چک کند که آیا پیچی همان‌دازه‌ی این مهره دارد یا نه.



اندازه‌ی پیچ‌ها را به صورت مرتب شده در آرایه ذخیره کند. اگر به مهره‌ی جدیدی برخورد کند می‌تواند به راحتی و با استفاده از جستجوی دودویی بسیار سریع‌تر و کاراتر از اینکه تمامی اعضای آرایه را ببیند و تنها با دیدن محدودی از اعضای آرایه بفهمد که آیا مهره‌ای همان‌اندازه‌ی آن پیچ جدید دارد یا خیر. اما نقطه ضعف این روش این است که اگر آرایه یک پیچ جدید بخرد اضافه کردن این پیچ جدید به آرایه بدون آن که ساختارمندی آن (مرتب بودن اعضا) به هم برخورد دشوار و زمانبر است.



## پشته، صفحه و لیست پیوندی

به نظر شما چگونه یک سری داده را که توالی در آنها مهم است ذخیره کنیم؟ در نظر بگیرید که داده‌ها یک به یک در حال وارد شدن به برنامه یا خارج شدن از آن هستند و باید آنها را در داده‌ساختاری ذخیره کنیم که توالی آنها را حفظ کند. در این جلسه داده‌ساختارهایی را بررسی می‌کنیم که یک توالی از عناصر را ذخیره می‌کنند و هر کدام برای نوع خاصی از اضافه شدن یا حذف شدن عناصر بهینه شده‌اند.

صف داده‌ساختاری است که عنصر جدید فقط در انتهای آن اضافه می‌شود و حذف عنصر تنها از اول آن اتفاق می‌افتد. مانند یک صف نانوایی که هر کسی وارد می‌شود به انتهای صف می‌رود و هر کسی بعد از گرفتن تان از ابتدای صف خارج می‌شود.

صف باید از دستورات زیر پشتیبانی کند:

- **enqueue(x)**: عنصر  $x$  را در انتهای صف درج می‌کند.
- **() dequeue**: اولین عنصر را حذف می‌کند و آن را باز می‌گرداند.
- **() front**: اولین عنصر صف را باز می‌گرداند.
- **() size**: تعداد عناصر موجود در صف را باز می‌گرداند.
- **() isEmpty**: خالی بودن یا نبودن صف را مشخص می‌کند.
- **() isFull**: پر بودن یا نبودن صف را مشخص می‌کند.

پیاده‌سازی صف معمولاً به کمک آرایه یا لیست پیوندی که در ادامه با آن آشنا خواهید شد انجام می‌شود.

توجه: تمامی این اعمال در  $O(1)$  انجام می‌شوند.

## پیاده‌سازی صف به کمک آرایه

برای پیاده‌سازی صف به کمک آرایه ابتدا یک آرایه و دو متغیر که پکی ابتدای صف (first) و دیگری تعداد عناصری که در صف قرار دارند (num) را نشان می‌دهد، تعریف می‌کنیم.

هنگامی که عنصر جدیدی اضافه می‌شود آن را در انتهای صف قرار می‌دهیم و سپس مقدار num را یک واحد افزایش می‌دهیم. برای حذف کردن عنصر ابتدایی صف نیز مقدار num را یک واحد کم می‌کنیم و همچنین مقدار first را یک واحد افزایش می‌دهیم. (توجه: البته در این توضیح باید در نظر داشته باشیم که از آرایه به صورت دوری استفاده می‌کنیم. برای مثال اگر سایز آرایه برابر ۵ باشد، عنصر بعد از عنصر پنجم، عنصر اول است).

In [1]:

```
class Queue:
    def __init__(self, max_size):
        # متد سازنده: صف را با ظرفیت مشخص شده مقداردهی اولیه می‌کند.
        self.max_size = max_size # حداکثر ظرفیت صف
        self.Q = [0] * max_size # آرایه داخلی برای ذخیره عناصر صف
        self.num = 0 # تعداد عناصر فعلی در صف
        self.first = 0 # آندیس اولین عنصر در صف

    def enqueue(self, item):
        # عنصر جدید را به انتهای صف اضافه می‌کند.
        if self.num >= self.max_size:
            raise Exception("Queue overflow") # خطأ در صورت بر بودن صف
        # محاسبه آندیس انتهای صف به صورت دوری (circular)
        self.Q[(self.num + self.first) % self.max_size] = item
        self.num += 1 # تعداد عناصر را افزایش می‌دهد

    def dequeue(self):
        # اولین عنصر را از صف حذف کرده و بر می‌گرداند.
        if self.num == 0:
            raise Exception("Queue empty") # خطأ در صورت خالی بودن صف
        item = self.Q[self.first] # آولین عنصر را بر می‌دارد
        self.first = (self.first + 1) % self.max_size # آندیس اولین عنصر را به صورت دوری به روزرسانی می‌کند
        self.num -= 1 # تعداد عناصر را کاهش می‌دهد
        return item

    def front(self):
        # اولین عنصر صف را بدون حذف کردن بر می‌گرداند.
        if self.num == 0:
            raise Exception("Queue empty") # خطأ در صورت خالی بودن صف
        return self.Q[self.first]

    def is_empty(self):
        # این متد بررسی می‌کند که صف خالی است یا نه.
        return self.num == 0
```

```
# <span style="direction:rtl; unicode-bidi:embed;">بررسی می‌کند که آیا صفحه خالی است یا خیر</span>
return self.num == 0

def size(self):
# <span style="direction:rtl; unicode-bidi:embed;">تعداد عناصر فعلی در صف را برمی‌گرداند</span>
return self.num

def is_full(self):
# <span style="direction:rtl; unicode-bidi:embed;">بررسی می‌کند که آیا صفحه پر است یا خیر</span>
return self.num >= self.max_size
```

In [2]: # مثال استفاده از کلاس Queue

```
یک صف با ظرفیت ۱۰ ایجاد می‌کند. (جلوی صف) [[پشت صف)] # q = Queue(10)
q.enqueue("ra'na") # صف: ["ra'na"]
q.enqueue("vez") # صف: ["ra'na", "vez"]
q.enqueue("Arya") # صف: ["ra'na", "vez", "Arya"]
print("Queue size is:", q.size())
print(q.dequeue(), "left the queue.") # از صف خارج شد. صف: ["vez", "Arya"]
print("Front of queue is:", q.front()) # عنصر ابتدای صف: "vez"
q.enqueue("milda") # به صف اضافه شد. صف: ["vez", "Arya", "milda"]
q.dequeue() # عنصر ابتدای صف خارج شد. صف: ["Arya", "milda"]
q.dequeue() # عنصر ابتدای صف خارج شد. صف: ["milda"]
q.dequeue() # عنصر ابتدای صف خارج شد. صف: []
بررسی می‌کند که آیا صف خالی است؟ (توجه: این خط باید برای صف باشد، نه پشته). # (()
print("Stack is empty?", q.is_empty())
print("It was a queue!")
```

```
Queue size is: 3
ra'na left the queue.
Front of queue is: vez
Stack is empty? True
It was a queue!
```

تجسم مثال بالا:



## تمرین

به پیاده‌سازی بالا از صف این قابلیت را اضافه کنید که که عنصر `#am` صف را خروجی دهد.

## پشته

در بعضی موارد عناصر تنها از یک طرف اضافه می‌شوند و از همان طرف نیز خارج می‌شوند.

بعنوان مثال ظرف‌های کثیفی را در نظر بگیرید که بر روی هم قرار دارند و قصد شستن آنها را دارید. در هر مرحله اگر بخواهید ظرف را بردارید بالاترین آنها را انتخاب می‌کنید و آن را می‌شویید و اگر بخواهید ظرف کثیفی را به آنها اضافه کنید در بالای تمام آنها قرار می‌دهید.

یک پشته باید از توابع زیر پشتیبانی کند:

- در واقع در این موارد عنصری که دیرتر از همه اضافه شده است زودتر از همه خارج می‌گردد. به داده‌ساختاری که این‌گونه موارد را مدل می‌کند، پشته می‌گویند.
- **push(x)**:  $x$  را به بالای پشته اضافه می‌کند.
- **()pop**: عنصر بالای پشته را حذف می‌کند و آن را بازمی‌گرداند.
- **()top**: عنصر بالای پشته را باز می‌گرداند.
- **()size**: تعداد عناصر موجود در پشته را باز می‌گرداند.
- **()isEmpty**: خالی بودن پشته را مشخص می‌کند.
- **()isFull**: پر بودن پشته را مشخص می‌کند.

پیاده‌سازی پشته نیز به کمک آرایه یا لیست پیوندی انجام می‌شود.

توجه:

## پیاده‌سازی پشته به کمک آرایه

برای پیاده‌سازی پشته با آرایه از یک متغیر **num** استفاده می‌کنیم که همیشه به عنصر بالای پشته اشاره می‌کند.

In [3]:

```
class Stack:  
    def __init__(self, max_size):  
        متد سازنده: پشته را با ظرفیت مشخص شده مقداردهی اولیه می‌کند.  
        self.max_size = max_size # حداقل ظرفیت پشته  
        self.S = [0] * max_size # آرایه داخلی برای ذخیره عناصر پشته  
        self.num = 0 # تعداد عناصر فعلی در پشته (همچنین اندیس بالای پشته)  
  
    def push(self, item):  
        عنصر جدید را به بالای پشته اضافه می‌کند.  
        if self.num >= self.max_size:  
            خطأ در صورت پر بودن پشته # raise Exception("Stack overflow")  
            عنصر را در بالای پشته قرار می‌دهد # self.S[self.num] = item  
            تعداد عناصر را افزایش می‌دهد # self.num += 1  
  
    def pop(self):  
        عنصر بالای پشته را حذف کرده و برگرداند #  
        if self.num == 0:  
            خطأ در صورت خالی بودن پشته # raise Exception("Stack empty")  
            تعداد عناصر را کاهش می‌دهد (اشاره‌گر به عنصر قبلی) # self.num -= 1  
            عنصر حذف شده را برگرداند # return self.S[self.num]  
  
    def top(self):  
        عنصر بالای پشته را بدون حذف کردن برگرداند #  
        if self.num == 0:  
            خطأ در صورت خالی بودن پشته # raise Exception("Stack empty")  
            قرار دارد num-1 عنصر بالای پشته همیشه در # return self.S[self.num - 1]  
  
    def size(self):  
        تعداد عناصر فعلی در پشته را برگرداند #  
        return self.num  
  
    def is_full(self):  
        بررسی می‌کند که آیا پشته پر است یا خیر #  
        return self.num >= self.max_size  
  
    def is_empty(self):  
        بررسی می‌کند که آیا پشته خالی است یا خیر #  
        return self.num == 0
```

In [4]:

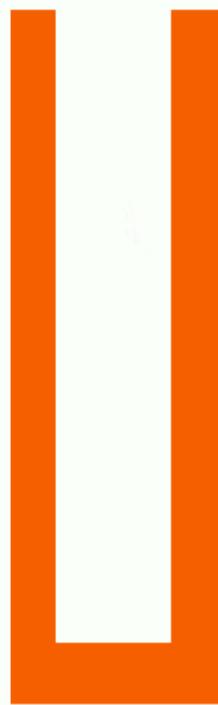
```
st = Stack(10) # [ ]: یک پشته با ظرفیت 10 ایجاد می‌کند. (بالای پشته)  
st.push(10) # [10]: پشته:  
st.push(30) # [10, 30]: پشته:  
st.push(1.5) # [10, 30, 1.5]: پشته:  
از پشته خارج شد. پشته: [1.5, 10]  
print(st.pop(), "is popped.") # 1.5 [10]  
st.push("salam") # "salam": به پشته اضافه شد. پشته: ["salam", 10]  
st.pop() # "salam": [10]: از پشته خارج شد. پشته:  
عنصر بالای پشته: 30  
print("Top of stack:", st.top()) # 30 [10]  
از پشته خارج شد. پشته: [ ]: از پشته خارج شد. پشته:  
بررسی می‌کند که آیا پشته خالی است؟ # st.is_empty() # True  
print("Stack is empty?", st.is_empty()) # False
```

1.5 is popped.

Top of stack: 30

Stack is empty? True

تجسم مثال بالا:



## صف دو طرفه

صف عادی محدودیت‌هایی برای مدل‌سازی دارد. برای مثال فرض کنید که داریم صف نانوایی را مدل می‌کنیم اما نانوا آشناییش را در ابتدای صف قرار می‌دهد!

برای همین لازم است که گاهی به سر صف هم عنصر اضافه شود یا حتی گاهی لازم است که از ته صف یک عنصر حذف شود (برای مثال یک نفر که حوصله‌ی صبر کردن ندارد چند نفر از آخر صف را منصرف می‌کند و سپس خودش بواشکی‌ای به آخر صف اضافه می‌شود).

در واقع صف دو طرفه قابلیت‌های پشتنه و صف را بهصورت همزمان پشتیبانی می‌کند:

- عنصر  $x$  را در ابتدای صف درج می‌کند: **push\_front(x)**
- عنصر  $x$  را در انتهای صف درج می‌کند: **push\_back(x)**
- عنصر ابتدای صف را حذف می‌کند و آن را باز می‌گرداند: **pop\_front(x)**
- عنصر انتهای صف را حذف و آن را باز می‌گرداند: **pop\_back(x)**
- عنصر ابتدای صف را باز می‌گرداند: **()front**
- عنصر انتهای صف را باز می‌گرداند: **()back**
- تعداد عناصر موجود در صف را باز می‌گرداند: **()size**
- خالی بودن یا نبودن صف را مشخص می‌کند: **()is\_empty**
- پر بودن یا نبودن صف را مشخص می‌کند: **()is\_full**

پیاده‌سازی صف دو طرفه هم به کمک آرایه یا لیست پیوندی انجام می‌شود.

توجه:

# پیاده‌سازی صف دو طرفه به کمک آرایه

تنها تفاوت این است که هنگام اضافه شدن عنصر در ابتدای صف باید مقدار متغیر `first` را یک واحد کاهش دهیم و `num` را یک واحد افزایش دهیم.

در هنگام حذف عنصر از انتهای صف باید تنها مقدار `num` را یک واحد کاهش دهیم.

```
In [5]: class DoubleEndedQueue:
    def __init__(self, max_size):
        مت د سازنده: صف دو طرفه را با ظرفیت مشخص شده مقداردهی اولیه می‌کند
        self.max_size = max_size # حداقل ظرفیت صف
        self.Q = [0] * max_size # آرایه داخلی برای ذخیره عناصر صف
        self.num = 0 # تعداد عناصر فعلی در صف
        self.first = 0 # انديس اولين عنصر در صف (سر صف)

    def push_back(self, item): # مانند صف يك طرفه
        # عنصر جديد را به انتهای صف اضافه می‌کند
        if self.num >= self.max_size:
            خطأ در صورت پر بودن صف
            raise Exception("Queue overflow")
        self.Q[(self.num + self.first) % self.max_size] = item # محاسبه انديس انتهای صف به صورت دوری
        self.num += 1 # تعداد عناصر را افزایش می‌دهد

    def push_front(self, item): # جديد: اضافه کردن از سر صف
        # عنصر جديد را به ابتدای صف اضافه می‌کند
        if self.num >= self.max_size:
            خطأ در صورت پر بودن صف
            raise Exception("Queue overflow")
        self.first = (self.first - 1) % self.max_size # انديس سر صف را به صورت دوری کاهش می‌دهد
        self.Q[self.first] = item # عنصر را در ابتدای صف قرار می‌دهد
        self.num += 1 # تعداد عناصر را افزایش می‌دهد

    def pop_front(self): # مانند صف يك طرفه
        # اولين عنصر را از ابتدای صف حذف کرده و برمی‌گرداند
        if self.num == 0:
            خطأ در صورت خالي بودن صف
            raise Exception("Queue empty")
        item = self.Q[self.first] # اولين عنصر را برمی‌دارد
        self.first = (self.first + 1) % self.max_size # انديس سر صف را به صورت دوری افزایش می‌دهد
        self.num -= 1 # تعداد عناصر را کاهش می‌دهد
        return item

    def front(self): # جديد: صف يك طرفه
        # اولين عنصر صف را بدون حذف کردن برمی‌گرداند
        if self.num == 0:
            خطأ در صورت خالي بودن صف
            raise Exception("Queue empty")
        return self.Q[self.first]

    def pop_back(self): # جديد: حذف کردن از انتهای صف
        # عنصر انتهای صف را حذف کرده و برمی‌گرداند
        if self.num == 0:
            خطأ در صورت خالي بودن صف
            raise Exception("Queue empty")
        self.num -= 1 # تعداد عناصر را کاهش می‌دهد (اشاره‌گر به عنصر قبلی)
        محاسبه انديس انتهای صف به صورت دوری برای دسترسی به عنصر
        return self.Q[(self.num + self.first) % self.max_size]

    def back(self): # مشاهده عنصر انتهای صف
        # عنصر انتهای صف را بدون حذف کردن برمی‌گرداند
        if self.num == 0:
            خطأ در صورت خالي بودن صف
            raise Exception("Queue empty")
        محاسبه انديس انتهای صف به صورت دوری برای دسترسی به عنصر
        return self.Q[(self.num + self.first - 1) % self.max_size]

    def is_empty(self):
        # بررسی می‌کند که آیا صف خالی است یا خیر
        return self.num == 0

    def size(self):
        # تعداد عناصر فعلی در صف را برمی‌گرداند
        return self.num

    def is_full(self):
        # بررسی می‌کند که آیا صف پر است یا خیر
        return self.num >= self.max_size
```

```
In [6]: # مثال استفاده از کلاس DoubleEndedQueue
dq = DoubleEndedQueue(7) # ایجاد می‌کند
dq.push_front(1) # [1]
dq.push_back(2) # [1, 2]
dq.push_front(4) # [4, 1, 2]
print("Queue size is:", dq.size())
dq.push_back(0) # [4, 1, 2, 0]
print("Back of queue is:", dq.back())
dq.pop_front() # [1, 2, 0]
print("Front of queue is:", dq.front())
dq.pop_back() # [1, 2]
dq.pop_back() # [1]
dq.pop_back() # []
```

Queue size is: 3  
Back of queue is: 0  
Front of queue is: 1

Out[6]: 1

تجسم مثال بالا:

push front/back 1

size=0

front

back

## لیست پیوندی

لیست پیوندی داده‌ساختاری بسیار پایه‌ای برای نگهداری توالی‌هاست. در لیست پیوندی هر عنصر تنها از عنصر بعدی خود مطلع است. البته در لیست پیوندی دوطرفه هر عنصر علاوه بر عنصر بعدی خود از عنصر قبلی خود نیز مطلع است.

در مورد عناصر اول و آخر می‌توان دو دیدگاه داشت: یکی اینکه عنصر قبل از عنصر اول، و عنصر بعد از عنصر آخر نداریم و دیگری اینکه اگر لیست پیوندی ما حلقوی باشد عنصر بعد از عنصر آخر عنصر اول است و عنصر قبل از عنصر اول عنصر آخر است.

لیست پیوندی از توابع زیر پشتیبانی می‌کند:

- عنصر  $x$  را پس از عنصر  $data$  درج می‌کند. **insert\_after(data,x)**
- عنصر  $x$  را حذف و آن را باز می‌گرداند. **delete(x)**
- اولین عنصر با  $data$  مساوی با  $value$  را باز می‌گرداند. **find(value)**
- عنصر  $i$ ام لیست را باز می‌گرداند. **get(i)**
- تعداد عناصر موجود در لیست را باز می‌گرداند. **()size**
- خالی بودن یا نبودن لیست را مشخص می‌کند. **()is\_empty**

به جدول زیر توجه کنید:

عملیات	آرایه	لیست پیوندی
درج در مکان مشخص و حذف	در جایی که $i$ -امین عنصر باشد، عناصر از $i+1$ تا $n$ را چنان‌که $i$ -امین عنصر را حذف شده بینشند.	search + time $O(n)$ $O(1)$
دسترسی به عنصر $i$	عنصر $i$ -ام را باز می‌گردانند.	$O(n)$

درج در یک مکان آرایه از  $O(n)$  است زیرا عناصر بعد از مکانی که عنصر جدید می‌خواهد در آنجا درج شود باید به جلو تغییر مکان دهنده و تعداد آنها از  $i$  است. اما برای درج یک عنصر مثل  $x$  در یک مکان مشخص از لیست پیوندی مانند درج در جایگاه  $i$ ، کافیست که عنصر بعدی  $i$ -امین عنصر در صف را برابر  $x$  و عنصر بعدی  $x$  را برابر عنصر  $i$ -ام قدیم قرار دهیم.

روند حذف نیز مشابه است. به عنصر  $i$ -ام در آرایه دسترسی مستقیم داریم اما در لیست پیوندی باید از عنصر اول  $i$  بار جلو برویم که بنابراین این عمل از  $O(n)$  است.

# پیاده‌سازی لیست پیوندی به کمک کلاس

یک کلاس برای **node** های لیستمان می‌گیریم و در آن برای هر **node** اطلاعات لازم را در **data** نگه می‌داریم. همچنین **node** بعد و قبل از آن را به ترتیب در **next** و **prev** آن قرار می‌دهیم.

حال به کمک کلاس **List** که در آن **head** به عنوان **node** قبل از اولین عنصر و بعد از آخرین عنصر قرار دارد لیست پیوندی دوسره را پیاده‌سازی می‌کنیم.

در واقع **head** یک **node** با مقدار **None** است که وظیفه‌اش نگه داشتن ابتدا و انتهای صف پیوندی ماست.

```
In [7]: class Node:
    def __init__(self, data):
        # جدید را با داده مشخص شده مقداردهی اولیه می‌کند (Node) متد سازنده: یک گره.
        self.data = data # داده‌ای که گره ذخیره می‌کند.
        self.next = None # اشاره‌گر به گره بعدی در لیست.
        self.prev = None # اشاره‌گر به گره قبلی در لیست (برای لیست دوطرفه).

class List:
    def __init__(self):
        # متد سازنده: یک لیست پیوندی دوطرفه خالی را مقداردهی اولیه می‌کند.
        # head به خودش اشاره می‌کند (sentinel node) یک گره نگهبان.
        self.head = Node(None)
        self.head.next = self.head
        self.head.prev = self.head
        self.n = 0 # تعداد عناصر واقعی در لیست.

    def get(self, ind):
        # را برمی‌گرداند (ind) عنصر در اندیس مشخص شده.
        if ind >= self.size():
            raise Exception("Out of list")
        x = self.head.next # شروع از اولین گره واقعی.
        for i in range(ind):
            x = x.next # پیمایش تا رسیدن به گره مورد نظر.
        return x

    def insert_after(self, x, data):
        # درج می‌کند 'x' را بعد از گره 'data' یک گره جدید با.
        y = Node(data) # ایجاد گره جدید.
        self.n += 1 # افزایش تعداد عناصر.
        y.prev = x # قرار می‌دهد 'x' را 'y' گره قبلی.
        y.next = x.next # قرار می‌دهد 'y' را گره بعدی که 'x' را گره ای که 'y' گره بعدی.
        x.next = y # قرار می‌دهد 'y' را 'x' گره بعدی.
        y.next.prev = y # قرار می‌دهد 'y' به آن اشاره می‌کند، گره قبلی اش را 'y' گره ای که.
        return y

    def delete(self, x):
        # را از لیست حذف می‌کند 'x' گره.
        self.n -= 1 # کاهش تعداد عناصر.
        x.prev.next = x.next # متصل می‌کند 'x' را به گره بعدی 'x' گره قبلی.
        x.next.prev = x.prev # متصل می‌کند 'x' را به گره قبلی 'x' گره بعدی.
        return x

    def find(self, val):
        # است را پیدا کرده و برمی‌گرداند 'val' اولین گره ای که داده آن برابر.
        x = self.head.next # شروع از اولین گره واقعی.
        for i in range(self.size()):
            if x.data == val:
                return x # گره یافت شد.
            x = x.next # پیمایش به گره بعدی.
        return None # گره یافت نشد.

    def size(self):
        # تعداد عناصر فعلی در لیست را برمی‌گرداند.
        return self.n

    def is_empty(self):
        # بررسی می‌کند که آیا لیست خالی است یا خیر.
        return self.n == 0
```

```
In [8]: # مثال استفاده از کلاس List
list_obj = List() # برای جلوگیری از تداخل نام با تابع (built-in)
list_obj.insert_after(list_obj.head, "milad") # head <-> milad
list_obj.insert_after(list_obj.get(0), "Arya") # head <-> Arya
```

```
list_obj.insert_after(list_obj.find("Arya").prev, "jabbar") # head <-> milad <-> jabbar <-> Arya  
list_obj.delete(list_obj.find("jabbar")) # head <-> milad <-> Arya  
print("Current size of list is:", list_obj.size())
```

Current size of list is: 2

تجسم مثال بالا:



# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل پنجم، بخش اول: ذخیره‌سازی و پیمایش درخت

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

## فهرست

• مقدمه

• تعاریف در درخت‌ها

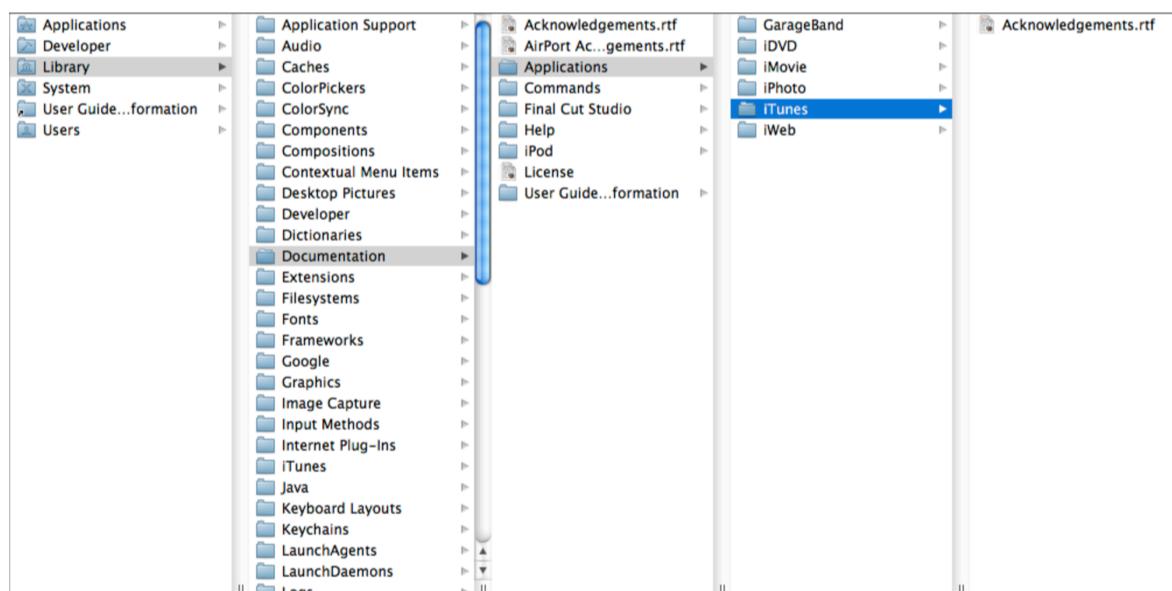
• ذخیره‌سازی درخت

• پیمایش درخت

## مقدمه

ازیلهای، لیست‌های پیوندی، صفحه‌ها و پشتنهای داده‌ساختارهای خطی هستند و تنها برای مشخص کردن ترتیبی از عناصر (که در خود ذخیره کرده‌اند) به کار می‌آیند. در کاربردهای فراوانی نیازمند ساختارهای دیگری هستیم که بتوانند روابط پیچیده‌تر و سلسله‌مراتبی بین داده‌ها را مدل کنند.

درخت‌ها بیانگر ساختارهایی سلسله‌مراتبی هستند که در بسیاری از مسائل به آن‌ها نیاز داریم. به طور مثال یکی از این کاربردها ذخیره‌سازی ساختار پوشش‌ها در کامپیوتر است که از ساختاری سلسله‌مراتبی پیروی می‌کند.



همچنین می‌توان درخت‌ها را در ساختارهایی کارآمدتر برای انجام عمل‌های خاص روی ساختارهای خطی نیز به کار گرفت، مانند درخت‌های جستجوی دودویی برای جستجوی سریع یا هیپ‌ها برای پیاده‌سازی صفحه‌ای اولویت.

## تعریف در درخت‌ها

در زبان نظریه گراف‌ها، درخت، گراف همبند بدون دور است. درخت ریشه‌دار نیز درختی است که در آن یک رأس خاص به عنوان ریشه انتخاب شده است و ساختاری سلسله‌مراتبی به آن بخشیده است. موارد زیر را برای درخت‌های ریشه‌دار تعریف می‌کنیم (اغلب اسامی به تقلید از درخت‌های تبارشناسی انتخاب شده‌اند):

- **عمق یک رأس:** فاصله‌ی رأس تا ریشه را عمق آن می‌نامیم. عمق ریشه صفر است.
- **ارتفاع درخت:** ماکسیمم عمق در بین همه‌ی رأس‌ها را ارتفاع درخت می‌نامیم.

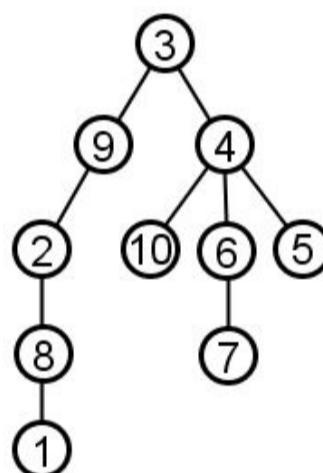
- پدر و فرزند: وقتی دو رأس مجاور باشند (با یال به هم وصل شده باشند)، رأسی که به ریشه نزدیکتر است را پدر دیگری می‌نامیم و رأس دورتر را فرزند رأس نزدیکتر. هر رأس (به جز ریشه) یک پدر و صفر یا چند فرزند دارد.
- اجداد: اجداد یک رأس همه‌ی رئوسی هستند که روی مسیر آن رأس به ریشه قرار دارند. منظور از جد  $\neq$  ایک رأس جدی است که فاصله‌اش تا آن رأس  $\neq$  است.
- نوادگان: نوادگان یک رأس همه‌ی رئوسی هستند که این رأس جدشان است. زیردرخت یک رأس مجموعه‌ی همه‌ی نوادگان آن رأس به همراه خود آن رأس است.
- برگ: رأسی که هیچ فرزندی نداشته باشد (این اصطلاح از درختان واقعی گرفته شده است).

## ذخیره سازی درخت

راههای گوناگونی برای ذخیره‌سازی یک درخت وجود دارد. ساده‌ترین راه برای ذخیره‌سازی یک درخت آن است که رأس‌های آن را شماره‌زده و شماره‌ی پدر هر رأس را در آن ذخیره کنیم. چون ریشه تنها رأس درخت است که پدر ندارد می‌توانیم قرارداد کنیم که پدر ریشه را 1 – یا خودش در نظر بگیریم. ما در این متن از قرارداد دوم استفاده می‌کنیم (پدر ریشه، خودش است).

ذخیره‌کردن پدرهای رئوس گرچه برای تعیین یک درخت کافی است، اما پیمایش آن را بسیار دشوار و زمان‌بر می‌کند. برای مثال، یافتن فرزندان یک رأس خاص، نیازمند پیمایش کل لیست پدرهای است. راهکار بهتر برای ذخیره‌سازی این است که برای هر رأس شماره‌ی پدرش و یک لیست از شماره‌های فرزندانش را ذخیره کنیم.

برای مثال یک ذخیره‌سازی از درخت زیر را در ادامه می‌آوریم.



رأس 3 ریشه‌ی درخت فرض شده است. در کد زیر از روی جفت‌های پدر و فرزند، لیست فرزندان هر رأس بدست آمده است:

```

In [1]: vertices = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] # لیست تمام رئوس موجود در درخت
parent = {
    3: 3, # ریشه است، پدرش خودش است.
    9: 3, # پدر رأس 3 است.
    4: 3, # پدر رأس 4، رأس 3 است.
    2: 9, # پدر رأس 2، رأس 9 است.
    10: 4, # پدر رأس 10، رأس 4 است.
    6: 4, # پدر رأس 6، رأس 4 است.
    5: 4, # پدر رأس 5، رأس 4 است.
    7: 6, # پدر رأس 7، رأس 6 است.
    8: 2, # پدر رأس 8، رأس 2 است.
    1: 8 # پدر رأس 1، رأس 8 است
}

# محاسبات برای ایجاد لیست فرزندان هر رأس
children = dict()
for vertex in vertices:
    children[vertex] = list() # برای هر رأس یک لیست خالی برای فرزندانش ایجاد می‌کند
    for vertex in vertices:
        if vertex != parent[vertex]:
            children[parent[vertex]].append(vertex) # رأس فعلی را به لیست فرزندان پدرش اضافه می‌کند

# چاپ لیست فرزندان برای هر رأس
for vertex in vertices:
    print ("children of %d are: %s" % (vertex, str(children[vertex])))
  
```

```

children of 1 are: []
children of 2 are: [8]
children of 3 are: [4, 9]
children of 4 are: [5, 6, 10]
children of 5 are: []
children of 6 are: [7]
children of 7 are: []
children of 8 are: [1]
children of 9 are: [2]
children of 10 are: []

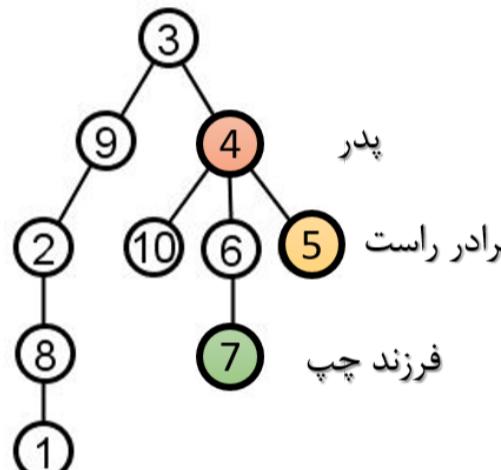
```

راه دیگر ذخیره‌سازی درخت استفاده از یک شیء برای نمایش هر رأس است. این روش به خصوص برای درخت‌هایی که ساختارشان زیاد تغییر می‌کند یا نیاز به دسترسی سریع به روابط پدر-فرزندی و برادری دارند، مفید است.

برای استفاده از حافظه می‌توان برای هر رأس تنها ۳ اشاره‌گر انتخاب کرد:

- که به چپ‌ترین فرزند یک رأس اشاره می‌کند.
- که به رأس بعدی در سمت راست خود اشاره می‌کند (در واقع به برادر سمت راست این رأس). این اشاره‌گر برای اتصال فرزندان یک پدر به صورت یک لیست پیوندی استفاده می‌شود.
- که به پدر خود اشاره می‌کند.

در شکل زیر این سه اشاره‌گر برای رأس شماره 6 مشخص شده‌اند.



در قطعه کد زیر، پیاده‌سازی یک رأس با استفاده از تعاریف فوق در قالب کلاس **TreeNode** آمده است.

```
In [2]: class TreeNode:
    def __init__(self, label):
        # مشخص شده مقداردهی اولیه می‌کند (label) متاد سازنده: یک گره درخت را با برجسب اشاره‌گر به گره پدر
        self.parent = None
        self.left_child = None # اشاره‌گر به چپ‌ترین فرزند
        self.right_sibling = None # اشاره‌گر به برادر سمت راست
        self.label = label # برجسب یا داده مربوط به گره

    def __str__(self):
        # متاد برای نمایش رشته‌ای از گره (برای چاپ)
        return "TreeNode(%s)" % str(self.label)
```

در قطعه کد زیر، درختی با استفاده از رأس‌های بالا (کلاس **TreeNode**)، تعریف می‌کنیم. این کلاس شامل متدهایی برای مدیریت ساختار درخت، اضافه کردن گره‌ها و جستجو در آن‌ها است.

```
In [3]: class Tree:
    def __init__(self):
        # متاد سازنده: یک درخت خالی ایجاد می‌کند.
        self.root = None # است ریشه درخت، در ابتدا None.

    def assign_root(self, label):
        # ریشه درخت را تعیین می‌کند. فقط یک بار قابل فراخوانی است.
        assert self.root is None # اطمینان از اینکه ریشه قبلًا تعیین نشده باشد.
        self.root = TreeNode(label) # ایجاد گره ریشه.

    def is_empty(self):
        # بررسی می‌کند که آیا درخت حالی است یا خیر.
        return self.root is None

    def add_new_node_1(self, parent, label):
        # این روش فرزند جدید را به عنوان چپ‌ترین فرزند والد قرار می‌دهد و فرزند قبلی را به سمت راست شیفت می‌دهد.
        new_node = TreeNode(label)
        left_child = parent.left_child
        parent.left_child = new_node
        new_node.right_sibling = left_child
        new_node.parent = parent
        return new_node
```

```

def add_new_node_2(self, parent, label):
    روش دوم برای اضافه کردن فرزند جدید به یک گره والد.
    # این روش فرزند جدید را به عنوان راستترین فرزند در لیست خواهان قرار می‌دهد
    new_node = TreeNode(label)
    new_node.parent = parent
    if parent.left_child is None:
        اگر والد فرزندی ندارد، این گره اولین فرزند (چپترین) خواهد بود
        parent.left_child = new_node
    else:
        در غیر این صورت، به انتهای لیست خواهان می‌روم
        left_child = parent.left_child
        while left_child.right_sibling is not None:
            left_child = left_child.right_sibling
        left_child.right_sibling = new_node
    return new_node

def add_new_node(self, parent, label):
    # (استفاده می‌کند از روش) متد اصلی برای اضافه کردن گره جدید
    return self.add_new_node_2(parent, label)

def find_in_subtree(self, label, node):
    یک گره را با بررسی مشخص شده در زیردرخت یک گره خاص پیدا می‌کند (بازگشتی)
    if node.label == label:
        گره یافت شد
        return node
    child = node.left_child
    while child is not None:
        result = self.find_in_subtree(label, child) # جستجو در زیردرخت فرزند
        if result is not None:
            اگر در زیردرخت فرزند یافت شد
            return result
        child = child.right_sibling # رفتن به برادر بعدی
    گره در این زیردرخت یافت نشد
    return None

def find_by_label(self, label):
    یک گره را با بررسی مشخص شده در کل درخت پیدا می‌کند
    if self.is_empty():
        درخت خالی است
        return None
    return self.find_in_subtree(label, self.root) # شروع جستجو از ریشه

def add_new_node_by_label(self, parent_label, label):
    یک گره جدید را با یافتن گره والد بر اساس بررسی آن اضافه می‌کند
    parent_node = self.find_by_label(parent_label)
    if parent_node is None:
        raise Exception(f"Parent with label {parent_label} not found.")
    self.add_new_node(parent_node, label)

def get_subtree_size(self, node):
    اندازه (تعداد گره‌ها) یک زیردرخت را محاسبه می‌کند
    if node is None:
        زیردرخت خالی است
        return 0
    خود گره فعلی را می‌شمارد
    child = node.left_child
    while child is not None:
        count += self.get_subtree_size(child) # جمع کردن اندازه زیردرخت فرزندان
        child = child.right_sibling
    return count

def get_size(self):
    اندازه (تعداد گره‌ها) کل درخت را محاسبه می‌کند
    if self.is_empty():
        return 0
    return self.get_subtree_size(self.root)

```

حال در قطعه کد زیر ابتدا درختی که در شکل ابتدای این محتوا نمایش داده شده بود را می‌سازیم.

اگر بخواهیم رأسی را بر حسب **label** آن بیابیم از تابع **find\_by\_label** استفاده می‌کنیم.

اگر بخواهیم اندازه‌ی یک درخت را بیابیم کافی است از تابع **get\_size** درخت استفاده کنیم.

با توجه به قطعه کد بالا موارد زیر را بررسی کنید:

- همان طور که می‌بینید دو تابع برای اضافه کردن راس به درخت وجود دارد (**add\_new\_node\_1** و **add\_new\_node\_2**)

فرق این دو پیاده‌سازی مختلف در چیست؟ \*\*پاسخ:

**add\_new\_node\_1** ▪  
می‌کند. اگر والد قبلی فرزند چپ داشته باشد، آن فرزند چپ و تمام خواهانش به سمت راست شیفت داده می‌شوند تا فرزند جدید در ابتدای لیست فرزندان قرار گیرد. این روش شبیه به اضافه کردن به ابتدای یک لیست پیوندی است که زمان آن  $O(1)$  است.

**add\_new\_node\_2** ▪  
والد اضافه می‌کند. اگر والد قبلاً فرزندی داشته باشد، تابع باید تمام لیست خواهران را پیمایش کند تا به آخرین خواهر برسد و فرزند جدید را بعد از آن اضافه کند. این روش شبیه به اضافه کردن به انتهای یک لیست پیوندی است که زمان آن  $O(k)$  است، که  $k$  تعداد فرزندان فعلی والد است.

• **تابع find\_in\_subtree چگونه عمل می‌کند؟** \*

تابع **find\_in\_subtree** به صورت بازگشتی عمل می‌کند. ابتدا بررسی می‌کند که آیا **label** گره فعلی با **label** مورد جستجو برابر است یا خیر. اگر برابر بود، همان گره را برمی‌گرداند. در غیر این صورت، به سراغ چپ‌ترین فرزند گره فعلی می‌رود و به صورت بازگشتی، همین تابع را روی آن فرزند فراخوانی می‌کند. اگر نتیجه‌ای از جستجو در زیردرخت آن فرزند به دست آمد، آن را برمی‌گرداند. اگر نه، به سراغ برادر سمت راست آن فرزند می‌رود و همین روند را برای تمام فرزندان و زیردرخت‌های آن‌ها ادامه می‌دهد تا گره مورد نظر یافته شود یا تمام زیردرخت پیمایش شود. این یک نوع پیمایش عمق اول (Depth-First Search) است.

• **چگونه می‌توان اندازه زیر درخت فرزند چپ ریشه را به دست آورد؟** \*

برای به دست آوردن اندازه زیردرخت فرزند چپ ریشه، ابتدا باید گره ریشه را پیدا کنیم (مثلاً با **tree.root**). سپس به فرزند چپ آن (**tree.root.left\_child**) دسترسی پیدا می‌کنیم. در نهایت، تابع **.tree.get\_subtree\_size(tree.root.left\_child)** را روی این فرزند چپ فراخوانی می‌کنیم:

```
In [4]: tree = Tree() # ایجاد یک شریعه درخت جدید
tree.assign_root(3) # تعیین گره 3 به عنوان ریشه درخت
tree.add_new_node_by_label(3, 9) # افزودن 9 به عنوان فرزند 3
tree.add_new_node_by_label(9, 2) # افزودن 2 به عنوان فرزند 9
tree.add_new_node_by_label(3, 4) # افزودن 4 به عنوان فرزند 3
tree.add_new_node_by_label(4, 10) # افزودن 10 به عنوان فرزند 4
tree.add_new_node_by_label(4, 6) # افزودن 6 به عنوان فرزند 4
tree.add_new_node_by_label(4, 5) # افزودن 5 به عنوان فرزند 4
tree.add_new_node_by_label(2, 8) # افزودن 8 به عنوان فرزند 2
tree.add_new_node_by_label(6, 7) # افزودن 7 به عنوان فرزند 6
tree.add_new_node_by_label(8, 1) # افزودن 1 به عنوان فرزند 8

print(tree.find_by_label(2)) # پیدا کردن و چاپ گره با برچسب 2
print(tree.get_size()) # چاپ اندازه کل درخت
```

```
TreeNode(2)
10
```

# پیمایش درخت

فرض کنید که یک درخت ریشه‌دار داریم و می‌خواهیم با حرکت روی یال‌های درخت، همه‌ی گره‌های آن را هرکدام یکبار ملاقات کنیم. به این کار پیمایش درخت می‌گویند.

پیمایش درخت‌ها و ترتیب حاصل از نوع پیمایش در بسیاری از مسائل کاربرد دارد. برای مثال، برای کپی کردن یک درخت، سریالایز کردن آن، یا اجرای عملیات خاص روی گره‌ها به ترتیبی مشخص، از پیمایش‌ها استفاده می‌شود. در اینجا به ۳ نوع پرکاربرد پیمایش درخت‌ها می‌پردازیم:

اولین نوع پیمایش، پیمایش پیش‌ترتیب یا **preorder** است که در آن ابتدا ریشه و پس از آن به شکل بازگشتی هر یک از زیر درخت‌ها (فرزندها) از چپ به راست، پیمایش می‌شود.

حاصل پیمایش **preorder** درخت آمده در شکل ابتدای محتوا، پس از اجرای کد زیر قابل مشاهده است.

```
In [6]: def pre_order(tree_instance, node):
    """
    را روی یک درخت انجام می‌دهد (Preorder Traversal). این تابع پیمایش پیش‌ترتیب ترتیب: ریشه -> فرزند چپ -> برادران راست (بازگشتی).
    Args:
        tree_instance (Tree): نمونه‌ای از کلاس Tree.
        node (TreeNode): گره فعلی که از آن پیمایش را شروع می‌کنیم.
    Returns:
        list: لیستی از برچسب‌های گره‌ها به ترتیب پیمایش پیش‌ترتیب.
    """
    order_list = list()
    if node is None:
        return order_list # اگر گره خالی بود، لیست خالی برمی‌گرداند.
    order_list.append(node.label) # ابتدا برچسب گره فعلی را اضافه می‌کند (ریشه).
    order_list.append(*pre_order(tree_instance, node.left)) # از زیر درخت چپ
    order_list.append(*pre_order(tree_instance, node.right)) # از زیر درخت راست
```

```

child = node.left_child # به سراغ چپ‌ترین فرزند می‌رود.
while child is not None:
    به صورت بازگشتی، پیمایش پیش‌ترتیب را روی زیردرخت فرزند انجام می‌دهد
    و نتایج را به لیست فعلی اضافه می‌کند
    order_list.extend(pre_order(tree_instance, child)) # فراخوانی مستقیم تابع، نه از طریق tree.pre_order
    child = child.right_sibling # به سراغ برادر سمت راست می‌رود

return order_list

```

# لازم است tree.pre\_order اختیاری، اما برای استفاده به شکل) به عنوان یک متده کلاس Tree به متده pre\_order اتصال تابع

```

Tree.pre_order = pre_order
print(tree.pre_order(tree.root)) # روی نمونه درخت و ریشه آن pre_order فراخوانی متده
[3, 9, 2, 8, 1, 4, 10, 6, 7, 5]

```

نوع دوم پیمایش، پیمایش پس‌ترتیب یا **postorder** است که در آن ابتدا زیردرخت فرزندان به ترتیب از چپ به راست به صورت بازگشتی پیمایش می‌شود و در نهایت ریشه درخت پیمایش می‌شود. این نوع پیمایش برای حذف درخت یا آزاد کردن منابع از برگ‌ها به سمت ریشه مفید است.

پیمایش **postorder** درخت آمده در شکل ابتدای محتوا، پس از اجرای کد زیر قابل مشاهده است.

```

In [8]: def post_order(tree_instance, node):
    """
    را روی یک درخت انجام می‌دهد (Postorder Traversal) این تابع پیمایش پس‌ترتیب
    ترتیب: فرزند چپ -> برادران راست (بازگشتی) -> ریشه

    Args:
        tree_instance (Tree): نمونه‌ای از کلاس Tree
        node (TreeNode): گره فعلی که از آن پیمایش را شروع می‌کنیم

    Returns:
        list: لیستی از برچسب‌های گره‌ها به ترتیب پیمایش پس‌ترتیب
    """
    order_list = list()
    if node is None:
        return order_list # اگر گره خالی بود، لیست خالی برمی‌گرداند

    child = node.left_child # به سراغ چپ‌ترین فرزند می‌رود
    while child is not None:
        به صورت بازگشتی، پیمایش پس‌ترتیب را روی زیردرخت فرزند انجام می‌دهد
        order_list.extend(post_order(tree_instance, child)) # فراخوانی مستقیم تابع
        child = child.right_sibling # به سراغ برادر سمت راست می‌رود

    در نهایت، برچسب گره فعلی (ریشه) را اضافه می‌کند
    order_list.append(node.label)

    return order_list

```

```

# به کلاس post_order اتصال تابع
Tree.post_order = post_order
print(tree.post_order(tree.root)) # فراخوانی متده post_order
[1, 8, 2, 9, 10, 7, 6, 5, 4, 3]

```

سومین نوع پیمایش، پیمایش میان‌ترتیب یا **inorder** است که در آن ابتدا زیردرخت فرزند چپ، پس از آن ریشه و در نهایت زیردرخت بقیه فرزندان به ترتیب از چپ به راست به شکل بازگشتی پیمایش می‌شود. این نوع پیمایش به طور خاص برای درخت‌های جستجوی دودویی (Binary Search Trees) کاربرد دارد و عناصر را به ترتیب مرتب شده برمی‌گرداند.

پیمایش **inorder** درخت آمده در شکل ابتدای محتوا، پس از اجرای کد زیر قابل مشاهده است.

```

In [10]: def in_order(tree_instance, node):
    """
    را روی یک درخت انجام می‌دهد (Inorder Traversal) این تابع پیمایش میان‌ترتیب
    ترتیب: زیردرخت فرزند چپ -> ریشه -> زیردرخت برادران راست (بازگشتی)

    Args:
        tree_instance (Tree): نمونه‌ای از کلاس Tree
        node (TreeNode): گره فعلی که از آن پیمایش را شروع می‌کنیم

    Returns:
        list: لیستی از برچسب‌های گره‌ها به ترتیب پیمایش میان‌ترتیب
    """
    order_list = list()
    if node is None:
        return order_list # اگر گره خالی بود، لیست خالی برمی‌گرداند

    # پیمایش زیردرخت فرزند چپ (اگر وجود داشته باشد)
    child = node.left_child
    if child is not None: # اطمینان از وجود فرزند چپ قبل از فراخوانی بازگشتی
        order_list.extend(in_order(tree_instance, child))

    order_list.append(node.label) # اضافه کردن برچسب گره فعلی (ریشه) پس از پیمایش فرزند چپ

    return order_list

```

```

پیمایش زیردرخت برادران راست (اگر وجود داشته باشد) #
اشاره دارد child در اینجا همچنان به next_sibling: توجه: در این پاده‌سازی #
برای پیمایش برادران استفاده کرد باید از # اصلاح برای پیمایش صحیح برادران راست #
شروع از اولین فرزند (که قبلاً پیمایش شده) # current_sibling = node.left_child
# if current_sibling is not None:
    current_sibling = current_sibling.right_sibling # رفتن به اولین برادر (اگر وجود دارد)
# return order_list

# اتصال تابع in_order به کلاس Tree
Tree.in_order = in_order

print(tree.in_order(tree.root)) # فراخوانی متدهای in_order.

[1, 8, 2, 9, 3, 10, 4, 7, 6, 5]

```

## خودآزمایی ۱

تابع root\_left\_subtree\_size را طوری کامل کنید که با گرفتن لیست پیمایش شده‌ی inorder و preorder یک گراف، اندازه‌ی زیردرخت فرزند چپ ریشه را برگرداند.

```

In [13]: def root_left_subtree_size(preorder_list, inorder_list):
    """
    یک درخت preorder و inorder با استفاده از لیست‌های پیمایش
    اندازه زیردرخت فرزند چپ ریشه را محاسبه می‌کند.

    Args:
        preorder_list (list): لیست برچسب‌های گره‌ها به ترتیب پیمایش preorder.
        inorder_list (list): لیست برچسب‌های گره‌ها به ترتیب پیمایش inorder.

    Returns:
        int: اندازه زیردرخت فرزند چپ ریشه.
    """
    if not preorder_list:
        return 0 # اگر لیست خالی بود، اندازه ۰ است.

    root_label = preorder_list[0] # preorder در # پیدا کردن اندیس ریشه در
    # در زیردرخت چپ ریشه قرار دارند، inorder تمام عناصر قبل از ریشه در
    root_inorder_index = inorder_list.index(root_label)

    # قبل از ریشه آمده‌اند inorder اندازه زیردرخت چپ برابر است با تعداد عناصری که در
    left_subtree_size = root_inorder_index

    return left_subtree_size

    # مثال استفاده:
    # در محیط استاندارد پایتون تعریف نشده است 'Tester' تابع
    # بنابراین برای تست مستقیم تابع، آن را فراخوانی کرده و نتیجه را چاپ می‌کنیم
    preorder_example = [3, 9, 2, 8, 1, 4, 10, 6, 7, 5]
    inorder_example = [1, 8, 2, 9, 3, 10, 4, 7, 6, 5]

    result = root_left_subtree_size(preorder_example, inorder_example)
    print(f"اندازه زیردرخت فرزند چپ ریشه: {result}")

    # خروجی مورد انتظار برای این مثال:
    print("اندازه زیردرخت فرزند چپ ریشه: 4")

```

اندازه زیردرخت فرزند چپ ریشه: 4

## خودآزمایی ۲

تابع left\_subtree\_size را طوری کامل کنید که با گرفتن لیست پیمایش شده‌ی inorder و preorder یک گراف و همچنین لیبل یک رأس آن، اندازه‌ی زیردرخت فرزند چپ رأس با آن لیبل را برگرداند.

```

In [18]: def left_subtree_size(preorder_list, inorder_list, node_label):
    """
    یک درخت preorder و inorder این تابع با استفاده از لیست‌های پیمایش
    و برچسب یک گره، اندازه زیردرخت فرزند چپ آن گره را محاسبه می‌کند.

    Args:
        preorder_list (list): لیست برچسب‌های گره‌ها به ترتیب پیمایش preorder.
        inorder_list (list): لیست برچسب‌های گره‌ها به ترتیب پیمایش inorder.
        node_label: برچسب گره‌ای که می‌خواهیم اندازه زیردرخت چپ فرزنش را پیدا کنیم.

    Returns:
        int: اندازه زیردرخت فرزند چپ گره با node_label.
    """
    if not preorder_list or node_label not in preorder_list:
        return 0 # اگر لیست خالی بود یا گره وجود نداشت

    # (اولین عنصر آن زیردرخت) preorder پیدا کردن اندیس گره اصلی در
    # node_preorder_index = preorder_list.index(node_label)

    # این خط در این تابع مستقیماً استفاده نمی‌شود اما برای درک ساختار مفید است
    # node_preorder_index = preorder_list.index(node_label)

```

## خودآزمایی ۳

تابع left\_subtree\_size را طوری کامل کنید که با گرفتن لیست پیمایش شده‌ی inorder و preorder یک گراف و همچنین لیبل یک رأس آن، اندازه‌ی زیردرخت فرزند چپ رأس با آن لیبل را برگرداند.

```

In [18]: def left_subtree_size(preorder_list, inorder_list, node_label):
    """
    یک درخت preorder و inorder این تابع با استفاده از لیست‌های پیمایش
    و برچسب یک گره، اندازه زیردرخت فرزند چپ آن گره را محاسبه می‌کند.

    Args:
        preorder_list (list): لیست برچسب‌های گره‌ها به ترتیب پیمایش preorder.
        inorder_list (list): لیست برچسب‌های گره‌ها به ترتیب پیمایش inorder.
        node_label: برچسب گره‌ای که می‌خواهیم اندازه زیردرخت چپ فرزنش را پیدا کنیم.

    Returns:
        int: اندازه زیردرخت فرزند چپ گره با node_label.
    """
    if not preorder_list or node_label not in preorder_list:
        return 0 # اگر لیست خالی بود یا گره وجود نداشت

    # (اولین عنصر آن زیردرخت) preorder پیدا کردن اندیس گره اصلی در
    # node_preorder_index = preorder_list.index(node_label)

    # این خط در این تابع مستقیماً استفاده نمی‌شود اما برای درک ساختار مفید است
    # node_preorder_index = preorder_list.index(node_label)

```

```

# پیدا کردن اندیس گره اصلی در inorder
node_inorder_index = inorder_list.index(node_label)

قبل از گره اصلی قرار دارند، عناصر زیردرخت چپ در
اندازه زیردرخت چپ برابر است با تعداد این عناصر
left_subtree_inorder = inorder_list[:node_inorder_index]

برای یافتن فرزندان چپ گره اصلی استفاده می‌کنیم preorder از لیست
بعد از گره اصلی است preorder فرزند چپ گره اصلی، اولین عنصر در
هم در زیردرخت چپ گره اصلی قرار دارد که در inorder راه ساده‌تر: اندازه زیردرخت چپ برای است با تعداد عناصری که در
می‌آیند node_label هم بعد از preorder_list فرزند قرار دارند و در node_label قبل از
خودش شامل تمام عناصر زیردرخت چپ است inorder_list_left_part اما چون
طول آن برابر با اندازه زیردرخت چپ است

return len(left_subtree_inorder)

```

# مثال استفاده برای left\_subtree\_size:  
در محیط استاندارد پایتون تعریف نشده است' تابع  
preorder\_example\_node = [3, 9, 2, 8, 1, 4, 10, 6, 7, 5]  
inorder\_example\_node = [1, 8, 2, 9, 3, 10, 4, 7, 6, 5]  
node\_label\_example = 2  
result\_node = left\_subtree\_size(preorder\_example\_node, inorder\_example\_node, node\_label\_example)  
print(f"اندازه زیردرخت فرزند چپ گره {node\_label\_example}: {result\_node}")  
خروجی مورد انتظار برای این مثال: 2  
اندازه زیردرخت فرزند چپ گره 2 : 2

## خودآزمایی ۳

تابع change\_order را طوری کامل کنید که با گرفتن لیست پیمایش شده‌ی inorder و preorder یک گراف، لیست پیمایش postorder آن را برگرداند.

در این سوال فرض شده است که هر رأس درخت حداکثر دو فرزند دارد! (چرا این فرض برای حل سوال لازم است؟)

\*\*پاسخ:\*\* این فرض (حداکثر دو فرزند) برای حل این سوال لازم است زیرا در پیمایش میان‌ترتیب (inorder) تنها می‌توانیم زیردرخت چپ و زیردرخت راست را به سادگی از ریشه جدا کنیم. اگر یک گره بیش از دو فرزند داشته باشد (مثلاً سه فرزند: چپ، میانی، راست)، پیمایش میان‌ترتیب به تنهایی نمی‌تواند به طور یکتا مزه‌های بین زیردرخت‌های میانی را مشخص کند. در واقع، با داشتن فقط preorder و inorder، تنها می‌توانیم ساختار درخت‌های دودویی (با حداکثر دو فرزند) را به طور یکتا بازسازی کنیم.

\*\*توجه:\*\* کد زیر با ترتیب postorder مثال آورده شده در کد زیر با ترتیب inorder آورده شده در کد زیر با ترتیب postorder ای که برای شکل بالا به دست آوردیم متفاوت شد. چون در آن درجه بعضی رؤوس بیشتر از ۲ است.

```

In [19]: def change_order(preorder_list, inorder_list):
    """
    یک درخت دودویی inorder و preorder این تابع با گرفتن لیست‌های پیمایش
    آن را برمی‌گرداند postorder لیست پیمایش.
    """

    Args:
        preorder_list (list): لیست برچسب‌های گره‌ها به ترتیب پیمایش preorder.
        inorder_list (list): لیست برچسب‌های گره‌ها به ترتیب پیمایش inorder.

    Returns:
        list: لیستی از برچسب‌های گره‌ها به ترتیب پیمایش postorder.
    """
    if not preorder_list or not inorder_list:
        return []
    # اگر یکی از لیست‌ها خالی بود، لیست خالی برمی‌گرداند #
    root_label = preorder_list[0] # preorder، ریشه است.
    root_inorder_index = inorder_list.index(root_label)

    # به زیردرخت چپ و راست inorder تقسیم
    left_inorder = inorder_list[:root_inorder_index]
    right_inorder = inorder_list[root_inorder_index + 1:]

    # به زیردرخت چپ و راست preorder تقسیم
    # اندازه زیردرخت چپ برابر با طول left_inorder است.
    left_preorder = preorder_list[1 : 1 + len(left_inorder)]
    right_preorder = preorder_list[1 + len(left_inorder) :]

    # فرخوانی بازگشتی برای زیردرخت چپ و راست
    postorder_list = []
    postorder_list.extend(change_order(left_preorder, left_inorder))
    postorder_list.extend(change_order(right_preorder, right_inorder))

    postorder_list.append(root_label) # اضافه کردن ریشه در انتهای postorder.

    return postorder_list

```

# مثال change\_order:  
در محیط استاندارد پایتون تعریف نشده است' تابع

```
preorder_example_order = [3, 9, 2, 8, 1, 4, 10, 6, 7, 5]
inorder_example_order = [1, 8, 2, 9, 3, 10, 4, 7, 6, 5]

result_order = change_order(preorder_example_order, inorder_example_order)
print(f"پیماش خروجی مورد انتظار برای این مثال: {result_order}")

# [3 ,4 ,6 ,5 ,7 ,10 ,9 ,2 ,8 ,1]
print(f"پیماش Postorder: {result_order}")
```

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل پنجم، بخش دوم: درخت دودویی جست وجو

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

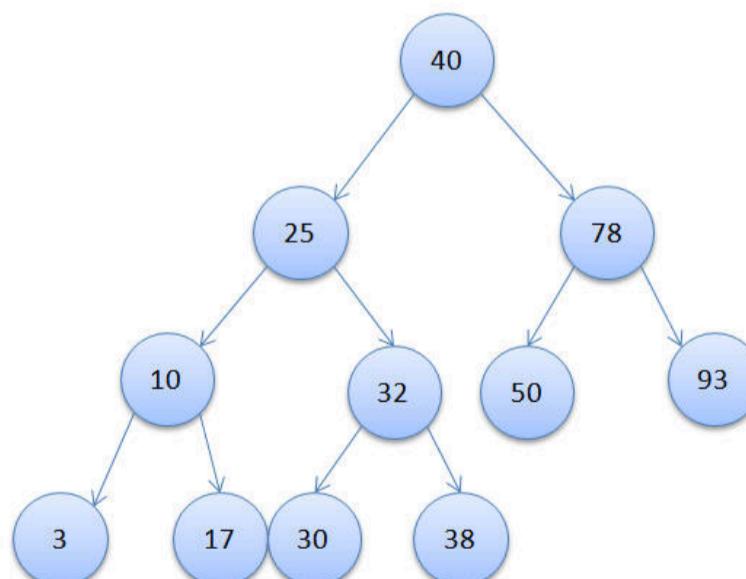
### فهرست

- مقدمه
- جستجو
- درج
- حذف
- میانگین ارتفاع درخت جست‌و‌جو
- پایین‌ترین جد مشترک

## مقدمه

درخت دودویی جست‌و‌جو (به انگلیسی: **Binary Search Tree** یا به اختصار BST) درختی ریشه‌دار و دودویی است که به ازای هر رأس مانند  $v$ :

- مقادیر تمامی رأس‌های زیردرخت فرزند سمت چپ آن از مقدار رأس  $v$  کوچک‌تر است.
- مقادیر تمامی رأس‌های زیردرخت فرزند سمت راستش از  $v$  بزرگ‌تر است.



هر رأس درون BST دارای یک برچسب (label) برای ذخیره مقدار، اشاره‌گر به فرزند چپ (`leftChild`)، اشاره‌گر به فرزند راست (`rightChild`) و اشاره‌گر به پدرش (`parent`) است.

به غیر از رأس ریشه، بقیه رأس‌ها حتماً پدر خواهند داشت. این ساختار امکان جستجو، درج و حذف کارآمد را فراهم می‌کند.

In [1]:

```
class Node:  
    def __init__(self, label, parent):  
        # را مقداردهی اولیه می‌کند BST جدید برای (Node) متولد می‌سازد: یک گره.  
        self.label = label # برچسب یا مقدار گره  
        self.parent = parent # اشاره‌گر به گره پدر  
        self.leftChild = None # اشاره‌گر به فرزند چپ  
        self.rightChild = None # اشاره‌گر به فرزند راست
```

```

def __str__(self):
    # جلوگیری شود فقط برچسب آن را برمی‌گرداند تا از نمایش رشته‌ای گردد
    return str(self.label)

```

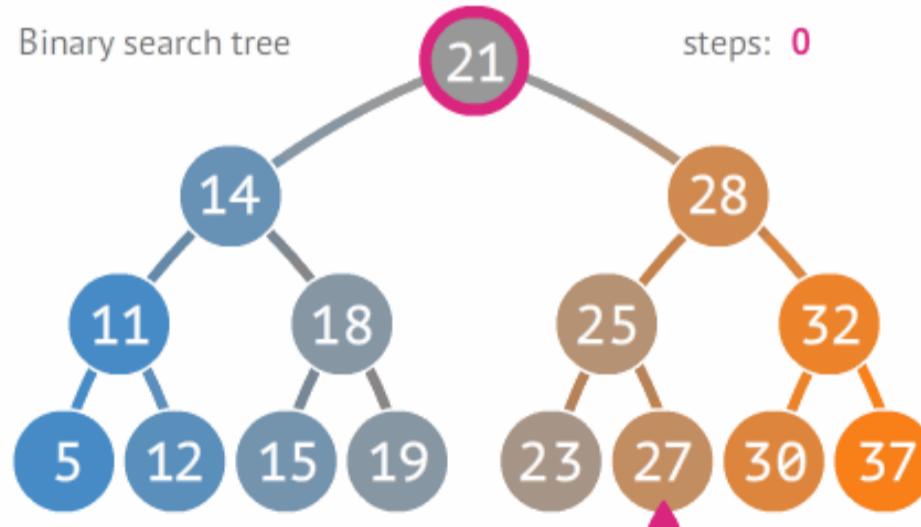
برای کار با این ساختمان داده، عمل اصلی متصور هستیم که کارایی آنها به دلیل خاصیت مرتب بودن BST، بسیار بالا است:

- **جستجو (Search):** یافتن یک عنصر خاص در درخت.
- **درج (Insertion):** اضافه کردن یک عنصر جدید به درخت.
- **حذف (Deletion):** حذف یک عنصر موجود از درخت.

(می‌توانید در [اینجا](#) عملیات‌های گفته شده را به صورت تعاملی مشاهده کنید.)

## جستجو

برای جستجو یک عنصر در BST، با شروع از ریشه، به ازای هر رأسی که درون آن هستیم، مقداری که می‌خواهیم پیدا کنیم را با مقدار رأس فعلی مقایسه می‌کنیم. در صورت مساوی بودن، عنصر مورد نظر را پیدا کرده‌ایم. اما در غیر این صورت، بسته به بزرگتر یا کوچکتر بودن مقدار عنصر مورد نظر ما از برچسب رأسی که داخلش هستیم، جستجو را در زیردرخت سمت راست یا چپ این رأس ادامه می‌دهیم. این رویه به صورت بازگشتی یا تکراری ادامه می‌یابد تا گره یافت شود یا به یک گره None (برگ) برسیم.



```

In [2]: def search(node, label):
    """
    به صورت بازگشتی جستجو می‌کند BST این تابع یک عنصر را در
    Args:
        گره فعلی که جستجو از آن شروع می‌شود.
        node (Node): برچسب (مقدار) مورد جستجو.
    Returns:
        Node or None: اگر یافت نشد None مورد نظر است، یا label گره‌ای که شامل label گردد.
    """
    if node is None or node.label == label:
        return node # برمی‌گرداند.
    elif node.label > label:
        return search(node.leftChild, label) # برو
    else: # node.label < label
        return search(node.rightChild, label) # برو

```

در صورتی که جستجو موفقیت‌آمیز باشد، تابع به ما رأس با مقدار مورد نظر را برمی‌گرداند اما در صورتی که جستجو موفقیت‌آمیز نباشد، None برمگردانده می‌شود.

اگر از ریشه شروع کنیم و به سمت بچه‌ی چپ هر رأس حرکت کنیم تا به رأسی مانند `v` برسیم که بچه‌ی چپ نداشته باشد، رأس `v` مقدار کمینه را در بین رأس‌های BST خواهد داشت. این به این دلیل است که در BST تمام مقادیر در زیردرخت چپ یک گره، از آن گره کوچکتر هستند.

```
In [3]: def findMin(node):
    """
    کوچکترین گره (با کمترین برچسب) را در زیردرخت مشخص شده پیدا می‌کند.

    Args:
        node (Node): گره‌ای که جستجو از آن شروع می‌شود.

    Returns:
        Node: گره‌ای با کمترین برچسب در زیردرخت.
    """
    if node.leftChild is None:
        return node # اگر گره فرزند چپ نداشت، خودش کمترین است.
    else:
        return findMin(node.leftChild) # به صورت بازگشتی به سمت چپ‌ترین فرزند برو.

def findMax(node):
    """
    بزرگترین گره (با بیشترین برچسب) را در زیردرخت مشخص شده پیدا می‌کند.

    Args:
        node (Node): گره‌ای که جستجو از آن شروع می‌شود.

    Returns:
        Node: گره‌ای با بیشترین برچسب در زیردرخت.
    """
    if node.rightChild is None:
        return node # اگر گره فرزند راست نداشت، خودش بیشترین است.
    else:
        return findMax(node.rightChild) # به صورت بازگشتی به سمت راست‌ترین فرزند برو.
```

# درج

برای درج یک عنصر جدید در BST، رویه‌ی بازگشتی زیر را در نظر می‌گیریم:

با شروع از ریشه، رأس  $v$  که در حال حاضر در آن هستیم را در نظر می‌گیریم. مقداری که می‌خواهیم درج کنیم را با برچسب رأس  $v$  مقایسه می‌کنیم. اگر از برچسب رأس  $v$  کوچکتر بود، پس جایش در زیردرخت سمت چپ  $v$  است و اگر مقدارش از برچسب رأس  $v$  بزرگتر بود، جایش در زیردرخت سمت راست  $v$  است. بسته به این مقایسه، به یکی از بچه‌های چپ و راست  $v$  می‌رویم و کار را ادامه می‌دهیم. این کار وقتی پایان می‌یابد که  $v$  مساوی **None** شود. به طور مثال فرض کنید به زیردرخت بچه سمت راست  $v$  برویم ولی  $v$  بچه‌ی سمت راست ندارد، در این صورت یک رأس با مقدار مورد نظرمان ایجاد می‌کنیم و آن را بچه‌ی سمت راست  $v$  قرار می‌دهیم.

[www.penjee.com](http://www.penjee.com)

```
In [4]: def insert(node, value):
    """
    درج می‌کند BST این تابع یک مقدار جدید را به صورت بازگشتی در
    آن قرار دارد.

    Args:
        node (Node): گره فعلی که در آن قرار داریم.
        value: مقداری که قرار است درج شود.

    """
    if value < node.label:
        if node.leftChild is not None:
            insert(node.leftChild, value) # اگر فرزند چپ وجود داشت، به صورت بازگشتی ادامه بده.
        else:
            node.leftChild = Node(value, node) # اگر فرزند چپ نداشت، گره جدید را به عنوان فرزند چپ اضافه کن (فرض می‌کنیم مقادیر برابر به سمت راست می‌روند).
    else: # value >= node.Label
        if node.rightChild is not None:
            insert(node.rightChild, value) # اگر فرزند راست وجود داشت، به صورت بازگشتی ادامه بده.
```

else:

node.rightChild = Node(value, node) # اگر فرزند راست نداشت، گره جدید را به عنوان فرزند راست اضافه کن.

نکته: اگر BST را به صورت میانترتیب (inorder) پیمایش کنیم، اعداد را به صورت مرتب شده بدست میآوریم. چرا؟

\*پاسخ:\*\* در پیمایش میانترتیب، ابتدا زیردرخت چپ (که شامل مقادیر کوچکتر است)، سپس گره فعلی، و در نهایت زیردرخت راست (که شامل مقادیر بزرگتر است) پیمایش میشود. به دلیل خاصیت BST که مقادیر چپ کوچکتر و مقادیر راست بزرگتر هستند، این ترتیب پیمایش دقیقاً عناصر را به صورت صعودی مرتب شده برمیگرداند.

In [5]:

```
def inOrderPrint(node):
    """
    پیمایش کرده و برحسب گرهها را چاپ میکند (Inorder Traversal) را به صورت میانترتیب BST این تابع
    ایجاد کرده و برحسب گرهها را چاپ میکند.
    اگر گره خالی بود، برگرد #.
    inOrderPrint(node.leftChild) # پیمایش زیردرخت چپ.
    چاپ برحسب گره فعلی #.
    inOrderPrint(node.rightChild) # پیمایش زیردرخت راست.

    # مثال ساخت و استفاده از
    root = Node(5, None) # ایجاد ریشه درخت با مقدار 5
    insert(root, 9)
    insert(root, 10)
    insert(root, 3)
    insert(root, 8)
    insert(root, 7)
    insert(root, 2)
    insert(root, 1)
    insert(root, 4)
    insert(root, 6)

    print("InOrder print of the tree:")
    inOrderPrint(root) # چاپ درخت به صورت میانترتیب

    print ("Searching value 8 in tree:", str(search(root, 8))) # جستجوی مقدار 8
    print ("Path to the min node in tree:", findMin(root)) # یافتن گره کمینه
    print ("Path to the max node in tree:", findMax(root)) # یافتن گره بیشینه
```

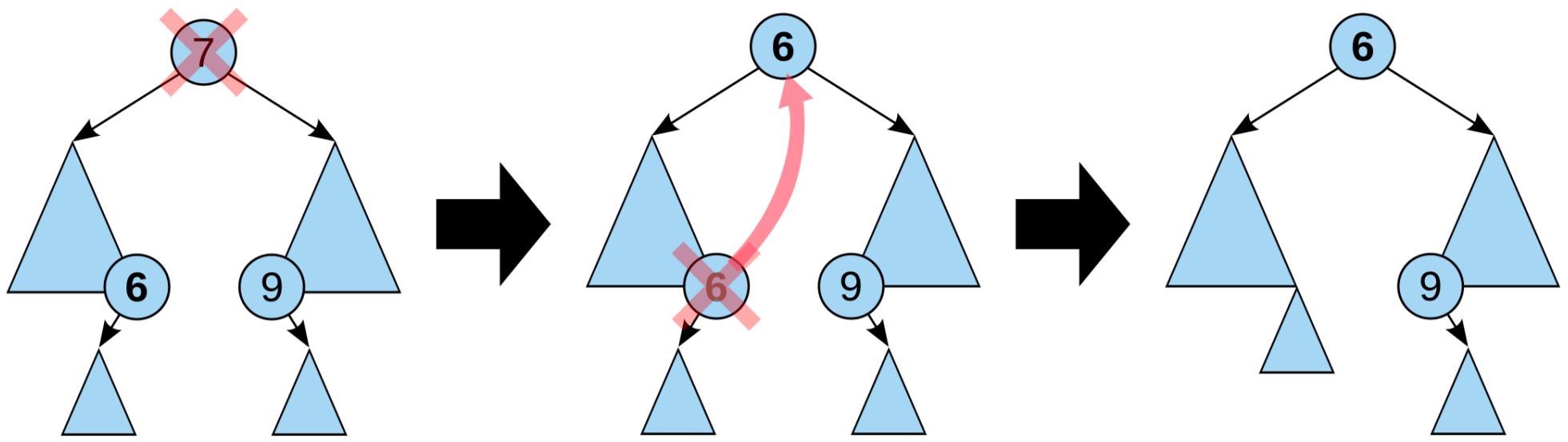
InOrder print of the tree:

```
1
2
3
4
5
6
7
8
9
10
Searching value 8 in tree: 8
Path to the min node in tree: 1
Path to the max node in tree: 10
```

## حذف

برای حذف یک مقدار از درخت، ابتدا رأس  $v$  دارای این مقدار را پیدا میکنیم. در این مرحله ۳ حالت اصلی پیش میآید که باید به دقت مدیریت شوند:

- **حالت ۱: رأس  $v$  بچه نداشته باشد (برگ باشد):** در این صورت، به سادگی آن را حذف میکنیم. یعنی اشارهگر پدرش به آن را **None** میکنیم.
- **حالت ۲: رأس  $v$  یک بچه داشته باشد:** در این حالت،  $v$  را حذف میکنیم و بچه اش را جایش قرار میدهیم. یعنی اشارهگر پدر  $v$  را به بچه‌ی  $v$  متصل میکنیم و اشارهگر پدر بچه‌ی  $v$  را به پدر  $v$  تغییر میدهیم.
- **حالت ۳: رأس  $v$  دو بچه داشته باشد:** در این صورت، رأس مینیمم در زیردرخت سمت راست  $v$  (یا رأس  $v$  ماکزیمم در زیردرخت سمت چپ  $v$ ) را، به نام  $u$  پیدا میکنیم. دقت کنید هرکدام از این رأس‌ها را که به جای  $v$  قرار دهیم و  $v$  را حذف کنیم، هنوز شرط لازم برای BST بودن درخت برقرار است. برحسب رأس  $u$  را در  $v$  قرار میدهیم و سپس رأس  $u$  را حذف میکنیم. از آنجا که  $u$  رأس مینیمم یا ماکزیمم یک زیردرخت است، حداکثر یک بچه خواهد داشت و میشود طبق ۲ حالت اول این رأس را حذف کرد.



```
In [6]: def replaceNodeInParent(oldNode, newNode, root_ref):
    """
    در والدش جایگزین میکند oldNode را با newNode با این تابع گره
    برای بروزرسانی ریشه سراسری استفاده میشود
    """
    if root_ref[0] == oldNode: # اگر گره قدیمی ریشه بود، ریشه را به بروزرسانی کن
        root_ref[0] = newNode
    if oldNode.parent: # اگر گره قدیمی پدر داشت (ریشه نبود)
        if oldNode == oldNode.parent.leftChild:
            oldNode.parent.leftChild = newNode # اگر فرزند چپ بود، اشارهگر چپ پدر را بروز کن
        else:
            oldNode.parent.rightChild = newNode # اگر فرزند راست بود، اشارهگر راست پدر را بروز کن
    if newNode:
        newNode.parent = oldNode.parent # پدر گره جدید را به پدر گره قدیمی تنظیم کن

def delete(node, value, root_ref):
    """
    حذف میکند BST این تابع یک مقدار را از
    """

Args:
    گره فعلی که جستجو از آن شروع میشود
    node (Node)
    مقداری که قرار است حذف شود
    value:
    لیستی که ریشه درخت را به عنوان تنها عنصر خود نگه میدارد
    root_ref (list):
    تا بتوان ریشه سراسری را در صورت نیاز ببروزرسانی کرد
    """
    if node is None:
        return # اگر گره خالی بود، کاری نکن

    if node.label < value:
        # اگر مقدار بزرگتر بود، به راست برو
        # باید مطمئن شویم که فرزند راست وجود دارد قبل از فراخوانی بازگشتی
        if node.rightChild:
            delete(node.rightChild, value, root_ref)
    elif node.label > value:
        # اگر مقدار کوچکتر بود، به چپ برو
        # باید مطمئن شویم که فرزند چپ وجود دارد قبل از فراخوانی بازگشتی
        if node.leftChild:
            delete(node.leftChild, value, root_ref)
    else: # گره برگ است (فرزنده ندارد)
        if node.leftChild is None and node.rightChild is None: # حالت 1: گره برگ است (فرزنده ندارد)
            replaceNodeInParent(node, None, root_ref)
        elif node.leftChild is None: # حالت 2: گره فقط فرزند راست دارد
            replaceNodeInParent(node, node.rightChild, root_ref)
        elif node.rightChild is None: # حالت 2: گره فقط فرزند چپ دارد
            replaceNodeInParent(node, node.leftChild, root_ref)
        else: # حالت 3: گره دو فرزند دارد
            # پیدا کردن کوچکترین گره در زیردرخت راست (جانشین میانی)
            newNode = findMin(node.rightChild)
            node.label = newNode.label # برجسب گره جانشین را به گره فعلی منتقل کن
            # حذف گره جانشین (که حالا برگ یا دارای یک فرزند است)
            # را از موقعیت اصلی اش حذف میکند
            # این بخش از کد اصلی شما نیاز به اصلاح دارد تا گره جانشین به درستی حذف شود
            # فراخوانی کنیم newNode را روی delete به جای دستکاری دستی پیوندها، بهتر است
            # (اما با توجه به ساختار فعلی، این روش جایگزین را استفاده میکنیم)
            if newNode.parent.leftChild == newNode: # فرزند چپ بدرس بود اگر
                newNode.parent.leftChild = newNode.rightChild # فرزند راستش را به جای آن قرار بده
            else: # فرزند راست بدرس بود اگر
                newNode.parent.rightChild = newNode.rightChild # فرزند راستش را به جای آن قرار بده
            if newNode.rightChild: # فرزند داشت، پدر آن فرزند را بروز کن
                newNode.rightChild.parent = newNode.parent

```

```
In [7]: # مثال ساخت و استفاده از BST
root_node_ref = [Node(5, None)] # ایجاد ریشه درخت با مقدار 5 در یک لیست برای امکان بروزرسانی ریشه سراسری
root = root_node_ref[0]

insert(root, 9)
insert(root, 10)
insert(root, 3)
insert(root, 8)
insert(root, 7)
insert(root, 2)
insert(root, 1)
insert(root, 4)
insert(root, 6)

print("InOrder print of the tree (Initial):")
inOrderPrint(root) # چاپ درخت به صورت میانترتیب

print("\nSearching value 8 in tree:", str(search(root, 8))) # جستجوی مقدار 8 در درخت
print("Path to the min node in tree:", findMin(root)) # بافتمن گره کمینه
```

```

print ("Path to the max node in tree:", findMax(root)) # یافتن گره بیشینه از تابع
# مثال استفاده از تابع delete
print("\n--- حذف گرهها ---")

# حذف یک برگ (مثال: حذف 1)
print("\n1: حذف")
delete(root, 1, root_node_ref)
root = root_node_ref[0] # به روزرسانی ریشه پس از حذف احتمالی ریشه
print("InOrder print after deleting 1:")
inOrderPrint(root)

# حذف گره با یک فرزند (مثال: حذف 2, که حالا برگ شده)
print("\n2: حذف")
delete(root, 2, root_node_ref)
root = root_node_ref[0]
print("InOrder print after deleting 2:")
inOrderPrint(root)

# حذف گره با دو فرزند (مثال: حذف 9)
print("\n9: حذف")
delete(root, 9, root_node_ref)
root = root_node_ref[0]
print("InOrder print after deleting 9:")
inOrderPrint(root)

# حذف ریشه (مثال: حذف 5)
print("\n5 (ریشه): حذف")
delete(root, 5, root_node_ref)
root = root_node_ref[0]
print("InOrder print after deleting 5 (new root):")
inOrderPrint(root)

# حذف یک گره ناموجود
print("100 (ناموجود): حذف")
delete(root, 100, root_node_ref)
root = root_node_ref[0]
print("InOrder print after trying to delete 100:")
inOrderPrint(root)

print("\n--- تست‌های جدید حذف ---")

ریشه را در یک لیست قرار می‌دهیم تا بتوانیم آن را در توابع تغییر دهیم # [ ]
root_list = [Node(2, None)]
insert(root_list[0], 1)
insert(root_list[0], 3)
print("Tree before deletion (InOrder):")
inOrderPrint(root_list[0])
چاپ درخت قبل از حذف # [ ] ( )

delete(root_list[0], 2, root_list) # مقدار 2
print("\nTree after deleting 2 (InOrder):")
inOrderPrint(root_list[0])
چاپ درخت بعد از حذف # [ ] ( )

# مثال دیگر برای تست حذف دو فرزند
root_list = [Node(50, None)]
insert(root_list[0], 30)
insert(root_list[0], 70)
insert(root_list[0], 20)
insert(root_list[0], 40)
insert(root_list[0], 60)
insert(root_list[0], 80)
print("\nTree before deleting 50 (InOrder):")
inOrderPrint(root_list[0])
delete(root_list[0], 50, root_list)
print("\nTree after deleting 50 (InOrder):")
inOrderPrint(root_list[0])

```

```
InOrder print of the tree (Initial):
1
2
3
4
5
6
7
8
9
10
```

```
Searching value 8 in tree: 8
Path to the min node in tree: 1
Path to the max node in tree: 10
```

--- حذف گرهای ---

```
1 حذف:
InOrder print after deleting 1:
2
3
4
5
6
7
8
9
10
```

```
2 حذف:
InOrder print after deleting 2:
3
4
5
6
7
8
9
10
```

```
9 حذف:
InOrder print after deleting 9:
3
4
5
6
7
8
10
```

```
(حذف 5 (ریشه):
InOrder print after deleting 5 (new root):
3
4
6
7
8
10
```

```
100 حذف (ناموجود):
InOrder print after trying to delete 100:
3
4
6
7
8
10
```

--- نسخهای جدید حذف ---

```
Tree before deletion (InOrder):
1
2
3
```

```
Tree after deleting 2 (InOrder):
1
3
```

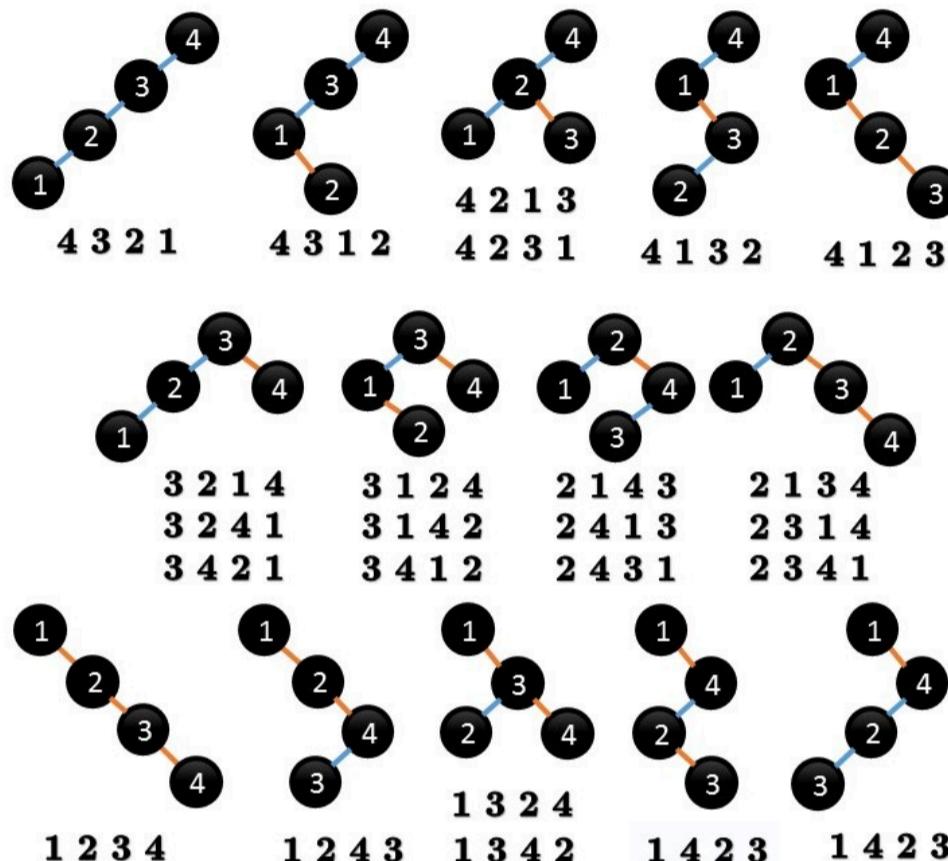
```
Tree before deleting 50 (InOrder):
20
30
40
50
60
70
80
```

```
Tree after deleting 50 (InOrder):
20
30
40
60
70
80
```

# میانگین ارتفاع درخت جستجو

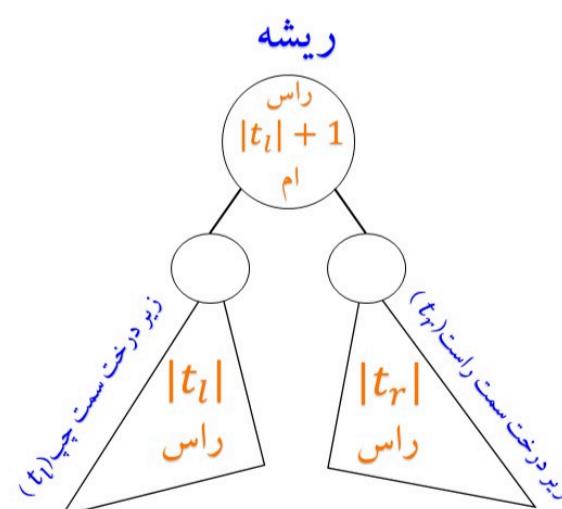
در اینجا به دادن شهود کلی بسنده می‌کنیم. (اثبات ریاضی دقیق آن را می‌توانید در کتاب دکتر قدسی مشاهده فرمایید.)

بدین منظور، تمامی جایگشت‌های ممکن را که می‌خواهیم در BST درج کنیم، توجه کنید عملیات درج هر جایگشتی هنگامی که به ترتیب و از ابتدای آن صورت بگیرد به دقیقاً یک BST یکتا می‌انجامد، به عنوان مثال شکل زیر تمامی ZBST‌های حاصل از درج اعداد ۱ تا ۴ را نشان می‌دهد:



همان طور که مشاهده می‌کنید، هر چه قدر BST متعادل‌تر باشد، تعداد بیشتری جایگشت با آن متناظر خواهد بود. این نکته کلیدی را به صورت زیر در حالت کلی بیان می‌کنیم:

تعداد جایگشت‌های متناظر با BST  $t$  را با  $p_t$  نشان می‌دهیم، و همچنین زیردرخت سمت راست آن را با  $t_r$  و زیردرخت سمت چپ آن را با  $t_l$  نشان می‌دهیم، به شکل زیر توجه کنید:



تعداد جایگشت‌هایی که با این درخت متناظر می‌شوند را می‌توان از فرمول زیر به دست آورد:

$$p_t = \underbrace{\left( \frac{|t_l| + |t_r|}{|t_l|} \right)}_{\uparrow} \cdot p_{t_l} \cdot p_{t_r}$$

حال در این فرمول، می‌توان نشان داد که هنگامی که  $|t_l|$  و  $|t_r|$  به هم نزدیک باشند، یعنی BST متعادل باشد، جزء انتخاب آن بسیار بزرگ تر از حالت خواهد بود که این دو از هم فاصله داشته باشند. بنابراین هر چه درخت متعادل‌تر باشد، تعداد جایگشت بیشتری با آن متناظر خواهد شد.

پس به صورت شهودی در حالت میانگین، با یک درخت متعادل، که مرتبه پیچیدگی ارتفاع آن از  $O(\lg n)$  است مواجه خواهیم شد.

در نمودار زیر هم که حاصل محاسبه میانگین ارتفاع درخت برای حالت‌های مختلف است می‌توانید ببینید که این مقدار عددی بین  $\lg n \times 3$  و  $\lg n \times 2$  است!

In [8]: وارد کردن کتابخانه‌های لازم

```
import math
import random
import numpy as np # برای استفاده از np.power
import matplotlib.pyplot as plt # برای رسم نمودار
```

باید قبلاً تعریف شده باشند و در دسترس باشند `insert` و تابع `Node` توجه: کلاس `Node` را مقداردهی اولیه می‌کند `BST` جدید برای `(Node)` متدهای سازنده: یک گره برحسب یا مقدار گره اشاره‌گر به گره پدر اشاره‌گر به فرزند چپ اشاره‌گر به فرزند راست

```
class Node:
    def __init__(self, label, parent):
        # را مقداردهی اولیه می‌کند BST جدید برای (Node) متدهای سازنده: یک گره
        self.label = label # برحسب یا مقدار گره
        self.parent = parent # اشاره‌گر به گره پدر
        self.leftChild = None # اشاره‌گر به فرزند چپ
        self.rightChild = None # اشاره‌گر به فرزند راست
```

```
def __str__(self):
    # فقط برحسب آن را برمی‌گرداند تا از نمایش رشته‌ای گره
    return str(self.label)
```

# تعریف تابع insert و استگی برای get\_avg\_height
def insert(node, value):

"""  
درج می‌کند BST این تابع یک مقدار جدید را به صورت بازگشته در.

Args:

گره فعلی که در آن قرار داریم.  
مقداری که قرار است درج شود.

"""

```
if value < node.label:
    if node.leftChild is not None:
        insert(node.leftChild, value) # اگر فرزند چپ وجود داشت، به صورت بازگشته ادامه بده
    else:
        node.leftChild = Node(value, node) # اگر فرزند چپ نداشت، گره جدید را به عنوان فرزند چپ اضافه کن
        (فرض می‌کنیم مقادیر برابر به سمت راست می‌روند)
else: # value >= node.label
    if node.rightChild is not None:
        insert(node.rightChild, value) # اگر فرزند راست وجود داشت، به صورت بازگشته ادامه بده
    else:
        node.rightChild = Node(value, node) # اگر فرزند راست نداشت، گره جدید را به عنوان فرزند راست اضافه کن
```

def find\_Height(node, height):

"""  
ارتفاع یک گره (زیردرخت) را به صورت بازگشته محاسبه می‌کند.

Args:

گره فعلی.  
ارتفاع فعلی از ریشه (برای فراخوانی اولیه 0.0).

Returns:

ارتفاع زیردرخت.

"""

```
if node is None: # اگر گره خالی بود، ارتفاع فعلی را برگردان
    return height
# ارتفاع زیردرخت چپ و راست را به صورت بازگشته پیدا می‌کند و حداثر را برگرداند
return max(find_Height(node.leftChild, height + 1.0), find_Height(node.rightChild, height + 1.0))
```

def get\_avg\_height(a):

"""  
گره، با ساخت چندین نمونه تصادفی محاسبه می‌کند 'a' را برای درختانی با BST میانگین ارتفاع.

Args:

a (int): تعداد گره‌ها در BST.

Returns:

میانگین ارتفاع.

"""

تعداد نمونه‌های تصادفی برای محاسبه میانگین #

avg = 0.0

```
for i in range(max_Sample_Size):
    # تولید یک لیست تصادفی از اعداد برای درج در BST.
    # اعداد بین 0 تا 15999 # هستند
    random_numbers = [random.randrange(16000) for _ in range(a)]
```

if not random\_numbers: # اگر لیست خالی بود، ادامه بده
 continue

# با اولین عدد تصادفی BST ساخت ریشه
root = Node(random\_numbers[0], None)

# درج بقیه اعداد تصادفی در BST.
for j in range(1, len(random\_numbers)):
 insert(root, random\_numbers[j]) # استفاده از تابع insert

اضافه کردن ارتفاع درخت فعلی به مجموع #

return avg / float(max\_Sample\_Size) # بردگرداندن میانگین ارتفاع

```

def plot2(N_values):
    """
    رسم میکند (n*lg(n) و 3*lg(n) را در مقایسه با 2 BST این تابع نمودار میانگین ارتفاع
    از تعداد گرهها برای محور X. لیستی از مقدار گرهها برای محور Y برجسته میگیرد.

    Args:
        N_values (list): مقدار گرهها برای محور X.
    """
    plt.figure(figsize=(15, 10))
    plt.xlabel("Number of Nodes") # برچسب محور X
    plt.ylabel("Height of BST") # برچسب محور Y

    # برای هر اندازه گره در BST میانگین ارتفاع N_values.
    y_height = [get_avg_height(a) for a in N_values]
    plt.plot(N_values, y_height, 'r', label='Height of BST', linewidth=5)

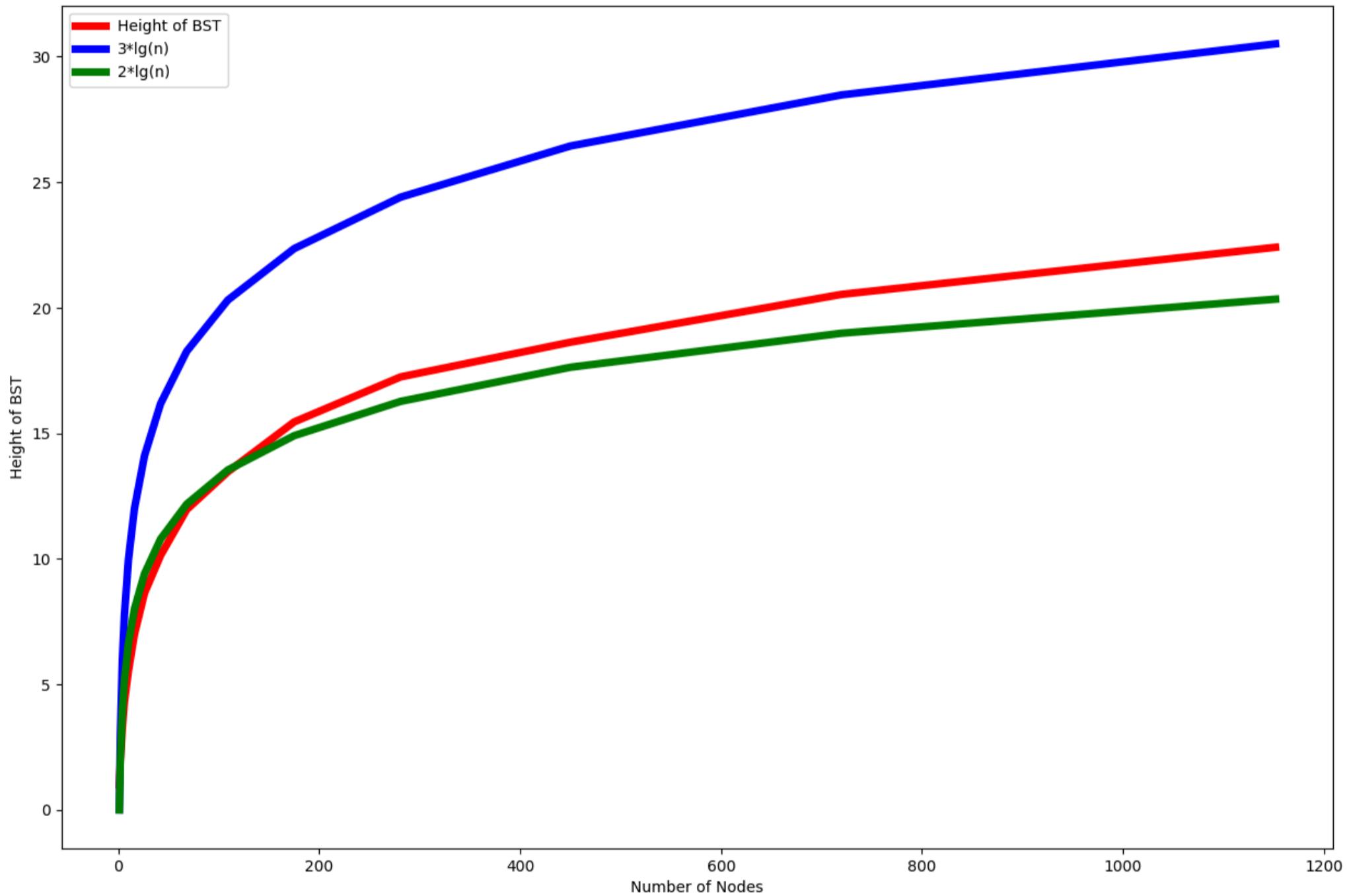
    # 3*lg(n) مقدار برای محاسبه مقادیر
    y_3lgn = [3 * math.log(a, 2) if a > 0 else 0 for a in N_values] # برای اطمینان از a > 0 Log.
    plt.plot(N_values, y_3lgn, 'b', label='3*lg(n)', linewidth=5)

    # 2*lg(n) مقدار برای محاسبه مقادیر
    y_2lgn = [2 * math.log(a, 2) if a > 0 else 0 for a in N_values] # برای اطمینان از a > 0 Log.
    plt.plot(N_values, y_2lgn, 'g', label='2*lg(n)', linewidth=5)

    plt.legend(loc=2) # نمایش راهنمایی در موقعیت بالا-چپ
    plt.show()

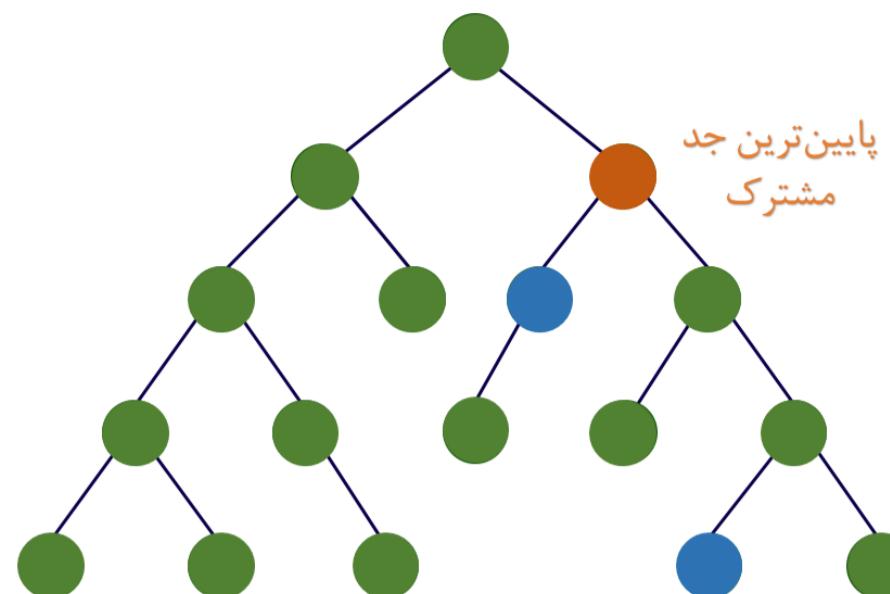
# برای فراخوانی plot2.
# np.power(base, exponent) - 1.6 در range(1, 16)
# .astype(int). برای تبدیل به عدد صحیح
N_for_plot = np.power(1.6, range(1, 16)).astype(int)
plot2(N_for_plot)

```



پایین‌ترین جد مشترک

در یک درخت ریشه‌دار، پایین‌ترین جد مشترک (LCA) دو گره را دورترین رأس از ریشه تعریف می‌کنیم که جد هر دو گره باشد. به عبارت دیگر، LCA گره‌ای است که در مسیر از ریشه به هر دو گره قرار دارد و عمیق‌ترین دورترین از ریشه) گره مشترک در این مسیرها است.



در اینجا به بررسی الگوریتم حل این مسئله برای یک BST می‌پردازیم.

همان‌طور که می‌دانیم تمامی گره‌ها در BST منحصر به فرد هستند. گره‌های سمت راست ریشه از آن بزرگ‌تر و گره‌های سمت چپ آن از آن کوچک‌تر هستند و این موضوع به حالت بازگشتی برای همه گره‌ها برقرار است.

\*\*الگوریتم حل LCA در BST:\*\*  
از ریشه شروع کرده و مقدار آن را با عدد داده شده ( $x$  و  $y$ ) مقایسه می‌کنیم;

- اگر مقدار ریشه از هردو ( $x$  و  $y$ ) بزرگ‌تر بود: این یعنی هر دو گره در زیردرخت چپ ریشه قرار دارند. پس به سراغ فرزند چپ ریشه می‌رویم و جستجو را ادامه می‌دهیم.
- اگر مقدار ریشه از هردو ( $x$  و  $y$ ) کوچک‌تر بود: این یعنی هر دو گره در زیردرخت راست ریشه قرار دارند. پس به سراغ فرزند راست ریشه می‌رویم و جستجو را ادامه می‌دهیم.
- در غیر این صورت (یعنی مقدار ریشه بین  $x$  و  $y$  قرار دارد، یا برابر با یکی از آن‌هاست): این گره فعلی (ریشه) همان LCA است. زیرا یکی از گره‌ها در زیردرخت چپ و دیگری در زیردرخت راست قرار می‌گیرد، یا یکی از گره‌ها خود ریشه است.

این الگوریتم بازگشتی را آنقدر ادامه می‌دهیم تا به گره‌ای برسیم که مقدار آن بین دو عدد داده شده مسئله باشد.

```
In [9]: def lca(root_node, x, y):
    """
    پیدا می‌کند BST را در یک y و x دو مقدار (LCA) این تابع پایین‌ترین جد مشترک
    Args:
        root_node (Node): ریشه BST.
        x: مقدار اولین گره.
        y: مقدار دومین گره.
    Returns:
        Node: y و x مقدار LCA گره‌ای که.
    """
    # وجود ندارد LCA، اگر ریشه خالی بود
    if root_node is None:
        return None
    # از برچسب گره فعلی کوچک‌تر باشند y و x اگر هر دو مقدار
    # در زیردرخت چپ قرار دارد
    if root_node.label > x and root_node.label > y:
        return lca(root_node.leftChild, x, y)
    # از برچسب گره فعلی بزرگ‌تر باشند y و x اگر هر دو مقدار
    # در زیردرخت راست قرار دارد
    elif root_node.label < x and root_node.label < y:
        return lca(root_node.rightChild, x, y)
    # در غیر این صورت (یکی در چپ و دیگری در راست، یا یکی از آن‌ها خود گره فعلی است)
    # است LCA گره فعلی همان
    else:
        return root_node # LCA را برگرداند
```

In [10]: # و تابع  $LCA$  مثال ساخت و استفاده از

```
# نمونه BST ساخت یک:
#      10
#     / \
#    -10   30
#   /   / \
#  8   25  60
# / \ / /
# 6  9 29 72 ...
root = Node(10, None)
insert(root, -10)
insert(root, 30)
insert(root, 60)
insert(root, 25)
insert(root, 72)
insert(root, 29)
insert(root, 8)
insert(root, 6)
insert(root, 9)

# چاپ برحسب گره LCA فراخوانی تابع
lca_node = lca(root, 29, 72)
if lca_node:
    print(f'LCA of 29 and 72 is: {lca_node.label}')
else:
    print("LCA not found (or tree is empty).")

# مثال‌های بیشتر:
lca_node = lca(root, 6, 9)
if lca_node: print(f'LCA of 6 and 9 is: {lca_node.label}')
lca_node = lca(root, -10, 8)
if lca_node: print(f'LCA of -10 and 8 is: {lca_node.label}')
lca_node = lca(root, 25, 29)
if lca_node: print(f'LCA of 25 and 29 is: {lca_node.label}')
lca_node = lca(root, 10, 60) # یکی از گره‌ها ریشه است
if lca_node: print(f'LCA of 10 and 60 is: {lca_node.label}')
lca_node = lca(root, 10, 25) # یکی از گره‌ها ریشه است
if lca_node: print(f'LCA of 10 and 25 is: {lca_node.label}')
lca_node = lca(root, 5, 15) # گره‌های ناموجود
if lca_node: print(f'LCA of 5 and 15 is: {lca_node.label}')
else: print("LCA of 5 and 15 not found (or values not in tree).")
```

```
LCA of 29 and 72 is: 30
LCA of 6 and 9 is: 8
LCA of -10 and 8 is: -10
LCA of 25 and 29 is: 25
LCA of 10 and 60 is: 10
LCA of 10 and 25 is: 10
LCA of 5 and 15 is: 10
```

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل پنجم، بخش سوم: داده‌ساختار درخت پیشوندی یا Trie

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

- مقدمه
- Trie
- Insert
- Find
- چند سوال

## مقدمه

اگر مجموع طول رشته‌های ذخیره شده در دیکشنری  $l_1$  و مجموع طول رشته‌های درخواست شده توسط کاربران  $l_2$  باشد، طراحی الگوریتمی با پیچیدگی زمانی  $O(l_1 \cdot l_2)$  ساده است. این پیچیدگی به این دلیل است که می‌توان هر رشته‌ی درخواست شده را با تمام رشته‌های موجود در دیکشنری مقایسه کرد. اما با استفاده از داده‌ساختارهای مناسب مانند \*Trie\*\*، می‌توان این مسئله را با پیچیدگی زمانی بهتری حل کرد. در ادامه، با داده‌ساختار \*Trie\*\* و عملیات‌های آن آشنا می‌شویم.

## داده‌ساختار Trie

داده‌ساختار \*Trie\*\* (درخت پیشوندی) یک درخت ریشه‌دار است که برای ذخیره و جستجوی رشته‌ها به کار می‌رود. این داده‌ساختار از یک راس ریشه شروع می‌شود که نشان‌دهنده رشته‌ی تهی است. هر راس می‌تواند چندین یال به فرزندان خود داشته باشد، و هر یال با یک حرف از الفبا برچسب‌گذاری شده است. این ساختار برای عملیات‌هایی مانند افزودن رشته (\*Insert\*\*) و جستجوی وجود یک رشته (\*Find\*\*) بسیار کارآمد است. دو عملیات اصلی این داده‌ساختار به شرح زیر است:

- افزودن یک رشته به .Trie
- بررسی وجود یک رشته در .Trie

در ادامه، جزئیات این عملیات‌ها را بررسی می‌کنیم.

# عملیات Insert

برای افزودن یک رشته‌ی  $S$  به Trie، از ریشه شروع می‌کنیم:

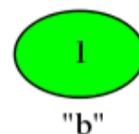
1. اگر یالی از راس فعلی وجود داشته باشد که با حرف اول  $S$  برچسب‌گذاری شده، به راس فرزند متناظر می‌رویم.

2. اگر چنین یالی وجود نداشته باشد، یک راس جدید می‌سازیم و آن را با یالی برچسب‌گذاری شده با حرف اول  $S$  به راس فعلی متصل می‌کنیم.

3. حرف اول  $S$  را حذف کرده و همین فرآیند را برای رشته‌ی باقی‌مانده تکرار می‌کنیم.

4. وقتی  $S$  تهی شود، راس فعلی را علامت‌گذاری می‌کنیم تا نشان دهیم یک رشته در این نقطه به پایان رسیده است.

مثال: فرض کنید بخواهیم رشته‌های aab، b، ab، aa، aaba را به ترتیب اضافه کنیم. مراحل این فرآیند در شکل زیر نشان داده شده است (راس فعلی



با رنگ سبز و رؤوس علامت‌گذاری شده با رنگ آبی).

```
In [1]: # برای رئوس درخت Node تعریف کلاس
class Node:
    def __init__(self):
        self.mark = False # آیا این راس پایان یک رشته است؟
        self.edges = [None] * 26 # تا a تا z لیست یالها برای حروف

# تابع افزودن رشته به Trie
def trie_insert(node, string, idx=0):
    # اگر به انتهای رشته رسیدیم، راس را علامت‌گذاری کن
    if idx == len(string):
        node.mark = True
        return
    # محاسبه اندیس حرف در الفبا (a=0, b=1, ...)
    e = ord(string[idx]) - ord('a')
    # اگر یال وجود ندارد، یک راس جدید بساز
    if node.edges[e] is None:
        node.edges[e] = Node()
    # به راس بعدی برو و ادامه رشته را پردازش کن
    trie_insert(node.edges[e], string, idx + 1)

# تابع جستجوی رشته در Trie
def trie_search(node, string, idx=0):
    # اگر به انتهای رشته رسیدیم، بررسی کن که راس علامت‌گذاری شده است یا نه
    if idx == len(string):
        return node.mark
    # محاسبه اندیس حرف در الفبا
    e = ord(string[idx]) - ord('a')
    # نیست اگر یال وجود ندارد، رشته در
    if node.edges[e] is None:
        return False
    # به راس بعدی برو و ادامه رشته را جستجو کن
    return trie_search(node.edges[e], string, idx + 1)

# تابع حذف رشته از Trie
def trie_delete(node, string, idx=0):
    # اگر به انتهای رشته رسیدیم، علامت راس را حذف کن
    if idx == len(string):
        node.mark = False
        return
    e = ord(string[idx]) - ord('a')
    trie_delete(node.edges[e], string, idx + 1)
```

```

if idx == len(string):
    if not node.mark:
        raise Exception("وجود ندارد رشته در Trie")
    node.mark = False
    return
# محاسبه اندیس حرف در الفبا
e = ord(string[idx]) - ord('a')
# نیست اگر یال وجود ندارد، رشته در
if node.edges[e] is None:
    raise Exception("وجود ندارد رشته در")
    به راس بعدی برو و ادامه رشته را حذف کن
    trie_delete(node.edges[e], string, idx + 1)

مثال استفاده
root = Node()
trie_insert(root, "hello") # افزودن رشته "hello"
trie_insert(root, "how") # افزودن رشته "how"
trie_insert(root, "what") # افزودن رشته "what"
print(trie_search(root, "how")) # True
print(trie_search(root, "hell")) # False
print(trie_search(root, "hello")) # True
trie_delete(root, "hello") # حذف رشته "hello"
print(trie_search(root, "hello")) # False
try:
    trie_delete(root, "hello")
except Exception as e:
    print(e)

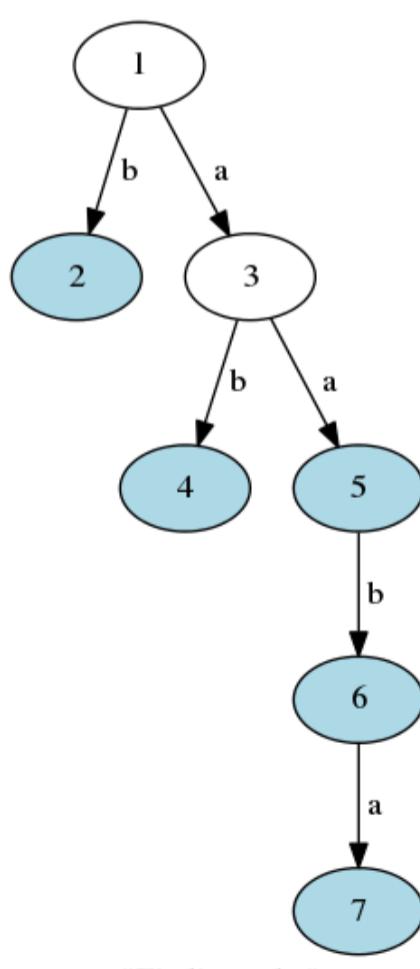
```

True  
False  
True  
False  
وجود ندارد Trie رشته در

# عملیات Find

برای جستجوی یک رشته‌ی  $S$  در Trie، از ریشه شروع می‌کنیم:

- اگر یالی با برچسب حرف اول  $S$  وجود داشته باشد، به راس فرزند متناظر می‌رویم.
- اگر چنین یالی وجود نداشته باشد، رشته در Trie وجود ندارد و `False` برمی‌گردانیم.
- حرف اول  $S$  را حذف کرده و همین فرآیند را برای رشته‌ی باقیمانده تکرار می‌کنیم.
- وقتی  $S$  تهی شود، بررسی می‌کنیم که آیا راس فعلی علامت‌گذاری شده است یا خیر. اگر علامت‌گذاری شده باشد، رشته در Trie وجود دارد؛ در غیر این صورت، وجود ندارد.



مثال: مراحل جستجوی برخی رشته‌ها در Trie ساخته شده در بخش Insert در شکل زیر نشان داده شده است.

```
In [2]: def trie_find(node, string, idx = 0):
    if idx == len(string):
        return node.mark
```

```

found = False
for edge in node.edges:
    if edge.label == string[idx]:
        found = True
return trie_find(edge.end, string, idx + 1)
if not found:
    return False

```

In [3]: برای مدیریت عملیات‌ها `Trie` تعریف کلاس #

```

class Trie:
    def __init__(self):
        self.root = Node() # ریشه درخت Trie

    def insert(self, string):
        # افزودن رشته به Trie
        trie_insert(self.root, string, 0)

    def find(self, string):
        # جستجوی رشته در Trie
        return trie_search(self.root, string, 0)

# مثال استفاده
trie = Trie()
shahrzad_strings = ['b', 'ab', 'aa', 'aab', 'aab'] # رشته‌های شهرزاد
farhad_strings = ['aab', 'aac', 'c', 'a', 'aab', 'ab', 'b', 'ba'] # رشته‌های فرهاد

# افزودن رشته‌های شهرزاد
for string in shahrzad_strings:
    trie.insert(string)

# جستجوی رشته‌های فرهاد
for string in farhad_strings:
    print(f" وجود دارد؟ {string} در Trie در '{string}' رشته {trie.find(string)}")

```

وجود دارد؟ 'aab' در Trie در 'aab' رشته True  
 وجود دارد؟ 'aac' در Trie در 'aac' رشته False  
 وجود دارد؟ 'c' در Trie در 'c' رشته False  
 وجود دارد؟ 'a' در Trie در 'a' رشته False  
 وجود دارد؟ 'aab' در Trie در 'aab' رشته True  
 وجود دارد؟ 'ab' در Trie در 'ab' رشته True  
 وجود دارد؟ 'b' در Trie در 'b' رشته True  
 وجود دارد؟ 'ba' در Trie در 'ba' رشته False

## سوال‌ها

- ۱- پیچیدگی زمانی: پیچیدگی زمانی عملیات‌های Insert و Find چیست؟ پاسخ: هر دو عملیات Insert و Find به ازای یک رشته‌ی با طول  $L$ , پیچیدگی

زمانی ( $O(L)$ ) دارند، زیرا در هر مرحله یک حرف از رشته پردازش می‌شود و برای هر حرف، یک یال بررسی یا ساخته می‌شود. این کارایی بالا به دلیل ساختار درختی Trie است که امکان پیمایش مستقیم بر اساس حروف را فراهم می‌کند.

- ۲- تعداد یال‌ها و رئوس: تعداد یال‌ها و رئوس در Trie چقدر است؟ پیچیدگی حافظه چطور است؟ پاسخ: اگر  $N_{nodes}$  تعداد کل رئوس منحصر به

فرد در Trie باشد (که حداقل برابر با مجموع طول تمام رشته‌های ذخیره شده به اضافه یک برای ریشه است)، تعداد رئوس حداقل  $N_{nodes}$  و تعداد یال‌ها حداقل  $N_{nodes}-1$  است. پیچیدگی حافظه مصرفی در بدترین حالت  $O(N_{nodes} \times \Sigma)$  است، که  $\Sigma$  اندازه الفبا (در اینجا 26 برای حروف کوچک انگلیسی) است. این به این دلیل است که هر گره یک آرایه به اندازه الفبا برای اشاره‌گرهای فرزندان خود نگهداری می‌کند.

- ۳- استفاده از آرایه برای یال‌ها: اگر برای ذخیره یال‌های هر راس از یک آرایه به طول ۲۶ (برای حروف الفبای انگلیسی) استفاده کنیم، چه تأثیری بر

پیچیدگی زمانی و حافظه‌ای دارد؟ پاسخ: استفاده از آرایه به طول ۲۶ باعث می‌شود دسترسی به یال‌ها در  $O(1)$  انجام شود (یعنی دسترسی مستقیم با استفاده از اندیس حرف). این روش دسترسی به فرزندان را بسیار سریع و تضمین شده می‌کند. اما از نظر حافظه، این کار منجر به افزایش مصرف حافظه می‌شود، زیرا برای هر راس، حتی اگر یال‌های کمی داشته باشد، ۲۶ خانه (اشاره‌گر) در نظر گرفته می‌شود. بنابراین، حافظه‌ی مصرفی به  $O(N_{nodes} \times \Sigma)$  افزایش می‌یابد که تعداد رئوس  $N_{nodes}$  و  $\Sigma$  اندازه الفبا است.

# مثال: بررسی پیشوند بودن رشته

فرض کنید بخواهیم بررسی کنیم که آیا یک رشته‌ی داده شده پیشوند یک کلمه در دیکشنری است یا خیر. برای این منظور، باید تغییراتی در Trie ایجاد کنیم. ایده این است که تمام رئوس در مسیر یک رشته را علامت‌گذاری کنیم، نه فقط راس پایانی. این کار باعث می‌شود هر راسی که علامت‌گذاری شده، نشان‌دهنده‌ی پیشوند بودن رشته‌ی متناظر باشد.

## پاسخ

در حل بسیاری از مسائل که با استفاده از داده‌ساختار Trie حل می‌شوند لازم است مقداری عملیات‌ها را تغییر دهیم تا با خواسته‌ی ما سازگار شوند. برای مثال با تغییری اندک در پیاده‌سازی استاندارد این داده‌ساختار، می‌توانیم این مسئله را حل کنیم.

در تابع insert، فقط زمانی راس را علامت می‌زدیم که رشته تهی باشد و به عبارت دیگر به آخرین راس رسیده باشیم. برای حل این مسئله تنها کافی است تمامی رئوس مسیر را علامت بزنیم.

```
In [4]: تابع بھبودیافته برای افزودن رشته با علامت‌گذاری تمام رئوس مسیر
def trie_insert_mark_all(node, string, idx=0):
    علامت‌گذاری راس فعلی به عنوان پیشوند
    اگر به انتهای رشته رسیدیم
        return
    e = ord(string[idx]) - ord('a') # محاسبه اندیس حرف
    اگر یال وجود ندارد، راس جدید بساز
    if node.edges[e] is None:
        node.edges[e] = Node()
    ادامه با راس بعدی
    trie_insert_mark_all(node.edges[e], string, idx + 1) # ادامه با راس بعدی

# در کلاس insert جایگزینی تابع
Trie.insert = lambda self, string: trie_insert_mark_all(self.root, string, 0)

# مثال استفاده
trie = Trie()
shahrzad_strings = ['b', 'ab', 'aa', 'aab', 'aab'] # رشته‌های شهرزاد
farhad_strings = ['aab', 'aac', 'c', 'a', 'aab', 'ab', 'b', 'ba'] # رشته‌های فرهاد

# افزودن رشته‌های شهرزاد
for string in shahrzad_strings:
    trie.insert(string)

# بررسی پیشوند بودن رشته‌های فرهاد
for string in farhad_strings:
    print(f"{'' است؟' {string} ' پیشوند یک کلمه در ' {trie.find(string)} ''")
```

است؟' aaba' پیشوند یک کلمه در ' True  
است؟' aac' پیشوند یک کلمه در ' False  
است؟' b' پیشوند یک کلمه در ' False  
است؟' c' پیشوند یک کلمه در ' True  
است؟' a' پیشوند یک کلمه در ' True  
است؟' ab' پیشوند یک کلمه در ' True  
است؟' aab' پیشوند یک کلمه در ' True  
است؟' ba' پیشوند یک کلمه در ' True  
است؟' b' پیشوند یک کلمه در ' True  
است؟' ba' پیشوند یک کلمه در ' False

# مثال: یافتن حداقل XOR در آرایه

فرض کنید آرایه‌ای از اعداد حداقل ۳۰ بیتی داریم و می‌خواهیم برای هر عدد، عددی دیگر در آرایه پیدا کنیم که حاصل XOR آن با عدد موردنظر حداقل شود.

## پاسخ

برای حل این مسئله، از یک Trie باینری استفاده می‌کنیم:

1. تمام اعداد را به رشته‌های باینری ۳۰ بیتی تبدیل می‌کنیم (با افزودن صفرهای ابتدایی در صورت نیاز).

2. هر عدد را به صورت یک رشته‌ی باینری از بیت پارازش (MSB) به Trie اضافه می‌کنیم.

3. برای هر عدد  $x$ ، به دنبال عددی می‌گردیم که نقیض آن باشد. یعنی اگر بیت  $x$  صفر باشد، ترجیحاً یال ۱ را انتخاب می‌کنیم و اگر ۱ باشد، یال ۰ را

انتخاب می‌کنیم. اگر یال موردنظر وجود نداشته باشد، یال دیگر را انتخاب می‌کنیم.

4. این فرآیند را تا آخرین بیت ادامه می‌دهیم تا عددی با حداقل XOR پیدا شود.

چرا این روش جواب می‌دهد؟ با انتخاب بیت‌های نقيض در هر مرحله، مقدار XOR را حداقل می‌کنیم، زیرا بیت‌های متفاوت (۰ و ۱) در XOR مقدار ۱ تولید

می‌کنند که باعث افزایش مقدار نهایی می‌شود. تحلیل پیچیدگی زمانی:

- افزودن هر عدد به Trie یا به طور کل  $O(k)$  که  $k$  تعداد بیت‌هاست.
- جستجوی حداقل XOR برای هر عدد:  $O(30)$ .
- برای  $n$  عدد در آرایه، کل پیچیدگی زمانی  $O(n \cdot k)$  است.
- پیچیدگی حافظه:  $O(n \cdot k)$  برای ذخیره Trie.

```
In [5]: # تعریف Trie برای مسئله XOR
class BinaryNode:
    def __init__(self):
        self.children = [None, None] # ۰ و ۱
        self.value = None # مقدار ذخیره شده در راس پایانی

# پاینری افزودن عدد به Trie
def insert_binary_trie(root, num, bits=30):
    node = root
    for i in range(bits - 1, -1, -1): # از بیت پر ازش به کم ارزش
        bit = (num >> i) & 1 # استخراج بیت
        if node.children[bit] is None:
            node.children[bit] = BinaryNode()
        node = node.children[bit]
    node.value = num # ذخیره عدد در راس پایانی

# یافتن عددی با حداقل XOR
def find_max_xor(root, num, bits=30):
    node = root
    for i in range(bits - 1, -1, -1): # بیت فعلی عدد
        bit = (num >> i) & 1
        opposite_bit = 1 - bit # بیت نقيض
        if node.children[opposite_bit] is not None:
            node = node.children[opposite_bit] # ترجیحاً نقيض را انتخاب کن
        else:
            node = node.children[bit] # اگر نقيض نبود، بیت فعلی
    return node.value # را می‌دهد XOR عددی که حداقل
```

# مثال استفاده
numbers = [3, 10, 5, 25, 2, 8] # آرایه اعداد
root = BinaryNode()
for num in numbers:
 insert\_binary\_trie(root, num) # افزودن اعداد به Trie
for num in numbers:
 max\_xor = find\_max\_xor(root, num) # یافتن حداقل XOR
 print(f"عدد {num}: حداقل XOR با {max\_xor} = {num ^ max\_xor}")

In [ ]:

3: حداقل XOR 26 = 25 با  
10: حداقل XOR 19 = 25 با  
5: حداقل XOR 28 = 25 با  
25: حداقل XOR 28 = 5 با  
2: حداقل XOR 27 = 25 با  
8: حداقل XOR 17 = 25 با

## داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

۱۴۰۴-۱۴۰۵ سال تحصیلی اول ترم

## فصل پنجم، بخش چهارم: هرم

استاد: دکتر امین اسکندری

نیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست

- مقدمه
  - معرفی هر
  - عملیات ها
  - پیاده سازی
  - روش های
  - کاربردهای

مقدمة

یکی از مهمترین داده‌ساختارهایی که در این درس فرمائی‌گیریم، هرم است. هرم یک داده‌ساختار درختی است که برای پیاده‌سازی کارآمد صفاتی اولویت و الگوریتم‌های مرتب‌سازی خاصی مانند Heapsort به کار می‌رود. در این نوشته ابتدا این داده‌ساختار را معرفی می‌کنیم، سپس با پیاده‌سازی‌ها و کاربردهای آشنا خواهیم شد.

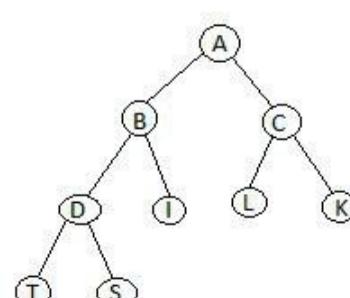
معرفی هرم

اورژانس یک بیمارستان را در نظر بگیرید که همیشه می‌خواهد به اورژانسی‌ترین بیمار رسیدگی کند. برای این کار باید بتواند به صورت سریع اورژانسی‌ترین بیمار را شناسایی کند. همچنین باید بتواند افراد جدیدی که وارد می‌شوند را سریعاً در محل خود در صف قرار دهد. این سناریو یک مثال کلاسیک از نیاز به یک صف اولویت (Priority Queue) است.

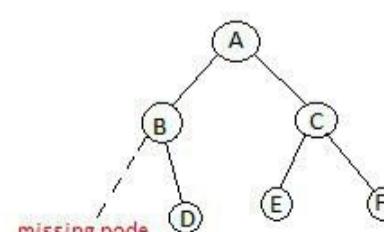
## تمرين ١:

کار خواسته شده (شناسایی سریع اورژانسی ترین بیمار و قرار دادن سریع افراد جدید در صف) را با آرایه مرتب شده انجام دهید و پیچیدگی زمانی عملیات‌های گفته شده را پیابید. (جواب در جدول ۱ آمده است)

درخت کامل درختی است که تمام سطوح درخت به غیر از احتمالاً آخرین سطح پر بوده و برگ‌های سطح آخر از چپ به راست قرار گرفته‌اند.



## Complete Binary Tree



### In-Complete Binary Tree

شکل ۱: تفاوت درخت کامل و ناکامل

## تذکر:

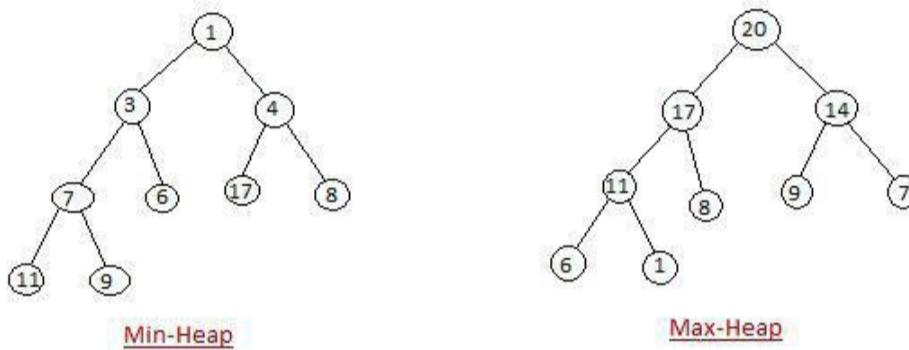
ساختار درختی هرم لزومی ندارد دودویی باشد ولی در این درس نوع دودویی آن را بررسی و استفاده می‌کنیم.

انواع دیگر هرم مانند هرم چندجمله‌ای (Fibonacci Heap) و هرم فیبوناچی (Binomial Heap) را بررسی کنید. این هرم‌ها ساختارهای پیچیده‌تری دارند و برای کاربردهای خاصی بهینه‌سازی شده‌اند.

با توجه به این که هرم درختی دودویی و کامل است، می‌توان نتیجه گرفت که ارتفاع آن همیشه برابر  $O(\log n)$  است، که  $n$  تعداد گره‌ها در هرم است. این ویژگی کلید کارایی بالای عملیات‌های هرم است.

هرم‌ها بر حسب رابطه بین رئوس با فرزندانشان دو نوع هستند:

- **هرم بیشینه (Max-Heap):** ارزش هر رأس از ارزش بچه‌هایش بیشتر است. بنابراین، بزرگ‌ترین عنصر همیشه در ریشه قرار دارد.
- **هرم کمینه (Min-Heap):** ارزش هر رأس از ارزش بچه‌هایش کمتر است. بنابراین، کوچک‌ترین عنصر همیشه در ریشه قرار دارد.



شکل ۲: مثال هرم کمینه و بیشینه

## عملیات‌های هرم

هرم بیشینه از عملیات زیر پشتیبانی می‌کند (عملیات هرم کمینه مشابه است، فقط مقایسه‌ها برعکس می‌شوند):

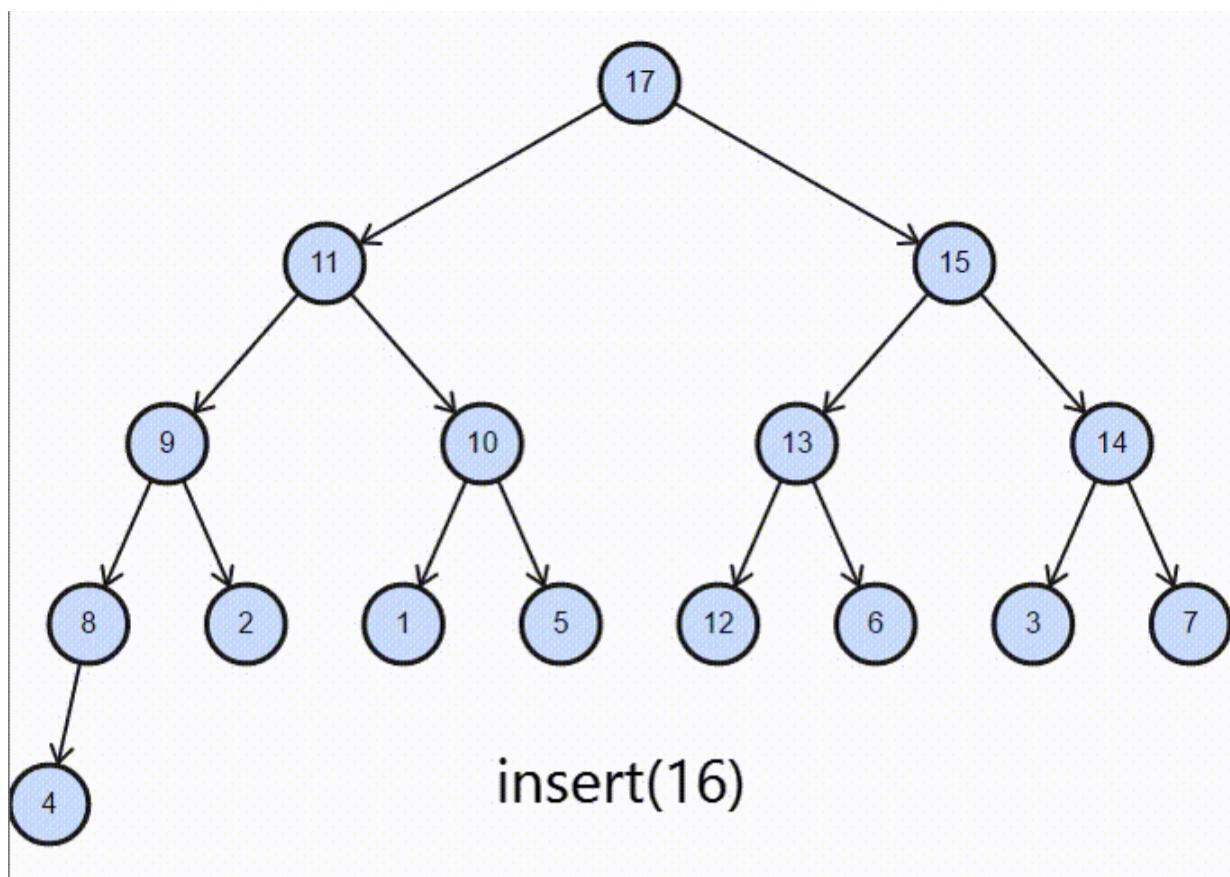
- **درج (Insert):** اضافه کردن یک عنصر جدید به هرم.
- **پیدا کردن عضو بیشینه (Get Max):** دسترسی به بزرگ‌ترین عنصر در هرم.
- **حذف عضو بیشینه (Delete Max):** حذف بزرگ‌ترین عنصر از هرم.

## درج در هرم:

برای درج یک عضو در هرم، ابتدا آن را در اولین مکان خالی (با حفظ خاصیت کامل بودن درخت) قرار می‌دهیم. این مکان معمولاً در انتهای آرایه پیاده‌سازی هرم است. سپس با تعدادی عمل جابجایی (bubble-up) عضو جدید را به مکان درستش منتقل می‌کنیم تا خاصیت هرم (Max-Heap Property) حفظ شود.

در bubble-up، عضو مورد نظر را با پدر خود مقایسه می‌کنیم و در صورتی که از آن بزرگ‌تر بود، جای آن دو با هم عوض می‌کنیم. این کار را آنقدر ادامه می‌دهیم تا عضو مورد نظر از پدر خود کوچک‌تر شود، یا آنکه خودش ریشه درخت شود.

چون هر عمل جابجایی عضو جدید را یک سطح در درخت بالاتر می‌برد، **bubble-up** حداقل به اندازه ارتفاع درخت یا همان  $O(\log n)$  طول خواهد کشید. پس پیچیدگی درج از  $O(\log n)$  است.



شکل ۳: درج در هرم

## یافتن عضو بیشینه در هرم:

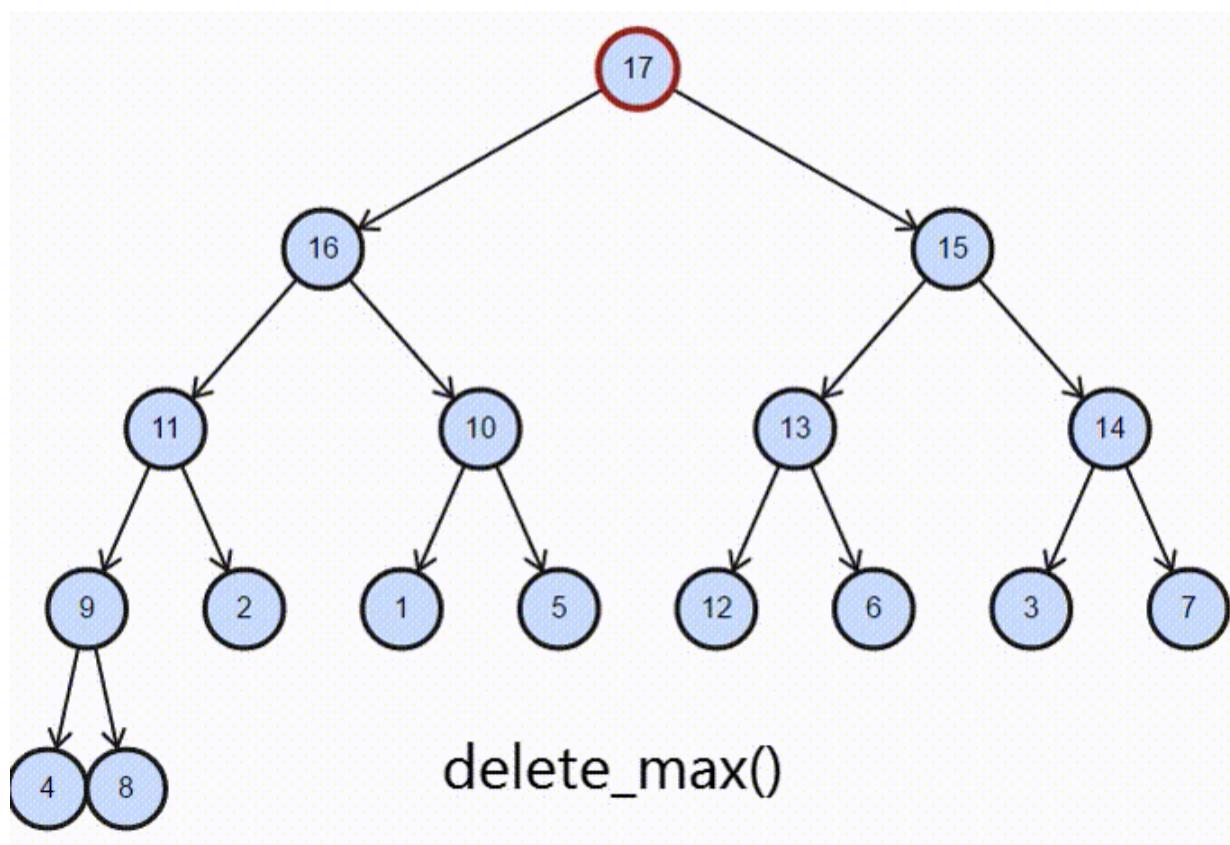
ریشه هرم بیشینه همیشه عضو بیشینه را در خود نگه می‌دارد. بنابراین، می‌توان در  $O(1)$  به آن دسترسی پیدا کرد.

## حذف بیشینه در هرم:

برای حذف عضو بیشینه در هرم، ریشه را از درخت حذف می‌کنیم و آخرین عضو هرم (راستترین برگ در پایین‌ترین عمق) را در جایگاه ریشه قرار می‌دهیم. این کار باعث می‌شود خاصیت کامل بودن درخت حفظ شود، اما ممکن است خاصیت هرم نقض شود. سپس با تعدادی عمل جابجایی (**heapify-down** یا **bubble-down**) این عضو جدید را به مکان درست منتقل می‌کنیم تا خاصیت هرم دوباره برقرار شود.

در **bubble-down**، عضو مورد نظر را با بزرگ‌ترین فرزند خود مقایسه می‌کنیم و در صورتی که از آن کمتر بود، جای آن دو را با هم عوض می‌کنیم. این کار را آنقدر ادامه می‌دهیم تا عضو مورد نظر از هر دو فرزند خود بزرگ‌تر شود، یا آنکه خودش برگ درخت شود.

چون هر عمل جابجایی عضو جدید را یک سطح در درخت پایین‌تر می‌برد، **bubble-down** حداقل به اندازه ارتفاع درخت یا همان  $O(\log n)$  طول خواهد کشید. پس پیچیدگی حذف از  $O(\log n)$  است.



شکل ۴: حذف ماقزیم در هرم

هرم	آرایه مرتب شده	عمل
$O(\log n)$	$O(n)$	درج
$O(1)$	$O(1)$	یافتن عضو بیشینه
$O(\log n)$	$O(1)$	حذف عضو بیشینه

جدول ۱: پیچیدگی زمانی عملیات مختلف در آرایه مرتب شده و هرم

## پیاده‌سازی هرم

هرم را می‌توان با استفاده از آرایه پیاده‌سازی کرد. به این صورت که اگر یک عضو در خانه  $i$ ام آرایه قرار گرفت (با فرض اینکه اندیس‌ها از ۱ شروع می‌شوند:  $i = 1, 2, \dots, n$ ), فرزندان چپ و راستش (در صورت وجود) به ترتیب در خانه‌های  $2i + 1$  و  $2i + 2$  قرار می‌گیرند. بهوضوح پدر یک رأس در خانه  $\left\lfloor \frac{i}{2} \right\rfloor$  خواهد بود.

با این روش، پیاده‌سازی هرم با حافظه  $O(n)$  امکان‌پذیر است، زیرا آرایه به طور مستقیم عناصر را ذخیره می‌کند و نیازی به اشاره‌گرهای اضافی برای ساختار درختی نیست.

### تذکر:

در بسیاری از پیاده‌سازی‌ها همچون پیاده‌سازی بالا، برای سهولت کار با اندیس‌ها، خانه صفر آرایه را خالی می‌گذارند. این کار باعث می‌شود محاسبات اندیس فرزندان و پدر ساده‌تر شود (مثلاً  $2i + 1$  به جای  $2i + 2$ ). می‌توانید به عنوان تمرین، هرم را با آرایه‌ای که از خانه صفر پر می‌شود پیاده‌سازی کنید و نحوه تغییر محاسبات اندیس‌ها را بررسی نمایید.

```
In [1]: def parent(i):
    # را برمی‌گرداند i این تابع اندیس پدر گره‌ای با اندیس.
    # فرض می‌شود اندیس‌ها از 1 شروع می‌شوند.
    return i // 2

def left_child(i):
    # را برمی‌گرداند i این تابع اندیس فرزند چپ گره‌ای با اندیس.
    # فرض می‌شود اندیس‌ها از 1 شروع می‌شوند.
    return 2 * i

def right_child(i):
    # را برمی‌گرداند i این تابع اندیس فرزند راست گره‌ای با اندیس.
    # فرض می‌شود اندیس‌ها از 1 شروع می‌شوند.
    return 2 * i + 1

class Max_Heap:
    def __init__(self):
        # متدهای سازنده: یک هرم بیشینه خالی ایجاد می‌کند
        self.heap = [0] # خانه صفر آرایه خالی گذاشته می‌شود تا اندیس‌ها از 1 شروع شوند.

    def size(self):
        # تعداد عناصر واقعی در هرم را برمی‌گرداند.
        return len(self.heap) - 1

    def bubble_down(self, ind):
        # به سمت پایین هرم حرکت می‌دهد 'ind' این تابع یک عنصر را از موقعیت حفظ شود.
        # تا زمانی که فرزند چپ وجود داشته باشد.
        while left_child(ind) <= self.size():
            newInd = ind # اندیس گره‌ای که باید با آن جایجا شود.

            # را به روزرسانی کن: اگر فرزند چپ بزرگتر از گره فعلی بود.
            if self.heap[left_child(ind)] > self.heap[ind]:
                newInd = left_child(ind)

            # را به روزرسانی کن: بود (یا فرزند چپ ind) اگر فرزند راست وجود داشت و بزرگتر از بزرگترین.
            if right_child(ind) <= self.size() and self.heap[right_child(ind)] > self.heap[newInd]:
                newInd = right_child(ind)

            # اگر گره فعلی بزرگتر از فرزندانش بود، نیازی به جایگایی نیست.
            if ind == newInd:
                break

            # جایگایی گره فعلی با بزرگترین فرزندش.
            self.heap[ind], self.heap[newInd] = self.heap[newInd], self.heap[ind]
            ind = newInd # به روزرسانی اندیس برای ادامه bubble_down.
```

```

def bubble_up(self, ind):
    به سمت بالای هرم حرکت می‌دهد 'ind' این تابع یک عنصر را از موقعیت
    # تا خاصیت هرم حفظ شود.
    while ind > 1 and self.heap[ind] > self.heap[parent(ind)]: # تا زمانی که گره ریشه نباشد و از پدرش بزرگتر باشد
        جایگاهی گره فعلی با پدرش.
        self.heap[ind], self.heap[parent(ind)] = self.heap[parent(ind)], self.heap[ind]
        ind = parent(ind) # به روزرسانی اندیس برای ادامه bubble_up.

def insert(self, item):
    یک عنصر جدید را به هرم اضافه می‌کند.
    اضافه کردن عنصر به انتهای آرایه # برای قرار دادن عنصر در جایگاه صحیح انجام # (())
    self.heap.append(item)
    self.bubble_up(self.size()()) # به روزرسانی اندیس برای ادامه bubble_up.

def get_max(self):
    بزرگترین عنصر در هرم را برمی‌گرداند (ریشه هرم).
    if self.size() == 0:
        خطأ در صورت خالی بودن هرم # raise Exception("Heap is empty")
        بزرگترین عنصر در اندیس 1 قرار دارد # return self.heap[1]

def del_max(self):
    بزرگترین عنصر را از هرم حذف کرده و آن را برمی‌گرداند.
    if self.size() == 0:
        خطأ در صورت خالی بودن هرم # raise Exception("Heap is empty")
        بزرگترین عنصر را ذخیره می‌کند.
    self.heap[1] = self.heap[-1] # آخرین عنصر را به جای ریشه قرار می‌دهد
    del(self.heap[-1]) # حذف آخرین عنصر از آرایه
    self.bubble_down(1) # برای ریشه جدید انجام bubble_down
    return MAX # بزرگترین عنصر حذف شده را برمی‌گرداند.

def build_heap_with_bubble_up(self, L):
    می‌سازد bubble_up با استفاده از روش L هرم را از یک لیست
    این روش شبیه درج تک تک عناصر در یک هرم خالی است
    (خانه 0 خالی) L مقداردهی اولیه هرم با لیست [0] + L.
    for i in range(1, self.size() + 1):
        از اندیس 1 تا انتهای هرم # را انجام می‌دهد
        self.bubble_up(i) # برای هر عنصر bubble_up

def build_heap_with_bubble_down(self, L):
    می‌سازد bubble_down با استفاده از روش L هرم را از یک لیست
    این روش کارآمدتر است و از آخرین گره غیربرگ شروع می‌کند
    self.heap = [0] + L # (خانه 0 خالی) L مقداردهی اولیه هرم با لیست
    شروع از آخرین گره غیربرگ تا ریشه
    for i in range(self.size() // 2, 0, -1):
        را انجام می‌دهد
        self.bubble_down(i) # برای هر گره bubble_down

def clear(self):
    هرم را خالی می‌کند.
    self.heap = [0]

```

```

In [2]: # مثال استفاده از کلاس Max_Heap
heap = Max_Heap()
ایجاد یک شیء هرم بیشینه # ().

# (روش کارآمدتر) ساخت هرم از یک لیست با استفاده از build_heap_with_bubble_down.
heap.build_heap_with_bubble_down([1, 10, 9, 4, 7, 11, 2, 3, 6, 5])
print("وضعیت هرم پس از ساخت (آرایه داخلی)", heap.heap)

```

```

print("حذف بزرگترین عنصر حذف شده()", heap.del_max()) # (11)
heap.insert(8) # درج عنصر 8
print("وضعیت هرم پس از درج 8", heap.heap)

print("بزرگترین عنصر حذف شده()", heap.del_max())
print("وضعیت نهایی هرم", heap.heap)
print("اندازه هرم", heap.size())

```

```

وضعیت هرم پس از ساخت (آرایه داخلی): [0, 11, 10, 9, 4, 3, 2, 1, 7, 6, 9, 1, 2, 6, 10, 11, 4, 5]
بزرگترین عنصر حذف شده: 11
وضعیت هرم پس از درج 8: [0, 10, 8, 9, 2, 1, 7, 6, 3, 4, 11, 0, 5]
بزرگترین عنصر حذف شده: 10
بزرگترین عنصر حذف شده: 9
بزرگترین عنصر حذف شده: 8
بزرگترین عنصر حذف شده: 7
وضعیت نهایی هرم: [0, 6, 4, 5, 3, 2, 1]
اندازه هرم: 6

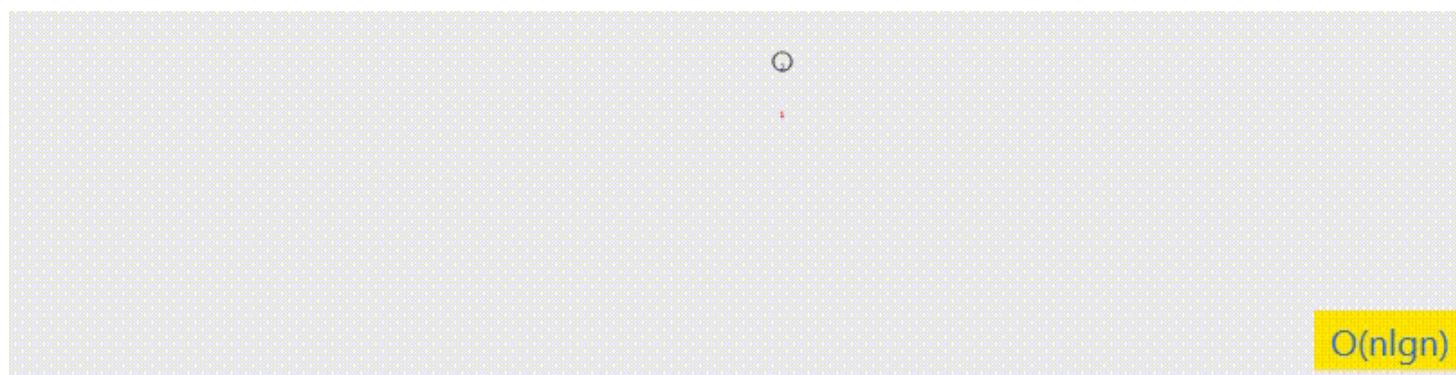
```

## روش‌های ساخت هرم

حال فرض کنید یک لیست عادی از اعداد داریم و می‌خواهیم آن را تبدیل به یک هرم کنیم. برای این کار لیست داده شده را یک هرم در نظر می‌گیریم و سپس به یکی از روش‌های زیر آن را اصلاح می‌کنیم:

## روش اول: ساخت هرم با bubble-up

با شروع از اول لیست، اعداد را یک به یک در هرم bubble-up می‌کنیم. با این روش در هر مرحله، هرم تولید شده با عناصری که تا آن لحظه در هرم bubble-up شده‌اند، یک هرم درست خواهد بود. این کار را آنقدر ادامه می‌دهیم تا همه عناصر در محل درستشان قرار گیرند.



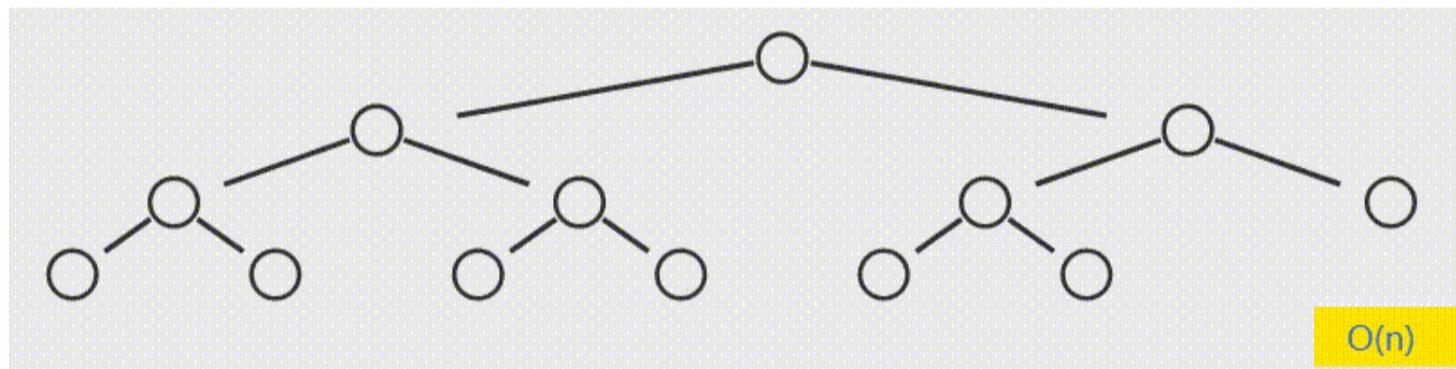
شکل ۵: ساخت هرم با bubble-up (اعداد قرمز اندیس عناصر هستند)

چون در هر bubble-up حداکثر به اندازه فاصله رأس شروع تا ریشه، جابجایی انجام می‌شود (که برابر با ارتفاع گره است)، تعداد کل جابجایی‌ها در بدترین حالت برابر خواهد بود با مجموع ارتفاعات گره‌ها در یک درخت کامل. این مجموع را می‌توان به صورت زیر محاسبه کرد:

$$\begin{aligned}
 & \sum_{i=1}^{\log n} i \cdot 2^i = 1 \cdot 2^1 + 2 \cdot 2^2 + \cdots + \log n \cdot 2^{\log n} \\
 &= (2 + 4 + 8 + \dots + 2^{\log n+1} - 2) + (4 + 8 + \dots + 2^{\log n+1} - 2^2) + \cdots + (2^{\log n+1} - 2^{\log n}) \\
 &= 2^{\log n+1} \cdot \log n - \sum_{i=1}^{\log n} 2^i = 2n \log n - (2^{\log n+1} - 2) \in O(n \log n)
 \end{aligned}$$

## روش دوم: ساخت هرم با bubble-down

با شروع از آخر لیست (یعنی از آخرین گره غیربرگ به سمت ریشه)، عناصر را یک به یک در هرم bubble-down می‌کنیم. با این کار تعدادی هرم کوچک ساخته می‌شود که در مراحل بعد با هم ترکیب می‌شوند تا هرم اصلی را بسازند.



شکل ۶: ساخت هرم با bubble-down (اعداد قرمز اندیس عناصر هستند)

در هر bubble-down حداکثر به اندازه فاصله رأس شروع تا پایین‌ترین سطح درخت، عمل جابجایی انجام می‌شود (که برابر با ارتفاع گره است). پس تعداد کل جابجایی‌ها حداکثر برابر خواهد بود با:

$$\sum_{i=0}^{\lfloor \log n \rfloor} (\text{عافترا نآرد مرگ دادعت}) \times (\text{هرگ عافترا}) = \sum_{h=0}^{\lfloor \log n \rfloor} h \cdot 2^{\lfloor \log n \rfloor - h} \approx \sum_{h=0}^{\log n} h \cdot 2^{\log n - h}$$

این مجموع را می‌توان به صورت دقیق‌تر نیز محاسبه کرد و نشان داد که پیچیدگی آن از  $O(n)$  است:

$$\begin{aligned}
 & \sum_{i=1}^{\log n} (\log n - i) \cdot 2^i = \log n \sum_{i=1}^{\log n} 2^i - \sum_{i=1}^{\log n} i \cdot 2^i \\
 &= \log n \cdot (2^{\log n+1} - 2) - (\log n \cdot 2^{\log n+1} - (2^{\log n+1} - 2)) = 2n - 2 \in O(n)
 \end{aligned}$$

کد هر دو روش در قسمت قبل آمده است. در نمودار زیر زمان اجرا دو روش گفته شده با هم مقایسه شده است:

```
In [3]: # وارد کردن کتابخانه‌های لازم
import timeit
from random import randrange
from matplotlib.pyplot import figure, xlabel, ylabel, plot, legend, show # برای رسم نمودار
```

```

import numpy as np # برای np.power

# تابع get_time که زمان اجرای یک تابع را بر حسب میکروثانیه می‌دهد
def get_time(f, input_data): # تغییر نام 'input' به 'input_data' برای جلوگیری از تداخل با built-in
    start = timeit.default_timer()
    f(input_data)
    stop = timeit.default_timer()
    return (stop - start) * 1000 * 1000 # microseconds

# تابع get_avg_time که متوسط حداقل زمان اجرای یک تابع را محاسبه می‌کند
def get_avg_time(f, input_data):
    مقدار اولیه بسیار بزرگ (بی‌نهایت)
    res = 1 << 30 # تعداد مراحل برای تکرار کل فرآیند و انتخاب حداقل
    s = 10 # تعداد دفعات اجرای تابع در هر مرحله برای گرفتن میانگین
    r = 5 # برای اطمینان از اینکه تابع مرتب‌سازی روی یک کپی از داده اجرا شود
    و داده اصلی برای تکرارهای بعدی تغییر نکند
    for i in range(s):
        sum_times = 0
        for j in range(r):
            برای اطمینان از اینکه تابع مرتب‌سازی روی یک کپی از داده اجرا شود
            # داده اصلی برای تکرارهای بعدی تغییر نکند
            # یک متد از کلاس باشد f اگر
            # باید نمونه جدیدی از کلاس را در هر بار ایجاد کرد
            if hasattr(f, '__self__') and isinstance(f.__self__, Max_Heap):
                temp_heap = Max_Heap()
                temp_heap.heap = [0] + input_data # فرض می‌کنیم build_heap متغیر نکند
                داریم تا تغییر نکند input_data اینجا نیاز به یک کپی عمیق از
                یک تابع مستقل است که لیست را می‌گیرد f برای این مثال، فرض می‌کنیم
                # یا اگر متد کلاس است، روی یک کپی از لیست کار می‌کند
                را به تابع پاس دهیم Max_Heap باید یک نمونه جدید از build_heap برای
                # را طوری تغییر دهیم که خودش یک کپی از لیست بگیرد f یا تابع
                # یک لیست جدید است که تغییر نمی‌کند input_data برای سادگی در اینجا، فرض می‌کنیم
                # را می‌گیرد input_data خودش یک کپی از f یا اینکه
                # راه حل صحیح‌تر:
                current_heap_instance = Max_Heap()
                current_heap_instance.heap = [0] + list(input_data) # کپی عمیق از لیست
                sum_times += get_time(f.__self__.build_heap_with_bubble_up if f.__name__ == 'build_heap_with_bubble_up' else f.__self__.build_heap_with_bubl

            else:
                sum_times += get_time(f, list(input_data)) # ارسال کپی از لیست برای جلوگیری از تغییر ورودی
        res = min(res, sum_times / float(r))
    return res

# برای رسم نمودار زمان اجرای دو روش ساخت هر متد
def plot1(N_values, numbers_list): # تغییر نام پارامترها برایوضوح
    figure(figsize=(20, 10))
    xlabel("Size of input (n)")
    ylabel("Time (microseconds)")

    # برای محاسبه زمان برای build_heap_with_bubble_up
    # برای فراخوانی متدها Max_Heap نیاز به یک نمونه از
    temp_heap_up = Max_Heap()
    y_bubble_up = [get_avg_time(temp_heap_up.build_heap_with_bubble_up, a) for a in numbers_list]
    plot(N_values, y_bubble_up, 'r', label='build heap with bubble up', linewidth=5)

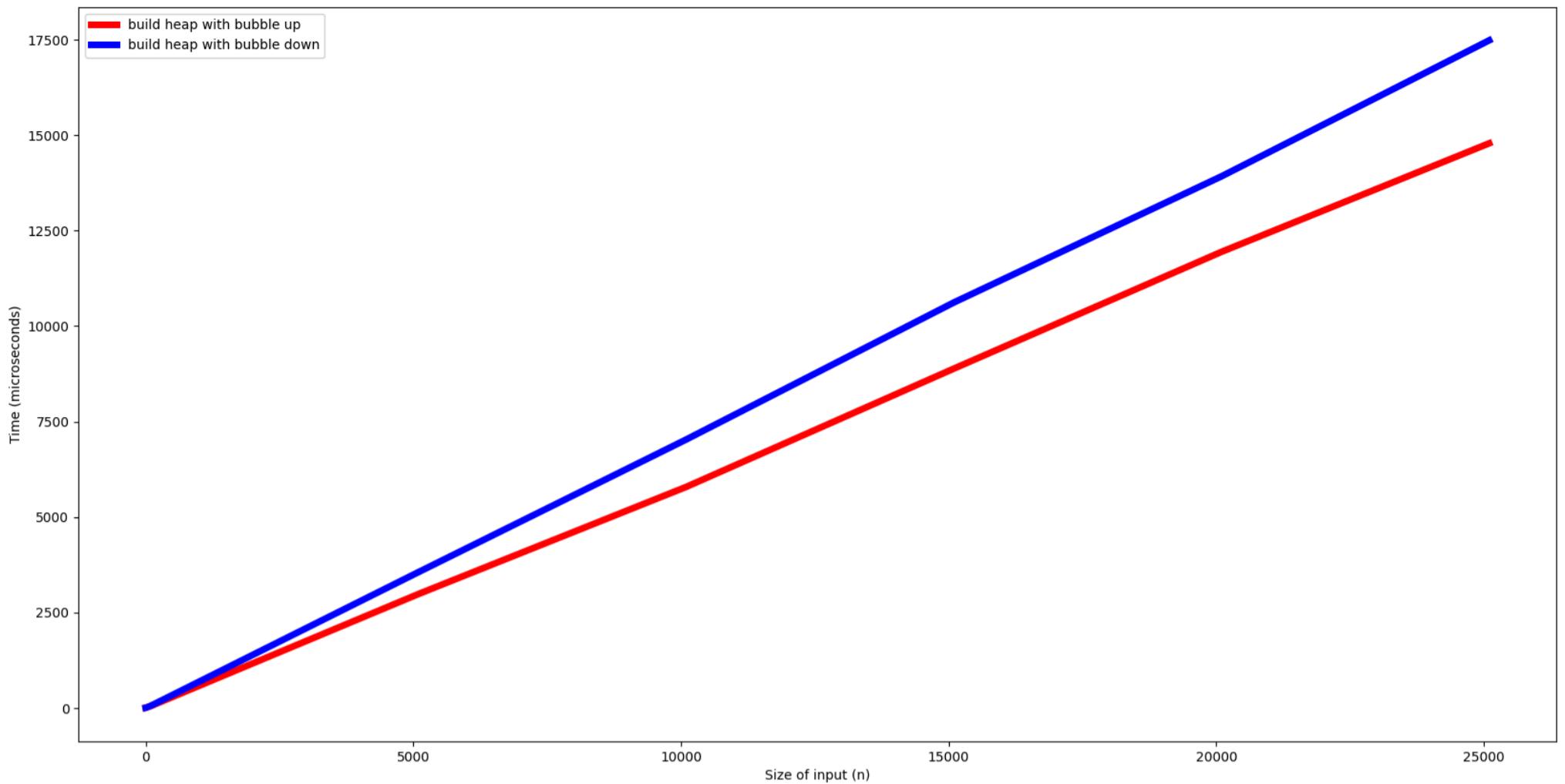
    # برای محاسبه زمان برای build_heap_with_bubble_down
    temp_heap_down = Max_Heap()
    y_bubble_down = [get_avg_time(temp_heap_down.build_heap_with_bubble_down, a) for a in numbers_list]
    plot(N_values, y_bubble_down, 'b', label='build heap with bubble down', linewidth=5)

    legend(loc=2)
    show()

# برای تست N و numbers مقداردهی
max_N = 30000
N = [1, 5] + list(range(100, max_N, 5000)) # اندازه‌های ورودی
numbers = [[randrange(100000) for j in range(i)] for i in N] # تولید لیست‌های تصادفی

# برای رسم نمودار مقایسه plot1 فراخوانی تابع
plot1(N, numbers)

```



## کاربردهای هرم

### صف اولویت

صف اولویت (Priority Queue) داده‌ساختاری شبیه صف و پشته است با این تفاوت که برخلاف صف و پشته که اولویت ورود و خروج را بر حسب زمان ورود تعیین می‌کنند، می‌تواند برای ورود و خروج عناصر اولویت‌های دیگری هم تعیین کند. به این معنی که عنصری که بالاترین اولویت را دارد (نه لزوماً اولین ورودی با آخرين ورودی) زودتر از صف خارج می‌شود. یک روش پیاده‌سازی صف اولویت با استفاده از هرم است که به دلیل کارایی بالای هرم در عملیات‌های درج و حذف بیشینه/کمینه، بسیار کارآمد است. البته این داده‌ساختار را می‌توان با داده‌ساختارهای دیگری نظیر صف و لیست پیوندی هم پیاده‌سازی کرد، اما با پیچیدگی زمانی متفاوت.

برای مطالعه بیشتر می‌توانید به [این لینک](#) مراجعه کنید.

### استفاده از هرم در مسائل

از هرم در بسیاری از سوالات و الگوریتم‌های معروف استفاده می‌شود. هرم به دلیل قابلیت دسترسی سریع به عنصر بیشینه یا کمینه و همچنین درج و حذف کارآمد، در بهینه‌سازی الگوریتم‌های دیگر نقش کلیدی دارد. چند نمونه از این الگوریتم‌ها عبارت‌اند از:

- الگوریتم دایکسترا (Dijkstra's Algorithm) برای یافتن کوتاه‌ترین مسیر در گراف‌ها. هرم برای مدیریت صف اولویت گره‌هایی که باید بازدید شوند، استفاده می‌شود.
- الگوریتم پریم (Prim's Algorithm) برای یافتن زیردرخت پوشای کمینه (Minimum Spanning Tree) در گراف‌ها. هرم برای انتخاب یال با کمترین وزن به کار می‌رود.
- الگوریتم‌های یافتن عناصر خاص مانند میانه (Median) و k-امین عضو کوچک (k-th smallest element) در یک مجموعه داده بزرگ.

# خودآزمایی ۱: ادغام لیست‌های مرتب شده

در ورودی یک لیست از لیست‌های مرتب شده داده می‌شود. با استفاده از هرم، لیست‌های داده شده را با هم ادغام کنید و لیست مرتب شده حاصل را به عنوان خروجی تابع برگردانید. سعی کنید الگوریتم شما از  $O(n \log k)$  باشد که  $n$  و  $k$  به ترتیب تعداد کل اعداد و تعداد لیست‌ها هستند.

In [4]: `from src.tests.tester import tester # وارد کردن تابع`

```
برای خودآزمایی tester # از هرم استفاده کنید و لیست مرتب شده حاصل را به عنوان خروجی تابع برگردانید. سعی کنید الگوریتم شما از  
در دسترس است (یا یک پیاده‌سازی از صف اولویت) Min_Heap فرض می‌شود کلاس # استفاده می‌کنیم تا کوچکترین عنصر را به راحتی استخراج کنیم Min_Heap برای این سوال، از  
توابع کمکی برای محاسبات انديس در هرم (برای آرایه ۱-انديس)  
def parent(i):  
    # انديس والد يك گره را برمي‌گرداند.  
    return i // 2  
  
def left_child(i):  
    # انديس فرزند چپ يك گره را برمي‌گرداند.  
    return 2 * i  
  
def right_child(i):  
    # انديس فرزند راست يك گره را برمي‌گرداند.  
    return 2 * i + 1  
  
class Min_Heap_for_Merge:  
    def __init__(self):  
        # متاد سازنده: هرم را مقداردهی اولیه می‌کند. خانه صفر آرایه خالی گذاشته می‌شود (برای ۱-انديس)  
        self.heap = [0]  
  
    def size(self):  
        # تعداد عناصر واقعی در هرم را برمي‌گرداند.  
        return len(self.heap) - 1  
  
    def bubble_up(self, ind):  
        # را به سمت بالا در هرم حرکت می‌دهد تا خاصیت هرم حفظ شود 'ind' عنصر در انديس.  
        # مقایسه بر اساس مقدار (اولین عنصر تاپل) انجام می‌شود.  
        while ind > 1 and self.heap[ind][0] < self.heap[parent(ind)][0]:  
            self.heap[ind], self.heap[parent(ind)] = self.heap[parent(ind)], self.heap[ind]  
            ind = parent(ind)  
  
    def bubble_down(self, ind):  
        # را به سمت پایین در هرم حرکت می‌دهد تا خاصیت هرم حفظ شود 'ind' عنصر در انديس.  
        while left_child(ind) <= self.size():  
            smaller_child_ind = left_child(ind)  
            # اگر فرزند راست وجود داشت و از فرزند چپ کوچکتر بود، آن را به عنوان فرزند کوچکتر انتخاب کن.  
            if right_child(ind) <= self.size() and self.heap[right_child(ind)][0] < self.heap[left_child(ind)][0]:  
                smaller_child_ind = right_child(ind)  
  
            # اگر گره فعلی از فرزند کوچکترش کوچکتر بود، نیازی به ادامه نیست.  
            if self.heap[ind][0] < self.heap[smaller_child_ind][0]:  
                break  
  
            # جایگایی گره فعلی با فرزند کوچکترش.  
            self.heap[ind], self.heap[smaller_child_ind] = self.heap[smaller_child_ind], self.heap[ind]  
            ind = smaller_child_ind  
  
    def insert(self, item_tuple):  
        # يك عنصر جدید را به هرم اضافه می‌کند.  
        # item_tuple (value, list_index, element_index_in_list) به صورت  
        self.heap.append(item_tuple)  
        self.bubble_up(self.size())  
  
    def get_min(self):  
        # کوچکترین عنصر هرم را بدون حذف برمی‌گرداند.  
        if self.size() == 0:  
            raise Exception("Heap is empty")  
        return self.heap[1]  
  
    def del_min(self):  
        # کوچکترین عنصر هرم را حذف کرده و برمی‌گرداند.  
        if self.size() == 0:  
            raise Exception("Heap is empty")  
        MIN = self.heap[1] # کوچکترین عنصر (ريشه هرم)  
        self.heap[1] = self.heap[-1] # آخرین عنصر را به جای ریشه قرار بده.  
        del(self.heap[-1]) # عنصر آخر را حذف کن.  
        if self.size() > 0: # را انجام بده، فقط اگر هرم خالی نشد.  
            self.bubble_down(1)  
        return MIN  
  
def merge_lists(list_of_sorted_lists):  
    """  
        اين تابع لیستی از لیست‌های مرتب شده را با استفاده از هرم ادغام می‌کند.  
        تعداد لیست‌ها است  $k$  تعداد کل عناصر و  $n$  که  $O(n \log k)$ : پیچیدگی زمانی.  
    """  
    sorted_list = []  
    min_heap = Min_Heap_for_Merge()  
  
    مرحله ۱: اضافه کردن اولین عنصر از هر لیست به هرم.  
    # هر عنصر به صورت یک تاپل (value, list_index, element_index_in_list) ذخیره می‌شود.  
    for list_idx, current_list in enumerate(list_of_sorted_lists):  
        if current_list: # اگر لیست خالی نبود  
            min_heap.insert((current_list[0], list_idx, 0))
```

مرحله 2: استخراج کوچکترین عنصر از هرم و اضافه کردن عنصر بعدی از همان لیست #

```
# استخراج کوچکترین عنصر
while min_heap.size() > 0:
    value, list_idx, elem_idx = min_heap.del_min() # کوچکترین عنصر دیگری وجود داشت
    sorted_list.append(value) # اضافه کردن به لیست نهایی

    # اگر در لیست اصلی که این عنصر از آن آمده بود، عنصر دیگری وجود داشت
    if elem_idx + 1 < len(list_of_sorted_lists[list_idx]):
        next_value = list_of_sorted_lists[list_idx][elem_idx + 1]
        min_heap.insert((next_value, list_idx, elem_idx + 1)) # اضافه کردن عنصر بعدی به هرم

return sorted_list
```

# مثال استفاده:

```
example_lists = [[1, 5, 9], [2, 6, 10], [3, 7, 11]]
merged_result = merge_lists(example_lists)
print(f"لیست‌های ادغام شده: {merged_result}")
```

خروجی مورد انتظار برای این مثال: # [11, 10, 9, 7, 6, 5, 3, 2, 1]

لیست‌های ادغام شده: [11, 10, 9, 7, 6, 5, 3, 2, 1]

## مرتب‌سازی هرمی

بکی از کاربردهای مهم هرم، مرتب‌سازی هرمی (Heapsort) است که یک الگوریتم مرتب‌سازی مقایسه‌ای است. این الگوریتم در فصل بعد به آن اشاره می‌شود. Heapsort از ویژگی‌های هرم (ساخت هرم در  $O(n)$  و حذف مکرر عنصر بیشینه در  $O(\log n)$ ) برای مرتب کردن یک آرایه استفاده می‌کند. پیچیدگی زمانی آن در بدترین حالت  $O(n \log n)$  است و حافظه اضافی  $(1)$  نیاز دارد، که آن را به یک الگوریتم مرتب‌سازی کارآمد و با ثبات تبدیل می‌کند.

In [ ]:

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل ششم، بخش اول: مرتب‌سازی هرمی

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

• مقدمه

• یادآوری هرم

• مرتب‌سازی هرمی

### مقدمه

در بخش پنجم با داده‌ساختار هرم آشنا شده‌اید. هرم یک داده‌ساختار درختی است که به ما اجازه می‌دهد به سرعت به بزرگترین یا کوچکترین عنصر دسترسی پیدا کنیم و عملیات درج و حذف را به صورت کارآمد انجام دهیم. در این فصل، به بررسی یکی از مهم‌ترین کاربردهای هرم، یعنی **\*مرتب‌سازی هرمی (Heapsort)** **\*\*** می‌پردازیم.

در این روش، اعدادی که می‌خواهیم مرتب کنیم را درون یک هرم (مثلاً هرم بیشینه) قرار می‌دهیم. سپس کافیست مقدار بیشینه را از هرم بخوانیم و آن را ثبت و از هرم حذف کنیم. این کار را آنقدر ادامه می‌دهیم تا هرم خالی شود. با هر بار حذف عنصر بیشینه، بزرگترین عنصر بعدی در دسترس قرار می‌گیرد و به این ترتیب، لیست به صورت مرتب شده استخراج می‌شود.

### یادآوری هرم

هرم داده‌ساختاری است که با استفاده از آرایه پیاده‌سازی می‌شود و خاصیت درختی کامل را حفظ می‌کند. در یک هرم بیشینه، ارزش هر گره از فرزندانش بیشتر است و در یک هرم کمینه، ارزش هر گره از فرزندانش کمتر است.

عملیات‌های اصلی هرم شامل درج، حذف بزرگترین (یا کوچکترین) عنصر و تغییر کلید یک عنصر موجود است که همگی در زمان  $O(\log n)$  انجام می‌گیرند، که  $n$  تعداد عناصر در هرم است.

برای توضیحات بیشتر در مورد ساختار و عملیات‌های هرم، می‌توانید به **\*فصل پنجم، بخش اول: ذخیره‌سازی و پیمایش درخت\*** (که در متن شما به عنوان "قسمت چهارم بخش درخت ها" اشاره شده) مراجعه کنید. برای الگوریتم مرتب‌سازی هرمی که در ادامه می‌آید، از توابع تعریف شده در آن بخش (مانند `bubble_up` و `bubble_down`) استفاده خواهیم کرد.

### مرتب‌سازی هرمی

مرتب‌سازی هرمی به شرح زیر است:

1. **ساخت هرم (Build Heap):** در ابتدا، آرایه‌ی ورودی را با رویه‌ی ساخت هرم که در بخش هرم‌ها ذکر شد (معمولًاً روش `build_heap_with_bubble_down` که کارآمدتر است)، به صورت یک هرم بیشینه در می‌آوریم. این مرحله تضمین می‌کند که بزرگترین عنصر در ریشه هرم (اولین خانه آرایه) قرار گیرد.

2. استخراج عناصر (Extract Max): پس از آن، بزرگترین عنصر آرایه (که ریشه هرم است) را با عنصر آخر آرایه جابجا می‌کنیم. این کار باعث می‌شود بزرگترین عنصر به جایگاه نهایی خود در انتهای آرایه منتقل شود. سپس، تعداد عناصر در نظر گرفته شده برای هرم را یک واحد کم می‌کنیم و عناصر باقیمانده (به جز عنصر تازه جابجا شده) را با استفاده از عملیات `bubble\_down` (که در بخش هرمهای شرح داده شد) دوباره به صورت یک هرم در می‌آوریم.

با 1 -  $n$  بار انجام این عمل (جابجایی ریشه با آخرین عنصر و سپس `bubble\_down` روی هرم کوچکتر شده)، در نهایت آرایه به صورت مرتب شده در می‌آید.

همانطور که در بخش هرم گفته شد، پیچیدگی ساخت هرم از  $O(n)$  است.

همچنین، هزینه هر بار اجرای `bubble\_down` (که در اینجا معادل `MaxHeapify` است) از  $O(\log i)$  است، که  $i$  تعداد عناصر باقیمانده در هرم است. بنابراین، هزینه کل مرتبسازی هرمی عبارت است از:

$$O(n) + \sum_{i=2}^{n-1} O(\log i) = O(n \log n)$$

به سوالات زیر درباره مرتبسازی هرمی جواب دهید:

## 1. برای این مرتبسازی چقدر حافظه اضافی لازم است؟ آیا می‌توان این مرتبسازی را به صورت درجا (in-place) پیاده‌سازی کرد؟

\*پاسخ: مرتبسازی هرمی را می‌توان به صورت درجا (in-place) پیاده‌سازی کرد، به این معنی که به حافظه اضافی  $O(1)$  (به جز حافظه لازم برای ذخیره آرایه ورودی) نیاز دارد. این به این دلیل است که هرم مستقیماً در همان آرایه ورودی ساخته می‌شود و عناصر مرتب شده نیز در همان آرایه قرار می‌گیرند. در هر مرحله، بزرگترین عنصر به انتهای بخش نامرتب آرایه منتقل می‌شود و بخش مرتب شده به تدریج از انتهای سمت ابتدا رشد می‌کند.

## 2. آیا می‌توان این مرتبسازی را به صورت پایدار (stable) پیاده‌سازی کرد؟

\*پاسخ: مرتبسازی هرمی به صورت پیشفرض \*نایپایدار (unstable)\* است. پایداری به این معنی است که اگر دو عنصر با مقدار یکسان در آرایه وجود داشته باشند، ترتیب نسبی آنها در آرایه مرتب شده حفظ شود. در Heapsort، عملیات‌های `bubble\_down` و جابجایی عناصر ممکن است ترتیب نسبی عناصر با مقادیر یکسان را برابر نمایند. پیاده‌سازی آن به صورت پایدار پیچیده است و معمولاً به حافظه اضافی نیاز دارد، بنابراین در عمل به عنوان یک الگوریتم نایپایدار در نظر گرفته می‌شود.

## خودآزمایی 1:

به کمک هرم کمینه و به روش مرتبسازی هرمی، لیست ورودی را مرتب کنید و آن را برگردانید.

In [1]: `from src.tests.tester import tester # وارد کردن تابع tester`

```
In [2]: # توابع کمکی برای محاسبه اندیس پدر و فرزندان در آرایه (برای هرم)
def parent(i):
    return i // 2

def left_child(i):
    return 2 * i

def right_child(i):
    return 2 * i + 1

class Min_Heap:
    # پیاده‌سازی یک هرم کمینه
    def __init__(self):
        # متدهای سازنده: یک هرم کمینه خالی ایجاد می‌کند
        self.heap = [0] * 1
        # خانه صفر آرایه خالی گذاشته می‌شود تا اندیس‌ها از 1 شروع شوند.

    def size(self):
        # تعداد عناصر واقعی در هرم را برگرداند.
        return len(self.heap) - 1

    def bubble_up(self, ind):
        # به سمت بالای هرم حرکت می‌دهد 'ind' این تابع یک عنصر را از موقعیت # تا خاصیت هرم کمینه حفظ شود.
        while ind > 1 and self.heap[ind] < self.heap[parent(ind)]:
            self.heap[ind], self.heap[parent(ind)] = self.heap[parent(ind)], self.heap[ind]
            ind = parent(ind)

    def bubble_down(self, ind):
        # به سمت پایین هرم حرکت می‌دهد 'ind' این تابع یک عنصر را از موقعیت #
```

```

# تا خاصیت هرم کمینه حفظ شود.
# زمانی که فرزند چپ وجود داشته باشد # فرض می‌کنیم فرزند چپ کوچکتر است.
while left_child(ind) <= self.size(): # را بهروزرسانی کن
    smaller_child_ind = left_child(ind) # اگر فرزند راست وجود داشت و کوچکتر از فرزند چپ بود
    if right_child(ind) <= self.size() and self.heap[right_child(ind)] < self.heap[left_child(ind)]:
        smaller_child_ind = right_child(ind)

    # اگر گره فعلی کوچکتر از کوچکترین فرزندش بود، نیازی به جایگایی نیست
    if self.heap[ind] < self.heap[smaller_child_ind]:
        break

    # جایگایی گره فعلی با کوچکترین فرزندش
    self.heap[ind], self.heap[smaller_child_ind] = self.heap[smaller_child_ind], self.heap[ind]
    ind = smaller_child_ind # بهروزرسانی اندیس برای ادامه bubble_down.

def insert(self, item):
    # یک عنصر جدید را به هرم اضافه می‌کند
    self.heap.append(item)
    # برای قرار دادن عنصر در جایگاه صحیح bubble_up انجام
    self.bubble_up(self.size() - 1)

def get_min(self):
    # کوچکترین عنصر در هرم را برمی‌گرداند (ریشه هرم).
    if self.size() == 0:
        raise Exception("Heap is empty")
    return self.heap[1]

def del_min(self):
    # کوچکترین عنصر را از هرم حذف کرده و آن را برمی‌گرداند
    if self.size() == 0:
        raise Exception("Heap is empty")
    MIN = self.heap[1] # کوچکترین عنصر را ذخیره می‌کند
    self.heap[1] = self.heap[-1] # آخرین عنصر را به جای ریشه قرار می‌دهد
    del(self.heap[-1]) # حذف آخرین عنصر از آرایه
    if self.size() > 0: # فقط اگر هرم خالی نشد، bubble_down
        self.bubble_down(1)
    return MIN

def build_heap_with_bubble_down(self, L):
    # می‌سازد bubble_down با استفاده از روش L هرم را از یک لیست
    # این روش کارآمدتر است و از آخرين گره غیربرگ شروع می‌کند
    # (خانه 0 خالی) L مقداردهی اولیه هرم با لیست [0] + L
    # شروع از آخرين گره غیربرگ تا ریشه
    for i in range(self.size() // 2, 0, -1):
        self.bubble_down(i) # برای هر گره bubble_down انجام می‌دهد

```

```

In [3]: def heap_sort(list_of_numbers):
    """
    مرتب می‌کند (Heapsort) این تابع یک لیست از اعداد را با استفاده از مرتبسازی هرمی
    از پیاده‌سازی هرم کمینه استفاده می‌کند
    """

    if not list_of_numbers:
        return []

    # مرحله 1: ساخت یک هرم کمینه از لیست ورودی
    # استفاده می‌کنیم که کارآمدتر است و build_heap_with_bubble_down از روش
    min_heap_instance = Min_Heap()
    min_heap_instance.build_heap_with_bubble_down(list_of_numbers)

    sorted_list = []
    # مرحله 2: استخراج عناصر از هرم به ترتیب صعودی
    # هر بار کوچکترین عنصر (ریشه) را حذف کرده و به لیست مرتب شده اضافه می‌کنیم
    while min_heap_instance.size() > 0:
        sorted_list.append(min_heap_instance.del_min())

    return sorted_list

tester("heap sort", heap_sort)

```

Your code passes all 3 test(s).

In [ ]:

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل ششم، بخش دوم: درخت تصمیم، حد پایین مرتب‌سازی مقایسه‌ای

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

• مقدمه

• حد پایین

• تلاشی برای آوردن حد پایین مرتب‌سازی مقایسه‌ای

• درخت تصمیم

• حد پایین برای الگوریتم‌های مرتب‌سازی مقایسه‌ای در حالت میانگین

## مقدمه

در این بخش از درس، وارد مبحثی عمیق‌تر در تحلیل الگوریتم‌ها می‌شویم: **\*یافتن کران پایین (Lower Bound)**. تا کنون، ما عمدتاً به تحلیل الگوریتم‌های موجود و یافتن پیچیدگی زمانی آن‌ها (کران بالا) پرداخته‌ایم. اما آیا می‌توانیم ثابت کنیم که هیچ الگوریتمی، حتی اگر هنوز کشف نشده باشد، نمی‌تواند کاری را سریع‌تر از یک حد مشخص انجام دهد؟

در این بخش، با مفهوم **\*\*درخت تصمیم (Decision Tree)** آشنا می‌شویم که ابزاری قدرتمند برای تحلیل کران پایین الگوریتم‌های مقایسه‌ای است. ما به طور خاص، کران پایینی برای تعداد مقایسه‌ها در الگوریتم‌های مرتب‌سازی مقایسه‌ای (مانند Quick Sort و Insertion Sort، Bubble Sort، Selection Sort، Merge Sort) به دست خواهیم آورد.

## حد پایین

در ابتدا ببینیم که محاسبه‌ی حد پایین برای یک کار چه معنایی دارد؟ تا اینجا درس ما با این آشنا شده‌ایم که الگوریتمی به ما داده شود و ما پیچیدگی زمانی این الگوریتم را حساب کنیم. برای مثال می‌توانیم بگوییم که پیچیدگی زمانی الگوریتم مرتب‌سازی هرمی از  $\Theta(n \log n)$  است.

اگر از ما پرسیده شود که الگوریتم با بهترین پیچیدگی زمانی برای انجام یک کار کدام الگوریتم است، احتمالاً تمام الگوریتم‌هایی را که برای آن کار بد هستیم بررسی می‌کنیم و الگوریتمی را که پیچیدگی زمانی کمتری داشته باشد را اعلام می‌کنیم. اما آیا امکان دارد که اعلام کنیم الگوریتمی با پیچیدگی زمانی بهتر از  $\Theta(f(n))$  وجود ندارد؟ در اینجاست که می‌گوییم ما کرانی پایین برای الگوریتم‌هایی که یک کار مشخص را انجام می‌دهند را اعلام می‌کنیم. البته این بیان کمی نادقيق است زیرا دقیقاً باید مشخص شود که پیچیدگی زمانی یک الگوریتم برای انجام یک کار را چه تعریف می‌کنیم (مثلاً تعداد مقایسه‌ها، تعداد عملیات ریاضی، یا دسترسی به حافظه).

برای گرفتن شهود بیشتر یک مثال ساده را بررسی می‌کنیم:

مسئله: الگوریتمی طراحی کنید که  $n$  عدد را از ورودی بگیرد و بزرگ‌ترین عدد را در بین آن‌ها خروجی بدهد.

حد پایین: پیچیدگی این الگوریتم را تعداد بارهایی که بین دو عضو از دنباله‌ی ورودی مقایسه انجام می‌شود، تعریف می‌کنیم.

مشخص است که حداقل هر عضو باید در یک مقایسه شرکت کند، وگرنه اگر آن عضو در هیچ مقایسه‌ای شرکت نکند، هیچ اطلاعاتی درباره‌ی بزرگی یا کوچکی آن نسبت به بقیه به دست نمی‌آید. در این صورت، اگر خروجی الگوریتم همان عضو باشد، ممکن است آن عضو، عضوی مبینیم باشد (و نه ماقسیم)؛ یا اگر خروجی آن الگوریتم آن عضو نباشد، ممکن است آن عضو، خود عضو ماقسیم باشد. در واقع می‌توان گفت که ما از آن عضو هیچ اطلاعاتی جمع‌آوری نکرده‌ایم که این سبب می‌شود الگوریتم درست کار نکند.

پس می‌توان گفت لاقل  $\lceil \frac{n}{2} \rceil$  مقایسه داشته باشیم (اگر عضوها را جفت جفت مقایسه کنیم). بنابراین هر الگوریتمی که برای انجام این کار پیدا شود، لاقل  $\Omega(n)$  مقایسه نیاز خواهد داشت.

سوال: در مورد مساله‌ی بالا ثابت کنید که لاقل  $1 - n$  مقایسه نیاز است.

راهنمایی: یک گراف را در نظر بگیرید که هر رأس آن متناظر با یکی از اعداد داده شده در دنباله‌ی ورودی باشد. حال هر مقایسه بین دو عضو از اعداد را با رسم یک یال بین آنها نشان دهید. برای اینکه بتوانیم بزرگترین عنصر را به طور قطع مشخص کنیم، باید مطمئن شویم که آن عنصر بزرگتر از تمام  $1 - n$  عنصر دیگر است. این به این معنی است که باید مسیری از مقایسه‌ها وجود داشته باشد که بزرگترین عنصر را به تمام عناصر دیگر متصل کند و برتری آن را نشان دهد. آیا این گراف می‌تواند ناهمبند باشد؟ خیر، برای یافتن بزرگترین عنصر، باید تمام عناصر به نوعی با هم مقایسه شوند تا یک "برنده" نهایی مشخص شود، که این نیازمند یک گراف همبند است. یک گراف همبند با  $n$  رأس، حداقل  $1 - n$  یال نیاز دارد.

## تلashی برای بدست آوردن حد پایین مرتب‌سازی مقایسه‌ای

مرتب‌سازی از مهمترین مسائل دنیای الگوریتم‌هاست. آیا الگوریتم‌های مرتب‌سازی هم حد پایین دارند؟ در زیر به این نتیجه می‌رسیم که آن نوعی از الگوریتم‌های مرتب‌سازی که بر مبنای مقایسه هستند و در واقع با پرسیدن رابطه‌ی کوچکتری یا بزرگتری یا مساوی بودن بین دو عنصر، دنباله را مرتب می‌کنند، حد پایین دارند. صورت دقیق‌تر در زیر آورده شده است:

سوال: دنباله‌ای از  $n$  عدد به صورت  $a_1, a_2, \dots, a_n$  به ما داده شده است. آنها را تنها با پرسش‌های به صورت مقایسه‌ی بین دو عنصر مرتب کنید. (پیچیدگی چنین الگوریتمی را تعداد این مقایسه‌ها تعريف می‌کنیم)

سوال اساسی این است که آیا الگوریتم‌هایی که به سوال بالا پاسخ‌می‌دهند، حد پایین دارند؟

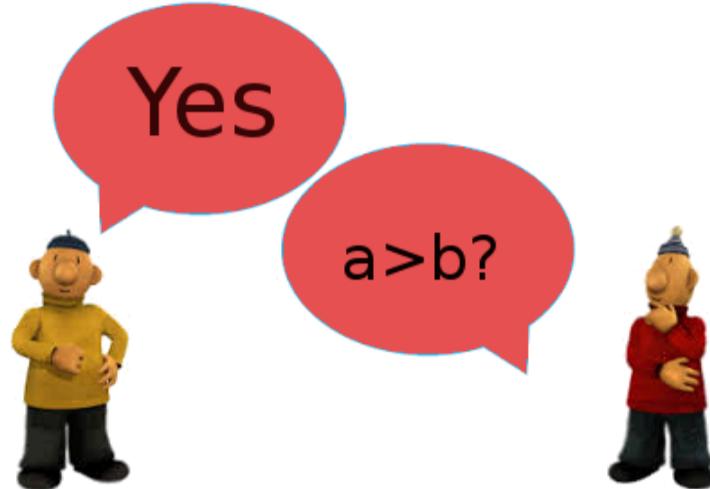
فرض کنید پت دنباله‌ای ۳ حرفی از اعداد را در ذهن خودش در نظر گرفته است و از مت می‌خواهد تا تنها با پرسیدن سوال‌هایی به صورت مقایسه‌ای دنباله‌اش را حدس بزند.

مت به این صورت عمل می‌کند که فرض می‌کند دنباله‌ی پت به صورت  $c, b, a$  است. ابتدا تمامی دنباله‌های ممکن مرتب شده را در نظر می‌گیرد (همانند تصویر زیر) تا سپس بتواند دنباله‌ی مرتب‌شده‌ی درست را به دست آورد:

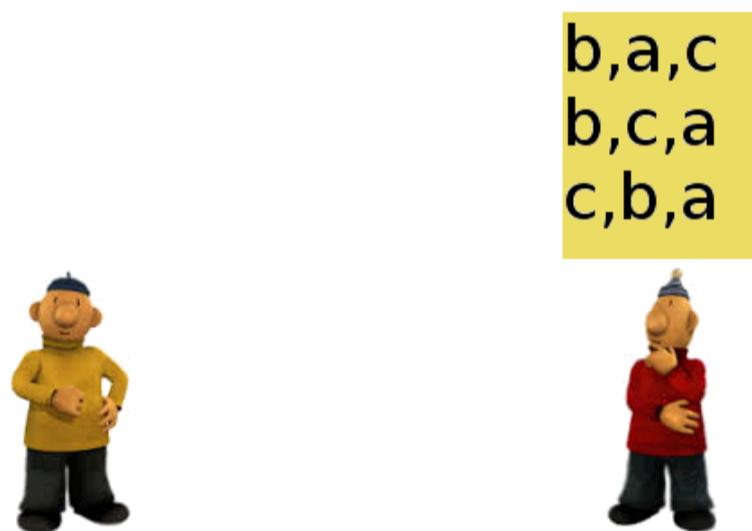
a,b,c  
a,c,b  
b,a,c  
b,c,a  
c,b,a  
c,a,b



حال مت به عنوان اولین سوال از پت می‌پرسد که آیا  $a > b$  و جواب بله می‌شود:



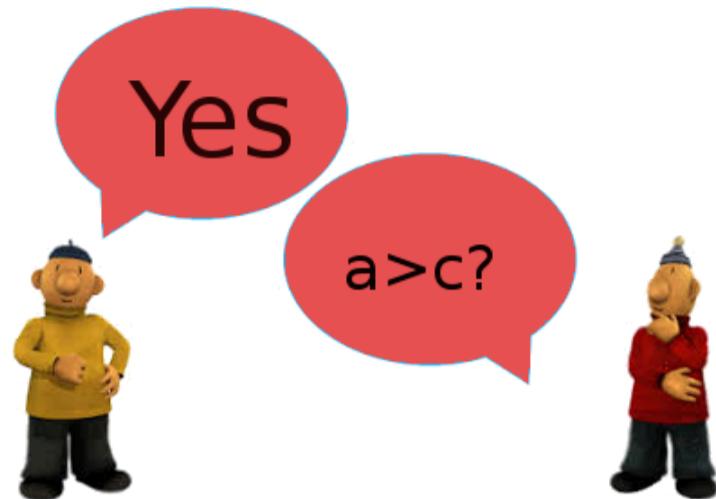
و حالا مت حالاتی که این شرط را ندارند از ذهننش حذف می‌کند:



فکرک: آیا مت می‌توانست سوالی را بپرسد که حالات بیشتری را برای او حذف کند؟

\*\*پاسخ:\*\* بله، مت می‌توانست سوالی بپرسد که فضای جستجو را به طور متعادل‌تری تقسیم کند. مثلًا اگر سوالی می‌پرسید که تقریباً نیمی از حالات را در صورت "بله" و نیمی دیگر را در صورت "خیر" حذف می‌کرد، بهینه‌تر عمل می‌کرد. هدف در هر مرحله، بیشترین کاهش ممکن در تعداد حالات باقی‌مانده است.

حالا مت از پت می‌پرسد که آیا  $c > a$  و باز هم جواب مثبت می‌شنود:



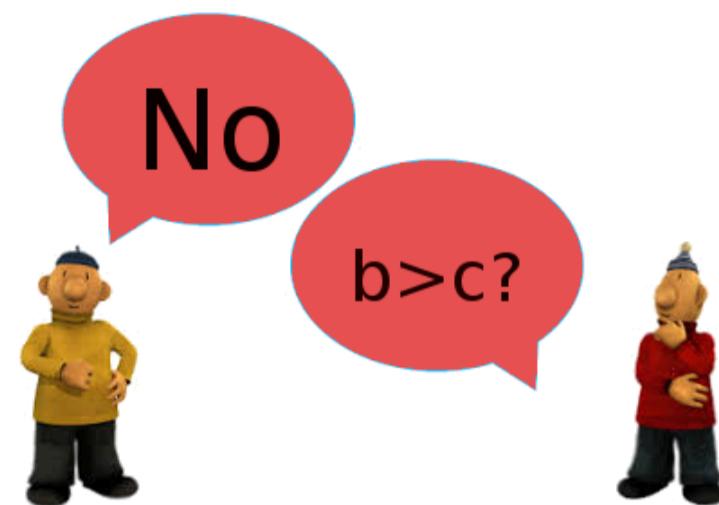
فکرک: آیا مت می‌توانست سوال هوشمندانه‌تری بپرسد؟

\*\*پاسخ:\*\* بله، در این مرحله نیز مت می‌توانست سوالی بپرسد که حالات باقی‌مانده را به طور متعادل‌تری تقسیم کند. مثلًا سوال  $c > b$  می‌توانست در این مرحله نیز مطرح شود.

و حالا مت حالاتی را که در این شرایط صدق نمی‌کنند از ذهننش حذف می‌کند:



حالا مت با یک پرسش میتواند دنباله‌ی مرتب شده را بیابد. او از پت میپرسد که آیا  $b > c$ . و جواب منفی میگیرد:



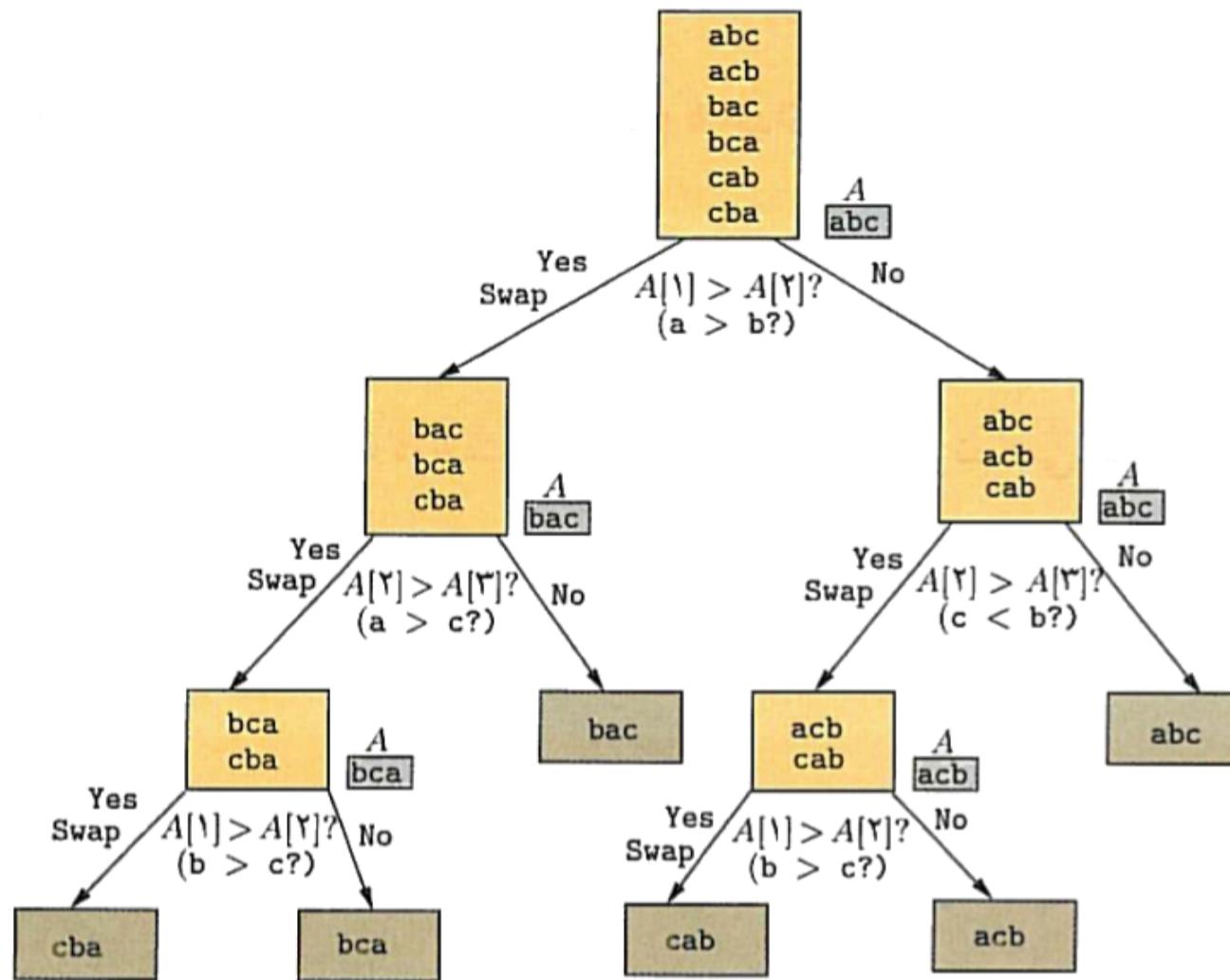
و حالا مت رشته‌ی مرتب شده را می‌داند:

b,c,a



درخت تصمیم

درخت تصمیم در واقع تنها یک مدل شهودی برای بررسی عملکرد بعضی الگوریتم‌هاست. شهود ساده‌ی پشت مفهوم «درخت تصمیم» این است که یک الگوریتم غیر احتمالاتی بر اساس داده‌هایی که تا کنون به دست آورده، ممکن است چند تصمیم مختلف بگیرد. برای مثال بالا برای فهمیدن رشته‌ی پت از درخت تصمیم زیر استفاده می‌کند. دقت کنید که اتفاقی که در بالا افتاد بخشی از درخت تصمیم مت بود!



(عکس بالا از کتاب «داده‌ساختارها و مبانی الگوریتم‌ها» نوشته‌ی دکتر محمد قدسی گرفته شده است.) برای هر الگوریتم مرتب‌سازی غیر احتمالاتی می‌توان برای هر  $n$  چنین درختی را رسم کرد.

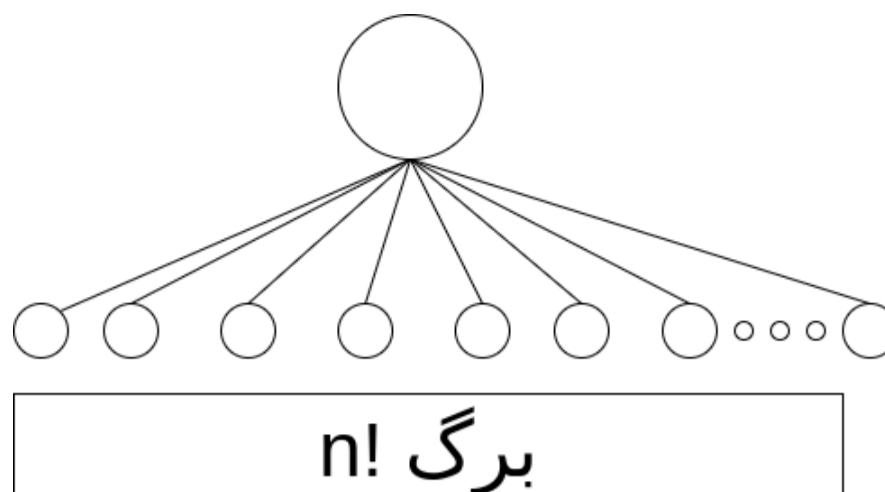
دقت کنید که این درخت قطعاً  $n!$  تا برگ دارد، زیرا در هر رأس مجموعه‌ی جواب‌هایی را می‌بینید که ممکن است جواب باشند و هنوز طبق اطلاعات گرفته شده ردنشده‌اند. هر مسیر از ریشه تا برگ در درخت تصمیم، یک دنباله از مقایسه‌ها را نشان می‌دهد که به یک ترتیب خاص از عناصر منجر می‌شود. پس هر مسیر درخت تصمیم را که بگیریم در نهایت آنقدر حالات مختلف را حذف می‌کند تا تنها یک حالت برای جواب وجود داشته باشد و از آنجایی که مطمئن بودیم جواب در بین اعضای همان مجموعه‌ای بوده که از اول گرفتیم، پس همین عضو باقیمانده جواب است.

چون در کل وقتی یک دنباله از  $n$  عدد داریم،  $n!$  حالت برای مرتب‌شدن آن‌ها ممکن است (فرض کنید اعداد متمایزند، اگر چه در ابتدا که الگوریتم هیچ‌چیز نمی‌داند واقعاً  $n!$  برایش مطرح خواهد بود)، پس باید  $n!$  برگ هم داشته باشیم که اگر الگوریتم در مسیر رشد خود به هر یک از آن‌ها برسد، به این معناست که جواب را یافته‌است.

## هدفمان را فراموش نکنیم

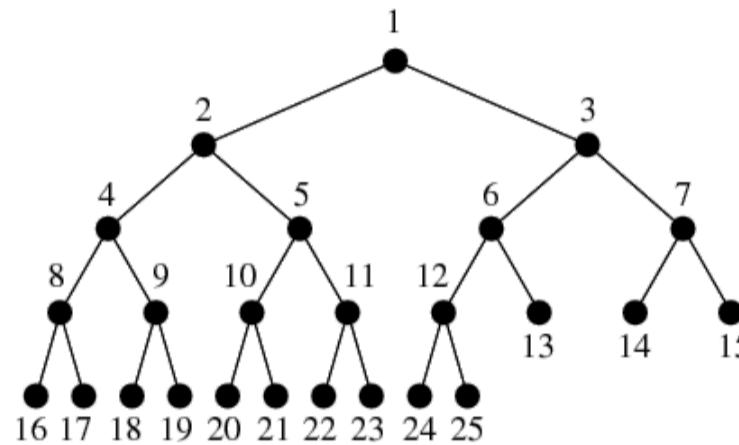
چرا درخت تصمیم را معرفی کردیم؟ اگر به یاد داشته باشید هدف ما پیدا کردن حد پایینی برای الگوریتم‌های مرتب‌سازی مقایسه‌ای بود. توجه کنید که اگر درخت تصمیم را برای یک الگوریتم خاص برای یک  $n$  خاص ترسیم کنیم، عمق دورترین برگ این درخت پیچیدگی آن الگوریتم در بدترین حالت را نشان می‌دهد (عمق ریشه را صفر در نظر بگیرید) زیرا برای هر بار عمیق شدن باید یک سوال مقایسه بپرسیم.

پس اگر بتوانیم ثابت کنیم که هر درخت با  $n!$  برگ لاقل عمقی از  $(n \log n)$  دارد، توانسته‌ایم حد پایینی برای الگوریتم‌های مرتب‌سازی مقایسه‌ای بیابیم.



اما یک لحظه صبر کنید، موضوعی را به نظر فراموش کرده‌ایم، درخت تصمیم خاصیتی دارد که مانع از رشد بسیار سریع آن می‌شود و آن هم این است که جواب هر سوال با «بله» است یا «خیر».

در واقع شکل درخت‌های تصمیم به صورت زیر است:



یک درخت دودویی با ارتفاع  $h$  حداقل  $2^h$  برگ دارد.

لم ۱:

در واقع درخت تصمیم در هر رأس تنها می‌تواند به دو شاخه تقسیم شود. در این زمینه لم زیر را داریم:

\*اثبات: اثبات این لم ساده‌است. تنها کافیست بر روی  $h$  استقرا بزنید.

- \*\*حالت پایه (Base Case):\*\* برای  $h = 0$  (درخت تنها ریشه است)، تعداد برگ‌ها  $= 1 = 2^0$  است.
- \*\*فرض استقرا (Inductive Hypothesis):\*\* فرض کنید برای هر درخت دودویی با ارتفاع  $1 - h$ ، حداقل  $2^{h-1}$  برگ وجود دارد.
- \*\*گام استقرا (Inductive Step):\*\* یک درخت دودویی با ارتفاع  $h$  را در نظر بگیرید. ریشه آن دارای حداقل دو فرزند (چپ و راست) است. زیردرخت‌های این فرزندان حداقل ارتفاع  $1 - h$  خواهند داشت. طبق فرض استقرا، هر یک از این زیردرخت‌ها حداقل  $2^{h-1}$  برگ دارند. بنابراین، کل درخت حداقل استقرا، هر یک از این زیردرخت‌ها حداقل  $2^{h-1} + 2^{h-1} = 2 \cdot 2^{h-1} = 2^h$  برگ خواهد داشت.

$$2^h \geq n! \rightarrow h \geq \lceil \log_2(n!) \rceil$$

حال از آنجا که درخت تصمیم یک درخت دودویی است و  $n!$  برگ دارد، اگر ارتفاع یا عمق آن را  $h$  بگیریم، خواهیم داشت:

$$\text{پس داریم که } h \text{ از } \Omega(n \log n)$$

$$\log_2(n!) = \Theta(n \log n)$$

:

و از قبل می‌دانیم که (از تقریب استرلینگ برای  $n!$  و خواص لگاریتم):

\*\*تبریک!\*\* ما حد پایینی برای عملکرد تمامی الگوریتم‌های مقایسه‌ای غیر احتمالاتی در بدترین حالت یافتیم. این بدان معناست که هیچ الگوریتم مرتب‌سازی مقایسه‌ای نمی‌تواند در بدترین حالت، سریع‌تر از  $O(n \log n)$  عمل کند.

**سوال:** ثابت کنید که نتیجه‌ی بالا برای الگوریتم‌های مقایسه‌ای احتمالاتی هم صادق است. در واقع منظور این است که اگر الگوریتم در تصمیم‌گیری خود برای تصمیم بعدیش تنها بر اساس «بله» و «خیر» هایی که تا کنون دریافت‌کرده‌است تصمیم نگیرد و معیاری تصادفی هم برای این تصمیم داشته باشد.

**راهنمایی:** فرض کنید شما  $n!$  حالت را در ابتدا در نظر دارید. حالا با هر سوال اگر جواب بله باشد  $a$  حالت و اگر جواب خیر باشد  $a - n!$  حالت باقی‌خواهد‌ماند. به همین ترتیب اگر برای ما در نقطه‌ای از اجرای الگوریتم  $A$  حالت مطرح باشد و سوالی بپرسیم اگر جواب «بله» باشد  $a$  حالت و اگر جواب «خیر» باشد  $a - A$  حالت باقی‌مانده داریم. حال از این استفاده کنید که ممکن است رشتی ما جزو آن مجموعه‌ی بزرگتر باشد، پس لاقل یک ورودی وجود دارد که به ازای آن با هر بار پرسش اندازه‌ی مجموعه‌ی حالت ممکن حداقل نصف می‌شود.

# حد پایین برای الگوریتم‌های مرتب‌سازی مقایسه‌ای در حالت میانگین

درست است که برای عملکرد الگوریتم‌های مرتب‌سازی مقایسه‌ای در بدترین حالت حد پایینی یافتیم اما باید بینیم که عملکرد الگوریتم‌ها در حالت میانگین هم حد پایینی دارد با خیر؟

در واقع می‌خواهیم بینیم که اگر تمامی ورودی‌ها با ترتیب‌های مختلف مرتب شدن (در واقع در بین  $n!$  حالت ممکن) با احتمال یکسان ظاهر شوند، عملکرد الگوریتم ما چگونه است.

اگر بخواهیم به زبان ریاضی صحبت کنیم، می‌توانیم بگوییم اگر متغیر تصادفی  $X$  برابر با تعداد مقایسه‌ها به ازای یکبار اجرای این الگوریتم بر روی ورودی‌ای باشد که با احتمال یکسان یکی از  $n!$  حالت ممکن است،  $E(X)$  (امید ریاضی  $X$ ) را می‌خواهیم. این نتیجه می‌دهد که باید تعداد مقایسه‌ها را به ازای هر حالت از  $n!$  حالت ممکن به دست بیاوریم و جمع بزنیم و بر  $n!$  تقسیم کنیم.

دقت کنید که این در واقع معادل این است که مجموع عمق برگ‌های درخت تصمیم را بر  $n!$  تقسیم کنیم.

در هر درخت دودویی با  $m$  برگ، مجموع عمق برگ‌ها دست کم  $m \log_2(m)$  است.

- \*\*حالت پایه (Base Case): برای  $m = 1$  (یک برگ)، عمق  $1 \cdot \log_2(1) = 0$  است.
- \*\*گام استقرای (Inductive Step): فرض کنید لم برای درختان با کمتر از  $m$  برگ درست است. درختی با  $m$  برگ را در نظر بگیرید. ریشه آن دارای دو زیردرخت چپ و راست است. فرض کنید زیردرخت چپ  $m_L$  برگ و زیردرخت راست  $m_R$  برگ دارد، به طوری که  $m_L + m_R = m$ . عمق برگ‌های زیردرخت چپ  $h_L + 1$  و عمق برگ‌های زیردرخت راست  $h_R + 1$  است. طبق فرض استقرای، مجموع عمق برگ‌های زیردرخت چپ  $\geq m_L \log_2(m_L)$  و برای زیردرخت راست  $\geq m_R \log_2(m_R)$  است. بنابراین، مجموع عمق برگ‌های کل درخت

$\geq m_L(\log_2(m_L) + 1) + m_R(\log_2(m_R) + 1) = m_L \log_2(m_L) + m_R \log_2(m_R) + m$  استفاده از نامساوی‌های ریاضی (مانند نامساوی Jensen) برای تابع  $x \log x$  می‌توان نشان داد که این عبارت  $\geq m \log_2(m)$  است. اثبات دقیق این لم را می‌توانید در کتاب «داده‌ساختارها و مبانی الگوریتم‌ها» دکتر قدسی در صفحه ۳۱۱ مشاهده کنید.

$$\sum_{l_i \text{ in leaves}} h(l_i) \geq n! \log_2(n!) \rightarrow \frac{\sum_{l_i \text{ in leaves}} h(l_i)}{n!} \geq \log_2(n!)$$

با پذیرفتن این لم داریم که الگوریتم‌های مرتب‌سازی مقایسه‌ای در حالت میانگین هم عملکردی از  $\Omega(\log(n!))$  دارند زیرا:

و از آنجایی که  $\log_2(n!) = \Theta(n \log n)$ ، پس میانگین تعداد مقایسه‌ها نیز از  $\Omega(n \log n)$  است.

## توجه!

دقت کنید که درخت تصمیم ابزاری قدرتمند برای اثبات حد پایین برای الگوریتم‌های مرتب‌سازی مقایسه‌ای و اینگونه نیست که تنها برای اثبات حد پایین برای الگوریتم‌های مرتب‌سازی کاربرد داشته باشد.

برای مثال اگر دو دنباله‌ی مرتب شده از اعداد یکی به طول  $m$  و دیگری به طول  $n$  داشته باشیم و بخواهیم تنها با انجام تعدادی مقایسه بین این دو دنباله آنها را با هم ادغام کنیم (Merge Two Sorted Arrays)، با استفاده از درخت تصمیم ثابت می‌شود که به لاقل  $\Omega(\log_2(\binom{m+n}{n}))$  مقایسه نیاز داریم.

\*\*آیا می‌توانید این موضوع را ثابت کنید؟

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل ششم، بخش سوم: مرتب‌سازی سریع

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

## فهرست

- مقدمه
- فرآیند الگوریتم
- افزار
- تقسیم و حل
- حالات بد مرتب‌سازی سریع
- حالات خوب مرتب‌سازی سریع

## مقدمه

در این بخش از درس، به بررسی الگوریتم \*\*مرتب‌سازی سریع (Quicksort)\*\* می‌پردازیم. شما تا کنون با مرتب‌سازی ادغامی (Merge Sort) آشنا شده‌اید و فهمیدید که مرتبه زمانی اجرای آن از  $O(n \log n)$  است. همان‌طور که در مبحث درخت تصمیم فراگرفته‌اید، این بهترین مرتبه زمانی ممکن برای الگوریتم‌های مرتب‌سازی مقایسه‌ای در حالت میانگین است. حال سوالی که مطرح می‌شود این است که چه نیازی به الگوریتم دیگری است؟

جواب این مسئله را می‌توان در سه نکته زیر خلاصه کرد:

1. مرتب‌سازی سریع، یک الگوریتم \*\*in-place\*\* درجا (driven by in-place) است، به همین دلیل حافظه کمتری مصرف می‌کند و نیاز کمتری به دسترسی به RAM دارد. این ویژگی برای کار با داده‌های بزرگ بسیار مهم است.
2. در حالت میانگین، \*\*ضرایب ثابت\*\* آن کمتر است. این بدان معناست که حتی اگر مرتبه زمانی آن مشابه الگوریتم‌های دیگر باشد، در عمل سریع‌تر اجرا می‌شود. شما می‌توانید در نمودار زیر مقایسه زمان اجرای چند الگوریتم را در حالت میانگین ببینید.
3. پیاده‌سازی آن آسان است و مفهوم آن به خوبی قابل درک است.

In [1]: وارد کردن کتابخانه‌های لازم برای رسم نمودار

```
# وارد کردن کتابخانه‌های لازم برای رسم نمودار
import math
from matplotlib.pyplot import figure, xlabel, ylabel, plot, legend, show
برای تولید داده‌های تصادفی
from random import randrange # اگرچه در این مثال مستقیم استفاده نشده، اما در نمونه‌های قبلی بود
import numpy as np # برای np.power
```

(اگرچه در این مثال مستقیم استفاده نشده، اما در نمونه‌های قبلی بود)

```
# فرض می‌شود تابع get_avg_time و get_time (insertion_sort, mergesort, heapsort, quicksort)
# وارد شده‌اند src.codes.mysorts
برای این مثال، فرض می‌کنیم تابع مرتب‌سازی به صورت فرضی در دسترس هستند
# و فقط برای نمایش نمودار از آنها استفاده می‌کنیم
```

```
def plot2(N_values):
```

این تابع نمودار زمان اجرای چهار الگوریتم مرتب‌سازی را برای مقایسه عملکرد آن‌ها رسم می‌کند.  
زمان‌ها بر اساس فرمول‌های تقریبی برای حالت میانگین هستند.

Args:

محدوده‌ای از اندازه‌های ورودی برای رسم نمودار: N\_values (range).

...

```
figure(figsize=(15, 10)) # ایجاد یک شکل
xlabel("Size") # X برجسب محور
ylabel("Time") # Y برجسب محور
```

```

# فرمول تقریبی Quicksort محاسبه و رسم نمودار برای
#  $11.667 * (n+1) * \log(n) - 1.74 * n - 18.74$ 
y_quicksort = [11.667 * (n + 1) * math.log(n) - 1.74 * n - 18.74 if n > 0 else 0 for n in N_values]
plot(N_values, y_quicksort, 'r', label='Quicksort', linewidth=5)

# فرمول تقریبی Mergesort محاسبه و رسم نمودار برای
#  $12.5 * n * \log(n)$ 
y_mergesort = [12.5 * n * math.log(n) if n > 0 else 0 for n in N_values]
plot(N_values, y_mergesort, 'g', label='Mergesort', linewidth=5)

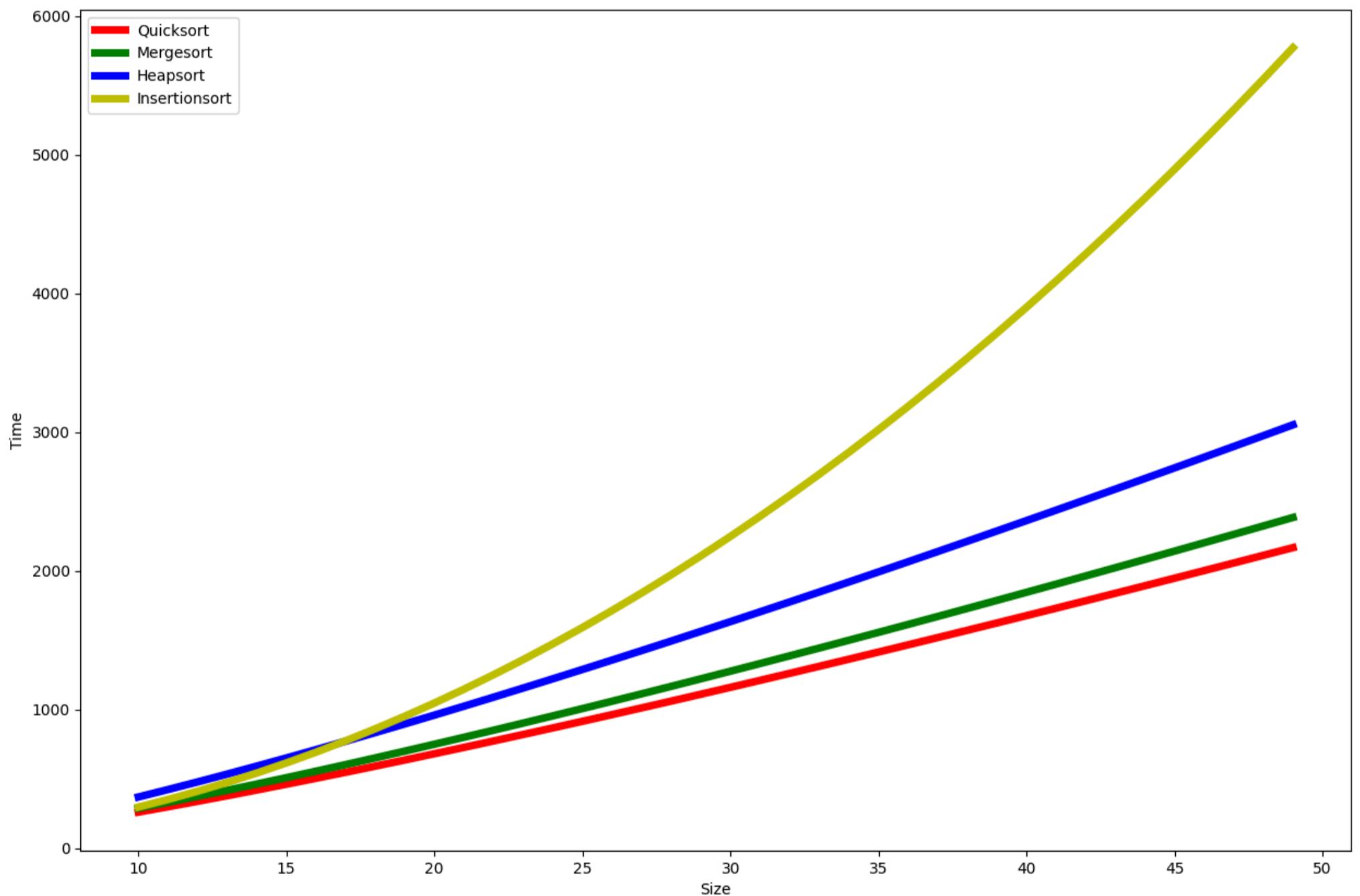
# فرمول تقریبی Heapsort محاسبه و رسم نمودار برای
#  $16 * n * \log(n) + 0.01 * n$ 
y_heapsort = [16 * n * math.log(n) + 0.01 * n if n > 0 else 0 for n in N_values]
plot(N_values, y_heapsort, 'b', label='Heapsort', linewidth=5)

# فرمول تقریبی Insertionsort محاسبه و رسم نمودار برای
#  $2.25 * n^2 + 7.75 * n - 3 * \log(n)$ 
y_insertionsort = [2.25 * (n ** 2) + 7.75 * n - 3 * math.log(n) if n > 0 else 0 for n in N_values]
plot(N_values, y_insertionsort, 'y', label='Insertionsort', linewidth=5)

legend(loc=2)
show()

# برای تست N مقداردهی
max_N = 50
N_values_for_plot = range(10, max_N, 1) # 49 تا 10 از ورودی از آندازه‌های
# برای رسم نمودار مقایسه plot2 فراخوانی تابع
plot2(N_values_for_plot)

```



## فرآیند الگوریتم

مرتب‌سازی سریع (Quicksort) یک الگوریتم<sup>\*</sup> تقسیم و حل (Divide and Conquer) است که به صورت بازگشتی عمل می‌کند. برای مرتب کردن یک آرایه  $A$  در بازه‌ی اندیس‌های  $p$  تا  $r$  (یعنی  $A[p \dots r]$ ، دو گام کلی زیر را داریم:

1. افزایش (Partition): آرایه به سه بخش (ممکن است بخش‌ها خالی باشند) افزایش می‌شود:  $A[p \dots q - 1]$ . این افزایش به گونه‌ای انجام می‌شود که هر عنصر در بخش  $A[q + 1 \dots r]$  و  $A[q]$

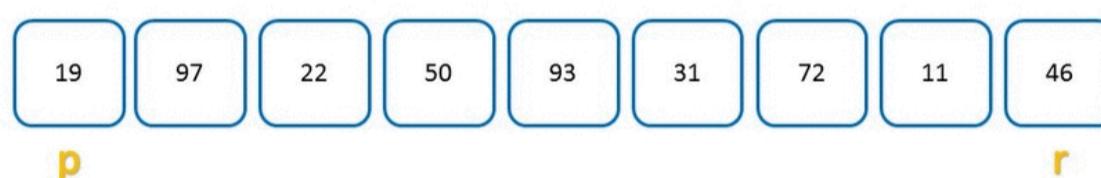
مساوی  $A[q]$  باشد و هر عنصر در بخش  $A[q + 1 \dots r]$  بزرگتر یا مساوی  $A[q]$  باشد. عنصر  $A[q]$  را \*\*عنصر محور (Pivot) این آرایه می‌نامیم. انتخاب عنصر محور مناسب، کلید کارایی این الگوریتم است.

2. تقسیم و حل (Divide and Conquer): هر یک از بخش‌های  $A[p \dots q - 1]$  و  $A[q + 1 \dots r]$  را با فراخوانی تابع بازگشته مرتب‌سازی سریع، مرتب می‌کنیم. بخش  $A[q]$  (عنصر محور) در جایگاه نهایی خود قرار گرفته است و نیازی به مرتب‌سازی ندارد. بنابراین در نهایت آرایه اصلی نیز مرتب خواهد بود.

## افراز

گیف زیر روند افزار در مرتب‌سازی سریع را نشان می‌دهد. در این فرآیند، عناصر آرایه نسبت به عنصر محور جایجا می‌شوند تا به شرط افزار (عناصر کوچکتر در یک طرف و بزرگتر در طرف دیگر) برسند.

### آرایه اولیه



قطعه کد زیر تابع افزار (partition) در مرتب‌سازی سریع است. این تابع یک بازه از آرایه را می‌گیرد و آن را حول یک عنصر محور افزار می‌کند.

برای بررسی درستی این تابع به این نکته توجه کنید که در طول اجرای حلقه خط چهارم (در کد پایتون)، همه اعداد سمت چپ leftPointer از محور در نظر گرفته کوچک‌تر هستند و همه اعداد سمت راست rightPointer هم بزرگ‌تر یا مساوی محور هستند.

همچنین این حلقه پایان‌پذیر است، چون در هر مرحله از اجرای حلقه، فاصله اشاره‌گرها (rightPointer و leftPointer) حداقل یکی کم می‌شود و در نهایت به هم می‌رسند یا از هم عبور می‌کنند.

In [2]:

```
import random برای استفاده از # random.randrange

def partition(A, left, right):
    """
    حول یک عنصر محور افزار می‌کند [left, right] را در بازه A این تابع آرایه عناصر کوچک‌تر از محور به سمت چپ و عناصر بزرگ‌تر به سمت راست منتقل می‌شوند.

    Args:
        A (list): آرایه‌ای که قرار است افزار شود.
        left (int): اندیس شروع بازه افزار.
        right (int): اندیس پایان بازه افزار.

    Returns:
        tuple: یک تاپل شامل
            index_of_pivot: اندیس نهایی عنصر محور پس از افزار.
            pivot_value: مقدار عنصر محور.

    """
    # انتخاب عنصر محور به صورت تصادفی و جایجا آن با آخرین عنصر
    # این کار به کاهش احتمال بدترین حالت کمک می‌کند.
    pivot_index = random.randrange(left, right + 1)
    A[right], A[pivot_index] = A[pivot_index], A[right]
    pivot = A[right] # آخرین عنصر پس از جایجا شده

    leftPointer = left # اشاره‌گر از سمت چپ
    rightPointer = right - 1 # اشاره‌گر از سمت راست (قبل از محور)

    while True:
        # به راست تا یافتن عنصری بزرگ‌تر از محور LeftPointer حرکت
        while leftPointer <= rightPointer and A[leftPointer] <= pivot:
            leftPointer = leftPointer + 1
        # به چپ تا یافتن عنصری کوچک‌تر یا مساوی محور RightPointer حرکت
        while rightPointer >= leftPointer and A[rightPointer] > pivot:
            rightPointer = rightPointer - 1
```

```

اگر اشاره‌گرها از هم عبور کردند، حلقه را بشکن.
if leftPointer >= rightPointer:
    break
else:
    جایگای عناصر نامناسب (عنصر بزرگ در چپ و عنصر کوچک در راست) #
    A[leftPointer], A[rightPointer] = A[rightPointer], A[leftPointer]

قرار دادن عنصر محور در جایگاه نهایی خود #
A[leftPointer], A[right] = A[right], A[leftPointer]

برگرداندن اندیس نهایی محور و مقدار آن # return leftPointer, pivot

```

## تقسیم و حل

روند بازگشتنی مرتب‌سازی سریع (تقسیم و حل) در گیف زیر نمایان است. پس از افزار، الگوریتم به صورت بازگشتنی روی دو زیرآرایه (سمت چپ و راست عنصر محور) فراخوانی می‌شود تا آن‌ها را مرتب کند. این فرآیند تا زمانی ادامه می‌یابد که زیرآرایه‌ها به اندازه‌ای کوچک شوند که مرتب باشند (مثلاً دارای صفر یا یک عنصر).



تابع بازگشتنی مرتب‌سازی سریع را در قطعه کد زیر مشاهده می‌کنید. درستی این تابع را می‌توان با استقرار روی طول آرایه ثابت کرد.

به این صورت که فرض می‌کنیم، هر آرایه‌ای با طول کمتر از  $n$  را می‌توان با این تابع مرتب کرد. حال برای آرایه با طول  $n$ ، توسط محور انتخاب شده، آرایه را به دو تکه تقسیم می‌کنیم و هر تکه را به صورت بازگشتنی مرتب می‌کنیم.

حالت پایه الگوریتم نیز آرایه‌هایی با طول یک و یا صفر است که در شرط خط پنجم (در کد) بررسی شده است. در این حالت‌ها، آرایه از قبل مرتب در نظر گرفته می‌شود و نیازی به عملیات بیشتر نیست.

In [3]:  
 باید قبل از این تابع تعریف شده باشد `partition` توجه: تابع  
 # به عنوان یک متغیر سراسری در نظر گرفته شده است  $A$ ، در این مثال  
 # پاس داده می‌شود معمولاً به عنوان یک آرگومان به  $A$ ، در پیاده‌سازی‌های واقعی

`A = [19, 97, 22, 50, 93, 31, 72, 11, 46]` # آرایه نمونه برای مرتب‌سازی  
`print(f"A: {A}\n")`

`def quickSort(arr, left, right):`  
 """  
 مرتب می‌کند (Quicksort) این تابع آرایه را با استفاده از الگوریتم مرتب‌سازی سریع.

Args:

آرایه‌ای که قرار است مرتب شود.  
`arr` (list):  
 اندیس شروع بازه مرتب‌سازی.  
`left` (int):  
 اندیس پایان بازه مرتب‌سازی.  
`right` (int):

"""  
 حالت پایه: اگر بازه دارای ۰ یا ۱ عنصر باشد، مرتب است #  
`return`

مرحله افزار: آرایه را حول یک عنصر محور افزار می‌کند  
`# index, pivot_value = partition(arr, left, right)`

چاپ وضعیت آرایه پس از هر افزار (برای مشاهده روند) #  
`print(f"left={left}, right={right}, pivot={pivot_value}, index={index}")`  
`print(arr)`  
`print("")`

مرحله تقسیم و حل: فراخوانی بازگشتنی برای زیرآرایه‌های چپ و راست #  
 مرتب کردن زیرآرایه سمت چپ محور # (۱ - `quickSort(arr, left, index - 1)`)

```

quickSort(arr, index + 1, right) # مرتب کردن زیرآرایه سمت راست محور # فراخوانی تابع A
# برای مرتب کردن آرایه quickSort(A)
# سراسری لازم است A برای دسترسی به Jupyter Notebook به این خط در # global A # این خط در
quickSort(A, 0, len(A) - 1) # از اندیس 0 تا n-1 فراخوانی با کل آرایه # آرایه مرتب شده نهایی f"A"

```

آرایه اصلی: [46, 11, 72, 31, 93, 50, 22, 97]

```
left=0, right=8, pivot=11, index=0
[11, 97, 22, 50, 93, 31, 72, 46, 19]
```

```
left=1, right=8, pivot=93, index=7
[11, 46, 22, 50, 19, 31, 72, 93, 97]
```

```
left=1, right=6, pivot=72, index=6
[11, 46, 22, 50, 19, 31, 72, 93, 97]
```

```
left=1, right=5, pivot=50, index=5
[11, 46, 22, 31, 19, 50, 72, 93, 97]
```

```
left=1, right=4, pivot=31, index=3
[11, 19, 22, 31, 46, 50, 72, 93, 97]
```

```
left=1, right=2, pivot=19, index=1
[11, 19, 22, 31, 46, 50, 72, 93, 97]
```

آرایه مرتب شده نهایی: [11, 19, 22, 31, 46, 50, 72, 93, 97]

## حالات بد مرتب‌سازی سریع

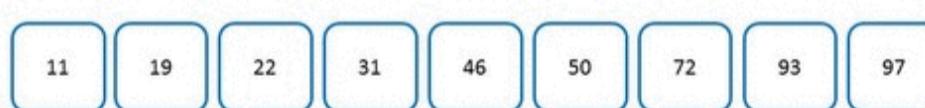
گرچه در ابتدای متن اشاره کردیم که حالت میانگین مرتب‌سازی سریع بهتر از سایر الگوریتم‌های مبتنی بر مقایسه است، اما حالات خاصی نیز وجود دارند که مرتبه زمانی اجرای الگوریتم از  $\Theta(n^2)$  می‌شود.

این حالات هنگامی رخ می‌دهند که افزای متوازن نباشد، بدین معنا که یکی از بخش‌های حاصل از افزای  $k$  عضوی باشد که  $k$  عدد ثابتی است که باعث می‌شود حل زیربخش دیگر، که  $n - k$  عضو دارد، تقریباً به اندازه حالت ابتدایی مشکل باشد.

یک مثال بارز از این موضوع، آرایه مرتب‌شده‌است، که همانطور که در شکل زیر مشاهده می‌کنید، یکی از بخش‌ها همواره صفر عضوی است (اگر عنصر محور کوچکترین یا بزرگترین عنصر باشد). بنابراین برای رابطه بازگشتی زمان اجرا خواهیم داشت:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \theta(n) \\ &= T(n-1) + \theta(n) \end{aligned}$$

که به راحتی می‌توان نشان داد از  $\Theta(n^2)$  است.



آرایه مرتب اولیه

یک روش برای بهبود شرایط در این وضعیت‌ها استفاده از الگوریتم‌های ترکیبی است. برای مثال الگوریتم مرتب‌سازی درون‌گرا (Introsort) که ترکیبی از الگوریتم‌های مرتب‌سازی سریع (تصادفی) و هرمی است. این الگوریتم با فراخوانی مرتب‌سازی سریع شروع می‌شود و هنگامی که بخش بازگشتی مرتب‌سازی سریع به عمق مشخصی (متناسب با  $\log n$ ) رسید، الگوریتم مرتب‌سازی هرمی را فراخوانی می‌کند. این کار تضمین می‌کند که پیچیدگی زمانی در بدترین حالت نیز از  $O(n \log n)$  فراتر نرود.

# حالات خوب مرتبسازی سریع

نکته کلیدی که برای این حالات وجود دارد، متوازن بودن افزار است، بدین معنا که دو بخش حاصل از افزار، به صورتی باشند که نسبت اعضای دو بخش یک عدد ثابت باشد. این موضوع باعث می‌شود که بخش‌های ما به اندازه کافی کوچک شوند که در نتیجه زمان کلی اجرای الگوریتم کمتر شود.

مثال این حالت، شبیه مرتبسازی ادغامی است، یعنی تعداد اعضای بخش‌های حاصل از افزار برابر با  $\lfloor(n-1)/2\rfloor$  و  $\lceil(n-1)/2\rceil$  است که رابطه بازگشتی زمان اجرای زیر را نتیجه می‌دهد:

$$T(n) = 2T(\lfloor(n-1)/2\rfloor) + \theta(n)$$

که به راحتی می‌توان نشان داد از  $\Theta(n \log n)$  است.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل ششم، بخش چهارم: تحلیل مرتب‌سازی سریع تصادفی

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

#### • مقدمه

• پیش نیاز تحلیل: احتمال

• مقدمات: تعریف مسئله به صورت احتمالاتی

• روش طلایی: شکستن مسئله

• محاسبات ریاضی

• انتخاب محور به صورت رندم

## مقدمه

همان‌طور که در فصل‌های قبل دیدید، الگوریتم مرتب‌سازی سریع (Quicksort) در بدترین حالت می‌تواند عملکردی از  $O(n^2)$  داشته باشد، که این برای یک الگوریتم مرتب‌سازی با پیچیدگی  $O(n \log n)$  در حالت میانگین، یک نقطه ضعف محسوب می‌شود.

اما با این وجود، این الگوریتم «عموماً» سریع است و در عمل بسیار پرکاربرد است. برای دقیق کردن این ویژگی، باید تحلیل را با دیدگاه<sup>\*</sup> احتمالاتی<sup>\*\*</sup> انجام دهیم. به عبارتی می‌گوییم الگوریتم باید در حالت میانگین بررسی شود. در زیر توضیحات بیشتری درباره معنای این کار می‌دهیم و سپس تحلیل را انجام می‌دهیم.

## پیش نیاز تحلیل: احتمال

یک شهود: ادعا کردیم که مرتب‌سازی سریع «در حالت میانگین» خوب عمل می‌کند. اما این به چه معناست؟ اولاً باید دقت داشت که «میانگین یک پدیده‌ی تصادفی» یک تعریف ریاضی دقیق دارد. در ادامه با بیان تعریف‌های ابتدایی از احتمال، این تعریف دقیق را بیان خواهیم کرد. اما چیزی که اهمیت دارد اینست که عددی که مطابق این تعریف محاسبه می‌شود چه معنایی برای ما دارد. این معنا از ویژگی‌هایی که «میانگین یک پدیده‌ی تصادفی» (با همان تعریف ریاضی) دارد قابل استخراج است. این ویژگی‌ها به صورت ریاضی به دست می‌آیند و عموماً نتایج ساده‌ای نیستند. قبل از بیان تعریف ریاضی میانگین، ویژگی‌هایی که از این تعریف قابل استخراج است را بیان می‌کنیم تا شهودی پیدا کنیم از اینکه چیزی که می‌خواهیم محاسبه کنیم اصلاً چه معنایی دارد. این ویژگی‌ها را دقیق بیان نمی‌کنیم و صرفاً معنای آن‌ها را در جهان واقع بیان می‌کنیم: (هر جا «میانگین پدیده‌ی تصادفی» نوشته شد به معنای تعریف ریاضی آن است.)

به طور نادقيق می‌توان گفت که اگر «میانگین یک پدیده‌ی تصادفی»  $x$  باشد، آنگاه اگر آن آزمایش را به تعداد زیاد تکرار کنیم و اعدادی که از آزمایش به دست آمده را با هم جمع کنیم و بر تعدادشان تقسیم کنیم (میانگین معمولی آن‌ها را حساب کنیم)، این عدد به احتمال خیلی بالایی به  $x$  خیلی نزدیک خواهد بود.

همچنین می‌توان گفت در هر بار انجام آزمایش عددی که به دست می‌آید به احتمال کمی فاصله خیلی زیادی (زیاد، نسبت به مقدار  $x$ ) از  $x$  خواهد داشت. گرچه احتمال اینکه فاصله خیلی کمی داشته باشد هم لزوماً زیاد نیست.

- فضای نمونه ( $\Omega$ ): تمامی خروجی‌های ممکن یک رویداد تصادفی را فضای نمونه می‌نامند. این فضا دو خاصیت زیر را دارد:

1. احتمال وقوع هر خروجی بزرگتر مساوی صفر است:

$$\exists i \in \Omega : p(i) \geq 0$$

2. مجموع احتمال خروجی‌ها برابر یک است:

$$\sum_{i \in \Omega} p(i) = 1$$

برای مثال، یک فضای نمونه می‌تواند همه‌ی دنباله‌های ممکن از حالت‌هایی باشد که آرایه از ابتدای زمان مرتب شدن پیدا می‌کند. می‌توانیم برای یک مسئله خاص در جهان واقع، فضاهای نمونه متمایزی در نظر بگیریم.

- پیشامد (Event): هر زیرمجموعه‌ای از فضای نمونه را یک پیشامد می‌نامند. احتمال پیشامد  $\Omega \subseteq S$  برابر است با جمع احتمالات خروجی‌های عضو آن زیرمجموعه:

$$\sum_{i \in S} p(i)$$

- متغیر تصادفی (Random Variable) :

در یک پدیده‌ی تصادفی ممکن است کمیت‌های گوناگون وجود داشته باشند که علاقه‌مند باشیم بدانیم که به چه احتمالی چه مقداری می‌گیرند. این علاقه انگیزه‌ی تعریف زیر است:

$$X : \Omega \rightarrow \mathbb{R}$$

به عنوان مثال اگر فضای نمونه همه دنباله‌های ممکن از حالت‌هایی باشد که آرایه از ابتدای زمان مرتب شدن پیدا می‌کند، یک متغیر تصادفی می‌تواند اندازه بخش‌هایی که در نتیجه افزار دوم در مرتب‌سازی سریع به دست می‌آید باشد.

یا در حالت کلی  $X$  می‌تواند متغیر تصادفی اندازه بخش بزرگتر در نتیجه افزار  $\Omega$  باشد.

یا تعداد مقایسه‌هایی که انجام می‌شود خودش یک متغیر تصادفی است که به نظر می‌رسد داشتن اطلاعاتی راجع به اینکه به چه احتمالاتی برابر با چه مقادیری است درباره زمان اجرای الگوریتم پیام می‌دهد.

تمرین: مثال‌هایی از متغیر تصادفی در همین فضای نمونه مطرح کنید.

- امید ریاضی (Expected Value): تعریف دقیق آن به صورت زیر است:

$$E[X] = \sum_{i \in \Omega} p(i) \cdot X(i)$$

دقیق است روی فضای نمونه است.

به عنوان مثال، امید ریاضی طول بخش اول که در اولین گام مرتب‌سازی سریع بر روی یک آرایه  $n$  عضوی اجرا می‌شود برابر با  $n/2$  است. (چرا؟)

\*پاسخ: این به این دلیل است که اگر عنصر محور به صورت تصادفی انتخاب شود، امید ریاضی موقعیت آن در آرایه پس از افزار، در میانه خواهد بود. بنابراین، امید ریاضی اندازه هر یک از دو بخش تقریباً  $n/2$  است.

- خطی بودن امید ریاضی (Linearity of Expectation): ادعا می‌کند که اگر  $X_1$  تا  $X_n$  متغیرهای تصادفی‌ای باشند که روی  $\Omega$  تعریف می‌شوند، آنگاه خواهیم داشت:

$$E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i]$$

تمرین: این ویژگی را با کمک همین تعریف‌ها ثابت کنید. (این تعریف‌ها ساده شده هستند و در حد نیاز این بخش بیان شده‌اند)

- **تابع مشخصه (Indicator Random Variable):** یک نوع خاص از متغیر تصادفی که به صورت زیر تعریف می‌شود:

$$X : \Omega \rightarrow \{0, 1\}$$

به عنوان مثال، مقایسه شدن یا نشدن دو عنصر در روند مرتب‌سازی سریع. این متغیر 1 می‌شود اگر رویداد مورد نظر رخ دهد و 0 می‌شود اگر رخ ندهد. اميد ریاضی یک تابع مشخصه برابر با احتمال وقوع آن رویداد است.

## مقدمات: تعریف مسئله به صورت احتمالاتی

برای تحلیل مرتب‌سازی سریع از دیدگاه احتمالاتی، ابتدا باید فضای نمونه و متغیر تصادفی مورد نظر را به دقت تعریف کنیم:

فضای نمونه  $\Omega$ : در هر مرحله از الگوریتم مرتب‌سازی سریع، یکی از اعداد آرایه را به عنوان محور (pivot) انتخاب می‌کنیم. همه‌ی دنباله‌های ممکن از انتخاب محورها را به عنوان فضای نمونه تعریف می‌کنیم. این دنباله تعیین می‌کند که الگوریتم چگونه پیش می‌رود و چه مقایسه‌هایی انجام می‌شود.

متغیر تصادفی  $C(\sigma)$ : برای هر  $\sigma \in \Omega$  (هر دنباله انتخاب محورها)،  $C(\sigma)$  برابر است با تعداد مقایسه‌هایی که در طول اجرای الگوریتم انجام می‌شود.

زمان اجرای الگوریتم در اردی تعداد مقایسه‌های انجام شده بین اعداد آرایه است. بنابراین کافی است تعداد مقایسه‌ها را محاسبه کنیم.

پس اگر بخواهیم زمان اجرای مرتب‌سازی سریع را محاسبه کنیم، می‌توانیم مقدار  $E[C(\sigma)]$  را در حالت میانگین حساب کنیم، به عبارت دیگر  $E[C(\sigma)]$  را به دست بیاوریم.

اما این کار به صورت مستقیم بسیار سخت و پیچیده است، زیرا تعداد دنباله‌های محورها می‌تواند بسیار زیاد باشد.



## روش طلایی: شکستن مسئله

هنگامی که با یک مسئله بسیار سخت رو به رو می‌شویم، می‌توانیم آن را به قطعات کوچکتر تقسیم کنیم، آن‌ها را حل کنیم و سپس با استفاده از آن‌ها به حل مسئله نهایی برسیم. (این روش، همان استراتژی

\*\* تقسیم و حل\*\* است که در درس طراحی الگوریتم‌ها نیز به طور گسترده استفاده خواهد کرد.)

در این مسئله، قطعات کوچکتر ما محاسبه‌ی یک \*\*تابع مشخصه\*\* است که به صورت زیر تعریف می‌شود:

$X_{ij}(\sigma)$ : متغیری تصادفی است که اگر  $z_i$  و  $z_j$  با هم مقایسه شوند (در حالی که دنباله‌ی محورها  $\sigma$  است)، مقدار 1 و در غیر این صورت مقدار 0 را می‌گیرد.

( $z_i$ ،  $z_1 < z_2 < \dots < z_n$ ) نامین عدد در آرایه مرتب شده است. یعنی

مقدار  $X_{ij}(\sigma)$  صفر یا یک است، زیرا دو عنصر فقط هنگامی با هم مقایسه می‌شوند که یکی از آن‌ها محور باشد و هر دو در یک دسته باشند. پس از این مقایسه، عنصر دیگر در بخش جداگانه‌ای از عنصر محور قرار می‌گیرد (یکی به چپ و دیگری به راست محور می‌رود)، بنابراین هیچ‌گاه دوباره با آن مقایسه نخواهد شد.

$$C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}(\sigma)$$

طبق تعریفی که برای  $C(\sigma)$  ارائه شد، می‌توان تعداد مقایسه‌ها را این گونه بیان کرد که چند جفت عنصر  $(z_i, z_j)$  داریم که آن‌ها یک است، یا به عبارت دیگر:

$$E[C(\sigma)] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}(\sigma)\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}(\sigma)]$$

حال با توجه به \*\*خطی بودن امید ریاضی\*\* (Linearity of Expectation) داریم:

که نشان می‌دهیم می‌توان  $E[X_{ij}(\sigma)]$  را به راحتی به دست آورد.



## محاسبات ریاضی

بنابراین کافی است  $z_j$  را به دست آوریم. طبق تعریف  $X_{ij}$ ، امید ریاضی  $E[X_{ij}]$  برابر با احتمال  $p_{ij}$  است: احتمال مقایسه شدن  $z_i$  و  $z_j$  را  $p_{ij}$  می‌نامیم.

گفته‌یم دو عدد فقط زمانی با هم مقایسه می‌شوند که یکی از آن‌ها محور باشد و هر دو در یک دسته باشند.

همچنانی، هنگامی یکی از  $z_i$  یا  $z_j$  به عنوان محور انتخاب می‌شود، که قبله هیچ یک از عناصر بینشان (یعنی  $z_{j-1}, \dots, z_{i+1}, z_i, z_j$ ) به عنوان محور انتخاب نشده باشد. در واقع، عدد  $z_i$  یا  $z_j$  اولین عدد بین  $z_i, z_{i+1}, \dots, z_j$  باشد که به عنوان محور انتخاب می‌شود.

در هر زیرآرایه، هر عنصر به طور تصادفی و با احتمال یکسان می‌تواند به عنوان محور انتخاب شود. بنابراین، در مجموعه  $\{z_i, z_{i+1}, \dots, z_j\}$  که شامل  $i + j - 1$  عنصر است، احتمال اینکه  $z_i$  یا  $z_j$  اولین عنصر انتخاب شده به عنوان محور باشد، برابر است با:

$$p_{ij} = \frac{2}{j - i + 1}$$

با استفاده از تغییر متغیر  $k = j - i$  (که از ۱ تا  $n - i$  تغییر می‌کند) و سپس تغییر ترتیب جمعها، داریم:

$$E[C(\sigma)] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1}$$

$$\sum_{k=1}^n \frac{1}{k} \leq \ln(n) + 1$$

به سری هارمونیک مشهور است و با استفاده از تقریب انتگرال ثابت می‌شود:

$$E[C(\sigma)] = 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k+1} \leq 2 \sum_{i=1}^{n-1} \sum_{k=2}^n \frac{1}{k} \leq 2n \sum_{k=2}^n \frac{1}{k}$$

در نتیجه خواهیم داشت:

بنابراین می‌توان نتیجه گرفت که مرتب‌سازی سریع از  $\Theta(n \log n)$  است. این تحلیل نشان می‌دهد که در حالت میانگین، مرتب‌سازی سریع بهینه‌ترین عملکرد را در بین الگوریتم‌های مقایسه‌ای ارائه می‌دهد.

تمرین: امید ریاضی تعداد نایه‌جایی‌ها در یک آرایه را محاسبه کنید (با فرض اینکه آرایه از میان جایگشت‌های مختلف اعضاء به صورت تصادفی انتخاب شده است) سپس برای مرتب‌سازی درجی در حالت میانگین نتیجه‌گیری کنید.

## انتخاب محور به صورت رندم

دقیق کنید در این تحلیل زمانی فرض شده است که آرایه به صورت تصادفی از بین همه جایگشت‌های ممکن اعداد به ما داده شده است. بنابراین فرض شده احتمال اینکه هر عددی از زیرآرایه‌ای که مرتب می‌کنیم انتخاب شود با هر عدد دیگری در آن زیرآرایه برابر است. (از این فرض در کجا اثبات استفاده شده است؟)

\*\*پاسخ:\*\* این فرض در مرحله محاسبه  $p_{ij} = \frac{2}{j-i+1}$  استفاده شده است. این احتمال بر اساس این است که هر عنصر در مجموعه  $\{z_i, z_{i+1}, \dots, z_j\}$  به یک اندازه احتمال دارد که اولین عنصر انتخاب شده به عنوان محور باشد.

حالا اگر چنین فرضی برای یک آرایه خاص (مثل آرایه‌ای که همیشه مرتب یا معکوس است) درست نباشد، تحلیل زمانی معتبر نخواهد بود. در این حالت با یک تغییر جزئی مشکل را رفع می‌کنیم: به جای اینکه عنصر انتهای آرایه را به عنوان محور انتخاب کنیم (که در پیاده‌سازی استاندارد ممکن است باعث بدترین حالت شود)، از تکنیکی به نام \*\*نمونه‌برداری تصادفی (Random Sampling)\*\* استفاده می‌کنیم: هر بار یک عنصر رندم را از بازه فعلی به عنوان محور انتخاب می‌کنیم.

بررسی کنید که در این حالت هم اثبات برقرار است و مستقل از اینکه آرایه اولیه به چه احتمالی چه جایگشتی از آرایه مرتب شده باشد، این روش، مرتب‌سازی سریع را به یک الگوریتم \*\*تصادفی تبدیل می‌کند که عملکرد میانگین آن، مستقل از ترتیب اولیه ورودی، تضمین می‌شود. (Randomized Algorithm)

In [ ]:

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل هفتم: مجموعه‌های مجزا

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

## فهرست

- یک مثال: روابط خانوادگی
- مجموعه‌های مجزا
- پیاده‌سازی با لیست
- پیاده‌سازی درختی

## مقدمه

در این فصل، به بررسی یکی از داده‌ساختارهای قدرتمند و پرکاربرد در علوم کامپیوتر، یعنی **\*مجموعه‌های مجزا (Disjoint Sets)** می‌پردازیم. این داده‌ساختار برای مدیریت مجموعه‌ای از عناصر که به گروههای (مجموعه‌های) جدا از هم تقسیم شده‌اند، بهینه شده است.

همان طور که در فصل‌های قبل دیدید، حالت‌هایی برای مرتب‌سازی سریع وجود دارد که باعث می‌شود عملکرد آن قابل قبول نباشد. اما با این وجود این الگوریتم «معمولًا» سریع است. برای دقیق کردن این ویژگی باید با دید احتمالاتی تحلیل را انجام دهیم. به عبارتی می‌گوییم الگوریتم باید در حالت میانگین بررسی شود. در زیر توضیحات بیشتری درباره معنای این کار می‌دهیم و سپس تحلیل را انجام می‌دهیم.

## یک مثال: روابط خانوادگی!

قبل از ورود به بحث مجموعه‌های مجزا، شاید بد نباشد مقداری به مسئله‌ی روابط خویشاوندی بین افراد جامعه توجه کنیم تا با دید بهتری وارد بحث اصلی شویم. هر فردی در جامعه در یک خانواده زندگی می‌کند و با مجموعه‌ای از افراد روابط خویشاوندی دارد. از زاویه‌ی دیگر اگر به دو خانواده متفاوت نگاه کنیم، در هر خانواده هر دو فردی با یکدیگر خویشاوند هستند و از طرف دیگر هیچ رابطه‌ی خویشاوندی بین فردی از یک خانواده با خانواده دیگر وجود ندارد. با توجه به این توضیحات، به بیان ریاضی می‌توان گفت رابطه‌ی خویشاوندی افراد جامعه را به مجموعه‌هایی مجزا (بدون اشتراک) افزایش می‌کند.

روابط خانوادگی چه زمانی تغییر می‌کند؟

زمانی که یک ازدواج جدید در جامعه شکل می‌گیرد، روابط خویشاوندی تغییر می‌کند و اگر ازدواج خویشاوند نبوده‌اند شکل گرفته باشد، باعث می‌شود تمام افراد این دو خانواده با یکدیگر خویشاوند شوند و یک خانواده بزرگ‌تر شکل گیرد.

تغییر دیگر در روابط خویشاوندی هنگام تولد یا مرگ یکی از افراد جامعه است که یک فرد به یک خانواده مشخص اضافه یا کم می‌شود. البته چون در مسئله‌ی اصلی که قرار است بررسی کنیم فرض اضافه یا کم شدن یک عضو وجود ندارد، این تغییر را هم در خانواده‌ها نادیده می‌گیریم.

حال فرض کنید می‌خواهیم در یک جامعه این روابط خوبشاوندی را ثبت کنیم و بتوانیم به ازای هر دو نفر تشخیص دهیم آیا آن دو در یک خانواده قرار دارند یا نه و همچنین در صورت به وجود آمدن تغییرات در روابط خوبشاوندی این تغییرات را هم بتوانیم به راحتی اعمال کنیم. چه راه حل‌هایی برای این کار وجود دارد؟

شاید اولین راه حلی که به ذهن برسد این باشد که به ازای هر خانواده لیستی داشته باشیم که همه‌ی اعضای آن خانواده را در آن لیست نوشته باشیم. حال به ازای هر دو نفر برای بررسی خوبشاوندی آن‌ها می‌توانیم در بین لیست‌ها بگردیم و آن دو نفر را پیدا کرده و ببینیم آیا در یک لیست قرار داشته‌اند یا نه! از طرف دیگر در صورت ازدواج دو نفر می‌توان باز لیست مربوط به هر یک را پیدا کرد و دو لیست را یکی کرد.

به نظر می‌رسد که پیدا کردن افراد در بین لیست همه‌ی خانواده‌ها خیلی وقت‌گیر باشد! باید سعی کنیم با راه حلی زمان صرف شده برای گشتن در بین همه لیست‌ها را از بین ببریم. فرض کنید که به هر خانواده یک اسم یکتا نسبت بدهیم (مثلاً نام خانوادگی یکی از بزرگان خانواده را) و از طرف دیگر لیست مرتب شده‌ای از کل افراد جامعه داشته باشیم و به ازای هر کدام اسم خانواده‌اش را مقابل نامشان در لیست نوشته باشیم! در این صورت چه طور می‌توانیم بررسی کنیم آیا دو نفر در یک خانواده هستند یا نه؟ کافی است هر کدام از آن دو را در لیست افراد پیدا کنیم و نام خانواده‌ی آن‌ها را با هم مطابقت دهیم! با توجه به مرتب‌بودن لیست اسامی همه افراد با استفاده از روش جست و جوی دودویی با هزینه‌ی نسبتاً کمی می‌توان فرد را در لیست پیدا کرد و نام خانواده‌ی وی را پیدا کرد. اما در صورت رخدان یک ازدواج باید یک نام برای خانواده جدید در نظر گرفت و در لیست افراد نام خانواده‌ی جدید را برای افراد این دو خانواده ثبت کرد.

برای کاهش این هزینه به نظر می‌رسد بهتر باشد پس از ادغام دو لیست خانواده، نام خانواده با تعداد اعضای کمتر را برابر نام خانواده دوم کرد تا برای تعداد افراد کمتری مجبور باشیم در لیست اسامی، نام خانواده را تغییر دهیم.

یک راه دیگر برای ثبت روابط خانوادگی توجه به سلسله‌مراتب نسلی در خانواده است. به طوری که برای هر فرد صرفاً پدر وی را نگه داریم (به جز بزرگ‌خاندان‌های هر خانواده که پدری ندارند). در این صورت به ازای هر فرد، بزرگ‌خاندانی که این فرد جز نوادگان او است را پیدا می‌کنیم. دو نفر در صورتی عضو یک خانواده‌اند که بزرگ‌خاندان یکسانی داشته باشند. حال با صورت گرفتن یک ازدواج روابط خانوادگی را چه طور تغییر دهیم که این دو خانواده با یک‌دیگر ادغام شوند؟!

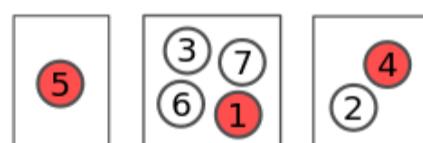
شاید این ایده با واقعیت خیلی مطابقت نداشته باشد اما به نظر می‌رسد اگر بزرگ‌خاندان خانواده‌ی با اعضای کمتر را به عنوان فرزند مجازی بزرگ‌خاندان خانواده‌ی دیگر ثبت کنیم، بزرگ‌خاندان هر دو خانواده یکی خواهد شد و این دو خانواده یک خانواده به شمار خواهند آمد.

در مثال بالا خانواده‌ها به عنوان مجموعه‌های مجزا در نظر گرفته شده بودند و ازدواج دو مجموعه را با یک‌دیگر ادغام می‌کرد. در ادامه به تعریف مسئله‌ی اصلی و راه حل‌هایی که شهود آن‌ها در مثال فوق مطرح شد می‌پردازیم.

## مجموعه‌های مجزا

در این فصل می‌خواهیم داده‌ساختاری بهینه برای نگهداری گردایه‌ای از مجموعه‌های مجزا از هم بسازیم که امکان ادغام دو مجموعه را علاوه بر بیان مجموعه‌ی هر عضو برای ما فراهم کند.

n شی داریم که هر کدام دقیقاً عضو یک مجموعه هستند. برای هر مجموعه یک نام در نظر می‌گیریم. معمولاً نام هر مجموعه برابر با یکی از اعضای آن که به عنوان نماینده آن مجموعه شناخته می‌شود قرار داده می‌شود. شکل زیر نمایی از سه مجموعه‌ی مجزا از هم با 7 عنصر است. عضوی از مجموعه که مجموعه با آن شناخته می‌شود (نام مجموعه) با رنگ قرمز مشخص شده است.



شکل 1: مثالی از سه مجموعه مجزا با 7 عنصر

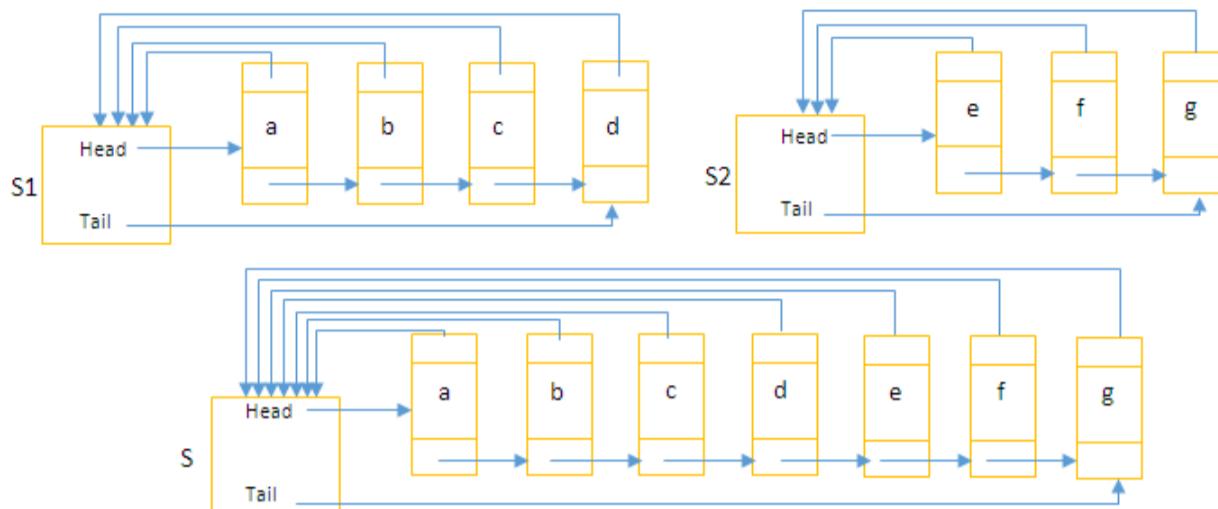
هدف ما طراحی داده‌ساختاری است که بتواند عملیات‌های زیر را بر روی اشیا پیاده‌سازی کند:

- **MakeSet(x)**: یک مجموعه‌ی جدید شامل تنها عنصر  $x$  ایجاد می‌کند.
- **Find(x)**: نام (نماینده) مجموعه‌ای که عنصر  $x$  در آن است را پیدا می‌کند.
- **Union(x, y)**: دو مجموعه‌ای که  $x$  و  $y$  عضو آن‌ها هستند را ادغام می‌کند و آن‌ها را به یک مجموعه واحد تبدیل می‌کند.

در ابتدا، همه‌ی عناصر در مجموعه‌های تکی و جدا از هم هستند (یعنی هر عنصر یک مجموعه مجازی خودش را تشکیل می‌دهد).

# پیاده‌سازی با لیست

یک روش ساده‌ی پیاده‌سازی مجموعه‌های مجزا، استفاده از **\*لیست‌های پیوندی (Linked Lists)** برای ذخیره هر مجموعه است. برای این کار، لازم است که به ازای هر مجموعه یک شیء (سرگروه) داشته باشیم که اشاره‌گری را به اولین (**tail**) و آخرین (**head**) عضو آن مجموعه نگه دارد. همچنین هر عضو لیست یک اشاره‌گر به سرگروه و عضو بعد از خود داشته باشد و هر مجموعه باید یک نماینده داشته باشد (متلا اولین عضو مجموعه می‌تواند نماینده باشد). این پیاده‌سازی در شکل زیر نشان داده شده است.



شکل ۲: پیاده‌سازی با لیست پیوندی

در این پیاده‌سازی، دستورات **find-set(x)** و **make-set(x)** در زمان  $O(1)$  انجام می‌شوند.

- **make-set(x)**: یک لیست پیوندی با یک عضو (x) می‌سازد.
- **head**: اول اشاره‌گر x به سرگروه مجموعه‌ی خود را دنبال می‌کند و سپس عنصری که اشاره‌گر **head** این شیء به آن اشاره می‌کند را برمی‌گرداند.

اما در مورد **union(x, y)** کار کمی پیچیده‌تر است. ساده‌ترین پیاده‌سازی آن به این صورت است که لیست y را به انتهای لیست x بچسبانیم و نماینده‌ی پیشین x بشود نماینده‌ی مجموعه‌ی حاصل.

برای پیدا کردن انتهای x از اشاره‌گر **tail** استفاده می‌شود و اشاره‌گر **next** این عضو انتهایی به اولین عضو لیست y تغییر می‌کند. همچنین برای همه‌ی اعضای y باید اشاره‌گر به سرگروه تغییر کند و به سرگروه مجموعه‌ی x اشاره کند.

با این پیاده‌سازی، به سادگی می‌توان دنباله‌ای از m عملیات روی n شیء انجام داد که زمانی از مرتبه‌ی  $\theta(n^2)$  ببرد. مثال زیر را در نظر بگیرید:

اعضای مجموعه‌های مجزا هستند. در آغاز n عمل **make-set** روی هر کدام انجام می‌دهیم و سپس 1 – n عمل **union** طبق جدول زیر:

دستور	تعداد بهروزرسانی‌ها
make-set( $x_1$ )	1
make-set( $x_2$ )	1
...	...
make-set( $x_n$ )	1
union( $x_1, x_2$ )	1
union( $x_2, x_3$ )	2
union( $x_3, x_4$ )	3
...	...
union( $x_{n-1}, x_n$ )	n-1

مجموع همه‌ی بهروزرسانی‌ها برای این 1 – 2n دستور برابر است با:

$$n + \sum_{i=1}^{n-1} i = n + \frac{(n-1)n}{2} = \theta(n^2)$$

پس هزینه‌ی سرشکن هر دستور از مرتبه‌ی  $\Theta(n)$  است. به راحتی می‌توان این هزینه را به  $O(\lg n)$  کاهش داد.

توجه داریم که در پیاده‌سازی بالا در صورت ادغام دو مجموعه با اندازه‌های متفاوت ممکن است مجبور باشیم عملیات بهروزرسانی را برای همه‌ی اعضای مجموعه‌ی بزرگتر انجام دهیم. این در حالی است که می‌توان با نگه داشتن اندازه‌ی لیست‌ها و گذاشتن این شرط که همیشه لیست بزرگ‌تر بچسبد، تعداد این بهروزرسانی‌ها را کم کرد.

با این تغییر همچنان ممکن است که یک دستور `union` هزینه‌ی زمانی  $\Omega(n)$  داشته باشد اما ثابت می‌کنیم که هزینه‌ی سرشکن برای  $m$  دستور شامل  $n$  دستور `make-set` بر روی  $n$  شیء، هزینه‌ی سرشکن  $O(\lg n)$  دارد:

اول یک کران بالا برای تعداد دفعاتی که اشاره‌گر یک عضو خاص به سرگروهش تغییر می‌کند به دست می‌آوریم. یک عضو به نام  $x$  را در نظر می‌گیریم. می‌دانیم هر بار که این اشاره‌گر  $x$  تغییر کرده،  $x$  در مجموعه‌ی کوچک‌تر (نسبت به مجموعه‌ای که در آن ادغام می‌شود) بوده. پس بار اول که تغییر کرده مجموعه‌ی حاصل باید حداقل دو عضو داشته باشد و بار دوم حداقل چهار عضو و بار  $\lceil \lg k \rceil$  ام، حداقل  $k$  عضو ( $k \leq n$ ).  
از آن جا که بزرگ‌ترین مجموعه حداکثر  $n$  عضو دارد، اشاره‌گر  $x$  حداکثر  $\lceil \lg n \rceil$  بار تغییر کرده. همچنین از آن جا که تعداد کل دستورات `union` حداکثر برابر  $1 - n$  است، کل زمان صرف شده برای بهروزرسانی اشاره‌گرها از  $O(n \lg n)$  است و هزینه‌ی سرشکن هر دستور `union` برابر  $O(1)$  است.

اگر تغییر اندازه‌ی لیست‌ها و تغییر اشاره‌گر `tail` در `union` که از  $O(1)$  است و دستورات `find-set` و `make-set` که از  $O(1)$  هستند را هم در نظر بگیریم، برای این  $m$  دستور زمان  $O(m + n \lg n)$  صرف می‌شود.

در ادامه پیاده‌سازی مجموعه‌های مجزا با لیست را برای ۷ رأس آورده‌ایم.

```
In [1]: class DisjointSet(object):
    def __init__(self, n):
        # عنصر مقداردهی اولیه می‌کند n متد سازنده: مجموعه‌های مجزا را با
        # هر عنصر در ابتدا یک مجموعه مجزا است.
        self.n = n
        # set_lists: لیستی از لیست‌ها که هر لیست یک مجموعه (کامپوننت) را نشان می‌دهد.
        # set_lists[i]: است i یک لیست است که شامل تمام عناصری است که نماینده‌شان [i] است.
        self.set_lists = [[i] for i in range(n)]
        # set: لیستی که برای هر عنصر، نماینده مجموعه آن را ذخیره می‌کند.
        # set[i]: نماینده مجموعه عنصر i.
        self.set = [i for i in range(n)]

    def find(self, u):
        # در آن است را پیدا می‌کند u مجموعه‌ای که عنصر (representative) این تابع نماینده.
        # دارد set_lists در این پیاده‌سازی، نماینده هر مجموعه، اندیس خودش را در
        # return self.set[u]

    def union(self, u, v):
        # عضو آنها هستند را ادغام می‌کند v و u این تابع دو مجموعه‌ای که
        # پیدا کردن نماینده‌گان دو عنصر # self.find(u), self.find(v) را کاهش می‌دهد.
        # if u_set == v_set: قبلاً در یک مجموعه بودند v و u اگر # اگر
        # return False. عملیات ادغام انجام نمی‌شود.
        # بهینه‌سازی: همیشه لیست کوچک‌تر را به لیست بزرگ‌تر ادغام می‌کنیم.
        # این کار تعداد بهروزرسانی‌های اشاره‌گرها را کاهش می‌دهد.
        if len(self.set_lists[u]) > len(self.set_lists[v]):
            u_set, v_set = v_set, u_set # نماینده مجموعه کوچک‌تر باشد اطمینان از اینکه
            u_set, v_set = v_set, u_set # اطمینان از اینکه u_set = []
        # به لیست بزرگ‌تر (u_set) ادغام لیست کوچک‌تر
        for i in self.set_lists[v]:
            self.set_lists[u].append(i) # بزرگ‌تر
            self.set[i] = u_set # بهروزرسانی نماینده برای هر عنصر منتقل شده.
        # پس از ادغام، لیست مجموعه کوچک‌تر را خالی می‌کنیم (اختیاری، اما برای وضوح)
        # برای جلوگیری از پردازش مجدد با سردرگمی # [] = []
        self.set_lists[v] = []
        # عملیات ادغام با موفقیت انجام شد
        return True

    def print_all_sets(self):
        # این تابع تمام مجموعه‌های مجزا فعلی را چاپ می‌کند.
        count = 0
        for i in range(self.n):
            # بیمامیش بر روی تمام اندیس‌های ممکن برای نماینده # نماینده یک مجموعه است i باشد، یعنی i
            # set[i] == i اگر # اینکه لیست خالی نباشد (پس از ادغام) # 0: > 0: اطمینان از اینکه
            if self.set[i] == i and len(self.set_lists[i]) > 0: # اینکه لیست خالی نباشد
                count += 1
                print("component " + str(count) + ":" + str(self.set_lists[i]))

    # مثال استفاده از کلاس
    n = 7 # تعداد کل عناصر (از 0 تا 6)
    unions = [(1, 3), (2, 4), (1, 6), (3, 6), (1, 4)] # که قرار است انجام شود union عملیات‌های
    # انجام دهد
    dsu = DisjointSet(n) # ایجاد یک شیء DisjointSet با 7 عنصر
```

```
dsu.print_all_sets() # از هر چاپ وضعیت مجموعه‌ها پس
print()

union of 1, 3 was OKAY
now components are:
component 1:[0]
component 2:[2]
component 3:[3, 1]
component 4:[4]
component 5:[5]
component 6:[6]

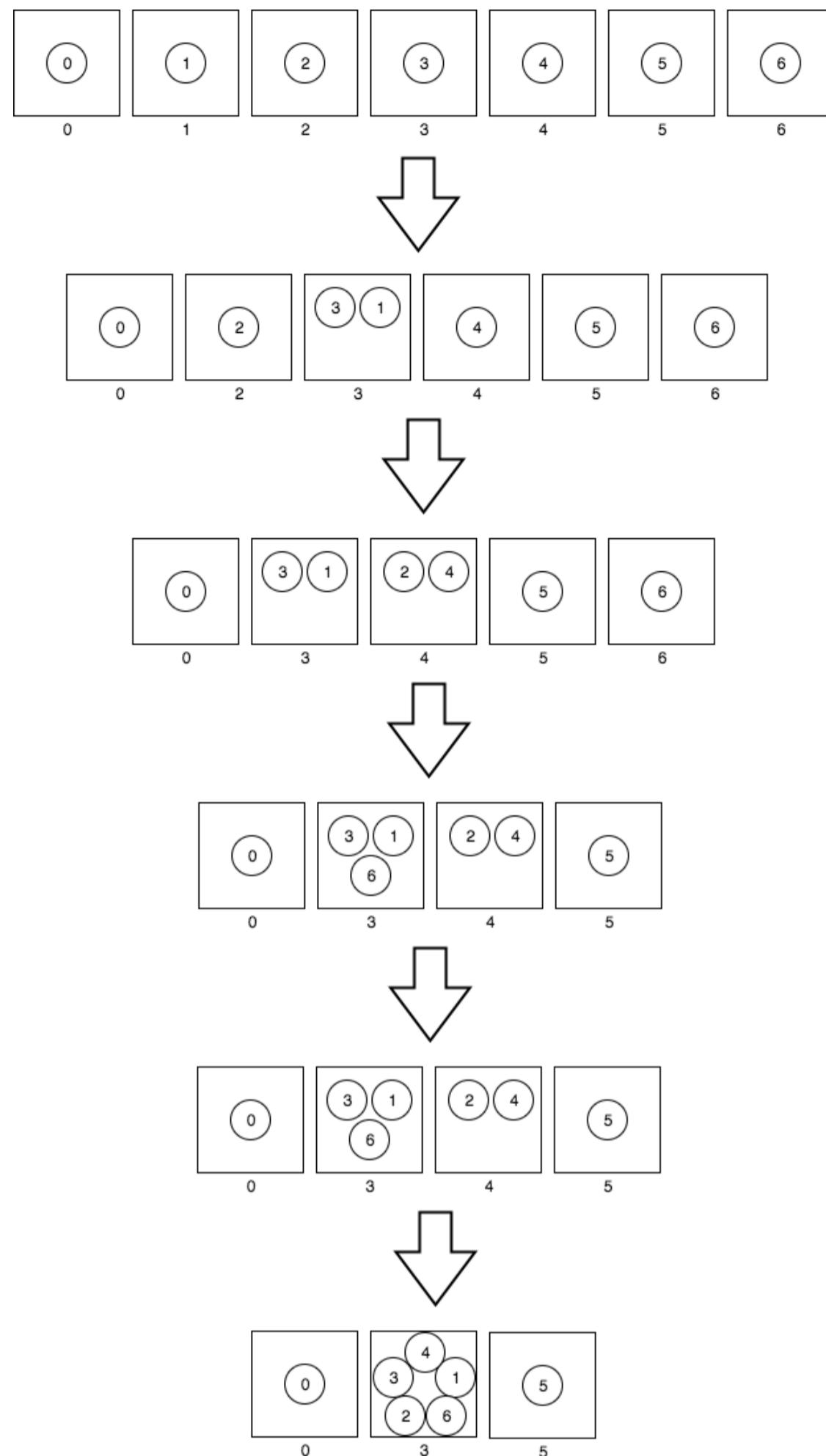
union of 2, 4 was OKAY
now components are:
component 1:[0]
component 2:[3, 1]
component 3:[4, 2]
component 4:[5]
component 5:[6]

union of 1, 6 was OKAY
now components are:
component 1:[0]
component 2:[3, 1, 6]
component 3:[4, 2]
component 4:[5]

union of 3, 6 was NOKAY
now components are:
component 1:[0]
component 2:[3, 1, 6]
component 3:[4, 2]
component 4:[5]

union of 1, 4 was OKAY
now components are:
component 1:[0]
component 2:[3, 1, 6, 4, 2]
component 3:[5]
```

وضعیت مجموعه‌ها پس از اجرای عملیات‌های ادغام در شکل زیر مشخص است:



شکل ۲: وضعیت مجموعه‌ها بر از اجرای هر ادغام

پیاده‌سازی درختی

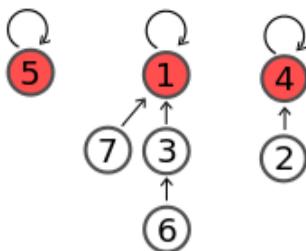
با توجه به مسئله‌ای، که قرار است با استفاده از این داده‌ساختار حل کنیم، ممکن است در شرایط مختلف بساده‌سازی‌های مختلف آن نسبت به هم برتری داشته باشند.

در ادامه بیادهسازی این داده‌ساختار را با استفاده از \*\*مجموعه‌ای از درخت‌ها (Forest of Trees)\*\* انجام می‌دهیم.

به این‌ها، هر عنصر یک رأس، دیگر رأس نمایم. هر رأس، یعنی دو قطب می‌باشد.

هر مجموعه به صورت یک گراف جهت‌دار بدون دور است، به طوری که از هر رأس متناظر از اعضای مجموعه به رأس سرگروه مسیر جهت‌دار وجود دارد. (فقط سرگروه مجموعه یال خروجی به عنصر دیگری ندارد و به خودش یا زادارد).

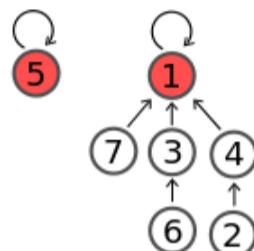
شکل زیر نمایی از یک پیاده‌سازی درختی برای مثال شکل-۱ است. در این پیاده‌سازی، نماینده هر مجموعه، ریشه درخت مربوط به آن مجموعه است.



شکل ۳: مثالی از سه مجموعه مجزا با ۷ عنصر (پیاده‌سازی درختی)

برای ادغام کردن دو مجموعه، پدر سرگروه اولی را سرگروه دومی قرار می‌دهیم؛ یعنی طوفه‌ی سرگروه اولی را حذف و یک یال جهت‌دار از آن به سرگروه دومی اضافه می‌کنیم.

شکل-۴ نمایی از مجموعه‌های مجزا بعد از اجرای دستور Union(1,4) است.



شکل ۴: ادغام عناصر ۱ و ۴ از مجموعه

برای پیدا کردن سرگروه  $x$ ، مسیر جهت‌دار آن را طی می‌کنیم تا به سرگروه برسیم. هزینه‌ی Find( $x$ ) در بدترین حالت از مرتبه ارتفاع درخت مجموعه‌ی  $x$  است. اما ارتفاع درخت می‌تواند از مرتبه  $(n/\Theta)$  باشد!

پس تا اینجا هیچ بهبودی نسبت به پیاده‌سازی با لیست نداشته‌ایم.

برای این پیاده‌سازی هم روش‌هایی برای بالا بردن کارایی وجود دارد. در این روش‌ها تلاش می‌شود ارتفاع درخت تا حد ممکن کم نگه داشته شود.

اولین روش شبیه همان است که در پیاده‌سازی با لیست انجام شد. یعنی اندازه‌ی هر درخت را نگه‌داریم و هنگام union کردن ریشه‌ی درخت کوچک‌تر را فرزند ریشه‌ی درخت بزرگ‌تر کنیم (union-by-size).

اما روش ساده‌تر آن است که برای هر رأس درخت، یک مرتبه (rank) نگه‌داریم. مرتبه یک کران بالا برای ارتفاع رأس است. در این روش که union-by-rank نام دارد، ریشه‌ی با مرتبه‌ی کمتر به ریشه‌ی با مرتبه‌ی بیشتر اشاره می‌کند.

توجه داریم که مرتبه با ارتفاع تفاوت است. مرتبه از تعریف زیر به دست می‌آید:

وقتی یک مجموعه ساخته می‌شود (make-set)، یک عضو با مرتبه‌ی صفر دارد. وقتی دو مجموعه با مرتبه‌های برابر union شوند، مرتبه‌ی درخت حاصل یکی بیشتر از مرتبه‌ی قبلی‌است.

دلیل استفاده از مرتبه به جای ارتفاع یا عمق آن است که \*\*فشرده‌سازی مسیر (Path Compression)\*\* ارتفاع یک درخت را تغییر می‌دهد اما مرتبه‌ی آن را نه!

با اجرای این ایده و استفاده از لم زیر می‌توان نشان داد که ارتفاع درخت‌ها هیچ وقت از  $n \lg n$  بیشتر نمی‌شود! و چون هزینه‌ی union متناسب با ارتفاع درخت است، هزینه‌ی آن از  $O(\lg n)$  است.

\*لم: \*\*یک درخت با ارتفاع  $h$  دست کم  $2^h$  رأس دارد.

تمرین ۱: لم بالا را ثابت کنید.

در ادامه پیاده‌سازی درختی را آورده‌ایم.

```
In [2]: class Node(object):
    def __init__(self, label):
        # جدید برای پیاده‌سازی درختی مجموعه‌های مجزا (Node) متد سازنده: یک گره
        # برچسب یا مقدار گره (مجموعاً اندیس عنصر) self.label = label #
        # اشاره‌گر به پدر گره. در ابتدا هر گره پدر خودش است (نماینده مجموعه خودش)
        self.par = self #
        # رتبه (rank) گره، برای بهینه‌سازی union (union-by-rank).
        self.rank = 0 #
```

```
class DisjointSet(object):
    def __init__(self, n):
        # عنصر مقداردهی اولیه می‌کند n متد سازنده: مجموعه‌های مجزا را با
        # هر عنصر در ابتدا یک مجموعه مجزا است که خودش نماینده آن است
        self.n = n
        self.nodes = [Node(i) for i in range(n)] # عنصر n برای لیستی از اشیاء #
```

```
def find(self, u_node): # است
    # در آن است را پیدا می‌کند u_node این تابع نماینده (ریشه) مجموعه‌ای که عنصر
    # به صورت بازگشتی پدرها را دنبال می‌کند تا به ریشه برسد #
```

```

اگر گره، پدر خودش بود، یعنی ریشه مجموعه است # اگر u_node == u_node.par: # بدون فشرده‌سازی مسیر
    return u_node
# بدون فشرده‌سازی: return self.find(u_node.par)
# با فشرده‌سازی مسیر: (این بخش در ادامه اضافه خواهد شد)
# فعلاً بدون فشرده‌سازی مسیر return self.find(u_node.par) # بدون فشرده‌سازی مسیر

def union(self, u_node, v_node): # ورویدی‌ها اشیاء Node. هستند.
    # عضو آنها هستند را ادغام می‌کند و u_node و v_node این تابع دو مجموعه‌ای که
    # u پیدا کردن ریشه مجموعه # این کار به حفظ ارتفاع کم درختان کمک می‌کند
    u_root = self.find(u_node) # پیدا کردن ریشه مجموعه
    v_root = self.find(v_node) # پیدا کردن ریشه مجموعه

    # اگر دو عنصر قبلاً در یک مجموعه بودند.
    if u_root == v_root: # return False # عملیات ادغام نمی‌شود.

    # (ریشه با رتبه کمتر را فرزند ریشه با رتبه بیشتر می‌کند) Union by Rank.
    # این کار به حفظ ارتفاع کم درختان کمک می‌کند
    if u_root.rank > v_root.rank:
        u_root, v_root = v_root, u_root # ریشه با رتبه کمتر باشد u_root اطمینان از اینکه

    # ادغام دو مجموعه: ریشه مجموعه کوچک‌تر را فرزند ریشه مجموعه بزرگ‌تر می‌کند
    u_root.par = v_root

    # به روزرسانی رتبه: اگر رتبه‌ها برابر بودند، رتبه ریشه جدید را افزایش می‌دهد
    if u_root.rank == v_root.rank:
        v_root.rank += 1

    # عملیات ادغام با موفقیت انجام شد
    return True # print_all_components(self):
    # این تابع تمام مجموعه‌های مجزا فعلی را چاپ می‌کند
    # برای هر گره، نماینده آن را پیدا کرده و گروه‌بندی می‌کند find با استفاده از
    components_map = {}
    for i in range(self.n):
        # به لیست مربوط به آن نماینده i و اضافه کردن گره i پیدا کردن نماینده گره
        representative = self.find(self.nodes[i]).label
        if representative not in components_map:
            components_map[representative] = []
        components_map[representative].append(i)

    count = 0
    for rep_label in sorted(components_map.keys()): # چاپ مجموعه‌ها به ترتیب نماینده
        count += 1
        print(f"component {count}: {components_map[rep_label]}") # که قرار است انجام شود union عملیات‌های

# پیاده‌سازی درختی DisjointSet مثال استفاده از کلاس
n = 7 # تعداد کل عناصر (از 0 تا 6)
edges = [(1, 3), (2, 4), (1, 6), (3, 6), (1, 4)] # این قرار است انجام شود union عملیات‌های

dsu = DisjointSet(n) # ایجاد یک شیء DisjointSet با 7 عنصر

for u, v in edges:
    # پاس می‌دهیم dsu را از لیست nodes در اشیاء برای union،
    # این را از لیست nodes در اشیاء برای union،
    success = dsu.union(dsu.nodes[u], dsu.nodes[v])
    print(f"union of {u}, {v} was {'OKAY' if success else 'NOKAY'}")
    print("now components are: ")
    dsu.print_all_components() # چاپ وضعیت مجموعه‌ها پس از هر union
    print("")
```

union of 1, 3 was OKAY  
now components are:  
component 1: [0]  
component 2: [2]  
component 3: [1, 3]  
component 4: [4]  
component 5: [5]  
component 6: [6]

union of 2, 4 was OKAY  
now components are:  
component 1: [0]  
component 2: [1, 3]  
component 3: [2, 4]  
component 4: [5]  
component 5: [6]

union of 1, 6 was OKAY  
now components are:  
component 1: [0]  
component 2: [1, 3, 6]  
component 3: [2, 4]  
component 4: [5]

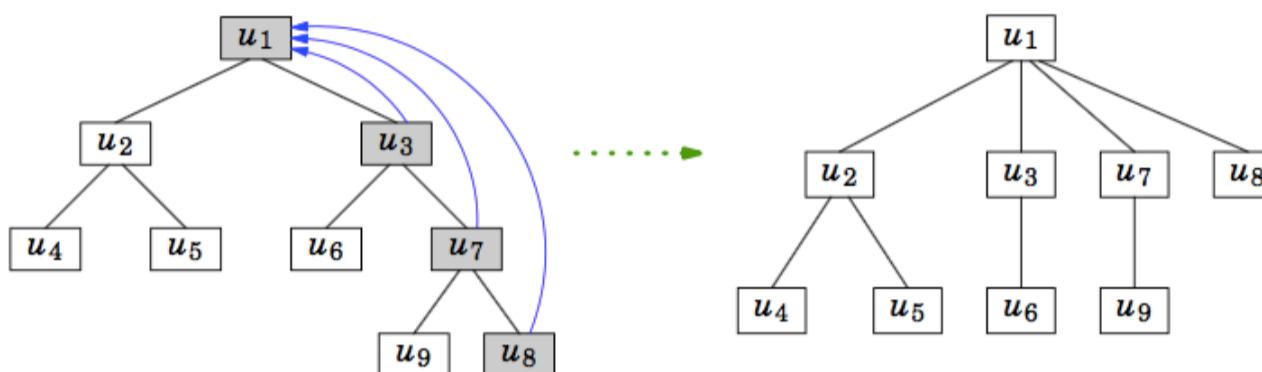
union of 3, 6 was NOKAY  
now components are:  
component 1: [0]  
component 2: [1, 3, 6]  
component 3: [2, 4]  
component 4: [5]

union of 1, 4 was OKAY  
now components are:  
component 1: [0]  
component 2: [1, 2, 3, 4, 6]  
component 3: [5]

فشرده‌سازی مسیر (Path Compression) یک بهینه‌سازی بسیار قدرتمند برای عملیات **Find** در پیاده‌سازی درختی مجموعه‌های مجزا است. این ایده به این صورت است که هر بار که دستور **Find**( $x$ ) اجرا می‌شود، یال خروجی همه‌ی رأس‌های مسیر از  $x$  تا سرگروه (ریشه) را مستقیماً به سرگروه وصل می‌کنیم. این کار باعث می‌شود که در فرآخوانی‌های بعدی **Find** برای همین گره‌ها یا گره‌های زیر آن‌ها، مسیر جستجو کوتاه‌تر شود.

در ابتدا شاید فشرده‌سازی مسیر خیلی سودمند به نظر نیاید، اما می‌توان نشان داد با اجرای این ایده (همراه با **union-by-size** یا **union-by-rank**)، هزینه‌ی دستور **Find** به طور سرشکن به  $O(\alpha(n))$  کاهش می‌یابد. ( $\alpha(n)$  تابع معکوس آکرمان (Inverse Ackermann Function) است که می‌توان گفت مقدار این تابع برای تمام  $n$ ‌هایی که در مسائل با آن‌ها سر و کار داریم (حتی برای  $n$ ‌های بسیار بزرگ مانند تعداد اتم‌های جهان!) کمتر از ۵ می‌باشد. این یعنی عملًا زمان اجرای **Find** به  $O(1)$  نزدیک است).

شکل زیر تاثیر فشرده‌سازی مسیر را در کاهش ارتفاع درخت نشان می‌دهد.



شکل ۵: فشرده‌سازی مسیر

In [3]:

```
# می‌شود find در کلاس DisjointSet این پیاده‌سازی جایگزین متد # تعريف شود: این تابع باید به عنوان یک متد در کلاس # برای تست، می‌توان آن را به کلاس موجود اضافه کرد #
```

```
def find_with_path_compression(self, u_node):
    """
    در آن است را پیدا می‌کند u_node این تابع نماینده (ریشه) مجموعه‌ی که عنصر گره‌ای که می‌خواهیم نماینده‌اش را پیدا کنیم: u_node (Node).
    Args:
        self: نمونه کلاس DisjointSet.
    Returns:
        گره ریشه (نماینده مجموعه): Node.
    """
    if u_node == u_node.par: # اگر گره، پدر خودش بود، یعنی ریشه است
        return u_node
    # بازگشتی: پدر گره را به نماینده نهایی وصل می‌کند (فسرده‌سازی مسیر)
    u_node.par = self.find_with_path_compression(u_node.par)
    return u_node.par

# را با این تابع جایگزین کنید find کلاس DisjointSet برای تست، می‌توانید متد # DisjointSet.find = find_with_path_compression

# در کلاس بهروز شده است find همان مثال قبلی، اما با فرض اینکه) مثال استفاده # n = 7
# edges = [(1, 3), (2, 4), (1, 6), (3, 6), (1, 4)]
# dsu_pc = DisjointSet(n)
# کار کند، باید آن را به کلاس اضافه کنید برای اینکه # را مستقیماً در کلاس تغییر دهید # # با این نسخه جایگزین شده است find در کلاس DisjointSet برای این مثال، فرض می‌کنیم متد # # dsu_pc.find = find_with_path_compression.__get__(dsu_pc, DisjointSet) برای اتصال به نمونه #
```

```
# for u, v in edges:
#     # این union، این union، Node را از لیست nodes در dsu_pc برای اینجا می‌دهیم
#     success = dsu_pc.union(dsu_pc.nodes[u], dsu_pc.nodes[v])
#     print(f"union of {u}, {v} was {'OKAY' if success else 'NOKAY'}")
#     print("now components are: ")
#     dsu_pc.print_all_components()
#     print("")

# # را فرآخوانی کنیم Find برای مشاهده تاثیر فشرده‌سازی مسیر، می‌توانیم چندین بار # print("\nAfter some Find operations (e.g., dsu_pc.find(dsu_pc.nodes[1])):")
# dsu_pc.find(dsu_pc.nodes[1])
# dsu_pc.find(dsu_pc.nodes[3])
# dsu_pc.find(dsu_pc.nodes[2])
# print("Components after path compression (may show flatter trees):")
# dsu_pc.print_all_components()
```

در هنگام استفاده از روش فشرده‌سازی مسیر، در عمل نیاز به چک کردن ارتفاع (یا رتبه) دو درخت در هنگام ادغام نیست. زیرا با یک بار فراخوانی تابع `Find`، یک مسیر به طول  $h$  تا ریشه به  $h$  مسیر تک یالی تبدیل می‌شوند (یعنی تمام گره‌های روی مسیر مستقیماً به ریشه وصل می‌شوند). این بهینه‌سازی، حتی اگر به تنها یک و بدون **union-by-rank** استفاده شود، به طور قابل توجهی عملکرد را بهبود می‌بخشد. با ترکیب هر دو بهینه‌سازی (**path compression** و **union-by-rank**)، به بهترین عملکرد ممکن برای مجموعه‌های مجزا دست پیدا می‌کنیم.

## برای مطالعه بیشتر:

اکیدا توصیه می‌شود برای درک بهتر داده‌ساختار درختی و به خصوص کارایی فشرده‌سازی مسیر به این [لينک](#) مراجعه کنید و حالت‌های مختلف را تست کنید. این ابزار بصری به شما کمک می‌کند تا تأثیر هر عملیات و بهینه‌سازی را به صورت گام به گام مشاهده کنید.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل هشتم، بخش اول: مرتبه‌آماری

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

- الگوریتم پیداکردن  $k$  امین مرتبه آماری
- انتخاب سریع
- میانه میانه‌ها

# الگوریتم پیداکردن $k$ امین مرتبه آماری

مسائله‌ای که در این فصل به آن می‌پردازیم، یافتن  $k$  امین کوچکترین عدد\*\* در یک لیست (یا آرایه) است. این مسئله با عنوان  $k^{**}$  امین مرتبه‌ی آماری ( $k$ -th Order Statistic) نیز شناخته می‌شود. به عنوان مثال، اگر  $k = 1$  باشد، به دنبال کوچکترین عنصر هستیم؛ اگر  $n = k$  باشد، به دنبال بزرگترین عنصر هستیم؛ و اگر  $\lfloor (n+1)/2 \rfloor = k$  باشد، به دنبال میانه (median) هستیم.

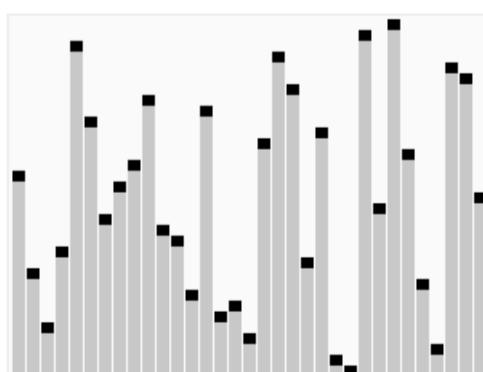
اولین راهی که برای این کار به ذهن می‌رسد، مرتب کردن آرایه و انتخاب عنصر  $k$  ام آن است. اگر از الگوریتم‌های مرتب‌سازی با پیچیدگی  $O(n \log n)$  (مانند Merge Sort یا Heapsort) استفاده کنیم، پیچیدگی زمانی کل این رویکرد نیز  $O(n \log n)$  خواهد داشت.

ما در این دفترچه سعی می‌کنیم الگوریتم‌هایی با زمان اجرای بهتر برای این مسئله پیدا کنیم، یعنی الگوریتم‌هایی که بتوانند این کار را در زمان خطی ( $O(n)$ ) انجام دهند.

## انتخاب سریع (Quicksort)

انتخاب سریع (Quicksort)\*\* از رویکرد یکسانی با مرتب‌سازی سریع (Quicksort) بهره می‌برد. این الگوریتم نیز یک الگوریتم  $^{**}$  تقسیم و حل (Divide and Conquer) است که به صورت بازگشتی عمل می‌کند.

فرآیند آن به این صورت است: ابتدا عضوی را به عنوان \*\*عنصر محوری (pivot)\*\* انتخاب کرده و داده‌ها را بر پایه‌ی آن به دو قسمت تقسیم می‌کند (عناصری که کمتر از عنصر محوری هستند و عناصری که بیشتر از عنصر محوری هستند). سپس با توجه به طول دو قسمت و مقدار  $k$  (مرتبه آماری مورد نظر)، بازگشت روی یکی از قسمت‌ها انجام می‌دهد.



شکل ۱: انتخاب سریع

همانند مرتب‌سازی سریع، انتخاب سریع نیز دارای عملکرد حالت متوسط خوبی است، اما به شدت وابسته به عنصرهای محوری انتخاب شده می‌باشد. اگر عنصرهای محوری انتخاب شده مناسب باشند (به این معنای که تعداد داده‌ها را در هر مرحله با نسبت معینی کاهش دهند، مثلًا به دو بخش تقریباً مساوی تقسیم کنند)، تعداد داده‌ها به صورت نمایی کاهش می‌یابد و در نتیجه کارکرد الگوریتم به صورت خطی می‌شود.

اما اگر عنصرهای محوری نامناسب به صورت پیاپی انتخاب شوند، مانند حالتی که در هر مرحله تنها یک عضو از داده‌ها کاسته شود (مثلاً همیشه کوچکترین یا بزرگترین عنصر به عنوان محور انتخاب شود)، به بدترین عملکرد الگوریتم با  $O(n^2)$  منجر خواهد شد.

در بخش بعد برای پیدا کردن محور مناسب (که تضمین کند افزار متعادل است)، الگوریتم [میانه میانه‌ها](#) (Median of Medians) را معرفی می‌کنیم.

## تمرین ۱:

$$T(n) \leq cn$$
$$T(n) \leq an + \frac{1}{n} \sum_{i=1}^n T(\max(i, n-i-1))$$

ثابت کنید پیچیدگی زمانی الگوریتم بالا به طور متوسط خطی است. راهنمایی:

## تمرین ۲:

چرا به طور متوسط پیچیدگی زمانی این الگوریتم از  $O(n \log n)$  است درحال که در مرتبسازی سریع  $O(n)$  است؟

\*\*پاسخ:\*\* تفاوت اصلی در این است که مرتبسازی سریع (Quicksort) به صورت بازگشتی روی \*\*هر دو بخش\*\* (چپ و راست محور) فراخوانی می‌شود، در حالی که انتخاب سریع (Quickselect) فقط روی \*\*یکی از دو بخش\*\* (آن بخشی که شامل  $k$ -امین عنصر است) فراخوانی می‌شود. این تفاوت باعث می‌شود که مجموع کار در هر سطح از درخت بازگشتی Quickselect خطی بماند، در حالی که در Quicksort به دلیل پیمایش هر دو شاخه، به  $O(n \log n)$  می‌رسد.

در ادامه پیاده‌سازی از الگوریتم بالا را ارائه می‌دهیم. خوب است که جستجو کنید و شبه‌کدهای دیگر از جمله کتاب CLRS را ببینید. به عنوان مثال [شکل ۱](#) پیاده‌سازی این الگوریتم به صورت درجا است. پیاده‌سازی ارائه شده زیر کمی متفاوت است و از لیست‌های جدید استفاده می‌کند.

```
In [1]: import random # برای انتخاب تصادفی عنصر محوری

def random_select(a, k):
    """
    پیدا می‌کند Quickselect با استفاده از الگوریتم 'a' امین کوچکترین عنصر را در لیست-k. این تابع
    (کوچکترین عنصر است k=0 از 0 شروع می‌شود، یعنی k اندیس).
    Args:
        a (list): لیست اعداد.
        k (int): اندیس مرتبه آماری مورد نظر (از 0 تا len(a)-1).
    Returns:
        any: امین کوچکترین عنصر در لیست-k.
    """
    # در محدوده معتبر است k اطمینان حاصل شود که
    if not (0 <= k < len(a)):
        raise ValueError("k must be a valid index within the list range.")

    # به صورت تصادفی از لیست (pivot) انتخاب یک عنصر محوری
    v = a[random.randint(0, len(a) - 1)]

    # تقسیم لیست به سه بخش: کوچکتر از محور، مساوی محور، بزرگتر از محور
    lt = [x for x in a if x < v] # عناصر کوچکتر از محور
    eq = [x for x in a if x == v] # عناصر مساوی محور
    gt = [x for x in a if x > v] # عناصر بزرگتر از محور

    # تصمیم‌گیری برای ادامه جستجو در کدام بخش
    if len(lt) > k:
        # در بخش عناصر کوچکتر قرار داشت، جستجو را در آن بخش ادامه بده اگر
        return random_select(lt, k)
    elif len(eq) + len(lt) > k:
        # امین عنصر است k در بخش عناصر مساوی محور قرار داشت، محور همان k اگر
        return v
    else:
        # در بخش عناصر بزرگتر قرار داشت، جستجو را در آن بخش ادامه بده اگر
        # تنظیم کن 'eq' و 'lt' جدید را با توجه به اندازه بخش‌های
        # k باشد
        return random_select(gt, k - len(eq) - len(lt))

# مثال استفاده
a = [5, 6, 0, 1, 10, 12, -1, 8, 100]

print("k=0:", random_select(a, 0)) # (کوچکترین عنصر)
print("k=1 باشد #:", random_select(a, 1)) # (سومین کوچکترین عنصر)
print("k=2 باشد #:", random_select(a, 2)) # (سومین کوچکترین عنصر)
print("k=5 باشد #:", random_select(a, 5)) # (ششمین کوچکترین عنصر)
print("k=8 باشد #:", random_select(a, 8)) # (بزرگترین عنصر)

k=0: (کوچکترین عنصر) -1
k=2: (سومین کوچکترین عنصر) 1
k=5: (ششمین کوچکترین عنصر) 8
k=8: (بزرگترین عنصر) 100
```

# میانه میانه‌ها (Median of Medians)

الگوریتم \*\*میانه میانه‌ها\*\* (Median of Medians) یک الگوریتم انتخاب (selection algorithm) است که به طور قطعی (deterministic) و در زمان خطی ( $O(n)$ ) عمل می‌کند. این الگوریتم برای پیدا کردن میانه‌ی آرایه و یا به طور کلی‌تر،  $i$  امین عنصر یک آرایه طراحی شده است و بخلاف Quicksort، عملکرد بدترین حالت آن نیز خطی است.

الگوریتم به صورت زیر می‌باشد:

\* تقسیم به دسته‌ها: آرایه را به  $5/n$  دسته‌ی ۵ تایی (به جز احتمالاً آخرین دسته که کمتر از ۵ عضو دارد) تقسیم می‌کنیم.

\*\* یافتن میانه هر دسته: هر دسته را مرتب کرده (مثلاً با Insertion Sort که برای اندازه‌های کوچک کارآمد است) و میانه‌ی آن را پیدا می‌کنیم.

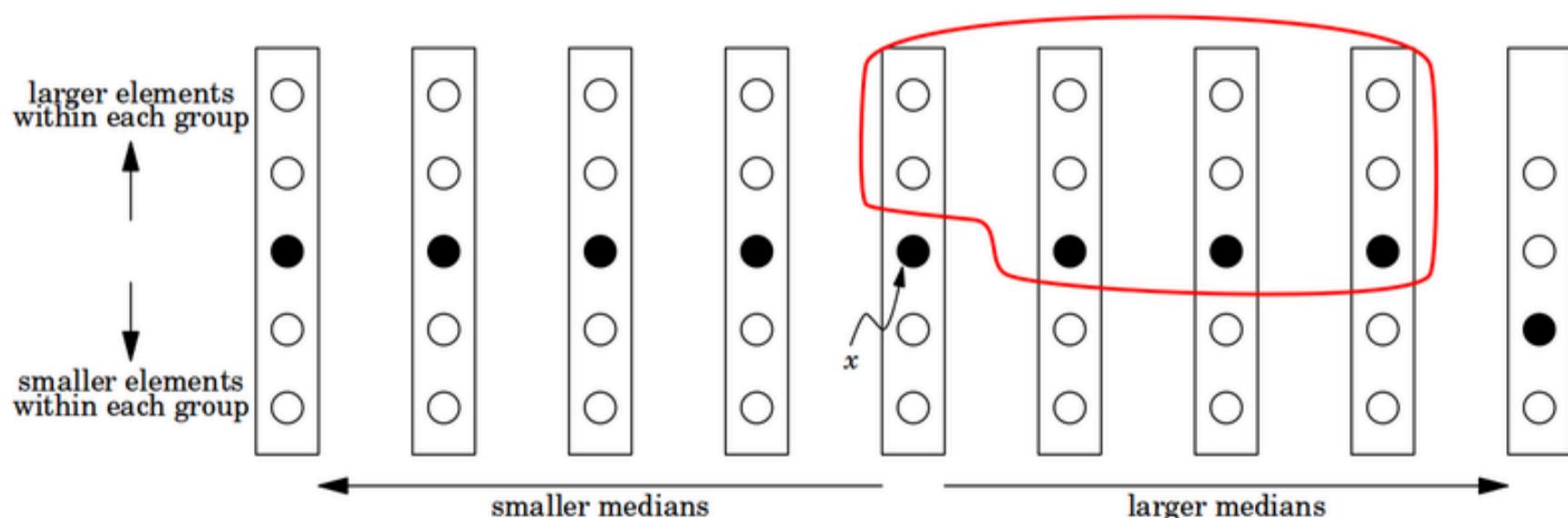
\*\* یافتن میانه میانه‌ها: به صورت بازگشتی، میانه‌ی میانه‌های  $5/n$  دسته‌ی یاد شده را انتخاب می‌کنیم. فرض کنید این میانه،  $x$  باشد. این  $x$  به عنوان عنصر محوری (pivot) برای مرحله بعدی استفاده می‌شود.

\* افزایش و بازگشت: حال می‌توانیم آرایه را براساس  $x$  به سه دسته (اعداد کوچکتر از  $x$ ، اعداد مساوی  $x$ ، و اعداد بزرگتر از  $x$ ) تقسیم کنیم. با توجه به اندازه‌ی دسته‌ی اول (عناصر کوچکتر از  $x$ ) و مقدار  $i$  (مرتبه آماری مورد نظر)، عدد مورد نظر را در دسته مناسب به صورت بازگشتی به کمک الگوریتم گفته شده پیدا کنیم.

\*\* تحلیل پیچیدگی: فرض کنید  $x$  میانه‌ای باشد که در مرحله ۳ بدست آورده‌ایم. حال تمام دسته‌هایی که میانه آنها از  $x$  کمتر است را در نظر بگیرید. تعداد آنها حداقل  $\lceil n/10 \rceil = \lceil n/5 \rceil / 2$  است.

در هر یک از این دسته‌ها (که میانه‌شان از  $x$  کمتر است)، حداقل دو عضو کوچکتر از میانه خود دسته به همراه میانه انتخاب شده از آن دسته، مقداری کمتر از  $x$  دارند. تعداد این عناصر حداقل  $\lceil n/10 \rceil \times 3$  است که تقریباً  $3n/10$  می‌شود.

به طور مشابه می‌توان ثابت کرد حداقل  $10/3n$  عنصر بزرگتر از  $x$  در آرایه وجود دارد.

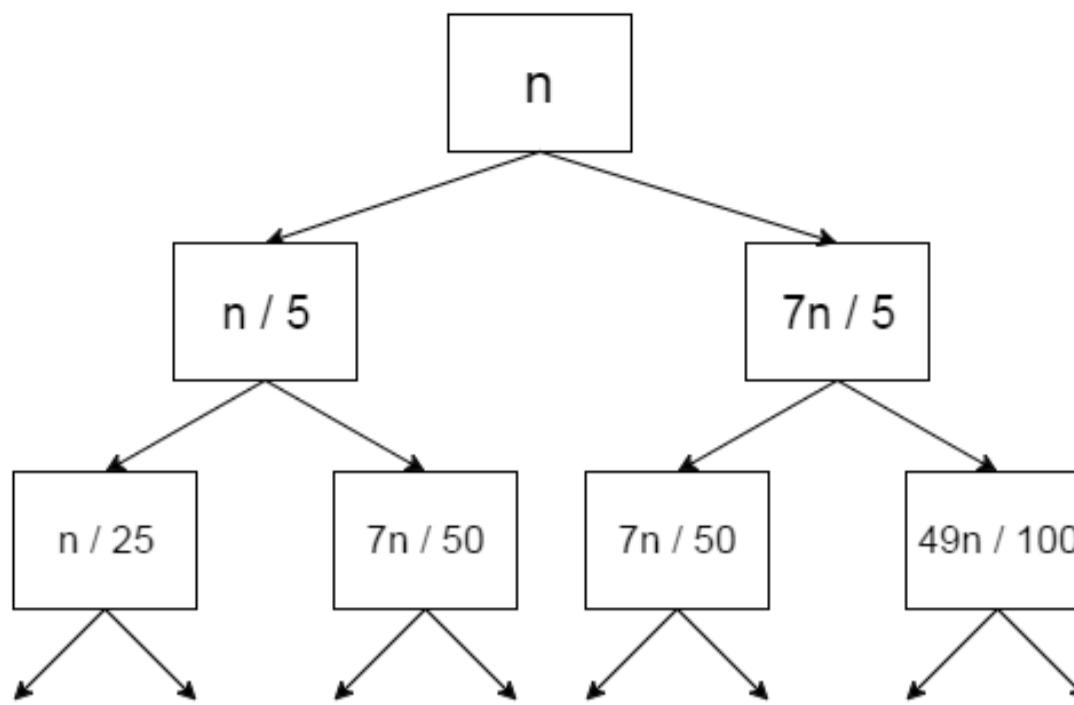


شکل ۲: پیچیدگی زمانی الگوریتم میانه میانه‌ها

با توجه به اینکه دسته اعداد کوچکتر از  $x$  و دسته اعداد بزرگتر از  $x$  حداقل  $10/3n$  عضو دارند، مرتبه زمانی الگوریتم فوق در بدترین حالت از فرمول بازگشتی زیر بدست می‌آید:

$$T(n) \leq T(n/5) + T(7n/10) + c \cdot n$$

از فرمول بالا می‌توان با استنادا دریافت که  $T(n)$  از مرتبه زمانی  $\Theta(n)$  است.



شکل ۳: درخت پیچیدگی زمانی الگوریتم میانه میانه‌ها

## تمرین ۳:

$$T_3(n) = T_3(n/3) + T_3(4n/6) + c \cdot n = \Theta(n \log n)$$

$$T_7(n) = T_7(n/7) + T_7(10n/14) + c \cdot n = \Theta(n)$$

راهنمایی:

در حالت دسته‌های ۳ و ۷ تایی، تابع  $T(n)$  را تعیین کنید و پیچیدگی زمانی آن را تعیین کنید.

در ادامه پیاده‌سازی از این الگوریتم ارائه می‌دهیم.

```
In [2]: def median_of_medians(A, i):
    """
    پیدا می‌کند Median of Medians با استفاده از الگوریتم A این کوچکترین عنصر را در لیست-i این تابع
    است (0(n)) این الگوریتم در بدترین حالت نیز خطی
    """

    Args:
        A (list): لیست اعداد.
        i (int): اندیس مرتبه آماری مورد نظر (از 0 شروع می‌شود).

    Returns:
        any: این کوچکترین عنصر در لیست-i.
    """
    # حالت پایه: اگر لیست کوچک باشد، آن را مرتب کرده و عنصر مورد نظر را برمی‌گردانیم
    # برای اندازه‌های کوچک، مرتب‌سازی ساده کارآمد است
    if len(A) <= 5:
        return sorted(A)[i]

    # به زیرلیست‌های 5 تایی A مرحله 1: تقسیم
    sublists = [A[j:j+5] for j in range(0, len(A), 5)]

    # مرحله 2: یافتن میانه هر دسته و جمع‌آوری آنها
    # برای هر زیرلیست، آن را مرتب کرده و میانه آن را استخراج می‌کنیم
    medians = [sorted(sublist)[len(sublist) // 2] for sublist in sublists]

    # مرحله 3: یافتن میانه میانه‌ها به صورت بازگشتی
    # استفاده خواهد شد (pivot) این میانه به عنوان عنصر محوری
    # را محاسبه می‌کنیم اندیس میانه در لیست
    pivot = median_of_medians(medians, len(medians) // 2)

    # مرحله 4: افزایش لیست
    low = [j for j in A if j < pivot] # عناصر کوچکتر از pivot
    high = [j for j in A if j > pivot] # عناصر بزرگتر از pivot

    # محاسبه تعداد عناصر مساوی با
    equal_to_pivot = [j for j in A if j == pivot]

    k = len(low) # اندازه بخش عناصر کوچکتر از pivot.

    # مرحله 5: بازگشت بر روی بخش مناسب
    if i < k:
        # در بخش عناصر کوچکتر قرار داشت، جستجو را در آن بخش ادامه بده اگر
        return median_of_medians(low, i)
    elif i >= k and i < k + len(equal_to_pivot):
        # این عنصر است-i همان pivot، قرار داشت در بخش عناصر مساوی i اگر
        return pivot
    else: # i >= k + len(equal_to_pivot)
        # در بخش عناصر بزرگتر قرار داشت، جستجو را در آن بخش ادامه بده اگر
        # تنظیم کن 'low' و 'equal_to_pivot' جدید را با توجه به اندازه بخش‌های i
        return median_of_medians(high, i - k - len(equal_to_pivot))
```

```

A_example = [1, 2, 3, 4, 5, 1000, 8, 9, 99]
B_example = [1, 2, 3, 4, 5, 6]

print("Median of A_example (k=0):", median_ofmedians(A_example, 0)) # کوچکترین عنصر
print("Median of A_example (k=7):", median_ofmedians(A_example, 7)) # هشتمین کوچکترین عنصر
print("Median of B_example (k=4):", median_ofmedians(B_example, 4)) # پنجمین کوچکترین عنصر

Median of A_example (k=0): 1
Median of A_example (k=7): 99
Median of B_example (k=4): 5

```

برای اطمینان پیدا کردن از عملکرد خطی حتی در بدترین حالت، می‌توان از الگوریتم **میانه میانهها**<sup>\*</sup> برای تعیین عنصر محوری در **انتخاب سریع (Quickselect)** استفاده کرد. این ترکیب، یک الگوریتم انتخاب خطی تضمین شده در بدترین حالت را فراهم می‌کند. البته سربار محاسباتی تعیین عنصر محوری در این روش (Median of Medians) زیاد است و در عمل، پیاده‌سازی خالص آن کمتر مورد استفاده قرار می‌گیرد.

به همین دلیل، می‌توان از **انتخاب درونگرا (Introsort)**<sup>\*\*</sup> استفاده کرد که با ترکیب انتخاب سریع با میانه میانهها، هم در بدترین حالت، عملکردی خطی را نتیجه می‌دهد. این الگوریتم، یک نسخه هیبریدی (ترکیبی) است که مزایای هر دو روش را با هم ترکیب می‌کند.

همچنین با همین دیدگاه می‌توان روش مرتب‌سازی جدیدی به نام **مرتب‌سازی درونگرا (Introsort)**<sup>\*\*</sup> را معرفی کرد که یکی از سریع‌ترین الگوریتم‌های مرتب‌سازی موجود است.

مرتب‌سازی درونگرا یک مرتب‌سازی ترکیبی است که از روش‌های **مرتب‌سازی سریع (Quicksort)**<sup>\*\*</sup>، **مرتب‌سازی هرمی (Heapsort)**<sup>\*\*</sup> و **مرتب‌سازی درجی (Insertion Sort)**<sup>\*\*</sup> استفاده می‌کند.

روش این الگوریتم به این صورت است که:

● در ابتدا با مرتب‌سازی سریع شروع می‌کند.

اگر عمق بازگشتی آن از یک حد مشخصی گذشت (که متناسب با  $\log n$  است)، به سراغ مرتب‌سازی هرمی می‌رود تا از گرفتار شدن در بدترین حالت الگوریتم مرتب‌سازی سریع که پیچیدگی زمانی برابر با  $O(n^2)$  دارد جلوگیری کند. این حد بیشینه عمق را برابر با  $(\log_2 N) \times 2$  (که  $N$  اندازه اولیه آرایه است) تعریف می‌کنیم.

● همچنین هر گاه که تعداد اجزای مرتب‌سازی (زیرآرایه‌ها) به حدی کوچک باشد که استفاده از مرتب‌سازی درجی بهینه تر باشد (مثلًا برای آرایه‌های با طول کمتر از ۱۶ عنصر)، از آن استفاده می‌کند. این حد را برابر با ۱۶ تعریف می‌کنیم.

● اگر هیچ کدام از این دو مورد رخ ندهد (یعنی عمق بازگشتی از حد نگذشته و اندازه زیرآرایه هم بزرگتر از حد درجی باشد)، همان مرتب‌سازی سریع را ادامه خواهیم داد.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل هشتم، بخش دوم: جست و جوی دودویی و کران بالا و پایین

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

- یک مثال واقعی: شرکت در همایش!
- جست و جوی دودویی
- کران بالا و پایین
- حل برخی مسائل پیچیده با استفاده از جست و جوی دودویی

## مقدمه

در این فصل، به بررسی یکی از پرکاربردترین و کارآمدترین الگوریتم‌ها در علوم کامپیوتر، یعنی **\*جستجوی دودویی (Binary Search)** می‌پردازیم. این الگوریتم به ما اجازه می‌دهد تا یک عنصر خاص را در یک دنباله مرتب شده، با سرعت بسیار بالا پیدا کنیم.

همان طور که در جلسات قبل دیدید، حالت‌هایی برای مرتب‌سازی سریع وجود دارد که باعث می‌شود عملکرد آن قابل قبول نباشد. اما با این وجود این الگوریتم «معمولًا» سریع است. برای دقیق کردن این ویژگی باید با دید احتمالاتی تحلیل را انجام دهیم. به عبارتی می‌گوییم الگوریتم باید در حالت میانگین بررسی شود. در زیر توضیحات بیشتری درباره معنای این کار می‌دهیم و سپس تحلیل را انجام می‌دهیم.

## یک مثال واقعی: شرکت در همایش!

تا به حال شده است که در همایشی شرکت کنید و قبل از شروع همایش به میزی برای گرفتن کارت خود مراجعه کنید؟! بعید می‌دانم تا به حال در چنین شرایطی قرار نگرفته باشید!



در این موقع، مخصوصاً اگر به عنوان نفرات اول به محل گرفتن کارت رسیده باشید، معمولاً یک مسئول ثبت‌نام با لبخند ایستاده است و پس از سلام و خوش‌آمدگویی نام شما را می‌پرسد. پس از این که نامتان را گفتید، تازه داستان آغاز می‌شود و مسئول ثبت‌نام شروع می‌کند به گشتن دنبال کارت‌تان.

- اگر مسئولین برگزاری همایش افراد نامنظمی بوده باشند یا اتفاق غیرمتربقه‌ای رخ داده باشد که کارت‌های شرکت‌کنندگان دیر آماده شده باشد، احتمالاً مسئولین همایش وقت نکرده‌اند که کارت‌ها را مرتب کنند و مسئول ثبت‌نام مجبور خواهد بود تک تک کارت‌ها را مشاهده کرده تا بالاخره کارت شما را پیدا کند. این روش، **جستجوی خطی (Linear Search)** نام دارد.
- اما اگر افراد لایقی دست‌اندرکار برگزاری همایش باشند، مسئولین آن قبلاً کارت‌ها را به ترتیب الفبا مرتب کرده‌اند تا شرکت‌کنندگان کمتر معطل شوند! به نظرتان این مرتب بودن کارت‌ها چه کمکی به پیدا کردن کارت شما می‌کند؟ اگر مسئول ثبت‌نام شروع کند به گشتن از ابتدای کارت‌ها و تک تک کارت‌ها را چک کند، به نظر می‌رسد که تغییر چندانی رخ نخواهد داد و عملاً مرتب بودن کارت‌ها بدون استفاده است.

اما اگر مسئول ثبت‌نام کوچکترین بویی از الگوریتم برده باشد، به گشتن تک به تک کارت‌ها بسنده نمی‌کند و سعی می‌کند از تلاش تیم برگزاری در مرتب کردن کارت‌ها بهره بگیرد. شما اگر مسئول ثبت‌نام بودید چه کاری انجام می‌دادید؟

شاید اولین راهی که به ذهن مسئول ثبت‌نام برسد این باشد که کارت‌ها را به چند دسته تقسیم کند و با چک کردن نام شما با اسمی نفر اول و آخر هر دسته، دسته‌ای که کارت شما در آن قرار گرفته را پیدا کند (چطور؟). حال اگر تعداد کارت‌های آن دسته کم باشد شاید تصمیم بگیرد تک تک کارت‌ها را چک کند. اما می‌تواند دوباره همان تقسیم بندی را روی این دسته‌ها انجام دهد و در دسته‌ی کوچکتری دنبال کارت شما بگردد. این ایده، اساس **جستجوی دودویی** است.

## جست و جوی دودویی

مسئله‌ای که در بالا دیدیم، نمونه‌ای از یک مسئله‌ی جستجو در دنباله‌ای از داده‌های مرتب شده بود. این جنس مسائل را به شکل عمومی‌تر می‌توان به این شکل تعریف کرد:

دنباله‌ای مرتب از داده‌ها (داده‌ی عددی یا هر داده‌ی دیگری که ترتیب روی آن معنی دارد مانند رشته‌ها) به ما داده شده است. الگوریتمی طراحی کنید که بگوید داده دلخواه  $x$  در این دنباله وجود دارد یا خیر و در صورت وجود در کدام جایگاه از دنباله قرار دارد.

حال با استفاده از شهودی که از مثال بالا گرفتیم، راه حل این مسئله را مطرح می‌کنیم. همان طور که دیدیم می‌توانیم بدون زحمت خاصی از ابتدای دنباله شروع کنیم و تک تک اعضای دنباله را با  $x$  مقایسه کنیم و در نهایت وجود با عدم وجود آن در دنباله را گزارش کنیم که به این روش **جستجوی خطی (Linear Search)** می‌گویند. اما دیدیم که چند دسته‌کردن کارت‌ها و پیدا کردن دسته‌ای که عدد مورد نظر در آن قرار می‌گیرد می‌تواند به کم کردن تعداد عملیات‌ها و به تبع آن افزایش سرعت جستجو بینجامد.

برای این کار می‌توانیم راحت‌ترین دسته‌بندی، یعنی تقسیم به دو دسته‌ی مساوی را برای پیش‌برد جستجو انتخاب کرد. در این صورت اگر عضو میانه دنباله را  $mid$  بنامیم، کافی است  $mid$  را با  $x$  مقایسه کنیم. در صورتی که از  $x$  از  $mid$  بزرگتر بود، باید در دسته‌ی دوم یعنی از جایگاه  $mid$  تا انتهای دنباله به دنبال  $x$  بگردیم. و در غیر این صورت (اگر  $x$  کوچکتر یا مساوی  $mid$  بود)، باید در دسته‌ی اول یعنی از ابتدای تا جایگاه دنباله، به جستجوی  $x$  بپردازیم.

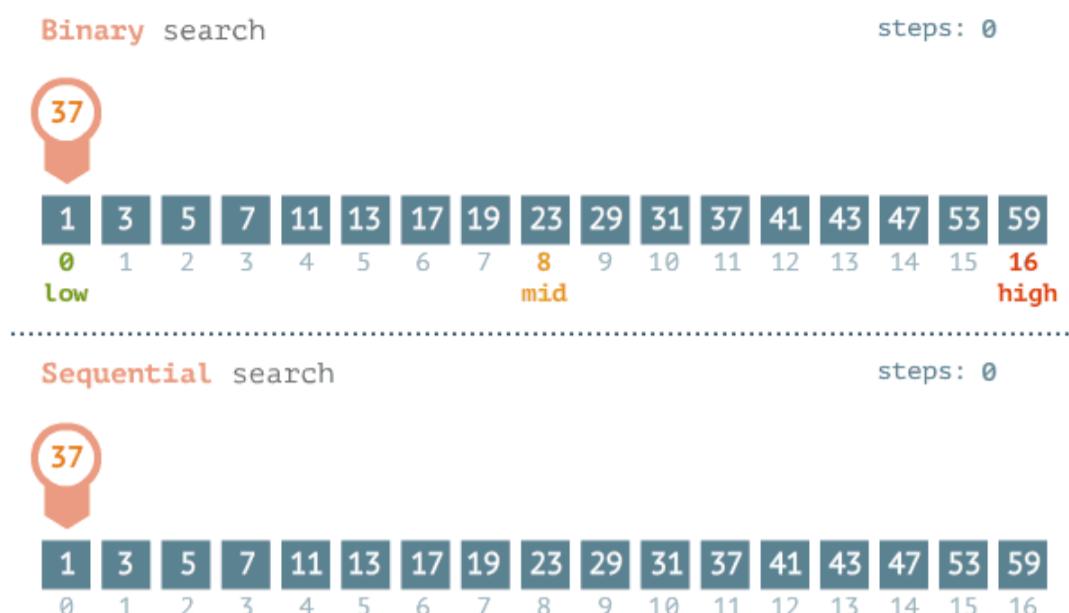
حال طول دنباله‌ای که در آن به دنبال  $x$  می‌گردیم نصف شده است و می‌توانیم به شکل بازگشتی مسئله را روی دنباله‌ی جدید حل کنیم. این روش را با بیانی دقیق‌تر می‌توان به این شکل بیان کرد:

- در هر مرحله فرض می‌کنیم جواب در بازه‌ی  $(r, l]$  باشد (ابتدای بازه‌ی جواب شامل کل آرایه است، مثلاً  $([0, n)$ ).
- در هر مرحله  $mid$  را برابر  $\lfloor(l + r)/2\rfloor$  قرار می‌دهیم.
- اگر  $a[mid]$  بزرگتر از  $x$  بود، جواب را در بازه  $(l, mid]$  جست و جو می‌کنیم.
- در غیر این صورت (اگر  $a[mid]$  کوچکتر یا مساوی  $x$  بود)، جواب را در بازه‌ی  $[mid, r)$  جست و جو می‌کنیم.

عملیات فوق را تا جایی ادامه می‌دهیم که یا  $x$  را پیدا کنیم یا طول بازه به ۱ برسد و دیگر نتوان آن را کوچکتر کرد. با توجه به اینکه طول بازه در ابتدا برابر  $n$  است و در هر مرحله طول آن نصف می‌شود، مرتبه زمانی الگوریتم از  $\theta(\log n)$  است.

$$T(n) = T(n/2) + 1 \Rightarrow T(n) = \theta(\log n)$$

به این روش جستجو در یک دنباله جستجوی دودویی می‌گویند. مقایسه این روش و جستجوی خطی در شکل زیر قابل مشاهده است.



[www.penjee.com](http://www.penjee.com)

در قطعه کد زیر، مسئله‌ی وجود عدد  $x$  در آرایه  $a$  را به روش جستجوی دودویی پیاده‌سازی می‌کنیم.

```
In [1]: def binary_search(a, x):
    """
    با استفاده از جستجوی دودویی بررسی می‌کند a را در آرایه مرتب شده x این تابع وجود عنصر.

    Args:
        a (list): آرایه مرتب شده اعداد.
        x: عنصری که به دنبال آن هستیم.

    Returns:
        bool: در غیر این صورت False، در آرایه یافت شود x اگر True.
    """
    begin, end = 0, len(a) # بازه جستجو [begin, end]

    # تا زمانی که بازه قابل تقسیم باشد (بیش از یک عنصر داشته باشد)
    while end - begin > 1:
        mid = (begin + end) // 2 # محاسبه اندیس میانی

        if a[mid] > x:
            # در نیمه اول بازه است x، بود x اگر عنصر میانی بزرگتر از
            end = mid
        else: # x <= a[mid]
            # در نیمه دوم بازه است x، بود x اگر عنصر میانی کوچکتر یا مساوی
            begin = mid

    # باشد x اندیس تنها عنصری است که ممکن است begin، پس از اتمام حلقه
    return a[begin] == x

# مثال استفاده
a = [1, 3, 5, 7, 9, 11, 13]

for i in range(5):
    print("Is %d in array? %s" % (i, binary_search(a, i)))
```

```
Is 0 in array? False
Is 1 in array? True
Is 2 in array? False
Is 3 in array? True
Is 4 in array? False
```

## کران بالا و پایین

تا اینجا الگوریتم را مشاهده کردیم که در یک آرایه مرتب شده، وجود یا عدم وجود یک عنصر را اطلاع می‌داد؛ حال می‌خواهیم یک مسئله‌ی کلی‌تر را بررسی کنیم:

آرایه‌ی صعودی  $a$  به همراه عدد دلخواه  $x$  داده شده است؛ آخرین (بزرگ‌ترین) اندیسی را پیدا کنید (مثل  $i$ ) که  $a[i] < x$ . به این مسئله  $lower bound$  نیز می‌گویند.

روش حل این مسئله مشابه جستجوی دودویی است که در قسمت قبل دیدیم؛ درواقع ایده به این صورت است که عنصر وسطی آرایه را نگاه می‌کنیم. اگر از  $x$  بزرگ‌تر باشد، پس تمام عناصر نیمه‌ی دوم آرایه هم از  $x$  بزرگ‌تر هستند و جواب (در صورت وجود) در نیمه‌ی اول آرایه است. به طریق مشابه، اگر عنصر وسطی از  $x$  کوچک‌تر باشد، باید جواب را در نیمه‌ی دوم جستجو کرد.

- شرط پایان الگوریتم چیست؟
- اگر عنصر وسطی برابر با  $x$  باشد باید به کدام نیمه برویم؟
- اگر اندیس عنصر وسطی را  $mid$  بنامیم، در صورتی که  $a[mid] \neq x$

پیشنهاد می‌شود کمی به این موارد فکر کنید زیرا بی‌دقیقی در این موارد عموماً باعث ایجاد الگوریتم‌های پایان‌ناپذیر، اشتباه و یا ناخوانانه می‌شود.

## روش حل عمومی

سعی می‌کنیم یک روش حل عمومی برای این جوهر سوال‌ها به دست آوریم. الگوریتم اصلی مبتنی بر **\*اصل ناوردایی (Loop Invariant)** است: با **\*فرض وجود جواب\***، یک بازه تعریف می‌کنیم که جواب در هر مرحله در آن قرار دارد. در اولین مرحله، بازه در کلی ترین حالت خود قرار دارد؛ یعنی در ابتدا بازه‌ی جواب را برابر با  $[0, n]$  می‌گیریم، زیرا فرض کردیم جواب وجود دارد پس قطعاً در این بازه قرار دارد.

**نکته:** همان‌طور که می‌بینید، بازه‌ی جواب را به صورت **\*بسنته-باز\*** تعریف کردیم. این کار به نسبت انتخاب بازه به صورت بسته-بسنته کار را بسیار راحت‌تر می‌کند که البته در ادامه متوجه آن خواهید شد.

حل این سوال با استفاده از اصل ناوردایی ریاضی این‌گونه است: در هر مرحله دو کار انجام می‌دهیم:

1. بازه‌ی جواب را کوچک‌تر می‌کنیم.
2. ثابت می‌کنیم شرط ناوردایی کماکان برقرار است، یعنی جواب همچنان در داخل بازه‌ی کوچک‌شده قرار دارد.

اگر الگوریتممان این دو شرط را برآورده کند؛ آنگاه حتماً درست است و نیازی به بررسی سایر موارد نیست: بازه‌ی جواب آنقدر کوچک می‌شود که در نهایت تک‌عضوی می‌شود و چون شرط ناوردایی در همه‌ی مراحل برقرار بوده، آن تک عنصر باقی‌مانده قطعاً جواب مساله است. البته حالت که مساله کلا جواب ندارد را باید جداگانه بررسی کنیم؛ هرچند در عموم موارد همان الگوریتم جواب غیرمعمولی در این حالت نمی‌دهد.

## بازگشت به مساله‌ی اصلی

با توجه به روش کلی، حل مساله ساده است؛ فرض می‌کنیم در یک مرحله دلخواه قرار داریم و بازه‌ی جواب به صورت  $[start, end]$  است. درمورد این بازه می‌دانیم که اولاً ناتهی است، زیرا طبق فرض اولیه مساله جواب دارد و طبق شرط ناوردایی، در تمام مراحل تاکنون جواب در داخل بازه باقی‌مانده است پس این بازه نمی‌تواند تهی باشد. ثانیاً، این بازه علاوه بر ناتهی بودن، طولش از یک بیشتر است. اگر طول بازه یک باشد طبق قسمت قبل آن عضو باقی‌مانده قطعاً جواب مساله است و الگوریتم باید پایان یابد.

پس فرض می‌کنیم  $end > start + 1$  است. عنصر میانی آرایه را به این شکل در نظر می‌گیریم:  $mid = \lfloor \frac{start+end}{2} \rfloor$ . با توجه به فرض  $mid \in [start, end]$  می‌توان ثابت کرد  $mid < start + 1$ . پس این اندیس قطعاً در داخل بازه‌ی جواب است.

حالا عنصر  $a[mid]$  را با  $x$  مقایسه می‌کنیم. دو حالت پیش می‌آید:

1.  $a[mid] \geq x$ : در این حالت همه اعضای بازه‌ی  $[mid, end]$  آرایه از  $x$  بزرگ‌تر یا مساوی هستند پس جواب قطعاً در این بازه نیست و در بازه‌ی  $[start, mid]$  است. در این صورت،  $mid$  را برابر  $start$  قرار می‌دهیم.

2.  $a[mid] < x$ : چون به دنبال آخرین عنصری می‌گردیم که از  $x$  کوچک‌تر است، عناصر بازه‌ی  $[start, mid]$  نمی‌توانند جواب باشند و جواب یا خود  $mid$  است یا اندیسی بزرگ‌تر از آن پس جواب در بازه‌ی  $[mid, end]$  موجود است. در این صورت،  $mid$  را برابر  $end$  قرار می‌دهیم.

بنابراین شرط ناوردایی برقرار است و جواب همواره در بازه‌ی کاهش‌یافته قرار دارد؛ دقت کنید که حالت تساوی ( $x == a[mid]$ ) یک حالت خاص نبود و شرایطی که ایجاد می‌کرد دقیقاً مشابه حالت بزرگتر است (یعنی  $a[mid] > x$  را شامل می‌شود)؛ همچنین اهمیت گرفتن بازه‌ی بسته-باز با این الگوریتم مشخص شد: اگر در مرحله‌ی اول بازه‌ی بسته-باز بگیریم، در تمام مراحل بعدی نیز بازه‌ی جواب به صورت بسته-باز باقی می‌ماند ولی اگر بازه را بسته-بسته می‌گرفتیم کار سخت‌تر می‌شد. (می‌توانید امتحان کنید!)

در گام دوم باید ثابت کنیم که بازه‌ی ما واقعاً «کاهش می‌یابد»؛ یعنی طولش کم می‌شود. باید هر دو حالت کاهش را درنظر بگیریم و با توجه به قرارداشتن  $mid$  در بازه‌ی  $[start, end]$  کاهش را اثبات کنیم:

1. اگر  $end = mid$  شود: طول جدید  $< start$  است.

2. اگر  $start = mid$  شود: طول جدید  $< end - start$  است.

پس در هر دو حالت طول بازه‌ی جدید از بازه‌ی قدیمی کمتر است و کاهش به درستی انجام شده است. بنابراین هر دو شرط ناوردایی در این الگوریتم به درستی برآورده شده‌اند و الگوریتم درست کار می‌کند.

با توجه به اینکه در هر مرحله طول بازه‌ی جواب نصف می‌شود، هزینه زمانی الگوریتم از  $O(\log n)$  است.

کد الگوریتم در ادامه آمده است:

```
In [2]: def lower_bound(a, x, start, end):
    """
    باشد  $x < a[i]$  پیدا می‌کند که  $a$  را در آرایه صعودی نه این تابع آخرین (بزرگترین) اندیس پیدا می‌کند.
    Args:
        a (list): آرایه صعودی مرتب شده.
        x: عددی که می‌خواهیم کران پایین آن را پیدا کنیم.
        start (int): اندیس شروع بازه جستجو (بسته).
        end (int): اندیس پایان بازه جستجو (باز).
    Returns:
        int: اندیس مورد نظر # همان جواب است start، حالت پایه: اگر طول بازه به 1 رسید.
    """
    if start == end - 1:
        return start

    محاسبه اندیس میانی (با استفاده از شیفت بیتی برای تقسیم بر 2)
    mid = (start + end) // 2

    if a[mid] >= x:
        بود، جواب در نیمه اول بازه است x اگر عنصر میانی بزرگتر یا مساوی # return lower_bound(a, x, start, mid)
    else: # a[mid] < x
        بود، جواب با خودش است یا در نیمه دوم بازه x اگر عنصر میانی کوچکتر از # return lower_bound(a, x, mid, end)

    # مثال استفاده
    a = [2, 3, 3, 5, 8, 8, 8, 10]
    n = len(a)

    index_for_5 = lower_bound(a, 5, 0, n)
    print(f"index_for_5 = {index_for_5} باشد # {a[index_for_5]} است اندیس x=5 برای".format(index_for_5))

    index_for_8 = lower_bound(a, 8, 0, n)
    print(f"index_for_8 = {index_for_8} باشد # {a[index_for_8]} است اندیس x=8 برای".format(index_for_8))

    index_for_1000 = lower_bound(a, 1000, 0, n)
    print(f"index_for_1000 = {index_for_1000} باشد # {a[index_for_1000]} است اندیس x=1000 برای".format(index_for_1000))

    index_for_0 = lower_bound(a, 0, 0, n)
    print(f"index_for_0 = {index_for_0} باشد # {a[index_for_0]} است اندیس x=0 برای".format(index_for_0))

    اندیس 2 و مقدار 3 است x=5، اندیس 3 و مقدار 5 است x=8، اندیس 7 و مقدار 10 است x=1000، اندیس 0 و مقدار 2 است x=0.
```

دقت کنید که خروجی تابع `lower_bound` یک اندیس است نه مقدار آن عضو از آرایه. در مثال اول (برای  $5 = x$ ) با توجه به وجود داشتن دو عدد ۳ در آرایه، عددی که اندیس بزرگتری دارد (اندیس ۳) جواب مساله است.

در سایر مثال‌ها صرفاً اندیس را چاپ کردیم؛ نکته مهم در مثال آخر (برای  $0 = x$ ) نهفته است: اگر جواب موجود نباشد (یعنی همه‌ی اعداد آرایه از  $x$  بزرگتر یا مساوی هستند)، پس در تمامی مراحل کاهش به نیمه‌ی اول بازه می‌رویم و در نهایت اندیس صفر را به عنوان جواب خروجی می‌دهد. این اندیس صفر، به اولین عنصری در آرایه اشاره می‌کند که از  $x$  بزرگتر یا مساوی است.

پس باید این حالت را جداگانه بررسی کنیم و یک جواب قراردادی ارائه دهیم (مثال ۱). در زیر همان کد بالا به صورت غیربازگشتی و با رعایت مورد آخر (برگرداندن ۱ در صورت عدم وجود جواب) زده شده است.

```
In [3]: def lower_bound_iterative(a, x): # تغییر نام برای تمایز از نسخه بازگشتی
    """
    بashed x < a[i] پیدا می‌کند که a را در آرایه صعودی نه این تابع آخرین (بزرگترین) اندیس پیاده‌سازی غیربازگشتی.
    ۱. برمی‌گرداند -، هستند x = > یعنی همه عناصر در صورت عدم وجود چنین عنصری.

    Args:
        a (list): آرایه صعودی مرتب شده.
        x: عددی که می‌خواهیم کران بالای آن را پیدا کنیم.

    Returns:
        tuple or int: پک تاپل (اندیس، مقدار) اگر جواب یافت شود، یا -۱ اگر یافت نشود.
    """
    نیست x بزرگتر با مساوی باشد، یعنی هیچ عنصری کوچکتر از x حالت خاص: اگر اولین عنصر آرایه از
    # if not a or a[0] >= x:
    return -1

    start = 0 # اندیس شروع بازه (بسته)
    end = len(a) # اندیس پایان بازه (باز)

    حلقه اصلی جستجو: تا زمانی که بازه قابل تقسیم باشد
    while end > start + 1:
        mid = (start + end) // 1 # محاسبه اندیس میانی

        if a[mid] >= x:
            بود، جواب در نیمه اول بازه است x اگر عنصر میانی بزرگتر یا مساوی
            end = mid
        else: # a[mid] < x
            بود، جواب یا خودش است یا در نیمه دوم بازه x اگر عنصر میانی کوچکتر از
            start = mid

    را برآورده می‌کند x < a[i] اندیس آخرین عنصری است که شرط start، پس از اتمام حلقه
    return (start, a[start])

    مثال استفاده
    print("--- مثال استفاده از lower_bound_iterative ---")
    باید (۱, ۳) باشد
    print(lower_bound_iterative([1, 3, 3, 3, 4], 3)) # باید -۱ باشد (هیچ عنصری کوچکتر از ۸ نیست)
    print(lower_bound_iterative([9, 10, 11], 8)) # (تقسیم بر ۲)

    --- مثال استفاده از lower_bound_iterative ---
    (0, 1)
    -1
    --- مثال عملیات بیتی
    4
```

تابع بالا در صورت وجود جواب یک زوج مرتب از اندیس و عنصر متناظر آن در آرایه خروجی می‌دهد. در حالت کلی اگر یک الگوریتم بازگشتی را بتوانید به صورت غیربازگشتی پیاده‌سازی کنید بهتر است زیرا فراخوانی تابع هم زمان بر است و هم حافظه‌بر؛ پس احتمال رد کردن محدودیت زمان و حافظه را افزایش می‌دهد.

## خودآزمایی ۱:

آرایه صعودی  $a$  به همراه عدد  $x$  داده شده است. کوچکترین اندیسی را پیدا کنید (مثل  $i$ ) که  $a[i] > x$  داشته باشد.

راهنمایی: این بار بازه‌ی جواب را به صورت باز-بسته بگیرید، یعنی در ابتدا بازه را برابر با  $(-1, n)$  در نظر بگیرید (که  $n$  طول آرایه است) و در هر مرحله سعی کنید بازه را به همین صورت نگه دارید.

همچنانیکنین اگر جواب موجود نیست (یعنی همه عناصر  $x \leq$  هستند) باید ۱ - خروجی دهید و در غیر این صورت مانند مثال قبل، زوج مرتبی از اندیس و مقدار عنصر متناظر آرایه را خروجی دهید.

```
In [4]: from src.tests.tester import tester # وارد کردن تابع
brای خودآزمایی tester

def upper_bound(a, x):
    """
    بashed x > a[i] پیدا می‌کند که a را در آرایه صعودی نه این تابع کوچکترین اندیس
    ایجاد کنید.

    Args:
        a (list): آرایه صعودی مرتب شده.
        x: عددی که می‌خواهیم کران بالای آن را پیدا کنیم.

    Returns:
        tuple or int: پک تاپل (اندیس، مقدار) اگر جواب یافت شود، یا -۱ اگر یافت نشود.
    """
    پیاده‌سازی الگوریتم خود را اینجا بنویسید.
    استفاده کنید (start, end].
    # start = -1, end = len(a) - 1
    # در هر مرحله: mid = (start + end) // 2
    # اگر a[mid] > x: جواب در نیمه چپ است (end = mid)
    # else: a[mid] <= x: جواب در نیمه راست است (start = mid)
    # شرط پایان: end - start == 1
    # معتبر است و end اندیس مورد نظر است. باید چک کنید که آیا end، در نهایت #
```

```

#مثال پیاده‌سازی:
start = -1
end = len(a)

while end - start > 1:
    mid = (start + end) // 2
    if a[mid] > x:
        end = mid
    else: # a[mid] <= x
        start = mid

if end == len(a): # به انتهای آرایه رسید، یعنی همه عناصر اگر end
    return -1
return (end, a[end])

tester("upper_bound", upper_bound)

```

Your code passes all 3 test(s).

## خودآزمایی ۲:

آرایه  $a$  به همراه عدد  $x$  داده شده است. بزرگترین اندیسی را پیدا کنید (مثل  $i$ ) که  $a[i] > x$

فرمت خروجی مثل تمرین قبل است.

```

In [5]: def magic_function(a, x):
    """
    پیدا می‌کند که a را در آرایه نزولی این تابع بزرگترین اندیس باشد x > a[i].
    Args:
        a (list): آرایه نزولی مرتب شده.
        x: عددی که می‌خواهیم مقایسه کنیم.
    Returns:
        tuple or int: یک تاپل (اندیس، مقدار) اگر جواب یافت شود، یا -1 اگر یافت نشود.
    """
    # پیاده‌سازی الگوریتم خود را اینجا بنویسید.
    # راهنمایی: بازه و منطق جستجو را برای آرایه نزولی تنظیم کنید
    # start = -1, end = len(a)
    # در هر مرحله: mid = (start + end) // 2
    # جواب در نیمه چپ یا خودش است (start = mid)
    # else (a[mid] <= x): جواب در نیمه راست است (end = mid)
    # شرط پایان: end - start == 1
    # معتبر است و start اندیس مورد نظر است. باید چک کنید که آیا start > x، در نهایت

    start = -1
    end = len(a)

    while end - start > 1:
        mid = (start + end) // 2
        if a[mid] > x:
            start = mid
        else: # a[mid] <= x
            end = mid

    if start == -1: # اگر start رسید، یعنی هیچ عنصری بزرگتر از start نیست
        return -1
    return (start, a[start])

tester("magic_function", magic_function)

```

Your code passes all 2 test(s).

## حل برخی مسائل پیچیده با استفاده از جست و جوی دودویی

در برخی مسائل، استفاده از جست و جوی دودویی و کرانهای بالا و پایین به طور روشن در صورت مسئله نیامده است؛ این مثال را درنظر بگیرید:

یک آرایه  $n$  عضوی به همراه عدد  $m$  داده شده است؛ می‌خواهیم از این آرایه  $m$  عدد را انتخاب کنیم بهطوری که کمترین اختلاف بین اعداد انتخاب شده در بین همه حالات انتخاب  $m$  عدد، بیشینه باشد.

دقت کنید که اختلاف دو عدد همواره مثبت است.

به عنوان مثال فرض کنید آرایه شامل اعداد 4, 9, 8, 8, 1 باشد و بخواهیم سه عدد را انتخاب کنیم. چهار حالت برای انتخاب این سه عدد وجود دارد:

1. اعداد 8, 9, 1 انتخاب شوند که کمترین اختلاف برابر است با  $9 - 8 = 1$
2. اعداد 4, 9, 1 انتخاب شوند که کمترین اختلاف برابر است با  $4 - 1 = 3$
3. اعداد 4, 8, 1 انتخاب شوند که کمترین اختلاف برابر است با  $4 - 1 = 3$

## ۴. اعداد ۹، ۸، ۴ انتخاب شوند که کمترین اختلاف برابر است با $1 = 9 - 8$

بیشینه‌ی این کمترین اختلاف‌ها برابر با ۳ است.

این سوال ظاهر پیچیده و مشکل دارد ولی حل آن به یک نکته‌ی اساسی بستگی دارد: تغییر در زاویه‌ی نگاه به مسئله؛ مسئله را به یک مساله‌ی تصمیم تبدیل می‌کنیم که ساده‌تر از مساله‌ی اصلی است:

یک آرایه به همراه اعداد  $m$  و  $x$  داده شده است؛ آیا می‌توان  $m$  عدد را از آن طوری انتخاب کرد که کمینه‌ی اختلاف بین این  $m$  عدد \*حداقل\*  $x$  باشد؟ یا به بیان دیگر اختلاف هر دو عدد انتخاب شده حداقل  $x$  باشد؟

حل این مساله ساده‌است؛ اگر جواب مساله «بله» باشد، حداقل یک حالت وجود دارد که عدد مینیمم آرایه جزو  $m$  عدد انتخاب شده است (چرا؟).

\*\*پاسخ:\*\* اگر جواب «بله» باشد، یعنی می‌توان  $m$  عدد را با حداقل اختلاف  $x$  انتخاب کرد. اگر کوچکترین عدد آرایه را انتخاب نکنیم، آنگاه  $m$  عدد را از بقیه انتخاب کرده‌ایم. در این صورت، کمترین اختلاف بین این  $m$  عدد همچنان حداقل  $x$  است. اما اگر کوچکترین عدد آرایه را انتخاب کنیم، این کار به ما کمک می‌کند تا بیشترین تعداد ممکن از اعداد را انتخاب کنیم. در واقع، اگر کوچکترین عدد را انتخاب نکنیم، ممکن است نتوانیم به  $m$  عدد برسیم.

اگر اختلاف دومین عدد آرایه (در ترتیب سورت شده) با عدد مینیمم از  $x$  کمتر باشد، نمی‌توان آن را همراه با مینیمم انتخاب کرد و گرنه آن را همراه با مینیمم طریق پیش می‌رویم. اصولاً کاندیداهای مجاز برای کمترین اختلاف، اعدادی هستند که در ترتیب مرتب شده آرایه پشت سر هم هستند (و بیان عددي انتخاب نشده است). پس از ابتدا می‌توان فرض کرد آرایه‌ی ورودی مرتب شده است و روی این ترتیب حرکت کرد. کد این الگوریتم به این شکل می‌شود:

In [6]:

```
def check(a, m, x):
    """
    انتخاب کرد a عدد را از آرایه مرتب شده m این تابع بررسی می‌کند که آیا می‌توان باشد x به طوری که کمینه اختلاف بین آنها حداقل
    """

Args:
    a (list): آرایه مرتب شده اعداد.
    m (int): تعداد اعدادی که باید انتخاب شوند.
    x (int): حداقل اختلاف مورد نظر بین اعداد انتخاب شده.

Returns:
    list or None: عنصر انتخاب شده اگر شرط برقرار باشد، در غیر این صورت m لیستی از None.
    """
    همیشه اولین (کوچکترین) عنصر را انتخاب می‌کنیم #.
    عنصر با حداقل اختلاف m پیمایش بقیه عناصر برای یافتن #.
    for i in range(1, len(a)):
        بود آن را انتخاب کن x اگر اختلاف عنصر فعلی با آخرین عنصر انتخاب شده حداقل #.
        if a[i] - ans[-1] >= x:
            ans.append(a[i])

    عنصر اول را برگردان m، بود m اگر تعداد عناصر انتخاب شده حداقل #.
    if len(ans) >= m:
        return ans[0:m] # بازگرداند
    return None # در غیر این صورت
```

هزینه زمانی اجرای تابع **check** از  $\Theta(n)$  است، زیرا فقط یک بار آرایه را پیمایش می‌کند.

حالا به این نکته‌ی مهم دقت می‌کنیم که اگر این الگوریتم برای عدد  $x$  خروجی «بله» بدهد، خروجی‌اش برای  $1 - x$  و برای تمامی اعداد کمتر از  $x$  هم «بله» خواهد بود (زیرا اگر می‌توان  $m$  عدد با حداقل اختلاف  $x$  انتخاب کرد، پس با حداقل اختلاف  $1 - x$  هم می‌توان). و اگر برای عدد  $y$  خروجی «خیر» بدهد، برای تمامی اعداد بزرگتر از  $y$  هم خروجی «خیر» می‌دهد (زیرا اگر نتوان  $m$  عدد با حداقل اختلاف  $y$  انتخاب کرد، پس با حداقل اختلاف بزرگتر از  $y$  هم نمی‌توان).

بنابراین اگر همه‌ی اعداد طبیعی را در نظر بگیریم، از عدد یک تا عددی مثل  $z$  خروجی الگوریتم «بله» است و از آن عدد به بعد «خیر»؛ در این حالت، کمینه‌ی اختلاف بین آن  $m$  عدد انتخاب شده دقیقاً  $z$  است و نمی‌تواند از  $z$  بیشتر باشد (چرا؟ انصافاً ساده است!).

\*\*پاسخ به "چرا؟":\*\* اگر کمینه‌ی اختلاف می‌توانست بیشتر از  $z$  باشد، مثلًاً  $z$  آنگاه تابع `check` برای  $z$  نیز باید «بله» برمی‌گرداند، که این با تعریف  $z$  (آخرین  $x$  که `check` برای آن «بله» می‌دهد) در تناقض است.

پس  $z$  جواب مساله‌ی اصلی است زیرا در هیچ حالتی کمینه‌ی اختلاف بین  $m$  عدد انتخاب شده از  $z$  بیشتر نمی‌شود. پس هدف سوال که یافتن حالتی است که کمینه‌ی اختلاف بین اعداد بیشینه شود معادل یافتن  $z$  است.

حال چگونه  $z$  را بیابیم؟ مساله‌ی کران پایین را بهیاد بیاورید؛ می‌توان آن مساله را اینگونه بیان کرد که برای یک آرایه‌ی مرتب شده و یک عدد  $x$  داده شده، اعضای آرایه که از  $x$  کوچکترند را به ۱ و سایر اعضای را به صفر تبدیل می‌کنیم و هدف، یافتن اندیس آخرین ۱ است.

پس مساله‌ی یافتن  $\geq$  راه حلی مشابه مساله‌ی کران پایین دارد و در واقع کاربری عملی از آن مساله است. مشابه حل آن مساله، یک بازه‌ی جواب تعریف می‌کنیم که ابتدا  $[0, D + 1]$  است.  $D$  را می‌توان برابر اختلاف بین ماسیموم و مینیمم آرایه درنظر گرفت زیرا جواب هرگز از  $D$  بیشتر نمی‌شود و ابتدا در داخل این بازه قرار دارد. حالا در هر مرحله عدد  $mid$  را به تابع تصمیم می‌دهیم؛ اگر خروجی «بله» بود مشابه مساله‌ی کران پایین در نیمه‌ی بالا و در غیر این صورت در نیمه‌ی پایین دنبال جواب می‌گردیم.

In [7]: `def solve_max_min_diff(a, m):`

```
    """
    انتخاب می‌کند به طوری که کمترین اختلاف بین آن‌ها بیشینه باشد a عدد را از آرایه m این تابع از جستجوی دودویی بر روی فضای جواب استفاده می‌کند.

    Args:
        a (list): آرایه‌ی از اعداد.
        m (int): تعداد اعدادی که باید انتخاب شوند.

    Returns:
        tuple: یک زوج مرتب (حداکثر کمینه اختلاف، لیست عناصر انتخاب شده).
    """
    a.sort() # ابتداء آرایه را مرتب می‌کنیم # O(n log n)

    # تعریف بازه جستجو برای جواب (کمینه اختلاف)
    # start: حداقل اختلاف ممکن (0).
    # end: (حداکثر اختلاف ممکن) (اختلاف بزرگترین و کوچکترین عنصر + 1)
    start = 0
    end = a[-1] - a[0] + 1

    # لیستی برای نگهداری عناصر انتخاب شده در بهترین حالت یافت شده
    ans_elements = [] # برمی‌گرداند برای آن check که تابع x جستجوی دودویی برای یافتن بزرگترین

    while end > start + 1:
        mid = (start + end) // 2 # محاسبه میانه بازه

        # فراخوانی تابع تصمیم (check).
        out = check(a, m, mid) # عنصر را انتخاب کرد m می‌توان 'mid' اگر با این
        # بزرگتری پیدا کنی 'mid' یک کاندیدای جواب است. سعی کن 'mid' این
        # عناصر انتخاب شده را ذخیره کن # ans_elements = out
        # بازه را به نیمه بالا منتقل کن start = mid
        else:
            # عنصر را انتخاب کرد m نمی‌توان 'mid' اگر با این
            # خیلی بزرگ است. بازه را به نیمه پایین منتقل کن # mid = end
            # بازه را به نیمه پایین منتقل کن end = mid

    return (start, ans_elements) # start همان زمانی است که انتخاب شده
```

همان‌طور که مشاهده می‌شود، خروجی تابع بالا یک زوج مرتب شامل  $\geq$  (حداکثر کمینه اختلاف) و یک روش انتخاب  $m$  عدد است که کمینه اختلافشان  $\geq$  شود.

هزینه‌ی زمانی این الگوریتم از  $\Theta(n \log(\max a - \min a))$  است. این به این دلیل است که:

- ابتداء آرایه مرتب می‌شود که  $O(n \log n)$  زمان می‌برد.
- سپس، یک جستجوی دودویی بر روی بازه ممکن از اختلافها (از 0 تا  $\max a - \min a$ ) انجام می‌شود.
- طول این بازه  $D = \max a - \min a$  است.
- تعداد مراحل جستجوی دودویی  $\log D$  است.
- در هر مرحله از جستجوی دودویی، تابع `check` فراخوانی می‌شود که  $O(n)$  زمان می‌برد.

بنابراین، هزینه کل  $O(n \log n + n \log D) = O(n(\log n + \log D))$  است. اگر  $D$  خیلی بزرگ نباشد، این الگوریتم بسیار کارآمد خواهد بود.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل نهم: توابع درهمسازی(قسمت اول)

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

## فهرست

• مقدمه

• معرفی تابع درهمسازی

• برخی توابع درهمسازی

• برخورد و روش‌های مقابله با آن

## مقدمه

در این فصل، به معرفی یکی از مهم‌ترین و پرکاربردترین مفاهیم در علوم کامپیوتر، یعنی **\*تابع درهمسازی (Hash Functions)** و داده‌ساختارهای مبتنی بر آن‌ها می‌پردازیم. توابع درهمسازی نقش کلیدی در بهینه‌سازی عملیات‌های جستجو، درج و حذف در مجموعه‌های بزرگ داده ایفا می‌کنند.

یک مسئله ساده:

فرض کنید مجموعه‌ای از اعداد در بازه صفر تا صد داریم و می‌خواهیم داده‌ساختاری برای نگهداری آن‌ها طراحی کنیم که امکان انجام عملیات‌های **Find** (یافتن)، **Insert** (درج) و **Delete** (حذف) را در زمان  $O(1)$  (زمان ثابت) داشته باشد.

راه حل:

از یک آرایه به طول صد استفاده می‌کنیم. اگر خانه  $i$  از آرایه مقدار صفر داشته باشد، یعنی عدد  $i$  در مجموعه وجود ندارد و اگر مقدار آن یک باشد، یعنی عدد  $i$  وجود دارد. به طور مشابه، حذف اعداد با صفر کردن خانه منتظر و اضافه کردن اعداد با یک کردن آن انجام می‌شود. این روش برای این مسئله ساده، بسیار کارآمد است.

یک مسئله سخت‌تر:

فرض کنید این‌بار اعداد در بازه صفر تا  $10^{12}$  هستند، اما می‌دانیم حداقل  $10^3$  عدد در مجموعه وجود خواهد داشت. آیا راه حل قبلی قابل استفاده است؟ **\*پاسخ:** خیر. اگر بخواهیم از یک آرایه به طول  $10^{12}$  استفاده کنیم، به حافظه بسیار زیادی نیاز خواهیم داشت (حتی اگر فقط  $10^3$  عنصر را ذخیره کنیم). این روش از نظر حافظه ناکارآمد است و بهینه‌سازی آن ضروری است. در اینجا، تابع درهمسازی به کمک ما می‌آیند.

## معرفی تابع درهمسازی

تابع درهمسازی (**Hash Function**)، تابعی مانند  $D : h : D \rightarrow L$  است که اعضای مجموعه  $D$  (که اندازه دلخواه دارند) را به مجموعه‌ای با اندازه ثابت  $L$  نگاشت می‌دهد.

معمولًا این تابع به گونه‌ای هستند که اندازه  $D$  (دامنه ورودی) بسیار بزرگ‌تر از اندازه  $L$  (دامنه خروجی یا اندازه جدول هش) است که مزایا و معایب خود را دارد.

برای مثال، می‌توانیم رشته‌های ساخته شده با حروف انگلیسی را با یک تابع درهمسازی به مجموعه اعداد یک بازه خاص بنگاریم. مثلاً، به هر رشته، تعداد حروف آن رشته را نسبت می‌دهیم. البته در ادامه خواهیم دید که تابع درهمسازی‌ای مانند مثال بالا کاربرد چندانی ندارد و یک تابع درهمسازی خوب باید ویژگی‌های خاصی را دارا باشد تا برخوردهای (collisions) کمتری ایجاد کند.

تابع درهمساز دامنه کاربردهای گسترده‌ای در مهندسی کامپیوتر دارد، از جمله:

- داده‌ساختارهای چون **جدول درهمسازی (Hash Table)**
- سیستم‌های رمزگاری و امنیت داده (مانند هش کردن رمز عبور)
- مبادلات اینترنتی (مانند بررسی یکپارچگی داده‌ها)
- سیستم‌های تعیین هویت دیجیتال مانند تشخیص اثر انگشت
- بررسی صحت فایل‌ها (Checksums)

حل مسائلی مشابه مسئله قبلی (مسئله سخت‌تر) هم از کاربردهای پایه‌ای درهمسازی است. راه حل این سوال را در ادامه می‌آوریم:

راه حل:

فرض کنید در مسئله قسمت قبل، تابعی داشته باشیم که اعداد بین  $0$  و  $10^{12}$  را به اعداد بین  $0$  تا  $10^4$  نگاشت کند. این تابع درهمسازی  $h$  نام دارد. با این تضمین که احتمال نگاشت شدن دو عدد متفاوت به یک عدد در این بازه قابل صرف نظر باشد (همچنین فرض کنید هزینه نگاشت کردن کم و قابل صرف نظر باشد که معمولاً همینطور است).

حال با کمک این تابع، می‌توانیم ایده مسئله اول را به مسئله بزرگتر تعمیم دهیم به گونه‌ای که برای درج، یافتن و حذف یک عنصر مانند  $x$  کافیست به خانه  $(x)$  از آرایه  $10^4$  عضوی رجوع کنیم.

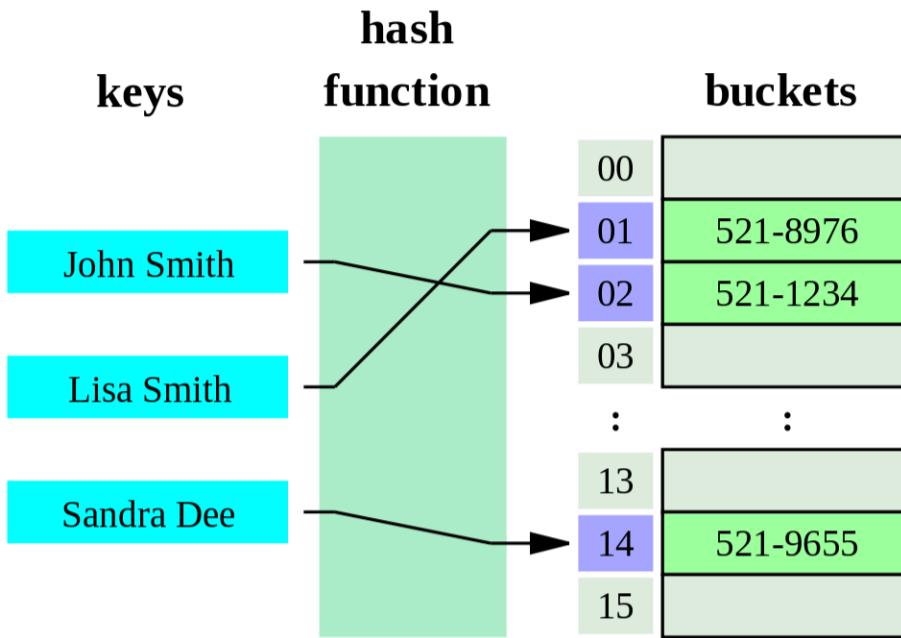
قطعه کد زیر پیاده‌سازی این مسئله را آورده است:

```
In [1]:  
# تعریف اندازه جدول هش (اندازه دامنه خروجی تابع درهمسازی)  
hash_size = 10**4 # 10,000  
  
# تعریف تابع درهمسازی ساده (روش باقیمانده تقسیم)  
def hash_function(x):  
    نگاشت می‌کند [0, hash_size-1] این تابع یک عدد را به یک اندیس در بازه #  
    return x % hash_size  
  
class my_set:  
    با استفاده از جدول هش ساده (set) پیاده‌سازی یک مجموعه #  
    # این روش فرض می‌کند که برخوردها نادر هستند یا به روش خاصی مدیریت می‌شوند #  
    # آرایه‌ای که به عنوان جدول هش عمل می‌کند.  
    # مقدار 0 به معنی خالی بودن خانه و 1 به معنی وجود عنصر است.  
    table = [0] * hash_size  
  
    def insert(self, x):  
        یک عنصر را به مجموعه اضافه می‌کند #  
        خانه متناظر با هش عنصر را به 1 تغییر می‌دهد #  
        self.table[hash_function(x)] = 1  
  
    def exists(self, x):  
        بررسی می‌کند که آیا یک عنصر در مجموعه وجود دارد یا خیر #  
        مقدار خانه متناظر با هش عنصر را برمی‌گرداند (0 یا 1) #  
        return self.table[hash_function(x)] == 1  
  
    def remove(self, x):  
        یک عنصر را از مجموعه حذف می‌کند #  
        خانه متناظر با هش عنصر را به 0 تغییر می‌دهد #  
        self.table[hash_function(x)] = 0  
  
# مثال استفاده از کلاس my_set  
s = my_set() # ایجاد یک شیء از کلاس my_set  
s.insert(10) # درج عدد 10  
s.insert(245) # درج عدد 245  
باشد True باید # آیا 245 در مجموعه وجود دارد؟  
print(s.exists(245)) # آیا 245 در مجموعه وجود دارد؟  
s.insert(8653) # درج عدد 8653  
s.insert(12543252) # درج عدد 12543252  
باشد True باید # آیا 10 در مجموعه وجود دارد؟  
s.remove(10) # حذف عدد 10  
باشد False باید # آیا 10 در مجموعه وجود دارد؟  
print(s.exists(10)) # آیا 10 در مجموعه وجود دارد؟  
باشد True باید # آیا 12543252 در مجموعه وجود دارد؟  
print(s.exists(12543252)) # آیا 12543252 در مجموعه وجود دارد؟  
باشد False باید # آیا 12543251 در مجموعه وجود دارد؟  
print(s.exists(12543251)) # آیا 12543251 در مجموعه وجود دارد؟  
باشد (به دلیل برخورد با 12543252) True باید # آیا 3252 در مجموعه وجود دارد؟  
print(s.exists(3252)) # آیا 3252 در مجموعه وجود دارد؟  
باشد True باید # آیا 8653 در مجموعه وجود دارد؟  
print(s.exists(8653)) # آیا 8653 در مجموعه وجود دارد؟  
  
آیا 245 در مجموعه وجود دارد؟ True  
آیا 10 در مجموعه وجود دارد؟ True  
آیا 10 در مجموعه وجود دارد؟ False  
آیا 12543252 در مجموعه وجود دارد؟ True  
آیا 12543251 در مجموعه وجود دارد؟ False  
آیا 3252 در مجموعه وجود دارد؟ True  
آیا 8653 در مجموعه وجود دارد؟ True
```

## یک کاربرد: جدول درهمسازی

همانطور که گفته شد، یکی از کاربردهای بسیار مهم درهمسازی در داده‌ساختاری به اسم **جدول درهمسازی (Hash Table)** است که تعمیمی از مثال زده در قسمت قبل (کلاس `my\_set`) است.

این داده‌ساختار از یک آرایه عادی برای نگهداری عناصر قرار داده شده در آرایه استفاده می‌کند و اندیس‌های مورد نظر را با استفاده از یک تابع درهمسازی به دست می‌آورد. به عبارت دیگر، در تعریف تابع درهمسازی،  $D$  همان مجموعه‌ایست که داده‌ها از آن می‌آیند و  $L$  مجموعه اندیس‌های آرایه (جدول هش) است.



شکل ۱: جدول درهمسازی

## برخی توابع درهمسازی

در این قسمت به معرفی چند تابع درهمسازی ساده، مانند آنچه در قسمت قبل گفته شد، می‌پردازیم. روش‌هایی که در این بخش گفته می‌شود برای حالتی استفاده می‌شود که بخواهیم یک عدد را هش کنیم (به عبارتی اعضای مجموعه‌ی  $D$  عده‌هایی با اندازه‌های مختلف هستند).

### روش باقیمانده تقسیم (Division Method)

در این روش به هر عدد، باقیمانده تقسیمیش بر یک عدد ثابت مانند  $m$  (که همان اندازه جدول هش است) را نسبت می‌دهیم. این دقیقا همان کاری است که در مثال قبلی (تابع `hash\_function` با  $\% \text{ } x$  انجام دادیم).

$$h(x) = x \bmod m$$

انتخاب مقدار مناسب برای  $m$  از اهمیت بالایی برخوردار است. به عنوان مثال، اگر  $m$  توانی از دو باشد (مثلاً  $2^k$ )، خروجی تابع درهمساز برابر  $k$  بیت کمارزش  $x$  می‌شود و در نتیجه بقیه بیت‌های عدد در خروجی تابع درهمساز نقشی ندارند. این می‌تواند منجر به توزیع نامناسب و برخوردهای زیاد شود، به خصوص اگر داده‌های ورودی الگوهای خاصی داشته باشند.

به طور کلی، بهتر است  $m$  یک عدد اول و دور از توانهای دو باشد تا توزیع بهتری از هش‌ها را فراهم کند و احتمال برخورد را کاهش دهد.

### روش ضرب (Multiplication Method)

یک استراتژی دیگر، روش ضرب است که دو مرحله دارد. ابتدا داده‌ی ورودی  $x$  ضرب در یک عدد ثابت  $A$  که  $0 < A < 1$  شده و قسمت اعشاری آن گرفته می‌شود. سپس این قسمت اعشاری در عدد صحیح  $m$  (اندازه جدول هش) ضرب شده و قسمت صحیح آن گرفته می‌شود.

$$h(x) = \lfloor m(xA \bmod 1) \rfloor$$

که منظور از باقیمانده به  $1$  ( $xA \bmod 1$ ) همان قسمت اعشاری عدد  $xA$  است.

خوبی این روش این است که دیگر انتخاب عدد  $m$  اینقدر مهم نیست. معمولاً در این روش  $m$  را توانی از  $2$  گرفته و  $A$  را به صورت عدد گویای  $\frac{s}{2^w}$  می‌گیریم که  $s$  یک عدد صحیح در بازه  $0 < s < 2^w$  است.

## روش همگانی (Universal Hashing)

حتی با انتخاب یک تابع درهمسازی خوب، بالاخره امکان دارد اعداد ورودی به صورتی انتخاب شوند (مثلاً توسط یک مهاجم بدخواه) که همه به یک خانه نگاشته شوند و عملکرد جدول هش را به بدترین حالت ( $O(n)$ ) برسانند.

یک روش برای مقابله با این مشکل این است که تعداد زیادی تابع درهمسازی آمده داشته باشیم و در ابتدای اجرای برنامه یکی از این توابع را به صورت مستقل از اعداد ورودی (به صورت تصادفی) انتخاب کرده و در ادامه‌ی برنامه از این تابع استفاده کنیم. به این کار \*\*روش همگانی درهمسازی (Universal Hashing)\*\* گفته می‌شود.

برای مثال، اگر از روش ضرب استفاده می‌کنیم، می‌توانیم تعداد زیادی زوج عدد  $A, m$  داشته باشیم و در هنگام اجرای برنامه یکی را انتخاب کرده و باقی کارها را براساس آن انجام دهیم.

تابت می‌شود که این روش به طور متوسط حتی روی بدترین ورودی ممکن، روش کارآمدی است و عملکرد متوسط ( $O(1)$ ) را برای عملیات‌های جدول هش تضمین می‌کند.

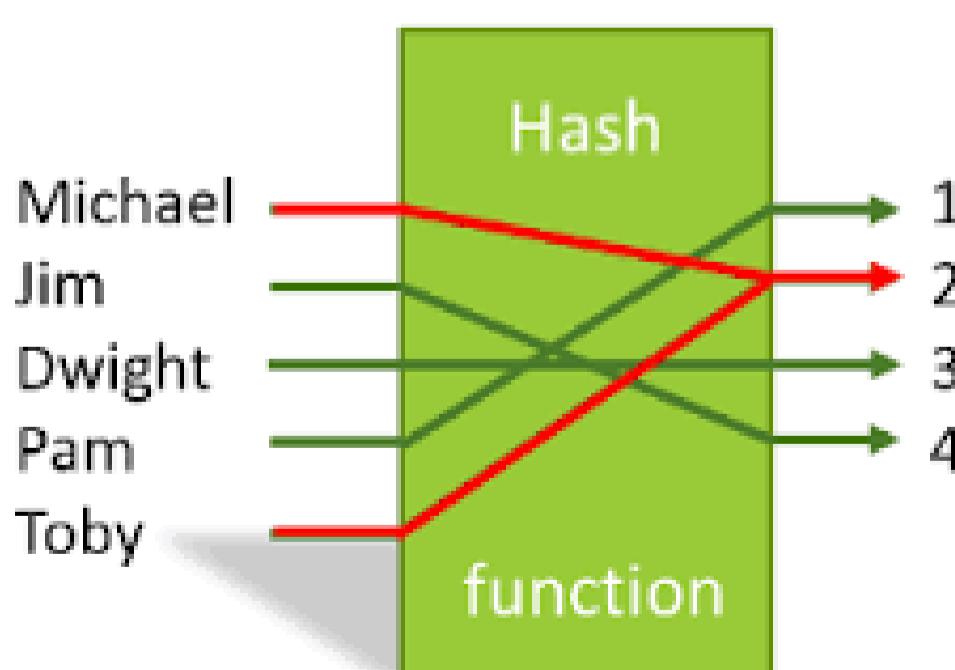
## برخورد و روش‌های مقابله با آن

بهوضوح نمی‌توانیم یک نگاشت یک به یک از یک مجموعه با کاردینالیتی  $C_1$  به مجموعه‌ای با کاردینالیتی  $C_2$  داشته باشیم که  $C_1 > C_2$ .

$L$  پس حتماً وقتی از تابع درهمساز  $h$  برای نگاشت مجموعه  $D$  (دامنه) به مجموعه  $L$  (هم‌دامنه یا جدول هش) که  $|L| > |D|$  استفاده می‌کنیم، حتماً عناصر متمایزی از  $D$  وجود دارند که به یک عنصر واحد در نگاشت شده‌اند.

به عنوان مثال، تابع درهمسازی ارائه شده در قسمت قبل ('%hash\_size'), هر دو عدد 20987 و 10987 را به عدد 987 می‌نگارد (به قطعه کدی که در ادامه آمده توجه کنید).

حال اگر بخواهیم همزمان این دو عدد را در مجموعه داشته باشیم، به مشکل می‌خوریم. به این اتفاق \*\*برخورد (Collision)\*\* می‌گوییم. حل کردن مشکل برخورد از مسائل پایه‌ای در طراحی توابع درهمسازی است. در عمل، ما دوست داریم از تابع درهمسازی‌ای استفاده کنیم که تعداد برخوردهای آن در سناریوهای مدنظر ما کمینه باشد.



شکل ۲: برخورد در تابع درهمسازی

```
In [3]: hash_size = 10**4 # 10,000
def hash_function(x):
    return x % hash_size
```

```

class my_set:
    table = [0] * hash_size
    def insert(self, x):
        self.table[hash_function(x)] = 1
    def exists(self, x):
        return self.table[hash_function(x)] == 1
    def remove(self, x):
        self.table[hash_function(x)] = 0

s = my_set() # يجاد يك شئء جديد از مجموعه

درج عدد آن 10987 = 10000 % 10987 است. هش آن 987 = 4^10 % 10987 است.
# آيا 10987 در مجموعه وجود دارد؟
print(s.exists(10987))
برخورد! برشورده! برشورده! برشورده!
# 987 = 4^10 % 10987 و 987 = 4^10 % 10987 است
# يرمي گرداند True به دليل برشورده!
exists(10987) # اين نشاندهنه مشكل برشورده است كه تابع ساده ما آن را مدیريت نمیکند
# آيا 10987 در مجموعه وجود دارد؟
print(s.exists(10987))

```

آيا 10987 در مجموعه وجود دارد؟  
آيا 10987 در مجموعه وجود دارد؟

## روش‌های مقابله با برشورده (Collision Resolution)

تا به حال سعی می‌کردیم تا توابعی پیدا کنیم که احتمال برشورده را کم کنیم. اما در هر صورت، اگر بخواهیم تعداد زیادی عدد را در یک جدول با اندازه‌ی کوچک ذخیره کنیم، حتماً برشورده خواهیم داشت.

برای حل این مشکل، روشهای زیادی وجود دارد از جمله:

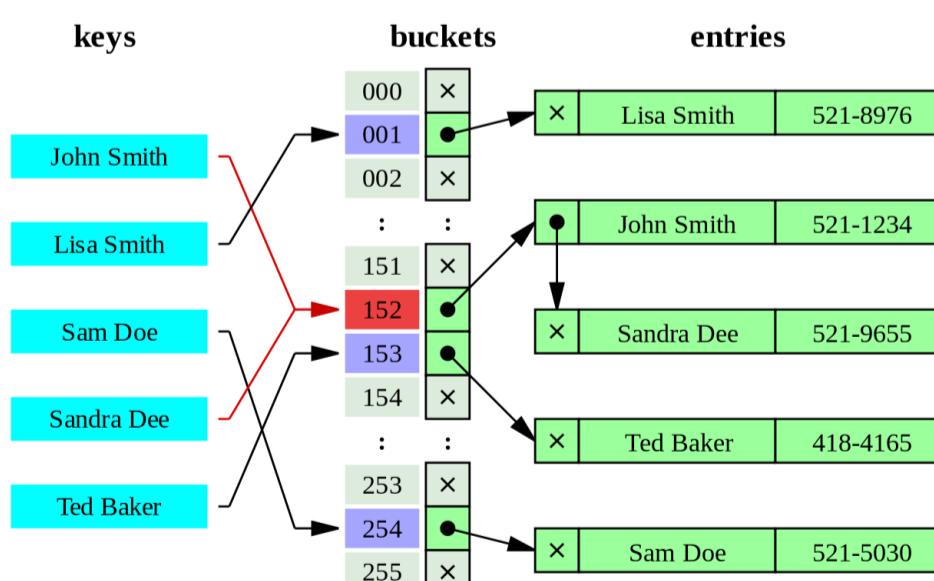
- زنجیره‌سازی (Chaining)
- آدرس‌دهی باز (Open Addressing)

که در ادامه این دو روشن را توضیح می‌دهیم.

## زنجیره‌سازی (Chaining)

در این روشن، هر خانه‌ی جدول (که به آن \*\*سطل (Bucket)\*\* هم می‌گویند) را یک \*لیست پیوندی\* (یا هر داده‌ساختار خطی دیگر مانند آرایه پویا) می‌گیریم.

حال هر عدد را که می‌خواهیم به خانه‌ای اضافه کنیم، خود عدد را در انتهای لیست آن خانه اضافه می‌کنیم. برای چک کردن وجود هم باید کل لیست آن خانه را بگردیم تا این که یا لیست تمام شود یا عدد مورد نظر را پیدا کنیم. برای حذف هم، عنصر را از لیست مربوطه حذف می‌کنیم.



شکل ۳: زنجیره‌سازی در جدول درهمسازی

مزیت این روشن، داینامیک بودن سایز داده‌ساختار است و مدیریت برشوردها را ساده می‌کند.

اگر طول یک لیست پیوندی خیلی زیاد شد، می‌توانیم از هش (جدول هش دیگر) و یا درخت دودوبی جستجو (مانند درخت قرمز-سیاه) به جای لیست پیوندی استفاده کنیم تا عملیات‌ها در آن خانه نیز کارآمد بمانند.

این روش شاید به نظر کارا نیاید، اما دقت کنید که فرض کرده بودیم قرار نیست تعداد زیادی برخورد داشته باشیم (یعنی توزیع هش‌ها نسبتاً یکنواخت است). در حالت میانگین، اگر  $n$  عنصر و  $m$  اندازه جدول هش باشد، طول متوسط هر لیست  $n/m$  است. اگر  $n/m$  یک ثابت کوچک باشد، عملیات‌ها در  $O(1)$  انجام می‌شوند.

کد این جدول درهمسازی تعمیم‌یافته در ادامه آمده است:

In [4]: # از سلول‌های قبلی تعریف شده‌اند hash\_size و hash\_function فرض می‌کنیم  
# hash\_size = 10\*\*4  
# def hash\_function(x): return x % hash\_size

class my\_chained\_set:  
 # با استفاده از جدول هش و روش زنجیره‌سازی (set) پیاده‌سازی یک مجموعه.  
 # table: آرایه‌ای که هر خانه آن یک لیست (زنگیره) برای نگهداری عناصر است.  
 # table = [[] for \_ in range(hash\_size)]  
  
 def insert(self, x):  
 # یک عنصر را به مجموعه اضافه می‌کند.  
 # ابتدا بررسی می‌کند که آپا عنصر قبلًا وجود دارد تا از درج تکراری جلوگیری کند.  
 if not self.exists(x):  
 self.table[hash\_function(x)].append(x) # (در حالت متوسط O(1)). عنصر را به انتهای لیست در سطл مربوطه اضافه می‌کند.  
  
 def exists(self, x):  
 # بررسی می‌کند که آپا یک عنصر در مجموعه وجود دارد یا خیر.  
 # لیست مربوط به هش عنصر را پیمایش می‌کند.  
 return x in self.table[hash\_function(x)] # در بدترین حالت (طول لیست 0) #  
  
 def remove(self, x):  
 # یک عنصر را از مجموعه حذف می‌کند.  
 # عنصر را از لیست مربوط به هش آن حذف می‌کند.  
 # ایجاد می‌شود ValueError، اگر عنصر وجود نداشته باشد.  
 if self.exists(x):  
 self.table[hash\_function(x)].remove(x) # در بدترین حالت (طول لیست 0) #  
  
# مثال استفاده  
s\_chained = my\_chained\_set() # ایجاد یک شیء جدید از کلاس my\_chained\_set  
s\_chained.insert(10)  
s\_chained.insert(245)  
print("آپا 10 در مجموعه زنجیره‌ای وجود دارد؟", s\_chained.exists(10)) # True  
  
s\_chained.insert(10987) # 987 = 10987 هش  
print("آپا 987 در مجموعه زنجیره‌ای وجود دارد؟", s\_chained.exists(987)) # True  
print("پاسخ صحیح") # True  
  
s\_chained.insert(20987) # 987 = 20987 (برخورد با 10987)  
print("آپا 987 در مجموعه زنجیره‌ای وجود دارد؟", s\_chained.exists(20987)) # True  
print("باید [10987, 20987] باشد # [20987, 10987] باشد") # محتوای سطل 987: [20987, 10987]

## آدرس دهنده باز (Open Addressing)

در این روش، بر عکس روش زنجیره‌سازی، اعداد را در همان خانه‌های جدول نگهداری می‌کنیم، با این تفاوت که اگر به خانه‌ی اصلی رفته و پر بود، به سراغ خانه‌ی ایمن (مشخص) دیگری می‌رویم.

مثلثاً فرض کنید که همیشه درست دنبال خانه‌ی بعدی برویم و اگر آنجا خالی بود، عددمان را در آنجا قرار دهیم، اگر نه باز هم ادامه دهیم تا به یک خانه‌ی خالی برسیم.

اگر فرض کنیم که اندازه‌ی جدول خیلی بزرگ‌تر از تعداد عناصر داخل آن باشد، به احتمال زیاد خیلی نیاز نمی‌شود که دنبال خانه‌ی دیگری برویم.

حال برای پیدا کردن یک عضو چه باید بکنیم؟ باید ابتدا سراغ خانه‌ی اصلی برویم و اگر خالی بود که عدد در جدول نیست و اگر هم خود عدد در داخلش قرار داشت که عدد پیدا شده و وجود دارد. اگر خالی نبود ولی عدد ما هم در آن نبود، باید به سراغ خانه‌ی بعدی برویم تا وقتی که به یک خانه‌ی خالی برسیم.

برای حذف کردن هم ابتدا باید عنصر پیدا شود، سپس تا آخر دنباله‌ی خانه‌های خالی رفته و آخرین عدد از این مجموعه (اگر وجود داشت) به جای خودش قرار دهیم. این کار برای جلوگیری از شکستن زنجیره جستجو در آینده است.

البته در این روش، اگر خود خانه خالی بود، حتماً به سراغ خانه‌ی بعدی نمی‌رویم، بلکه تابع درهمساز ما باید دو ورودی بگیرد  $(i, h(x))$  و ابتدا ما به خانه‌ی  $(0, h(x))$  می‌رویم. اگر پر بود به سراغ  $(1, h(x))$  و ... می‌رویم تا یک خانه‌ی خالی پیدا کنیم.

رسیدن به خانه‌ی خالی در جستجو برای یک کلید، به معنای نبود این کلید در جدول می‌باشد.

تمرین ۱:

\*\*پاسخ:\*\* خیر، الگوریتم به صورت ساده جواب نمی‌دهد. اگر یک عنصر را حذف کنیم و خانه آن را خالی کنیم، ممکن است زنجیره جستجو برای عناصر بعدی که از آن خانه عبور می‌کنند، شکسته شود. برای مثال، اگر  $h(y)=k$  و  $h(x)=k$  در  $k$  و  $x$  در  $k+1$  ذخیره شده باشد، با حذف  $x$ ، جستجو برای  $y$  ممکن است در  $k$  متوقف شود و  $y$  را پیدا نکند.

\*\*راحل:\*\* برای حل این مشکل، به جای اینکه خانه حذف شده را کاملاً خالی کنیم، آن را با یک مقدار خاص (مثلاً یک "پرچم حذف" یا "Tombstone") علامت‌گذاری می‌کنیم. این پرچم نشان می‌دهد که خانه خالی نیست اما عنصر آن حذف شده است. در عملیات جستجو، از روی این خانه‌ها عبور می‌کنیم تا به عنصر مورد نظر یا یک خانه واقع‌آغازی برسیم. در عملیات درج، می‌توان از خانه‌های علامت‌گذاری شده استفاده مجدد کرد.

برای ساختن این تابع (که مسیر جستجو را مشخص می‌کند) معمولاً یکی از سه روش زیر را استفاده می‌کنیم:

## ۱- کاوش خطی (Linear Probing)

این روش تقریباً همان روش رفتن به خانه‌ی بعدی است. روشی که در آن ترتیب بررسی ثابت و معمولاً با قدم‌های به اندازه 1 می‌باشد. به این صورت که اگر اعضای جدول اعداد صفر تا  $m-1$  باشند، تابع درهم‌ساز جدید را بر اساس تابع درهم‌ساز قبلی (تابع اولیه) به صورت زیر می‌سازیم:

$$h(x, i) = (h'(x) + i) \bmod m$$

که  $h'(x)$  تابع درهم‌ساز اولیه است و  $i$  تعداد تلاش برای یافتن جای خالی (از 0 شروع می‌شود).

اما در این روش معمولاً مشکلی به نام  $*\text{دسته‌بندی اولیه}$  (Primary Clustering)  $**\text{پیش می‌آید}$  که یعنی دنباله‌های پشت‌سر هم طولانی از خانه‌های پر ساخته شده و در نتیجه زمان جستجو افزایش می‌یابد.

امکان وجود برخورد همچنین در این روش وجود دارد.

## ۲- کاوش مربعی (Quadratic Probing)

روشی که در آن ترتیب بررسی تابعی از درجه دوم می‌باشد. در این روش فرم تابع مورد استفاده به صورت زیر خواهد بود:

$$h(x, i) = (h'(x) + c_1 i + c_2 i^2) \bmod m$$

که  $c_1$  و  $c_2$  ثابت‌هایی هستند. این روش مشکل دسته‌بندی اولیه را کاهش می‌دهد، اما باز هم اگر دو دسته تابع درهم‌ساز اولیه‌شان یکی باشد (یعنی  $h'(x)$  یکسانی داشته باشند)، باز هم دنباله‌شان یکسان خواهد بود.

به این مشکل نیز  $*\text{دسته‌بندی ثانوی}$  (Secondary Clustering)  $**\text{می‌گویند}$ .

## ۳- درهم‌سازی دوتایی (Double Hashing)

حالتی که توالی جستجو برای هر کلید ثابت است اما به وسیله یک درهم‌ساز دیگر محاسبه می‌شود. در این روش، برای حل مشکلات دو روش قبلی، از دو تابع درهم‌ساز متفاوت به صورت زیر استفاده می‌شود:

$$h(x, i) = (h_1(x) + i \cdot h_2(x)) \bmod m$$

که  $h_1(x)$  تابع درهم‌ساز اولیه و  $h_2(x)$  تابع درهم‌ساز ثانویه است.  $h_2$  باید طوری طراحی شود که هرگز صفر نشود و با  $m$  نسبت اول داشته باشد.

می‌توان گفت به طور تقریبی هیچ برخوردی در این روش اتفاق نمی‌افتد (یا به عبارت دقیق‌تر، توزیع برخوردها بسیار تصادفی‌تر و یکنواخت‌تر است) و بهترین عملکرد را در بین روش‌های آدرس‌دهی باز ارائه می‌دهد.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل نهم: توابع درهمسازی (قسمت دوم)

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

## فهرست

- تابع درهم سازی برای رشته ها
- آرایه با سایز پویا
- پیاده سازی آرایه پویا در زبان های مختلف
- در پایتون dict
- اهمیت تابع درهم سازی

## مقدمه

همان طور که در بخش قبلی دیدید، توابع درهمسازی (Hash Functions) ابزارهای قدرتمندی برای نگاشت داده‌های بزرگ به یک بازه کوچک‌تر هستند. همچنین با مفهوم **برخورد** (Collision) و روش‌های ابتدایی مقابله با آن آشنا شدیم.

در این بخش، به بررسی عمیق‌تر توابع درهمسازی، به خصوص برای **رشته‌ها**، و همچنین نحوه پیاده‌سازی و مدیریت جدول‌های درهمسازی در عمل (مانند آنچه در زبان پایتون استفاده می‌شود) خواهیم پرداخت. همچنین با مفهوم **آرایه‌های پویا** (Dynamic Arrays) و ارتباط آن‌ها با جدول‌های درهمسازی آشنا می‌شویم.

## تابع درهم سازی برای رشته ها

مثال: **add(s)**: داده‌ساختاری طراحی کنید که عملیات زیر را انجام دهد:

- exists(s)**: بررسی کن آیا رشته  $s$  در داده‌ساختار وجود دارد یا خیر.
- remove(s)**: رشته  $s$  را از داده‌ساختار حذف کن.

باید یک تابع هش برای یک رشته تعریف کنیم.

یکی از مرسوم‌ترین روش‌های هش کردن یک رشته، در نظر گرفتن یک رشته مثل  $S$  به صورت یک عدد چند رقمی در یک مبنای خاص (مثلاً مبنای  $P$ ) است و سپس استفاده از روش باقی‌مانده تقسیم روی آن است.

در این روش، هر کاراکتر رشته به یک عدد (مثلاً کد ASCII آن) تبدیل می‌شود و سپس این اعداد با وزن‌های مختلف (توان‌های  $P$ ) با هم جمع می‌شوند. برای جلوگیری از بزرگ شدن بیش از حد عدد، عملیات باقیمانده تقسیم بر یک عدد بزرگ (**mod**) در هر مرحله انجام می‌شود.

In [1]:  
P = 447 # مینا برای هش کردن رشته  
mod = 10\*\*9 + 7 # باقیمانده تقسیم

```
def h_string(s):  
    """  
    یک عدد اول بزرگ (مینا برای هش کردن رشته)  
    یک عدد اول بزرگ (برای عملیات باقیمانده تقسیم)  
    هش می‌کند (polynomial rolling hash).  
    این تابع یک رشته را با استفاده از روش چندجمله‌ای
```

```

Args:
    s (str): رشته ورودی.

Returns:
    int: مقدار هش رشته.
    """
result = 0
for c in s:
    # کاراکتر فعلی را اضافه ASCII ضرب کرده، کد P در هر مرحله، نتیجه قبلی را در
    # را محاسبه می‌کند mod و سپس باقیمانده تقسیم بر
    result = (result * P + ord(c)) % mod
return result

مثال استفاده
print('hello':, h_string("hello"))
print('world':, h_string("world"))

'hello': 100569579
'world': 849457674

```

در پایتون، تقریباً به همین روش تابع `hash()` برای اشیاء مختلف (از جمله رشته‌ها) پیاده‌سازی شده است:

```
In [2]: print(hash("Argfghfhgfan"), hash("Mohsen")) # چاپ مقادیر هش برای دو رشته
-1448292371394593810 -3893512945511981330
```

تابع  $h(s)$  (که در کد `h\_string` نامگذاری شد) عدد زیر را برمی‌گرداند:

$$s = c_1 c_2 \dots c_k$$

$$h(s) = (c_1 \cdot P^{k-1} + c_2 \cdot P^{k-2} + \dots + c_{k-1} \cdot P + c_k) \pmod{mod}$$

(عملیات باقیمانده تقسیم بر `mod` در هر مرحله اعمال می‌شود تا از سرریز شدن عدد جلوگیری شود.)

حال با استفاده از این تابع درهمسازی و استفاده از یکی از روش‌های مقابله با برخورد (که در بخش قبلی معرفی شد)، می‌توان مسئله را حل کرد.

(البته احتمال برخورد آن قدر کم است که استفاده از داده‌ساختار **BST** (درخت جستجوی دودویی) هم کافی است، اما جدول هش معمولاً عملکرد بهتری دارد.)

در روش اول، از \*\*«زنگیره‌سازی» (Chaining)\*\* استفاده می‌کنیم که در بخش قبلی توضیح داده شد. در این روش، هر خانه از جدول هش، یک لیست پیوندی (یا آرایه پویا) است که عناصر با هش یکسان در آن ذخیره می‌شوند.

زمان اجرای این روش برای عملیات‌های **add**, **exists**, **remove** در حالت میانگین  $O(L_{avg})$  است، که  $L_{avg}$  متوسط طول لیست‌ها (زنگیره‌ها) است. اگر  $n$  تعداد عناصر و  $m$  اندازه جدول هش باشد، زمان اجرای این روش برای  $O(n/m)$  خواهد شد. بنابراین، اگر  $n/m$  یک ثابت کوچک باشد، زمان اجرای عملیات‌ها از  $O(1)$  خواهد بود. در بدترین حالت، اگر همه عناصر به یک سطح هش شوند، زمان اجرا  $O(n)$  خواهد شد.

```

In [3]: # تعریف اندازه جدول هش (از بخش قبلی)
hash_size = 10**4 # 10,000

# تابع هش برای رشته‌ها (از بخش قبلی)
def h(s):
    return h_string(s) % hash_size # استفاده از h_string

class my_chained_set:
    # پیاده‌سازی یک مجموعه با استفاده از جدول هش و روش زنگیره‌سازی.

    def __init__(self):
        # متاداده: جدول هش را با لیست‌های خالی مقداردهی اولیه می‌کند.
        self.table = [[] for _ in range(hash_size)]

    def insert(self, x):
        # یک عنصر را به مجموعه اضافه می‌کند.
        # اگر عنصر قبلاً وجود نداشت، آن را به لیست مربوط به هش آن اضافه می‌کند.
        if not self.exists(x):
            self.table[h(x)].append(x) # در حالت متوسط برای O(1) append

    def exists(self, x):
        # بررسی می‌کند که آیا یک عنصر در مجموعه وجود دارد یا خیر.
        # پایتون برای جستجو در لیست استفاده می‌کند 'in' از عملگر.
        # در بدترین حالت (طول لیست) در بدلین حالت (طول لیست)
        return x in self.table[h(x)] # O(1)

    def remove(self, x):
        # یک عنصر را از مجموعه حذف می‌کند.
        # اگر عنصر وجود داشت، آن را از لیست مربوط به هش آن حذف می‌کند.
        # بهتر است قبل از حذف، وجود عنصر را چک کنیم #
        if self.exists(x): # self.exists(x) # در بدلین حالت (طول لیست)
            self.table[h(x)].remove(x) # O(1)

```

```
#مثال استفاده
htable1 = my_chained_set() # ایجاد یک شیء از کلاس my_chained_set

def add1(s):
    #تابع کمکی برای نمایش عملیات درج در htable1
    print(s, end=' ')
    در حالت متوسط (0(1))
        if htable1.exists(s): # چاپ میشود "exists"
            print("exists")
        else:
            print("inserted")
            htable1.insert(s) # 0(1)

add1("Ali")
add1("vezvayi")
add1("Ali") # این بار "exists" چاپ نمیشود
```

Ali inserted  
vezvayi inserted  
Ali exists

در روش دوم، میتوانیم از نوع داده‌ی داخلی پایتون یعنی **set** استفاده کنیم. **set** در پایتون خود از روش «آدرسدهی باز» یا ترکیبی از آن و زنجیره‌سازی (بسته به پیاده‌سازی داخلی پایتون) استفاده می‌کند.

زمان اجرای این روش برای عملیات‌های **add**, **exists**, **remove** در حالت میانگین ( $O(k)$ ) است، که طول رشته است. این به دلیل نیاز به محاسبه هش رشته در هر عملیات است.

In [4]: # htable2: در پایتون set استفاده از نوع داده داخلی.  
خود پایتون استفاده می‌کند: hash() پایتون به صورت داخلی از تابع set: توجه که ما تعریف کردیم (h(s) نه از تابع htable2 = set()

```
def add2(s):
    #تابع کمکی برای نمایش عملیات درج در htable2
    print(s, end=' ')
    #بررسی وجود عنصر در set
    #است (با فرض تابع هش خوب) 0(1) پایتون به طور متوسط set در 'in' عملیات
    if s in htable2: # 0(1)
        print("exists")
    else:
        print("inserted")
        htable2.add(s) # 0(1)

#در تابع هش داخلی پایتون بسیار پایین است (collision) این روش به دلیل اینکه احتمال برخورد به خوبی کار می‌کند

add2("Ali")
add2("alipour")
add2("Ali") # این بار "exists" چاپ نمیشود
```

Ali inserted  
alipour inserted  
Ali exists

روش بالا (استفاده از **set** پایتون) در  $n \leq 10^5$  کار می‌کند، چرا که احتمال برخورد خیلی پایین است (با  $h\_string \mod = 10^{89+7}$ ). این به دلیل توزیع بسیار خوب تابع هش است.

در روش سوم میتوان از **BST** (درخت جستجوی دودویی) استفاده کرد که زمان اجرایی ( $O(nk)$  برای ساخت درخت) و ( $O(k \log n)$  برای جستجو/درج/حذف) را خواهد داشت، که طول رشته است. این روش در مقایسه با جدول هش، معمولاً کندتر است اما در بدترین حالت، عملکرد تضمین شده‌ای دارد.

## تمرین

تابع درهمسازی پیاده‌سازی کنید که از قاعده زیر پیروی کند:

یک عدد ۱۶ بیتی (مثلاً `result`) در نظر بگیرید که در ابتداء ۰ است. سپس به ازای هر کاراکتر از رشته ورودی، آن را یک بار **شیفت دوری راست (Circular Right Shift)** داده و سپس آن را با عدد ASCII آن کاراکتر جمع کند.

(شیفت دوری راست به این معنی است که از سمت راست خارج می‌شوند، از سمت چپ وارد می‌شوند).

```
In [5]: def test_hash(s):
    """
    این تابع یک رشته را با استفاده از یک الگوریتم هش سفارشی (شیفت دوری راست و جمع) هش می‌کند.

    Args:
        s (str): رشته ورودی.

    Returns:
        int: مقدار هش ۱۶ بیتی.

    """
    result = 0 # عدد ۱۶ بیتی که در ابتداء ۰ است
    bit_length = 16 # طول بیت برای عملیات شیفت دوری.

    for ch in s:
        # ۱. شیفت دوری راست (Circular Right Shift)
        # بیت کم‌ارزش (سمت راست) را جدا کن
        lowest_bit = result & 1
        # شیفت راست عادی ( تقسیم بر ۲ )
        result = result >> 1
        # بیت جدا شده را به پرازش‌ترین موقعیت (سمت چپ) اضافه کن
        result = result | (lowest_bit << (bit_length - 1))

        # ۲. کاراکتر ASCII جمع می‌کند
        # اطمینان از ماندن در محدوده ۱۶ بیتی با استفاده از عملگر پیمانه
        result = (result + ord(ch)) % (1 << bit_length)

    return result

مثال استفاده (برای تست)
# ord('H') = 72
# ord('a') = 97
# ord('s') = 115
# ord('h') = 104
ممکن است ۱۰۴ ندهد، زیرا الگوریتم هش پیچیده‌تر از صرفاً آخرین کاراکتر است "Hash". این تست برای مقدار هش نهایی به ترتیب و مقدار تمام کاراکترها بستگی دارد
print(f"Hash": {test_hash('Hash') == 104})
print(f"Hash": {test_hash('Hash')})
print(f"Test": {test_hash('Test')})
print(f"Hello": {test_hash('hello')})
print(f"World": {test_hash('world')})
```

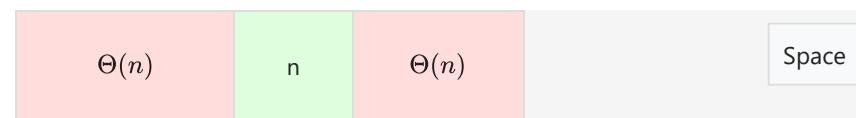
برابر ۱۰۴ است؟ آیا هش  
'Hash': 49346  
'Hash': 16593  
'Test': 8403  
'Hello': 53451  
'World': 53451

# آرایه با سایز پویا (Dynamic Array)

اگر شما بخواهید آرایه‌ای که سایز متغیری داشته باشد (یعنی بتوانید به آن عنصر اضافه یا از آن حذف کنید بدون اینکه نگران پرشدن یا خالی شدن حافظه باشید)، چه می‌کنید؟

در واقع ما به یک داده‌ساختار نیاز داریم که بتواند عملیات زیر را انجام دهد و در عین حال کارایی بالایی داشته باشد:

Dynamic array	Array	Linked list	عملیات
$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	Indexing
amortize $\Theta(1)$	N/A	$\Theta(1)$	Insertion/deletion at end
$\Theta(n)$	N/A	$\Theta(1)$	Insertion/deletion in middle
$\Theta(n)$	0	$\Theta(n)$	Wasted space (average)



# درج در Dynamic Array

اگر هر زمانی که سایز آرایه پر شود، یک آرایه دیگر با سایز یکی بزرگتر بسازیم و تمام اطلاعات را کپی کنیم، مشکل رفع می‌شود؟

\*پاسخ:\*\* خیر، این روش کارآمد نیست. اگر آرایه در هر بار پر شدن فقط یک واحد افزایش یابد، هزینه درج به صورت سرشکن  $O(n)$  خواهد بود، زیرا برای درج  $n$  عنصر، مجموعاً  $O(n^2)$  عملیات کپی انجام می‌شود.

یک روش پیشنهادی برای تغییر اندازه آرایه‌های پویا، \*\*دو برابر کردن ظرفیت\*\* است؛ مثلاً زمانی که آرایه پر شد، اندازه آن را دو برابر می‌کنیم.

\*چرا با این کار درج کردن به طور سرشکن از  $O(1)$  است؟\*

\*پاسخ:\*\* همانطور که در بخش تحلیل سرشکن دیدیم، اگر ظرفیت آرایه را هر بار دو برابر کنیم، هزینه کل  $m$  عملیات درج (شامل کپی کردن) از  $O(m)$  خواهد بود. به عنوان مثال، برای درج  $n$  عنصر، هزینه کل کپی کردن‌ها  $2n \approx 2 \cdot 2^k \approx 2 \cdot 2^k < 2 \cdot 2^k + 1 + 2 + 4 + \dots + 2^k < 2 \cdot 2^k$  است. بنابراین، هزینه سرشکن هر عملیات درج  $O(1)$  خواهد بود.

No description has been provided for this image 

In [6]: ما از لیست‌های پایتون در اینجا استفاده می‌کنیم، حتی اگر لیست پایتون خودش یک آرایه پویا باشد. این پیاده‌سازی برای نشان دادن مفهوم آرایه پویا است.

```
class DynamicArray:
    def __init__(self):
        متده سازنده: یک آرایه پویا خالی ایجاد می‌کند
        self.a = [None] # از اولیه 1
        تعداد عناصر واقعی در آرایه
        self.size = 0 # می‌باشد

    def insert(self, x):
        یک عنصر را به انتهای آرایه پویا اضافه می‌کند
        if self.size == len(self.a):
            اگر آرایه پر بود، ظرفیت را دو برابر کن # اینجا
            new_capacity = 2 * len(self.a) # فعالی قرار می‌دهد
            new_array = [None] * new_capacity # بیشتر
            for i in range(self.size):
                کپی کردن عناصر از آرایه قدیمی به جدید
                new_array[i] = self.a[i]
            به روزرسانی اشاره‌گر آرایه
            self.a = new_array # اینجا

        self.a[self.size] = x # اضافه کردن عنصر جدید در انتهای فضای استفاده شده
        self.size += 1 # افزایش تعداد عناصر

    # مثال استفاده
    da = DynamicArray()
    print("اندازه(a):", da.size) # 0
    print("آرایه(a):", da.a) # [None]

    da.insert(1)
    print("اندازه(a):", da.size) # 1
    print("آرایه(a):", da.a) # [1]

    اتفاق می‌افتد (ظرفیت از 1 به 2 می‌شود) در اینجا
    print("اندازه(a):", da.size) # 2
    print("آرایه(a):", da.a) # [1, 2]

    اتفاق می‌افتد (ظرفیت از 2 به 4 می‌شود) در اینجا
    print("اندازه(a):", da.size) # 3
    print("آرایه(a):", da.a) # [1, 2, 3, None]

    اتفاق می‌افتد (ظرفیت از 4 به 8 می‌شود) در اینجا
    da.insert(5)
    print("اندازه(a):", da.size) # 5
    print("آرایه(a):", da.a) # [1, 2, 3, 4, 5, None, None, None]
```

```
اندازه (size): 0
آرایه (a): [None]
اندازه (size): 1
آرایه (a): [1]
اندازه (size): 2
آرایه (a): [1, 2]
اندازه (size): 3
آرایه (a): [1, 2, 3, None]
اندازه (size): 5
آرایه (a): [1, 2, 3, 4, 5, None, None, None]
```

# حذف در Dynamic Array

با حذف کردن عناصر از آخر آرایه، برای اینکه حافظه از  $O(n)$  بماند، باید در موقعی ظرفیت آن را کم کرد.

اگر وقتی ظرفیت آرایه دو برابر سایز آرایه باشد، اندازه آن را کاهش دهیم، مشکل رفع می‌شود؟

پاسخ: خیر، این روش می‌تواند منجر به **لرزش (Thrashing)** شود. اگر ظرفیت را دقیقاً زمانی که به نصف می‌رسد کاهش دهیم، و سپس عملیات‌های درج و حذف متناوباً انجام شود، ممکن است آرایه به طور مداوم تغییر اندازه دهد (دو برابر شود، نصف شود، دو برابر شود، ...) که بسیار ناکارآمد است.

\*چه راه حلی پیشنهاد می‌دهید؟\*

پاسخ: یک راه حل معمول این است که ظرفیت آرایه را تنها زمانی کاهش دهیم که تعداد عناصر به  $\frac{1}{4}$  ظرفیت فعلی برسد. این کار باعث می‌شود که عملیات‌های تغییر اندازه (کپی کردن) کمتر اتفاق بیفتند و هزینه سرشکن حذف نیز  $O(1)$  باقی بماند. به عبارت دیگر، یک **فاکتور کاهش ظرفیت (Shrink Factor)** را معرفی می‌کنیم که معمولاً کمتر از فاکتور رشد (مثلاً  $1/4$  یا  $1/2$ ) است.

# پیاده سازی Dynamic Array در زبان های مختلف

## پیاده سازی ضریب رشد

(3/2) 1.5	Java ArrayList
(9/8) 1.125	Python PyListObject (list)
2	C++ 5.2.0 vector
(3/2) 1.5	Facebook folly/FBVector

# dict در پایتون

داده‌ساختار **dict** در پایتون، یک نوع دیکشنری (Dictionary) است که برای ذخیره جفت‌های کلید-مقدار (key-value pairs) استفاده می‌شود.

برخلاف رشته‌ها و لیست‌ها که اندیس‌گذاری آن‌ها به صورت عددی و ترتیبی است، اندیس داده‌ها در دیکشنری به دلخواه برنامه‌نویس مشخص می‌شود (که این اندیس‌ها **کلید (key)** گفته می‌شوند).

این کلیدها می‌توانند از هر جنسی باشند (مثلاً اعداد صحیح، رشته‌ها، تاپل‌ها) و لزومی ندارد که عدد صحیح باشند، به شرطی که `Comparable`\* باشند (یعنی بتوان برای آن‌ها یک مقدار هش منحصر به فرد تولید کرد).

```
In [7]: # مثال استفاده از دیکشنری در پایتون
          دیکشنری با کلیدهای عددی و اعشاری "20"
grade = {20: 0, 3.14: "20"} # اضافه کردن جفت کلید-مقدار با کلید رشته‌ای 10
grade["eshagh"] = 10
grade["bagher"] = 1
grade["salim"] = "??"
print(grade) # چاپ دیکشنری
{20: 0, 3.14: '20', 'eshagh': 10, 'bagher': 1, 'salim': '??'}
```

دیکشنری در پایتون به صورت \*\*«آدرس دهی باز» (Open Addressing) پیاده سازی شده است. این یعنی به جای استفاده از لیست های پیوندی برای حل برخوردها، عناصر مستقیماً در خانه های جدول هش ذخیره می شوند.

در ادامه به نحوه پیاده‌سازی دیکشنری در پایتون می‌پردازیم.

یک دیکشنری خالی، در واقع یک لیست داخلی با طول اولیه ۸ است (برای پایتون‌های جدیدتر). این لیست شامل خانه‌هایی برای ذخیره کلیدها، مقادیر و هشنهای آن‌هاست.

No description has been provided for this image

In [8]: d = {}

هش، هر کلید تولید می‌شود تا بتوان آن را در لیست ذخیره کرد.

```
In [9]: # تابعی برای نمایش نمایش دودویی (باینری) یک عدد (برای درک بهتر هشها)
def bits(n):
    # این تابع نمایش 32 بیتی (یا 64 بیتی بسته به سیستم) یک عدد را برمی‌گرداند.
    # برای اطمینان از مثبت بودن عدد در نمایش باینری (مخصوصاً برای اعداد منفی) n += 2**32
    # و گرفتن 32 بیت آخر bin() از ابتدای خروجی 'حذف' 0[-32:]
    return bin(n)[-32:] # نمایش 32 بیتی عدد 2
print("نمایش 32 بیتی عدد 2", bits(2), end='\n\n')

print("John Snow:", bits(hash("John Snow")), "John Snow")
print("Grades are not important:", bits(hash("Grades are not important")), "Grades are not important")
print("(3,2,1): هش تاپل", bits(hash((1, 2, 3))), (1, 2, 3)) # هستند hashable
```

هش 'John Snow': 0000000000101010001111000110001 John Snow  
هش 'Grades are not important': 1110010010100000001110111001111 Grades are not important  
هش تابل (3 , 2 , 1) 11100111000110111100110011101011 : (3 , 2 , 1)

هش، دو رشته مشابه می‌توانند تفاوت زیادی داشته باشد، حتی اگر فقط یک حرف آن‌ها متفاوت باشد. این ویژگی برای توانع هش، خوب، مطلوب است.

```
In [10]: k1 = bits(hash('Fail')) # هش رشته 'Fail'  
k2 = bits(hash('Faal')) # هش رشته 'Faal'  
  
# مقایسه بیت به بیت دو هش و مشخص کردن بیت‌های نابرابر  
diff = ['^' if a != b else ' ' for a, b in zip(k1, k2)] # '^' هش 'Fail':, k1  
print("هش 'Fail':", k1)  
print("هش 'Faal':", k2)  
print("چاپ تفاوت بیت‌ها:", ''.join(diff)) # تفاوت بیت‌ها
```

اندیس، یک کلید در جدول، هش، یا یعنون، بار  $n$  بست سمت راست (کمازش)، هش، آن کلید است، حاب، که  $2^n$  بار با `size\_of\_hash\_table` (ظرفیت فعل، جدول، هش) است.

در ابتدا  $n \equiv 3$  است (زیرا  $2^3 = 8$  که ظرفیت اولیه حدوداً است).

```
In [11]: d = { } # دیکشنری خالی
d["ali"] = 6 # دادن 'ali': 6
```

```

b = bits(hash("ali")) # محاسبه و نمایش هش 'ali'
print("هش 'ali':", b)
print("بیت آخر هش که اندیس را تعیین میکند 3 # بیت آخر هش 3")
b[-3:] # بیت آخر هش 3
b[3] # بیت آخر هش 3

```

No description has been provided for this image

```

In [12]: d["reza"] = 9 # درج 'reza': 9
b = bits(hash("reza")) # محاسبه و نمایش هش 'reza'
print("هش 'reza':", b)
print("بیت آخر هش 3 # بیت آخر هش 3")
b[-3:] # بیت آخر هش 3

```

No description has been provided for this image

```

In [13]: d["nasim"] = 8 # درج 'nasim': 8
b = bits(hash("nasim")) # محاسبه و نمایش هش 'nasim'
print("هش 'nasim':", b)
print("بیت آخر هش 3 # بیت آخر هش 3")
b[-3:] # بیت آخر هش 3

```

No description has been provided for this image

```

In [14]: d = {'ali': 6, 'reza': 9, 'nasim': 8} # ترتیب کلیدها در دیکشنری
print("ترتیب کلیدها در دیکشنری", d.keys())
# ایجاد یک دیکشنری دیگر با همان عناصر اما ترتیب درج متفاوت
d_reordered = {'nasim': 8, 'reza': 9, 'ali': 6}
print("ترتیب کلیدها در دیکشنری", d_reordered.keys())

```

No description has been provided for this

image

No description has been provided for this

image

\*\*پاسخ:

\*چرا این دو دیکشنری با اینکه ترتیب متفاوتی دارند، در پایتون ۲ خروجی یکسانی دارند؟ چرا در پایتون ۳ به این صورت نیست؟\*

- \*\*در پایتون 2:\*\* دیکشنری‌ها به صورت پیشفرض \*\*نامرتب (unordered)\*\* بودند. ترتیب نمایش کلیدها (مثلاً هنگام چاپ `d.keys()`) به پیاده‌سازی داخلی جدول هش بستگی داشت و معمولاً بر اساس ترتیب خانه‌های پر شده در جدول هش بود، نه ترتیب درج. بنابراین، حتی اگر کلیدها به ترتیب متفاوتی درج می‌شدند، اگر در نهایت به یک وضعیت داخلی جدول هش مشابه منجر می‌شدند، خروجی `keys()` یکسان بود.
- \*\*در پایتون 3.7 به بعد (و در پایتون 3.6 به عنوان یک ویژگی پیاده‌سازی):\*\* دیکشنری‌ها به صورت پیشفرض \*\*مرتب (ordered)\*\* شدند. این بدان معناست که ترتیب درج کلیدها حفظ می‌شود. یعنی اگر کلیدها را به ترتیبی درج کنید، هنگام پیمایش (مثلاً با `d.keys()`) به همان ترتیب درج شده ظاهر می‌شوند. این تغییر، یک ویژگی پیاده‌سازی بود که سپس به عنوان یک استاندارد در زبان پذیرفته شد. به همین دلیل، در پایتون 3، دو دیکشنری با ترتیب درج متفاوت، خروجی `keys()` متفاوت خواهند داشت.

# برخورد در dict

اگر برخورد (Collision) داشته باشیم باید چه کنیم؟

در پیاده‌سازی دیکشنری پایتون، از روش آدرس دهی باز (Open Addressing) برای حل برخورد استفاده می‌شود. به طور خاص، پایتون از یک نوع کاوش ترکیبی (Hybrid Probing) استفاده می‌کند که از نظر زمانی اصلاً ایده آل نیست، چرا که دنباله‌های پشت سر هم در جدول ساخته می‌شود و در طول زمان جدول کند می‌شود. این مشکل به دلیل دسته‌بندی (Clustering) رخ می‌دهد.

در پایتون، بر اساس  $n - 32$  بیت سمت چپ هش (که از آن‌ها استفاده نشده)، یک کاوش (که تقریباً تصادفی است) ایجاد می‌شود. (به دلیل پیچیدگی این کاوش، آن را بررسی نمی‌کنیم).

```
In [15]: d = {} # ایجاد یک دیکشنری خالی
d["arya"] = 666 # درج 'arya': 666
b = bits(hash("arya")) # هش 'arya'
print("هش 'arya':", b)
print("3 بیت آخر هش 'arya':", b[-3:]) # 3 بیت آخر هش 'arya': 00100110001001000100100110001010
3 بیت آخر هش 'arya': 010
```

No description has been provided for this image

```
In [16]: d["john"] = 858 # درج 'john': 858
b = bits(hash("john")) # هش 'john'
print("هش 'john':", b)
print("3 بیت آخر هش 'john':", b[-3:]) # 3 بیت آخر هش 'john': 00100001111010010101111100101110
3 بیت آخر هش 'john': 110
```

No description has been provided for this image

```
In [17]: d["snow"] = 1 # درج 'snow': 1
b = bits(hash("snow")) # هش 'snow'
print("هش 'snow':", b)
print("3 بیت آخر هش 'snow':", b[-3:]) # 3 بیت آخر هش 'snow': 0101111000000110101010111111101
3 بیت آخر هش 'snow': 101
```

No description has been provided for this image

```
In [18]: d["alive"] = 2 # درج 'alive': 2
b = bits(hash("alive")) # هش 'alive'
print("هش 'alive':", b)
print("3 بیت آخر هش 'alive':", b[-3:]) # 3 بیت آخر هش 'alive': 1001011011111000000100100100010
3 بیت آخر هش 'alive': 010
```

No description has been provided for this image

```
In [19]: d["dead"] = 3 # درج 'dead': 3
b = bits(hash("dead")) # هش 'dead'
print("هش 'dead':", b)
print("3 بیت آخر هش 'dead':", b[-3:]) # 3 بیت آخر هش 'dead': 0001000100000001000100010
3 بیت آخر هش 'dead': 000
```

'dead': 0111111011100000010001010100000  
'dead': 000 بیت آخر هش 3

No description has been provided for this image

```
In [20]: d = {'arya': 666, 'john': 858, 'snow': 1, 'alive': 2, 'dead': 3}  
print(d) ترتیب کلیدها در دیکشنری("d:=", d.keys())  
  
e = {'snow': 1, 'john': 858, 'arya': 666, 'dead': 3, 'alive': 2}  
print(e) ترتیب کلیدها در دیکشنری("e:=", e.keys())  
  
باشد، زیرا محتوا یکسان است True باید # برابر هستند؟ و d آیا(")  
print(d == e)  
  
d: dict_keys(['arya', 'john', 'snow', 'alive', 'dead'])  
e: dict_keys(['snow', 'john', 'arya', 'dead', 'alive'])  
برابر هستند؟ e و d آیا True
```

No description has been provided for this

image

No description has been provided for this

image

\*\*چرا این دو دیکشنری با اینکه مقادیر یکسانی دارند، ترتیب متفاوتی دارند؟\*\*

\*\*پاسخ:\*\* این سوال به تفاوت رفتار دیکشنری‌ها در پایتون 2 و پایتون 3 (قبل از 3.7) و همچنین به نحوه مدیریت برخوردها اشاره دارد.

- \*\*در پایتون 2:\*\* دیکشنری‌ها نامرتب بودند و ترتیب کلیدها هنگام چاپ یا پیمایش، به ترتیب داخلی خانه‌های پر شده در جدول هش بستگی داشت. این ترتیب می‌توانست حتی برای دیکشنری‌های با محتوای یکسان اما ترتیب درج متفاوت، فرق کند.
- \*\*در پایتون 3.6 به بعد:\*\* دیکشنری‌ها ترتیب درج را حفظ می‌کنند. یعنی کلیدها به همان ترتیبی که اضافه شده‌اند، پیمایش می‌شوند. بنابراین، اگر دو دیکشنری با محتوای یکسان اما ترتیب درج متفاوت ایجاد شوند، `d.keys()` آنها نیز ترتیب متفاوتی خواهد داشت.
- \*\*تأثیر برخورد:\*\* حتی در پایتون 3، اگر برخوردها اتفاق بیفتد، ترتیب نهایی کلیدها در `d.keys()` ممکن است تحت تأثیر قرار گیرد، زیرا عناصر باید در خانه‌های جایگزین ذخیره شوند. با این حال، پایتون تلاش می‌کند ترتیب درج را تا حد امکان حفظ کند.

## جستجو در dict

جستجو در دیکشنری (با استفاده از `dict[key]` یا `key in dict`) شبیه به درج کردن عمل می‌کند.

الگوریتم جستجو ابتدا مقدار هش کلید را محاسبه می‌کند و به خانه اصلی آن در جدول هش می‌رود. اگر کلید در آن خانه یافت شد، عملیات موفق است. اگر خانه پر بود اما کلید مورد نظر در آن نبود (یعنی یک برخورد رخداده و عنصر دیگری در آن خانه ذخیره شده)، الگوریتم به دنباله کاوش (probe sequence) ادامه می‌دهد.

جستجو تا زمانی ادامه می‌یابد که:

- کلید مورد نظر پیدا شود.

• به یک سطر (خانه) کاملاً خالی برسد (که نشان می‌دهد کلید در جدول نیست).

```
In [21]: # از متالهای قبلی پر شده است d فرض می‌کنیم دیکشنری
# d = {'arya': 666, 'john': 858, 'snow': 1, 'alive': 2, 'dead': 3}

b = bits(hash("fire")) # هش رشته 'fire'
print("هش 'fire':", b)
print("بیت آخر هش 3 'fire':", b[-3:]) # بیت آخر هش 3 وجود دارد؟ d در دیکشنری 'fire' آیا

print('جستجو برای \'fire\' وجود دارد؟', "fire" in d) # در دیکشنری 'fire' آیا

'fire': 10001001000100101111001111101100
3 'fire': 100
بیت آخر هش 3 وجود دارد؟ d در دیکشنری 'fire' وجود دارد؟ آیا
```

No description has been provided for this image

# حذف کردن در dict

برای حذف کردن یک کلید از دیکشنری باید چه کرد؟

آیا خالی کردن سطر کلید مورد نظر کافی است؟ پاسخ:\*

حذف "snow"

No description has been provided for this

image

No description has been provided for this

image

جست و جو "dead"

No description has been provided for this

image

No description has been provided for this

image

اگر فقط سطر را خالی کنیم، با جست و جو "dead" به نتیجه می‌رسیم که "dead" در جدول وجود ندارد، که غلط است. این اتفاق می‌افتد چون جستجو برای "dead" به خانه‌ای که "snow" در آن بود می‌رسد، آن خانه را خالی می‌بیند و فرض می‌کند که "dead" وجود ندارد، در حالی که "dead" در واقع در ادامه دنباله کاوش قرار دارد.

\*چه پیشنهادی برای رفع این مشکل دارید؟\*

\*\*پاسخ: برای رفع این مشکل، به جای اینکه خانه حذف شده را کاملاً خالی کنیم، آن را با یک مقدار خاص (مثلاً یک "پرچم حذف" یا \*\*Tombstone\*\*) علامت‌گذاری می‌کنیم. این پرچم نشان می‌دهد که خانه خالی نیست اما عنصر آن حذف شده است. در عملیات جستجو، از روی این خانه‌های علامت‌گذاری شده عبور می‌کنیم تا به عنصر مورد نظر یا یک خانه واقعاً خالی برسیم. در عملیات درج، می‌توان از خانه‌های علامت‌گذاری شده استفاده مجدد کرد.

می‌توان به جای خالی کردن سطر، مقدار \*\*dummy\*\* (یک شیء خاص برای نشان دادن حذف) را در آن قرار دهیم.

```
In [23]: d = {'arya': 666, 'john': 858, 'snow': 1, 'alive': 2, 'dead': 3}
del d['snow'] # از دیکشنری 'snow' حذف
print('پس از حذف 'dead' برسی وجود آیا') # در دیکشنری وجود دارد؟ آیا
در دیکشنری وجود دارد؟ 'dead' آیا True
```

## حذف "snow" و جست و جو "dead"

No description has been provided for this image

No description has been provided for this image

با اینکار در مرحله جست و جو "dead" وقتی به سطر 101 می‌رسیم، با توجه به اینکه hash آن خالی است ولی این سطر مقدار دارد از روی آن می‌گذریم.

```
In [24]: d = {'arya': 666, 'john': 858, 'snow': 1, 'alive': 2, 'dead': 3}
del d["snow"], d['john'], d['arya'], d['alive'] # حذف چند عنصر
print('بررسی وجود آیا') # در دیکشنری وجود دارد؟ آیا
در دیکشنری وجود دارد؟ 'dead' آیا True
```

## جست و جو "dead"

No description has been provided for this image

No description has been provided for this image

# پویا بودن سایز dict

وقتی دیکشنری تقریباً پر شود (مثلًا به ضرب بار مشخصی برسد)، چه مشکلی پیش می‌آید؟

\*\*پاسخ:\*\* با افزایش تعداد عناصر در جدول هش، تعداد برخوردها افزایش می‌یابد. این باعث می‌شود که عملیات‌های درج، جستجو و حذف کندتر شوند، زیرا الگوریتم مجبور است خانه‌های بیشتری را در دنباله کاوش بررسی کند. در نهایت، عملکرد از  $O(1)$  به  $O(n)$  نزدیک می‌شود.

\*\*برای حل این مشکل چه راه حلی پیشنهاد می‌کنید؟

\*\*پاسخ:\*\* راه حل این است که وقتی دیکشنری به ضرب بار مشخصی رسید، اندازه آن را افزایش دهیم (ممکن‌آلاً دو برابر یا بیشتر) و تمام عناصر موجود را به جدول هش جدید (با اندازه بزرگتر) دوباره هش کنیم. این فرآیند \*تغییر اندازه (Resizing)\* یا \*بازسازی (Resizing)\* نامیده می‌شود.

دیکشنری در پایتون خود مانند یک آرایه پویا عمل می‌کند.

در پایتون، وقتی ضرب بار (نسبت تعداد عناصر به ظرفیت جدول) به حدود  $2/3$  (یا  $0.66$ ) می‌رسد، یک دیکشنری جدید ساخته می‌شود و تمام اعضای دیکشنری داخل دیکشنری جدید ریخته می‌شوند.

سایز دیکشنری جدید در پایتون به صورت زیر است:

- (برای دیکشنری‌های کوچک، ظرفیت  $4$  برابر می‌شود.)
- (برای دیکشنری‌های بزرگتر، ظرفیت  $2$  برابر می‌شود.)

این کار برای صرفه‌جویی در حافظه در مقادیر بالا است و همچنین به حفظ عملکرد  $O(1)$  سرشکن کمک می‌کند.

\*\*پاسخ:\*\* این اثبات با استفاده از \*\*تحلیل سرشکن (Amortized Analysis)\*\* انجام می‌شود. هزینه یک عملیات تغییر اندازه ( $O(n)$ ) است (چون باید  $n$  عنصر را دوباره هش و درج کرد). اما این عملیات فقط پس از تعداد زیادی (مثلًا  $3/n$ ) عملیات درج اتفاق می‌افتد. بنابراین، هزینه  $O(n)$  به صورت سرشکن بین  $3/n$  عملیات قبلی تقسیم می‌شود، که منجر به هزینه سرشکن  $O(1)$  برای هر عملیات درج می‌شود.

فایل words یک دیکشنری حاوی تمام کلمات انگلیسی است.  
از آن برای اضافه کردن کلمات به dict استفاده می‌کنیم.

```
In [25]: # وجود دارد 'files/words' فرض می‌شود فایل) بار کردن فایل کلمات
wordfile = open('files/words')
text = wordfile.read() # خواندن کل محتوای فایل
# تقسیم متن به کلمات، فیلتر کردن کلمات کوچک و با طول کمتر از 6
words = [w for w in text.split() if w == w.lower() and len(w) < 6]

# چاپ 5 کلمه اول برای بررسی
print("5 کلمه اول:", words[:5])

# ایجاد یک دیکشنری از 5 کلمه اول
d = dict.fromkeys(words[:5])
# در این مرحله، دیکشنری ممکن است 2/3 بر باشد و در آستانه تغییر اندازه باشد.
# در این مرحله ممکن است حدود 40% باشد. # نرخ برخورد (Collision Rate)
print(d, "وضعیت دیکشنری پس از 5 کلمه")
# این خط فقط برای نمایش نرخ برخورد است # ("نرخ برخورد: 40")
# print(f"نرخ برخورد: {d['a']} - در آستانه تغییر اندازه")
# print(f"وضعیت: {d['a']} - در 2/3 بر - در آستانه تغییر اندازه")

# کلمه اول 5
{'a': None, 'abaci': None, 'aback': None, 'abaft': None, 'abase': None}
```

No description has been provided for this image

```
In [26]: d["izadi"] = None # درج کلمه 'izadi'
# پس از این درج، دیکشنری تغییر اندازه می‌دهد (ظرفیت 4 برابر می‌شود به 32)
# نرخ برخورد به 0% کاهش می‌یابد
print("izadi:", d)
# print(f"وضعیت دیکشنری پس از درج")
# print(f"نرخ برخورد به 32, نرخ برخورد به 0% کاهش می‌یابد")

# کلمه اول 'izadi': {'a': None, 'abaci': None, 'aback': None, 'abaft': None, 'abase': None, 'izadi': None}
```

No description has been provided for this image

```
In [27]: d = dict.fromkeys(words[:21]) # ایجاد دیکشنری از 21 کلمه اول
# در این مرحله، دیکشنری دوباره 2/3 بر است و نرخ برخورد حدود 29% است
# print(d, "وضعیت دیکشنری پس از 21 کلمه")
# print(f"نرخ برخورد 29%")
# کلمه اول 21
```

وضعیت دیکشنری پس از درج {"a": None, "abaci": None, "aback": None, "abaft": None, "abase": None, "abash": None, "abate": None, "abbey": None, "abbot": None, "abeam": None, "abed": None, "abet": None, "abets": None, "abhor": None, "abide": None, "able": None, "abler": None, "ably": None, "abode": None, "abort": None, "about": None}

No description has been provided for this image

```
In [28]: d['abode'] = None # درج کلمه 'abode'
# پس از این درج، دیکشنری تغییر اندازه می‌دهد (ظرفیت 4 برابر می‌شود به 128)
# نرخ برخورد به 9% کاهش می‌یابد
print("abode:", d)
# print(f"وضعیت دیکشنری پس از درج")
# print(f"نرخ برخورد به 128, نرخ برخورد به 9% کاهش می‌یابد")

# کلمه اول 128
```

وضعیت دیکشنری پس از درج {"a": None, "abaci": None, "aback": None, "abaft": None, "abase": None, "abash": None, "abate": None, "abbey": None, "abbot": None, "abeam": None, "abed": None, "abet": None, "abets": None, "abhor": None, "abide": None, "able": None, "abler": None, "ably": None, "abode": None, "abort": None, "about": None, "abode": None}

No description has been provided for this image

```
In [29]: d = dict.fromkeys(words[:85]) # ایجاد دیکشنری از 85 کلمه اول
# دوباره 2/3 بر است و نرخ برخورد 33% است
# print(d, "وضعیت دیکشنری پس از 85 کلمه")
# print(f"نرخ برخورد 33%")
```

{'a': None, 'abaci': None, 'aback': None, 'abaft': None, 'abase': None, 'abash': None, 'abate': None, 'abbey': None, 'abbot': None, 'abeam': None, 'abed': None, 'abet': None, 'abets': None, 'abhor': None, 'abide': None, 'able': None, 'abler': None, 'ably': None, 'abode': None, 'abort': None, 'about': None, 'above': None, 'abuse': None, 'abut': None, 'abuts': None, 'abuzz': None, 'abyss': None, 'ace': None, "ace's": None, 'acd': None, 'aces': None, 'ache': None, 'ached': None, 'aches': None, 'achoo': None, 'achy': None, 'acid': None, 'acids': None, 'acing': None, 'acme': None, 'acmes': None, 'acne': None, 'acorn': None, 'acre': None, 'acres': None, 'acrid': None, 'act': None, "act's": None, 'acted': None, 'actor': None, 'acts': None, 'acute': None, 'ad': None, "ad's": None, 'adage': None, 'adapt': None, 'add': None, 'added': None, 'adder': None, 'addle': None, 'adds': None, 'adept': None, 'adieu': None, 'adman': None, 'admen': None, 'admit': None, 'ado': None, "ado's": None, 'adobe': None, 'adopt': None, 'adore': None, 'adorn': None, 'ads': None, 'adult': None, 'adz': None, "adz's": None, 'adze': None, 'adzes': None, 'aegis': None, 'aeon': None, 'aeons': None, 'aerie': None, 'aery': None, 'afar': None, 'affix': None}

No description has been provided for this image

## تمرین

اسم و شماره دانشجویی خودتان را به دیکشنری اضافه کنید. سپس آن را جست و جو کرده و حذف نمایید.

```
In [30]: #یجاد یک دیکشنری جدید برای تمرین
my_info_dict = {}

#اضافه کردن اسم و شماره دانشجویی
my_name = "Hamid Namjoo"
my_student_id = "9812345" # <span style="direction:rtl;"></span>

my_info_dict["name"] = my_name
my_info_dict["student_id"] = my_student_id

print(f"دیکشنری اطلاعات من {my_info_dict}")

#جستجو در دیکشنری
print(f"در دیکشنری وجود دارد؟ آیا {'name' in my_info_dict}")
print(f"اسم من: {my_info_dict['name']}")
print(f"در دیکشنری وجود دارد؟ آیا {'student_id' in my_info_dict}")
print(f"شماره دانشجویی من: {my_info_dict['student_id']}")

#حذف از دیکشنری
del my_info_dict["name"]
print(f"دیکشنری پس از حذف اسم {my_info_dict}")

print(f"هنوز در دیکشنری وجود دارد؟ آیا {'name' in my_info_dict}")

{'name': 'Hamid Namjoo', 'student_id': '9812345'} در دیکشنری وجود دارد؟ آیا 'name' در دیکشنری وجود دارد؟ آیا
اسم من: Hamid Namjoo در دیکشنری وجود دارد؟ آیا 'student_id' در دیکشنری وجود دارد؟ آیا
شماره دانشجویی من: 9812345 دیکشنری پس از حذف اسم: {'student_id': '9812345'}
هنوز در دیکشنری وجود دارد؟ آیا False
```

نمودار زیر را برای زمان دسترسی به اعضاء میبینید.

با افزایش اندازه دیکشنری، تعداد برخوردهای میانگین از یک ضرب ب ثابتی بیشتر نمیشود. این به دلیل استراتژی تغییر اندازه پویا است که در پایتون استفاده میشود.

No description has been provided for this image

## اهمیت تابع درهم سازی

\*پاسخ:\*\* اگر تابع درهمسازی ضعیف باشد، حتی با وجود استراتژی‌های مقابله با برخورد، تعداد برخوردها به شدت افزایش می‌یابد. این باعث می‌شود که جدول هش به جای عملکرد  $O(1)$ ، به سمت  $O(n)$  (مانند لیست پیوندی) میل کند و کارایی آن به شدت کاهش یابد.

در پایتون، اعداد صحیح به طور خیلی ساده هش می‌شوند به طوری که `hash(x)` (برای اعداد صحیح کوچک).

$$\text{hash}(x) = x$$

و با دانستن این اطلاعات می‌توان تعداد زیادی برخورد تولید کرد.

In [31]: `print("5 هش عدد:", hash(5))`

```
# مثال ایجاد برخورد برای اعداد صحیح با استفاده از خاصیت hash(x) = x
# مثلاً 8 باشد، اعدادی که با قیمانده تقسیم‌شان بر 8 یکسان است، برخورد می‌کنند اگر
# 3 % 8 = 3
# 11 % 8 = 3
# 19 % 8 = 3
threes = {3: 1, 3 + 8: 2, 3 + 16: 3, 3 + 24: 4, 3 + 32: 5}
print("دیکشنری با کلیدهایی که هش یکسان دارند:", threes)
```

5 هش عدد: 5  
{3: 1, 11: 3, 19: 3, 27: 4, 35: 5} دیکشنری با کلیدهایی که هش یکسان دارند:

No description has been provided for this image

در جاوا (نسخه‌های قدیمی‌تر، مثلاً 1.1) برای صرفه‌جویی در زمان، از تابع زیر برای هش کردن رشته‌ها استفاده می‌شد:

```
public int hashCode()
{
    int hash = 0;
    int skip = Math.max(1, length() / 8);
    // skip every 8 characters
    for (int i = 0; i < length(); i += skip)
        hash = s[i] + (37 * hash);
    return hash;
}
```

با این کار (استفاده از `skip` در تابع هش)، رشته‌های زیر هش‌های یکسانی خواهند داشت:

No description has been provided for this image

اگر از روش زنجیره‌سازی استفاده شده باشد، همه‌ی این رشته‌ها در یک زنجیره ذخیره می‌شوند.

No description has been provided for this image

با دانستن تابع هش (که در بسیاری از زبان‌ها و سیستم‌ها عمومی است)، می‌توان حمله‌های زیادی به سرورها کرد (معمولاً حمله‌های \*\*Denial of Service (DoS)\*\*). در این حملات، مهاجم با ارسال تعداد زیادی کلید که همگی به یک سطل هش می‌شوند، عملکرد سیستم را به شدت کاهش می‌دهد.

مثلاً در کرنل 2.4.20 Linux، اگر نام فایل‌ها به صورت خاصی ذخیره می‌شوند، سیستم crash می‌کرد، زیرا تابع هش نام فایل‌ها ضعیف بود و مهاجم می‌توانست با ایجاد نام‌های خاص، باعث برخوردهای زیاد و از کار افتادن سیستم شود.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل نهم: توابع درهمسازی (قسمت سوم)

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

#### • مقدمه

• مثال: شبکه‌ی اجتماعی کتابخوان‌ها

• درهمسازی میانی

• شبیه‌ترین کاربر (نزدیک‌ترین همسایه)

• درهمسازی حساس به شباهت

• شبیه‌ترین کتاب!

## مقدمه

روش‌های درهمسازی (Hashing) که در دو بخش قبلی بررسی کردیم، عمدتاً برای کاهش ابعاد داده به منظور مقایسه‌ی راحت‌تر و سریع‌تر آن‌ها به وجود آمده‌اند. در آن بخش‌ها، توابع درهمسازی را بررسی کردیم که به احتمال بالایی برای هر دو داده‌ی متفاوت، هش متفاوتی تولید می‌کردند و به عبارتی سعی می‌کردیم \*تصادم (Collision) را تا بیشترین میزان کم کنیم. این توابع در کاربردهایی مثل تشخیص فایل‌های دقیقاً یکسان، صفحه‌های وب دقیقاً یکسان، یا ذخیره‌سازی امن رمزها کارآمد هستند.

در این بخش، نوع دیگری از توابع درهمسازی را بررسی می‌کنیم که برای \*\*داده‌های نزدیک\*\* (نسبت به یک تعریف مشخص از فاصله یا شباهت)، به احتمال بالایی هش یکسان تولید می‌کنند. این توابع در کاربردهایی مثل پیدا کردن صفحه‌های وب با محتوای نزدیک به هم، پیدا کردن صاحب اثر انگشت، یا پیشنهاد دادن دوست به کاربران در شبکه‌های اجتماعی کارآمد هستند! این دسته از توابع به \*درهمسازی حساس به شباهت معروف هستند. (Locality Sensitive Hashing - LSH)

## مثال: شبکه‌ی اجتماعی کتابخوان‌ها

فرض کنید یک شبکه‌ی اجتماعی برای افراد کتابخوان طراحی کرده‌اید. هر کسی در این شبکه می‌تواند کتاب‌هایی که مطالعه کرده است را مشخص کند و با افرادی که کتاب‌های مشابهی مطالعه کرده‌اند آشنا شود.

برای سادگی فرض کنید می‌خواهیم بخشی به سیستم اضافه کنیم که \*\*شبیه‌ترین کاربر\*\* را به ازای هر کاربر داده شده پیدا می‌کند.

## معیار شباهت

اطلاعات مربوط به هر کاربر را می‌توان با یک بردار  $m$  بعدی صفر و ۱ ذخیره کرد و نمایش داد. به این صورت که اگر کاربر، کتاب  $n$ -ام را مطالعه کرده باشد، خانه‌ی مربوط به آن ۱ و در غیر این صورت صفر باشد.

می‌توانیم تعاریف مختلفی از شباهت یا فاصله را بر اساس کاربرد مورد نظرمان انتخاب کنیم. مثلاً می‌توانیم شباهت دو نفر را تعداد کتاب‌های مشترکی که مطالعه کرده‌اند در نظر بگیریم.

\*\*ایراد این تعریف در عمل چیست؟\*

\*\*پاسخ:\*\* ایراد این تعریف است که تعداد کتاب‌های مشترک به تنها یک معیار خوبی برای شباهت نیست. مثلاً، دو کاربر ممکن است هر دو فقط یک کتاب مشترک داشته باشند، اما یکی از آن‌ها 1000 کتاب خوانده باشد و دیگری فقط 2 کتاب. در این حالت، شباهت واقعی آن‌ها کم است. برای رفع این مشکل، باید تعداد کل کتاب‌های خوانده شده توسط هر دو کاربر را نیز در نظر گرفت.

یکی از تعاریفی که در عمل خوب کار می‌کند، \*\*شاخص Jaccard\*\* است که شباهت دو مجموعه  $A$  و  $B$  را به صورت زیر تعریف می‌کند:

$$\frac{|A \cap B|}{|A \cup B|}$$

در این مثال، شباهت دو کاربر از دید شاخص Jaccard برابر با تعداد کتاب‌های مشترکی که مطالعه کردند تقسیم بر تعداد کل کتاب‌هایی که هر دو کاربر (بدون تکرار) خوانده‌اند است.

تمرین: برای دو زیرمجموعه‌ی  $m$  عضوی تصادفی از اعداد 1 تا  $n$ ، امید ریاضی شاخص Jaccard را محاسبه کنید.

```
In [1]: n = 4 # تعداد کاربران
m = 13 # تعداد کتاب‌ها (ابعاد بردار)
```

داده‌های نمونه: هر سطر یک کاربر و هر ستون یک کتاب است (1 یعنی خوانده، 0 یعنی نخوانده).

```
data = [[1, 1, 0, 1, 0, 0, 0, 0, 1, 0], # کاربر 0
        [0, 1, 0, 0, 0, 0, 0, 0, 1, 0], # کاربر 1
        [1, 0, 1, 0, 0, 1, 1, 0, 1, 0], # کاربر 2
        [0, 0, 0, 1, 1, 0, 0, 1, 0, 1], # کاربر 3]
```

```
def jaccard(A, B):
    """
    محاسبه می‌کند  $A$  و  $B$  را بین دو بردار دودویی Jaccard این تابع شاخص
```

Args:

بردار دودویی (لیست 0 و 1) برای مجموعه اول:  $A$  (list).
 بردار دودویی (لیست 0 و 1) برای مجموعه دوم:  $B$  (list).

Returns:

float: مقادیر شاخص Jaccard.

```
    """
    هر دو 1 باشند  $B$  و  $A$ ) تعداد عناصر مشترک # (0.0
    union_count = 0.0 # (حداقل یکی 1 باشد  $B$  یا  $A$ ) تعداد کل عناصر در اجتماع #
```

```
for i in range(0, len(A)):
    intersection_count = intersection_count + (A[i] and B[i]) #  $A[i] == 1$  و  $B[i] == 1$ 
    union_count = union_count + (A[i] or B[i]) #  $A[i] == 1$  یا  $B[i] == 1$ 
```

حلوگیری از تقسیم بر صفر اگر هر دو مجموعه خالی باشند # جلوگیری از تقسیم بر صفر اگر هر دو مجموعه خالی باشند # در این حالت معمولاً شباهت 100% در نظر گرفته می‌شود.

```
return intersection_count / union_count
```

```
# بین کاربران Jaccard مثال محاسبه شباهت
print("بین کاربران 0 و کاربر 2:", jaccard(data[0], data[2])) # بین کاربر 0 و کاربر 2 Jaccard شباهت
print("بین کاربر 0 و کاربر 1:", jaccard(data[0], data[1])) # بین کاربر 0 و کاربر 1 Jaccard شباهت
```

بین کاربر 0 و کاربر 2: 0.25
 بین کاربر 0 و کاربر 1: 0.4

## چالش‌ها

ذخیره کردن اطلاعات کاربران در داده‌ساختارهای ساده مثل آرایه‌ی صفر و 1 که در بخش قبل به آن اشاره کردیم، در عمل ممکن است بسیار پرهزینه باشد.

برای مثال اگر 1 میلیون کاربر و 10 هزار کتاب در سیستم داشته باشیم، حجم اطلاعات نمایش جدولی حدود  $10^{10} \times 10^4 = 10^{10}$  بیت (حدود 1.25 گیگابایت) می‌شود که بارگزاری آن به صورت یکجا در حافظه اصلی کامپیوترهای معمولی (یا حتی کامپیوترهای بسیار قدرتمند) ممکن نیست. در واقع، در این مثال، اگر هر بیت را به عنوان یک بایت ذخیره کنیم، حجم اطلاعات به 10 ترابایت می‌رسد که بسیار زیاد است.

جدای از مشکل حافظه، محاسبه‌ی شباهت Jaccard دو کاربر از مرتبه  $O(m)$  (تعداد کل کتاب‌ها) زمان لازم دارد و به همین ترتیب پیدا کردن پرشباخت‌ترین کاربر برای هر کاربر مشخص شده (که نیاز به مقایسه با تمام کاربران دیگر دارد) کار بسیار زمان‌بری است و از مرتبه  $O(nm)$  زمان می‌برد.

با توجه به این چالش‌ها، باید یک داده‌ساختار مناسب برای پیدا کردن پرشباخت‌ترین کاربر به ازای هر کاربر داده شده طراحی کنیم، که هم فضای کمتری برای ذخیره اطلاعات و هم زمان کمتری برای پاسخ دادن به درخواست‌ها نیاز داشته باشد.

## جواب تقریبی

اما در اکثر کاربردها، جواب‌های تقریبی هم کافی هستند. مثلاً اگر به جای معرفی کردن شبیه‌ترین کاربر، یکی از کاربرهایی که شباهت زیادی دارد را معرفی کنیم، احتمالاً کاربرها ناراضی نمی‌شوند.

به نظر شما صورت تقریبی مسئله را چگونه طرح کنیم؟

\*\*پاسخ:\*\* صورت تقریبی مسئله می‌تواند این باشد که "لیستی از  $K$  کاربر که بیشترین شباهت را با کاربر مورد نظر دارند (به جای فقط شبیه‌ترین کاربر) را برگردانید" یا "کاربرانی را پیدا کنید که شباهت آن‌ها از یک آستانه مشخص بیشتر است".

## درهمسازی میانی (MinHash)

ایده‌ی اصلی درهمسازی میانی (MinHash) استفاده از تعدادی تابع مستقل درهمساز مثل  $h_i(x)$  است که به هر مجموعه یک عدد صحیح نسبت می‌دهد، به طوری که به ازای هر  $A \neq B$  احتمال تساوی  $h_i(A) = h_i(B)$  برابر با شباهت  $A$  و  $B$  در شاخص Jaccard باشد.

اگر تعداد کافی از این تابع‌های درهمساز داشته باشیم، می‌توانیم به جای هر مجموعه  $A$ ، مقادیر  $(A)_i$ ‌ها را نگهداری کنیم. با توجه به قانون اعداد بزرگ، با افزایش تعداد  $(A)_i$ ‌ها برای هر دو مجموعه مختلف، نسبت تعداد هش‌های برابر به کل تعداد هش‌ها بسیار نزدیک به شباهت Jaccard آن‌ها خواهد بود.

## درهمسازی میانی در مثال شبکه اجتماعی کتابخوان‌ها

فرض کنید کتاب‌ها را با شماره‌های ۱ تا  $m$  شماره‌گذاری کرده‌ایم. مجموعه‌ی کتاب‌هایی که یک کاربر مطالعه کرده است را  $A$  در نظر بگیرید.

$$h(A) = \min_{x \in A} x$$

یعنی کمترین شماره بین شماره‌ی کتاب‌هایی که این کاربر مطالعه کرده است.

تمرین: دو مجموعه‌ی مختلف  $A$  و  $B$  را در نظر بگیرید و فرض کنید کتاب‌ها به صورت تصادفی شماره‌گذاری شده‌اند (یعنی جایگشت کتاب‌ها با احتمال برابر از بین همه‌ی جایگشت‌های ممکن انتخاب شده است).

$$\text{ثابت کنید احتمال } h(A) = h(B) \text{ برابر است با } \frac{|A \cap B|}{|A \cup B|}.$$

In [2]: `def min_hash(A_vector):` (در اینجا بردار دودویی است  $A$ ) تغییر نام پارامتر برای وضوح #

را برای یک بردار دودویی (مجموعه) محاسبه می‌کند MinHash این تابع MinHash برابر با اندیس اولین '1' در بردار است.

Args:

`A_vector (list):` بردار دودویی (لیست ۰ و ۱) که نشان‌دهنده مجموعه است.

Returns:

اندیس اولین '1' (کوچکترین عنصر) یا -1 اگر مجموعه خالی باشد.

`for i in range(0, len(A_vector)):`

`if A_vector[i] == 1:`

`return i # 1'ی`

`return -1 # اگر هیچ '1'ی وجود نداشت (مجموعه خالی بود)`

# که قبلاً تعریف شده بود استفاده از داده‌های

# `data = [[1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0], ...]`

`print("MinHash 0:", min_hash(data[0]))`

`print("MinHash 1:", min_hash(data[1]))`

`print("MinHash 2:", min_hash(data[2]))`

کاربر 0 :0

کاربر 1 :1

کاربر 2 :2

اگر کتاب‌ها را با یک جایگشت تصادفی جدید شماره‌گذاری کنیم، همان تابع  $h(A)$  تبدیل به یک تابع مستقل جدید می‌شود که شرایط درهمسازی میانی را دارد. این به این معنی است که می‌توانیم چندین تابع MinHash مستقل ایجاد کنیم، بدون اینکه نیاز به تعریف توابع هش پیچیده جدید داشته باشیم.

پس برای محاسبه هش اطلاعات کاربران، می‌توانیم در ابتدا تعدادی جایگشت تصادفی انتخاب کنیم و نسبت به هر کدام تابع  $h(A)$  را محاسبه کنیم. این مقادیر هش شده (که به آن‌ها  $\text{MinHash}^*$  یا  $\text{MinHash Signature}^{**}$  می‌گویند) سپس برای تخمین شباهت Jaccard بین دو مجموعه استفاده می‌شوند.

In [3]:

```
import numpy برای استفاده از # numpy.random.permutation

def approximated_jaccard(hashed_A, hashed_B):
    """
    محاسبه می‌کند MinHash را به صورت تقریبی بر اساس امضاهای Jaccard این تابع شباهت
    (تعداد کل هش‌ها) / (تعداد هش‌های یکسان) = Jaccard شباهت تقریبی
    """

    Args:
        hashed_A (list): برای مجموعه (امضا) MinHash لیست مقادیر A.
        hashed_B (list): برای مجموعه (امضا) MinHash لیست مقادیر B.

    Returns:
        float: مقدار تقریبی شباهت Jaccard.
    """
    matches = 0.0
    # بیمامیش روی امضاهای MinHash
    for i in range(0, len(hashed_A)):
        if hashed_A[i] == hashed_B[i]:
            matches += 1 # اگر مقادیر هش یکسان بودند، شمارنده را افزایش بده

    if len(hashed_A) == 0: # جلوگیری از تقسیم بر صفر
        return 1.0

    return matches / len(hashed_A)

# از مثال قبلی (تعداد کاربران و کتاب‌ها)
# n = 4
# m = 13

# P: می‌ستقل ایجاد می‌کند MinHash لیستی از جایگشت‌های تصادفی. هر جایگشت یک تابع
# numpy.random.permutation(m) یک آرایه از 0 تا m-1 را به صورت تصادفی جایگشت می‌دهد
num_permutations = 5 # تعداد توابع MinHash
P = [numpy.random.permutation(m) for _ in range(num_permutations)]

# برای هر کاربر MinHash لیستی برای نگهداری امضاهای
hashed_data = [[[] for _ in range(m)] for _ in range(n)]

# برای هر کاربر محاسبه امضای
for i in range(0, n):
    # بیمامیش روی کاربران # این فعلی کاربر امضای MinHash
    res = [] # برای هر جایگشت
    for j in range(0, len(P)): # هر جایگشت
        # برای هر تابع # روی بردار داده کاربر P[j] اعمال جایگشت
        # و محاسبه data[i] روی بردار داده کاربر P[j] MinHash
        # min_hash(data[i], P[j])
        # تعریف کنیم min_hash_with_permutation برای این کار، باید یک تابع
        # بگیرد MinHash که بردار را بر اساس جایگشت ورودی، "بازاریابی" کند و سپس
        # باید بر روی بردار جایگشت داده شده کار کند اصلاح: تابع
        # min_hash(data[i], P[j])
        # min_hash_with_permutation(original_vector, permutation_array)
        # پیدا کند این اعمال پس از original_vector را در
        # موجود برای سادگی و استفاده از
        # یک بردار جدید بر اساس جایگشت ایجاد می‌کنیم
        permuted_vector = [data[i][p_val] for p_val in P[j]]
        res.append(min_hash(permuted_vector))
    hashed_data[i].append(res)

print('کاربران MinHash امضاهای:', hashed_data)

sum_error = 0.0 # مجموع خطاهای مطلق Jaccard
# محاسبه و چاپ خطای تقریبی
for i in range(0, n):
    for j in range(0, n):
        # واقعی محاسبه شباهت Jaccard
        actual_jaccard = jaccard(data[i], data[j])
        # تقریبی از روی هشها محاسبه شباهت
        approx_jaccard = approximated_jaccard(hashed_data[i], hashed_data[j])

        error = abs(actual_jaccard - approx_jaccard)
        print(f"کاربر {i} و {j}: Jaccard واقعی={actual_jaccard:.2f}, تقریبی={approx_jaccard:.2f}, خطای={error:.2f}")
        sum_error += error

print('خطای میانگین خطا 2.% = % (sum_error / (n * n)))
```

کاربران: [[0, 4, 0, 2, 5, [1, 0, 0, 8, 0, 5], [5, 5, 3, 1, 6], [3, 1, 3, 1, 4]]] امضاهای MinHash  
 واقعی=0.00=1.00، تقریبی=1.00، خط=0: کاربر 0 و 0  
 واقعی=0.40=0، تقریبی=0.40، خط=0: کاربر 0 و 1  
 واقعی=0.25=0، تقریبی=0.00، خط=0: کاربر 0 و 2  
 واقعی=0.11=0، تقریبی=0.00، خط=0: کاربر 0 و 3  
 واقعی=0.40=0، تقریبی=0.40، خط=0: کاربر 1 و 0  
 واقعی=1.00=1، تقریبی=1.00، خط=0: کاربر 1 و 1  
 واقعی=0.00=0، تقریبی=0.00، خط=0: کاربر 1 و 2  
 واقعی=0.00=0، تقریبی=0.00، خط=0: کاربر 1 و 3  
 واقعی=0.25=0، تقریبی=0.00، خط=0: کاربر 2 و 0  
 واقعی=0.00=0، تقریبی=0.00، خط=0: کاربر 2 و 1  
 واقعی=1.00=0، تقریبی=1.00، خط=0: کاربر 2 و 2  
 واقعی=0.00=0، تقریبی=0.00، خط=0: کاربر 2 و 3  
 واقعی=0.11=0، تقریبی=0.00، خط=0: کاربر 3 و 0  
 واقعی=0.00=0، تقریبی=0.00، خط=0: کاربر 3 و 1  
 واقعی=0.00=0، تقریبی=0.00، خط=0: کاربر 3 و 2  
 واقعی=1.00=0، تقریبی=1.00، خط=0: کاربر 3 و 3  
 میانگین خط = 0.05

روشن است که هرچه تعداد جایگشت‌های تصادفی بیشتری انتخاب کنیم، دقت تقریب بیشتر می‌شود. یکی از سوالات مهم این است که برای رسیدن به یک دقت معین، چه تعداد جایگشت تصادفی کافی است؟

فرض کنید ۵۰۰ کاربر و ۱۰۰۰ کتاب در سیستم داریم. سعی کنید با تولید داده‌های تصادفی (مثلاً می‌توانید فرض کنید هر کاربر هر کتاب را به احتمال ۰.۰۰۵ مطالعه کرده‌است) تعداد جایگشت‌های تصادفی‌ای را پیدا کنید که میانگین خط با در نظر گرفتن داده‌های هش شده به جای داده‌های اصلی حداقل ۰.۰۵ شود.

In [7]:

```
import random

n_users = 500 # تعداد کاربران
m_books = 1000 # تعداد کتابها

random_data = [] # لیست برای نگهداری داده‌های تصادفی کاربران
for user in range(0, n_users):
    user_books = [] # لیست کتاب‌های خوانده شده توسط کاربر فعلی
    for book in range(0, m_books):
        r = random.uniform(0, 1) # تولید یک عدد تصادفی بین 0 و 1
        if r <= 0.005: # با احتمال ۰.۰۰۵، کتاب خوانده شده است
            user_books.append(1)
        else:
            user_books.append(0)
    random_data.append(user_books)

# TODO: complete this code guys :)
```

```
min_permutations_needed = 10
current_avg_error = float('inf')
while current_avg_error > 0.05:
    min_permutations_needed += 10
    P_new = [numpy.random.permutation(m_books) for _ in range(min_permutations_needed)]
    hashed_data_new = []
    for user_idx in range(n_users):
        res_user = []
        for perm_idx in range(min_permutations_needed):
            permuted_vector = [random_data[user_idx][p_val] for p_val in P_new[perm_idx]]
            res_user.append(min_hash(permuted_vector))
        hashed_data_new.append(res_user)

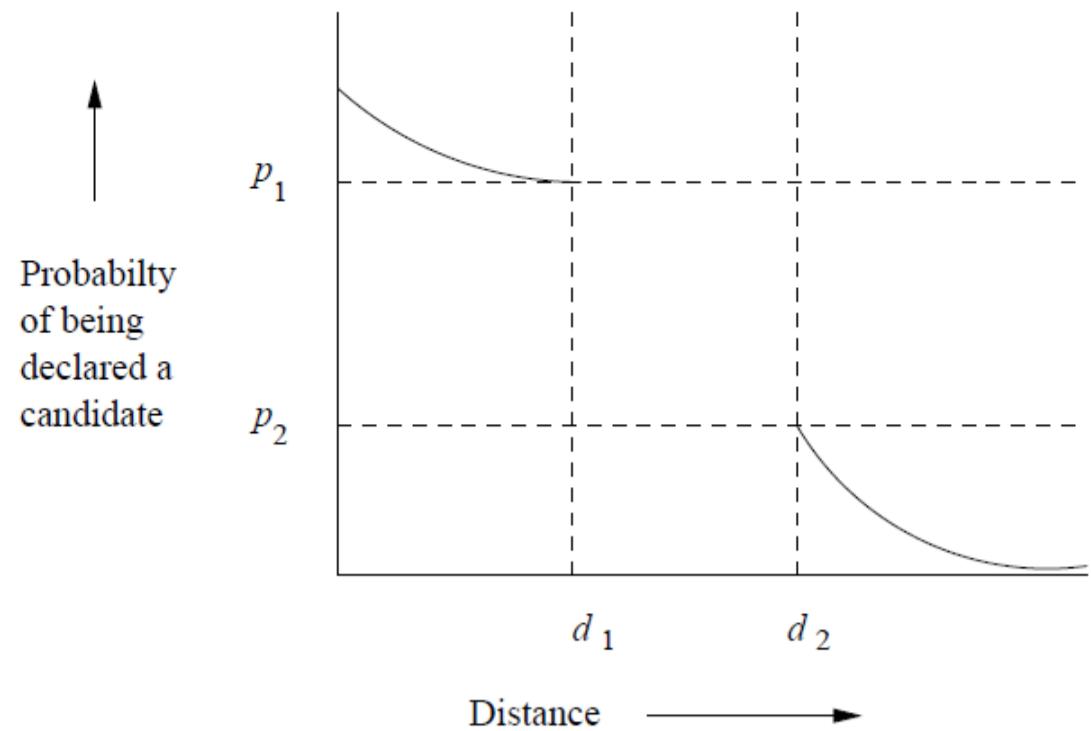
    current_sum_error = 0.0
    for i in range(n_users):
        for j in range(n_users):
            مقایسه با خودش معنی ندارد # continue
            actual_jaccard = jaccard(random_data[i], random_data[j])
            approx_jaccard = approximated_jaccard(hashed_data_new[i], hashed_data_new[j])
            current_sum_error += abs(actual_jaccard - approx_jaccard)
    current_avg_error = current_sum_error / (n_users * (n_users - 1))
    print(f"تعداد جایگشت‌های میانگین خط: {min_permutations_needed}, تعداد جایگشت‌های میانگین خط: {current_avg_error:.4f}")
print("تعداد جایگشت‌های لازم برای خطی حداقل ۰.۰۵: {min_permutations_needed}")
```

تعداد جایگشت: 20، میانگین خط: 0.0013  
 تعداد جایگشت‌های لازم برای خطی حداقل 0.05

## درهمسازی حساس به شباهت

درهمسازی میانی یک نمونه از درهمسازی حساس به شباهت است. به طور کلی یک درهمسازی حساس به شباهت مجموعه‌ای از توابع مستقل از هم است که با چهار پارامتر  $d_1 < d_2$  و  $p_1 > p_2$  تعریف می‌شود و هر کدام از توابع باید خواص زیر را برای هر دو نقطه‌ی  $x$  و  $y$  داشته باشد:

۱. اگر فاصله‌ی  $x$  و  $y$  کمتر از  $d_1$  است احتمال یکی شدن hash این دو نقطه باید دست کم  $p_1$  باشد.
۲. اگر فاصله‌ی  $x$  و  $y$  بیشتر از  $d_2$  است احتمال یکی شدن hash این دو نقطه باید دست بالا  $p_2$  باشد.

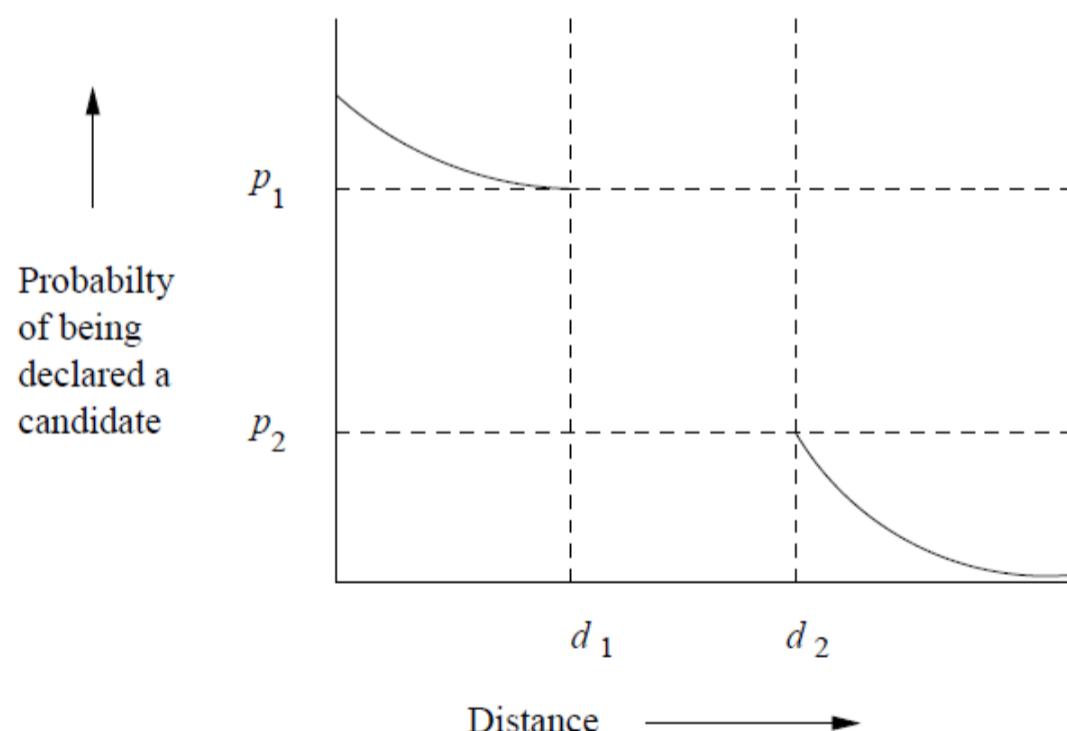


همانطور که در مثال درهمسازی میانی دیدیم، با توجه به استقلال این توابع می‌توان با ترکیب آنها به دقت مطلوب رسید. می‌توانید این چهار پارامتر را برای روش درهمسازی میانی تعیین کنید؟

## درهمسازی حساس به شباهت (Locality Sensitive Hashing - LSH)

درهمسازی میانی (MinHash) یک نمونه از \*\*درهمسازی حساس به شباهت (LSH)\*\* است. به طور کلی، یک درهمسازی حساس به شباهت، مجموعه‌ای از توابع مستقل از هم است که با چهار پارامتر  $d_1 < d_2$  و  $p_1 > p_2$  تعریف می‌شود و هر کدام از توابع باید خواص زیر را برای هر دو نقطه‌ی  $x$  و  $y$  داشته باشد:

1. اگر فاصله‌ی  $x$  و  $y$  کمتر از  $d_1$  است، احتمال یکی شدن هش این دو نقطه باید دست کم  $p_1$  باشد.
2. اگر فاصله‌ی  $x$  و  $y$  بیشتر از  $d_2$  است، احتمال یکی شدن هش این دو نقطه باید دست بالا  $p_2$  باشد.



همانطور که در مثال درهمسازی میانی دیدیم، با توجه به استقلال این توابع، می‌توان با ترکیب آنها به دقت مطلوب رسید. \*

- \*\*فاصله (Distance):\*\* در LSH، معمولاً فاصله را تهابش – 1 تعریف می‌کنند. پس، فاصله Jaccard
- برابر  $Jaccard Similarity = 1 - \frac{d}{n}$  است.
- $(h(A) = h(B))$  (احتمال بخورد برای نقاط نزدیک): در MinHash، احتمال بخورد دو امضا (یعنی  $p_1$ ) دقيقاً برابر با شباهت Jaccard آنها است. پس  $p_1 = Jaccard Similarity$  وقتی فاصله کمتر از  $d_1$  است).
- $(h(A) \neq h(B))$  (احتمال بخورد برای نقاط دور): وقتی فاصله بیشتر از  $d_2$  است).

•  $d_1$  و  $d_2$ : این‌ها آستانه‌های فاصله هستند. برای MinHash، می‌توانیم  $d_1$  و  $d_2$  را به عنوان آستانه‌هایی برای فاصله Jaccard تعریف کنیم که در آن‌ها احتمال برخورد به ترتیب  $p_1$  و  $p_2$  باشد.

In [4]:  
# TODO:  
# Example needed here.

## شبیه‌ترین کاربر (نزدیک‌ترین همسایه)

با اینکه توانستیم با استفاده از روش درهمسازی میانی، فضای مورد نیاز برای ذخیره‌سازی داده‌ها و همچنین زمان مقایسه‌ی دو کاربر را کاهش دهیم (از  $O(m)$  به  $O(K)$  که  $K$  تعداد توابع MinHash است)، ولی هنوز پیدا کردن نزدیک‌ترین کاربر به یک کاربر دلخواه دست کم از مرتبه  $O(n)$  است (چون باید با همه  $n$  کاربر دیگر مقایسه کرد) و هنوز زمانبر است.

یک ایده، چند بار استفاده از روش درهمسازی میانی است! با این کار به احتمال بالا، بعد از چند مرحله، کاربرانی که به هم شبیه هستند در یک دسته قرار می‌گیرند. این ایده به Banding\*\* معروف است.

در روش Banding، امضای MinHash (که یک بردار از  $K$  مقدار MinHash است) به  $b$  نوار (band) تقسیم می‌شود. برای هر نوار، یکتابع هش جدایگانه اعمال می‌شود. اگر دو کاربر در حداقل یک نوار، هش یکسانی داشته باشند، آن‌ها به عنوان کاندیداهای همسایگی نزدیک در نظر گرفته می‌شوند و سپس شباهت دقیق آن‌ها محاسبه می‌شود. این کار باعث کاهش تعداد مقایسه‌های دقیق می‌شود.

## شبیه‌ترین کتاب!

در بخش‌های قبل این درسنامه سعی کردیم نزدیک‌ترین کاربر را به صورت تقریبی و با هزینه‌ی مناسب پیدا کنیم. فرض کنید متن کتاب‌ها را داریم و اینبار می‌خواهیم به ازای هر کتاب داده شده، نزدیک‌ترین کتاب را پیدا کنیم.

مهمترین سوالی که باید به آن پاسخ بدهیم این است که معیار شباهت دو کتاب یا دو رشته‌ی متنی چیست؟ تا اینجا فقط با معیار شباهت Jaccard آشنا شدیم. آیا می‌توانیم برای تعیین شباهت دو کتاب یا دو رشته از معیار Jaccard استفاده کنیم؟

## کیسه‌ی کلمات! (Bag of Words)

می‌توانیم از روی هر متن یک \*\*مجموعه‌ی نظری شامل کلمات آن متن (Bag of Words) بسازیم. این مجموعه شامل تمام کلمات منحصر به فردی است که در متن وجود دارند.

با توجه به این که مجموعه‌ی کلماتی که در دو متن متفاوت با محتوای شبیه به هم استفاده می‌شوند اشتراکات بیشتری از مجموعه کلمات دو متن با موضوعات متفاوت دارند، شاخص Jaccard مجموعه‌های نظری دو متن، معیار قابل قبولی از شباهت متن‌ها به هم ارائه می‌کند.

به عنوان مثال، احتمال تکرار کلمات "الگوریتم" یا "گراف" در یک متن با موضوع علوم کامپیوتر بسیار بیشتر از احتمال تکرار این کلمات در متنی با موضوع فیلم‌سازی است.

برای مثال، چهار متن از ابتدای چهار مقاله‌ی مختلف، دو تا با موضوع درهمسازی حساس به شباهت (LSH) و دو تا با موضوع فیلم‌سازی را مقایسه می‌کنیم.

In [8]:  
# از قبل تعریف شده است jaccard فرض می‌شود تابع  
# from jaccard\_module import jaccard # اگر در فایل جداگانه باشد #

# 0.txt, 1.txt about LSH  
# 2.txt, 3.txt about Filmmaking

مجموعه تمام کلمات منحصر به فرد در همه متن‌ها  
lisensi از مجموعه‌های کلمات برای هر متن #  
data\_vectors = [] برای هر متن # (0 و 1) برای هر متن #

for i in range(0, 4):  
 with open(f'./src/{i}.txt', 'r', encoding='utf-8') as f: # باز کردن فایل با  
 مناسب encoding جایگزین نقطه با فاصله و سپس تقسیم به کلمات #  
 استخراج کلمات: جایگزین نقطه با فاصله و سپس تقسیم به کلمات #  
 برای حذف کلمات تکراری در هر متن set تبدیل به #  
 set\_of\_words = set([word.lower() for line in f for word in line.replace('.', ' ').split() if word.isalpha()])  
 book\_words.append(set\_of\_words)  
 all\_words = all\_words.union(set\_of\_words) # افزودن به مجموعه کل کلمات

# تبدیل مجموعه‌های کلمات به بردارهای دو دویی  
# است all\_words هر سهون نشان‌دهنده یک کلمه منحصر به فرد از #

all\_words\_list = sorted(list(all\_words)) # مرتب کردن کلمات برای داشتن ترتیب ثابت در بردارها #

for i in range(0, 4): # برای هر متن #

```

d_vector = []
for w in all_words_list: # برای هر کلمه در مجموعه کل کلمات
    if w in book_words[i]: # اگر کلمه در من بود، ۱
        d_vector.append(1) # در غیر این صورت، ۰
    else:
        d_vector.append(0) # ۰
data_vectors.append(d_vector)

# بین متون Jaccard محاسبه و چاپ ماتریس شباهت
print("بین متون Jaccard ماتریس شباهت:")
for i in range(0, 4):
    sim_row = []
    for j in range(0, 4):
        # بین بردارهای دو دوی دو متون Jaccard محاسبه شباهت.
        sim_row.append(jaccard(data_vectors[i], data_vectors[j]))
    print(" ".join(["%.2f" % s for s in sim_row]))

```

بین متون Jaccard ماتریس شباهت:

```

1.00 0.13 0.11 0.11
0.13 1.00 0.09 0.06
0.11 0.09 1.00 0.13
0.11 0.06 0.13 1.00

```

نتایج خیلی خوب نیست! (احتمالاً شباهت بین مقالات LSH و فیلم‌سازی هم بالا به نظر می‌رسد).

اگر کلمات پر تکرار (مانند "the", "a", "is") و کلمات کم تکرار (مانند اصطلاحات تخصصی) وزن متفاوتی داشته باشند، نتایج به مرتب بهتر می‌شود. این ایده به مدل‌های TF-IDF (Term Frequency-Inverse\*\*) و کلمات می‌تواند ارزش معنایی (مانند حروف ربط، حروف اضافه، نقطه‌گذاری‌ها و کلمات متوقف‌کننده - \*\*Stop Words\*\*) به نتیجه‌ی بهتری برسیم.

\*\*Document Frequency)

همچنین می‌توانیم با حذف کلمات و علائم تکراری و بدون ارزش معنایی (مانند حروف ربط، حروف اضافه، نقطه‌گذاری‌ها و کلمات متوقف‌کننده - \*\*Stop Words\*\*) به نتیجه‌ی بهتری برسیم.

## گروه کلمات به جای تک کلمه (N-grams)

یک ایده برای افزایش دقیق شاخص Jaccard در مقایسه متون، اضافه کردن همه‌ی ترکیب‌های حاصل از دو یا چند کلمه‌ی مجاور (که به آنها \*\*N-grams\*\* می‌گویند) به جای کلمات تکی است. مثلاً برای  $N=2$ ، به جای کلمات "data" و "structure" از "data structure" استفاده می‌کنیم.

\*\*می‌توانید این بهبود عملکرد را توجیه کنید؟\*\*

\*\*پاسخ:\*\* استفاده از N-grams (به جای فقط کلمات تکی) به ما کمک می‌کند تا \*\*ترتیب کلمات\*\* و \*\*عبارات معنی‌دار\*\* را در نظر بگیریم. کلمات تکی ممکن است در موضوعات مختلفی به کار روند، اما ترکیب‌های دو یا چند کلمه‌ای، معمولاً معنای دقیق‌تر و خاص‌تری دارند. مثلاً، کلمه "bank" می‌تواند به معنی "بانک پول" یا "کناره رودخانه" باشد، اما "river bank" به وضوح به معنی "کناره رودخانه" است. این کار باعث می‌شود که شباهت بین متون با موضوعات مشابه، دقیق‌تر و بالاتر ارزیابی شود.

## منابع

1. Mining of Massive Datasets by Anand Rajaraman and Jeffrey D. Ullman
2. Nearest Neighbors in High-dimensional Spaces by Piotr Indyk

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۵-۱۴۰۴

## فصل دهم، بخش اول: ذخیره‌سازی و نمایش گراف

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

• مقدمه

• تعاریف

• انواع ذخیره‌سازی گراف

• ذخیره‌سازی با ماتریس مجاورت

• ذخیره‌سازی با لیست پیوندی

• ذخیره‌سازی با ماتریس وقوع

### مقدمه

گراف‌ها داده‌ساختارهایی بسیار بنیادی و پرکاربرد در حل انواع مسائل در علوم کامپیوتر، ریاضیات، و مهندسی هستند. آن‌ها برای مدل‌سازی روابط بین اشیاء به کار می‌روند.

مسائل ساده‌ای مانند یافتن کوتاه‌ترین مسیر در یک شبکه راه‌ها یا شبکه‌های کامپیوتری) تا مسائل پیچیده‌ای مانند تعیین بیشترین مقدار آبی که می‌توان از مجموعه‌ای از لوله‌ها عبور داد که هر کدام حداقل ظرفیت مشخصی دارند (مسئله<sup>\*</sup>حداکثر جریان - برش حداقل (Maximum-Flow Minimum-Cut)، می‌توانند به کمک گراف‌ها مدل شوند.

استفاده از داده‌ساختار مناسب در پیاده‌سازی و حل مسائل گراف از نکات مهمی است که باید به آن توجه کرد. انتخاب نمایش مناسب برای گراف می‌تواند به طور چشمگیری بر کارایی عملیات‌های مختلف (مانند افزودن/حذف رأس/یال، بررسی وجود یال، یا پیمایش گراف) تأثیر بگذارد.

در ادامه این بخش به معرفی گراف‌ها و روش‌های مختلف نمایش و پیاده‌سازی آن‌ها می‌پردازیم.

### تعاریف در گراف

همانطور که می‌دانید، گراف داده‌ساختاری است که از دو جزء اصلی زیر تشکیل شده است:

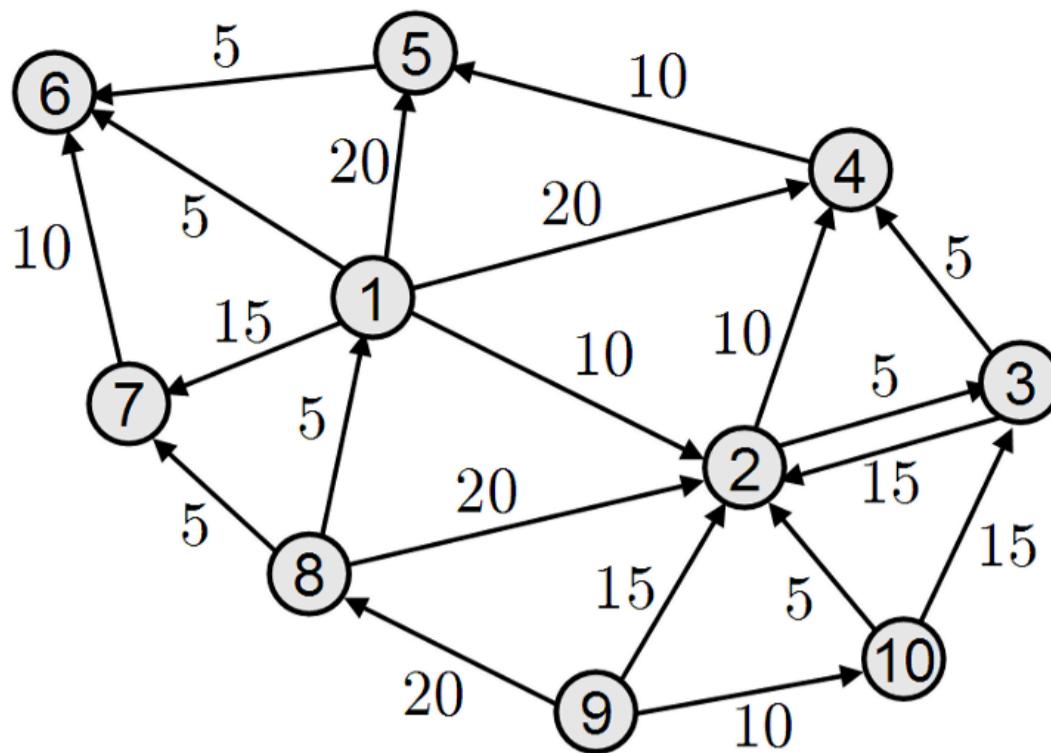
• **گره‌ها (Vertices / Nodes):** مجموعه‌ای متناهی از رأس‌ها. هر گره می‌تواند اطلاعات گوناگونی را در خود ذخیره کند (برای مثال یک ID، اسم، یا هر داده‌ی مرتبط دیگر).

• **یال‌ها (Edges):** مجموعه‌ای متناهی از جفت رأس‌های به هم متصل. یال‌ها نشان‌دهنده رابطه یا اتصال بین گره‌ها هستند.

■ در صورتی که گراف<sup>\*\*</sup>جهت‌دار (Directed) باشد، ترتیب رأس‌ها در جفت (مثلاً از  $u$  به  $v$ ) اهمیت دارد.

■ در صورتی که گراف<sup>\*\*</sup>بدون جهت (Undirected) باشد، ترتیب رأس‌ها اهمیت ندارد (یال بین  $u$  و  $v$  همان یال بین  $v$  و  $u$  است).

■ همچنین در بعضی مسائل به هر یال یک عدد یا ارزش نسبت می‌دهیم که اصطلاحاً به آن<sup>\*</sup> وزن یال (Edge Weight) می‌گویند.



## ذخیره سازی و نمایش گراف

برای نمایش یک گراف در حافظه کامپیوتر، دو روش رایج وجود دارد که هر کدام مزایا و معایب خاص خود را دارند و بسته به نوع گراف و عملیات های مورد نیاز، یکی بر دیگری ارجحیت دارد:

- **ماتریس مجاورت (Adjacency Matrix)**
- **لیست مجاورت (Adjacency List)**

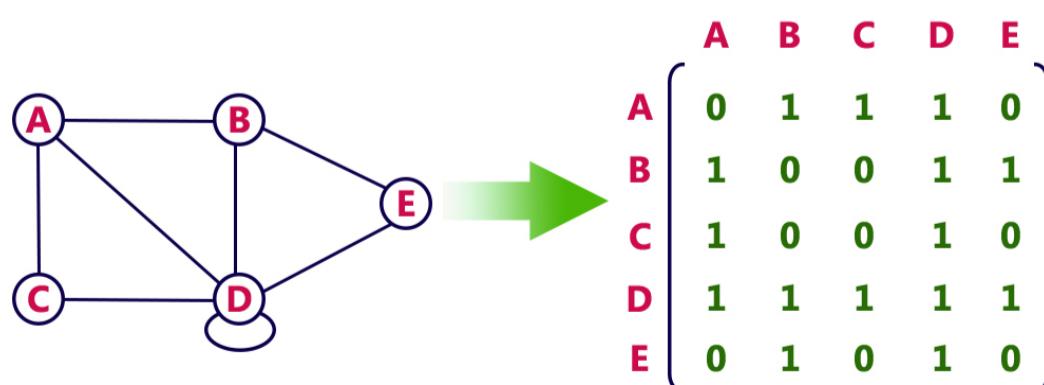
همچنین روش های دیگری همانند استفاده از ماتریس وقوع (Incidence Matrix) نیز وجود دارد. به صورت کلی نمایش و پیاده سازی یک گراف بسته به کاربردی که برای آن طراحی می شود می تواند متفاوت باشد. هر چه پیاده سازی ما باعث شدن عملیات روی گراف شود، پیاده سازی بهتری محسوب می شود.

در ادامه هر کدام از روش های فوق را توضیح می دهیم و پیاده سازی های ساده ای از آن ها را ارائه می کنیم.

## نمایش با ماتریس مجاورت

ماتریس مجاورت یک ماتریس دو بعدی (آرایه دو بعدی) از ۰ و ۱ ها است که به تعداد رأس های گراف، سطر و ستون دارد. فرض کنید این ماتریس را  $\text{adj}[[i][j]]$  بنامیم. در این صورت  $\text{adj}[i][j] = 1$  اگر و تنها اگر بالی از رأس  $i$  به رأس  $j$  وجود داشته باشد. بنابراین در صورتی که گراف جهت دار نباشد، ماتریس مجاورت همواره متقارن خواهد بود.

همچنین با تغییر کوچکی در ماتریس مجاورت می توان از آن برای نمایش گراف های وزن دار استفاده کرد. به این صورت که  $\text{adj}[i][j] = w$  اگر و تنها اگر یالی با وزن  $w$  از رأس  $i$  به رأس  $j$  وجود داشته باشد. اگر یالی وجود نداشته باشد، می توان از یک مقدار خاص (مثلاً ۰ یا بینهایت) استفاده کرد.



در شکل بالا نمایش یک گراف ۵ رأسی با ماتریس مجاورت آن را مشاهده می کنید.

- **مزایا:** پیاده‌سازی با این روش ساده و واضح‌تر است. بررسی وجود یک یال بین دو رأس (مثلاً `adj[i][j]`) در  $O(1)$  مرتبه زمانی ( $O(1)$  انجام‌پذیر است. حذف یک یال نیز با تغییر مقدار خانه مربوطه به ۰ در مرتبه زمانی ( $O(1)$  انجام می‌شود).
- **معایب:** ذخیره‌سازی گراف با این روش به حافظه  $O(V^2)$  نیاز دارد، که  $V$  تعداد رأس‌ها است. این روش برای گراف‌های \*\*(Sparse Graphs) (گراف‌هایی که تعداد یال‌هایشان بسیار کمتر از  $V^2$  است) ناکارآمد است. همچنین اضافه کردن یک رأس جدید به گراف نیز در مرتبه زمانی  $O(V^2)$  (نیاز به تغییر اندازه ماتریس) امکان‌پذیر است.

در قطعه کد زیر، پیاده‌سازی یک گراف با استفاده از ماتریس مجاورت را مشاهده می‌کنید.

```
In [1]: class GraphAdjacencyMatrix:
    def __init__(self, num_vertices, directed=False):
        متد سازنده: یک گراف را با استفاده از ماتریس مجاورت مقداردهی اولیه می‌کند
        # تعداد رأس‌ها
        self.num_vertices = num_vertices
        آیا گراف جهت‌دار است؟
        self.directed = directed
        # با تمام مقادیر ۰ (بدون یال) ایجاد یک ماتریس
        self.adj_matrix = [[0 for _ in range(num_vertices)] for _ in range(num_vertices)]  
  

    def add_edge(self, u, v, weight=1):
        به رأس u اضافه کردن یک یال از رأس v.
        # اگر گراف وزن‌دار باشد، وزن را ذخیره می‌کند؛ در غیر این صورت ۱
        if u < 0 or u >= self.num_vertices or v < 0 or v >= self.num_vertices:
            raise ValueError("Vertices out of bounds.")
        self.adj_matrix[u][v] = weight
        if not self.directed:
            self.adj_matrix[v][u] = weight # اگر گراف بدون جهت است، یال برگشتی را هم اضافه کن  
  

    def remove_edge(self, u, v):
        # حذف یک یال از رأس v.
        if u < 0 or u >= self.num_vertices or v < 0 or v >= self.num_vertices:
            raise ValueError("Vertices out of bounds.")
        self.adj_matrix[u][v] = 0
        if not self.directed:
            self.adj_matrix[v][u] = 0  
  

    def has_edge(self, u, v):
        وجود دارد یا خیر v به رأس u بررسی می‌کند که آیا یالی از رأس v به رأس u برمی‌گرداند
        # ایا یالی از رأس u به رأس v برمی‌گرداند
        if u < 0 or u >= self.num_vertices or v < 0 or v >= self.num_vertices:
            raise ValueError("Vertices out of bounds.")
        return self.adj_matrix[u][v] != 0  
  

    def get_weight(self, u, v):
        # را برمی‌گرداند v به u وزن یال از
        if self.has_edge(u, v):
            return self.adj_matrix[u][v]
        return None # یا می‌توانیم یک مقدار بی‌نهایت یا خطاب برمی‌گردانیم  
  

    def print_graph(self):
        # جاپ ماتریس مجاورت
        print("Adjacency Matrix:")
        for row in self.adj_matrix:
            print(row)  
  

# مثال استفاده از GraphAdjacencyMatrix
g_matrix = GraphAdjacencyMatrix(5, directed=False) # یک گراف بدون جهت با 5 رأس
g_matrix.add_edge(0, 1)
g_matrix.add_edge(0, 4)
g_matrix.add_edge(1, 2)
g_matrix.add_edge(1, 3)
g_matrix.add_edge(1, 4)
g_matrix.add_edge(2, 3)
g_matrix.add_edge(3, 4)
g_matrix.print_graph()
print(f"آیا یالی بین 0 و 1 وجود دارد؟ {g_matrix.has_edge(0, 1)}")
print(f"آیا یالی بین 0 و 2 وجود دارد؟ {g_matrix.has_edge(0, 2)}")
g_matrix.remove_edge(0, 1)
g_matrix.print_graph()
```

```
Adjacency Matrix:  
[0, 1, 0, 0, 1]  
[1, 0, 1, 1, 1]  
[0, 1, 0, 1, 0]  
[0, 1, 1, 0, 1]  
[1, 1, 0, 1, 0]
```

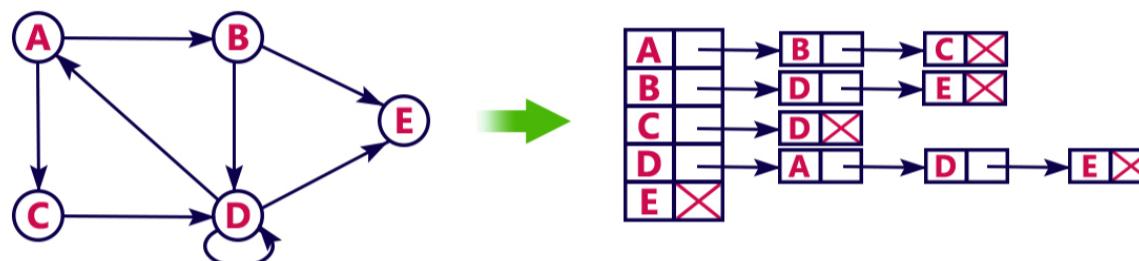
```
آیا یالی بین 0 و 1 وجود دارد?  
آیا یالی بین 0 و 2 وجود دارد?  
Adjacency Matrix:  
[0, 0, 0, 0, 1]  
[0, 0, 1, 1, 1]  
[0, 1, 0, 1, 0]  
[0, 1, 1, 0, 1]  
[1, 1, 0, 1, 0]
```

## نمایش با لیست مجاورت

برای پیادهسازی در این روش، از آرایه‌ای از لیست‌های پیوندی (یا لیست‌های پایتون) استفاده می‌شود. اندازه‌ی این آرایه برابر با تعداد رأس‌های گراف است. این آرایه را `adj\_list` می‌نامیم.

در این صورت، `adj\_list[i]` یک لیست پیوندی (یا لیست پایتون) از رأس‌هایی که یال جهت‌داری از رأس  $i$  به آنها وجود دارد را نشان می‌دهد.

این نمایش همچنین می‌تواند برای گراف‌های وزن‌دار به کار گرفته شود. به طوری که وزن یال‌ها در عناصر لیست‌های پیوندی (مثلًاً به صورت تاپل `(neighbor, weight)` ذخیره شود).



در شکل بالا نمایش یک گراف ۵ رأسی با لیست مجاورت آن را مشاهده می‌کنید.

مزایا و معایب این نوع ذخیره‌سازی به شرح زیر است:

- مزایا:** پیادهسازی با این روش حافظه‌ای از مرتبه  $O(|V| + |E|)$  نیاز دارد، که  $|V|$  تعداد رأس‌ها و  $|E|$  تعداد یال‌ها است. این روش برای گراف‌های \*\*تنک (Sparse Graphs)\*\* بسیار کارآمدتر از ماتریس مجاورت است. حداقل تعداد یال‌های گراف برابر  $\binom{|V|}{2}$  است، بنابراین در بدترین حالت (گراف کامل)، حافظه مورد نظر از  $O(V^2)$  است. همچنین افزودن یک رأس معادل با اضافه کردن یک سطر جدید به آرایه است که نسبت به ماتریس مجاورت ساده‌تر است.
- معایب:** مرتبه زمانی بررسی وجود یال بین دو رأس ( $O(\text{degree}(u, v))$ ) پیچیدگی (مثلًاً `has\_edge(u, v)` (در بدترین حالت  $O(V)$ ) را دارد، زیرا باید در لیست پیوندی مربوطه در آرایه، به جستجو پرداخت. بنابراین از این لحاظ بهینه نخواهد بود. حذف یک یال نیز در بدترین حالت  $O(V)$  زمان می‌برد.

در قطعه کد زیر، پیادهسازی یک گراف با استفاده از لیست مجاورت را مشاهده می‌کنید.

```
In [2]: class GraphAdjacencyList:  
    def __init__(self, num_vertices, directed=False):  
        متد سازنده: یک گراف را با استفاده از لیست مجاورت مقداردهی اولیه می‌کند  
        self.num_vertices = num_vertices # تعداد رأس‌ها  
        self.directed = directed # آیا گراف جهت‌دار است؟  
        ایجاد لیستی از لیست‌های خالی برای هر رأس  
        self.adj_list = [[] for _ in range(num_vertices)]  
  
    def add_edge(self, u, v, weight=1):  
        # به رأس u اضافه کردن یک یال از رأس v.  
        # اگر گراف وزن‌دار باشد، وزن را به همراه رأس همسایه ذخیره می‌کند  
        if u < 0 or u >= self.num_vertices or v < 0 or v >= self.num_vertices:  
            raise ValueError("Vertices out of bounds.")  
        self.adj_list[u].append((v, weight))  
        if not self.directed:  
            self.adj_list[v].append((u, weight))  
  
    def remove_edge(self, u, v):  
        # یال از رأس u حذف یک یال از رأس v.  
        if u < 0 or u >= self.num_vertices or v < 0 or v >= self.num_vertices:  
            raise ValueError("Vertices out of bounds.")  
  
        # حذف از لیست u
```

```

self.adj_list[u] = [(neighbor, w) for neighbor, w in self.adj_list[u] if neighbor != v]

if not self.directed:
    # برای گراف بدون جهت) v حذف از لیست
    self.adj_list[v] = [(neighbor, w) for neighbor, w in self.adj_list[v] if neighbor != u]

def has_edge(self, u, v):
    # وجود دارد یا خیر v به رأس u بررسی می کند که آیا بالی از رأس.
    if u < 0 or u >= self.num_vertices or v < 0 or v >= self.num_vertices:
        raise ValueError("Vertices out of bounds.")
    for neighbor, _ in self.adj_list[u]:
        if neighbor == v:
            return True
    return False

def get_neighbors(self, u):
    # را بر می گرداند u لیست همسایگان رأس.
    if u < 0 or u >= self.num_vertices:
        raise ValueError("Vertex out of bounds.")
    return self.adj_list[u]

def print_graph(self):
    # چاپ لیست مجاورت
    print("Adjacency List:")
    for i, neighbors in enumerate(self.adj_list):
        print(f"Vertex {i}: {neighbors}")

# مثال استفاده از GraphAdjacencyList
g_list = GraphAdjacencyList(5, directed=True) # یک گراف جهت دار با 5 رأس
g_list.add_edge(0, 1, 10)
g_list.add_edge(0, 4, 20)
g_list.add_edge(1, 2, 30)
g_list.add_edge(1, 3, 40)
g_list.add_edge(1, 4, 50)
g_list.add_edge(2, 3, 60)
g_list.add_edge(3, 4, 70)
g_list.print_graph()
print(f"\nآیا بالی از 0 به 1 وجود دارد؟ {g_list.has_edge(0, 1)}")
print(f"آیا بالی از 0 به 2 وجود دارد؟ {g_list.has_edge(0, 2)}")
g_list.remove_edge(0, 1)
g_list.print_graph()

```

Adjacency List:  
Vertex 0: [(1, 10), (4, 20)]  
Vertex 1: [(2, 30), (3, 40), (4, 50)]  
Vertex 2: [(3, 60)]  
Vertex 3: [(4, 70)]  
Vertex 4: []

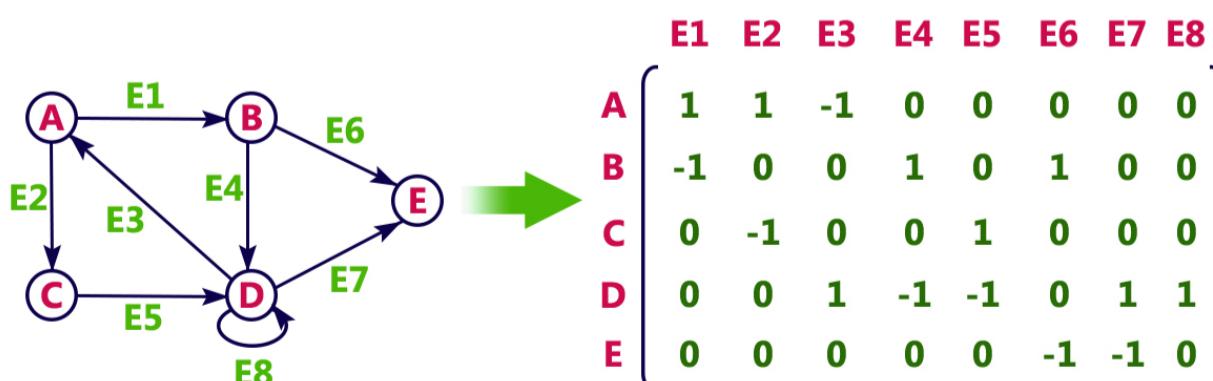
آیا بالی از 0 به 1 وجود دارد? True  
آیا بالی از 0 به 2 وجود دارد? False  
Adjacency List:  
Vertex 0: [(4, 20)]  
Vertex 1: [(2, 30), (3, 40), (4, 50)]  
Vertex 2: [(3, 60)]  
Vertex 3: [(4, 70)]  
Vertex 4: []

## نمایش با ماتریس وقوع

برای پیاده سازی در این روش، از آرایه ای دو بعدی (ماتریس) استفاده می شود که تعداد سطرهای آن برابر با تعداد رأس های گراف و تعداد ستون های آن برابر با تعداد یال های گراف است.

در صورتی که گراف بدون جهت باشد، اعضای این ماتریس 0 و 1 هستند. اگر یال زام بین رأس u و v باشد، آنگاه در سطر u و v در ستون j مقدار 1 قرار می گیرد و در بقیه سطرها در ستون j مقدار 0.

اما برای گراف های وزن دار یا جهت دار می توان مشابه قسمت های قبل نمایش معادلی را ارائه کرد. در حالت ساده شده، اگر یال ها را نیز شماره گذاری کنیم، در سطر iام و ستون jام ماتریس 1 قرار می دهیم اگر و تنها اگر یک سر یال شماره j رأس i قرار داشته باشد.



در شکل بالا نمایش یک گراف 5 رأسی با ماتریس وقوع آن را مشاهده می کنید. همانطور که دیده می شود، چون این گراف جهت دار است، یال های خروجی از رأس با +1 و یال های ورودی با -1 نشان داده شده اند. اگر یال وجود نداشته باشد، مقدار 0 قرار می گیرد.

- \*\*مزایا:\*\* این روش برای گرافهایی که تعداد یال‌هایشان بسیار زیاد است (گرافهای چگال) و نیاز به دانستن ارتباط بین یال‌ها و رأس‌ها داریم، مفید است. همچنین برای نمایش گرافهای چندگانه (Multiple) و حلقه‌ها (Loops) مناسب است.
- \*\*معایب:\*\* استفاده از حافظه  $O(|V| \times |E|)$  که می‌تواند بسیار زیاد باشد. عملیات‌هایی مانند افزودن/حذف یال یا رأس نیز می‌تواند پرهزینه باشد. بررسی وجود یال بین دو رأس خاص نیز پیچیده است.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل دهم، بخش دوم: الگوریتم‌های پیمایش گراف

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

- مقدمه
- انواع الگوریتم‌های جستجو
- الگوریتم BFS
- الگوریتم DFS

### مقدمه

فرض کنید دنبال فایل یا پوشه‌ای در لپتاپتان می‌گردد. در قدم اول از یک پوشه شروع می‌کنید. ممکن است بعد از آن چند روش پیش بگیرد. مثلاً فرض کنید اکنون در پوشه‌ی A هستید. فایل/پوشه‌های A1، A2 و ... داخل آن هستند.

یک روش احتمالی این است که ابتدا فایل/پوشه‌های A1 تا An را بررسی کنید. سپس به ترتیب فایل‌های داخل F1، داخل F2 و همینطور تا Fn را بررسی کنید. سپس دوباره به ترتیب فایل‌های داخل این‌ها الی آخر. این روش شبیه به پیمایش **BFS** (Breadth-First) است.

روش دیگر ممکن است این باشد که ابتدا پوشه‌ی A1 را بررسی کنید، سپس اولین فایل داخل آن و مجدد اولین فایل داخل آن تا آخر، سپس برگردید و به صورت بازگشتی این عملیات را تکرار کنید. این روش شبیه به پیمایش **DFS** (Depth-First) است.

البته روش آسان‌تر هم این است که قسمت search لپتاپتان را باز کنید!

با روش‌های بالا، در این مثال مشکلی پیش نمی‌آید، زیرا نمی‌شود فولدر C در B، فولدر B در A، فولدر C در A باشد (اگر در لپتاپ شما چنین پدیده‌ای رخ داده‌است سریع به تعمیرکار رجوع کنید). به عبارتی در گراف متناظر دور نداریم. در حالی که در اکثر گراف‌هایی که ما با آن‌ها سر و کار داریم نمی‌توانیم چنین فرضی بکنیم (گراف‌ها معمولاً شامل دور هستند).

در واقع مسئله‌ی پیش رو، مسئله‌ی **پیمایش گراف** (Graph Traversal) است. شما تاکنون با شیوه‌های مختلف نمایش و ذخیره‌سازی گراف‌ها آشنا شدید. یکی از دلایل مدل کردن مسئله‌های گوناگون با گراف‌ها، فراهم نمودن امکان جستجو و استفاده از الگوریتم‌های جستجوی گراف است. الگوریتم‌های گوناگونی برای جستجو در گراف وجود دارد که در این بخش با انواع روش‌های آن آشنا می‌شویم.

## انواع الگوریتم‌های جستجو

ابتدا مفهوم جستجو در گراف را بیان می‌کنیم. جستجو یا پیمایش یک گراف (Graph Traversal / Graph Search) به معنای طی کردن رأس‌های گراف به یک ترتیب خاص است. این پیمایش می‌تواند برای یافتن رأس‌هایی با ویژگی خاص در گراف (مثلاً نزدیکترین رأس به مبدأ) و یا آپدیت کردن وضعیت گره‌ها (مانند محاسبه کوتاه‌ترین مسیر) به کار برد شود.

الگوریتم‌های جستجو بر حسب ترتیب پیمایش رأس‌ها به دسته‌های گوناگون تقسیم می‌شوند. در ادامه به معرفی مهم‌ترین این الگوریتم‌ها یعنی\*\* BFS (Breadth-First Search) و DFS (Depth-First Search) می‌پردازیم. سایر الگوریتم‌های جستجو در حالت کلی تغییر یافته و یا الهام گرفته از این دو الگوریتم هستند.

## الگوریتم BFS

پیمایش اول سطح (BFS - Breadth-First Search)\*\* در گراف شبیه به روش پیشنهادی اول برای جستجو در فایل است. یعنی از یک رأس شروع کنیم، سپس همه‌ی فرزندان آن را نگاه کنیم، سپس به ترتیب همه‌ی فرزندان فرزندان آن‌ها الی آخر (یعنی تمام گره‌های یک سطح را قبل از رفتن به سطح بعدی بررسی می‌کنیم).

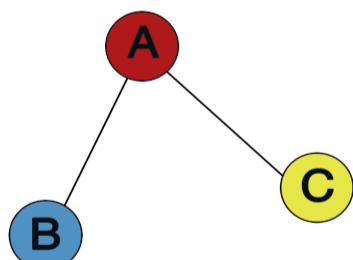
به نظر شما استفاده از چه داده‌ساختاری برای پیاده‌سازی این عملیات مناسب است؟

فرض کنید از رأس  $s$  شروع می‌کنیم. به ترتیب رأس‌های متصل به آن را نگاه می‌کنیم. برای اینکه بعد از بررسی آن‌ها، بتوانیم رأس‌هایی متصل به آن‌ها را نیز به ترتیب بررسی کنیم، باید چه کار کنیم؟ با کمی تفکر، متوجه می‌شویم که روش پردازش مانند شیوه‌ی FIFO (First-In First-Out) است. پس باید از داده‌ساختار \*\*صف (Queue)\*\* استفاده کنیم.

مراحل زیر را ط می‌کنیم:

1. یک رأس برای شروع انتخاب می‌کنیم و در ابتدای صف قرار می‌دهیم.
2. تا زمانی که صف خالی نیست، به ترتیب عملیات زیر را انجام دهیم:
  - A. از ابتدای صف رأسی برداریم و بررسی کنیم.
  - B. همه‌ی رئوس متصل به آن را به انتهای صف اضافه کنیم.

دققت کنید که این عملیات مشکل اولیه‌ی ما را حل نمی‌کنند. یعنی اگر در گراف مربوطه دور داشته باشیم، صف هرگز خالی نمی‌شود و ما در حلقه‌ی بینهایت گیر می‌افتیم. حتی گراف زیر را در نظر بگیرید:

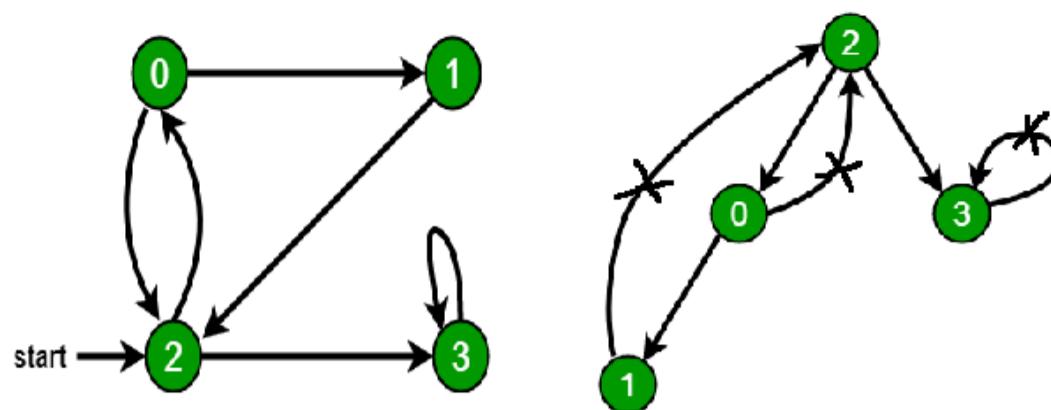


فرض کنید رأس A را برای شروع انتخاب کرده‌ایم. آن را در اول صف قرار می‌دهیم، سپس در مرحله‌ی 2. A. آن را از صف خارج می‌کنیم. در مرحله‌ی 2. B. رئوس B و C را در انتهای صف قرار می‌دهیم. سپس مجدد در مرحله‌ی 2. A. رأس B را از صف خارج می‌کنیم و رأس A را در صف قرار می‌دهیم. هر گاه نوبت A شود دوباره این مراحل تکرار می‌شود و ... .

برای حل این مشکل، راه حل زیر را پیش می‌گیریم:

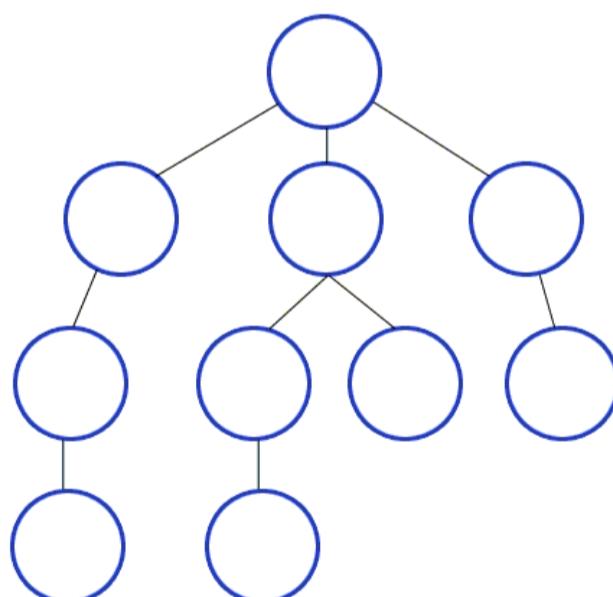
نتها رئوسی که قبلاً بررسی نکردہ‌ایم را به صف اضافه می‌کنیم.

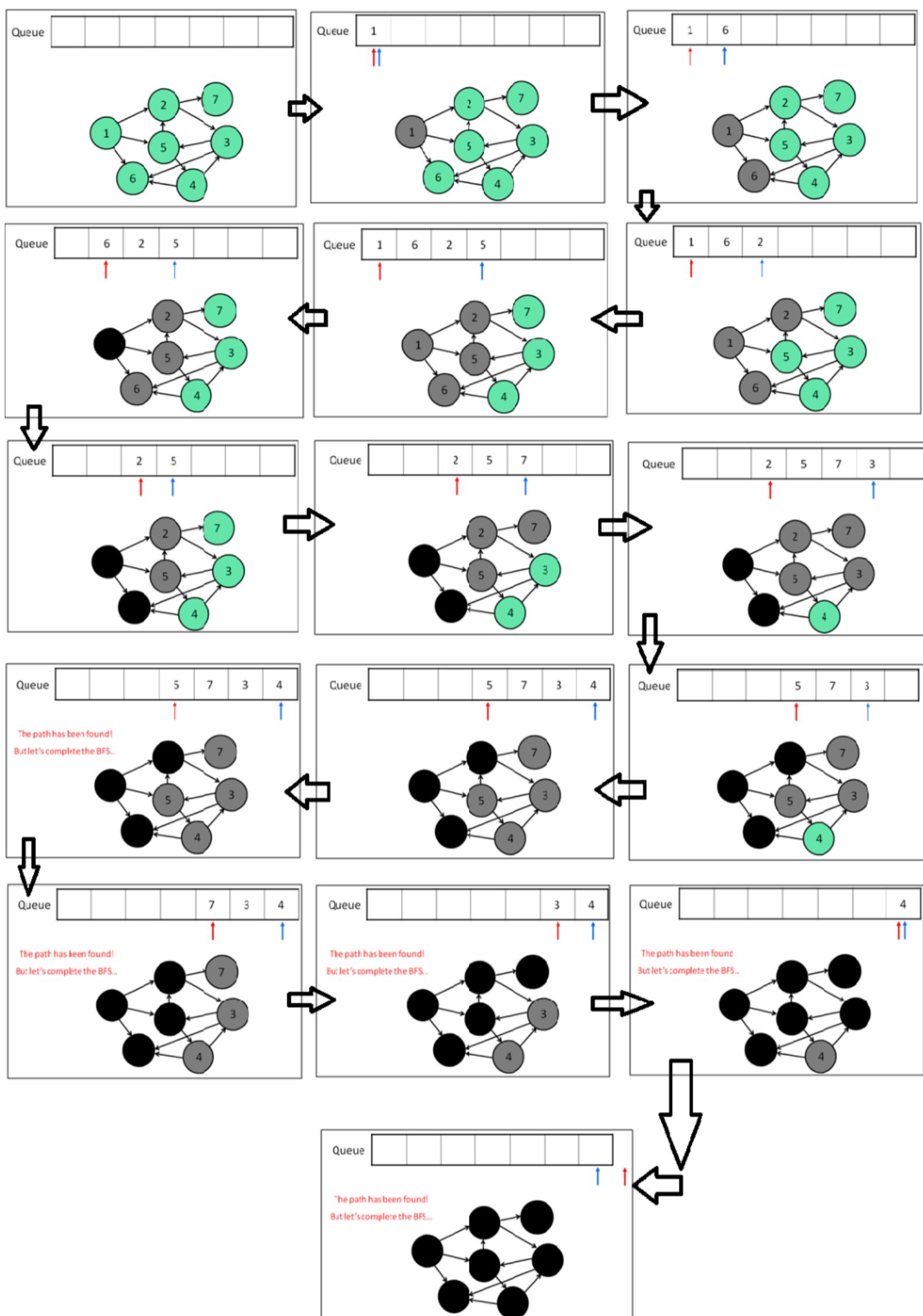
برای تشخیص اینکه رأسی را قبلاً دیده‌ایم یا نه، می‌توانیم یک متغیر (مثلاً یک آرایه `visited` یا یک مجموعه `set`) به ویژگی‌های آن اضافه کنیم یا وضعیت آن را در یک لیست جداگانه نگهداری کنیم.



- از رأس شماره‌ی 2 شروع به پیمایش می‌کنیم. رأس 2 را علامت‌گذاری می‌کنیم و رئوس 0 و 3 را به صف اضافه می‌کنیم. سپس به رأس 0 می‌رویم (2 انتخاب داریم که فرض کنید ترتیب انتخاب‌ها به ترتیب اعداد باشد).
- رأس‌های مجاور رأس 0، رأس‌های 1 و 2 هستند. رأس 2 قبلاً دیده شده است. اگر رأس‌های دیده شده را علامت‌گذاری نکنیم، ممکن است در برخی حالات همانند این حالت پیمایش در یک دور نامتناهی بیفتد که مطلوب نیست. بنابراین با فرض علامت‌گذاری رأس‌های پیمایش شده، رأس 1 به انتهای صف اضافه می‌شود و به رأس شماره‌ی 3 می‌رویم (که الان در ابتدای صف قرار دارد).
- تنها رأس مجاور رأس 3، خود رأس 3 است که قبلاً پیمایش شده. بنابراین نمی‌توان پیمایش را از این مسیر ادامه داد و رأسی به صف اضافه نمی‌شود. بنابراین به رأس 1 که آخرین رأس موجود در صف است می‌رویم.
- رأس شماره‌ی 1 نیز فقط به رأس شماره‌ی 2 راه دارد که قبلاً پیمایش شده، پس آن را به صف اضافه نمی‌کند و پیمایش به پایان می‌رسد.
- دباله‌ی پیمایش BFS این گراف به صورت 1, 0, 3, 2 است.

در زیر تصویرسازی این الگوریتم را می‌بینید:





## پیاده‌سازی

ابتدا گراف را به صورت زیر پیاده‌سازی می‌کنیم. در پیاده‌سازی زیر از لیست همسایگان (Adjacency List) استفاده شده است.

In [1]: `from collections import defaultdict # برای استفاده از defaultdict می‌کند`

```
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
```

```
# و u اضافه کردن یک یال بدون جهت بین رأس v.
# self.graph[u].append(v) اضافه کردن v به لیست همسایگان u
# self.graph[v].append(u) اضافه کردن u به لیست همسایگان v (برای گراف بدون جهت)
```

کد پیمایش سطح اول (BFS) به صورت زیر است:

```
In [2]: def BFS(g, s):
    """
    انجام می‌دهد s از رأس شروع g را روی گراف (BFS) این تابع پیمایش سطح اول را انجام می‌دهد.
    رأس‌های پیمایش شده را چاپ می‌کند.
```

Args:

```
g (Graph): شیء گراف.
s (int): رأس شروع پیمایش.
```

"""

```
# اگر بازدید شده باشد True) لیستی برای نگهداری وضعیت بازدید هر رأس.
# اندازه آن برابر با تعداد کل رأس‌ها در گراف است.
# len(g.graph) که حداقل یک یال دارند را می‌دهد (برای گراف‌های با رأس‌های ایزوله، باید از استفاده کرد max(g.graph.keys()) + 1
# باید از این روش استفاده کرد رأس‌ها را در سازنده گراف ذخیره کرد.
# num_vertices = max(g.graph.keys()) + 1 if g.graph else 0
visited = [False] * num_vertices
```

صف برای نگهداری رأس‌هایی که باید بازدید شوند. رأس شروع را اضافه می‌کند.
visited[s] = True # علامت‌گذاری می‌کند
print(s) # چاپ رأس شروع.

تا زمانی که رأس برای بررسی ادامه می‌دهیم # (FIFO). رأس ابتدای صف را برمندی‌داریم.

```
q = [s] # پیمایش بر روی تمام همسایگان رأس
for v in g.graph[u]:
    if not visited[v]: # قبلآ بررسی نشده است v اگر همسایه.
        آن را به صف اضافه می‌کنیم # q.append(v)
        آن را نشانه گذاری می‌کنیم تا دیگر بررسی نشود # visited[v] = True
        چاپ رأس بازدید شده # print(v)
```

آن را امتحان می‌کنیم:

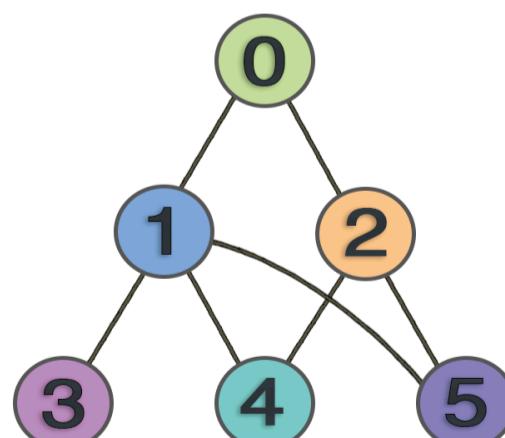
```
In [3]: ایجاد یک شیء گراف جدید.
```

```
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 4)
g.add_edge(2, 5)
g.add_edge(5, 1) # ایجاد یک دور در گراف.
```

از رأس 0 فراخوانی BFS 0.

```
0  
1  
2  
3  
4  
5
```

توجه کنید که گرافی که ساختیم به صورت زیر بود.



سعی کنید خودتان روی کاغذ برای این گراف الگوریتم BFS را اجرا کنید و ببینید مطابق خروجی کد می‌شود یا نه.

احتمالاً تا اینجا متوجه شده‌اید که روش فوق برای گراف‌های غیرهمبند (Disconnected Graphs) کار نمی‌کند ( فقط کامپوننت متصل به رأس شروع را پیمایش می‌کند). این روش را تغییر دهید تا این گراف‌ها را نیز بتواند درست پیمایش کند.

\*\*پاسخ:\*\* برای پیمایش گراف‌های غیرهمبند، باید یک حلقه بیرونی اضافه کنیم که تمام رأس‌های گراف را پیمایش کند. اگر رأسی هنوز بازدید نشده بود، BFS را از آن رأس شروع کنیم.

```
def BFS_Disconnected(g):
    num_vertices = max(g.graph.keys()) + 1 if g.graph else 0
    visited = [False] * num_vertices

    for s in range(num_vertices): # پیمایش بر روی تمام رأس‌ها به عنوان نقطه شروع
        if not visited[s]: # هنوز بازدید نشده بود s اگر رأس
            q = [s]
            visited[s] = True
            print(f"--- {s} از رأس BFS شروع ---")
            print(s)

            while len(q) > 0:
                u = q.pop(0)
                for v in g.graph[u]:
                    if not visited[v]:
                        q.append(v)
                        visited[v] = True
                        print(v)
```

## الگوریتم DFS

\*پیمایش اول عمق (Depth-First Search - DFS)\*\* در گراف شبیه به روش پیشنهادی دوم برای جستجوی فایل است. یعنی از یک رأس شروع کنیم، سپس فرزند اول آن را نگاه کنیم و همین کار را به صورت بازگشتی برای آن انجام دهیم (تا زمانی که به یک برگ یا رأس بازدید شده برسیم)، پس از بازگشت از یک مسیر، همین مراحل را برای فرزند بعدی انجام دهیم الی آخر.

به نظر شما استفاده از چه داده‌ساختاری برای پیاده‌سازی این عملیات مناسب است؟

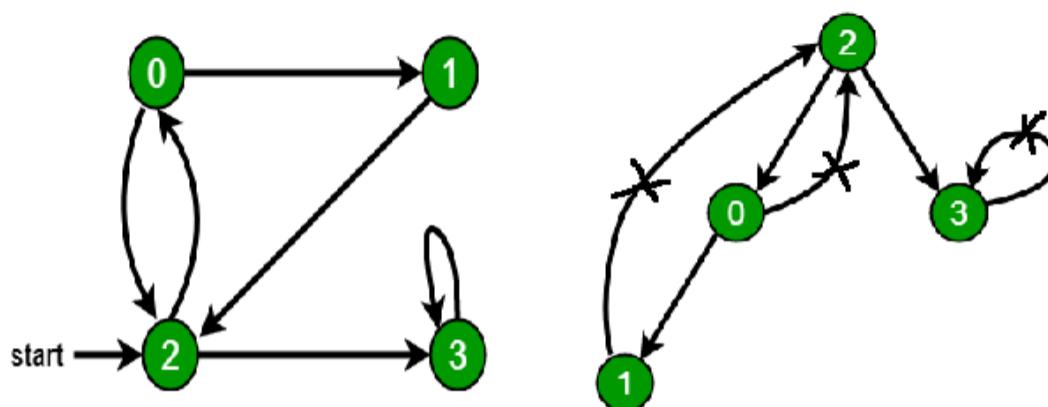
با توجه به اینکه عملیات توصیف شده ماهیت بازگشتی دارد، متوجه می‌شویم که روش پردازش مانند شیوه‌ی LIFO (Last-In First-Out) است. پس باید از داده‌ساختار \*پیشنهادی Stack استفاده کنیم. البته استفاده از داده‌ساختار پیشنهادی است، زیرا فراخوانی‌های بازگشتی توابع به صورت داخلی از یک پیشنهادی (پیشنهادی فراخوانی) استفاده می‌کنند.

مراحل زیر را طی می‌کنیم:

1. یک رأس را برای شروع انتخاب کنیم و بررسی کنیم (آن را به عنوان بازدید شده علامت‌گذاری کنیم).
2. برای همه‌ی رأس‌های مجاور آن (همسایگان) که هنوز بازدید نشده‌اند، عملیات را به صورت بازگشتی انجام دهیم. پس از اتمام بررسی تمام همسایگان یک رأس، بازگردیم.

تنها رئوسی که قبلاً بررسی نکرده‌ایم را بازدید می‌کنیم.

برای توضیح بهتر از مثال زیر استفاده می‌کنیم:



- از رأس شماره‌ی 2 شروع به پیمایش می‌کنیم. آن را علامت‌گذاری کرده و چاپ می‌کنیم. سپس به سراغ همسایگانش می‌رویم. فرض کنید ترتیب انتخاب‌ها به ترتیب اعداد باشد، پس به رأس 0 می‌رویم.
- رأس 0 را علامت‌گذاری و چاپ می‌کنیم. همسایگانش 1 و 2 هستند. رأس 2 قبلاً دیده شده است. پس رأس 1 را انتخاب و به آن می‌رویم.
- رأس 1 را علامت‌گذاری و چاپ می‌کنیم. تنها همسایه‌اش رأس 2 است که قبلاً دیده شده. بنابراین پیمایش از این مسیر به پایان می‌رسد و به رأس 0 بازمی‌گردیم.
- از رأس 0، دیگر همسایه ندیده‌ایم. پس به رأس 2، همسایه 0 و 1 را دیده‌ایم. همسایه 5 را انتخاب می‌کنیم.
- رأس 5 را علامت‌گذاری و چاپ می‌کنیم. همسایه‌اش 2 و 1 هستند که هر دو قبلاً دیده شده‌اند. پس از اینجا بازمی‌گردیم.
- از رأس 2، همسایه 5 را هم دیده‌ایم. همسایه 3 را انتخاب می‌کنیم.
- رأس 3 را علامت‌گذاری و چاپ می‌کنیم. همسایه‌اش 2 است که قبلاً دیده شده. پس از اینجا بازمی‌گردیم.
- پیمایش به پایان می‌رسد.
- دنباله‌ی پیمایش DFS این گراف به صورت 3, 2, 0, 1, 5, 3 (بسته به ترتیب همسایگان در لیست مجاورت) است.

در شکل زیر عملکرد این الگوریتم را در یک گراف جهت‌دار ۶ رأسی مشاهده می‌کنید.

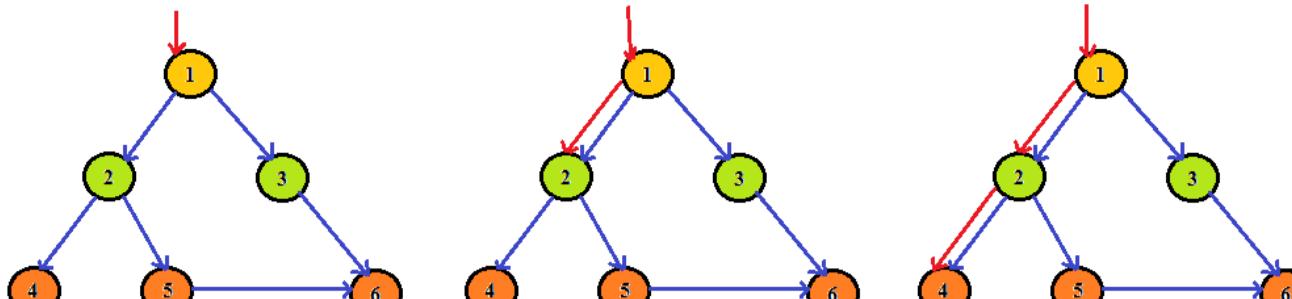


Figure - 1

Figure - 2

Figure - 3

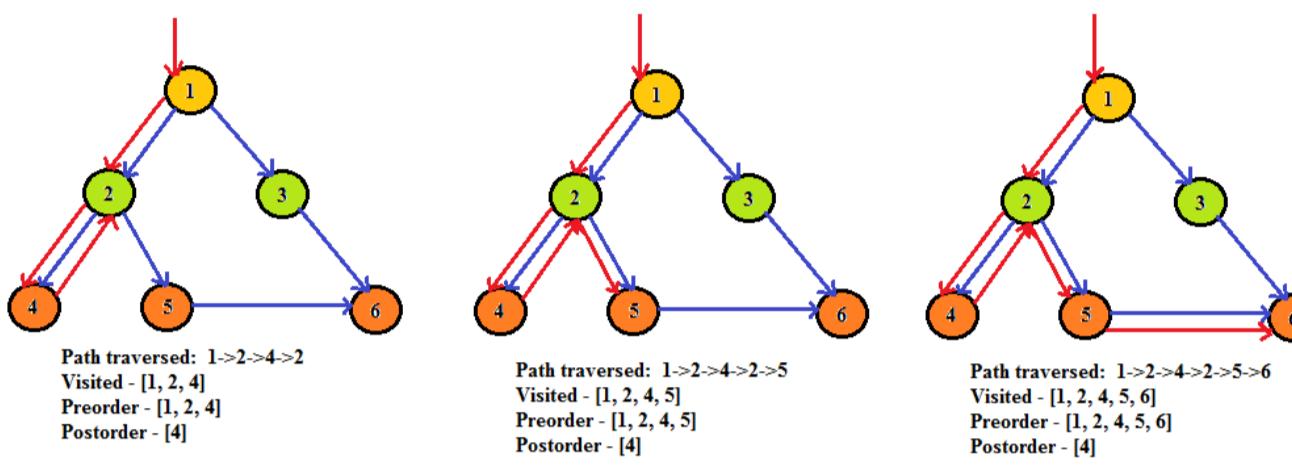


Figure - 4

Figure - 5

Figure - 6

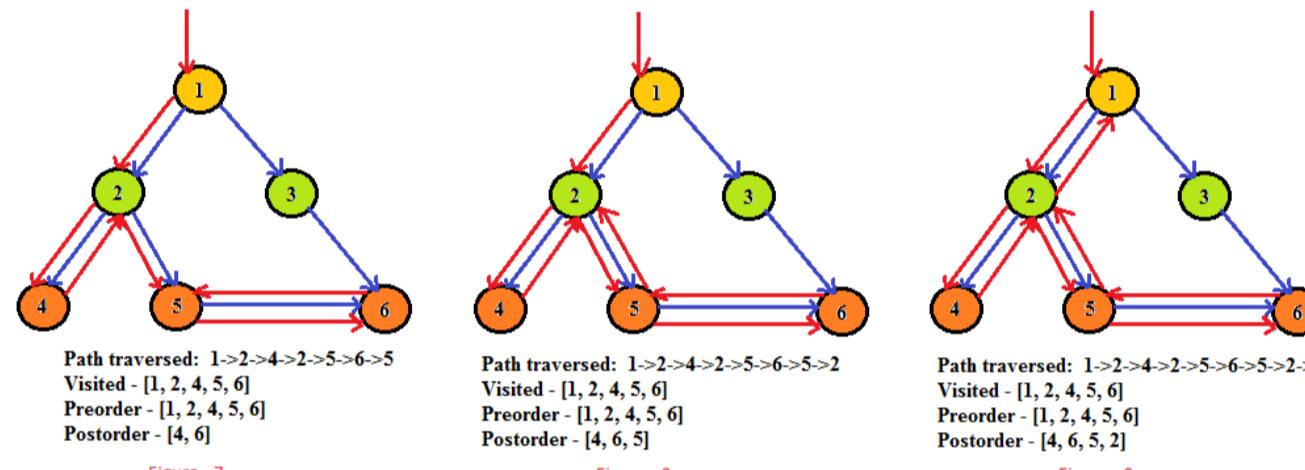


Figure - 7

Figure - 8

Figure - 9

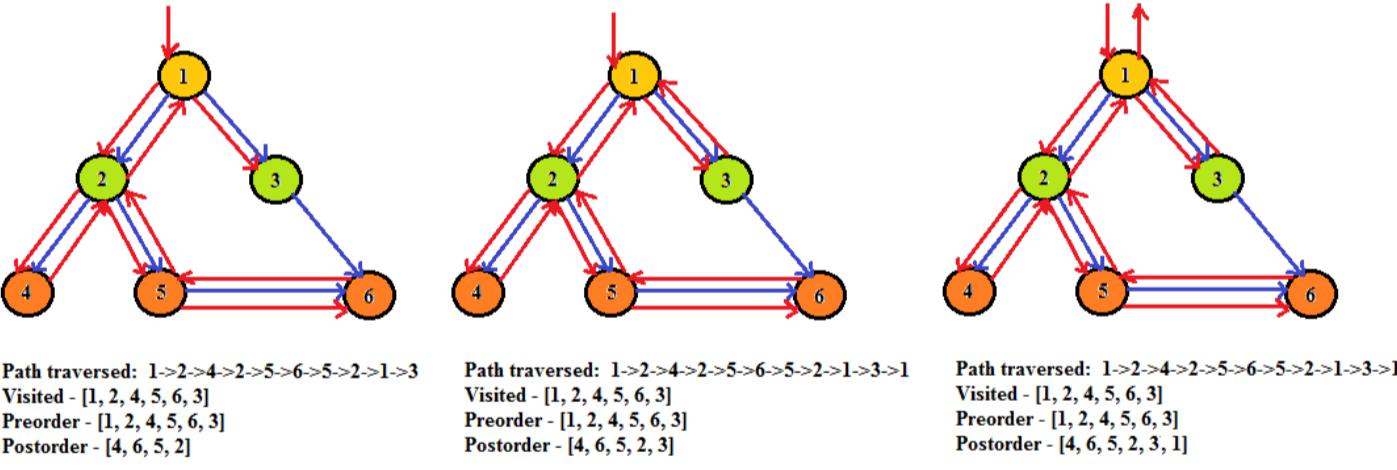


Figure - 10

Figure - 11

Figure - 12

## پیاده‌سازی

کد پیمایش اول عمق (DFS) به صورت زیر است:

In [4]: # باید قبلًا تعریف شده باشد Graph توجه: کلاس #

```
def DFS(g, s):
    """
    انجام می‌دهد s از رأس شروع g را روی گراف (DFS) این تابع پیمایش عمقدار.
    را فراخوانی می‌کند DFS_util این تابع، تابع کمکی
    """

    Args:
        g (Graph): شیء گراف.
        s (int): رأس شروع پیمایش.
    """

    # visited# بازدید نشده باشد False) لیستی برای نگهداری وضعیت بازدید هر رأس.
    # اندازه آن برابر با تعداد کل رأس‌ها در گراف است.
    num_vertices = max(g.graph.keys()) + 1 if g.graph else 0
    visited = [False] * num_vertices
```

DFS\_util(g, s, visited) فراخوانی تابع کمکی بازگشتی DFS\_util.

```
def DFS_util(g, u, visited):
    """
    (DFS) تابع کمکی بازگشتی برای پیمایش عمق اول.

    Args:
        g (Graph): شیء گراف.
        u (int): رأس فعلی که در حال بازدید است.
        visited (list): لیستی از وضعیت بازدید رأسها.

    """
    print(u) # به معنی بازدید شدن
    visited[u] = True

    چاپ رأس فعلی (به معنی بازدید شدن)
    رأس فعلی را به عنوان بازدید شده علامت‌گذاری می‌کند

    # پیمایش بر روی تمام همسایگان رأس u.
    for v in g.graph[u]:
        if not visited[v]: # اگر همسایه قبلاً بازدید نشده است v
           DFS_util(g, v, visited) # ادامه می‌دهد v را از همسایه DFS به صورت بازگشتی.
```

برروی همان گراف قبلی آن را امتحان می‌کنیم:

In [5]: # قبلاً ایجاد شده است BFS از مثال 'g' فرض می‌شود شیء گراف.

```
# g = Graph()
# g.add_edge(0, 1); g.add_edge(0, 2); g.add_edge(1, 3); g.add_edge(1, 4);
# g.add_edge(2, 4); g.add_edge(2, 5); g.add_edge(5, 1)

DFS(g, 0) # فراخوانی DFS از رأس 0
```

0  
1  
3  
4  
2  
5

مجدد سعی کنید خودتان روی کاغذ برای این گراف الگوریتم DFS را اجرا کنید و ببینید مطابق خروجی کد می‌شود یا نه.

## سوال)

این روش را همانند BFS به شیوه‌ای تغییر دهید تا برای گراف‌های غیرهمبند نیز کار کند.

\*\*پاسخ:\*\* برای پیمایش گراف‌های غیرهمبند، باید یک حلقه بیرونی اضافه کنیم که تمام رأس‌های گراف را پیمایش کند. اگر رأسی هنوز بازدید نشده بود، DFS را از آن رأس شروع کنیم.

```
def DFS_Disconnected(g):
    num_vertices = max(g.graph.keys()) + 1 if g.graph else 0
    visited = [False] * num_vertices

    for s in range(num_vertices): # پیمایش بر روی تمام رأس‌ها به عنوان نقطه شروع
        if not visited[s]: # هنوز بازدید نشده بود s اگر رأس
            print(f"--- {s} ---")
            DFS_util(g, s, visited) # از این رأس DFS_util فراخوانی
```

## تحلیل زمان اجرا

در انتها، می‌خواهیم زمان اجرای دو الگوریتم DFS و BFS را تحلیل کنیم.

در هر یک از این الگوریتم‌ها، به ازای هر رأس، همه‌ی رأس‌های متصل به آن را (چه قبلاً بررسی شده باشند چه نه) نگاه می‌کنیم. هر رأس و هر یال در گراف دقیقاً یک بار توسط الگوریتم بازدید می‌شود.

$$T = \sum_{v \in V} \left( \text{لای شزادرب منیزه } v + \sum_{u \in \text{adj}[v]} \text{سأر دیذاب منیزه } (v, u) \right)$$

پس به عبارتی می‌توان نوشت:

اولین بخش (هزینه بازدید رأس  $v$ ) مربوط به زمان لازم برای مشاهده‌ی خود رأس است. در BFS به عبارتی نمایانگر خارج کردن از صف و علامت‌گذاری، و در DFS نمایانگر فراخوانی تابع بازگشتی و علامت‌گذاری رأس شروع است. این هزینه برای هر رأس  $O(1)$  است.

مقدار  $O(1)$  داخل جمع دوم، مربوط به عملیات لازم برای هر رأس مجاور است (مثلاً اضافه کردن به صف/پیشته و بررسی `visited`).

$$T = \sum_{v \in V} O(1) + \sum_{v \in V} \sum_{u \in \text{adj}[v]} O(1) = O(|V|) + O(|E|) = O(|V| + |E|)$$

رابطه‌ی بالا را می‌توان به صورت زیر نوشت:

مقدار  $\sum_{v \in V} \sum_{u \in \text{adj}[v]}$  در واقع متناسب با مجموع درجه‌های رئوس است که در گراف‌های بدون جهت برابر  $|E|^2$  و در گراف‌های جهتدار برابر  $|E|$  (مجموع درجه‌های خروجی) است.

به عبارتی ما با استفاده از دو الگوریتم DFS و BFS می‌توانیم یک گراف را در زمان  $O(|V| + |E|)$  پیمایش کنیم. این کارایی بالا، آن‌ها را به ابزارهای بسیار مهمی در الگوریتم‌های گراف تبدیل می‌کند.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل دهم، بخش سوم: گراف‌های جهت‌دار

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

## فهرست

- ترتیب توپولوژیک
- الگوریتم ترتیب توپولوژیک
- تحلیل پیچیدگی الگوریتم ترتیب توپولوژیک
- مولفه‌های قویا همبند
- الگوریتم محاسبه مولفه‌های قویا همبند گراف
- تحلیل صحت و پیچیدگی مولفه‌های قویا همبند

## مقدمه

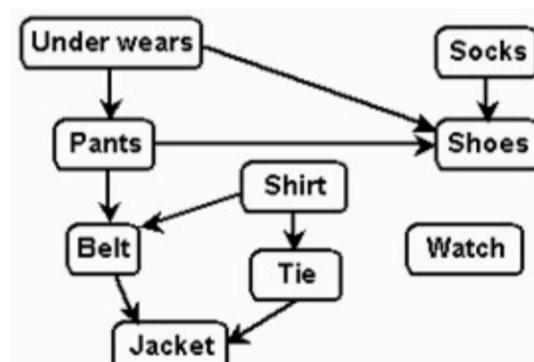
در مباحث گذشته با جستجوی عمق اول (DFS) و جستجوی سطح اول (BFS) آشنا شدیم. این دو الگوریتم، ابزارهای بنیادی برای پیمایش گراف‌ها هستند و می‌توانند برای حل طیف وسیعی از مسائل به کار روند. حال در این بخش می‌خواهیم راجع به کاربردهای مهم و جالب آن‌ها در گراف‌های جهت‌دار کمی بحث و گفت‌وگو کنیم!

## ترتیب توپولوژیک (Topological Sort)

شناخت مسئله

گراف‌های بدون دور جهت‌دار که به آن‌ها DAG (Directed Acyclic Graph) می‌گوییم، ساختارهای بسیار متداولی در علوم کامپیوتر می‌باشند. مسائلی که در آن‌ها تعدادی وظیفه داریم که این وظایف به یکدیگر وابستگی دارند (یعنی یک وظیفه پیش‌نیاز وظیفه دیگری است)، معمولاً به یک DAG مدل می‌شوند.

حال می‌خواهیم با یک مثال، شهود بیشتری نسبت به مسئله پیدا کنیم. اکبر که یک دانشجو خواب‌آلود و خسته است، صبح از خواب بلند شده است و می‌خواهد هر چه سریعتر به دانشگاه برود تا غیبت نخورد. او برای آماده شدن و پوشیدن لباس می‌داند که می‌بایست از سلسله مراتب زیر پیروی کند.



شکل ۱ : گراف وابستگی برای آماده شدن اکبر

در این گراف، یک یال میان دو رأس نشان‌دهنده رابطه پیش‌نیازی است. یعنی مثلاً اکبر ابتدا باید جوراب بپوشد و سپس کفش به پا کند.

حال اکبر می‌خواهد بداند که فرآیند آماده شدن خود را می‌بایست از کجا شروع کند و با چه ترتیبی ادامه دهد تا تمام روابط پیش‌نیازی رعایت شوند. همان‌طور که در شکل می‌بینید، گراف حاصل یک DAG است. معمولاً مدلسازی وظایف که رابطه پیش‌نیازی با یکدیگر دارند، یک DAG می‌شود. چرا که روابط علت و معلولی هستند. مثلاً این‌گونه نیست که هم جوراب پیش‌نیاز کفش باشد هم کفش پیش‌نیاز جوراب. چون در این صورت اکبر بیچاره می‌خواست کدام مورد را در ابتدا انجام دهد؟

در واقع ترتیب توپولوژیکی این DAG، یک ترتیب و نظم برای آماده شدن به اکبر می‌دهد.

\*ترتیب توپولوژیکی (Topological Sort) برای یک گراف بدون دور جهت دار (**DAG**)، یک ترتیب خطی از همه رؤوس آن است به طوری که هر گره قبل از همه گره هایی می آید که از آن به آنها یال خارج شده است. در واقع در گراف باید یک ترتیب از رأس ها داشته باشیم به گونه ای که به ازای هر یال  $(u, v)$ ، رأس  $u$  قبل از رأس  $v$  در ترتیب ظاهر شود.

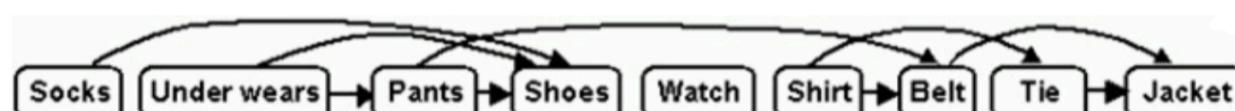
همچنین باید به این نکته توجه کرد که برای تعریف کردن یک ترتیب توپولوژیکی بر روی یک گراف، لازم است که آن گراف بدون دور (**DAG**) باشد.

\*\*پاسخ: اگر در گراف دوری وجود داشته باشد (مثلاً  $A \rightarrow B \rightarrow C \rightarrow A$ )، نمی توان یک ترتیب خطی ایجاد کرد که در آن تمام پیش نیازها رعایت شوند. در این دور،  $A$  پیش نیاز  $B$  و  $C$  پیش نیاز  $A$  است. این یک تناقض است و امکان قرار دادن تمام رأس ها در یک ترتیب خطی که این روابط را حفظ کند، وجود ندارد.

گراف  $G$  یک DAG است اگر و تنها اگر  $G$  ترتیب توپولوژیکی داشته باشد.

طبعی الگوریتم: ابتدا سعی کنیم خودمان ترتیب توپولوژیکی گراف اکبر را بکشیم.

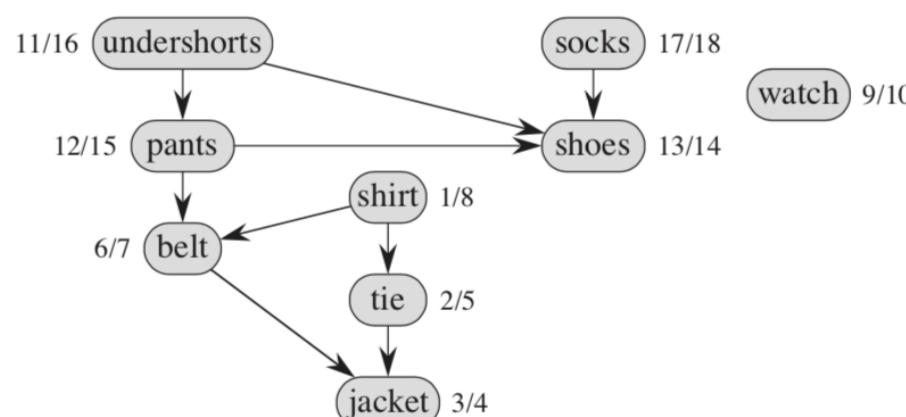
\*\*پاسخ: خیر، لزوماً فقط یک ترتیب توپولوژیکی وجود ندارد. اگر چندین رأس وجود داشته باشند که هیچ پیش نیازی ندارند (یا تمام پیش نیازها براورد شده است)، می توان آنها را به هر ترتیبی در ابتدا قرار داد. به عنوان مثال، در گراف اکبر، "پیزماه" و "جوراب" هر دو می توانند در ابتدا قرار گیرند.



شکل ۲: یک ترتیب توپولوژیکی از گراف اکبر

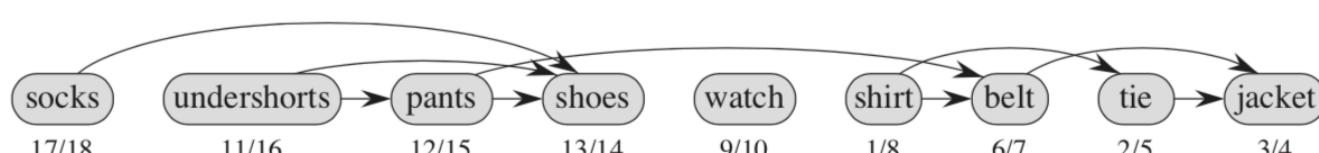
حال آیا به نظرتان شهود خاصی از این ترتیب می توان دریافت؟

بایاید یکبار الگوریتم **DFS** را روی این DAG اجرا کنیم. برای هر رأس زمان **start** (زمان شروع بازدید) و **finish** (زمان پایان بازدید) را هم یادداشت می کنیم.



شکل ۳: زمان سمت چپ، زمان شروع و زمان سمت راست، زمان پایان در DFS است

طبعی در ترتیب توپولوژیک داریم:



شکل ۴: ترتیب توپولوژیک به همراه زمان شروع و خاتمه در DFS

حال اگر به **finish time** رؤوس در ترتیب توپولوژیک دقت کنیم، می فهمیم که یک ترتیب نزولی دارند. پس به نظر یک ایده خوب بیدا کرده ایم!

الگوریتم ترتیب توپولوژیک

۱. الگوریتم DFS را روی گراف اجرا می کنیم و برای هر رأس زمان **finish** را محاسبه می کنیم.

2. هر رأسی که finished شد (یعنی تمام همسایگانش و زیردرخت‌هایشان بازدید شدند)، آن را به ابتدای لیست (یا پشته) نتیجه اضافه می‌کنیم.

3. در انتهای، لیست بوجود آمده جواب مسئله است.

برای آشنایی بیشتر به صورت عملی و گرافیکی با مرتبسازی توپولوژیکی به این [لینک](#) مراجعه کنید و حالت‌های مختلف را تست کنید.

برای مطالعه بیشتر:

بررسی و اثبات الگوریتم

طبق تعریف، باید بررسی کنیم به ازای هر یال  $E \in (v, u)$  (یالی از  $u$  به  $v$ ، رأس  $u$  قبل از رأس  $v$  در الگوریتم ما ظاهر شود. پس کافی است نشان دهیم که زمان خاتمه  $u$  بیشتر از  $v$  است. (چرا که در الگوریتم ما در این حالت است که  $u$  قبل از  $v$  در لیست نهایی ظاهر می‌شود).

مبحث Edge Classification (دسته‌بندی یال‌ها در DFS) را به خاطر بیاورید. در آنجا به این نکته پی بردم که برای یال‌های Cross Edge و Tree Edge و Forward Edge، این خاصیت (زمان پایان پدر < زمان پایان فرزند) برقرار است و تنها برای یال Back Edge این خاصیت برقرار نیست. حال آنکه می‌دانیم اصلاً Back Edge نمی‌تواند موجود باشد، زیرا اگر موجود باشد با بدون دور بودن گراف (DAG بودن) در تناقض است.

چرا وجود DAG در Back Edge\*\*

\*\*پاسخ: یک Back Edge (یال برگشتی) یالی است که از یک رأس  $u$  به یک رأس  $v$  می‌رود، در حالی که  $v$  جد  $u$  در درخت DFS است. وجود چنین یالی به معنای وجود یک دور در گراف است (از  $v$  به  $u$  در درخت DFS و سپس از  $u$  به  $v$  با یال برگشتی). از آنجایی که DAG به تعریف گرافی بدون دور است، پس نمی‌تواند شامل Back Edge باشد.

پس درستی الگوریتم ثابت شد.

تحلیل پیچیدگی زمانی الگوریتم زمان لازم برای اجرای هر خط را بررسی می‌کنیم:

1. برای اجرای DFS (که شامل پیمایش تمام رأس‌ها و یال‌ها است) به  $O(|V| + |E|)$  زمان نیاز داریم.
2. در زمان  $O(1)$  می‌توانیم عناصر را به ابتدای لیست (یا پشته) اضافه کنیم. این عملیات برای هر رأس یک بار انجام می‌شود، پس  $O(|V|)$  در کل.

پس در نهایت پیچیدگی زمانی الگوریتم از مرتبه  $O(|V| + |E|)$  می‌باشد که خطی است.

In [2]:

```
from collections import defaultdict

class Graph:
    def __init__(self, N):
        # رأس مقداردهی اولیه می‌کند N متدهای سازنده: یک گراف جهت‌دار را با
        # به صورت خودکار لیست‌های خالی ایجاد می‌کند. defaultdict برای ذخیره یال‌ها
        self.graph = defaultdict(list)
        self.Vertices_num = N # تعداد کل رأس‌ها در گراف
        self.Answer_list = [] # لیستی برای ذخیره ترتیب توپولوژیکی

    def add_Edge(self, u, v):
        # به u اضافه کردن یک یال جهت‌دار از v.
        if u in self.graph:
            self.graph[u].append(v)
        else:
            self.graph[u] = [v]

    def DFS(self, v, visited):
        # برای پیمایش عمق‌اول DFS تابع کمکی
        # این تابع زمان پایان رأس‌ها را به صورت ضمنی برای ساخت ترتیب توپولوژیکی استفاده می‌کند
        # را به عنوان بازدید شده علامت‌گذاری می‌کند v رأس
        visited[v] = True
        # پیمایش بر روی همسایگان رأس v.
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFS(i, visited)

    def TopologicalSort(self):
        # این تابع ترتیب توپولوژیکی گراف را محاسبه و بر می‌گرداند
        # لیستی برای پیگیری رأس‌های بازدید شده
        visited = [False for _ in range(self.Vertices_num)]
        # اطمینان از خالی بودن لیست پاسخ قبل از شروع
        self.Answer_list = []
        # پیمایش بر روی تمام رأس‌ها. این حلقه برای گراف‌های غیرهمبند ضروری است
        #
```

```

for i in range(self.Vertices_num):
    if visited[i] == False: # اگر رأس
        self.DFS(i, visited) # DFS
    else:
        continue

return self.Answer_list # لیست ترتیب توپولوژیکی.

# مثال استفاده از Topological Sort
DAG = Graph(8) # ایجاد یک DAG با 8 رأس (0 تا 7)
DAG.add_Edge(0, 3)
DAG.add_Edge(1, 3)
DAG.add_Edge(1, 4)
DAG.add_Edge(2, 4)
DAG.add_Edge(2, 7)
DAG.add_Edge(3, 5)
DAG.add_Edge(3, 6)
DAG.add_Edge(3, 7)
DAG.add_Edge(4, 6)

print("ترتیب توپولوژیکی گراف:", DAG.TopologicalSort())

```

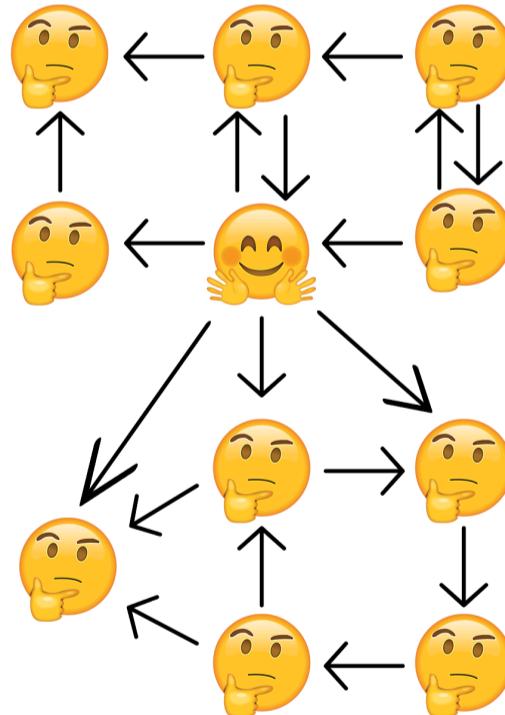
ترتیب توپولوژیکی گراف: [5, 6, 7, 3, 0, 4, 1, 2]

## Strongly Connected Components مولفه‌های قویاً همبند

در این بخش می‌خواهیم با مسئله «مولفه‌های قویا همبند» آشنا شویم. بباید از یک مثال شروع کنیم. تصور کنید که یک نرم‌افزار شبکه‌ی اجتماعی داریم. اعضای این نرم‌افزار با هم مطالبی را به اشتراک می‌گذارند و هر کس می‌تواند تعدادی عضو دیگر را فالو کند.

می‌دانیم که رابطه‌ی افرادی که با هم دوستند به صورت یک رابطه‌ی دوطرفه است. یعنی اگر اکبر و اصغر دو دوست باشند و اکبر اصغر را فالو کند، در صورت از دماغ فیل نیفتادن اصغر، او هم اکبر را فالو می‌کند. حال ما به عنوان مدیران این شبکه‌ی اجتماعی به دنبال این هستیم که گروه‌ها و جوامع مختلف را پیدا کنیم. در صورتی که بتوانیم این کار را به خوبی انجام دهیم، می‌توانیم قابلیت‌هایی مثل پیشنهاد کردن دوست جدید (برای فالو)، تبلیغات هدفمند، پیشنهاد کردن پست‌های مربوط به علایق هر فرد را به نرم‌افزار خود اضافه کنیم و پارو کردن پول بپردازیم. 😊

بباید تا این مسئله را به صورت دقیقتی بیان کنیم. در شبکه‌های اجتماعی، هر فرد یک رأس در گراف روابط و هر رابطه‌ی فالو کردن یک یال جهت‌دار از فالوکنده به فالوشنونده است. هر جامعه را نیز به این شکل تعریف می‌کنیم که از هر عضو جامعه مسیری به همه‌ی اعضای آن وجود داشته باشد. یعنی دو نفر در صورتی در یک جامعه هستند که یا با هم دوست باشند یا هر دو بتوانند از طریق تعدادی دوست با هم آشنا شوند. برای مثال به گراف روابط زیر دقت کنید:

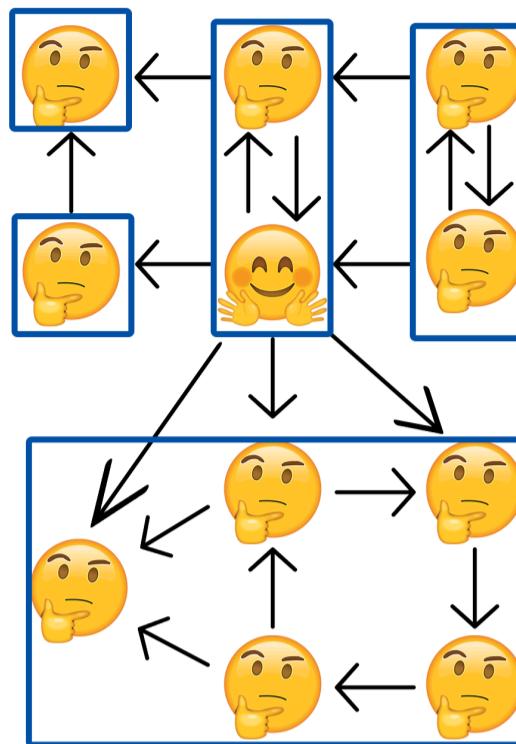


به نظر شما در شبکه‌ی اجتماعی بالا چه جوامعی وجود دارد؟

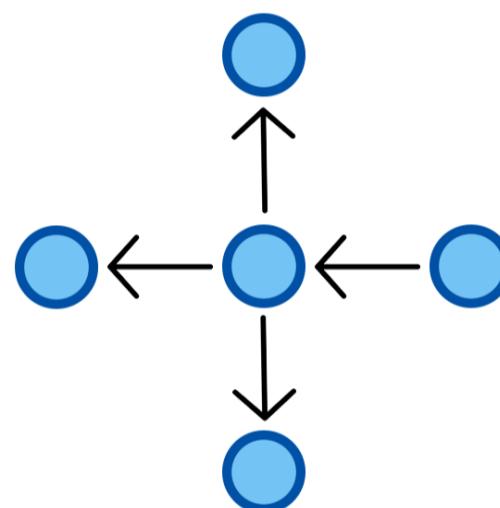
برای پاسخ به این سوال، بار دیگر به تعریف \*\*مولفه‌های قویاً همبند (SCCs)\*\* مراجعه می‌کنیم. هر مولفه‌ی قویاً همبند، زیرمجموعه‌ای از رئوس یک گراف جهت‌دار است که به ازای هر دو رأس  $w$  و  $v$  در آن زیرمجموعه، هم از  $w$  به  $v$  مسیری وجود داشته باشد و هم از  $v$  به  $w$ . با توجه به این تعریف، می‌توانیم نتیجه بگیریم که اگر به جای تمامی رئوس یک مولفه، یک رأس را جایگذاری کنیم، گراف حاصل DAG خواهد بود.

\*\*چرا؟

پاسخ: \*\*اگر گراف حاصل (که هر SCC را به یک رأس تبدیل کرده‌ایم) دارای دور باشد، این بدان معناست که بین دو  $SCC$ , مثلاً  $C_1$  و  $C_2$ , یال‌هایی وجود دارد که یک دور را تشکیل می‌دهند (مثالاً  $C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_1$ ). این به این معنی است که رأس‌هایی از  $C_1$  می‌توانند به رأس‌هایی از  $C_2$  برسند و برعکس. اما طبق تعریف SCC، اگر دو رأس به هم مسیر دوطرفه داشته باشند، باید در یک SCC باشند. بنابراین، اگر دور بین SCC‌ها وجود داشته باشد، تعریف SCC نقض می‌شود. پس گراف حاصل از فشرده‌سازی SCC‌ها به یک رأس، حتماً یک DAG خواهد بود.



DAG مربوط به این گراف در شکل زیر آمده است:



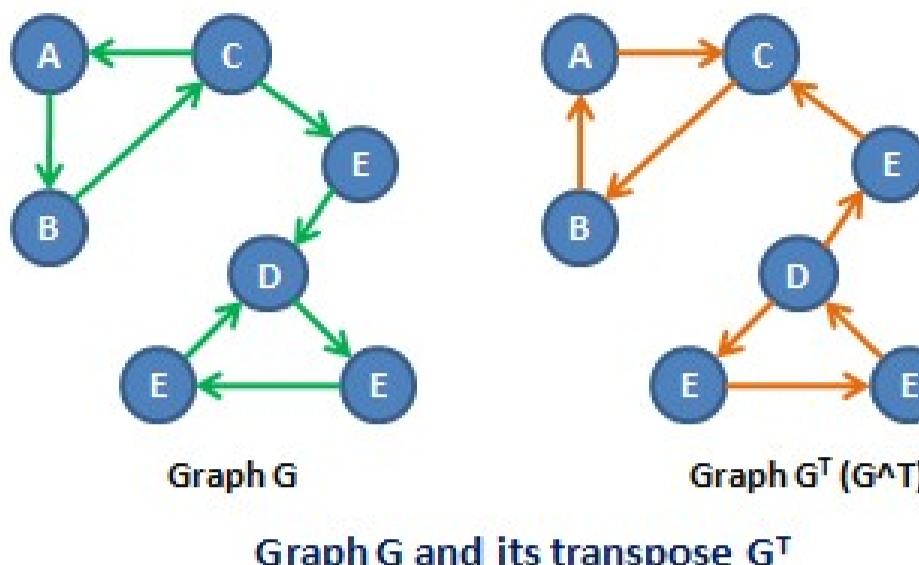
#### الگوریتم محاسبه مولفه‌های قویا همبند گراف

پس از آشنایی با مسئله پیدا کردن مولفه‌های قویا همبند گراف، حال می‌خواهیم الگوریتمی برای محاسبه این مولفه‌ها ارائه کنیم. پیش از ارائه الگوریتم لازم است تا با مفهوم گراف  $G^T$  آشنا شویم.

**تعریف:** اگر جهت تمام یال‌های گراف جهتدار  $G$  را برعکس کنیم، گراف حاصل گراف جهتدار  $G^T$  یا \*\*گراف ترانهاده (Transpose Graph)\*\* به دست خواهد آمد.

به گراف زیر و گراف ترانهاده‌اش کمی دقت کنید. آیا مولفه‌های همبندی این دو گراف یکسان است؟ چرا؟

پاسخ:\*\* بله، مولفه‌های قویاً همبند یک گراف  $G$  و گراف ترانهاده‌اش  $G^T$  یکسان\*\* هستند. این به این دلیل است که اگر مسیری از  $u$  به  $v$  در  $G$  وجود داشته باشد، آنگاه مسیری از  $v$  به  $u$  در  $G^T$  وجود دارد (با یال‌های برعکس). بنابراین، اگر  $u$  و  $v$  در  $G$  قویاً همبند باشند (یعنی مسیر  $v \rightarrow u$  و  $u \rightarrow v$  وجود داشته باشد)، همین مسیرها با جهت‌های برعکس در  $G^T$  نیز وجود دارند و  $u$  و  $v$  در  $G^T$  نیز قویاً همبند خواهند بود.



حالا با توجه به این نکته که مولفه‌های همبندی هر گراف و ترانهاده‌ش یکسانند، به ارائه الگوریتم محاسبه مولفه‌های قویاً همبند می‌پردازیم. این الگوریتم به Kosaraju's Algorithm (کوساراجو) معروف است.

الگوریتم به طور کلی در سه مرحله انجام می‌شود:

.1

ابتدا یک پشته خالی در نظر می‌گیریم. گراف جهت‌دار داده شده را،  $G$  در نظر می‌گیریم. این گراف را به وسیله DFS (جستجوی عمق اول) پیمایش می‌کنیم. وقتی الگوریتم جستجوی عمق اول کار خود را با یک رأس به اتمام رساند (یعنی تمام همسایگان و زیردرخت‌های آن بازدید شدند)، آن رأس را در داخل پشته قرار می‌دهیم. به همین ترتیب ادامه می‌دهیم تا کل گراف پیمایش شود. در این مرحله، رأس‌هایی که زمان پایان DFS بالاتری دارند، در بالای پشته قرار می‌گیرند.

.2 در این مرحله، تمامی رأس‌ها در داخل پشته قرار دارند.

.3

سپس گراف معکوس  $G$  یعنی  $G^T$  را به دست می‌آوریم. این بار گراف معکوس را با توجه به اولویت رأس‌ها در پشته، پیمایش می‌کنیم. به این ترتیب که در ابتدا از رأسی شروع می‌کنیم که در بالای پشته قرار دارد. جستجوی عمق اول را از آن رأس (در  $G^T$ ) شروع می‌کنیم. فرض کنید در رأسی به نام  $w$  به پایان برسد. تمام رأس‌هایی که در این بین ملاقات شده‌اند، جزو یک مؤلفه قویاً همبند هستند. تمام رأس‌های ملاقات شده را از پشته خارج کرده و همین الگوریتم را بر روی رأسی که در بالای پشته قرار دارد، تکرار می‌کنیم، تا زمانی که پشته خالی شود.

در ادامه پیاده‌سازی‌ای از این الگوریتم را مشاهده می‌کنید. به تابع `printSCCs` دقت کنید.

In [3]:

```
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        متد سازنده: یک گراف جهت‌دار را با تعداد رأس‌های مشخص شده مقداردهی اولیه می‌کند
        # تعداد رأس‌ها
        self.V = vertices
        self.graph = defaultdict(list)
        لیست مجاورت برای ذخیره یال‌ها

    def addEdge(self, u, v):
        # v به u اضافه کردن یک یال جهت‌دار از
        self.graph[u].append(v)

    def DFSUtil(self, v, visited):
        برای پیمایش عمق اول و چاپ رأس‌های بازدید شده DFS تابع کمکی
        # را به عنوان بازدید شده علامت‌گذاری می‌کند v رأس
        visited[v] = True
        چاپ رأس فعلی # print(v, end=" ")
        # پیمایش بر روی همسایگان رأس
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)
                ادامه می‌دهد i را از همسایه DFS به صورت بازگشتی

    def fillOrder(self, v, visited, stack):
        در پشته قرار می‌دهد DFS را اجرا می‌کند و رأس‌ها را بر اساس زمان پایان DFS این تابع
        # را به عنوان بازدید شده علامت‌گذاری می‌کند v رأس
        visited[v] = True
        # پیمایش بر روی همسایگان رأس
        for i in self.graph[v]:
            if visited[i] == False:
                self.fillOrder(i, visited, stack)
                ادامه می‌دهد i را از همسایه DFS به صورت بازگشتی
```

و تمام زیردرختش، آن را به پشته اضافه می‌کند  $v$  پس از اتمام بازدید از رأس #.

```
def getTranspose(self):
    # را برمی‌گرداند ( $G^T$ ) این تابع گراف ترانهاده.
    g = Graph(self.V)
    # ایجاد یک گراف جدید با همان تعداد رأس‌ها.
    p = []
    for i in self.graph:
        for j in self.graph[i]:
            g.addEdge(j, i) # i به j از اضافه کردن یال معکوس #.
    return g

def printSCCs(self):
    # گراف را پیدا کرده و چاپ می‌کند (SCCs) این تابع مولفه‌های قویاً همبند.
    stack = [] # پشته برای ذخیره رأس‌ها بر اساس زمان پایان.
    # اول و پر کردن پشته DFS مرحله 1: اجرای.
    visited = [False] * (self.V)
    for i in range(self.V):
        if visited[i] == False:
            self.fillOrder(i, visited, stack)

    # مرحله 2: به دست آوردن گراف ترانهاده.
    gr = self.getTranspose()

    # دوم روی گراف ترانهاده با ترتیب از پشته DFS مرحله 3: اجرای.
    visited = [False] * (self.V)
    # تا زمانی که پشته خالی نباشد.
    while stack:
        i = stack.pop() # رأس را از بالای پشته برمی‌دارد.
        if visited[i] == False: # (جديد SCC یعنی یک) اگر رأس هنوز بازدید نشده است.
            visited[i] = True # را چاپ می‌کند SCC را از این رأس در گراف ترانهاده شروع می‌کند و DFSUtil(i, visited) # DFS برای جداسازی SCC‌ها.
            print()

# برای یافتن Kosaraju SCCs مثال استفاده از الگوریتم
g = Graph(5) # ایجاد یک گراف با 5 رأس (0 تا 4).
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 1)
g.addEdge(0, 3)
g.addEdge(3, 4)

print("مولفه‌های قویاً همبند در گراف داده شده")
g.printSCCs()
```

مولفه‌های قویاً همبند در گراف داده شده:

0 1 2  
3  
4

### تحلیل صحت و پیچیدگی

الگوریتم ذکر شده دوبار گراف را به وسیله‌ی DFS پیمایش می‌کند. که هر بار از  $O(|V| + |E|)$  هزینه زمانی صرف می‌شود. برای معکوس کردن گراف نیز کافی است یک بار گراف را پیمایش کنیم و هر یال را برعکس کنیم، که این نیز  $O(|V| + |E|)$  زمان می‌برد.

پس در مجموع، الگوریتم از نظر پیچیدگی نسبت به اندازه ورودی خطی عمل می‌کند و مرتبه زمانی آن  $O(|V| + |E|)$  است.

پیش از اثبات صحت الگوریتم، باید با یک لم آشنا شویم.

لم: می‌دانیم بین دو مولفه قویاً همبند در یک گراف، حداقل یک یال می‌تواند وجود داشته باشد (اگر بیش از یک یال وجود داشته باشد، آنگاه آن دو مولفه خود بخشی از یک مولفه بزرگتر خواهند بود).

همچنین توجه داشته باشید که  $f(v)$  زمان پایان اجرای الگوریتم 'DFS' بر روی رأس  $v$  را نشان می‌دهد.

حال دو مولفه‌ی قویاً همبند مانند  $C_1$  و  $C_2$  را در نظر بگیرید. این لم بیان می‌کند اگر از  $C_2$  یالی وجود داشته باشد، آنگاه:

$$\max_{v \in C_1} f(v) > \max_{w \in C_2} f(w)$$

سعی کنید این لم را اثبات کنید.

اثبات صحت الگوریتم:

از این لم نتیجه می‌شود که در اولین اجرای الگوریتم (که زمان‌های پایان  $u$ )  $f$  محاسبه می‌شوند و رأس‌ها در پشته قرار می‌گیرند، رأسی که بیشینه  $(u)$  را در کل گراف دارد، در بالای پشته قرار می‌گیرد. این رأس (مثلاً  $u$ ) حتماً متعلق به یک SCC است.

در مرحله دوم، در  $G^T$ ، DFS را از رأسی که بیشینه  $(u)$  را در گراف  $G$  دارد، شروع می‌کنیم. فرض کنید این رأس  $u$  است و مولفه‌ای که  $u$  در آن وجود دارد را  $C_u$  بنامیم.

الگوریتم جستجوی عمق اول در این مرحله، ابتدا تمامی همسایگان  $u$  را در  $C_u$  ملاقات می‌کند. حال ادعا می‌کنیم که جستجوی عمق اول در این مرحله به پایان می‌رسد و از  $C_u$  خارج نمی‌شود. چون در غیر این صورت، باید از  $C_u$  به یکی دیگر از مؤلفه‌ها مانند  $C_i$  یال وجود داشته باشد. در حالیکه از  $C_u$  به هیچ مؤلفه دیگری مانند  $C_i$ ، یال وجود ندارد (در  $G^T$ ). زیرا اگر یالی از یک رأس در  $C_u$  به یک رأس در مؤلفه دیگر مانند  $C_i$  در  $G^T$  وجود داشت، به معنای وجود یالی از  $C_i$  به  $C_u$  در گراف اصلی  $G$  است. طبق لم بالا،  $\max_{w \in C_i} f(w) > \max_{v \in C_u} f(v)$  (که در بالای پشته قرار دارد) تناقض است.

بنابراین، الگوریتم به درستی اولین مؤلفه قویاً همبند را مشخص می‌کند. در مراحل بعدی، رأسی که در بالای پشته قرار دارد (و هنوز بازدید نشده) انتخاب می‌شود و همین روند تکرار می‌شود تا تمام SCC‌ها شناسایی شوند.

### تحلیل پیچیدگی زمانی الگوریتم

الگوریتم ذکر شده دوبار گراف را به وسیله DFS پیمایش می‌کند. که هر بار از  $O(|V| + |E|)$  هزینه زمانی صرف می‌شود. برای معکوس کردن گراف نیز کافیست یک بار گراف را پیمایش کنیم و هر یال را برعکس کنیم، که این نیز  $O(|V| + |E|)$  زمان می‌برد.

پس در مجموع، الگوریتم از نظر پیچیدگی نسبت به اندازه ورودی خطی عمل می‌کند و مرتبه زمانی آن  $O(|V| + |E|)$  است.

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل دهم، بخش چهارم: یافتن کوتاه‌ترین مسیر در گراف وزن‌دار

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

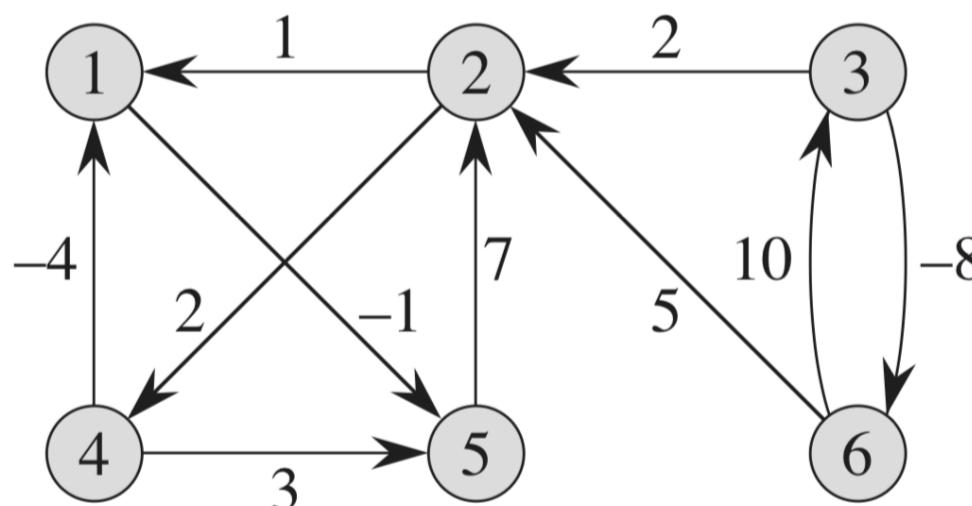
- تعريف مسئله
- تعريف relaxation در گراف
- الگوریتم Dijkstra
- الگوریتم Bellman-Ford

## مسئله یافتن کوتاه‌ترین مسیر در گراف

در این بخش، به یکی از مسائل کلاسیک و مهم در نظریه گراف و علوم کامپیوتر می‌پردازیم: \*یافتن کوتاه‌ترین مسیر در گراف‌های وزن‌دار\*\*.

یک گراف وزن‌دار و جهت‌دار  $G(V, E)$  (که  $V$  مجموعه رأس‌ها و  $E$  مجموعه یال‌ها است) داده شده است. وزن‌های گراف از نگاشت مقادیر تابع  $w : E \rightarrow \mathbb{R}$  به یال‌های گراف بدست می‌آیند. این وزن‌ها می‌توانند نشان‌دهنده فاصله، زمان، هزینه، یا هر معیار دیگری باشند.

به عنوان مثال، گراف زیر نمونه‌ای از گراف مورد نظر می‌باشد:



$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

وزن یک مسیر دلخواه مانند  $p$  در گراف که شامل رأس‌های  $p = \$$  می‌باشد، به صورت جمع وزن‌هایی که در مسیر می‌باشند تعریف می‌شود:

به عنوان مثال، وزن مسیر  $< 3, 6, 2, 4, 5 >$  در گراف بالا برابر با ۲ می‌باشد.

حال کوتاه‌ترین مسیر از رأس  $u$  به رأس  $v$  در گراف برابر است با مسیری از  $u$  به  $v$  که در بین تمام مسیرهای ممکن بین این دو رأس، کمترین وزن را داشته باشد که وزن آن را با نماد  $(v, u)\delta$  نمایش می‌دهیم.

به عنوان مثال، کوتاه‌ترین مسیر از رأس ۶ به رأس ۱ در گراف بالا، مسیر  $< 1, 2, 4, 6 >$  می‌باشد که وزن آن برابر با  $3 = (6, 1)\delta$  است.

در این دفترچه قصد داریم الگوریتم‌هایی برای پیدا کردن کوتاه‌ترین مسیر از یک رأس مبدأ دلخواه مانند  $s$  به سایر رؤوس گراف بیان کنیم که با عنوان \*\*مسئله کوتاه‌ترین مسیر تک‌مبدأ (Single-Source Shortest)\*\* شناخته می‌شود.

In [1]: `import math # برای استفاده از math.inf (بینهایت)`

```
class Node:  
    def __init__(self, num):
```

مقداردهی می‌کند  $d$  متند سازنده: یک گره (رأس) را با شماره و مقدار اولیه #

شماره یا شناسه رأس;"> self.num = num #direction:rtl;">

```

تخمین کوتاهترین فاصله از مبدأ (بعداً توضیح داده می‌شود)>">
گره پدر در مسیر کوتاهترین (برای بازسازی مسیر)>">

def __str__(self):
    # <span style="direction:rtl;">نمايش رشته‌ای گره برای چاپ</span>
    return f"Node({self.num}, d={self.d:.2f})"

class Graph:
    def __init__(self, nodes):
        متد سازنده: یک گراف را با استفاده از ماتریس مجاورت (برای وزن‌ها) مقداردهی می‌کند.
        که رأس‌های گراف را نشان می‌دهند Node لیستی از اشیاء>">
        self.V = nodes # direction:rtl;">
        # self.E: 0 است
        ماتریس مجاورت برای ذخیره وزن یال‌ها. 0 به معنی عدم وجود یال یا وزن 0 است.
        # فرض می‌شود وزن یال‌ها غیر صفر است (در صورت عدم وجود یال، از مقدار بینهایت استفاده می‌شود)
        self.E = [[0 for _ in range(len(nodes))] for _ in range(len(nodes))]] # استفاده شود تا عدم وجود یال از math.inf 0 بیشتر است
        # self.E = [[math.inf for _ in range(len(nodes))] for _ in range(len(nodes))]] # وزن 0 در نظر گرفته شود (v به v) برای یال‌های خود به خود>">
        # for i in range(len(nodes)): self.E[i][i] = 0

    def addEdge(self, u_num, v_num, w):
        # با وزن v به رأس u اضافه کردن یک یال جهت‌دار از رأس
        # شماره رأس‌ها هستند
        if u_num < 0 or u_num >= len(self.V) or v_num < 0 or v_num >= len(self.V):
            raise ValueError("Vertices out of bounds.")
        self.E[u_num][v_num] = w

```

## آرام‌سازی Relaxation

الگوریتم‌هایی که در ادامه بیان خواهیم کرد، از تکنیک آرام‌سازی (Relaxation)\*\* یال‌ها طی اجرای الگوریتم استفاده می‌کنند و در واقع تفاوتشان در تعداد و ترتیب استفاده از آن است.

در ابتدا، برچسبی بر روی هر رأس  $V \in V$  می‌گذاریم که نشان‌دهنده تخمین کوتاهترین فاصله از رأس مبدأ ( $s$ ) به آن رأس می‌باشد و آن را با  $d_v$  نمایش می‌دهیم.

مقدار ابتدایی  $d_v$  برای هر رأس  $\{s\} - V$  برابر با  $\infty$  (بینهایت) و برای  $s$  (رأس مبدأ) مقدار آن را صفر قرار می‌دهیم. این مرحله\*\* مقداردهی اولیه (Initialization) نام دارد.

In [2]: `import math # برای استفاده از math.inf`

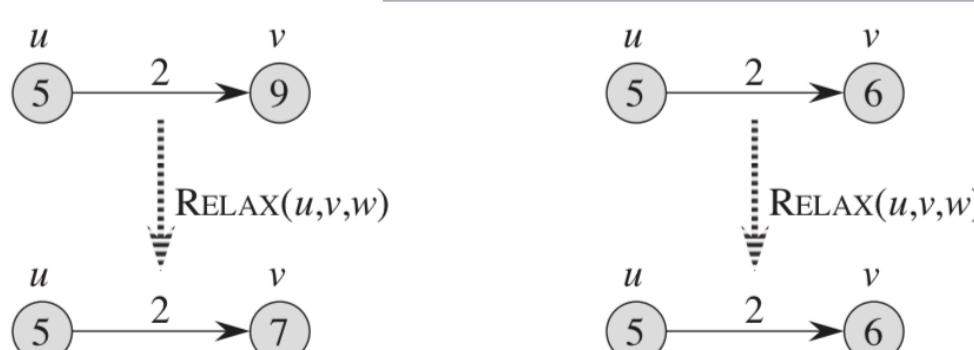
```

def initialize_single_source(G, s_node):
    """
    برای همه رأس‌ها را مقداردهی اولیه می‌کند (d) این تابع تخمین فاصله
    Args:
        G (Graph): شیء گراف.
        s_node (Node): گره مبدأ.
    """
    for v_node in G.V: # بیمایش بر روی تمام گره‌های گراف.
        تخمین فاصله برای همه گره‌ها را بینهایت قرار می‌دهد # پر گره‌ها را None می‌داند
        v_node.d = math.inf
        v_node.parent = None # پر گره‌ها را None می‌داند
        تخمین فاصله برای گره مبدأ را 0 قرار می‌دهد.
    s_node.d = 0 # قرار می‌دهد

```

حال در هر مرحله از اجرای الگوریتم، آرام‌سازی (relax)\*\* یال ( $v, u$ ) بدين معناست که چک کنیم آیا می‌توانیم تخمین کوتاهترین فاصله به  $v$  را با گذرکردن از رأس  $u$  و در نتیجه یال ( $v, u$ ) بهتر (کمتر) کنیم یا نه. اگر توانستیم، مقدار  $d_v$  را بهروزرسانی (update) می‌کنیم.

در هر مرحله از آرام‌سازی یک یال، ممکن است مقدار تخمین کوتاهترین فاصله به  $v$  ( $v.d$ ) کاهش یابد و یا تغییری نکند.



In [3]: `def relax(G, u, v):
 if v.d > u.d + G.E[u.num][v.num]:
 v.d = u.d + G.E[u.num][v.num]`

## الگوریتم Dijkstra (دایکسترا)

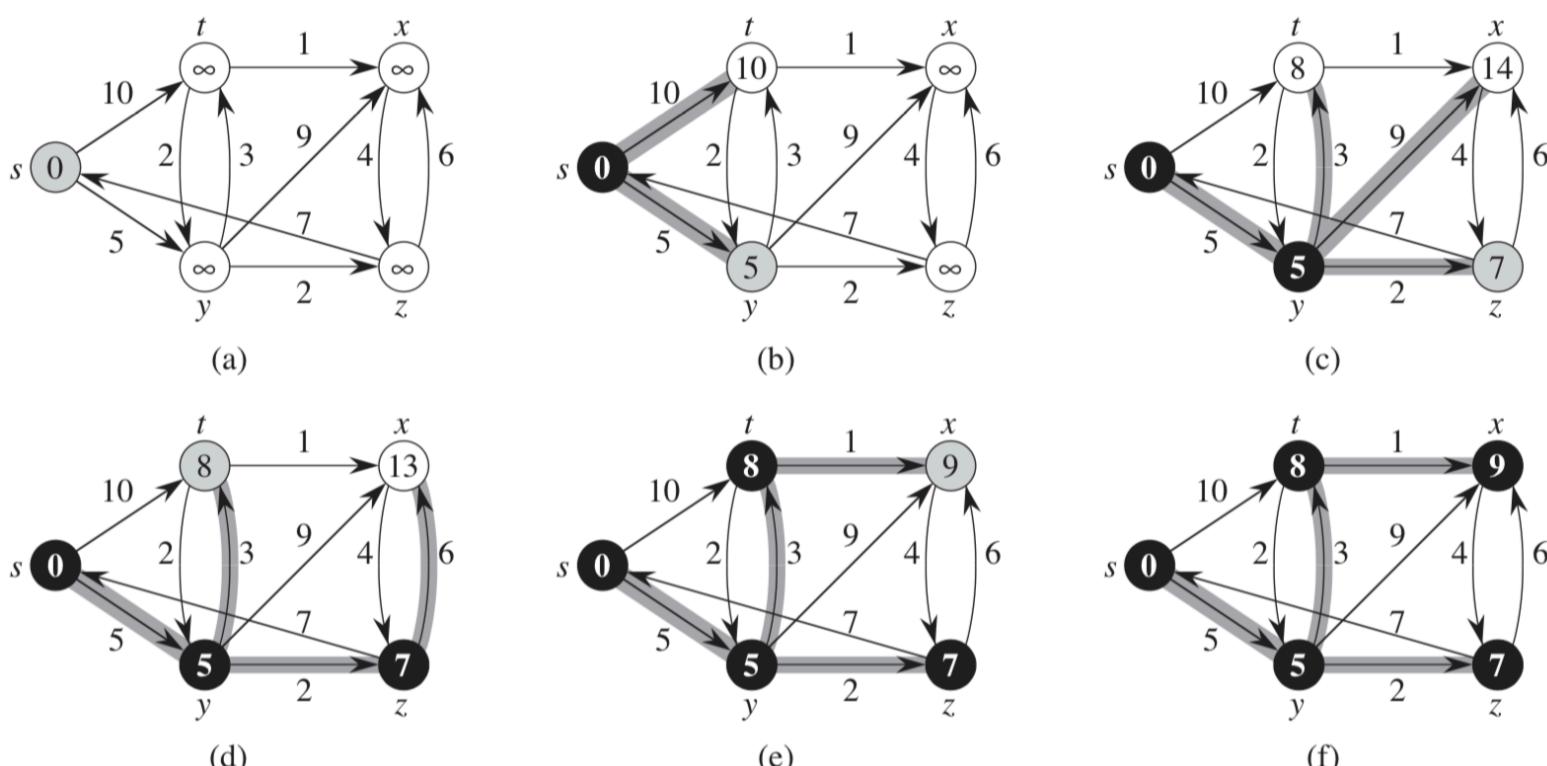
الگوریتم دایکسترا (Dijkstra's Algorithm)\*\* یکی از معروف‌ترین الگوریتم‌ها برای یافتن کوتاهترین مسیر از یک مبدأ واحد به تمامی رئوس دیگر در یک گراف وزن‌دار است، به شرطی که \*وزن یال‌ها منفی نباشد.\*

برای اجرای الگوریتم، مجموعه‌ای به نام  $S$  را برابر با رأس‌هایی در نظر می‌گیریم که وزن کوتاهترین مسیر از  $s$  (مبدأ) به آنها را یافته‌ایم. این مجموعه در ابتدا تهی می‌باشد.

حال در هر مرحله، رأس  $u$  از رأس‌های  $S - V$  (رأس‌هایی که در  $S$  نیستند) را که دارای کمترین مقدار  $d_u$  (تخمین کوتاهترین فاصله) می‌باشد، انتخاب کرده و به  $S$  اضافه می‌کنیم. سپس تمام یال‌هایی را که از  $u$  خارج می‌شوند،  $**$ آرامسازی (relax) می‌کنیم.

این کار را تا زمانی که تمام رأس‌های گراف به مجموعه  $S$  اضافه نشده‌اند، انجام می‌دهیم.

مراحل اجرای الگوریتم روی یک گراف به شکل زیر می‌باشد:



\*\*توضیح شکل:\*\* در ابتدا، از بین رؤوس گراف، رأس  $s$  به مجموعه  $S$  اضافه می‌شود، زیرا مقدار  $d_s = 0$  برابر با صفر می‌باشد در صورتی که سایر رأس‌ها مقدار  $\infty$  دارند.

سپس با آرامسازی یال‌های متصل به  $s$ ، تخمین فاصله‌ها به روز می‌شوند (مثلًا  $y. d = 5$  و  $t. d = 10$ ).

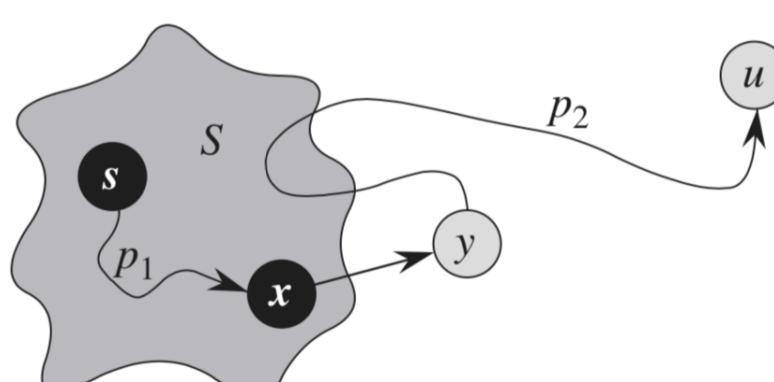
حال از بین رؤوس خارج از  $S$ ، رأس  $y$  که دارای کمترین تخمین است به  $S$  اضافه می‌شود و یال‌های متصل به آن آرامسازی می‌شوند.

اضافه کردن رؤوس و آرامسازی یال‌های آن تا زمانی که  $V \neq S$  است، ادامه پیدا می‌کند.

ادعا می‌کنیم برای هر رأس  $v$  ای که به مجموعه  $S$  اضافه می‌شود، داریم  $(s, v) = \delta(s, v)$  (یعنی  $d_v = \delta(s, v)$ ) برابر با وزن کوتاهترین مسیر واقعی از  $s$  به  $v$  است) و همچنین کوتاهترین مسیر از  $s$  به  $v$  تنها شامل رأس‌هایی است که قبلاً در مجموعه  $S$  قرار گرفته‌اند.

برای اثبات این ادعا، از  $**$ فرض خلف (Proof by Contradiction) استفاده می‌کنیم. بدین منظور، رأس  $u$  را اولین رأسی بگیرید که با ویژگی  $d_u > \delta(s, u)$  به مجموعه  $S$  اضافه می‌شود.

در نتیجه، کوتاهترین مسیر از  $s$  به  $u$  (که آن را  $P$  می‌نامیم) نمی‌تواند تماماً در مجموعه  $S$  قرار داشته باشد. این مسیر  $P$  را برابر با  $u \rightsquigarrow x \rightsquigarrow \dots \rightsquigarrow y \rightsquigarrow s$  در نظر می‌گیریم، که  $y$  اولین رأس در مسیر  $P$  است که در  $S$  قرار ندارد و  $x$  رأسی است که درست قبل از  $y$  روی مسیر  $P$ ، در  $S$  قرار داشته باشد. (بخشی از این مسیر حتماً در  $S$  قرار دارد زیرا رأس  $s$  در این مجموعه می‌باشد.)



طبق اجرای الگوریتم، می‌دانیم:

زیرا در هر مرحله، الگوریتم رأسی را انتخاب می‌کند که دارای کمترین  $d$  باشد و در این مرحله رأس  $u$  انتخاب شده است، پس این رأس بین رأس‌هایی که در  $S$  قرار ندارند، دارای کمترین مقدار  $d$  می‌باشد.

که با فرض  $u. d > \delta(s, u)$  در تناقض است، پس حکم ثابت شد.

$$\begin{aligned}
u. d &\leq y. d \leq x. d + w(x, y) \\
&= \delta(s, x) + w(x, y) \\
&= \delta(s, y) \leq \delta(s, y) + w(P_{y \rightarrow u}) \\
&= \delta(s, u) \\
\implies u. d &\leq \delta(s, u)
\end{aligned}$$

پس خواهیم داشت:

خط سوم (در اثبات ریاضی) با فرض اینکه وزن یال‌ها بزرگتر مساوی با صفر نباشد، درست است. در نتیجه الگوریتم Dijkstra\*\* فقط برای گراف‌هایی که وزن منفی ندارند درست کار می‌کند.

به عنوان مثال، قطعه کد زیر را بر روی گرافی که در ابتدای دفترچه بیان کردیم (گراف نمونه) که دارای یال با وزن منفی نمی‌باشد، اجرا کنید.

In [4]:

```
def find_min_distance(G, S_set_flags):
    """
    را دارد d.d پیدا می‌کند که کمترین S_set_flags اینتابع رأسی را از بین رأس‌های خارج از
    Args:
        G (Graph): شیء گراف.
        S_set_flags (list): قرار دارد یا خیر S لیستی از پرچم‌های بولی که نشان می‌دهد آیا رأس در مجموعه.
    Returns:
        Node: نیست S_set_flags که هنوز در d.d گره‌ای با کمترین.
    """
    minimum = math.inf
    min_node = None
    for v_node in G.V:
        # نیاشد S کمتر از حداقل فعلی باشد و رأس هنوز در (v_node.d) اگر تخمین فاصله.
        if v_node.d < minimum and S_set_flags[v_node.num] == False:
            minimum = v_node.d
            min_node = v_node
    return min_node

def dijkstra(G, s_node):
    """
    پیاده‌سازی الگوریتم دایکسترا برای یافتن کوتاه‌ترین مسیر از یک مبدأ واحد.
    این الگوریتم برای گراف‌هایی با وزن یال غیرمنفی کار می‌کند.
    Args:
        G (Graph): شیء گراف.
        s_node (Node): گره مبدأ.
    """
    initialize_single_source(G, s_node) # مقداردهی اولیه تخمین فاصله‌ها # حلقه اصلی دایکسترا.
    # S_set_flags: قرار دارد با خیر (رأس‌های نهایی شده) S لیستی از پرچم‌های بولی که نشان می‌دهد آیا رأس در مجموعه.
    S_set_flags = [False] * len(G.V)

    # بار تکرار می‌شود |V|: # در گراف v برای هر گره
    for _ in range(len(G.V)):
        u_node = find_min_distance(G, S_set_flags) # کمترین
        if u_node is None: # برای گراف‌های غیرهمبند)
            break
        S_set_flags[u_node.num] = True # رأس را به مجموعه u اضافه کن S را با خود می‌شود.
        # آرام‌سازی تمام یال‌های خروجی از u.
        for v_node in G.V:
            # نبود S هنوز در v وجود داشت (وزن غیر صفر) و v به u اگر یالی از
            if G.E[u_node.num][v_node.num] != 0 and S_set_flags[v_node.num] == False:
                relax(G, u_node, v_node)

    # چاپ نتایج: کوتاه‌ترین فاصله از مبدأ به هر رأس
    for v_node in G.V:
        print(f"vertex num {v_node.num}      min distance from Source: {v_node.d}")

# مثال استفاده
nodes = [Node(i) for i in range(5)] # (4 تا 0) ایجاد 5 گره.
g = Graph(nodes) # ایجاد شیء گراف.
# s = 0, t = 1, x = 2, z = 3, y = 4 (نام‌های فرضی برای گره‌ها)
# اضافه کردن یال‌ها با وزن‌های مثبت.
g.addEdge(0, 1, 10) # 0 -> 1 (وزن 10)
g.addEdge(0, 4, 5) # 0 -> 4 (5)
g.addEdge(1, 2, 1) # 1 -> 2 (1)
g.addEdge(1, 4, 2) # 1 -> 4 (2)
g.addEdge(2, 3, 4) # 2 -> 3 (4)
g.addEdge(3, 2, 6) # 3 -> 2 (6)
g.addEdge(3, 0, 7) # 3 -> 0 (7)
g.addEdge(4, 1, 3) # 4 -> 1 (3)
g.addEdge(4, 2, 9) # 4 -> 2 (9)
g.addEdge(4, 3, 2) # 4 -> 3 (2)

print("--- اجرای الگوریتم دایکسترا از مبدأ 0")
dijkstra(g, g.V[0]) # اجرای دایکسترا از گره 0.

--- 0 --- اجرای الگوریتم دایکسترا از مبدأ 0
vertex num 0      min distance from Source: 0
vertex num 1      min distance from Source: 8
vertex num 2      min distance from Source: 9
vertex num 3      min distance from Source: 7
vertex num 4      min distance from Source: 5
```

مرتبه زمانی کد بالا از  $O(|V|^2)$  می‌باشد. این به دلیل این است که در هر مرحله، تابع `find\_min\_distance` برای یافتن رأس با کمترین  $d$  در بین تمام رأس‌های خارج از  $S$ ، نیاز به پیمایش ( $O(|V|)$  رأس دارد و این عملیات  $|V|$  بار تکرار می‌شود.

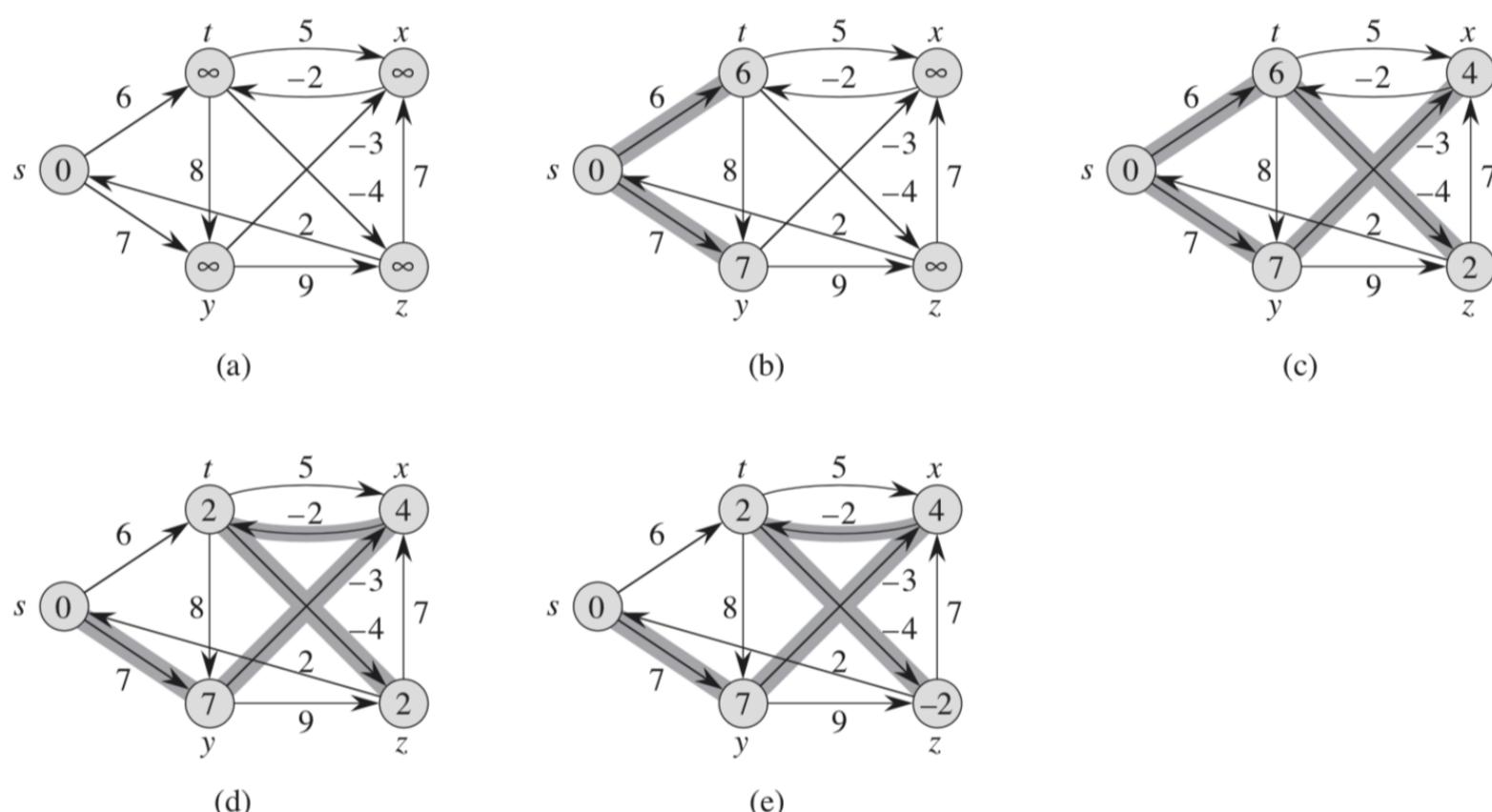
می‌توان آن را با استفاده از داده‌ساختار \*\*صف اولویت\*\* (Priority Queue) مبتنی بر `min-Heap` (برای بدست آوردن کمترین مقدار  $d$ )، تا مرتبه زمانی ( $O(|E| \log |V|)$  بهبود بخشد که می‌توانید به عنوان تمرین در مربوط به این دفترچه انجام دهید.

## الگوریتم Bellman-Ford (بلمن-فورد)

دیدیم که الگوریتم \*\*Dijkstra\*\* برای گراف‌هایی که دارای یال‌هایی با وزن منفی هستند، درست کار نمی‌کرد. حال الگوریتمی که در ادامه خواهیم گفت، برای این نوع گراف‌ها نیز به درستی کار می‌کند: \*الگوریتم بلمن-فورد\* (Bellman-Ford Algorithm).

در ابتدا باید برای گراف داده شده این شرط را داشته باشیم که گراف دارای دور با وزن منفی نباشد، زیرا اگر گراف دارای چنین دوری باشد، می‌توان با گذرکردن در این دور به هر تعداد دلخواهی وزن مسیر را کاهش داد. در نتیجه مسئله دارای جواب نیست (مسیر کوتاه‌ترین به سمت منفی بینهایت میل می‌کند). در ادامه بیان می‌کنیم که می‌توان با استفاده از الگوریتم \*\*Bellman-Ford\*\* وجود چنین دوری را در گراف تشخیص داد.

حال فرض می‌کنیم گراف داده شده دارای دور با وزن منفی نباشد. الگوریتم به این صورت کار می‌کند که همه‌ی یال‌های گراف را  $1 - |V|$  بار \*\*آرام‌سازی (relax)\*\* می‌کند. در این صورت، در انتهای برای هر رأس  $v \in V$  خواهیم داشت که  $v \cdot d = \delta(s, v)$  است (یعنی  $d_v$  برابر با وزن کوتاه‌ترین مسیر واقعی از  $s$  به  $v$  است).



برای سادگی کد و بهتر شدن زمان اجرای الگوریتم، از \*\*لیست یال‌ها (Edge List)\*\* برای ذخیره‌سازی گراف استفاده می‌کنیم (به جای ماتریس مجاورت).

چون تمامی یال‌های گراف  $1 - |V|$  بار آرام‌سازی می‌شوند، و هر آرام‌سازی (1)  $O(1)$  است، مرتبه زمانی الگوریتم از  $O(|V| \cdot |E|)$  است.

```
In [5]: class Graph_BF: قبلي Graph تغيير نام کلاس برای جلوگيري از تداخل با #
    def __init__(self, vertices_nodes): # ورودی لیستی از اشیاء # Node.
        self.V = vertices_nodes # لیستی از اشیاء # Node.
        self.E = [] # لیستی از یال‌ها. هر یال به صورت [u_node, v_node, weight].
    def addEdge(self, u_num, v_num, w):
        # اضافه کردن یک یال به لیست یال‌ها.
        # u_num و v_num هستند شماره رأس‌ها.
        # باید اشیاء self.V را از Node کنیم.
        u_node = self.V[u_num]
        v_node = self.V[v_num]
        self.E.append([u_node, v_node, w])
    def bellman_ford(G, s_node):
        """
        پیاده‌سازی الگوریتم بلمن-فورد برای یافتن کوتاه‌ترین مسیر از یک مبدأ واحد.
        این الگوریتم برای گراف‌هایی با یال‌های منفی نیز کار می‌کند
        به شرطی که دور با وزن منفی وجود نداشته باشد.
        """

```

Args:

G (Graph\_BF): شیء گراف.

```

s_node (Node): گره مبدأ.

"""
initialize_single_source(G, s_node) # اولیه تخمین فاصله‌ها.

    بار تکرار می‌شود ۱-۷ / |V| : حلقه اصلی بلمن-فورد
        در هر تکرار، تمام یال‌ها آرامسازی می‌شوند
        for _ in range(len(G.V) - 1):
            پیمایش بر روی تمام یال‌ها # آرامسازی یال (u, v).
            # u.d != math.inf: وزن یال (u, v) بزرگ‌تر از نهایت است.
            if u_node.d != math.inf and u_node.d + w < v_node.d:
                v_node.d = u_node.d + w
                v_node.p = u_node

    # برای جلوگیری از سرریز در جمع با وزن یال‌های منفی بزرگ
    # u.d != math.inf: وزن یال (u, v) بزرگ‌تر از نهایت است.
    if u_node.d != math.inf and u_node.d + w < v_node.d:
        v_node.d = u_node.d + w
        v_node.p = u_node

# بررسی وجود دور منفی (Negative Cycle)
    یک بار دیگر تمام یال‌ها را آرامسازی می‌کیم. اگر فاصله‌ای تغییر کرد، دور منفی وجود دارد
    for u_node, v_node, w in G.E:
        if u_node.d != math.inf and u_node.d + w < v_node.d:
            print("گراف شامل دور با وزن منفی است")
            return False # نشان‌دهنده وجود دور منفی

# چاپ نتایج: کوتاه‌ترین فاصله از مبدأ به هر رأس
for v_node in G.V:
    print(f"vertex num {v_node.num}      min distance from Source: {v_node.d}")
return True # نشان‌دهنده عدم وجود دور منفی و موفقیت در یافتن مسیرها

مثال استفاده
ایجاد ۵ گره (۰ تا ۴)
g_bf = Graph_BF(nodes_bf) # ایجاد شیء گراف برای بلمن-فورد
# s = 0, t = 1, x = 2, z = 3, y = 4 (نام‌های فرضی برای گره‌ها)

# اضافه کردن یال‌ها (با وزن‌های مثبت و منفی)
g_bf.addEdge(0, 1, 6)
g_bf.addEdge(0, 4, 7)
g_bf.addEdge(1, 2, 5)
g_bf.addEdge(1, 3, -4) # یال با وزن منفی
g_bf.addEdge(1, 4, 8)
g_bf.addEdge(2, 1, -2) # یال با وزن منفی
g_bf.addEdge(3, 2, 7)
g_bf.addEdge(3, 0, 2)
g_bf.addEdge(4, 3, 9)
g_bf.addEdge(4, 2, -3) # یال با وزن منفی

print("\n--- اجرای الگوریتم بلمن-فورد از مبدأ ۰")
bellman_ford(g_bf, g_bf.V[0]) # اجرای بلمن-فورد از گره ۰

مثال برای گراف با دور منفی
--- مثال گراف با دور منفی
nodes_neg_cycle = [Node(i) for i in range(3)]
g_neg_cycle = Graph_BF(nodes_neg_cycle)
g_neg_cycle.addEdge(0, 1, 1)
g_neg_cycle.addEdge(1, 2, 1)
g_neg_cycle.addEdge(2, 0, -3) # ۱- = ۳-۱+۱ با وزن ۰
دور منفی: bellman_ford(g_neg_cycle, g_neg_cycle.V[0])

--- اجرای الگوریتم بلمن-فورد از مبدأ ۰
vertex num 0      min distance from Source: 0
vertex num 1      min distance from Source: 2
vertex num 2      min distance from Source: 4
vertex num 3      min distance from Source: -2
vertex num 4      min distance from Source: 7

--- مثال گراف با دور منفی
--- گراف شامل دور با وزن منفی است.

Out[5]: False

```

## اثبات درستی الگوریتم:

می‌خواهیم اگر گراف دور با وزن منفی نداشته باشد، برای هر  $v \in V$  مقدار  $d_k(v)$  (منظور از  $d_{|V|-1}(v) = \delta(s, v)$ ) پس از  $k$  بار آرامسازی تمام یال‌ها است.

حال با استقرار بر روی  $k$  نشان می‌دهیم که  $(v)$  برابر است با کمترین وزن مسیر از  $s$  به  $v$  در صورتی که حداقل از  $k$  یال استفاده کند. با اثبات این حکم، درستی الگوریتم نشان داده می‌شود، زیرا هر مسیر ساده از  $s$  به سایر رؤس حداقل دارای  $|V| - 1$  یال می‌باشد.

\*پایه استقرار: اگر  $k = 0$  باشد، در این صورت با استفاده از صفر یال فقط به رأس  $s$  می‌رسیم که مقدار  $d_0$  آن صفر می‌باشد و کمترین مقدار برای رأس  $s$  است. در نتیجه حکم برقرار است.

\*\*فرض استقرار: حال فرض می‌کنیم برای هر رأس  $V$   $d_{k-1}(u)$ ,  $u \in V$  کمترین وزن مسیر از  $s$  به  $u$  با استفاده از حداقل  $k - 1$  یال است.

\*\*گام استقرار: مسیر  $P$  را کوتاه‌ترین مسیر به رأس  $v$  با حداقل  $k$  یال از  $s$  در نظر بگیرید. رأس قبلی  $v$  در مسیر  $P$  را  $u$  و مسیر  $s$  به  $u$  را  $Q$  می‌نامیم.

حال چون مسیر  $Q$  دارای حداقل  $1 - k$  یال می‌باشد، پس کوتاهترین مسیر از  $s$  به  $u$  است (زیرا اگر مسیر کوتاهتری به  $u$  با کمتر از  $1 - k$  یال وجود داشت، آنگاه مسیر  $P$  کوتاهترین مسیر با  $k$  یال نبود). در نتیجه طبق فرض استقرار،  $w(Q)$  برابر با  $d_{k-1}(u)$  خواهد شد.

در مرحله  $k$ ام از اجرای الگوریتم (که تمام یال‌ها آرام‌سازی می‌شوند)، مقدار  $(v)$  به  $d_k(v)$  بهروزرسانی می‌شود.

چون  $(v)$  برابر با  $d_{k-1}(P)$  است، و  $(v)$  نیز یک کران بالا برای کوتاهترین مسیر با  $1 - k$  یال است، پس  $(u, v)$  به  $d_k(v)$  (وزن مسیر  $P$ ) یا کمتر از آن بهروزرسانی می‌شود. از آنجایی که  $P$  کوتاهترین مسیر با حداقل  $k$  یال است، پس  $(P) = w(P)$  است و حکم استقرا ثابت می‌شود.

\*\* تشخیص دور با وزن منفی:

برای اینکه تشخیص دهیم گراف دارای دور با وزن منفی هست یا نه، کافی است ابتدا الگوریتم Bellman-Ford\*\* را روی گراف اجرا کنیم و مقادیر  $d$  را برای هر رأس  $v \in V$  یادداشت کنیم (پس از  $1 - |V|$  بار آرام‌سازی). سپس یک بار دیگر تمامی یال‌های گراف را آرام‌سازی کرده و مقادیر  $d$  جدید را با قبلی مقایسه کنیم. اگر رأسی وجود داشت که مقدار  $d$  آن در این مرحله  $|V|$ ام کاهش یافته باشد، در آن صورت گراف دارای دور با وزن منفی می‌باشد. (زیرا اگر دور منفی وجود نداشت، پس از  $|V|$  مرحله، تمام کوتاهترین مسیرها پیدا شده بودند و هیچ  $d$ ای نباید دیگر کاهش می‌یافت.)

```
In [93]: def detecting_negative_cycle(G):
    bellman_ford(G, G.V[0])
    first_list = []
```

```
    for v in G.V:
        first_list += [v.d]

    for u, v, w in G.E:
        if u.d + w < v.d: # relax
            v.d = u.d + w

    for i in range(len(G.V)):
        if G.V[i].d != first_list[i]:
            return True
```

```
    return False
```

```
nodes = [Node(i) for i in range(5)]
g = graph(nodes) # s = 0, t = 1, x = 2, z = 3, y = 4
```

```
g.addEdge(0, 1, 6)
g.addEdge(0, 4, 7)
g.addEdge(1, 2, 5)
g.addEdge(1, 3, -4)
g.addEdge(1, 4, 8)
g.addEdge(2, 1, -2)
g.addEdge(3, 2, 7)
g.addEdge(3, 0, 2)
g.addEdge(4, 3, 9)
g.addEdge(4, 2, -3)
```

```
detecting_negative_cycle(g)
```

```
vertex num 0  min distance from Source: 0
vertex num 1  min distance from Source: 2
vertex num 2  min distance from Source: 4
vertex num 3  min distance from Source: -2
vertex num 4  min distance from Source: 7
```

```
Out[93]: False
```

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل یازدهم: درخت فنویک و درخت بازه

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

- درخت فنویک
- پیاده سازی درخت فنویک
- درخت بازه
- پیاده سازی درخت بازه

## درخت فنویک (Fenwick Tree)

در این فصل، به بررسی دو داده‌ساختار پیشرفته و کارآمد می‌پردازیم که برای حل مسائل مربوط به **پرس‌وجو (Query)** و **بروزرسانی (Update)** روی بازه‌ها در آرایه‌ها به کار می‌روند: **درخت فنویک (Fenwick Tree)** و **درخت بازه (Segment Tree)**. این داده‌ساختارها امکان انجام عملیات‌های پیچیده را در زمان لگاریتمی فراهم می‌کنند.

فرض کنید می‌خواهیم مسئله زیر را حل کنیم:

آرایه‌ای  $n$  عضوی مانند  $A[1 \dots n]$  داریم و هر مرحله یکی از دو عملیات زیر را انجام دهد:

- یکی از اعضای آرایه را تغییر دهد (مثلًا  $A[i] += \text{delta}$  یا  $A[i] = \text{new\_value}$ )
- جمع اعضای یک بازه از اعضای آرایه را به عنوان خروجی بدهد، یعنی یک  $i$  و  $j$  بگیرد و  $A[i] + A[i + 1] + \dots + A[j]$  را خروجی بدهد.

یک راه حل ساده برای این مسئله به این صورت است که اعداد را در یک آرایه ذخیره می‌کنیم و طبق معمول با  $O(n)$  جمع اعداد یک زیرآرایه از آن را پیدا می‌کنیم.

همان طور که مشخص است با این راه حل در  $m$  مرحله از  $O(nm)$  عملیات لازم است. حال می‌خواهیم کاری کنیم که به جای این در  $O(m \log n)$  عملیات کارمن را انجام دهیم.

برای این کار به داده‌ساختاری نیاز داریم که هم تعویض اعضا و هم جمع کردن اعضا یک زیرآرایه را در  $O(\log n)$  انجام دهد. **درخت فنویک** یا **درخت دودویی-اندیس‌گذاری شده (-BIT)** برای این نیاز را برآورده می‌کند.

دقت کنید هر بازه که سرش در نقطه‌ی اول آرایه نباشد را می‌توان به صورت اختلاف دو بازه که سرشان در نقطه اول آرایه است نوشت، یعنی:

پس کافیست تنها بازه‌هایی بررسی شود که یک سرشان در نقطه اول آرایه است (یعنی جمع پیشوندی  $Sum[1 \dots k]$ ) چرا که بقیه بازه‌ها نیز با استفاده از این بازه‌ها در  $O(1)$  (با یک عمل تفریق) به دست می‌آیند.

## ساختار درخت فنویک

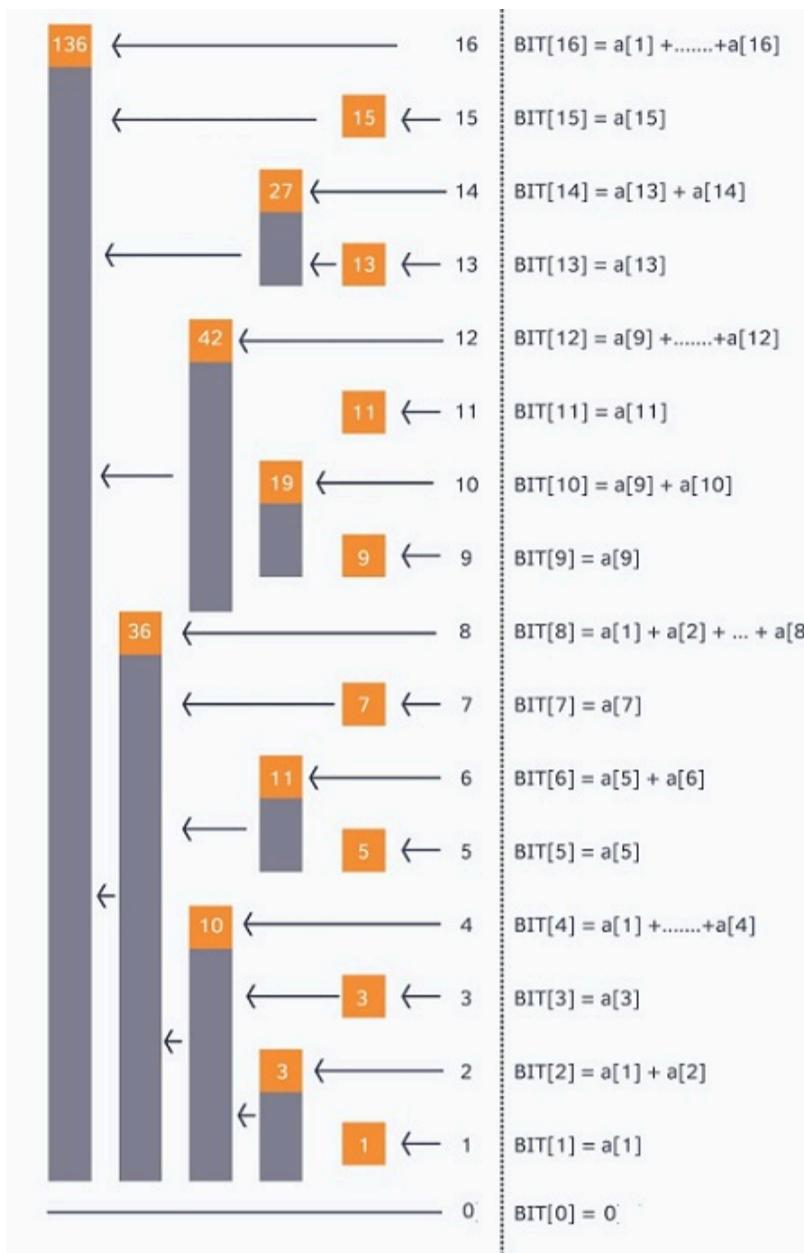
هر عدد طبیعی به طور یکتا به صورت جمع توانهای دو قابل بیان است. از طرفی هر بازه که از ۱ آغاز شود را نیز می‌توان به صورت جمع بازه‌هایی مجزا با طول‌هایی که توانی از دو هستند نوشت. برای مثال:

$$Sum[1 \dots 13] = Sum[1 \dots 8] + Sum[9 \dots 12] + Sum[13 \dots 13]$$

درخت فنويک از اين ايده بهره ميبرد. فرض کنيد  $r$  کوچکترین عددی باشد که بيت  $r$  ام عدد  $x$  برابر 1 باشد (مثلاً برای  $2^2 = 4$ ). آريه (BIT) را به

شكل زير تعریف ميکنیم:

$$\begin{aligned} & \text{ BIT}[x] : \text{ جمع عناصر بازه } A[x - 2^r + 1 \dots x] \\ & .\text{BIT}[0] = 0 \end{aligned}$$



شکل 1: نحوه تعریف آرایه BIT

دقت داشته باشید در تعریف آرایه BIT هدف این بوده که آرایه طوری باشد که زیرآرایه  $\hat{a}_m$  به  $\hat{n}$  ختم شود و طول آن برابر با کمترین رقم 1 موجود در بسط مبنای دوی  $i$  باشد. مثلاً در

$.2^2 = 12 = (1100)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$ ، این کمترین رقم مربوط به ضریب  $2^2$  است، بنابراین طول زیرآرایه منتهی به 12 برابر است با

انگیزه‌ی ما از این کار این است که هر زیرآرایه‌ای که از 1 آغاز شود را بتوان به صورت جمع تعداد محدودی (از  $O(\log n)$ ) عضو از آرایه BIT بدون نیاز به محاسبه‌ی سخت نوشتن. این نیز به دلیل نحوه تعریف آرایه به راحتی به دست آمد، چرا که کافیست اندیس نهایی بازه (مثلاً 13) را به صورت عددی در مبنای دو نوشته و توانهای دوی تشکیل دهنده‌ی آن را به ترتیب جمع کرد تا به جمع کل زیرآرایه برسیم. مثلاً:

$$\text{Sum}[1 \dots 13] = \text{BIT}[8] + \text{BIT}[12] + \text{BIT}[13]$$

رابطه فوق دقیقاً معادل رابطه

$$\begin{aligned} \text{Sum}[1 \dots 13] \\ = \text{Sum}[1 \dots 8] \\ + \text{Sum}[9 \dots 12] \\ + \text{Sum}[13 \dots 13] \end{aligned}$$

است.

\*\*چرا؟\*\*

پاسخ:\*\* این به دلیل ساختار دودویی درخت فنويک است. هر اندیس  $x$  در BIT، جمع بازه‌ای را نشان می‌دهد که از  $\text{LSB}(x)$  شروع شده و به  $x$  ختم می‌شود، که (کمترین بیت 1 در  $x$  را نشان می‌دهد). می‌توان با جمع کردن  $\text{BIT}[k]$ ,  $\text{BIT}[k - \text{LSB}(k)]$ ,  $\text{BIT}[k - \text{LSB}(k) - \text{LSB}(k - \text{LSB}(k))]$  و الی آخر، تا زمانی که به عدد  $k$  را می‌توان به صورت  $k = b_p 2^p + \dots + b_0 2^0$  نوشت. جمع  $\text{Sum}[1 \dots k]$  برابر با  $b_p 2^p + \dots + b_0 2^0$  است. هر عدد  $k$  را می‌توان به صورت  $b_p 2^p + \dots + b_0 2^0$  نوشت. این دقتاً همان تجزیه بازه به زیربازه‌هایی با طول 2 است.

# پیاده سازی درخت فنویک

## یافتن کم ارزش ترین بیت 1 (Least Significant Bit)

در پیاده سازی درخت فنویک، نیاز زیادی به داشتن کم ارزش ترین بیت 1 (LSB) در نمایش دودویی اندیس ها داریم. برای عدد  $x$  با محاسبه  $x \wedge -x$  می توان آن را در  $O(1)$  به دست آورد.

فرض کنید  $\bar{x}$  نشانگر معکوس منطقی (bitwise NOT) یا همان  $not(x)$  باشد. \*\*اثبات:

اگر نمایش دودویی  $x$  را به صورت  $a1b$  بدانیم که  $b$  فقط شامل بیت های 0 باشد و در نتیجه 1 بیت مورد نظر ما باشد (یعنی اولین 1 از سمت راست)، آنگاه:

$$x \wedge -x = (a1b) \wedge (\bar{a}1b) = 0 \dots 01b \quad \text{پس:} \quad -x = \bar{x} + 1 = \bar{a}1b + 1 = \bar{a}0(\overline{0 \dots 0}) + 1 = \bar{a}0(1 \dots 1) + 1 = \bar{a}1(0 \dots 0) = \bar{a}1b$$

این نتیجه، دقیقاً همان کم ارزش ترین بیت 1 در  $x$  را برمی گرداند.

به عنوان مثال برای :

$$\begin{aligned} x &= 12 \\ &= (1100)_2 \end{aligned}$$

$$\text{که } 2^2 = 4 \text{ و } 2^2 \text{ همان کم ارزش ترین بیت 1 در نمایش دودویی 12 است.} \quad x \wedge -x = (1100)_2 \wedge (0100)_2 = (0100)_2 = 4$$

## پیاده سازی

تابع موجود در پیاده سازی زیر:

- مقدار  $v$  را به عنصر  $ind$  لیست اصلی اضافه می کند. این عملیات در  $O(\lg n)$  انجام می شود.
- جمع اعداد عنصر اول تا  $ind$  لیست اصلی (جمع پیشوندی) را برمی گرداند. این عملیات نیز در  $O(\lg n)$  انجام می شود.

```
In [1]: class FenwickTree:  
    def __init__(self, arr):  
        # را از یک آرایه اولیه می سازد (BIT) متده: یک درخت فنویک.  
        self.n = len(arr) # اندازه آرایه اصلی.  
        self.BIT = [0] * (self.n + 1) # آرایه BIT خالی.  
        # با اضافه کردن تک تک عناصر آرایه اصلی BIT ساخت درخت.  
        for i in range(self.n):  
            self.add(i + 1, arr[i]) # اضافه کردن مقدار (1-based).  
  
    def add(self, ind, v):  
        # ام لیست اصلی اضافه می کند 'ind' را به عنصر v این تابع مقدار.  
        # که تحت تأثیر قرار می گیرند را به روزرسانی می کند BIT و تمام خانه های.  
        while(ind <= self.n):  
            self.BIT[ind] += v # مقدار v فعلی ind را به خانه فعلی ind اضافه می کند.  
            ind += ind & (-ind) # می رود ind بعدی (پدر در ساختار ضمنی درخت).  
  
    def sum(self, ind):  
        # ام لیست اصلی را برمی گرداند (جمع پیشوندی) 'ind' این تابع جمع عناصر از اندیس 1 تا ind.  
        s = 0 # مجموع.  
        while(ind > 0):  
            s += self.BIT[ind] # مقدار خانه فعلی ind را به مجموع اضافه می کند.  
            ind -= ind & (-ind) # می رود ind قبلی (پدر در ساختار ضمنی درخت).  
        return s  
  
# مثال استفاده  
ft = FenwickTree([1, 2, 3, 4, 5, 7, 8]) # [1,2,3,4,5,7,8]  
# Sum[1..4] = 1+2+3+4 = 10
```

```
باید 10 باشد # جمع عناصر تا اندیس 4
```

```
بروزرسانی عنصر سوم لیست اصلی (3) با اضافه کردن 2.  
لیست اصلی به صورت مجازی: [8, 7, 5, 4, 5, 2, 1] = [8, 7, 5, 4, 2+3, 1]  
# Sum[1..3] 8 = 5+2+1 = جدید  
باید 8 باشد # جمع عناصر تا اندیس 3 پس از بروزرسانی
```

جمع عناصر تا اندیس 4: 10

جمع عناصر تا اندیس 3 پس از بروزرسانی: 8

## درخت بازه (Segment Tree)

حال فرض کنید میخواهیم ساختاری داشته باشیم که بر روی آرایه  $A[0 \dots n - 1]$  دو عملیات زیر را انجام دهد:

- به تمامی اعضای یک بازه از آرایه مقداری را اضافه کند (Range Update).
- جمع عناصر یک بازه از آرایه را خروجی دهد (Range Query).

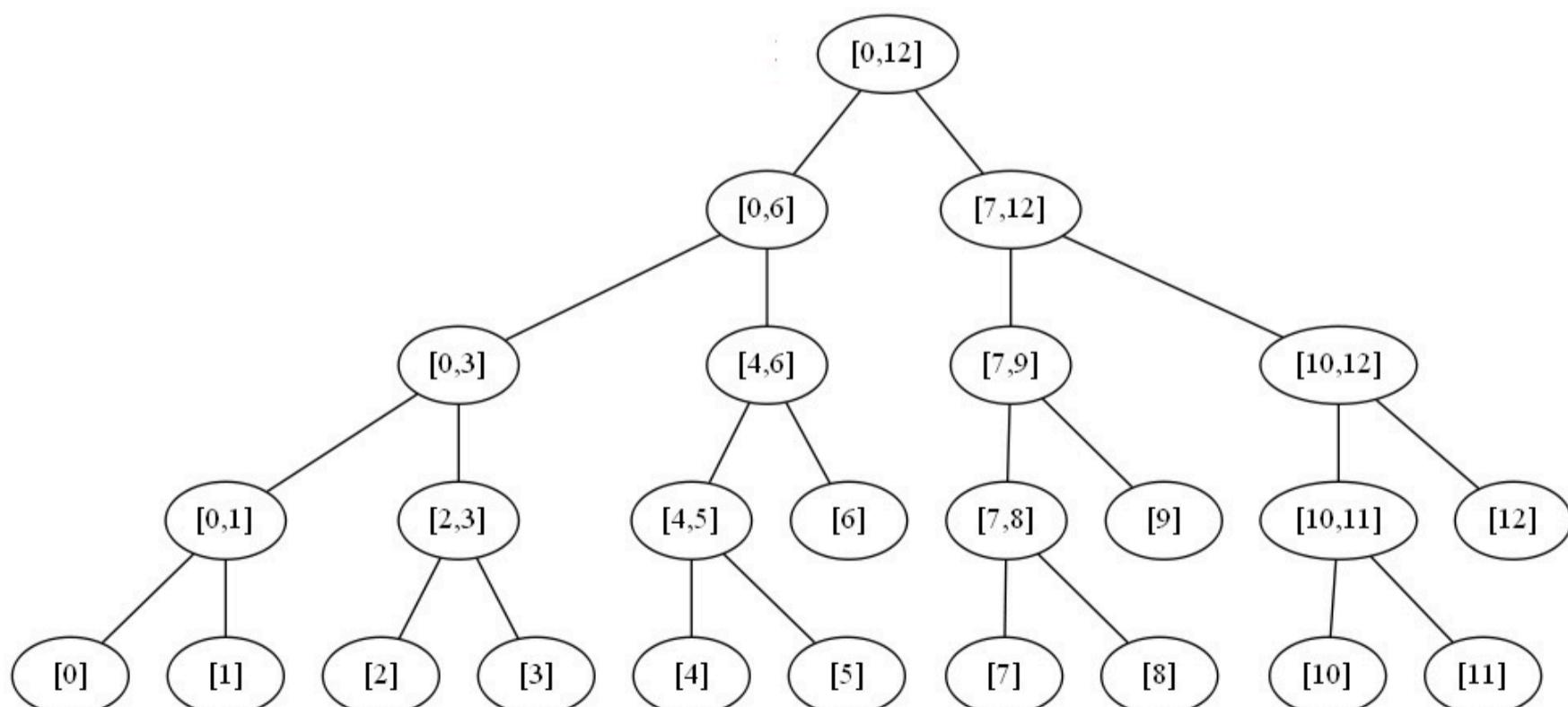
دقت کنید که دیگر نمیتوان از داده‌ساختار فنويک برای حل اين مسئله استفاده کرد.

\*\*پاسخ:\*\* درخت فنويک برای بروزرسانی تک نقطه و پرسش‌وجوی جمع پیشوندی بهینه است. عملیات Range Update (اضافه کردن به تمام اعضای یک بازه) در درخت فنويک به سادگی قابل پیاده‌سازی نیست و یا کارایی  $O(n)$  خواهد داشت. بنابراین نیاز به داده‌ساختار قویتری داریم که از ایده‌ای شبیه به ایده درخت فنويک بهره می‌جوید. این داده‌ساختار \*\*درخت بازه (Segment Tree)\*\* نام دارد.

## ساختار درخت بازه

درخت بازه درختی دودویی است که در آن هر گره معادل یک بازه از آرایه اصلی می‌باشد. به این ترتیب:

- ریشه معادل کل آرایه یعنی  $A[0 \dots n]$  است.
- برای هر گره معادل با  $(l \dots r)$ , اگر قرار دهیم  $mid = \lfloor (l + r) / 2 \rfloor$ , فرزند چپ آن معادل با  $A[l \dots mid]$  و فرزند راست آن معادل با  $A[mid \dots r]$  می‌باشد.
- هر برگ معادل با یک بازه تک عضوی یا به عبارتی یک عنصر آرایه است.



شکل 2: مثالی از درخت بازه یک آرایه سیزده عضوی [2]

سوالی که پیش می‌آید این است که این ساختار چگونه می‌تواند برای حل این مسئله به ما کمک کند. دقت کنید که مشابه فنويک، می‌توان هر بازه پرسش‌وجو را به صورت اجتماع مجذای تعدادی بازه از بازه‌های درخت بازه نوشت. برای مثال در شکل بالا، بازه [3, 8] را می‌توان به صورت اجتماع بازه‌های [3], [4, 6], [7, 8] و [10, 11] نوشت.

حال دقت کنید که هنگام نوشتن یک بازه به طور اجتماع مجزای اعضای درخت، می‌توان اعضا را طوری انتخاب کرد که هیچ سه عضو انتخاب شده‌ای هم ارتفاع نباشند. پس اگر ارتفاع درخت برابر  $h$  باشد، حداقل تعداد اعضای استفاده شده برابر  $2h$  است و با توجه به اینکه  $O(\lg n)$  است. بنابراین حدس می‌زنیم که بتوان در  $O(\lg n)$  به یک بازه به طور کامل دسترسی یافت. این را در ادامه می‌گوییم.

\*\*چرا این که تعداد بازه‌ها از  $O(\lg n)$  است برای بیان اثبات این حکم کافی نیست؟\*

$O(\log n)$  \*\*پاسخ: صرف اینکه تعداد بازه‌ها  $O(\log n)$  است، به تنها یک کافی نیست. باید نشان دهیم که هر یک از این بازه‌ها را می‌توان در زمان ثابت ( $O(1)$ ) پردازش کرد و همچنین فرآیند یافتن این بازه‌ها نیز در  $O(\log n)$  انجام می‌شود. این ترکیبی از ساختار درختی و نحوه پیمایش آن است که به این کارایی منجر می‌شود.

## پیاده سازی درخت بازه

درخت بازه را معمولاً با آرایه و به صورت **based-1** پیاده سازی می‌کنند. این به این معنی است که ریشه در اندیس ۱ آرایه قرار می‌گیرد و فرزندان چپ و راست گره  $i$  به ترتیب در اندیس‌های  $2i$  و  $1 + 2i$  قرار می‌گیرند.

1: [0, 16)															
2: [0, 8)								3: [8, 16)							
4: [0, 4)				5: [4, 8)				6: [8, 12)				7: [12, 16)			
8: [0, 2)		9: [2, 4)		10: [4, 6)		11: [6, 8)		12: [8, 10)		13: [10, 12)		14: [12, 14)		15: [14, 16)	
16: 0	17: 1	18: 2	19: 3	20: 4	21: 5	22: 6	23: 7	24: 8	25: 9	26: 10	27: 11	28: 12	29: 13	30: 14	31: 15

شکل 3: درخت بازه در قالب یک آرایه [3]

درخت بازه پیاده سازی مشخصی ندارد و ساختار آن بسته به نوع عملیاتی که قرار است روی بازه‌ها انجام دهد (مثلًا جمع، مینیمم، ماکسیمم، XOR و غیره) متفاوت است. درخت بازه پایین مختص مسئله بالا (جمع بازه‌ها) است.

در صورتی که به جای جمع بازه، تابع شرکت‌پذیر دیگری (مثلًا مینیمم یا ب.م.م.) مد نظر بود، می‌توان پیاده سازی مشابهی برای آن انجام داد. تابع شرکت‌پذیر (Associative Function) تابعی است که می‌توان آن را به صورت بازگشتی روی زیربازه‌ها اعمال کرد.

توجه داشته باشید که در پیاده سازی زیر، همه بازه‌ها به صورت  $[l, r]$  و  $[st, e]$  در نظر گرفته شده‌اند (بازه بسته از چپ و باز از راست).

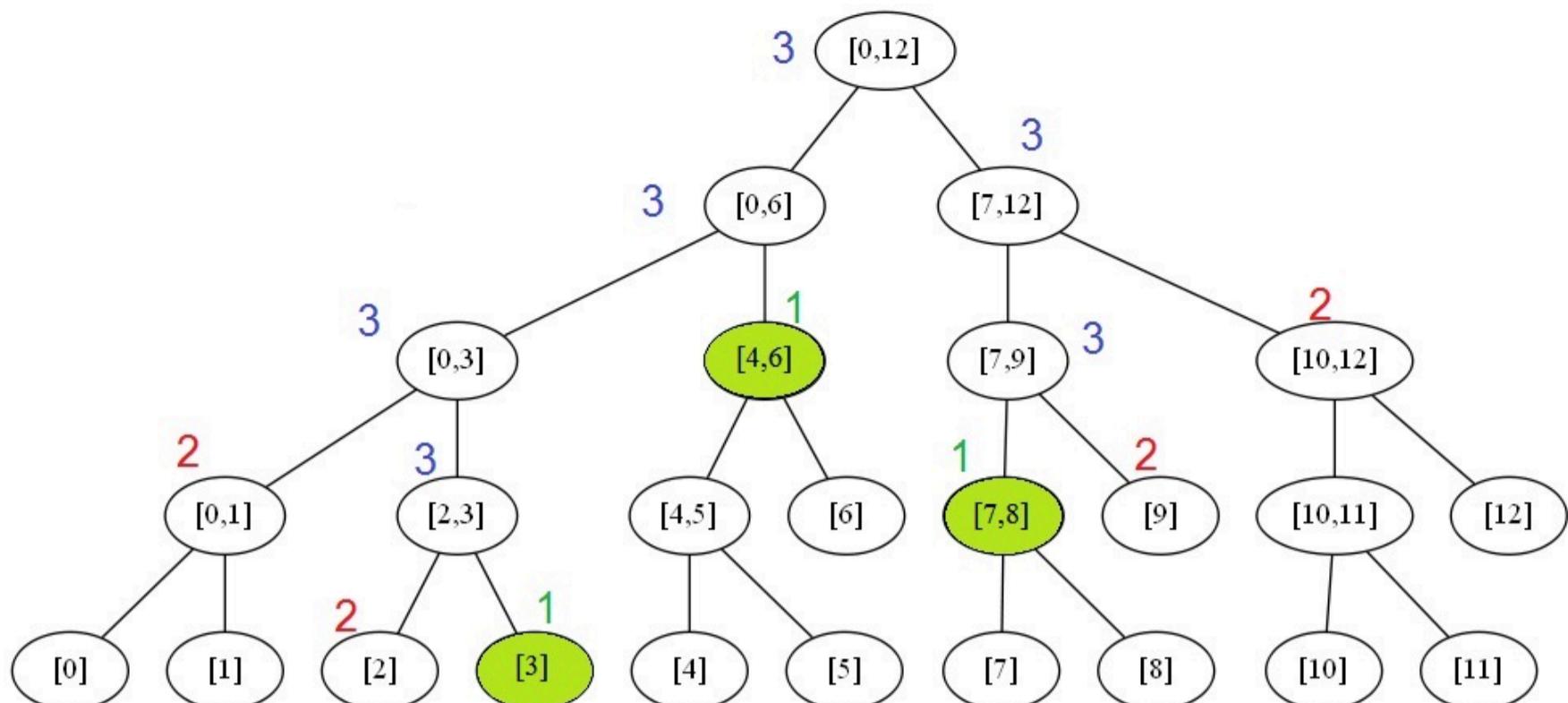
## تقسیم یک بازه به زیربازه‌های مجزای موجود در درخت بازه

در درخت بازه، نیاز زیادی به تقسیم بازه ورودی (برای پرس‌وجو یا بهروزرسانی) به زیربازه‌های موجود در درخت داریم. روشی بهینه برای این کار به این صورت است که بازه ورودی را در نظر گرفته و از ریشه شروع کنیم.

سپس به صورت بازگشتی:

1. اگر بازه معادل رأس فعلی کاملاً زیربازه‌ی بازه ورودی بود (یعنی بازه رأس فعلی کاملاً درون بازه پرس‌وجو قرار داشت)، آن را یکی از زیربازه‌های مورد نظر در نظر می‌گیریم و دیگر به فرزندانش نمی‌رویم.
2. اگر بازه معادل رأس فعلی هیچ اشتراکی با بازه ورودی نداشت، کاری نمی‌کنیم و از این شاخه خارج می‌شویم.
3. در صورت برقرار نبودن هیچیک از دو شرط فوق (یعنی بازه رأس فعلی با بازه پرس‌وجو همپوشانی جزئی داشت)، به صورت بازگشتی به فرزندان رأس می‌رویم تا بازه پرس‌وجو را به زیربازه‌های کوچکتر تقسیم کنیم.

در هر سطح از درخت، حداقل برای دو رأس مرزی (رأس‌های بازه پرس‌وجو همپوشانی دارد) حالت 3 رخ می‌دهد؛ پس در کل در  $O(\lg n)$  تقسیم مورد نظر انجام می‌شود. در توابع به روزرسانی و خروجی‌گیری درخت در پایین از این روش استفاده شده است.



شکل 4: تقسیم کردن بازه  $[3, 8]$  به زیربازه‌های مجزای موجود در درخت با روشن فوچ، در درخت بازه شکل 1

## ساخت درخت

برای ساخت درخت بازه، مقدار هر عنصر آرایه اصلی را در برگ معادلش قرار می‌دهیم (برگ‌ها متناظر با عناصر آرایه اصلی هستند). سپس مقدار معادل هر رأس داخلی (پدر) را برابر جمع مقدار فرزندانش قرار می‌دهیم. مقدار هر رأس در  $O(1)$  مشخص می‌شود، پس پیچیدگی کل ساخت درخت از  $O(n)$  است.

در پیاده‌سازی زیر، این تابع علاوه بر کار فوق، بازه‌های مربوط به هر اندیس آرایه را برای استفاده‌های بعدی (در توابع `update` و `get\_sum`) ذخیره می‌کند.

In [2]:

```
class Segment_Tree:
    def __init__(self, arr):
        متند سازنده: یک درخت بازه را از یک آرایه اولیه می‌سازد.
        اندازه آرایه اصلی # self.n = len(arr)
        لیست نگهداری اندیس شروع بازه هر گره: L
        # r: لیست نگهداری اندیس پایان بازه هر گره (باز).
        # r: اطمینان از فضای کافی برای درخت در آرایه # 4 * اندازه
        self.l = [0] * (4 * self.n + 4)
        self.r = [0] * (4 * self.n + 4)
        self.lazy = [0] * (4 * self.n + 4) # بعداً توضیح داده می‌شود) propagation
        self.sum = [0] * (4 * self.n + 4) # آرایه اصلی درخت بازه که جمع بازه‌ها را ذخیره می‌کند.

        شروع ساخت درخت به صورت بازگشتی از ریشه (اندیس 1) برای کل آرایه # self.make_tree(arr, 0, self.n, 1)

    def make_tree(self, arr, current_l, current_r, ind):
        این تابع درخت بازه را به صورت بازگشتی می‌سازد.
        # current_l: اندیس شروع بازه فعلی گره.
        # current_r: اندیس پایان بازه فعلی گره (باز).
        # ind: اندیس گره فعلی در آرایه.
        self.l[ind] = current_l # ذخیره اندیس شروع بازه.
        self.r[ind] = current_r # ذخیره اندیس پایان بازه.

        حالت پایه: اگر بازه تک عضوی باشد (برگ).
        مقدار برگ را از آرایه اصلی می‌گیرد # return

        mid = (current_l + current_r) // 2 # محاسبه نقطه میانی بازه # اندیس 2 و فرزند راست (اندیس ind + 1).
        self.make_tree(arr, current_l, mid, 2 * ind) # ساخت فرزند چپ
        self.make_tree(arr, current_l, mid, 2 * ind + 1) # ساخت فرزند راست

        مقدار گره فعلی برایر با جمع مقادیر فرزندانش است.
        self.sum[ind] = self.sum[2 * ind] + self.sum[2 * ind + 1]

# ساخت یک درخت بازه برای آرایه [11, 9, 7, 5, 3, 1]
my_array = [1, 3, 5, 7, 9, 11]
seg_tree = Segment_Tree(my_array)

print(f"آرایه اصلی: {my_array}")
print(f"اندازه درخت بازه (تعداد گره‌ها): {seg_tree.n}")
که مقادیر جمع بازه‌ها را در گره‌ها ذخیره می‌کند sum نمایش آرایه
# استفاده شده‌اند make_tree ممکن است پر نباشد، فقط آن‌هایی که توسط sum توجه: همه خانه‌های آرایه
# نمایش 20 خانه اول برای مثال # درخت بازه (بخشی از آن پر شده است) sum آرایه
print(f"درخت بازه (بخشی از آن پر شده است) sum آرایه: {seg_tree.sum[:20]}")
```



```

self.lazy[ind] = 0 # گره فعلی را صفر می‌کند lazy مقدار.

def update(self, st, e, v, ind = 1):
    """
    در آرایه اصلی اضافه می‌کند (st, e) را به تمامی عناصر در بازه v این تابع مقدار
    با استفاده از lazy propagation.
    """

Args:
    self: نمونه کلاس Segment_Tree.
    st (int): شروع بازه بروزرسانی (بسته).
    e (int): پایان بازه بروزرسانی (باز).
    v: مقداری که باید اضافه شود.
    ind (int): اندیس گره فعلی (شروع از ریشه = 1).
    """
    l = self.l[ind] # شروع بازه گره فعلی.
    r = self.r[ind] # پایان بازه گره فعلی.

    self.propagate_lazy(ind, l, r) # قبل از هر عملیات اعمال lazy propagation.

    # اگر بازه گره فعلی هیچ همیوشانی با بازه بروزرسانی نداشت.
    if e <= l or st >= r:
        return

    # اگر بازه گره فعلی کاملاً درون بازه بروزرسانی قرار داشت
    if l >= st and r <= e:
        self.lazy[ind] += v # را برای این گره بروز می‌کند lazy مقدار.
        self.propagate_lazy(ind, l, r) # بلافاصله lazy sum (برای بروزرسانی) را اعمال می‌کند.
        return

    # اگر همیوشانی جزئی بود، به فرزندان برو
    self.update(st, e, v, 2 * ind) # به روزرسانی فرزند چپ (sum).
    self.update(st, e, v, 2 * ind + 1) # به روزرسانی فرزند راست (sum).

    # گره فعلی را به روز می‌کند sum، پس از بازگشت از فرزندان
    self.sum[ind] = self.sum[2 * ind] + self.sum[2 * ind + 1]

# مثال استفاده از Segment Tree
# [11, 9, 7, 5, 3, 1] ساخت یک درخت بازه برای آرایه
my_array = [1, 3, 5, 7, 9, 11]
seg_tree = Segment_Tree(my_array)

print(f"آرایه اصلی {my_array}")
print(f"اندازه درخت بازه {seg_tree.n}")
# که مقادیر جمع بازه‌ها را در گره‌ها ذخیره می‌کند sum نمایش آرایه
# استفاده شده‌اند make_tree ممکن است پر نباشد، فقط آن‌هایی که توجه sum را می‌نمایند
# در درخت بازه (بخشی از آن پر شده است) sum آرایه
print(f"نمایش 20 خانه اول برای مثال {seg_tree.sum[0:20]}")

# است که در این کد ارائه نشده است query برای مشاهده جمع یک بازه خاص، نیاز به پیاده‌سازی متدهای
# ساخت درخت را تا حدی تأیید کنید. sum اما می‌توانید با بررسی آرایه
# باید جمع کل آرایه باشد [1] مثلاً
# باید 1 = seg_tree.sum[1] # جمع کل آرایه (ریشه درخت)
print(f"1 = 11+9+7+5+3+1") # باشد

# مثال استفاده از متدهای update
print("\n--- مثال استفاده از متدهای update ---")
# اضافه کردن 10 به عناصر بازه [1, 4] (یعنی اندیس‌های 1, 2, 3)
# [11, 9, 17, 15, 13, 1] = [11, 9, 10+7, 10+5, 10+3, 1] آرایه اصلی مجازی: [1, 1, 1, 1, 1, 1]
# جمع جدید باید باشد 66 = 11+9+17+15+13+1 باشد
seg_tree.update(1, 4, 10)
print(f"4 با 10 (جمع کل آرایه بعد از بروزرسانی بازه [1, 4]) باشد # باشد") # باشد

```

جمع کل آرایه بعد از بهروز سانه، بازه ۱۱.۴ تا ۱۰.۶

## خروجی گفتن از درخت (Range Query)

جمع کل بازه را با جمع کردن مقدار درخت برای زیربازه‌های مجزا محاسبه می‌کنیم. در اینجا هم برای درست بودن مقادیر، باید از **lazy propagation** استفاده کرد.

سخندگ، زمان اجراء، این تابع از  $O(\lg n)$  است.

```
In [5]: class Segment_Tree:  
    def __init__(self, arr):  
        متد سازنده: یک درخت بازه را از یک آرایه اولیه می‌سازد  
        # اندازه آرایه اصلی.  
        self.n = len(arr) #  
        لیست نگهداری اندیس شروع بازه هر گره: l: #  
        # r: لیست نگهداری اندیس پایان بازه هر گره (باز)  
        # برای اطمینان از فضای کافی برای درخت در آرایه اندیزه 4  
        self.l = [0] * (4 * self.n + 4)  
        self.r = [0] * (4 * self.n + 4)  
        self.lazy = [0] * (4 * self.n + 4) # propagation  
        self.sum = [0] * (4 * self.n + 4) # ذخیره می‌کند  
  
        # شروع ساخت درخت به صورت بازگشتنی از ریشه (اندیس 1) برای کل آرایه #  
        self.make_tree(arr, 0, self.n, 1)
```

```

این تابع درخت بازه را به صورت بازگشته می‌سازد.
# current_l: شروع بازه فعلی گره.
# current_r: اندیس پایان بازه فعلی گره (باز).
# ind: اندیس گره فعلی در آرایه sum.
self.l[ind] = current_l # ذخیره اندیس شروع بازه.
self.r[ind] = current_r # ذخیره اندیس پایان بازه.

حالت پایه: اگر بازه تک عضوی باشد (برگ).
    مقدار برگ را از آرایه اصلی می‌گیرد #.
        return

محاسبه نقطه میانی بازه #.
# اندیس 2 و فرزند راست (اندیس ind + 1) ساخت فرزند چپ.
self.make_tree(arr, current_l, mid, 2 * ind)
self.make_tree(arr, mid, current_r, 2 * ind + 1)

مقدار گره فعلی برابر با جمع مقادیر فرزندانش است #.
self.sum[ind] = self.sum[2 * ind] + self.sum[2 * ind + 1]

def propagate_lazy(self, ind, l, r):
    """
    آن اعمال کرده و به فرزندانش منتقل می‌کند sum گره فعلی را به lazy این تابع مقدار.

    Args:
        self: نمونه کلاس Segment_Tree.
        ind (int): اندیس گره فعلی.
        l (int): شروع بازه گره فعلی.
        r (int): پایان بازه گره فعلی (باز).

    """
    if self.lazy[ind] != 0: # داشت اگر گره فعلی مقدار.
        # گره اعمال می‌کند sum را به lazy مقدار.
        # طول بازه است (r - l).
        self.sum[ind] += self.lazy[ind] * (r - l)

        # فرزندان منتقل می‌کند lazy اگر گره برگ نبود، مقدار #.
        if l + 1 < r: # اگر گره برگ نبود (عنی فرزند داشت).
            self.lazy[2 * ind] += self.lazy[ind] # lazy فرزند چپ را به روز می‌کند.
            self.lazy[2 * ind + 1] += self.lazy[ind] # lazy فرزند راست را به روز می‌کند.

        self.lazy[ind] = 0 # گره فعلی را صفر می‌کند Lazy مقدار.

    def update(self, st, e, v, ind = 1):
        """
        در آرایه اصلی اضافه می‌کند (e, st) را به تمامی عناصر در بازه v این تابع مقدار با استفاده از lazy propagation.

        Args:
            self: نمونه کلاس Segment_Tree.
            st (int): شروع بازه بهروزرسانی (بسته).
            e (int): پایان بازه بهروزرسانی (باز).
            v: مقداری که باید اضافه شود.
            ind (int): اندیس گره فعلی (شروع از ریشه = 1).

        """
        l = self.l[ind] # شروع بازه گره فعلی.
        r = self.r[ind] # پایان بازه گره فعلی.

        self.propagate_lazy(ind, l, r) # قبل از هر عملیات اعمال Lazy propagation.

        # اگر بازه گره فعلی هیچ همیوشانی با بازه بهروزرسانی نداشت.
        if e <= l or st >= r:
            return

        # اگر بازه گره فعلی کاملاً درون بازه بهروزرسانی قرار داشت #.
        if l >= st and r <= e:
            # را برای این گره به روز می‌کند Lazy مقدار.
            self.lazy[ind] += v
            self.propagate_lazy(ind, l, r) # اعمال می‌کندLazy بلاfacleه sum.
            return

        # اگر همیوشانی جزئی بود، به فرزندان برو #.
        self.update(st, e, v, 2 * ind) # بهروزرسانی فرزند چپ.
        self.update(st, e, v, 2 * ind + 1) # بهروزرسانی فرزند راست.

        # گره فعلی را به روز می‌کند sum، پس از بازگشت از فرزندان #.
        self.sum[ind] = self.sum[2 * ind] + self.sum[2 * ind + 1]

    def get_sum(self, st, e, ind = 1):
        """
        در آرایه اصلی برمی‌گرداند (e, st) این تابع جمع عناصر را در بازه بازه با استفاده از lazy propagation.

        Args:
            self: نمونه کلاس Segment_Tree.
            st (int): شروع بازه پرسنحو (بسته).
            e (int): پایان بازه پرسنحو (باز).
            ind (int): اندیس گره فعلی (شروع از ریشه = 1).

        Returns:
            int: جمع عناصر در بازه [st, e].
        """
        l = self.l[ind] # شروع بازه گره فعلی.
        r = self.r[ind] # پایان بازه گره فعلی.

        self.propagate_lazy(ind, l, r) # قبل از هر عملیات اعمال Lazy propagation.

        # اگر بازه گره فعلی هیچ همیوشانی با بازه پرسنحو نداشت.
        if e <= l or st >= r:
            return 0 # هیچ مقداری به مجموع اضافه نمی‌شود.

        # اگر بازه گره فعلی کاملاً درون بازه پرسنحو قرار داشت #.
        if l >= st and r <= e:
            # این گره را برمی‌گرداند sum مقدار.
            return self.sum[ind]

```

١٣

اللگوریتم طراحی کنید که مجموعه ای از  $n$  عکس را در یک مجموعه محدودیت نداشته باشد.

- یک  $i$  و یک  $t$  ورودی بگیرد و  $A[i]$  را برابر  $t$  قرار دهد (Point Update).
  - یک  $i$  و یک  $j$  ورودی بگیرد و مینیمم اعداد زیرآرایه  $A[i \dots j]$  را خروجی دهد (Range Minimum).

(Query - RMO)

<sup>10</sup> See also *ibid.*, 1993, 1, 1–12.

```
In [6]: import math
```

```

class Min_Segment_Tree:
    def __init__(self, arr):
        # متدهای سازنده: یک درخت بازه برای یافتن مینیمم بازه را از آرایه اولیه می‌سازد.
        self.n = len(arr)
        self.l = [0] * (4 * self.n + 4)
        self.r = [0] * (4 * self.n + 4)
        # نشاندهنده مقداری است که مقدار Lazy Propagation در Min Segment Tree، برای
        # مثلاً برای Range Update: Add). باید به زیردرخت اعمال شود
        # (مقداردهی یک بازه به یک مقدار) برای Lazy propagation، در این پیاده‌سازی
        # می‌تواند استفاده شود (اضافه کردن مقدار به یک بازه) یا
        # Point Update (به روزرسانی یک نقطه) به صورت update در این تمرن، چون

```

```

# propagate_lazy نمی‌شود، اما برای حفظ ساختار کامل update در متدهاست.
# آن را حفظ می‌کنیم و قابلیت توسعه به
self.lazy = [float("inf")] * (4 * self.n + 4) # برای مینیمم
self.min_val = [float("inf")] * (4 * self.n + 4) # ذخیره می‌کند
self.make_tree(arr, 0, self.n, 1)

def make_tree(self, arr, l, r, ind):
    # این تابع درخت بازه را به صورت بازگشتی می‌سازد.
    self.l[ind] = l
    self.r[ind] = r
    if r <= l + 1: # حالت پایه: اگر بازه تک عضوی باشد
        self.min_val[ind] = arr[l]
        return
    mid = (l + r) // 2
    self.make_tree(arr, l, mid, 2 * ind)
    self.make_tree(arr, mid, r, 2 * ind + 1)
    self.min_val[ind] = min(self.min_val[2 * ind], self.min_val[2 * ind + 1]) # مینیمم فرزندان.

def propagate_lazy(self, ind, l, r):
    # آن اعمال کرده و به فرزندانش منتقل می‌کند گردد فعلی را به این تابع مقدار
    # لازم است Range Update با Set/Add.
    # فراخوانی نمی‌شود update این تابع به طور مستقیم در ، در این تمرين
    # به گره‌های پایین‌تر منتقل شوند Lazy ممکن است نیاز باشد تا مقادیر اما برای
    # وجود داشت Lazy اگر مقدار # به گردد فعلی اعمال Lazy.
    if self.lazy[ind] != float("inf"):
        self.min_val[ind] = min(self.min_val[ind], self.lazy[ind])
        if l + 1 < r: # اگر گردد بروگ نبود
            # فعلی ترکیب می‌کند فرزندان را با مقدار Lazy انتقال: مقدار
            self.lazy[2 * ind] = min(self.lazy[2 * ind], self.lazy[ind])
            self.lazy[2 * ind + 1] = min(self.lazy[2 * ind + 1], self.lazy[ind])
        self.lazy[ind] = float("inf") # Lazy را پاک می‌کند.

def update(self, target_ind, new_value, ind=1):
    """
    تغییر میدهد new_value را در آرایه اصلی به target_ind این تابع مقدار عنصر
    است. این یک عملیات Point Update است.

    Args:
        self: نمونه کلاس Min_Segment_Tree.
        target_ind (int): (0-based). اندیس عنصری که باید بهروزرسانی شود.
        new_value: مقدار جدید برای عنصر.
        ind (int): اندیس گردد فعلی در درخت (شروع از ریشه = 1).
    """
    l = self.l[ind] # شروع بازه گردد فعلی.
    r = self.r[ind] # پایان بازه گردد فعلی.

    # اگر گردد برگ به target_ind مربوط باشد.
    if r <= l + 1: # این گردد یک برگ است
        self.min_val[ind] = new_value # مقدار برگ را بهروزرسانی می‌کند
        return

    mid = (l + r) // 2 # محاسبه میانه.

    # تصمیم‌گیری برای رفتن به فرزند چپ یا راست
    if target_ind < mid:
        self.update(target_ind, new_value, 2 * ind) # بهروزرسانی در فرزند چپ.
    else:
        self.update(target_ind, new_value, 2 * ind + 1) # بهروزرسانی در فرزند راست.

    # گردد فعلی را بهروزرسانی می‌کند، بس از بازگشت از فرزندان
    self.min_val[ind] = min(self.min_val[2 * ind], self.min_val[2 * ind + 1])

def get_min(self, st, e, ind=1):
    """
    در آرایه اصلی برمی‌گرداند (e, st) این تابع مینیمم عناصر را در بازه
    می‌گیرد.

    Args:
        self: نمونه کلاس Min_Segment_Tree.
        st (int): شروع بازه پرسی (بسته).
        e (int): پایان بازه پرسی (باز).
        ind (int): اندیس گردد فعلی (شروع از ریشه = 1).
    """
    Returns:
        int: مینیمم عناصر در بازه [st, e].
    """
    l = self.l[ind] # شروع بازه گردد فعلی.
    r = self.r[ind] # پایان بازه گردد فعلی.

    # اگر بازه گردد فعلی هیچ همیوشانی با بازه پرسی نداشت
    if e <= l or st >= r:
        return float("inf") # گرفتن تأثیری نداشته باشد min بینهایت برگردان تا در.

    # اگر بازه گردد فعلی کاملاً درون بازه پرسی و قرار داشت
    if l >= st and r <= e:
        return self.min_val[ind] # مینیمم این گردد را برمی‌گرداند.

    # اگر همیوشانی جزئی بود، به فرزندان برو و مینیمم نتایج را بگیر
    min_left_child = self.get_min(st, e, 2 * ind) # پرسی در فرزند چپ
    min_right_child = self.get_min(st, e, 2 * ind + 1) # پرسی در فرزند راست.

    # مینیمم دو نتیجه را برمی‌گرداند
    return min(min_left_child, min_right_child)

# مثال استفاده
segment_tree = Min_Segment_Tree([1, 2, 3, 4, 5, 6, 7, 8])
# 3 = (5, 4, 3, 2, 1, 0) یعنی عناصر با اندیس 3, 4, 5, 6, 7, 8 (مقادیر 2, 3, 4, 5, 6, 7) مینیمم بازه [2, 5] است
print("5, 2]", segment_tree.get_min(2, 5))

# 4 # بروزرسانی عنصر با اندیس 2 (مقادیر 3) به
# [8, 7, 6, 5, 4, 4, 2, 1] آرایه اصلی به صورت مجازی: (مقادیر 4, 3, 2, 1, 0, 5, 6, 7)
segment_tree.update(2, 4)
# 4 = (5, 4, 4, 3, 2, 1, 0) جدید: عناصر با اندیس 3, 4, 5, 6, 7, 8 (مقادیر 2, 3, 4, 5, 6, 7) مینیمم بازه [2, 5] است
print("5, 2]", segment_tree.get_min(2, 5))

```

```
#مثال دیگر
segment_tree_2 = Min_Segment_Tree([10, 20, 5, 30, 15])
print("5 ,0] مینیمم:", segment_tree_2.get_min(0, 5)) # باید 5 باشد
segment_tree_2.update(2, 1) # 1 به اندیس 2 (مقدار 5) را بروزرسانی کنیم
print("5 ,0] مینیمم:", segment_tree_2.get_min(0, 5)) # باید 1 باشد
3 : مینیمم بازه [2,5]
4 : مینیمم بازه [2,5] پس از بروزرسانی:
5 : مینیمم [0,5]
1 : مینیمم [0,5] پس از بروزرسانی:
```

## پانویس

[hackerearth.com .1](https://www.hackerearth.com/practice/algorithms/trees/segment-trees/tutorial/)

[techienotes.info .2](https://techienotes.info/segment-tree-in-python/)

[codeforces.com .3](https://codeforces.com/edu/course/1/lesson/4/1/practice)

# داده‌ساختارها و الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم اول سال تحصیلی ۱۴۰۴-۱۴۰۵

## فصل دوازدهم: درخت‌های دودویی جستجوی متوازن

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری(علمی و ادبی): حمید نامجو و امیرحسین همتی

### فهرست

• مقدمه

• اهمیت متوازن بودن

• انواع د.د.ج‌های متوازن

• درخت قرمز-سیاه

• چرخش

• درج

• حذف

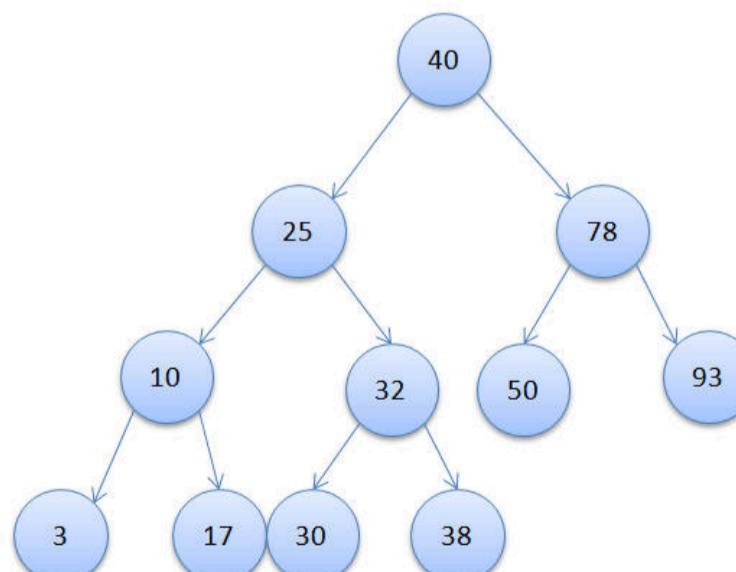
• چرا درخت قرمز-سیاه متوازن است؟

• تمرین عملی

### مقدمه

درخت دودویی جستجو (به انگلیسی: **Binary Search Tree** یا به اختصار **BST**) درختی ریشه‌دار و دودویی است که به ازای هر رأس مانند  $v$ :

- مقادیر تمامی رأس‌های زیردرخت فرزند سمت چپ آن از مقدار رأس  $v$  کوچک‌تر است.
- مقادیر تمامی رأس‌های زیردرخت فرزند سمت راستش از  $v$  بزرگ‌تر است.



هر رأس درون BST دارای ویژگی‌های زیر است:

- برچسب (label) برای ذخیره مقدار
- اشارهگر به فرزند چپ (leftChild)

- اشاره‌گر به فرزند راست (**rightChild**)
- اشاره‌گر به پدرش (**parent**)

به غیر از رأس ریشه، بقیه رأس‌ها حتماً پدر خواهند داشت. این ساختار امکان جستجو، درج و حذف کارآمد را فراهم می‌کند.

## اهمیت متوازن بودن

می‌دانیم که اکثر عملیات روی BST‌ها (درج، حذف، جستجو، پیدا کردن عنصر قبلی و بعدی، و پیدا کردن عنصر کمینه و بیشینه) از مرتبه زمانی ارتفاع درخت ( $O(h)$ ) است. همچنین می‌دانیم که ارتفاع یک BST می‌تواند بین  $O(\log n)$  (در بهترین حالت، یعنی درخت کاملاً متعادل) و  $O(n)$  (در بدترین حالت، یعنی درخت کاملاً کج و شبیه لیست پیوندی) متغیر باشد.

درخت‌های دودویی جستجوی متوازن (Balanced Binary Search Trees - BBSTs) \*\* با انجام تغییرات و چرخش‌هایی در حین انجام عملیات مورد نیاز (درج و حذف)، تضمین می‌کنند ارتفاع درخت از  $O(\log n)$  باقی بماند و در نتیجه همه‌ی این عملیات در مرتبه زمانی  $O(\log n)$  انجام شوند. این تضمین، عملکرد کارآمد را حتی در بدترین حالت نیز حفظ می‌کند.

## انواع د.د.ج‌های متوازن

تضمین کم بودن ارتفاع BST به روش‌های مختلفی انجام می‌شود که هر کدام قوانین و الگوریتم‌های تعادل‌سازی خاص خود را دارند. تعدادی از این روش‌ها عبارتند از:

- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

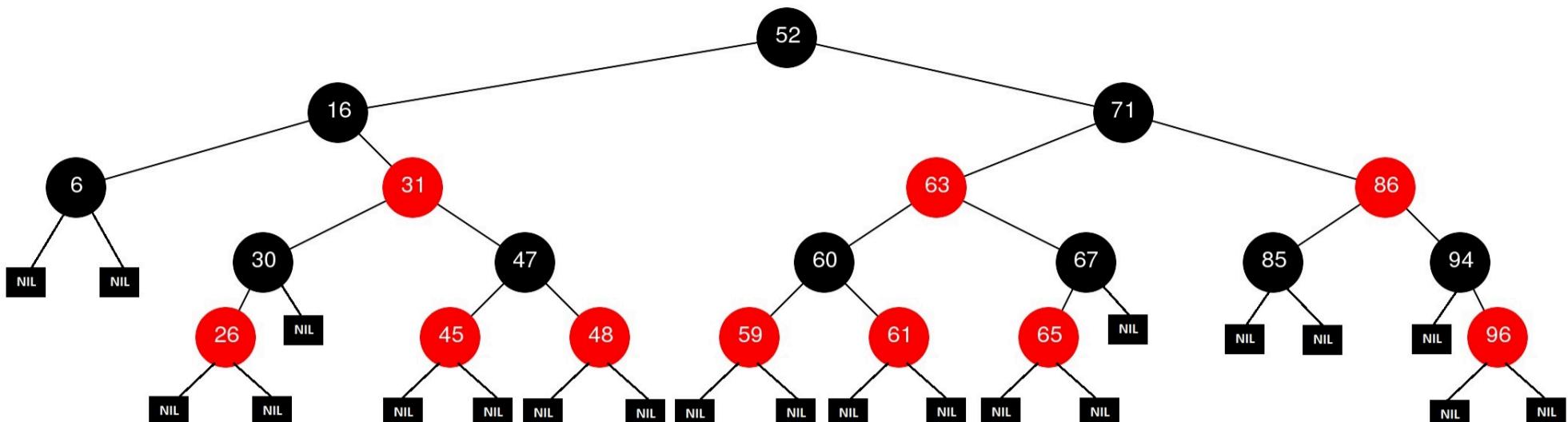
در ادامه، \*\*Red-Black Tree\*\* (درخت قرمز-سیاه) که عموماً بر استفاده‌های تعدادی از آن‌ها است را بررسی می‌کنیم.

## درخت قرمز-سیاه

یک \*\*درخت قرمز-سیاه\*\* (Red-Black Tree) یک BST است که خواص زیر را داشته باشد:

- خاصیت ۱: هر رأس یک رنگ دارد که یا سیاه است یا قرمز.
- خاصیت ۲: رأس ریشه همیشه سیاه است.
- خاصیت ۳: هر رأس غیر NIL (گره‌های پوچ یا نگهبان که برگ‌های واقعی را نشان می‌دهند) دقیقاً ۲ فرزند دارد (یعنی تمامی برگ‌ها NIL یا پوچ هستند) و رنگ تمامی برگ‌ها (گره‌های NIL) حتماً سیاه است.
- خاصیت ۴: اگر رأسی قرمز باشد، حتماً هر دو فرزند آن سیاه هستند (یعنی هیچ دو رأس قرمزی نمی‌توانند پشت سر هم در یک مسیر قرار گیرند).

- خاصیت ۵: برای هر رأس، هر مسیر ساده از آن به یکی از برگ‌های زیردرختش (گره‌های NIL) دارای تعداد مساوی از رأس‌های سیاه است (این تعداد به عنوان "عمق سیاه" شناخته می‌شود).



این خواص باعث می‌شوند طول بلندترین مسیر از ریشه به یکی از برگ‌ها حداقل دو برابر طول کوتاهترین مسیر از ریشه به یکی از برگ‌ها باشد که توازن خوبی را در درخت ایجاد می‌کند.

(تعداد رئوس سیاه مسیرها که مساوی است، از آنجایی که فرزندان رئوس قرمز حتما سیاه هستند، تعداد رئوس قرمز حداقل ۰ و حداقل به تعداد رئوس سیاه است که خاصیت بالا را نتیجه می‌دهد). این تضمین می‌کند که ارتفاع درخت همواره لگاریتمی باقی بماند.

## چرخش (Rotation)

خواص گفته شده برای درخت قرمز-سیاه باید پس از هر عمل بر روی این درخت (مانند درج و حذف) باز هم برقرار باشند. برای حفظ خواص درخت قرمز-سیاه از دو عمل \*\*تغییر رنگ رأس‌ها\*\* و \*\*چرخش\*\* به راست یا چپ استفاده می‌شود.

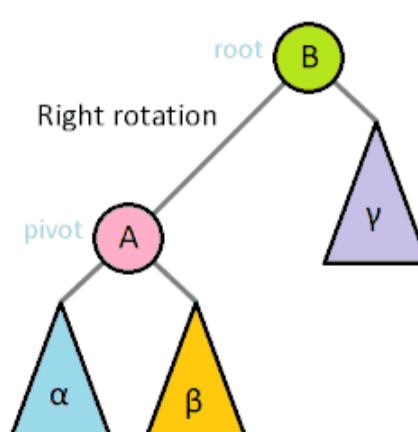
چرخش به راست در واقع تغییر جهت یالی از درخت است که به سمت چپ بوده، به سمت راست با یک چرخش و سپس انجام تغییرات لازم است تا منطق درخت جستجو بودن در درخت حفظ شود. دقت کنید که در یک چرخش، خاصیت‌های دیگر درخت قرمز-سیاه لزوماً حفظ نمی‌شوند و نیاز به اصلاحات رنگی و یا چرخش‌های بیشتر است.

(چرخش در واقع برای درخت‌های دودویی جستجو تعریف می‌شود و منطق درخت جستجو بودن آنها را بر هم نمی‌زند، بلکه ساختار سلسله‌مراتبی را تغییر می‌دهد.)

در چرخش به راست روی رأس  $x$  (که در شکل با A نشان داده شده)، فرض می‌کنیم فرزند چپ آن (مثلاً  $y$  که در شکل با B نشان داده شده) پوج (NIL) نیست. اگر فرزند سمت راست  $x$  را  $\alpha$  بنامیم، و فرزند راست  $y$  را  $\beta$  بنامیم، با چرخش به راست:

1.  $y$  ریشه‌ی جدید این زیردرخت می‌شود.
2.  $x$  فرزند راست  $y$  می‌شود.
3.  $\beta$  (و زیردرختش) فرزند چپ  $x$  می‌شود.

چرخش به چپ مشابه است.



In [1]: # Look at this code only as a psuedo code

# تابع \_left\_rotate

```

def _left_rotate(self, x):
    """
    را انجام می‌دهد x چرخش به چپ روی گره.
    Args:
        self: نمونه کلاس rbtree.
        x (rbnode): گره‌ای که چرخش حول آن انجام می‌شود.
    """
    y = x.right # y را فرزند راست.
    x.right = y.left # تبدیل می‌کند x را به فرزند راست y زیردرخت چپ.
    if y.left != self.nil:
        y.left.p = x # قرار می‌دهد x را پدر فرزند چپ.
        y.p = x.p # قرار می‌دهد x را پدر y را.
    if x.p == self.nil: # ریشه جدید می‌شود y، ریشه بود x اگر
        self._root = y
    elif x == x.p.left: # فرزند چپ بدرش بود x اگر
        x.p.left = y # قرار می‌دهد y را x فرزند چپ پدر.
    else: # فرزند راست بود x اگر
        x.p.right = y # قرار می‌دهد y را x فرزند راست پدر.
    y.left = x # قرار می‌دهد y را فرزند چپ x.
    x.p = y # قرار می‌دهد y را x پدر.

```

## درج (Insertion)

برای درج عنصر  $x$  در یک درخت قرمز-سیاه، ابتدا آن را به همان روشی که در BST عادی درج می‌کردیم، درج می‌کنیم. گره جدید را \*\*قرمز\*\* می‌کنیم و ۲ فرزند سیاه پوچ (NIL) برای آن در نظر می‌گیریم.

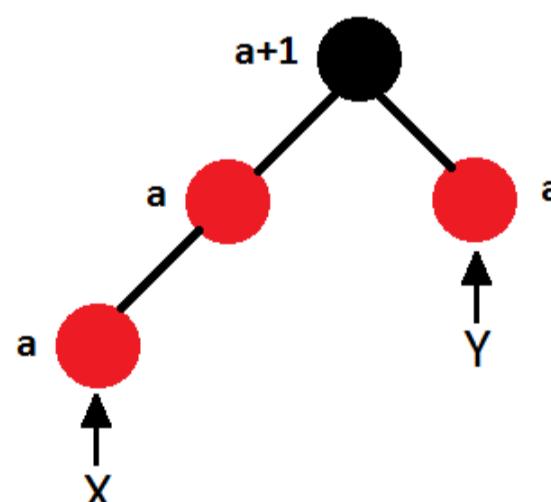
پس از درج، ممکن است یکی از قواعد درخت قرمز-سیاه نقض شده باشد. تنها قاعده‌ای که ممکن است نقض شود، \*\*خاصیت ۴\*\* است: "اگر رأسی قرمز باشد، حتماً هر دو فرزند آن سیاه هستند." این نقض زمانی رخ می‌دهد که پدر گره درج شده ( $p[x]$ ) نیز قرمز باشد.

باید با تغییرات در رنگ‌ها و با چرخش‌ها این مشکل را حل کنیم. چون در بقیه‌ی درخت این قواعد رعایت شده‌اند، پس پدر  $x$  (یعنی  $p[p[x]]$ ) قطعاً سیاه است (زیرا اگر قرمز بود، خاصیت ۴ در گره  $p[x]$  نقض می‌شد که فرض می‌کنیم تا قبل از درج  $x$  برقرار بوده است).

فرض می‌کنیم  $p[x]$  فرزند چپ  $p[p[x]]$  است (حالت قرینه نیز وجود دارد و به صورت مشابه حل می‌شود). چند حالت پیش می‌آید:

۱. \*\*حالت ۱:\*\* عمومی  $x$  (فرزنده راست  $p[p[x]]$  که آن را  $y$  می‌نامیم) هم قرمز باشد.

در این حالت،  $y$  و  $p[x]$  را سیاه می‌کنیم. حالا فرض می‌کنیم  $p[p[x]]$  عنصر درج شده است و همین عملیات را برای آن اجرا می‌کنیم (یعنی مشکل به سمت بالا در درخت منتقل می‌شود).

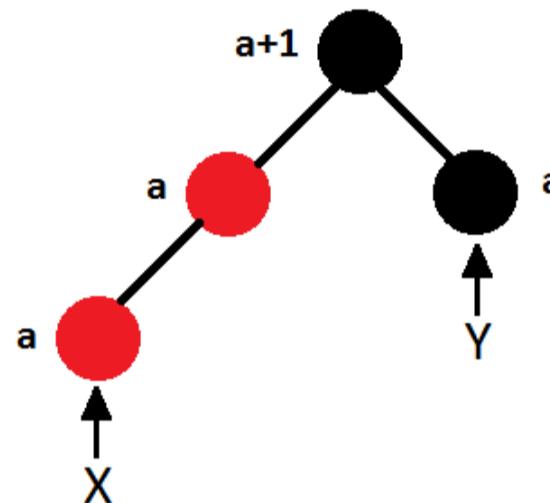


(اگر به ریشه رسیدیم، ریشه را سیاه می‌کنیم و عملیات به پایان می‌رسد).

۲. \*\*حالت ۲:\*\* عمومی  $x$  (یعنی  $y$ ) سیاه باشد.

در این صورت،  $x$  را به جای  $p[x]$  قرار می‌دهیم و روی  $x$  (که حالا نقش  $p[x]$  را دارد) چرخش به چپ انجام می‌دهیم. این کار باعث می‌شود  $x$  فرزند چپ پدر جدیدش (که همان  $p[p[x]]$  قبلی است) شود. و از این پس فرض می‌کنیم  $p[x]$  همان  $x$  است (یعنی گره‌ای که قبلًاً  $x$  بود، حالا در موقعیت  $p[x]$  قرار گرفته است). این وضعیت را به حالت ۲.۲ تبدیل می‌کند.

را سیاه می‌کنیم،  $p[p[x]]$  را قرمز می‌کنیم و در نهایت روی  $p[p[x]]$  یک چرخش به راست انجام می‌دهیم و عملیات به پایان می‌رسد.



در صورتی که  $p[x]$  فرزند راست پدرش باشد، همه‌ی این کارها به صورت قرینه انجام می‌شود. این فرآیند تضمین می‌کند که پس از درج، خواص درخت قرمز-سیاه حفظ شده و ارتفاع درخت لگاریتمی باقی بماند.

In [2]: در صورت نیاز، اگرچه در این پیاده‌سازی مستقیماً استفاده نمی‌شود  $\mathit{math.inf}$  برای استفاده از.

```
import math
# در صورت نیاز، اگرچه در این پیاده‌سازی مستقیماً استفاده نمی‌شود math.inf برای استفاده از.

class rbnode(object):
    """
    در درخت قرمز-سیاه (Node) یک گره.
    در اینجا، گره را به عنوان یک کلاس در Python نمایش داده شده است. این کلاس دارای مقداری از اعضای خصوصی است که در اینجا آنها را معرفی نمی‌کنیم.
    در اینجا، گره را به عنوان یک کلاس در Python نمایش داده شده است. این کلاس دارای مقداری از اعضای خصوصی است که در اینجا آنها را معرفی نمی‌کنیم.

    def __init__(self, key):
        """
        مقدار (کلید) گره.
        self._key = key # گره.
        self._red = False # برای سیاه، برای قرمز True.
        self._left = None # اشاره‌گر به فرزند چپ.
        self._right = None # اشاره‌گر به فرزند راست.
        self._p = None # اشاره‌گر به پدر گره.

        # برای دسترسی به اعضای داخلی با نامهای عمومی‌تر (properties) تعریف ویژگی‌ها.
        key = property(fget=lambda self: self._key, doc="The node's key")
        red = property(fget=lambda self: self._red, doc="Is the node red?")
        left = property(fget=lambda self: self._left, doc="The node's left child")
        right = property(fget=lambda self: self._right, doc="The node's right child")
        p = property(fget=lambda self: self._p, doc="The node's parent")

    def __str__(self):
        """
        نمایش رشته‌ای گره.
        return str(self.key)

    def __repr__(self):
        """
        نمایش رشته‌ای برای بازنمایی (representation).
        return f"rbnode({self.key}, {'R' if self.red else 'B'})"

class rbtree(object):
    """
    (Red-Black Tree). پیاده‌سازی یک درخت قرمز-سیاه.
    در اینجا، درخت قرمز-سیاه را به عنوان یک کلاس در Python نمایش داده شده است. این کلاس دارای مقداری از اعضای خصوصی است که در اینجا آنها را معرفی نمی‌کنیم.

    def __init__(self, create_node=rbnode):
        """
        مقدار درخت.
        self._nil = create_node(key=None) # NIL گره.
        self._nil._red = False # اطمینان از سیاه بودن NIL.
        self._root = self._nil # در ابتدا NIL ریشه درخت است.
        self._create_node = create_node # تابعی برای ایجاد گره جدید.

        root = property(fget=lambda self: self._root, doc="The tree's root node")
        nil = property(fget=lambda self: self._nil, doc="The tree's nil node")

    def search(self, key, x=None):
        """
        جستجو می‌کند (داده نشود  $x$  یا ریشه درخت اگر)  $x$  یک کلید را به صورت تکراری در زیردرخت ریشه
        از آن شروع می‌کند. این کار را تا آنکه یافته شود یا در درختی خارج از آن انتها می‌کند. این کار را تا آنکه یافته شود یا در درختی خارج از آن انتها می‌کند.
```

Args:

کلید مورد جستجو.  
key: گره‌ای که جستجو از آن شروع می‌شود. پیش‌فرض ریشه است.

Returns:  
اگر یافت نشود `rbnode` یا `None`: گرهای که شامل کلید است، یا `self.nil`.

```
"""
if x is None:
    x = self.root
while x != self.nil and key != x.key:
    if key < x.key:
        x = x.left
    else:
        x = x.right
return x
```

def minimum(self, x=None):
"""
پیدا میکند `x` کوچکترین مقدار (گره) را در زیردرخت ریشه.

Args:  
`x` (`rbnode`, optional): گرهای که جستجو از آن شروع میشود. پیشفرض ریشه است.

Returns:  
گرهای با حداقل مقدار `rbnode`.

```
"""
if x is None:
    x = self.root
while x.left != self.nil:
    x = x.left
return x
```

def maximum(self, x=None):
"""
پیدا میکند `x` بزرگترین مقدار (گره) را در زیردرخت ریشه.

Args:  
`x` (`rbnode`, optional): گرهای که جستجو از آن شروع میشود. پیشفرض ریشه است.

Returns:  
گرهای با حداکثر مقدار `rbnode`.

```
"""
if x is None:
    x = self.root
while x.right != self.nil:
    x = x.right
return x
```

def \_left\_rotate(self, x):
"""
را انجام میدهد `x` چرخش به چپ روی گره.

Args:  
گرهای که چرخش حول آن انجام میشود `x`.

"""
y = x.right # را فرزند راست `y` فرمود
x.\_right = y.left # تبدیل میکند `x` به فرزند راست `y` زیردرخت چپ.

if y.left != self.nil:
 y.left.\_p = x # قرار میدهد `x` را پدر فرزند چپ
 y.\_p = x.p # قرار میدهد `x` را پدر `y` میکند.

if x.p == self.nil: # ریشه جدید میشود `y`, ریشه بود `x` اگر
 self.\_root = y
elif x == x.p.left: # فرزند چپ بود `x` اگر
 x.p.\_left = y # قرار میدهد `y` را فرزند چپ پدر
else: # فرزند راست بود `x` اگر
 x.p.\_right = y # قرار میدهد `y` را فرزند راست پدر.

`y._left = x` # قرار میدهد `y` را فرزند چپ
`x._p = y` # قرار میدهد `y` را پدر `x` میکند.

def \_right\_rotate(self, y):
"""
را انجام میدهد `y` چرخش به راست روی گره.

Args:  
گرهای که چرخش حول آن انجام میشود `y`.

"""
x = y.left # را فرزند چپ
y.\_left = x.right # تبدیل میکند `y` را به فرزند چپ `x` زیردرخت راست.

if x.right != self.nil:
 x.right.\_p = y # قرار میدهد `y` را پدر فرزند راست
 x.\_p = y.p # قرار میدهد `y` را پدر `x` میکند.

if y.p == self.nil: # ریشه جدید میشود `x`, ریشه بود `y` اگر
 self.\_root = x
elif y == y.p.right: # فرزند راست بود `y` اگر
 y.p.\_right = x # قرار میدهد `x` را فرزند راست پدر
else: # فرزند چپ بود `y` اگر
 y.p.\_left = x # قرار میدهد `x` را فرزند چپ پدر.

`x._right = y` # قرار میدهد `x` را فرزند راست
`y._p = x` # قرار میدهد `x` را پدر `y` میکند.

def insert\_key(self, key):
"کلید را در درخت درج میکند."
self.insert\_node(self.\_create\_node(key=key))

def insert\_node(self, z):
"را در درخت درج میکند."
y = self.nil # پدر گره جدید خواهد بود `y`.

```

x = self.root # برای پیمایش درخت استفاده می‌شود x.
پیمایش درخت برای یافتن مکان درج.

while x != self.nil:
    y = x
    if z.key < x.key:
        x = x.left
    else:
        x = x.right

z._p = y # قرار می‌دهد y پدر گره جدید را.

# اتصال گره جدید به پدرش.
if y == self.nil: # بود، درخت خالی است و y NIL.
    self._root = z
elif z.key < y.key: # اگر می‌شود y بود، فرزند چپ y کوچکتر از z.
    y._left = z
else: # اگر می‌شود y بود، فرزند راست y بزرگتر یا مساوی z.
    y._right = z

قرار می‌دهد NIL فرزندان گره جدید را.
z._left = self.nil
z._right = self.nil
z._red = True # گره جدید را قرمز می‌کند.

فراخوانیتابع اصلاح برای حفظ خواص قرمز-سیاه.

def _insert_fixup(self, z):
    """ بازیابی می‌کند z خواص قرمز-سیاه را پس از درج گره.
    قرمز باشد (نقض خاصیت 4) تا زمانی که پدر #.
    while z.p.red:
        فرزند چپ پدر بزرگش بود اگر پدر.
        if z.p == z.p.p.left:
            است (z برادر پدر) عموی y z.
            if y.red: # عموی.
                z.p._red = False # پدر z را سیاه کن.
                y._red = False # عموی.
                z.p.p._red = True # پدر بزرگ را قرمز کن z.
                z = z.p.p # منتقل کن z مشکل را به سمت پدر بزرگ.
            else: # سیاه است z.
                فرزند راست پدرش است z: حالت 2.1.
                if z == z.p.right: # 2.1
                    را به پدرش منتقل کن.
                    self._left_rotate(z) # چرخش به چپ روی پدر z.
                    z.p._red = False # را سیاه کن z.
                    z.p.p._red = True # را قرمز کن z پدر بزرگ.
                    self._right_rotate(z.p.p) # چرخش به راست روی پدر بزرگ.
                فرزند راست پدر بزرگش بود (حالت قرینه).
        else: # اگر پدر.
            y = z.p.left # y عموی z.
            if y.red: # عموی (قرینه).
                z.p._red = False
                y._red = False
                z.p.p._red = True
                z = z.p.p
            else: # سیاه است z: حالت 2 (قرینه).
                فرزند چپ پدرش است z: حالت 2.1 (قرینه).
                if z == z.p.left: # 2.1
                    چرخش به راست روی پدر z.
                    self._right_rotate(z) # چرخش به چپ روی پدر z.
                    z.p._red = False # را سیاه کن z.
                    z.p.p._red = True # را قرمز کن z پدر بزرگ.
                    self._left_rotate(z.p.p) # چرخش به چپ روی پدر بزرگ.
                فرزند همیشه سیاه است #.
    self.root._red = False #.

def check_invariants(self):
    """ بررسی می‌کند که آیا درخت تمام خواص درخت قرمز-سیاه را برآورده می‌کند """
    def is_red_black_node(node):
        """ بررسی می‌کند node به صورت بازگشتی خواص را برای زیردرخت گره "برگ‌ها) NIL بررسی گره‌های #.
        if node == self.nil:
            سیاه است و عمق سیاه 1.
        return 1, True # NIL 1.

        خاصیت 4: اگر گره قرمز است، فرزندانش باید سیاه باشند #.
        if node.red:
            if node.left.red or node.right.red:
                return 0, False # 4. نقض خاصیت.

        بررسی بازگشتی فرزندان #.
        left_counts, left_ok = is_red_black_node(node.left)
        if not left_ok:
            return 0, False
        right_counts, right_ok = is_red_black_node(node.right)
        if not right_ok:
            return 0, False

        خاصیت 5: تعداد گره‌های سیاه در مسیرها یکسان باشد #.
        if left_counts != right_counts:
            return 0, False # 5. نقض خاصیت.

        برگرداندن عمق سیاه و وضعیت صحت #.
        return left_counts + (1 if not node.red else 0), True

        خاصیت 2: ریشه باید سیاه باشد #.
        if self.root.red:
            print("نقض خاصیت 2: ریشه قرمز است")
            return False

        num_black, is_ok = is_red_black_node(self.root)
        if not is_ok:
            print("نقض خواص قرمز-سیاه در زیردرخت")
        return is_ok

```

```

def inorder_traverse(self, node):
    if node != self.nil:
        self.inorder_traverse(node.left)
        print(f"Key: {node.key}, Color: {'R' if node.red else 'B'}, Parent: {node.p.key if node.p != self.nil else 'None'}")
        self.inorder_traverse(node.right)

# -----
# مثال استفاده
# ----

print("--- ساخت و درج در درخت قرمز-سیاه ---")
rbt_instance = rbtree()
keys_to_insert = [10, 20, 30, 15, 25, 5]

for key in keys_to_insert:
    print(f"\nدرج کلید: {key}")
    rbt_instance.insert_key(key)
    print("درخت بعد از درج (پیماش میانترتب) :")
    rbt_instance.inorder_traverse(rbt_instance.root)
    print(f"آیا خواص درخت قرمز-سیاه حفظ شده است؟ {rbt_instance.check_invariants()}")


print("\n--- بررسی نهایی درخت ---")
print("درخت نهایی (پیماش میانترتب) :")
rbt_instance.inorder_traverse(rbt_instance.root)
print(f"آیا خواص درخت قرمز-سیاه حفظ شده است؟ {rbt_instance.check_invariants()}")


# تست جستجو
print("\n--- تست جستجو ---")
found_node = rbt_instance.search(15)
if found_node != rbt_instance.nil:
    print(f"کلید 15 یافت شد: {found_node}")
else:
    print("کلید 15 یافت نشد")

not_found_node = rbt_instance.search(100)
if not_found_node != rbt_instance.nil:
    print(f"کلید 100 یافت شد: {not_found_node}")
else:
    print("کلید 100 یافت نشد")

# تست حداقل و حداکثر
print("\n--- تست حداقل و حداکثر ---")
min_node = rbt_instance.minimum()
print(f"کوچکترین کلید در درخت: {min_node.key}")
max_node = rbt_instance.maximum()
print(f"بزرگترین کلید در درخت: {max_node.key}")

```

درج کلید: 10

: درخت بعد از درج (پیمایش میانترتب)

Key: 10, Color: B, Parent: None

آیا خواص درخت قرمز-سیاه حفظ شده است؟ True

درج کلید: 20

: درخت بعد از درج (پیمایش میانترتب)

Key: 10, Color: B, Parent: None

Key: 20, Color: R, Parent: 10

آیا خواص درخت قرمز-سیاه حفظ شده است؟ True

درج کلید: 30

: درخت بعد از درج (پیمایش میانترتب)

Key: 10, Color: R, Parent: 20

Key: 20, Color: B, Parent: None

Key: 30, Color: R, Parent: 20

آیا خواص درخت قرمز-سیاه حفظ شده است؟ True

درج کلید: 15

: درخت بعد از درج (پیمایش میانترتب)

Key: 10, Color: B, Parent: 20

Key: 15, Color: R, Parent: 10

Key: 20, Color: B, Parent: None

Key: 30, Color: B, Parent: 20

آیا خواص درخت قرمز-سیاه حفظ شده است؟ True

درج کلید: 25

: درخت بعد از درج (پیمایش میانترتب)

Key: 10, Color: B, Parent: 20

Key: 15, Color: R, Parent: 10

Key: 20, Color: B, Parent: None

Key: 25, Color: R, Parent: 30

Key: 30, Color: B, Parent: 20

آیا خواص درخت قرمز-سیاه حفظ شده است؟ True

درج کلید: 5

: درخت بعد از درج (پیمایش میانترتب)

Key: 5, Color: R, Parent: 10

Key: 10, Color: B, Parent: 20

Key: 15, Color: R, Parent: 10

Key: 20, Color: B, Parent: None

Key: 25, Color: R, Parent: 30

Key: 30, Color: B, Parent: 20

آیا خواص درخت قرمز-سیاه حفظ شده است؟ True

--- بررسی نهایی درخت ---

: درخت نهایی (پیمایش میانترتب)

Key: 5, Color: R, Parent: 10

Key: 10, Color: B, Parent: 20

Key: 15, Color: R, Parent: 10

Key: 20, Color: B, Parent: None

Key: 25, Color: R, Parent: 30

Key: 30, Color: B, Parent: 20

آیا خواص درخت قرمز-سیاه حفظ شده است؟ True

--- تست جستجو ---

کلید 15 یافت شد: 15

کلید 100 یافت نشد.

--- تست حداقل و حداکثر ---

کوچکترین کلید در درخت: 5

بزرگترین کلید در درخت: 30

```
In [3]: RB = rbtree()
RB.insert_key(10)
RB.insert_key(11)
RB.insert_key(12)

# جستجو برای کلید 12
if RB.search(12) != RB.nil and RB.search(12).key == 12:
    print("12 is in tree")
else:
    print("12 is not in tree")

# جستجو برای کلید 13
if RB.search(13) != RB.nil and RB.search(13).key == 13:
    print("13 is in tree")
else:
    print("13 is not in tree")
```

12 is in tree

13 is not in tree

## حذف (Deletion)

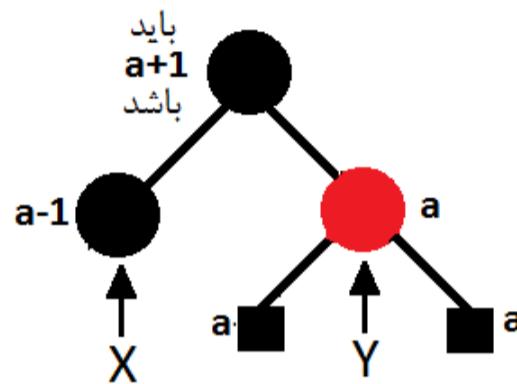
در عملیات حذف نیز ابتدا مانند عمل حذف در یک BST عادی عمل می‌کنیم. اگر رأسی که حذف کردیم قرمز باشد، هیچ کدام از خواص درخت نقض نخواهد شد و عملیات به پایان می‌رسد. ولی اگر رأس حذف شده

سیاه باشد، خاصیت پنجم (تعداد رأس‌های سیاه در مسیرها) نقض خواهد شد.

فرض می‌کنیم  $x$  فرزندی است که یا واقعاً حذف شده (در حالت برگ یا یک فرزند) یا جای آن با جانشینیش عوض شده و حالا  $x$  نشان‌دهنده گره‌ای است که مشکل رنگ سیاه را ایجاد کرده است. همچنین فرض می‌کنیم  $x$  فرزند چپ پدرش است (حالت قرینه نیز وجود دارد و به صورت مشابه حل می‌شود). اگر فرزند راست  $p[x]$  را  $y$  (برادر  $x$ ) بنامیم، چند حالت داریم:

\*\*حالت ۱: برادر  $x$  ( $y$ ) قرمز باشد. ۱.

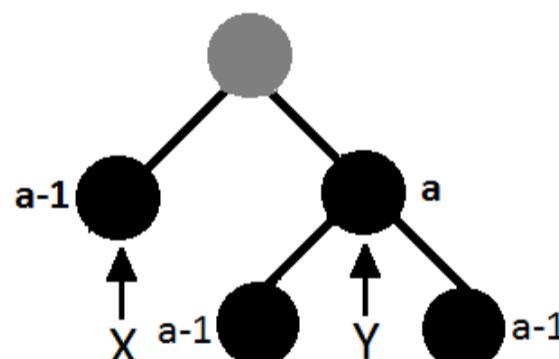
در این حالت،  $y$  را سیاه می‌کنیم و  $p[x]$  را همان  $[y]$  است (که همان  $p[x]$  است) را قرمز می‌کنیم. بعد روی  $p[x]$  یک چرخش به چپ انجام می‌دهیم. در این صورت، فرزند راست  $p[x]$  که فرزند چپ قبلی  $y$  بوده، قطعاً سیاه است (زیرا  $y$  قرمز بوده و فرزندانش باید سیاه باشند). و فرزند راستش قرمز است و وارد حالت دوم می‌شود.



\*\*حالت ۲: برادر  $x$  ( $y$ ) سیاه باشد. ۲.

\*\*حالت ۲.۱: هر دو فرزند  $y$  سیاه باشند. A.

$y$  را قرمز می‌کنیم و از این پس فرض می‌کنیم  $p[x]$  همان  $x$  جدید است. و همین عملیات را برای آن از اول انجام می‌دهیم (مشکل به سمت بالا منتقل می‌شود).

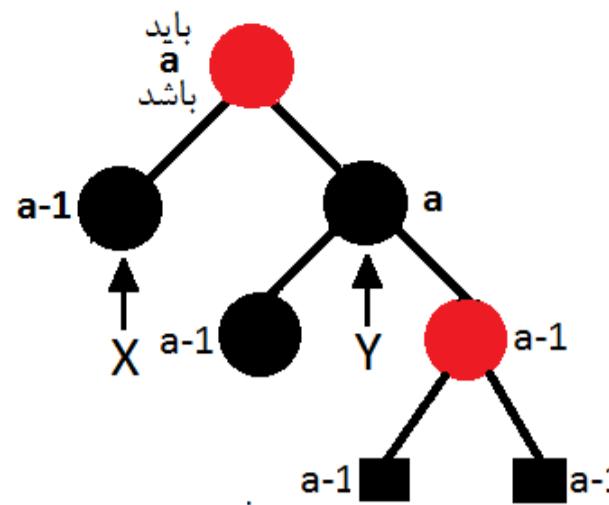


\*\*حالت ۲.۲: فرزند چپ  $y$  قرمز و فرزند راستش سیاه باشد. B.

فرزند چپ  $y$  را سیاه می‌کنیم و خود  $y$  را قرمز می‌کنیم و روی  $y$  یک چرخش به راست انجام می‌دهیم. با این کار، فرزند چپ رأسی که جای  $y$  می‌آید سیاه می‌شود (زیرا فرزند چپ  $y$  قبل از شروع تغییرات، قرمز بوده است). و فرزند راستش قرمز است و وارد حالت ۲.۳ می‌شود.

\*\*حالت ۲.۳: فرزند راست  $y$  قرمز و فرزند چپش سیاه باشد. C.

رنگ  $y$  را با رنگ  $p[x]$  عوض می‌کنیم.  $p[x]$  را سیاه می‌کنیم و فرزند راست  $y$  را سیاه می‌کنیم. در نهایت، روی  $p[x]$  یک چرخش به چپ انجام می‌دهیم و عملیات تمام می‌شود.



اگر  $x$  فرزند راست پدرش باشد، همهی این عملیات به صورت قرینه انجام می‌شود. این فرآیند تضمین می‌کند که پس از حذف، خواص درخت قرمز-سیاه حفظ شده و ارتفاع درخت لگاریتمی باقی بماند.

In [4]: `import math # برای استفاده از math.inf برای ایجاد نمی‌شود`.

```
class rbnode(object):
    """
    در درخت قرمز-سیاه (Node) یک گره
    (ویرایش دوم، صفحه 273 Cormen, Leiserson, Rivest, Stein) بر اساس پیاده‌سازی
    """

    def __init__(self, key):
        """
        مقدار (کلید) گره
        سیاه است
        مقدار (کلید) گره
        سیاه است
        اشاره‌گر به فرزند چپ
        اشاره‌گر به فرزند راست
        اشاره‌گر به پدر گره
        """
        self._key = key # مقدار (کلید) گره
        self._red = False # رنگ گره
        self._left = None # اشاره‌گر به فرزند چپ
        self._right = None # اشاره‌گر به فرزند راست
        self._p = None # اشاره‌گر به پدر گره

        # برای دسترسی به اعضای داخلی با نامهای عمومی‌تر (properties) تعریف ویژگی‌ها
        key = property(fget=lambda self: self._key, doc="The node's key")
        red = property(fget=lambda self: self._red, doc="Is the node red?")
        left = property(fget=lambda self: self._left, doc="The node's left child")
        right = property(fget=lambda self: self._right, doc="The node's right child")
        p = property(fget=lambda self: self._p, doc="The node's parent")

    def __str__(self):
        """
        نمایش رشته‌ای گره
        برای رشته‌ای بازنمایی
        """
        return str(self.key)

    def __repr__(self):
        """
        نمایش رشته‌ای برای بازنمایی (representation).
        """
        return f"rbnode({self.key}, {'R' if self.red else 'B'})"

class rbtree(object):
    """
    پیاده‌سازی یک درخت قرمز-سیاه (Red-Black Tree).
    (ویرایش دوم، صفحه 273 Cormen, Leiserson, Rivest, Stein) بر اساس پیاده‌سازی
    """

    def __init__(self, create_node=rbnode):
        """
        مرا برگ‌ها استفاده می‌شود و همیشه سیاه است (بوج) NIL گره
        اطمینان از سیاه بودن NIL
        در ابتدا NIL ریشه درخت، در ابتدا
        تابعی برای ایجاد گره جدید
        """
        self._nil = create_node(key=None) # گره NIL
        self._nil._red = False # اطمینان از سیاه بودن NIL
        self._root = self._nil # ریشه درخت، در ابتدا
        self._create_node = create_node # تابعی برای ایجاد گره جدید

        root = property(fget=lambda self: self._root, doc="The tree's root node")
        nil = property(fget=lambda self: self._nil, doc="The tree's nil node")

    def search(self, key, x=None):
        """
        جستجو می‌کند (داده نشود x یا ریشه درخت اگر) x یک کلید را به صورت تکراری در زیردرخت ریشه
        """
        Args:
            key: کلید مورد جستجو
            x (rbnode, optional): گره‌ای که جستجو از آن شروع می‌شود. پیش‌فرض ریشه است
        Returns:
            rbnod or None: اگر یافت نشود گره‌ای که شامل کلید است، یا
        """
        if x is None:
            x = self.root
        while x != self.nil and key != x.key:
            if key < x.key:
                x = x.left
            else:
                x = x.right
        return x

    def minimum(self, x=None):
        """
        پیدا می‌کند x کوچکترین مقدار (گره) را در زیردرخت ریشه
        """
        Args:
```

گره‌ای که جستجو از آن شروع می‌شود. پیش‌فرض ریشه است: `x` (rbnode, optional).

Returns:

گره‌ای با حداقل مقدار.

"""

`if x is None:`

`x = self.root`

`while x.left != self.nil:`

`x = x.left`

`return x`

`def maximum(self, x=None):`

"""

پیدا می‌کند `x` بزرگترین مقدار (گره) را در زیردرخت ریشه.

Args:

گره‌ای که جستجو از آن شروع می‌شود. پیش‌فرض ریشه است: `x` (rbnode, optional).

Returns:

گره‌ای با حداکثر مقدار.

"""

`if x is None:`

`x = self.root`

`while x.right != self.nil:`

`x = x.right`

`return x`

`def _left_rotate(self, x):`

"""

را انجام می‌دهد `x` چرخش به چپ روی گره.

Args:

گره‌ای که چرخش حول آن انجام می‌شود: `x` (rbnode).

"""

قرار می‌دهد `x` را فرزند راست `y` قرار می‌دهد.

تبدیل می‌کند `x` را به فرزند راست `y` زیردرخت چپ.

`if y.left != self.nil:`

`y.left._p = x #` را پدر فرزند چپ.

`y._p = x.p #` را پدر `y` پدر.

`if x.p == self.nil:` ریشه جدید می‌شود `y`, ریشه بود `x` اگر.

`self._root = y`

`elif x == x.p.left:` فرزند چپ پدرش بود `x` اگر.

`x.p._left = y #` را پدر فرزند چپ پدر.

`else:` فرزند راست پدرش بود `x` اگر.

`x.p._right = y #` را فرزند راست پدر.

`y._left = x #` را فرزند چپ.

`x._p = y #` را پدر.

`def _right_rotate(self, y):`

"""

را انجام می‌دهد `y` چرخش به راست روی گره.

Args:

گره‌ای که چرخش حول آن انجام می‌شود: `y` (rbnode).

"""

قرار می‌دهد `y` را فرزند چپ.

تبدیل می‌کند `y` را به فرزند چپ `x` زیردرخت راست.

`if x.right != self.nil:`

`x.right._p = y #` را پدر فرزند راست.

`x._p = y.p #` را پدر.

`if y.p == self.nil:` ریشه جدید می‌شود `x`, ریشه بود `y` اگر.

`self._root = x`

`elif y == y.p.right:` فرزند راست پدرش بود `y` اگر.

`y.p._right = x #` را پدر فرزند راست پدر.

`else:` فرزند چپ پدرش بود `y` اگر.

`y.p._left = x #` را فرزند چپ پدر.

`x._right = y #` را فرزند راست.

`y._p = x #` را پدر.

`def insert_key(self, key):`

"کلید را در درخت درج می‌کند"

`self.insert_node(self._create_node(key=key))`

`def insert_node(self, z):`

"را در درخت درج می‌کند `z` گره."

`y = self.nil #` پدر گره جدید خواهد بود `y`.

برای پیمایش درخت استفاده می‌شود `x`.

#پیمایش درخت برای یافتن مکان درج.

`while x != self.nil:`

`y = x`

`if z.key < x.key:`

`x = x.left`

`else:`

`x = x.right`

`z._p = y #` را پدر گره جدید.

#اتصال گره جدید به پدرش.

`if y == self.nil: #` را اگر `y NIL` ریشه می‌شود `z` بود، درخت خالی است و.

`self._root = z`

می‌شود `z` بود، فرزند چپ `y` کوچکتر از `z` اگر.

`y._left = z`

می‌شود `y` بود، فرزند راست `y` بزرگتر یا مساوی `z` است.

`else: #` اگر `z`

```

y._right = z

# قرار می‌دهد NIL فرزندهان گره جدید را
z._left = self.nil
z._right = self.nil
z._red = True # گره جدید را قرمز می‌کند.

فراخوانیتابع اصلاح برای حفظ خواص قرمز-سیاه (self._insert_fixup(z) # فرمز خواص قرمز-سیاه).

def _insert_fixup(self, z):
    """بازیابی می‌کند z خواص قرمز-سیاه را پس از درج گره.
    قرمز باشد (نقض خاصیت 4) z تا زمانی که پدر # while z.p.red:
        فرزند چپ پدر بزرگش بود اگر پدر:
        if z.p == z.p.p.left:
            است (z برادر پدر) z عمومی y است:
            if y.red: # عمومی
                قرمز است z حالت 1: عمومی
                z.p._red = False # پدر z سیاه کن
                y._red = False # سیاه کن z عمومی
                را قرمز کن z پدر بزرگ
                z.p.p._red = True # مشکل را به سمت پدر بزرگ
                z = z.p.p # منتقل کن z
            else: # سیاه است z حالت 2: عمومی
                فرزند راست پدرش است z: حالت 2.1
                z = z.p # z را به پدرش منتقل کن
                چرخش به چپ روی پدر # z
                self._left_rotate(z)
                z.p._red = False # سیاه کن z
                را قرمز کن z پدر بزرگ
                z.p.p._red = True # سیاه کن z
                self._right_rotate(z.p.p) # z
            else: # اگر پدر z
                فرزند راست پدر بزرگش بود (حالت قرینه)
                y = z.p.p.left # y عمومی
                است y z قرمز است حالت 1 (قرینه): عمومی
                z.p._red = False
                y._red = False
                z.p.p._red = True
                z = z.p.p
            else: # سیاه است z حالت 2 (قرینه): عمومی
                فرزند چپ پدرش است z: حالت 2.1 (قرینه)
                z = z.p
                چرخش به راست روی پدر # z
                را سیاه کن z
                را قرمز کن z پدر بزرگ
                self._left_rotate(z.p.p) # z
                چرخش به چپ روی پدر بزرگ # z
                ریشه همیشه سیاه است
                self.root._red = False # ریشه همیشه سیاه است

def check_invariants(self):
    """بررسی می‌کند که آیا درخت تمام خواص درخت قرمز-سیاه را برآورده می‌کند"""

def is_red_black_node(node):
    """بررسی می‌کند node به صورت بازگشتی خواص را برای زیردرخت گره "برگ‌ها" NIL بررسی گردهای
    # if node == self.nil:
        ریشه است و عمق سیاه 1
    return 1, True # NIL 1

    خاصیت 4: اگر گره قرمز است، فرزندانش باید سیاه باشند
    if node.red:
        if node.left.red or node.right.red:
            return 0, False # 4 نقض خاصیت 4

    بزرگداشت عمق سیاه و وضعیت صحت
    left_counts, left_ok = is_red_black_node(node.left)
    if not left_ok:
        return 0, False
    right_counts, right_ok = is_red_black_node(node.right)
    if not right_ok:
        return 0, False

    خاصیت 5: تعداد گردهای سیاه در مسیرها یکسان باشد
    if left_counts != right_counts:
        return 0, False # 5 نقض خاصیت 5

    برگرداندن عمق سیاه و وضعیت صحت
    return left_counts + (1 if not node.red else 0), True

    خاصیت 2: ریشه باید سیاه باشد
    if self.root.red:
        print("نقض خاصیت 2: ریشه قرمز است")
        return False

    num_black, is_ok = is_red_black_node(self.root)
    if not is_ok:
        print("نقض خواص قرمز-سیاه در زیردرخت")
    return is_ok

def inorder_traverse(self, node):
    if node != self.nil:
        self.inorder_traverse(node.left)
        print(f"Key: {node.key}, Color: {'R' if node.red else 'B'}, Parent: {node.p.key if node.p != self.nil else 'None'}")
        self.inorder_traverse(node.right)

    # --- توابع حذف ---
def _transplant(self, u, v):
    """
    جایگزین می‌کند v را با زیردرخت ریشه u این تابع زیردرخت ریشه
    """
    if u.p == self.nil: # اگر
        self._root = v
    elif u == u.p.left: # اگر u
        u.p._left = v
    else: # اگر u
        u.p._right = v
    v._p = u.p # ریشه می‌دهد u را پدر v پسر

```

```

def delete_key(self, key):
    """
    کلید را از درخت حذف می‌کند.
    """

    z = self.search(key)
    if z == self.nil:
        return False # کلید یافت نشد.
    self.delete_node(z)
    return True

def delete_node(self, z):
    """
    را از درخت حذف می‌کند گرہ.
    """

    y = z # y به موقعیت دیگری منتقل می‌شود
    y_original_color = y.red # رنگ اصلی.

    if z.left == self.nil: # گره 1: فرزند چپ ندارد
        x = z.right # x است فرزند راست z.
        را با فرزند راستش جایگزین می‌کند
        self._transplant(z, z.right) # z فقط فرزند راست ندارد (فقط چپ دارد) z حالت 2: گره 2
        elif z.right == self.nil: # گره 2: فرزند چپ دارد
            x = z.left # x است فرزند چپ z.
            self._transplant(z, z.left) # z دو فرزند دارد z حالت 3: گره 3
        else: # گره 3: کوچکترین در زیردرخت راست z جاشینیم
            y = self.minimum(z.right) # y را ذخیره می‌کند و رنگ اصلی.
            y_original_color = y.red # را با فرزند چپش جایگزین می‌کند
            self._transplant(z, y) # z است (کوچکترین در زیردرخت راست) z جاشینیم
            y = y.right # x است y فرزند راست.
            if y.p == z: # بود y فرزند مستقیم y اگر
                x.p = y # همان nil قرار می‌دهد y را پدر.
            else:
                self._transplant(y, y.right) # y را با فرزند راست z فرزند چپ
                قرار می‌دهد y را فرزند راست z فرزند چپ
                قرار می‌دهد y را پدر فرزند راست
                y.right.p = y # قرار می‌دهد y را پدر فرزند راست.

            جایگزین می‌کند y را با
            self._transplant(z, y) # z قرار می‌دهد y را فرزند چپ z
            قرار می‌دهد y را فرزند چپ y.left # z فرزند چپ
            قرار می‌دهد y را فرزند چپ y.left.p = y # پدر فرزند چپ
            قرار می‌دهد z را رنگ y._red = z.red # رنگ y را رنگ z می‌زنیم.

        if not y_original_color: # سیاه بود، (y) اگر گره ای که واقعاً حذف شد
            self._delete_fixup(x) # فراخوانی تابع اصلاح برای حفظ خواص قرمز-سیاه.

    def _delete_fixup(self, x):
        """
        خواص قرمز-سیاه را پس از حذف بازیابی می‌کند.
        """

        while x != self.root and not x.red: # ریشه نباشد و سیاه باشد x تا زمانی که
            if x == x.p.left: # فرزند چپ پدرش بود اگر
                w = x.p.right # x برادر
                if w.red: # برادر 1: قرمز است x حالت 1
                    w.red = False # w سیاه کن
                    x.p._red = True # x قرمز کن
                    self._left_rotate(x.p) # چرخش به چپ روی پدر
                    w = x.p.right # x برادر جدید
                if not w.left.red and not w.right.red: # برادر 2: سیاه است و هر دو فرزندش سیاه هستند
                    w.red = True # w قرمز کن
                    x = x.p # منتقل کن x مشکل را به سمت پدر
                else:
                    if not w.right.red: # برادر 3: سیاه است، فرزند چپش قرمز و فرزند راستش سیاه است
                        w.left._red = False # را سیاه کن w فرزند چپ
                        w._red = True # w قرمز کن
                        self._right_rotate(w) # چرخش به راست روی w
                        w = x.p.right # x برادر جدید
                    if not w.red and not w.left.red: # برادر 4: سیاه است و فرزند راستش قرمز است
                        w._red = x.p.red # قرار می‌دهد x را رنگ پدر w رنگ
                        x.p._red = False # را سیاه می‌کند x پدر
                        w.right._red = False # فرزند راست w
                        self._left_rotate(x.p) # چرخش به چپ روی پدر
                        x = self.root # x را به ریشه منتقل کن تا حلقه پایان یابد x، عملیات تمام شد
                    else: # فرزند راست پدرش بود (حالت قرینه) x اگر
                        w = x.p.left # x برادر
                        if w.red: # برادر 1 (قرینه): قرمز است x حالت 1
                            w._red = False
                            x.p._red = True
                            self._right_rotate(x.p)
                            w = x.p.left
                        if not w.right.red and not w.left.red: # برادر 2 (قرینه): سیاه است و هر دو فرزندش سیاه هستند
                            w._red = True
                            x = x.p
                        else:
                            if not w.left.red: # برادر 3 (قرینه): سیاه است، فرزند راستش قرمز و فرزند چپش سیاه است
                                w.right._red = False
                                w._red = True
                                self._left_rotate(w)
                                w = x.p.left
                            if not w.red and not w.left.red: # برادر 4 (قرینه): سیاه است و فرزند چپش قرمز است
                                w._red = x.p.red
                                x.p._red = False
                                w.left._red = False
                                self._right_rotate(x.p)
                                x = self.root
                    x._red = False # گره نهایی را سیاه می‌کند
        # -----
        # مثال استفاده از حذف

```

```

# -----
print("--- ساخت و درج در درخت قرمز-سیاه")
rb_tree_del = rbtree()
keys = [10, 20, 30, 15, 25, 5, 35]
for key in keys:
    rb_tree_del.insert_key(key)
    print(f"درج کلید: {key}. وضعیت درخت: {rb_tree_del.check_invariants()}")


print("\n--- درخت پس از درج (پیمایش میانترتب) ---")
rb_tree_del.inorder_traverse(rb_tree_del.root)
print(f"آیا درخت قرمز-سیاه است? {rb_tree_del.check_invariants()}")


# حذف یک گره (مثال 20)
print("\n--- حذف گره 20 ---")
rb_tree_del.delete_key(20)
print("درخت پس از حذف 20 (پیمایش میانترتب):")
rb_tree_del.inorder_traverse(rb_tree_del.root)
print(f"آیا درخت قرمز-سیاه است? {rb_tree_del.check_invariants()}")


# حذف یک گره دیگر (مثال 10)
print("\n--- حذف گره 10 ---")
rb_tree_del.delete_key(10)
print("درخت پس از حذف 10 (پیمایش میانترتب):")
rb_tree_del.inorder_traverse(rb_tree_del.root)
print(f"آیا درخت قرمز-سیاه است? {rb_tree_del.check_invariants()}")


# حذف یک گره بزرگ (مثال 5)
print("\n--- حذف گره 5 (بزرگ) ---")
rb_tree_del.delete_key(5)
print("درخت پس از حذف 5 (پیمایش میانترتب):")
rb_tree_del.inorder_traverse(rb_tree_del.root)
print(f"آیا درخت قرمز-سیاه است? {rb_tree_del.check_invariants()}")


# حذف ریشه فعلی (مثال 25)
print("\n--- حذف گره 25 (ریشه فعلی) ---")
rb_tree_del.delete_key(25)
print("درخت پس از حذف 25 (پیمایش میانترتب):")
rb_tree_del.inorder_traverse(rb_tree_del.root)
print(f"آیا درخت قرمز-سیاه است? {rb_tree_del.check_invariants()}")


# تلاش برای حذف گره ناموجود
print("\n--- تلاش برای حذف گره 100 (ناموجود) ---")
if not rb_tree_del.delete_key(100):
    print("کلید 100 در درخت یافت نشد و حذف نشد")
print("درخت پس از تلاش برای حذف 100 (پیمایش میانترتب):")
rb_tree_del.inorder_traverse(rb_tree_del.root)
print(f"آیا درخت قرمز-سیاه است? {rb_tree_del.check_invariants()}")

```

--- ساخت و درج در درخت قرمز-سیاه ---  
درج کلید: 10. وضعیت درخت True  
درج کلید: 20. وضعیت درخت True  
درج کلید: 30. وضعیت درخت True  
درج کلید: 15. وضعیت درخت True  
درج کلید: 25. وضعیت درخت True  
درج کلید: 5. وضعیت درخت True  
درج کلید: 35. وضعیت درخت True

--- درخت پس از درج (پیمایش میانترتب) ---  
Key: 5, Color: R, Parent: 10  
Key: 10, Color: B, Parent: 20  
Key: 15, Color: R, Parent: 10  
Key: 20, Color: B, Parent: None  
Key: 25, Color: R, Parent: 30  
Key: 30, Color: B, Parent: 20  
Key: 35, Color: R, Parent: 30  
آیا درخت قرمز-سیاه است? True

--- حذف گره 20 ---  
درخت پس از حذف 20 (پیمایش میانترتب)  
Key: 5, Color: R, Parent: 10  
Key: 10, Color: B, Parent: 25  
Key: 15, Color: R, Parent: 10  
Key: 25, Color: B, Parent: None  
Key: 30, Color: B, Parent: 25  
Key: 35, Color: R, Parent: 30  
آیا درخت قرمز-سیاه است? True

--- حذف گره 10 ---  
درخت پس از حذف 10 (پیمایش میانترتب)  
Key: 5, Color: R, Parent: 15  
Key: 15, Color: B, Parent: 25  
Key: 25, Color: B, Parent: None  
Key: 30, Color: B, Parent: 25  
Key: 35, Color: R, Parent: 30  
آیا درخت قرمز-سیاه است? True

--- حذف گره 5 (برگ) ---  
درخت پس از حذف 5 (پیمایش میانترتب)  
Key: 15, Color: B, Parent: 25  
Key: 25, Color: B, Parent: None  
Key: 30, Color: B, Parent: 25  
Key: 35, Color: R, Parent: 30  
آیا درخت قرمز-سیاه است? True

--- حذف گره 25 (ریشه فعلی) ---  
درخت پس از حذف 25 (پیمایش میانترتب)  
Key: 15, Color: B, Parent: 30  
Key: 30, Color: B, Parent: None  
Key: 35, Color: B, Parent: 30  
آیا درخت قرمز-سیاه است? True

--- تلاش برای حذف گره 100 (ناموجود) ---  
کلید 100 در درخت یافت نشد و حذف نشد.  
درخت پس از تلاش برای حذف 100 (پیمایش میانترتب)  
Key: 15, Color: B, Parent: 30  
Key: 30, Color: B, Parent: None  
Key: 35, Color: B, Parent: 30  
آیا درخت قرمز-سیاه است? True

## چرا درخت قرمز-سیاه متوازن است؟

بعد از تعریف کردن الگوریتم‌های بالا، سوالی که پیش می‌آید این است که چرا این ساختار ارتفاع لگاریتمی را تضمین می‌کند؟

مستقل از الگوریتم‌های ذکر شده و با توجه به خاصیت‌های درخت قرمز-سیاه که در ابتدای توصیف درخت ذکر شدند، ثابت کنید که در هر زیردرخت، فاصله‌ی بین دورترین برگ از ریشه آن زیردرخت نمی‌تواند بیشتر از دو برابر فاصله‌ی نزدیکترین برگ از ریشه باشد.

\*\*: اثبات شهودی

- خاصیت ۵ (عمق سیاه):\*\* تمام مسیرهای ساده از یک گره به گرههای **NIL** زیردرختش، تعداد یکسانی گره سیاه دارند.
- خاصیت ۴ (قرمز-قرمز ممنوع):\*\* هیچ دو گره قرمز نمی‌توانند پشت سر هم در یک مسیر قرار گیرند. این بدان معناست که در هر مسیر، حداقل به تعداد گرههای سیاه، گره قرمز نیز وجود دارد (یا به عبارت دیگر، تعداد گرههای قرمز حداقل برابر با تعداد گرههای سیاه است).

با ترکیب این دو خاصیت، می‌توان نتیجه گرفت که طول بلندترین مسیر (که بیشترین تعداد گرههای قرمز را دارد) حداقل دو برابر طول کوتاهترین مسیر (که فقط گرههای سیاه را دارد) است. این تضمین می‌کند که ارتفاع درخت همواره در حدود  $O(\log n)$  باقی بماند، زیرا درخت هرگز به طور کامل کج نمی‌شود.

## تمرین عملی با درختهای سیاه قرمز

در این لینک می‌توانید به درج کردن و حذف در یک درخت قرمز-سیاه بپردازید و نحوه تعادلسازی و چرخشها را به صورت بصری مشاهده کنید. [این لینک را ببینید.](#)

نمونه‌ای از کار با این ابزار جذاب را در زیر می‌بینید:

