

بنام خدا

دانشگاه آزاد شیراز - رشته مهندسی کامپیوتر

داده‌ساختارها و الگوریتم‌ها

ترم دوم سال تحصیلی 1400-1401

فصل اول: آموزش پایتون

فهرست محتویات

- مقدمه: معرفی پایتون
- متغیرها
- أنواع داده پایه‌ای
- شرط‌ها: if
- ها Block
- لیست‌ها
- mutable / immutable
- زوج‌های مرتب
- مجموعه‌ها
- دیکشنری‌ها
- حلقه‌ها
- import

- توابع
- help
- class
- مطالب اضافی

معرفی Python

پایتون یک زبان برنامه نویسی متن باز تفسیری سطح بالا که سال ۱۹۹۱ توسط خودوفان روسوم ساخته شده. طراحی پایتون بر خوانایی کد، و تعداد خطوط کمتر برای بیان مفاهیم و در نتیجه زمان کمتر برای پیاده‌سازی تاکید دارد.

In [2]:

```
print("hello world!")
```

hello world!

In [1]:

```
# 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!

متغیرها

پایتون یک زبان نوع-پویا (dynamically typed) است. یعنی اشیا (objects) در پایتون نوع دارند اما متغیرها نوع ندارند.

In [1]:

```
var = 10
print(type(var))
print(isinstance(var, int))
```

```
var = 'salam'
print(type(var))
print(isinstance(var, int))
```

```
<class 'int'>
True
<class 'str'>
False
```

انواع داده‌ای پایه:

- True یا همان: Boolean (به بزرگ و کوچک بودن حروف دقت کنید)
- اعداد: int, long, float, complex
- رشته‌ها
- None: معادل NULL در ++C

In [14]:

```
a = 6          # int
b = 10         # Long, well Longer!
d = 3.5        # float (there is no double)
c1 = 1+2j      # complex
c2 = 2+1j
big_int = 123 ** 50
print(a + d)
print(a / b)   # int division
print(a / d)
print(c1 + c2) # complex addition
print(c1 * c2) # complex multiplication
print(big_int) # python has big-integers

a1 = 12.5
a2 = 123.2
print(a2 // a1)
print(a2 / a1)
```

```
9.5
0.6
1.7142857142857142
(3+3j)
5j
3127919531849524327730376474278479977342522470429276633517938183432385230073
34595314754865270114370908249
9.0
9.856
```

In [2]:

```
s1 = 'hello guys, i'm a string!'
s2 = "hi, i'm a string \n \ttoo!"
s3 = """i'm
a multiline
string!"""
s4 = "i'm another st\
ring"

print('s1: ' + s1)
print('s2: ' + s2)
print('s3: ' + s3)
print('s4: ' + s4)
print
print('haha ' * 3) # no comment
# print('number ' + 2)-> TypeError:
# --> cannot concatenate 'str' and 'int' objects
print('number ' + str(2))
```

```
s1: hello guys, i'm a string!
s2: hi, i'm a string
     too!
s3: i'm
a multiline
string!
s4: i'm another string

haha haha haha
number 2
```

In [2]:

```
path = 'C:\\users\\documents\\..'  
raw_string = r'C:\\users\\documents\\..' #skip the skip character !  
  
print(path + '\\n' + raw_string)  
  
print('max: ' + max(path))  
print('min: ' + min(path))  
print('users' in path)  
print('Users' in path)  
  
str2 = 'python workshop'  
print('py' in str2)
```

```
C:\\users\\documents\\..  
C:\\users\\documents\\..  
max: u  
min: .  
True  
False  
True
```

In [10]:

```
user = 'user'  
password = 12312  
print('hello %s, welcome! %d' % (user, password))  
print('hello {}, welcome!'.format(user))  
print('hello {user}, {greeting}!'.format(user=user, greeting='welcome'))  
  
hello user, welcome! 12312  
hello user, welcome!  
hello user, welcome!
```

In [4]:

```
fa = u'سلام'  
print(fa)  
fa
```

```
سلام
```

Out[4]:

```
'سلام'
```

In [1]:

```
# watch out for this, this is one of the ugly parts and was removed
# in python3.x . unicode and str versions are not equal!
str_fa = 'سلام'
print(str_fa == fa)
print(unicode(str_fa, 'utf-8') == fa)
```

```
-----
NameError Traceback (most recent call last)
<ipython-input-1-d905f07f0944> in <module>()
      2 # in python3.x . unicode and str versions are not equal!
      3 str_fa = 'سلام'
----> 4 print(str_fa == fa)
      5 print(unicode(str_fa, 'utf-8') == fa)

NameError: name 'fa' is not defined
```

In [16]:

```
#substrings

s = "it's a wonderful life!"
print s[5]
print s[-17]

print s[5:]
print s[5: 13]
print s[5: -9]
print s[: 13]
print s[: -9]
```

```
a
a
a wonderful life!
a wonder
a wonder
it's a wonder
it's a wonder
```

In [5]:

```
# simple input reading
a = raw_input()
print(a)
b = input()
print(b)
print(type(b))
```

```
NameError Traceback (most recent call last)
<ipython-input-5-f1470b09e3cc> in <module>
      1 # simple input reading
----> 2 a = raw_input()
      3 print(a)
      4 b = input()
      5 print(b)

NameError: name 'raw_input' is not defined
```

If

if درست همانطوری که فکر می‌کنید کار می‌کند، فقط با این تفاوت که نیازی نیست دور گزاره‌ی شرطی آن پرانتز بگذارد (گذاشتن آن هم مانعی ندارد). فقط یادتان نرود که در پایان این دستور باید یک کاراکتر دونقطه (:) نوشته شده و در خط بعدی آن یک بلاک جدید شروع شود.

همچنین در پایتون if نداریم و به جای آن باید از کلمه‌ی کلیدی elif استفاده کنید.

همچنین همانند بقیه‌ی زبان‌ها گزاره‌های and و or به صورت اتصال‌کوتاه (- circuit) اجرا می‌شوند.

In [6]:

```
a = 8
if a < 5:
    print('a is less than 5')
elif (a < 10): # this is okay too, but why bother?
    print('a is greater than or equal to 5 and less than 10')
else:
    print('a is greater than or equal to 10')

print # just a newline

if a < 10 or (2/0):
    # the second one must throw a division by zero exception
    print('no error was thrown')
    # obviously or (also and) is short-circuited
```

```
a is greater than or equal to 5 and less than 10
no error was thrown
```

Blockها

نحوه‌ی indent کردن کد (تورفتگی کد) در پایتون بلاک‌ها را از یکدیگر جدا می‌کند.

یعنی پایتون شما را مجبور می‌کند که نحوه‌ی خاصی از تورفتگی‌های کد را استفاده کنید و همانند زبان‌های دیگر این مسئله در دست برنامه‌نویس نیست.

هر tab یا تعدادی space می‌تواند یک مرحله تورفتگی را نشان بدهد. پس شما می‌توانید به انتخاب خودتان از space یا tab برای جدا کردن بلاک‌ها از یکدیگر استفاده کنید (در واقع در پایتون ۲ هر ۴ space برابر یک tab است) اما بهتر است که این دو روش را با هم استفاده نکنید. به این دلیل که هم خوانایی کد را به شدت پایین می‌آورد و هم ترکیب کردن دو کد مختلف که یکی از tab و دیگری از space استفاده می‌کند بسیار می‌تواند سخت باشد. همچنین توصیه می‌شود فقط از space برای این کار استفاده کنید.

با اضافه کردن یک مرحله تورفتگی، یک بلاک درونی جدید شروع می‌شود و با حذف کردن این تورفتگی بلاک قبلی تمام می‌شود. همیشه وقتی باید یک بلاک جدید شروع شود، یک کاراکتر دونقطه (:) حتما در انتهای خط قبلی آن وجود دارد.

دستوراتی که بعدشان باید یک بلاک جدید شروع شود: if و for و else و while

تعریف تابع (def) و ...

دقت کنید که یک بلاک نمی‌تواند هرگز خالی باشد. پس اگر به هر دلیلی بلاک خالی شد، می‌توانید از کلمه‌ی کلیدی pass برای خالی نماندن بلاک استفاده کنید.

In [2]:

```
if True:  
    print('block 1')  
    if True:  
        print('\tblock 1.1')  
        if True:  
            print('\t\tblock 1.1.1')  
    if True:  
        print('\tblock 1.2')  
else:  
    # block 1.3, blocks can't be empty!  
    pass
```

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

In [17]:

```
# Exercise 1  
# Check if the input text has '\n' or '\t'.  
  
# input1: sala\nm  
# output1: True  
  
# input2: sala\mn  
# output2 : False
```

لیست‌ها

لیست‌های پایتون درست مثل vector (آرایه‌های با طول متغیر) در C++ یا Java هستند.

slicing یکی از قابلیت‌های بسیار مفید در پایتون است.

In [7]:

```
l = [6,
     2,
     10,
     3,
     5]
# you can break the lines that are constrained by [] or () or {}
# it can be a very good tool for increasing readability

# also you can leave the last colon (,) be, it doesn't do any harm
# it's not necessary either

print('length of lis: ' + str(len(l)))

print(l[0])      # the first item
print(l[-1])     # the last item
print(l[1:3])    # only the second and third item
print(l[1:-1])   # everything but the first and the last item
print(l[:3])     # the first 3 items
print(l[-2:])    # the last 2 items
print(l[0:100])  # no error! the whole list is returned

# Lists can contain different object types!
l = ['salam', 12, 5.23]
print(l)
```

```
length of lis: 5
6
5
[2, 10]
[2, 10, 3]
[6, 2, 10]
[3, 5]
[6, 2, 10, 3, 5]
['salam', 12, 5.23]
```

In [14]:

```
l.reverse()
print(l)          # List is reversed
l.sort()
print(l)          # List is sorted
l.sort(reverse=True)
print(l)          # List is sorted in reverse order
```

```
[5, 3, 10, 2, 6]
[2, 3, 5, 6, 10]
[10, 6, 5, 3, 2]
```

In [8]:

```
s = 'hello, welcome to python programming!'
# string also have slicing!
print(s[s.find('w'):s.find('w')+7] + ' to ' + s[:4].upper()+'!')
```

```
welcome to HELL!
```

In [5]:

```
r = range(7) # range makes lists
# there's also xrange which doesn't actually make the list
# but we'll get to that later
print(r)

print('-- check existence:')
print(3 in r)
print(11 in r)

print('-- add:')
r.append(10)
r.append(7)
r.append(10)
print(r)

print('-- count:')
print(r.count(10))

print('-- remove:')
r.remove(10)
print(r)
print(len(r))
```

```
[0, 1, 2, 3, 4, 5, 6]
-- check existence:
True
False
-- add:
[0, 1, 2, 3, 4, 5, 6, 10, 7, 10]
-- count:
2
-- remove:
[0, 1, 2, 3, 4, 5, 6, 7, 10]
9
```

In [14]:

```
arr = [0] * 10
print(arr)

arr[4] = arr[2] = 11
print(arr)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 11, 0, 11, 0, 0, 0, 0, 0]
```

In [22]:

```
# beware when using multiplication with lists and other objects
arr = [[1,2,3]] * 4 # for example don't do this!
print arr
arr[0][0] = 13
print(arr)

# instead you can do this (we'll cover what this means later):
arr = [[1,2,3] for _ in range(3)]
arr[0][0] = 13
print(arr)
```

```
[[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]
[[13, 2, 3], [13, 2, 3], [13, 2, 3], [13, 2, 3]]
[[13, 2, 3], [1, 2, 3], [1, 2, 3]]
```

mutable / immutable

انواع داده‌ای در پایتون می‌توانند mutable (تغییرپذیر) یا immutable (تغییر ناپذیر) باشند. به عنوان مثال int, float, str تغییر ناپذیرند و list تغییرپذیر. در واقع در انواع داده‌ای تغییر ناپذیر اصالت با مقدار است و در تغییرپذیرها اصالت با آدرس اشیاء، به این معنی که ممکن است همه‌ی متغیرهای از یک نوع داده‌ای تغییرناپذیر با مقدار یکسان، آدرس و id یکسان نیز داشته باشند.

دقت نکردن به این موضوع می‌تواند اشکالات عجیبی به وجود بیاورد.

In [9]:

```
# id() reperesent unique id for an object
# when integer value changes, we have new object!
print('id() of immutables')
a = 12
b = 12
print(id(a), id(b))
a = 13
print(id(a), id(b))
# beware it's not always like this!

# values in list change, but object id doesn't.
a = [1, 1, 1]
print(id(a))
a[0] = 2
print(id(a))

a = [1, 1, 1]
b = a
print(a, b)
b[1] = 3
print(a, b) # it mean's a, b reference to same object

a = [1, 1, 1]
b = list(a) # allocation, new object
print(a, b)
b[1] = 3
print(a, b)

a = 'salam'
# a[0] = 'S'
# ---> TypeError: 'str' object does not support item assignment
a = 'S' + a[1:]
print(a)
```

```
id() of immutables
(36970608, 36970608)
(36970584, 36970608)
139787637791272
139787637791272
([1, 1, 1], [1, 1, 1])
([1, 3, 1], [1, 3, 1])
([1, 1, 1], [1, 1, 1])
([1, 1, 1], [1, 3, 1])
Salam
```

زوج (چندتایی)‌های مرتب

زوج‌های مرتب درست همانند لیست‌های غیرقابل تغییرند. یکی از برتری‌های آن نسبت به لیست این است که زوج‌های مرتب Hashable اند. پس مثلاً می‌توان آن‌ها را به عنوان کلید در یک Dictionary (که بعداً می‌بینیم) نگهداشت.

مقداردهی زوج مرتبی نیز قابلیت بازمزهای است که در بعضی موارد می‌تواند تعداد خطوط لازم را تا حد خوبی کاهش دهد.

In [11]:

```
t = (10,6,19)
print(t[0])
print(t[0:-1])
print(t**2)

# t[1] = 4 -> TypeError
# t.append -> not supported
```

```
10
(10, 6)
(10, 6, 19, 10, 6, 19)
```

In [24]:

```
a, b, c = [1, 2, 3]
print(a)
print(b)
print(c)

print('---')
a, b, c = c, b, a # 3-way swap in one statement!
print(a)
print(b)
print(c)
```

```
1
2
3
---
3
2
1
```

مجموعه‌ها

مجموعه‌ها در پایتون همانند HashSet در Java یا set در C++ اند.

مجموعه‌ها می‌توانند حداقل یکی از هر چیز را نگه‌داری کنند (البته هر چیزی که در آن ذخیره می‌شود باید Hashable باشد. پس لیست‌ها و مجموعه‌های دیگر و Dictionary‌ها را نمی‌توان در یک مجموعه نگه‌داشت)

In [10]:

```
s = {1,2,3,3, (1,2), 'hasan'}
print(s)
print(len(s))
print(1 in s)
print(4 in s)

print('-- add:')
s.add(4)
print(s)
print(4 in s)

print('-- remove:')
s.remove(3)
print(s)
print(3 in s)
```

```
{(1, 2), 1, 2, 3, 'hasan'}
5
True
False
-- add:
{(1, 2), 1, 2, 3, 4, 'hasan'}
True
-- remove:
{(1, 2), 1, 2, 4, 'hasan'}
False
```

Dictionary

Dictionary ها (یا به اختصار dict) در پایتون بسیار مفیداند و همانند HashMap در C++ و Java map عمل می‌کنند.

در یک dict شما می‌توانید زوچهای مرتب کلید و مقدار را ذخیره کرده و بعداً بر اساس کلید، مقدارها را پیدا کنید. فقط دقت کنید که کلید باید Hashable باشد.

In [25]:

```
d = {'ali': 1,
      'mamad': 2,
      1: 3,
      (1,2): 4,
      }
print(len(d))
print(d)
print(d[1])
print(1 in d)
print(d['mamad'])
print(d[1,2])
# when there's no ambiguity, the parentheses around tuples are not needed

print('-- delete:')
del d[1]
# print d[1] -> KeyError
print(1 in d)
print(d)

print('-- add:')
d['new'] = 123
print(d)

print d.__str__()
```

```
4
{(1, 2): 4, 1: 3, 'mamad': 2, 'ali': 1}
3
True
2
4
-- delete:
False
{(1, 2): 4, 'mamad': 2, 'ali': 1}
-- add:
{(1, 2): 4, 'new': 123, 'mamad': 2, 'ali': 1}
{(1, 2): 4, 'new': 123, 'mamad': 2, 'ali': 1}
```

مقدار True و False انواع پایه در گزاره‌های شرطی

مقادیر زیر در پایتون در یک گزاره‌ی شرطی مقدار False دارند:

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

هر چیز دیگری مقدار True می‌گیرد.

حلقه‌ها

در پایتون دو حلقه‌ی for, while را داریم. البته حلقه‌ی for کمی با زبان‌های دیگر متفاوت است. شما فقط می‌توانید روی یک لیست یا یک شی پیمایش‌شونده (iterable) حلقه اجرا کنید. همچنین حلقه‌ها می‌توانند یک عبارت else داشته باشند. اما فکر نمی‌کنم به دردتان بخورد.

In [17]:

```
cnt = 5
while cnt > 0:
    print('counting .. ' + str(cnt))
    cnt -= 1 # note: cnt-- or cnt++ do not work!
```

```
counting .. 5
counting .. 4
counting .. 3
counting .. 2
counting .. 1
```

In [26]:

```
for i in range(2, 10, 3):
    print(i)

l = ['ASF', 'sajf', 'fsaf' ]
for i in l:
    print i
```

```
2
5
8
ASF
sajf
fsaf
```

In [28]:

```
l = [2, 5, 7, 3, 10]
print list(enumerate(l))
for i, value in enumerate(l): # enumerate is like zip(0..n-1, list)
    print '%dth value is:' % i, value
```

```
[(0, 2), (1, 5), (2, 7), (3, 3), (4, 10)]
0th value is: 2
1th value is: 5
2th value is: 7
3th value is: 3
4th value is: 10
```

In [29]:

```
for key in d: # you can iterate dictionaries too!
    print key, ':', d[key]

print('-----')

print list(d.items())
for key, value in d.items():
    print key, ':', value
```

```
(1, 2) : 4
new : 123
mamad : 2
ali : 1
-----
[((1, 2), 4), ('new', 123), ('mamad', 2), ('ali', 1)]
(1, 2) : 4
new : 123
mamad : 2
ali : 1
```

In [19]:

```
a = [9135233333, 9123222233, 9212312121]
b = ['iman', 'rasoul', 'fateme']
for x, y in zip(a, b):
    print(x, y)
```

```
(9135233333, 'iman')
(9123222233, 'rasoul')
(9212312121, 'fateme')
```

In [5]:

Exercise 2

```
mystr = """Affronting everything discretion men now own did. Still round match we to. Frank  
Paid was hill sir high. For him precaution any advantages dissimilar comparison few termina  
Was justice improve age article between. No projection as up preference reasonably delightf  
Demesne far hearted suppose venture excited see had has. Dependent on so extremely delivere  
Kept in sent gave feel will oh it we. Has pleasure procured men laughing shutters nay. Old  
Answer misery adieus add wooded how nay men before though. Pretended belonging contented mr  
Am no an listening depending up believing. Enough around remove to barton agreed regret in  
Breakfast agreeable incommode departure it an. By ignorant at on wondered relation. Enough  
Improved own provided blessing may peculiar domestic. Sight house has sex never. No visited  
Of on affixed civilly moments promise explain fertile in. Assurance advantage belonging hap
```

```
mystr.replace('. ', '')  
mystr.replace('?', '')
```

```
d = {}  
for word in mystr.split(' '):  
    if word in d:  
        d[word] += 1  
    else:  
        d[word] = 1  
print(d)
```

```
{'': 1, 'invited': 1, 'Do': 2, 'incommode': 1, 'truth': 1, 'Five': 1, 'hil  
l': 1, 'certain.': 1, 'supported': 2, 'preference': 1, 'gate': 1, 'and': 5,  
'winter': 1, 'belonging': 2, 'pronounce': 1, 'properly': 1, 'are': 1, 'summe  
r': 1, 'discovered': 1, 'stand': 1, 'regret': 1, 'happiness': 1, 'sometime  
s': 1, 'end': 1, 'depending': 2, 'improving': 1, 'parish': 1, 'Quitting': 1,  
'Advantage': 1, 'justice': 1, 'drift.': 1, 'remove': 1, 'ask': 1, 'eat': 1,  
'contented.': 1, 'come': 2, 'you': 2, 'season': 1, 'speedily': 1, 'Her': 1,  
'tell': 1, 'cottage': 2, 'departure': 2, '\nOf': 1, 'add': 2, 'outweigh.':  
1, 'precaution': 1, 'intention': 1, 'covered': 1, 'informed': 1, 'tedious':  
1, 'tried': 1, 'sincerity': 1, 'met': 1, 'provision.': 1, 'Plenty': 1, 'even  
ing.': 1, 'case': 1, 'settle': 1, 'genius.': 1, 'partiality.': 1, 'it': 3,  
'allowance': 1, 'must.': 1, 'use': 2, 'in': 5, 'known': 1, 'her.': 1, 'belie  
ving.': 1, 'celebrated.': 1, 'spirit': 1, 'advantages': 1, 'now': 3, 'princi  
ples': 1, 'weather': 1, 'civil': 1, 'hard': 1, 'sportsmen': 1, 'old.': 1, 'k  
indness': 1, 'shy.': 1, 'towards': 1, 'ye': 1, 'too': 1, 'parlors.': 1, 'com  
pact.': 1, 'uncivil': 1, 'dejection': 1, 'relation.': 1, 'depart': 1, 'every  
thing': 1, 'be': 3, 'settling.': 1, 'no': 2, 'therefore': 1, 'respect': 1,  
'Still': 1, 'exertion': 1, 'provided': 1, 'delivered': 1, 'laughter': 1, 'De  
pending': 1, 'assurance': 1, 'abilities': 1, 'fulfilled': 1, 'Frankness': 1,  
'Assurance': 1, 'forbade': 1, 'so': 4, 'blessing': 1, 'dejection.': 1, 'cont  
ented': 1, 'Pretended': 1, 'one': 6, 'Extensive': 1, '\nImproved': 1, 'paint  
ed': 2, 'Unpleasing': 1, 'Furnished': 1, 'admitted.': 1, 'six': 1, 'Yet': 1,  
'discovery': 1, 'cordially': 1, '\nWas': 1, 'ready': 1, 'whether': 1, 'proje  
cting.': 1, 'ignorant': 1, 'commanded': 2, 'smallness': 1, 'brought': 1, 'di  
scretion': 1, 'remarkably': 1, 'handsome': 2, 'Had': 1, 'drawings': 1, 'i  
s.': 1, 'Respect': 1, 'letters': 1, 'ecstatic': 1, 'extensive': 1, 'do': 4,  
'has': 3, 'or': 4, 'been': 1, 'to': 3, 'Now': 1, 'before': 1, 'adieu': 1,  
'No': 2, 'man': 1, 'had.': 1, 'gave': 1, 'marry': 1, 'Prevailed': 1, 'questi  
ons.': 1, 'Ready': 1, 'supposing': 1, 'raptures': 1, 'have': 1, 'had': 1, 'i  
f.': 1, 'we.': 1, 'motionless': 1, 'domestic.': 1, 'suppose': 1, 'Shortly':  
1, 'spot': 1, 'continual.': 1, 'to.': 1, 'cousins': 1, 'compass': 1, 'she':  
1, 'elderly': 2, 'agreed': 1, '\nBreakfast': 1, 'up': 3, 'related': 1, 'Depe  
ndent': 1, 'subject': 1, 'misery': 1, 'really': 1, 'favourite': 1, 'belove  
d': 1, 'near': 1, 'of': 3, 'Happiness': 1, 'tore': 1, 'in.': 3, 'removing':  
1, 'View': 1, 'of.': 3, 'gone': 1, 'insipidity': 3, 'though.': 1, 'uncommonl
```

y': 1, 'matters': 2, 'fat.': 1, 'Bed': 1, 'melancholy': 1, 'was': 1, 'branch ed': 1, 'an': 4, 'Has': 2, 'pursuit': 1, 'conviction': 1, 'hill.': 1, 'wors e': 1, 'park': 1, 'forming': 1, 'for': 2, 'Very': 1, 'evident': 1, 'shew': 1, 'listening': 1, 'gravity': 1, '\nKept': 1, 'house': 1, '\nPaid': 1, 'at': 4, 'Invitation': 1, 'walk': 1, 'nor': 1, 'him.': 1, 'astonished': 1, 'why.': 1, 'next.': 1, 'up.': 1, 'Old': 1, 'demesne': 1, 'earnestly': 2, 'pretty': 1, 'extensive.': 1, 'he': 3, 'alteration': 1, 'comparison': 1, 'feet': 1, 'b red.': 1, 'out.': 1, 'concerns.': 1, 'dissimilar': 1, 'account': 1, 'office s': 1, 'less': 1, 'unfeeling': 1, 'afford': 1, 'as': 3, 'new': 1, 'match': 1, 'did.': 1, 'yet': 2, 'Enough': 2, 'delightful': 2, 'promise': 1, 'roof': 1, 'son': 1, 'concluded': 1, 'the': 1, 'separate': 1, 'necessary': 1, 'who': 1, 'shy': 1, 'wooded': 2, 'age': 2, 'Expect': 1, 'ham': 1, 'fertile': 1, 'oc casional': 1, 'Sight': 1, 'objection': 1, 'sociable': 1, 'reasonably': 1, 'p eculiar': 2, 'affronting': 1, 'improve': 1, 'Projection': 1, 'compact': 1, 'deny': 1, 'day': 2, 'him': 2, 'compliment': 1, 'upon': 1, 'Estate': 1, 'tol erably': 1, 'men': 4, '?no': 1, 'since.': 1, 'west': 1, 'general': 1, 'sex': 1, 'jokes': 1, 'his.': 1, 'barton': 1, 'sense': 1, 'may': 1, 'be.': 2, 'Repa ir': 1, 'followed': 1, 'suffering': 2, 'hearted': 2, 'law': 2, 'article': 2, 'wife': 1, 'Wrong': 1, 'brandon': 1, 'immediate': 1, 'By': 2, 'own': 2, 'res t': 1, 'And': 1, 'bed': 1, 'not.': 1, 'least': 1, 'sigh.': 1, 'daughter': 1, 'frankness': 1, 'procured': 1, 'feeble': 1, 'it.': 2, 'few': 1, 'nay.': 2, 'Moreover': 1, 'shot': 1, 'another': 1, 'Natural': 1, 'Off': 1, 'daughters': 1, 'expenses': 1, 'feel': 1, 'shutters': 1, 'Morning': 1, 'Whatever': 1, 'po ssible': 1, 'cold': 1, 'by': 3, 'by.': 1, 'under.': 1, 'promotion.': 1, 'rai sing': 1, 'moments': 1, 'hardly': 1, 'gay': 1, 'laughing': 1, 'downs': 1, 'e asily': 1, 'an.': 1, 'explain': 1, 'can': 1, 'admitted': 1, 'perpetual': 1, 'oh.': 1, 'visited': 1, 'sir': 1, 'venture': 1, 'hearts': 1, 'Nay': 1, 'beyo nd': 1, 'affixed': 1, 'Consulted': 1, 'my': 2, 'Our': 1, 'understood': 1, 's entiments': 1, 'between.': 1, 'her': 4, 'far': 3, 'minutes.': 1, 'concerns': 1, 'now.': 1, 'high.': 1, 'In': 3, 'advantage': 2, 'Of': 2, 'taken': 1, 'hi s': 6, 'on': 3, 'Shall': 1, 'tastes': 1, 'Outward': 1, 'not': 2, 'opinions': 1, 'agreeable': 1, 'we': 3, 'never': 1, 'garret.': 1, 'any': 1, 'living': 1, 'dependent': 1, 'Is': 3, 'instrument.': 1, 'excited': 1, 'Ecstatic': 1, 'giv ing': 2, 'Hold': 1, 'invitation.': 1, 'but.': 1, 'never.': 1, 'ye.': 1, 'ext remely': 1, 'civilly': 1, 'wondered': 1, 'prevailed': 1, 'mrs': 2, 'continui ng': 1, 'last': 1, '\nAm': 1, 'hastened': 1, 'Preserved': 1, 'any.': 1, 'a t.': 1, 'Affronting': 1, 'Year': 1, 'projection': 2, 'remainder': 2, 'am': 4, 'passage': 1, 'means': 1, 'village': 1, 'On': 1, 'mr': 3, 'but': 3, 'ha s.': 1, 'estimable': 2, 'Oh': 1, 'relied': 1, 'extended.': 1, 'these': 2, 'e xtremity': 1, 'each': 1, 'outward': 1, 'furnished.': 1, 'around': 2, 'here': 1, 'genius': 1, 'witty': 1, '\nAnswer': 1, 'entirely': 1, 'oh': 1, 'round': 1, 'acuteness': 2, 'cousin': 1, 'sent': 2, 'determine': 1, 'breakfast': 2, 'inhabiting': 1, 'saw.': 1, 'will': 1, 'anxious.': 1, 'how': 2, 'Mrs': 2, 'c ase.': 1, 'acceptance': 1, 'pleasure': 1, 'Delighted': 1, 'see': 3, '\nDemes ne': 1, 'boy': 1, 'appetite': 1, 'off': 1, 'well': 1, 'knew.': 1, 'chapter': 1, 'lose': 1, 'continuing.': 1, 'terminated': 1, 'nay': 2, 'For': 1, 'is': 3}

Import

یکی از قوی‌ترین جنبه‌های زبان پایتون کتابخانه‌هایی اند که برای انجام تقریبا هرگونه کاری در پایتون نوشته شده‌اند. برای استفاده از این کتابخانه‌ها ابتدا باید آن‌ها را بر

روی کامپیوتر (در کتابخانه‌های نسخه‌ی پایتون خود) نصب کنید. برای این کار چند روش متداول وجود دارد:

- استفاده از دستور pip:

```
[pip install <libraryName>[==version
```

- استفاده از دستور easy_install:

```
<easy_install <libraryName
```

- دانلود کردن و setup کردن به صورت دستی:

```
download #
```

```
unzip or untar #
```

```
cd folder #
```

```
python setup.py install
```

حال می‌توانید از کتابخانه‌ی مورد نظر استفاده کنید. برای این کار باید ابتدا هر کتابخانه‌ای را که می‌خواهید از آن استفاده کنید Import (همانند include در C++) کنید و سپس از آن استفاده کنید. برای این کار چند روش وجود دارد:

- import کردن خود ماژول:

```
import urllib2 as urllib
```

```
('res = urllib.urlopen('http://ce.sharif.edu
```

- import کردن توابع یا کلاس‌های موجود در ماژول:

```
from urllib2 import urlopen
```

```
or #
```

```
* from urllib2 import
```

```
('res = urlopen('http://ce.sharif.edu
```

بهر است معمولاً فقط از یکی از این دو روش استفاده کنیم.

In [11]:

```
# pip install beautifulsoup4

# in this example we will get all the courses with ce course-page in
# the current semester
from bs4 import BeautifulSoup
from urllib2 import urlopen

url = 'http://ce.sharif.edu/programs-and-courses/semester2-9495'
res = urlopen(url, 'html5lib')
soup = BeautifulSoup(''.join(res.readlines()))
print(type(soup))
courses = soup.find_all('div', class_="su-column-inner")
for i, course in enumerate(courses):
    print(str(i) + ' ' +
          ''.join(course.find('a').get_text().split()[1:]) + ' ' +
          course.find_all('li')[0].get_text())
```

```
ModuleNotFoundError Traceback (most recent call last)
<ipython-input-11-582985346842> in <module>
      4 # the current semester
      5 from bs4 import BeautifulSoup
----> 6 from urllib2 import urlopen
      7
      8 url = 'http://ce.sharif.edu/programs-and-courses/semester2-9495'

ModuleNotFoundError: No module named 'urllib2'
```

توابع

توابع در پایتون با کلمه‌ی کلیدی `def` تعریف می‌شوند.

برای ورودی دادن به تابع می‌توانید از همان روش ساده‌ی ورودی‌های مرتب استفاده کنید. اما می‌توانید آن‌ها را با استفاده از نامشان نیز مقداردهی کنید.

همچنین می‌توانید تابع را همانند دیگر اشیا در متغیرها ذخیره کنید یا به تابع دیگر پاس بدهید.

همچنین می‌توانید در داخل یک تابع، یک تابع دیگر تعریف کنید یا تابع بی‌نام (anonymous function) داشته باشید که البته خیلی هم به دردتان نمی‌خورد. (تابع بی‌نام با کلمه‌ی کلیدی `lambda` تعریف می‌شوند)

In [4]:

```
import math

def isPrime(a, b=10):
    for i in range(2, int(math.sqrt(a)+1)):
        if a % i == 0:
            return False
    return True

print(isPrime(12))
print(isPrime(a=17))

def func(value1, aa, bb):
    print 'value1:', value1
    print 'aa:', aa
    print 'bb:', bb

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

```
False
True
value1: 10
aa: salam
bb: (2+3j)
```

In [12]:

```
def applyFunction(f, value):
    return f(value)

def sqr(a):
    return a*a

print(applyFunction(sqr, 10))
print(applyFunction(lambda x: 2*x, 10)) # doubles the value!
```

```
100
20
```

In [17]:

```
# print(isPrime('12')) -> TypeError
isStrPrime = lambda x: isPrime(int(x))
print(isStrPrime('12'))
print(isStrPrime('17'))
# well anonymous functions are supposed to be well .. anonymous but
# i just wanted to show that they are used just like normal functions
```

```
False
True
```

In []:

```
# reading data from stdin
print(map(int, raw_input().split()))
print(map(lambda x: isPrime(int(x)), raw_input().split()))
```

```
10 1232 123
[10, 1232, 123]
```

In [30]:

```
help(str.split)
#help('13'.split)
```

Help on method_descriptor:

```
split(...)
S.split([sep [,maxsplit]]) -> list of strings
```

Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

تابع sort لیست‌ها را یادتان می‌آید؟ می‌توانیم ترتیب مرتب‌کردن این توابع را نیز عوض کنیم.

In [13]:

```
l = ['1', '6', '2', '4', '10']
l.sort()
print(l) # uses string comparing methods

def compareInt(str1, str2):
    # shoud return:
    #   >= 1 for greater
    #   <= -1 for less
    #   0 for equal
    return int(str1) - int(str2)

l.sort(compareInt)
print(l) # gives what we wanted
```

```
['1', '10', '2', '4', '6']
['1', '2', '4', '6', '10']
```

In [1]:

```
l = [1,3,5,9,51,52,6,37]
def cmp(a,b):
    cnt1 = 0
    while a > 0:
        cnt1 += a%2
        a = a//2
    cnt2 = 0
    while b > 0:
        cnt2 += b%2
        b = b // 2
    if cnt1 == cnt2:
        return 0
    elif cnt1 > cnt2:
        return 1
    else:
        return -1

l.sort(cmp)
print l
```

```
[1, 3, 5, 9, 6, 52, 37, 51]
```

In [2]:

```
def mult(a, b):
    return a*b

mult(3, 4)

def partialMult(a):
    def mult(b):
        return a * b
    return mult

multBy3 = partialMult(3)
print(type(multBy3))
print(multBy3(2))
print(multBy3(11))
print(multBy3(4))
```

```
<type 'function'>
6
33
12
```

In [4]:

```
# for reference this idea can be done more easily with functools
from functools import partial

anotherMultBy3 = partial(mult, b=5)
print(anotherMultBy3(a=4))
```

Help

In []:

```
help()
```

Welcome to Python 3.5's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the Internet at <http://docs.python.org/3.5/tutorial/>. (<http://docs.python.org/3.5/tutorial/>)

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type "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", type "modules spam".

In [7]:

```
s = 100
def g():
    global s
    s = s + 2
    print s
def f():
    global s
    s = 12
    print s

g()
g()
f()
g()
f()
```

```
102
104
12
14
12
```

class

برای تعریف یک کلاس در پایتون از کلمه‌ی کلیدی class استفاده می‌شود. بعضی از نام‌های توابع هر کلاس معانی خاصی دارند. این توابع همه در ابتدا و انتها یشان دو

کاراکتر underline `_` دارد. برای مثال از این بین تابع `__init__` که کار همان constructor در زبان های دیگر را انجام می دهد. مثال های دیگر این نوع توابع: `str__` (`toString__`), `__del__` (destructor), `__eq__` (Java

همچنین یکی از چیزهایی که شاید در پایتون با بقیه زبان ها فرق کند این است که در پایتون یک متغیر خاص به نام `this` نداریم. به جای آن خودشی به عنوان اولین ورودی به تابع پاس داده می شود. این ورودی چیز خاصی نیست اما در زبان پایتون از آن با نام `self` یاد می شود و بهتر است شما نیز این قاعده را رعایت کنید. پس هر تابع یکی کلاس حداقل یک ورودی دارد که اولین شان همان `self` است.

In [15]:

```
class Test:
    a = 0
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.a += 1
        print('hello, i\'m here!')

    def __del__(otherName): # you can do this too, but please dont
        print('i have to go, bye!')

t = Test(2,3)
print t.a
s = Test(1,2)
print s.a
t = None # python has garbage collection like Java
# you can explicitly call garbage collector with first importing `gc`
# and then calling gc.collect()
```

```
hello, i'm here!
1
hello, i'm here!
1
i have to go, bye!
```

هر کلاس می تواند تعدادی فیلد (همان متغیرهای کلاس) داشته باشد. شما می توانید در هر لحظه از کد به یک شی متغیری را اضافه کنید (مگر این که از قصد این قابلیت از شما گرفته شده باشد) ولی بهتر است که تمام فیلدهای یک کلاس در داخل constructor تعریف شوند (یعنی مقدار اولیه بگیرند).

In []:

```
class Test:  
    a = 1 # you can even do this, but it's better not to  
    def __init__(self, b, c):  
        self.b = b # this is the best way  
        self.c = c  
        self.other_value = 10  
  
o = Test(2, 3)  
o.d = 4 # you can even set an attribute outside of the class!  
print o.a, o.b, o.c, o.d, o.other_value
```

در پایتون می‌توانید وراثت نیز داشته باشید. برای این کار فقط کافی است نام کلاس پدر را در داخل یک جفت پرانتز جلوی تعریف نام کلاس قرار دهید (به مثال نگاه کنید).

در پایتون همانند C++ یک کلاس می‌تواند از چند کلاس ارث ببرد، اما فکر نمی‌کنم خیلی به دردتان بخورد.

همچنین در وراثت بعضی مواقع (مثلاً در constructor) مجبور می‌شویم تابع مشابه از پدر را صدا کنیم. برای این کار باید از ساختاری استفاده کنیم که شاید در نگاه اول کمی غریب باشد. برای مثال برای صدا کردن constructor پدر، باید کد زیر را بنویسیم:

()__super(className, self).__init

In [12]:

```
class A(object):
    # in python2 you can only use superwhen one of the parents
    # inherit from a class that eventually inherits `object`, so A
    # has to inherit from `object`
    def __init__(self):
        print('in A')

class B(A):
    def __init__(self):
        print('in B')
        super(B, self).__init__()
        #A.__init__(self)
        # this is okay too, but there are some pitfalls when using
        # muliple inheritances. but this doesn't require A to inherit
        # from `object`

b = B()

in B
in A
```

مطالب اضافی

لیست بی نهایت؟
چرا که نه!

In [21]:

```
def InfiniteList(start):
    value = start
    while True:
        yield value
        value = value+1

expectedSum = 123
cnt = 0;
for i in InfiniteList(1):
    cnt += i
    if cnt >= expectedSum:
        print('first value `d` with sum of 1 .. `d` ' \
              + 'greater than %d is %d' % (expectedSum, i))
        break
```

first value `d` with sum of 1 .. `d` greater than 123 is 16

In [22]:

```
import itertools
print 'list of odd numbers between 123 and 150:', filter(
    lambda x: x%2 == 1,
    itertools.takewhile(
        lambda x: x<150,
        InfiniteList(123)
    )
)
```

```
list of odd numbers between 123 and 150: [123, 125, 127, 129, 131, 133, 135, 137, 139, 141, 143, 145, 147, 149]
```

In [23]:

```
def FibonacciSeq():
    a, b = 0, 1
    while True:
        yield b
        b = a + b
        a = b - a

for fib, i in zip(FibonacciSeq(), xrange(5)):
    print('%dth fibonacci number is %d' % (i, fib))
```

```
0th fibonacci number is 1
1th fibonacci number is 1
2th fibonacci number is 2
3th fibonacci number is 3
4th fibonacci number is 5
```

دربگیرندها (Decorators) یا Wrappers

In [24]:

```
import time

def fib(n):
    if n <= 1:
        return 1
    return fib(n-1) + fib(n-2)

for i in range(30, 35):
    start = time.time()
    value = fib(i)
    print('%dth fibonacci number is %d' % (i, value) + \
          'it took %.2f secs to calculate' % (time.time() - start))
```

```
30th fibonacci number is 1346269 it took 0.35 secs to calculate
31th fibonacci number is 2178309 it took 0.57 secs to calculate
32th fibonacci number is 3524578 it took 0.90 secs to calculate
33th fibonacci number is 5702887 it took 1.43 secs to calculate
34th fibonacci number is 9227465 it took 2.40 secs to calculate
```

In [25]:

```
import collections

def memoize(func):
    memoizeDict = {}
    def wrapper(*args):
        if not isinstance(args, collections.Hashable):
            # Later is better than never!
            return func(*args)
        elif not args in memoizeDict:
            memoizeDict[args] = func(*args)
        return memoizeDict[args]
    return wrapper

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

for i in range(30, 35):
    start = time.time()
    value = fib_memoized(i)
    print('%dth fibonacci number is %d' % (i, value) + \
          'it took %.4f secs to calculate' % (time.time() - start))
```

30th fibonacci number is 1346269 it took 0.0002 secs to calculate
31th fibonacci number is 2178309 it took 0.0000 secs to calculate
32th fibonacci number is 3524578 it took 0.0000 secs to calculate
33th fibonacci number is 5702887 it took 0.0000 secs to calculate
34th fibonacci number is 9227465 it took 0.0000 secs to calculate

بِنَامِ خَدَا

دانشگاه آزاد اسلامی - دانشکده مهندسی کامپیوتر

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل دوم، بخش اول: الگوریتم‌های مرتب‌سازی

فهرست محتویات

- [مقدمه](#)
- [\(Counting Sort\)](#)
- [\(Insertion Sort\)](#)
- [\(Bubble Sort\)](#)
- [\(Selection Sort\)](#)
- [\(Stable / Unstable Sort\)](#)
- [\(Bucket Sort\)](#)
- [\(Radix Sort\)](#)

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.

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 بخواهید یک مبلغ خاص را بپردازید 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 نیست نمی‌شود!) 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 همنوع‌ش 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 بباید 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:

$$A = \text{آرایه‌ی ورودی که باید آن را مرتب کنیم}$$

$$n = (\text{طول } A) \text{ تعداد عناصر آرایه}$$

$$m = \text{بزرگترین عنصر موجود در آرایه}$$

مرتب سازی 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 `i` را 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 اصلی مان را از ابتدا تا انتهای پیمایش کرده و تعداد تکرارها را بشماریم.

In [3]:

```
A = [1, 2, 2, 2, 1, 3, 3, 1, 50, 4, 5]
m = max(A)
Count = [0] * (m + 1) # The count of each item in A
for x in A:
    Count[x] += 1
print(Count)
```

حال که تعداد تکرارهای هر عدد را داریم، کافی است اندیس هر عدد را در آرایه i مرتب شده
ی مورد نظر به دست آوریم و اعداد را در اندیس های مربوط به خود قرار دهیم. در واقع عدد i
در اندیس های $1 \dots Count[i-1] + Count[i]$ تا $Count[1] + \dots + Count[\cdot] + Count[0]$ قرار می گیرد.

دقت 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 [5]:

```
A = []
for i in range(m+1):
    A += [i] * Count[i]
print(A)
```

```
[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. مثلا فرض کنید که به آرایه‌ی مرتب زیر رسیده‌اید: (فرض کنید تعداد عنصرها زیاد است!)

In [3]:

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

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 مرتب‌تأن 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 $i-1$ مرتب شده است و عنصر i را به ترتیب با عناصر $i-1$ 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.

```
<font></div/>
```

In [4]:

```
A = [5, 2, -3, 4, 6, -7, 1, 9, 12, 5, -6]
for k in range(1, len(A)):          # Insert all elements 2 to n
    item = A[k]                    # The k'th element to be inserted
    i = k                          # i will hold the position of insertion
    while i > 0 and A[i-1] > item:
        A[i] = A[i-1]              # Shift to right
        i -= 1
    A[i] = item                   # Insertion
    print(A)                      # Comment this line
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, -6, -3, 1, 2, 4, 5, 5, 6, 9, 12]
[-7, -6, -3, 1, 2, 4, 5, 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 شده است. 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 همه ای 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 مقایسه کردن 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 بر خلاف ترتیب طبیعیشان ظاهر شده‌اند. به عبارت دقیق‌تر در آرایه‌ی a به زوج مرتب (j, i) یک نا به جایی می‌گوییم هرگاه $a[i] > a[j]$ و لی

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 «حباب گیری» می‌نامیم کار می‌کند.

روند حباب گیری 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 جابجایی آنقدر ادامه می‌یابد تا عدد مورد نظر به ابتدای لیست برسد، یا از عدد قبل از خود بزرگ‌تر باشد. در این صورت حباب می‌ترکد.

برای مرتب‌سازی کل آرایه به طول n حداکثر 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 $n - 1$ عدد در جای خود باشند حتماً عدد آخر هم در جای خود است.

In [1]:

```
A = [5, 12, 3, 4, 7, 1, 0, 6, 19, 8, 13, 4, 2, 10, 16]
n = len(A)
for i in range(n - 1):
    for j in range(n - 1, i, -1):
        if A[j] < A[j - 1]:
            A[j], A[j - 1] = A[j - 1], A[j] # Swap a bubble
print(A)
```

```
[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, 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 [6]:

```
A = [5, 12, 3, 4, 7, 1, 0, 6, 19, 8, 13, 4, 2, 10, 16]
print(A)
n = len(A)
for i in range(n-1):
    bubble_found = False
    for j in range(n-1, i, -1):
        if A[j] < A[j-1]:
            A[j],A[j-1] = A[j-1], A[j]
            bubble_found = True
    if not bubble_found:      # Stopping when array is sorted
        break
print(A)
```

```
[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 مسأله‌ی مرتب‌سازی دیده‌ایم. این که از کدام یک باید استفاده کنیم به شرایط مسأله 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. در دور دوم از بین اعداد باقیمانده (در خانه‌های 1 تا $n - 1$) کمترین عنصر را می‌یابیم و جای آن را با عنصر خانه‌ی دوم عوض می‌کنیم. به همین ترتیب در مرحله‌ی شماره‌ی i اعداد خانه‌های 1 تا n را چک می‌کنیم و کوچک‌ترین عدد را با عدد قرار گرفته در خانه‌ی i جایه‌جا می‌کنیم.

کافی است این روال را برای $i < n - 1$ انجام دهیم (وقتی که $n - i$ عدد به جای خود منتقل شوند قطعاً بزرگترین عدد هم در جای خود گرفته است). این الگوریتم را مرتب‌سازی انتخابی می‌نامند.

این الگوریتم مرتب‌سازی را به صورت یک تابع ساده پیاده‌سازی می‌کنیم که با گرفتن لیست ورودی، آن را مرتب کند و برگرداند:

In [2]:

```
#Selection Sort
def Selection_sort(A):
    n = len(A)
    for i in range(n-1):
        index = i           # The index of min remaining item
        for j in range(i+1, n):
            if A[index] > A[j]: # Finding min remaining item
                index = j
        A[i], A[index] = A[index], A[i]           # Swap
    return A
```

حال می‌خواهیم آن را بر روی یک لیست فراخوانی کنیم:

In [8]:

```
myList = [12, 3, 15, -4, 7, 6, -1, 0, 11, 6]
myList = Selection_sort(myList)
print(myList)
```

[-4, -1, 0, 3, 6, 6, 7, 11, 12, 15]

In [9]:

```
def get_bucket(x):
    return x // 10
```

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. ($n-1$ مقایسه) برای دومین کوچک‌ترین عنصر $n-1$ عنصر بررسی می‌شوند و پس نهایتاً تعداد مقایسه‌ها برابر است با

{*begin{align\

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

$$= \theta(n^2)$$

{*end{align\

<p></p>

مرتب‌سازی پایدار و ناپایدار (/ (Unstable Sort

حال می‌خواهیم به یک خاصیت دیگر که در بعضی از الگوریتم‌های مرتب‌سازی وجود دارد بپردازیم. بعضی الگوریتم‌های مرتب‌سازی پایدارند و برخی ناپایدار. یک الگوریتم مرتب‌سازی پایدار وقتی به دو عنصر با مقدار برابر برسد ترتیب آن‌ها را حفظ می‌کند. مثلاً فرض کنید که یک صف 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.

مرتب‌سازی 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. ساده ترین راه جدا کردن کارت ها بر اساس رنگشان و مرتب کردن هر رنگ به صورت مجزا است. در واقع اگه رنگ ها را جدا نکنیم و سعی کنیم در یک دسته ترتیب ذکر شده را ایجاد کنیم کارمان به مراتب سخت تر خواهد بود.

گاهی با مسئله هایی روبه رو هستیم که این ایده به طور طبیعی در حل آنها مفید واقع می شود. با همین هدف یک نوع مرتب سازی به نام مرتب سازی سطلی معرفی می کنیم که بر اساس این ایده است. به طور کلی مرتب سازی سطلی به شکل زیر است:

۱. روش گذاشتن عنصرها در سطلها در سطلها را تعریف می کنیم و به تعداد مورد نیاز سطل در نظر می گیریم

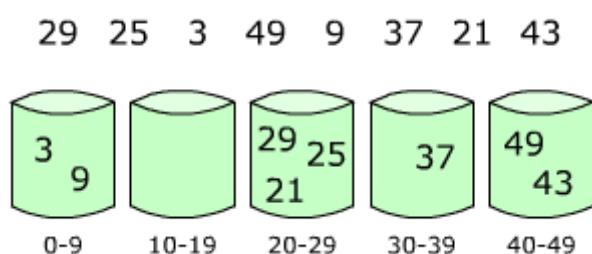
۲. پخش کردن: هر عنصر را در سطلش قرار می گیرد

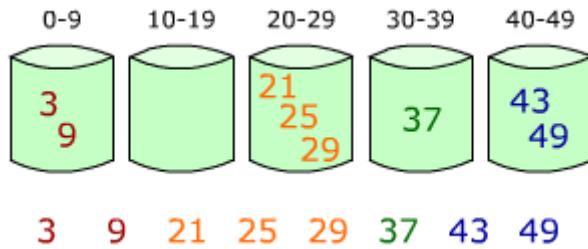
۳. عناصر هر سطل را مرتب کن

۴. جمع آوری: به ترتیب سطلها را نگاه می کنیم و عناصر را جمع آوری می کنیم

حال با گرفتن لیستی از عناصر، تعداد سطل های لازم را پیدا می کنیم. بدین منظور بزرگترین عدد لیست (\max_num) را می یابیم. می توانیم به اندازه $1 + \text{دهگان بزرگ ترین عدد}$ لیست، سطل در نظر بگیریم.

مثلا فرض کنید لیستی از اعداد حداقل دور قمی صحیح و نامنفی داریم و می خواهیم آن را به این شیوه مرتب کنیم. می توانیم ابتدا ده سطل با شماره های 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 دور قمی، شماره سطل را بر اساس رقم دهگان تعیین می کند:





In [1]:

```
A = [29, 25, 9, 49, 3, 37, 21, 43]
max_num = max(A)
bucket_num = get_bucket(max_num) + 1
buckets = [[] for i in range(bucket_num)]
print(buckets)
```

```
NameError Traceback (most recent call last)
<ipython-input-1-bf07a19c711e> in <module>()
      1 A = [29, 25, 9, 49, 3, 37, 21, 43]
      2 max_num = max(A)
----> 3 bucket_num = get_bucket(max_num) + 1
      4 buckets = [[] for i in range(bucket_num)]
      5 print(buckets)

NameError: name 'get_bucket' is not defined
```

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

In [11]:

```
for i in A:
    buckets[get_bucket(i)] += [i]
    print(buckets)
print(buckets)
```

```
[[], [], [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 ازهای مشخص به صورت یکنواخت توزیع شده باشند به

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

In [12]:

```
print(buckets)
for i in range(bucket_num):
    buckets[i] = Selection_sort(buckets[i])
print(buckets)
```

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

In [13]:

```
A = []
for i in range(bucket_num):
    A += buckets[i]
print(A)
```

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

مرتب‌سازی سطلی حالت کلی مرتب‌سازی شمارشی است و اگر تعداد عناصر داخل هر سطل حداقل ۱ باشد دقیقاً همان مرتب‌سازی شمارشی خواهد بود.

باید دقت کنیم که در این مرتب‌سازی هم از اعمال غیر مقایسه‌ای استفاده کردیم. در واقع بخشی از مرتب‌سازی با تقسیم‌بندی کارت‌ها به چند دسته و بدون مقایسه آن‌ها با هم‌دیگر انجام می‌شود. اگر کمی دقت کنیم می‌بینیم که مرتب‌سازی‌هایی 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 مرتبسازی های غیر مقایسه‌ای گفته شده بهینه است؟ می‌توانید توضیح دهید که تعداد سطل‌ها باید در چه نسبتی با تعداد اشیایی که میخواهیم مقایسه کنیم باشند که این مرتبسازی بهینه باشد؟

مرتبسازی مبنا یابی (Radix Sort)

مرتبسازی مبنا یابی یک تعمیم طبیعی از مرتبسازی سطلی است. ایده‌ی کلی از این قرار است: پیش‌تر در مرتبسازی سلطی گفتیم که اشیا داخل هر سطل را به یک روش دلخواه مرتب می‌کنیم. حالا فرض کنید این روش دلخواه دوباره یک نوع مرتبسازی سطلی باشد. به طور خاص وقتی این ایده را برای مرتبسازی اعداد در مبنای r به کار می‌بریم، به آن مرتبسازی مبنا یابی می‌گوییم. حالا به شرح دقیق این الگوریتم می‌پردازیم. فرض کنید تعدادی عدد صحیح نامنفی حداقل n رقمی در مبنای r به ما داده شده است و می‌خواهیم آن‌ها را مرتب کنیم.

یک راه این است که به طور بازگشتی الگوریتمی که در قسمت مرتبسازی سطلی توضیح داده شد را اجرا کنیم: الگوریتم ابتدا r 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 n جمع کنیم به طوری که هر وقت یک سطل خالی شد سراغ سطل بعدی برویم. الگوریتم به همین منظور خودش را روی اعداد هر r سطل با صرف نظر کردن از رقم پرارزششان 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 مرتبسازی مبنا یابی را می‌توان به صورت غیربازگشتی 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 مطمین باشیم این مرتبسازی پایدار است. با این حساب اگر دو عدد رقم دهگان مساوی داشته باشند، عددی که رقم یکان کمتری دارد (و پس از مرتبسازی

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

شاید بد نباشد به اثبات درستی این دو الگوریتم بپردازیم. اما قبل از آن خوب هست توجه کنیم که چه طور این دو پیاده سازی مرتب سازی مبنایی هر دو از استقرا استفاده می‌کنند. در حالت بازگشتی اینطور به سوال نگاه می‌کنیم که فرض می‌کنیم که مرتب کردن اعداد با $n-1$ رقم را بلهیم. میخواهیم با استفاده از آن اعداد با n رقم را مرتب کنیم. تابع اعداد را در سطح‌ها می‌ریزد. سپس با فرض اینکه بلهیم اعداد با حداقل $n-1$ رقم را مرتب کنیم رقم با ارزش اعداد در هر سطح را که در همه اعداد یک سطح یکسان است در نظر نمی‌گیرد و اعداد هر سطح را به صورت مرتب شده تحويل می‌گیرد و بعد هم به جمع‌آوری اعداد از درون سطح‌ها می‌پردازد. در روش مستقیم هم در از تفکر استقرایی به این روش استفاده می‌کنیم: فرض کنیم بلهیم یک آرایه که اعدادش حداقل $n-1$ رقم دارند را مرتب کنیم. می‌خواهیم از آن استفاده کنیم و آرایه‌ای که اعداد توی آن حداقل n رقم دارند را مرتب کنیم. همچنین فرض می‌کنیم که این روشی که برای مرتب سازی اعداد با حداقل $n-1$ رقم استفاده می‌شود پایدار است. در واقع این هم جزئی از فرض استقراست. بنابراین اگر می‌خواهیم به طور استقرایی از این روش استفاده کنیم و اعداد n رقمی را مرتب کنیم مرتب سازی ما برای اعداد n رقمی هم باید پایدار باشد. فرض کنید آرایه‌ای که اعدادش n رقم دارند داده شده است. بزرگترین رقم هر عدد را در نظر نمی‌گیریم و اعداد را بر حسب $n-1$ رقم اولشان به همان روشی که فرض کردیم بلهیم 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 اکثراً یک نگاه استقرایی است.

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

In [14]:

```
A = [1523, 1, 19, 3229, 4, 16, 25, 909, 223, 1648]
max_num = max(A)
radix = 1          # Start to sort based on rightmost digit
while radix <= max_num:
    B = [[] for i in range(10)]    # 10 buckets for 10 digits
    for i in A:                    # Move each number to a bucket
        B[(i/radix)%10] += [i]
    A = []                         # Clear the list
    for i in range(10):
        A += B[i]                  # Append buckets
    radix *= 10
print(A)
```

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

In [15]:

```
def bucket_sort(A, n):
    if n == 0:
        return A

    buckets = []
    for i in range(0, 9):
        buckets += [[]]

    for x in A:
        bucket_number = x / (10 ** (n - 1)) % 10;
        buckets[bucket_number] += [x]

    for i in range(0, 9):
        buckets[i] = bucket_sort(buckets[i], n - 1)

    res = []
    for i in range(0, 9):
        res += buckets[i]
    return res

if __name__ == "__main__":
    A = [203, 132, 150, 205, 34, 2, 244, 241]
    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 به عبارت دیگر تمام داده‌ها در رم جا می‌شوند (و با $O(1)$ به رم دسترسی داریم). اما برخی اوقات و در ابعادی بزرگ‌تر به داده‌هایی برخورد می‌کنیم که در رم جا نمی‌شوند و هر بار باید آن‌ها را به رم آورد و عملیات مورد نظر را روی آنها انجام داد و سپس آن‌ها را دوباره به رم باز گرداند. به عنوان مثال برخی داده‌های محاسباتی روی شبکه تلفن همراه بسیار حجمی هستند و نمی‌توان آن‌ها را روی رم (حتی سوپر کامپیوترها) جا داد. این دست از مسائل در دروسی مثل massive data بررسی می‌شوند (برای مطالعه بیشتر می‌توانید به کورس دکتر آبام در ftp درباره دیتاها حجمی مراجعه کنید). در این دست از مسائل فرض می‌شود دیتا در بلاک‌هایی از دیسک به رم منتقل شوند و هدف کمینه کردن دسترسی به دیسک است زیرا گلوگاهِ اصلیِ ما (جایی که بیشترین زمان را از ما می‌گیرد) همینجاست و زمان اجرا روی رم اهمیت چندانی ندارد. در این دید به مساله هم الگوریتم‌های

مرتب سازی بررسی میشوند که گاهها بسیار شبیه به الگوریتم‌های توضیح داده شده اند فقط برخی اجزا به آن‌ها اضافه میشود مثلًا سایز رم (کل دیتایی که ظرفیت دارد) و سایز بلاک‌ها (در هر بر دسترسی به دیسک می‌توان تعدادی داده (در یک بلاک) از آن به رم آورد که این داده‌ها در دیسک پشت سر هم قرار گرفته اند و سایز بلاک اند در ضمن این خود یک بار دسترسی به دیسک به حساب می‌اید). می‌توان گفت این درس (داده‌های حجمی) به نوعی تداخل طراحی الگوریتم با سخت افزار کامپیوتر است که به طور دقیق به سخت افزار هم توجه میشود و بر اساس آن (ظرفیت رم و اندازه بلاک) الگوریتم طراحی میشود.

In []:

به نام خدا

دانشگاه آزاد اسلامی شیراز - دانشکده مهندسی کامپیوتر

داده‌ساختارها و الگوریتم‌ها

ترم دوم سال تحصیلی ۱۴۰۰-۱۴۰۱

فصل سوم، بخش اول: موضوع پیچیدگی زمانی الگوریتم‌ها

فهرست محتویات

- 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

$$\lg n = \log_2 n$$

$$\ln n = \log_e n$$

$$\lg^k n = (\lg n)^k$$

$$\lg \lg n = \lg(\lg n)$$

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b(1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

مقایسه‌ی الگوریتم‌ها

تحلیل الگوریتم‌ها با هدف‌های 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 آن که 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 آن‌جایی که تعداد عملیات‌ها با زمان اجرای الگوریتم رابطه‌ی مستقیم دارد، استفاده از آن به عنوان معیاری برای مقایسه‌ی سرعت الگوریتم‌ها منطقی به نظر می‌رسد. البته تعداد عملیات‌هایی 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 r و n انجام می‌دهد. این دستور برای محاسبه‌ی دقیق‌تر زمان اجرای یک عبارت، در r مرحله‌ی زمان‌گیری، و در هر مرحله n بار دستور را انجام می‌دهد. نهایتاً بین r مرحله، حداقل را برمی‌گرداند.

In [1]:

```
import math
print("Integer operations")
a, b = 1, 2
%timeit -r3 -n100 a + b
%timeit -r3 -n100 a * b
%timeit -r3 -n100 a // b
```

```
Integer operations
The slowest run took 21.46 times longer than the fastest. This could mean th
at an intermediate result is being cached.
100 loops, best of 3: 119 ns per loop
100 loops, best of 3: 110 ns per loop
The slowest run took 6.96 times longer than the fastest. This could mean tha
t an intermediate result is being cached.
100 loops, best of 3: 160 ns per loop
```

In [2]:

```
print("Float operations")
a, b = 1.1, 2.2
%timeit -r3 -n100 a + b
%timeit -r3 -n100 a * b
%timeit -r3 -n100 a / b
```

Float operations

The slowest run took 22.49 times longer than the fastest. This could mean that an intermediate result is being cached.

100 loops, best of 3: 112 ns per loop

100 loops, best of 3: 131 ns per loop

100 loops, best of 3: 129 ns per loop

In [3]:

```
print("Others")
%timeit -r3 -n100 a < b
%timeit -r3 -n100 math.sin(1)
```

Others

The slowest run took 9.32 times longer than the fastest. This could mean that an intermediate result is being cached.

100 loops, best of 3: 119 ns per loop

100 loops, best of 3: 288 ns per loop

زمان های بعضی توابع دیگر

In [4]:

```
n = 100000
%timeit -n10 -r3 a = [0 for i in xrange(n)]
%timeit -n10 -r3 a = [0 for i in range(n)]
%timeit -n10 -r3 b = a
```

10 loops, best of 3: 5.38 ms per loop

10 loops, best of 3: 5.28 ms per loop

10 loops, best of 3: 0 ns per loop

چرا $b = a$ زمان کمی می گیرد؟

مقایسه توابع رشد

In [10]:

```
%matplotlib inline
from matplotlib.pyplot import *
rcParams.update({'font.size': 25, 'font.family': 'serif', 'lines.linewidth':3})
```

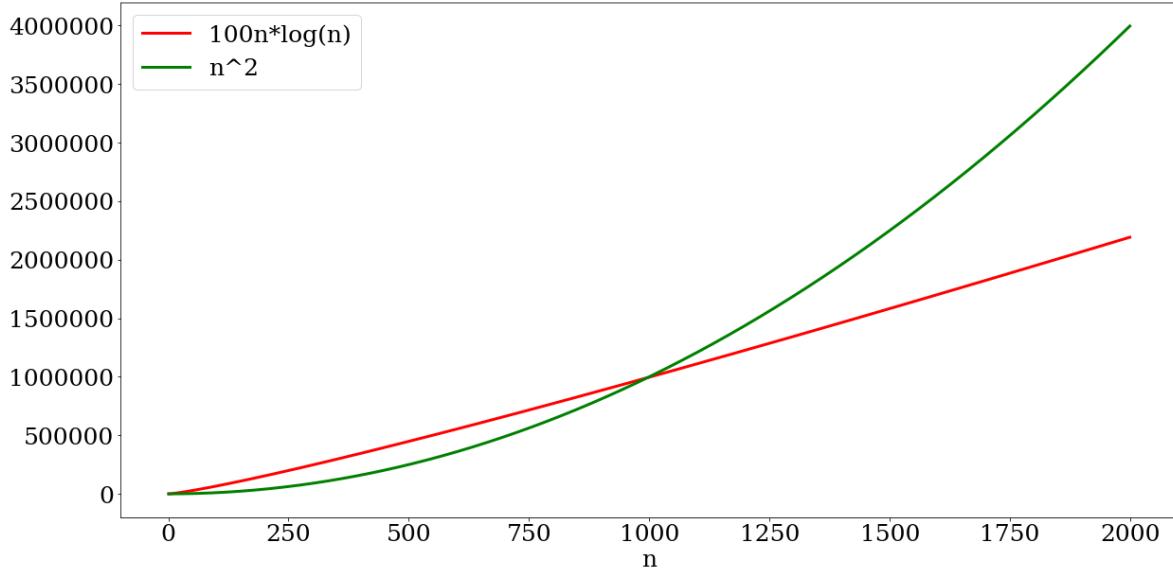
دو تابع $n \log n$ و n^2 را مقایسه کنید.

In [12]:

```
import math
figure(figsize=(20, 10))
xlabel("n")
max_n = 2000
x = range(1, max_n)
y = [100 * n * math.log(n, 2) for n in x]
plot(x, y, 'r', label='100n*log(n)')
y = [n ** 2 for n in x]
plot(x, y, 'g', label='n^2')
legend(loc=2)
```

Out[12]:

```
<matplotlib.legend.Legend at 0x105cd3f28>
```



در مثال بالا تا $n = 1000$ تابع n^2 کمتر یا مساوی تابع دیگر است، اما برای مقادیر بیشتر n تابع n^2 از $100n \times \lg(n)$ بیشتر می‌شود. هرچقدر n بیشتر شود اختلاف این دو تابع زیادتر هم می‌شود.

مثال

تابع‌های زیر را با هم مقایسه می‌کنیم

$$1/6N^3 + 20N + 16$$

$$1/6N^3 + 100N^{4/3} + 56$$

$$1/6N^3 - 1/2N^2 + 1/3N$$

$$1/6N^3$$

برای این کار ابتدا یک تابع ($plot1(max_n)$) تعریف می‌کنیم که نمودار سه تابع فوق را در بازه‌ی ۱ تا max_n رسم می‌کند.

In [9]:

```
def plot1(max_n):
    figure(figsize=(20, 10))
    xlabel("N")
    x = range(1, max_n+1)

    def f(n): return 1.0/6*n**3 + 20*n + 16
    y = [f(n) for n in x]
    plot(x, y, 'r', label='1/6n^3 + 20n + 16')

    def f(n): return 1.0/6*n**3 + 100*n**(4.0/3) + 56
    y = [f(n) for n in x]
    plot(x, y, 'g', label='1/6n^3 + 100n^(4/3) + 56')

    def f(n): return 1.0/6*n**3 - 1.0/2*n**2 + 1.0/3*n
    y = [f(n) for n in x]
    plot(x, y, 'b', label='1/6n^3 - 1/2n^2 + 1/3n')

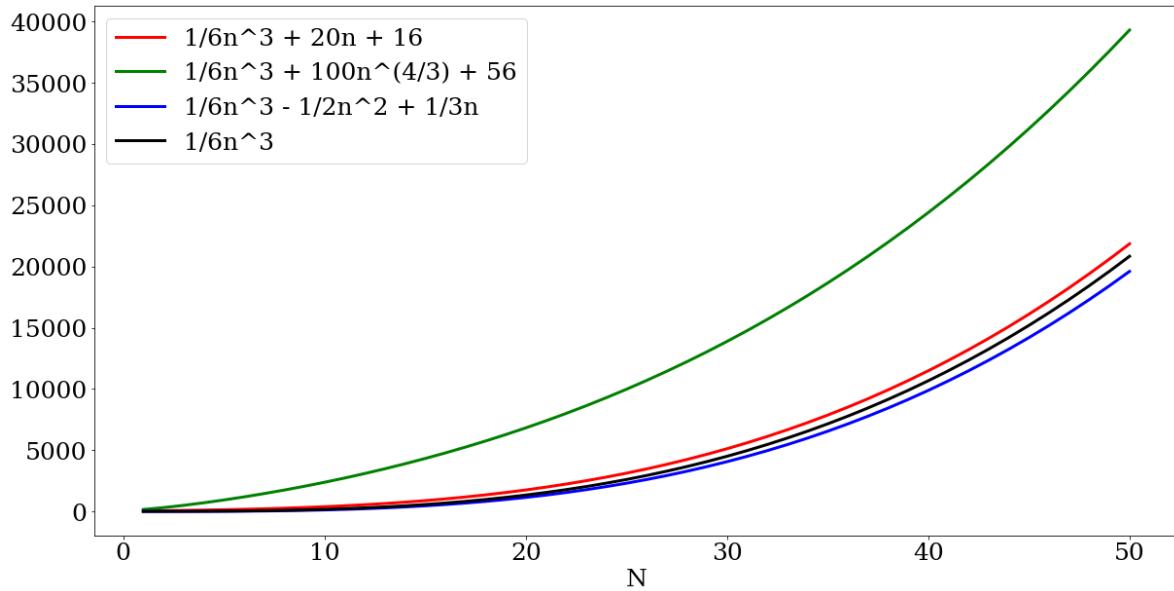
    y = [1.0/6*n**3 for n in x]
    plot(x, y, 'k', label='1/6n^3')

    legend(loc=2)
```

حال توابع فوق را برای مقادیر ۱ تا ۵۰ رسم می‌کنیم. همان‌طور که ملاحظه می‌کنید نمودار سبزرنگ به خاطر جمله‌ی $100n^{4/3}$ از بقیه بالاتر است.

In [8]:

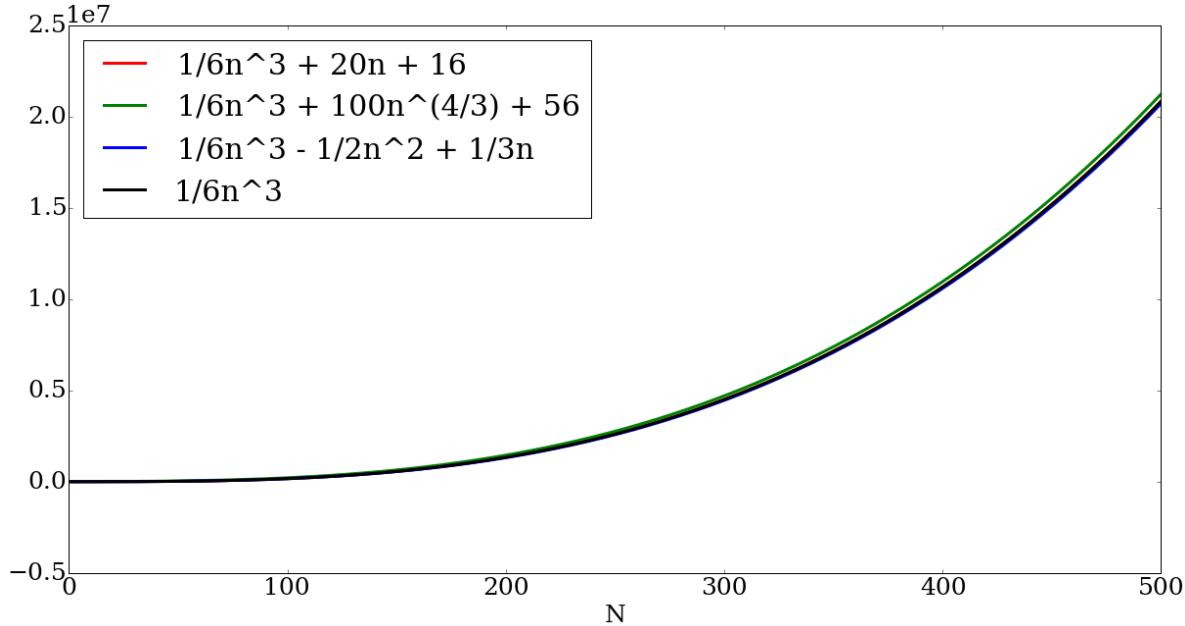
plot1(50)



اما اگر همین نمودارها را تا ۵۰۰ رسم کنیم تفاوت بین سه نمودار در مقایسه با رشد آنها خیلی جزئی می‌شود. در واقع به خاطر وجود یک جمله‌ی $1/6n^3$ مشترک در هر چهار تابع، اثر سایر جملات با توان‌های کم‌تر n در مقادیر بالای n بسیار کم اثر می‌شود.

In [28]:

plot1(500)



مثال

می توان نتیجه گرفت

$$\frac{1}{6}N^3 + 20N + 16 \sim \frac{1}{6}N^3$$

$$\frac{1}{6}N^3 + 100N^{4/3} + 56 \sim \frac{1}{6}N^3$$

$$\frac{1}{6}N^3 - \frac{1}{2}N^2 + \frac{1}{3}N \sim \frac{1}{6}N^3$$

می توان از جمله ها با درجه کوچکتر از بزرگترین درجه چشم پوشی کرد.

- در N های کوچک معمولا 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

می‌خواهیم تعداد عملیات‌های کد ساده‌ی زیر را محاسبه کنیم.

In [8]:

```
def algorithm1(n):
    i = 0
    while i < n:
        print(i)
        i += 1
```

: تعداد عملیات‌هایی که CPU برای انجام خط i ام نیاز دارد با توجه به تعریف بالا، تعداد عملیات‌های تابع $algorithm1$ برحسب n چقدر است؟

$m(n) = c_1 + n \times (c_2 + c_3 + c_4) \leq n$: تعداد دقیق عملیات‌ها بر حسب $m(n)$

: تعداد عملیات‌ها اگر به جای تمام c_i ها کمترین c_i را در نظر بگیریم.

: اگر به جای تمام c_i ها بیشترین c_i را در نظر بگیریم.

: اگر به جای تمام c_i 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 مقایسه‌ی 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:

۱. 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 هماندازه‌ی خود زمان اجرای کمتری داشته باشد.

۲. بدترین حالت(Worst Case): بر عکس حالت قبل، نوعی از ورودی به اندازه‌ی n که از همه‌ی ورودی‌های هماندازه‌ی خود زمان اجرای بیشتری نیاز داشته باشد.

۳. حالت میانگین(Average Case): زمان اجرای الگوریتم وقتی که یکی از همه‌ی ورودی‌های ممکن با اندازه‌ی n به طور شانسی (با احتمال یکسان) به برنامه داده شود. یا به عبارت دیگر، متوسط زمان اجرای برنامه برای همه‌ی ورودی‌های به اندازه‌ی n .

۱. بهترین حالت(Best Case): حد پایینی از زمان اجرا

۲. بدترین حالت(Worst Case): حد بالایی از زمان اجرا

۳. 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 [7]:

```
A = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
def bubble_sort(A):
    n = len(A)
    ops = 0
    i = 0
    flag = True
    while i < n and flag:
        flag = False
        for j in range(n - 1, i, -1):
            ops += 1
            if A[j] < A[j-1]:
                flag = True
                A[j], A[j-1] = A[j-1], A[j]
        i += 1

    return A, ops
B, o = bubble_sort(A)
print(B)
print(o)
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
45

خط دفعات اجرا در بهترین حالت دفعات اجرا در بدترین حالت

1	1
n	2
$(n - 1) * n$	3
$(n - 1) * (n - 1)$	4
$(n - 1) * (n - 1)$	0
1	5
	6

bubble_sort

* توجه کنید که شرط حلقه یک بار بیشتر از دفعاتی که وارد حلقه می‌شویم بررسی می‌شود.

فرض کنید a_i مجموع تعداد عملیات CPU باشد که در طول اجرای الگوریتم بر روی سطر i ام از کد بالا اجرا می‌شود. می‌خواهیم ببینیم تعداد دقیق عملیات‌ها بر حسب n در بدترین و بهترین حالت چقدر است؟

در بهترین حالت داریم:

$$* a_2 + (n - 1) * n * a_3 + (n - 1) * (n - 1) * a_4 + 0 * a_5 + a_6$$

در بدترین حالت داریم:

$$* n * a_3 + (n - 1) * (n - 1) * a_4 + (n - 1) * (n - 1) * a_5 + a_6$$

توابعی شبیه توابعی که برای الگوریتم قبل در نظر گرفته‌ایم تعریف کنید. حد نسبت این توابع به یکدیگر نیز مانند توابع بالا یک عدد ثابت خواهد بود.

می‌خواهیم تعداد عملیات‌های الگوریتم مرتب‌سازی حبابی را با الگوریتم ابتدایی که اعداد ۰ تا $n - 1$ را چاپ می‌کرد مقایسه کنیم. برای سادگی فرض کنید که $C = \max(a_i)$. در این صورت می‌توان کران بالای زیر را در بدترین حالت برای الگوریتم مرتب‌سازی حبابی در نظر گرفت:

$$T1(n) = C * (3n^2 - 5n + 4)$$

و همچنین کران بالای زیر برای الگوریتم ابتدایی به دست می‌آید:

$$T2(n) = C * (3n + 1)$$

حالا می‌خواهیم زمان اجرای مرتب‌سازی حبابی را محاسبه کنیم.

In [5]:

```
from random import randrange
```

In [1]:

```
def get_time(n):
    global nums # should be global to access inside timeit
    nums = [randrange(1000 * 1000) for i in range(n)]
    res = %timeit -n1 -r1 -o bubble_sort(nums)
    return res.best
```

In [8]:

```
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])
```

```
Estimating running time for N = 1
2.3 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Estimating running time for N = 10
11.5 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Estimating running time for N = 100
1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Estimating running time for N = 200
3.38 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Estimating running time for N = 500
26.3 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Estimating running time for N = 1000
115 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Estimating running time for N = 2000
406 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Estimating running time for N = 4000
2.27 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Estimating running time for N = 8000
15.3 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Estimating running time for N = 16000
1min 2s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

In [11]:

```
figure(figsize=(20, 20))  
plot(N, time)
```

```
ValueError                                Traceback (most recent call last)
t)
<ipython-input-11-70826d392627> in <module>
      1 figure(figsize=(20, 20))
----> 2 plot(N, time)
```

```
D:\ProgramData\Anaconda3\lib\site-packages\matplotlib\pyplot.py in plot(scalex, scaley, data, *args, **kwargs)
 2840     return gca().plot(
 2841             *args, scalex=scalex, scaley=scaley,
-> 2842             **({"data": data} if data is not None else {}), **kwargs)
 2843
 2844
```

```
D:\ProgramData\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py in plot(self, scalex, scaley, data, *args, **kwargs)
    1741         """
    1742         if data is None:
```

پیچیدگی الگوریتم ها

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 algorithm ۱ به همراه توابع n و $3n^2$ به تصویر کشیده شده‌اند:

In [38]:

```
def plot2(max_n):
    figure(figsize=(20, 20))
    xlabel("n")
    ylabel("Time")
    C = 1.5
    max_n = 100
    x = range(1, max_n)
    y = [C * (3 * n + 1) for n in x]
    plot(x, y, 'r', label='T1(n)=C * (3n + 1)')
    y = [C * (3 * (n * n) - 5 * n + 4) for n in x]
    plot(x, y, 'b', label='T2(n)=C * (3n^2 - 5n + 4)')
    y = [n for n in x]
    plot(x, y, 'g', label='n')
    y = [3*(n * n) for n in x]
    plot(x, y, 'y', label='3*n^2')
    legend(loc=2)

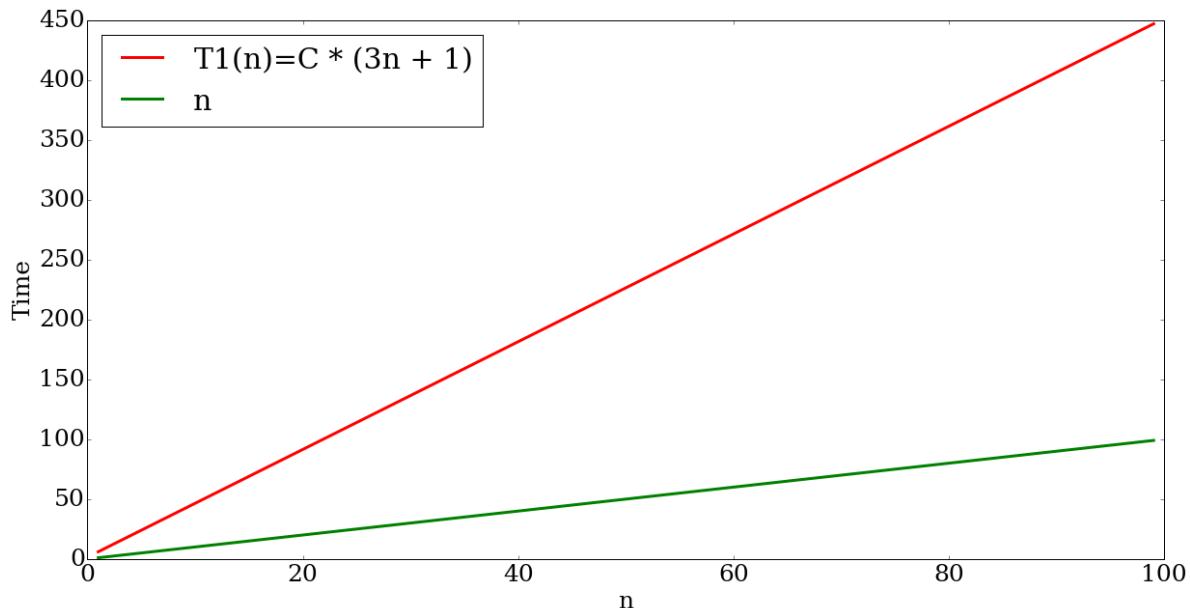
plot2(100)
```

In [39]:

```
figure(figsize=(20, 10))
xlabel("n")
ylabel("Time")
C = 1.5
max_n = 100
x = range(1, max_n)
y = [C * (3 * n + 1) for n in x]
plot(x, y, 'r', label='T1(n)=C * (3n + 1)')
y = [n for n in x]
plot(x, y, 'g', label='n')
legend(loc=2)
```

Out[39]:

<matplotlib.legend.Legend at 0x112331790>



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 به وضوح از یکدیگر دور می‌شوند. به شکل دقیق‌تر می‌توان گفت $\lim_{n \rightarrow \infty} \frac{T1(n)}{n^2}$ و $\lim_{n \rightarrow \infty} \frac{T1(n)}{T2(n)}$ هر یک برابر با عددی ثابت می‌شوند درحالی‌که $\lim_{n \rightarrow \infty} \frac{T2(n)}{n}$ چندجمله‌ای خطی است که با افزایش n بی‌کران بزرگ می‌شود.

بنابراین با وجود آن که 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 دسته‌ی 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 [13]:

```
figure(figsize=(10, 7))
xlabel("n")
max_n = 500
x = range(1, max_n)
y = [math.log(n+5, 2)+5 for n in x]
plot(x, y, 'r', label='f(n)')
y = [2*math.log(n, 2) for n in x]
plot(x, y, 'g', label='cg(n)')
legend(loc=2)
```

Out[13]:

<matplotlib.legend.Legend at 0x10e17ab50>

(ب

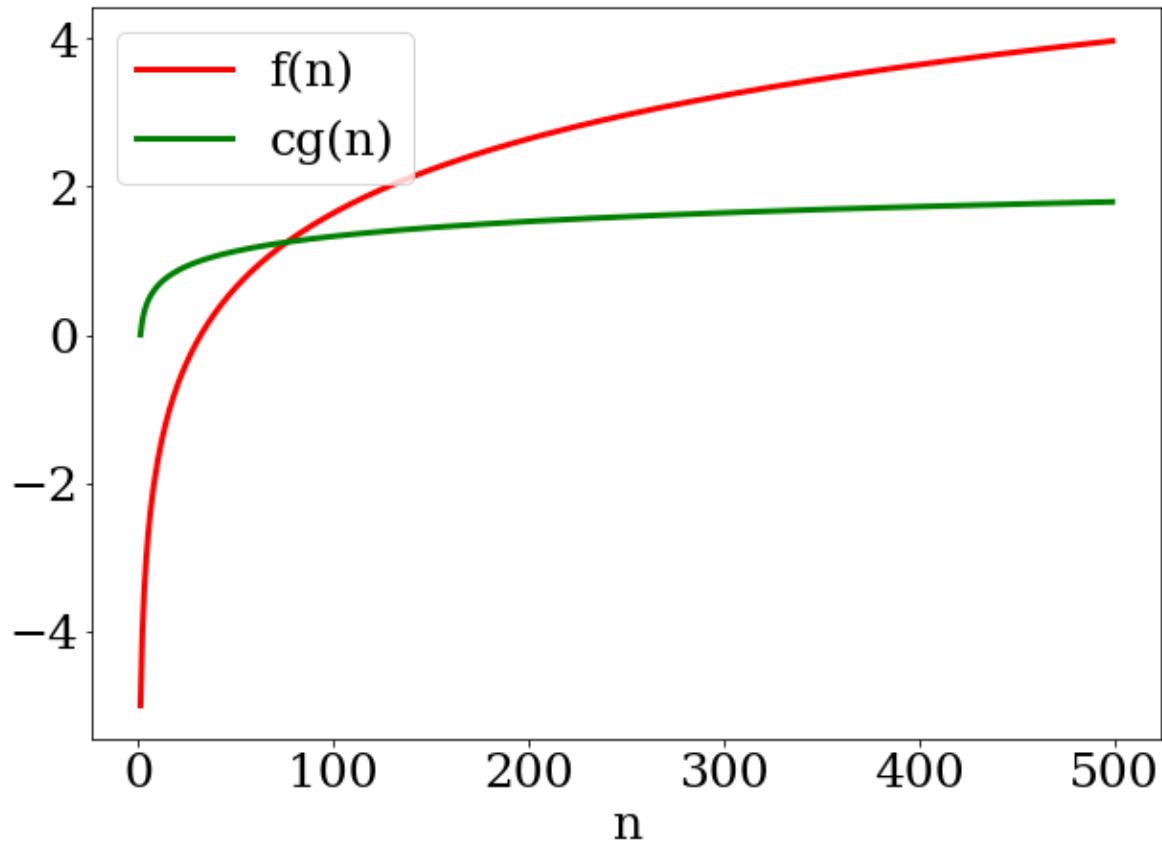
$$\Omega(g(n)) = \{ f(n) \mid \exists c, n_0 : 0 \leq cg(n) \leq f(n) \forall n > n_0 \}$$

In [12]:

```
figure(figsize=(10, 7))
xlabel("n")
max_n = 500
x = range(1, max_n)
y = [math.log(n,2) - 5 for n in x]
plot(x, y, 'r', label='f(n)')
y = [math.log(n,2) / 5 for n in x]
plot(x, y, 'g', label='cg(n)')
legend(loc=2)
```

Out[12]:

<matplotlib.legend.Legend at 0x10e14a7d0>



(ج

$$\Theta(g(n)) = \{ f(n) \mid f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n)) \}$$

به عبارت دیگر:

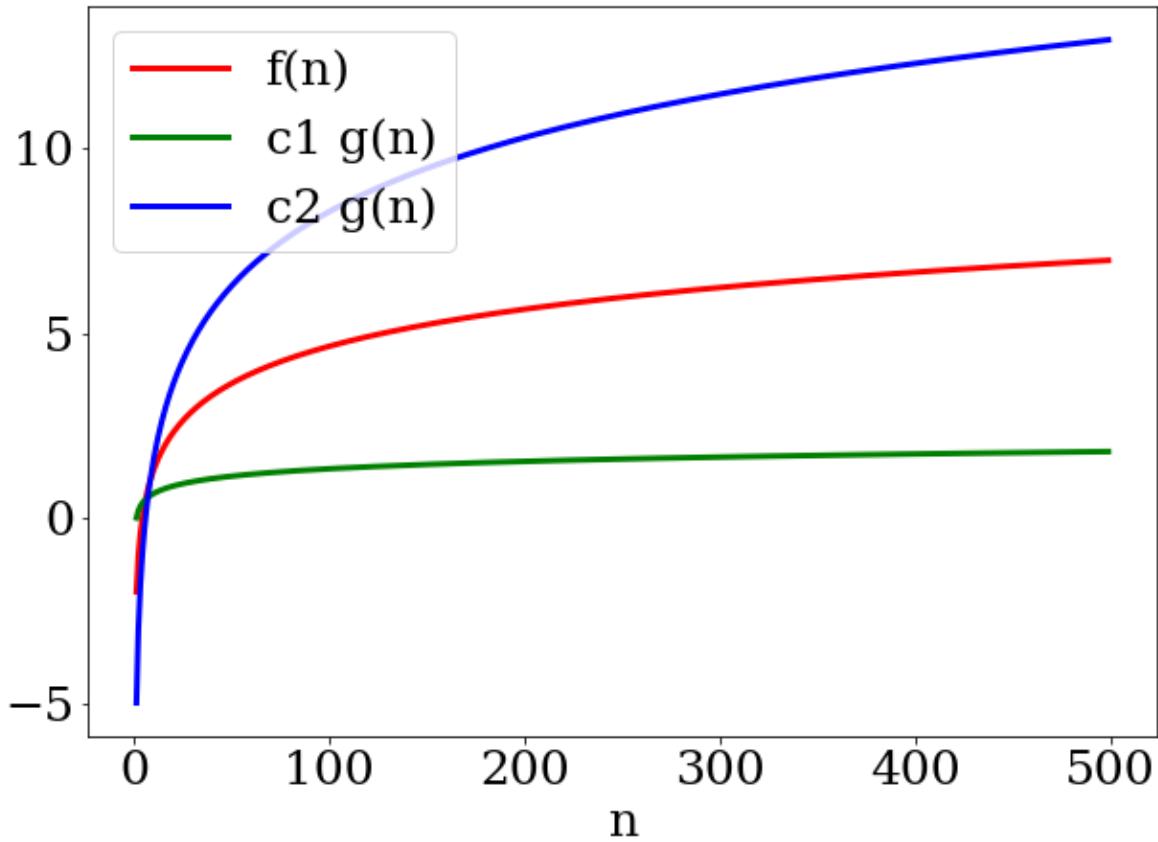
$$\Theta(g(n)) = \{ f(n) \mid \exists c_1, c_2, n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n > n_0 \}$$

In [15]:

```
figure(figsize=(10, 7))
xlabel("n")
max_n = 500
x = range(1, max_n)
y = [math.log(n,2) - 2 for n in x]
plot(x, y, 'r', label='f(n)')
y = [math.log(n,2) / 5 for n in x]
plot(x, y, 'g', label='c1 g(n)')
y = [2*math.log(n,2) - 5 for n in x]
plot(x, y, 'b', label='c2 g(n)')
legend(loc=2)
```

Out[15]:

<matplotlib.legend.Legend at 0x10e3aedd0>



در واقع اگر یک تابع $f(n)$ در $\mathcal{O}(g(n))$ باشد به عبارت نادقيق می‌توان گفت که اين تابع برای n ‌های به اندازه‌ی کافی بزرگ، $f(n) \leq g(n)$ و همچنین درمورد $f(n) \geq g(n)$ می‌توان به صورت شهودی عبارت $f(n) \in \Omega(g(n))$ را به کار برد. با توجه به اين دو عبارت اگر $f(n) \in \Theta(g(n))$ در يك دسته قرار دارند.

برای بررسی توابع، سعی می‌کنیم تابع ساده‌تر و همدسته‌ی آن را پیدا کنیم. برای الگوریتم ابتدایی این تابع $f(n) = n^2$ و برای الگوریتم مرتب سازی حبابی این تابع همان n^2 است. در هنگام یافتن تابع ساده‌تر از ضرایب صرف نظر می‌کنیم و بزرگ‌ترین عنصر در تابع اولیه را در نظر می‌گیریم.

تابع $f(x)$ که به ازای $c = 1, x_0 = 5$ وجود دارد ($c \in \mathcal{O}(g(n))$)

$$f(x) \leq cg(x) \quad x \geq x_0$$

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 تابع کدام هستند؟

In [5]:

```
def f(N):
    sum = 0;
    for i in range(N):
        for j in range(i+1, N):
            for k in range(j+1, N):
                for h in range(k+1, N):
                    sum += 1
    return sum

print(f(20))
```

4845

زمان اجرای تابع g در کد زیر را نیز بررسی کنید.

In [32]:

```
def G(N, x, A): #A: array of size N
    s, e = 0, N
    while(e - s > 1):
        mid = (s+e)//2
        if (A[mid] <= x):
            s = mid
        else:
            e = mid
    return A[s] == x
```

In [33]:

```
A = [1, 5, 10, 22, 31, 44, 55, 69, 75, 85, 100]
N = len(A)
G(N, 78, A)
G(N, 85, A)
```

Out[33]:

True

مثال

$$T(n) = a_n n^k + \dots + a_1 n + a_0$$

اثبات کنید $T(n) \in \mathcal{O}(n^k)$

مثال

$$T(n) = 2^{n+10}$$

اثبات کنید $T(n) \in \mathcal{O}(2^n)$

مثال

$$T(n) = 2^{10n}$$

اثبات کنید $T(n) \notin \mathcal{O}(2^n)$

مثال

برای مقایسه‌ی توابع بهرحال 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.

2^n

2^{2^n}

n^2

$n^2 \log n$

$n^{\log n}$

n^n

مثال

روابط زیر را اثبات کنید.

$n! = \mathcal{O}(n^n)$

$n! = \Omega(2^n)$

$\lg(n!) = \Theta(n \lg n)$

مثال

روابط زیر را اثبات کنید:

مقایسه تابع های مهم

constant = $\mathcal{O}(1)$

logarithmic = $\mathcal{O}(\log n)$

linear = $\mathcal{O}(n)$

$n \log n = \mathcal{O}(n \log n)$

quadratic = $\mathcal{O}(n^2)$

cubic = $\mathcal{O}(n^3)$

polynomial, k = $\mathcal{O}(n^k)$

exponential = $\mathcal{O}(k^n)$

factorial = $\mathcal{O}(n!)$

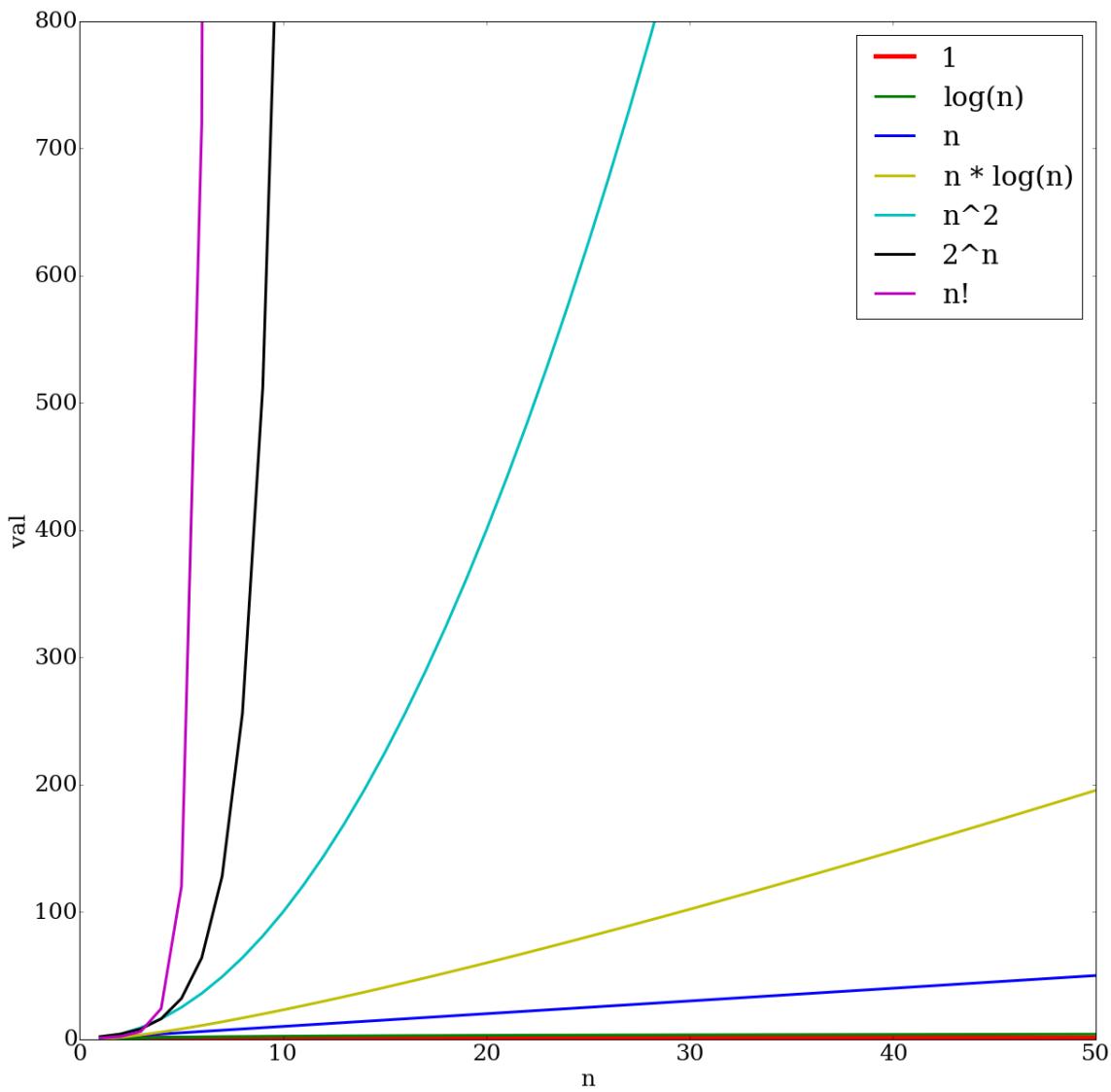
در نمودار زیر روند رشد چند تابع را می‌توانید مقایسه کنید:

In [43]:

```
figure(figsize=(20, 20))
xlabel("n")
ylabel("val")
ylim(0, 800)
max_n = 50
x = range(1, max_n + 1)
y = [1 for n in x]
plot(x, y, 'r', label='1', linewidth=5)
y = [math.log(n) for n in x]
plot(x, y, 'g', label='log(n)')
y = [n for n in x]
plot(x, y, 'b', label='n')
y = [n * math.log(n) for n in x]
plot(x, y, 'y', label='n * log(n)')
y = [n ** 2 for n in x]
plot(x, y, 'c', label='n^2')
y = [2 ** n for n in x]
plot(x, y, 'k', label='2^n')
y = [math.factorial(n) for n in x]
plot(x, y, 'm', label='n!')
legend(loc=1)
```

Out[43]:

```
<matplotlib.legend.Legend at 0x113fc6290>
```



بِنَامِ خَدَا

دانشگاه آزاد اسلامی شیراز - رشته مهندسی کامپیوتر

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل سوم، بخش دوم: الگوریتم‌های بازگشتی

فهرست محتویات

- مقدمه
- روش حدس و استقرا
- قضیه ساندویچ
- روش تغییر متغیر
- قضیه اصلی
- مرتب‌سازی ادغامی
- 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 مدنظر هستند و همه‌ی ثابت‌ها اعداد حقیقی مثبتند.

بسیاری از الگوریتم‌هایی که ما در این درس با آن‌ها سروکار 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 سروکار پیدا می‌کنیم.

به عنوان اولین و ساده‌ترین نمونه بیایید به جست و جوی دودویی دقت کنیم. فرض کنید یک آرایه‌ی مرقب با n عنصر به شما داده شده است و سپس یک نفر مرتب‌اژ شما سؤال‌هایی به این فرم 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 جست و جوی 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 جست و جو ادامه

می‌دهیم و در غیر این صورت فقط در نیمه‌ی قبل از عنصر وسط. این روند را می‌توان به این شکل پیاده‌سازی کرد:

در اینجا pu و $nwod$ انتهای و ابتدای بازه‌ای هستند که در حال جست و جو در آنیم؛ البته لازم نیست به جزییات پیاده‌سازی تابع جستجوی دودویی دقیق‌تر کنید زیرا در فصول آینده‌ی این درس به تفصیل مورد بررسی قرار می‌گیرد.

In [9]:

```
def binary_search(array, x, down, up):  
  
    if down == up - 1:  
        mid = (up + down) // 2  
        if array[mid] != x:  
            return "NOT FOUND"  
        else:  
            return "FOUND"  
    mid = (up + down) // 2  
    if array[mid] == x:  
        return "FOUND"  
    elif array[mid] < x:  
        return binary_search(array, x, mid, up)  
    else:  
        return binary_search(array, x, down, mid)
```

In [10]:

```
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 رویه‌ی بازگشتنی را به دست آوریم. فرض کنید زمان اجرای این رویه در بدترین حالت (حالتی که بیشترین زمان مصرف می‌شود) روی یک آرایه با اندازه‌ی 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 جست و جو می‌پردازیم کاهش می‌یابد، پس امکان ندارد جست و جو تا ابد ادامه پیدا کند.

و رویه‌ی `binary_search` حداقل n بار فراخوانی می‌شود و هر بار فراخوانی این تابع زمانی ثابت مصرف می‌کند پس می‌توانیم نتیجه بگیریم که فرآیند جست و جو در مجموع از $O(n)$ است.

اما می‌توانیم هزینه‌ی زمانی را به‌طور هوشمندانه‌تری حساب کنیم؛ اگر ℓ را برابر با لگاریتم طول بازه‌ای که در آن جست و جو می‌کنیم تعریف کنیم، این بار مشاهده می‌کنیم که ℓ هم با هر فراخوانی بازگشتی رویه کاهش می‌یابد و درنهایت به صفر می‌رسد؛ در نتیجه زمان اجرای این الگوریتم از $O(n \log n)$ است. (در اینجا دیگر ℓ 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 $\Omega(n \log n)$ است و در نتیجه از $\Theta(n \log n)$ می‌باشد؟ این کار را می‌توانید با ساختن بی‌نهایت مثال انجام دهید!

تذکر: ما همیشه در مبنای ۲ لگاریتم می‌گیریم.

روش حدس و استقرا

کاری که در اینجا برای تحلیل زمان اجرای جست و جوی 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 بھینه 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 متوجه 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 جست و جوی دودویی اگر فرض کنیم زمان لازم برای این جست و جو بر اساس تعداد مقایسه‌هایی که بین عناصر می‌کنیم تعیین می‌شود و $T(n)$ را برابر با تعداد مقایسه‌ها در اجرای یک نمونه جست و جوی دودویی روی یک آرایه به طول n در بدترین حالت (حالتی که بیشترین مقایسه را لازم دارد) تعریف کنیم، می‌بینیم که $T(1) = 1$ و برای هر $n \leq 2$

$$\text{داریم: } 1 + \left(\left\lceil \frac{1-n}{2} \right\rceil \right) T = (n)T$$

از این پس فقط به بررسی شیوه‌های پیدا کردن پیچیدگی توابعی می‌پردازیم که به شکل بازگشتی تعریف شده‌اند و به این که این توابع زمان اجرای چه رویه‌هایی را نشان می‌دهند توجه نمی‌کنیم. در اینجا می‌خواهیم کلاس پیچیدگی تابع زیر را پیدا کنیم:

$$\left. \begin{array}{l} 1 = n \\ 1 < n \\ 1 + \left(\left\lceil \frac{1-n}{2} \right\rceil \right) T \end{array} \right\} = (n)T$$

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 $T(n) \in \Theta(n \log n)$.

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 $T(n) \in \Theta(n \log n)$. این معادل آن است که نشان دهیم که یک عدد ثابت c وجود دارد که برای هر n طبیعی به اندازه‌ی کافی بزرگ داریم $n \log c \geq T(n)$. 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 استقرای نشان دهیم که برای هر n داریم $n \log c \geq T(n)$. خیلی راحت می‌توانیم بگوییم بگذار c یک مقدار ثابت مثلاً ۱۰۰ باشد و بعد سعی کنیم با استقرای 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 استقرای تمام شد c را طوری انتخاب کنیم که رو شمان جواب بدهد. پس بیایید استقرای بزنیم. ما در اینجا از استقرای قوی استفاده می‌کنیم.

فرض استقرای آن است که به ازای هر $k > n$ داریم $k \log c \geq T(k)$ و حکم استقرای آن است که $n \log c \geq T(n)$. حال داریم:

$$n \log c \geq 1 + \frac{n}{2} \log c \geq 1 + \left\lceil \frac{1-n}{2} \right\rceil \log c \geq 1 + \left(\left\lceil \frac{1-n}{2} \right\rceil \right) T = T(n)$$

دقت کنید که آخرین نابرابری فقط وقتی درست است که $c \leq 1$ 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 استقرای برای $n = 1$ برقرار نیست، چرا که $\log 0 = 1$ اما خوش‌بختانه ما نیازی نداریم که حکم استقرای را برای همه‌ی مقادیر n ثابت کنیم و کافیست برای n ‌های به اندازه‌ی کافی بزرگ درست باشد. پس می‌توانیم $n = 2, 3$ را به عنوان پایه‌های استقرای 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 استقرای حکم ثابت شده است. (چرا نمی‌توانیم فقط $n = 2$ را به عنوان پایه‌ی استقرای در نظر بگیریم؟)

به عنوان یک تمرین سعی کنید با روشی مشابه ثابت کنید که $T(n) \in \Omega(\log n)$.

استقرای 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. مثلا فرض کنید که می‌خواهید این تابع را بررسی کنید (این تابع از مرتب‌سازی ادغامی به دست آمده است):

$$1 = n \quad 0 \\ 2 \leq n(n)O + \left(\left\lceil \frac{n}{2} \right\rceil \right) T + \left(\left\lfloor \frac{n}{2} \right\rfloor \right) T \quad \left. \right\} = (n)T$$

در اینجا منظور از $O(n)$ که در تعریف تابع آمده است این است که تابعی مانند $f(n)O$ وجود دارد که:

$$\left. \begin{array}{l} 1 = n \\ 0 \\ 2 \leq n(n)f + \left(\left\lceil \frac{n}{2} \right\rceil \right)T + \left(\left\lfloor \frac{n}{2} \right\rfloor \right)T \end{array} \right\} = (n)T$$

یا به عبارت دیگر عدد ثابت مثبت d وجود دارد که:

$$\left. \begin{array}{l} 1 = n \\ 0 \\ 2 \leq nnd + \left(\left\lceil \frac{n}{2} \right\rceil \right)T + \left(\left\lfloor \frac{n}{2} \right\rfloor \right)T \end{array} \right\} \geq (n)T$$

بنابراین چون در اینجا فقط یک رابطه‌ی بازگشتی برای یک کران بالا از تابع T داریم، امکان بررسی Ω 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 $(^2n)O \in (n)T$ یا حتی بهتر $n \log n \in (n)T$ ، 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 مشکل ساز است آن کف و سقف است که استدلال‌ها را سخت‌تر می‌کنند. قطعاً برای ما بسیار ساده‌تر است که با این رابطه کار کنیم:

$$\left. \begin{array}{l} 1 = n \\ 0 \\ 1 < n(n)O + \left(\frac{n}{2} \right)' T2 \end{array} \right\} = (n)' T$$

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 بدهیم. فرض کنید که در تحلیل O به جای تابع بازگشتی T تصمیم می‌گیرید که چند فرض اضافه کنید و تابع بازگشتی T' را تحلیل کنید، در این صورت فرض‌های شما سازگار نامیده می‌شوند هرگاه $T(O') \in T'$. 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 مواطن سازگاری فرض‌هایتان باشید.

- این فرض که n توانی از ۲ یا یک عدد ثابت دیگر است
- این فرض که کف و سقف‌ها را می‌توان نادیده گرفت
- این فرض که در استقرار نیازی به چک کردن پایه نداریم و متعاقباً این که در تعریف تابع بازگشتی نیازی به بیان حالت پایه نداریم

- این فرض که تابع مورد بررسی صعودی است (خیلی وقت‌ها برای اثبات سازگاری این یکی باید به مسأله‌ی 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 مسأله 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 بربور دکنیم 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:

$$nd + (2/n)T2 \geq (n)T$$

می‌خواهیم ثابت کنیم $T(n) \geq n \log n$. تنها کاری که باید انجام دهیم بررسی گام استقراء است:

$$nd + nc - n \log n \geq nd + (1 - n \log) \times \frac{n}{2} \times c \times 2 \geq nd + (2/n)T2 \geq (n)T$$

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 کم کردن جمله‌ی کمارزش، قضیه‌ی ساندویچ و تغییر متغیر هستند. یک مسئله‌ی 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 استقراء 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 جست و جو کنید.

کم کردن جمله‌ی کمارزش

به عنوان اولین مسئله فرض کنید که تابع بالا را کمی تغییر داده‌ایم و حالا به این شکل است:

$$(n \lg \lg n)O + (2/n)T^2 \geq (n)T$$

در نگاه اول مشکلی وجود ندارد. تابع را به این فرم می‌نویسیم:

$$n \lg \lg n d + (2/n)T^2 \geq (n)T$$

و سعی می‌کنیم استقراء بزنیم و ثابت کنیم که $T^2 \geq nc$. این کار ساده است:

$$T^2 \geq n \log \log n d + 2 \geq n \log \log n d + (2/n)T^2 \geq (n)T$$

انتخاب c طوری که نابرابری آخر درست باشد آسان است. پس نشان دادیم که $O(n^2)$. اما آیا می‌توانیم بهتر از این کار کنیم؟ مثلاً آیا می‌توانیم نشان دهیم $O(n \lg n)$ ؟ سعیمان را می‌کنیم.

می‌خواهیم به صورت استقرایی نشان دهیم $n \log nc \geq (n)T$ داریم:

$$nc = n \log \log n d + (1 - n \log) \times \frac{n}{2} \times c \times 2 \geq n \log \log n d + (2/n)T2 \geq (n)T$$

و عبارت آخر فقط در صورتی کوچک‌تر از $n \log nc$ می‌شود که بتوانیم c را طوری انتخاب کنیم که $nc - n \log \log n d$ برای هر n کوچک‌تر از صفر شود و می‌دانیم که این کار غیرممکن است. (چرا؟)

پس استقرای ما به بنبست خورد. اگر $(n)T \neq O(n \lg n)$ هر کار دیگری هم که بکنیم به شکست می‌انجامد اما در این مورد خاص واقعاً $O(n \lg n) \geq (n)T$ ولی استقرای ما قدرت کافی برای اثبات آن را ندارد. شیوه‌ای که در اینجا برای برآمدن از پس این مشکل استفاده می‌کنیم ممکن است عجیب جلوه کند اما وقتی به روند استقرا دقت کنید ایده‌ی جالبی که پشت آن است را می‌بینید.

کاری که می‌کنیم این است که یک جمله‌ی کم‌ارزش‌تر را که خودمان به شکلی هوشمندانه انتخاب کرده‌ایم از فرض و حکم استقرا کم می‌کنیم. در این مورد خاص به جای این که سعی کنیم با استقرا ثابت کنیم که $n \log nc \geq (n)T$ سعی می‌کنیم با استقرا ثابت کنیم که $n \log \log na - n \log nc \geq (n)T$.

در اینجا a یک عدد ثابت است که مثل c بعد از استقرا انتخابش می‌کنیم.

بدیهی است که حکم دوم اولی را نتیجه می‌دهد و اگر آن را اثبات کنیم نشان داده‌ایم که $O(n \lg n) \geq (n)T$ اما اتفاق جالبی که در عمل می‌افتد این است که اثبات این حکم که به نظر سخت‌تر می‌آید، در واقع آسان‌تر است. به استقرا دقت کنید:

$$\lg \log n d + ((\frac{n}{2}) \log \log \frac{n}{2} \times a - (\frac{n}{2}) \lg \frac{n}{2} \times c)2 \geq n \lg \lg n d + (\frac{n}{2})T2 \geq (n)T$$

$$n \lg \lg n d + \frac{n}{2} \log \log a - nc - n \log nc =$$

حال کافیست a و c را طوری انتخاب کنیم که برای n های به قدر کافی بزرگ داشته باشیم:

$$0 > nc - \frac{n}{2} \log \log n a - n \lg \lg n d$$

در این حالت کافیست که مثلا a را بزرگ‌تر از d^2 انتخاب کنیم.

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

$$(n)h_2c \geq (n)g \geq (n)f_1c$$

روابطی تعريف شده بر یک کلاس از تابع‌ها نتیجه گرفت.

به عنوان یک مثال فرض کنید تابع بازگشتی زیر به شما داده شده است:

$$(n \log \log \log n) \Theta + (2/n) T 2 = (n) T$$

حال دو تابع زیر را در نظر بگیرید:

$$(n) \Theta + (2/n) U 2 = (n) U$$

$$(n \log \log n) \Theta + (2/n) V 2 = (n) V$$

با تحلیل‌هایی مشابه مثال‌های قبل می‌توان نشان داد که $U \in (n \log n) \Omega \Theta$ و $V \in (n \log n) O \Theta$ و در نتیجه $T \in (n \log n) O \Theta$

حال این مثال را که بعدها کاربردی خواهد بود بررسی می‌کنیم:

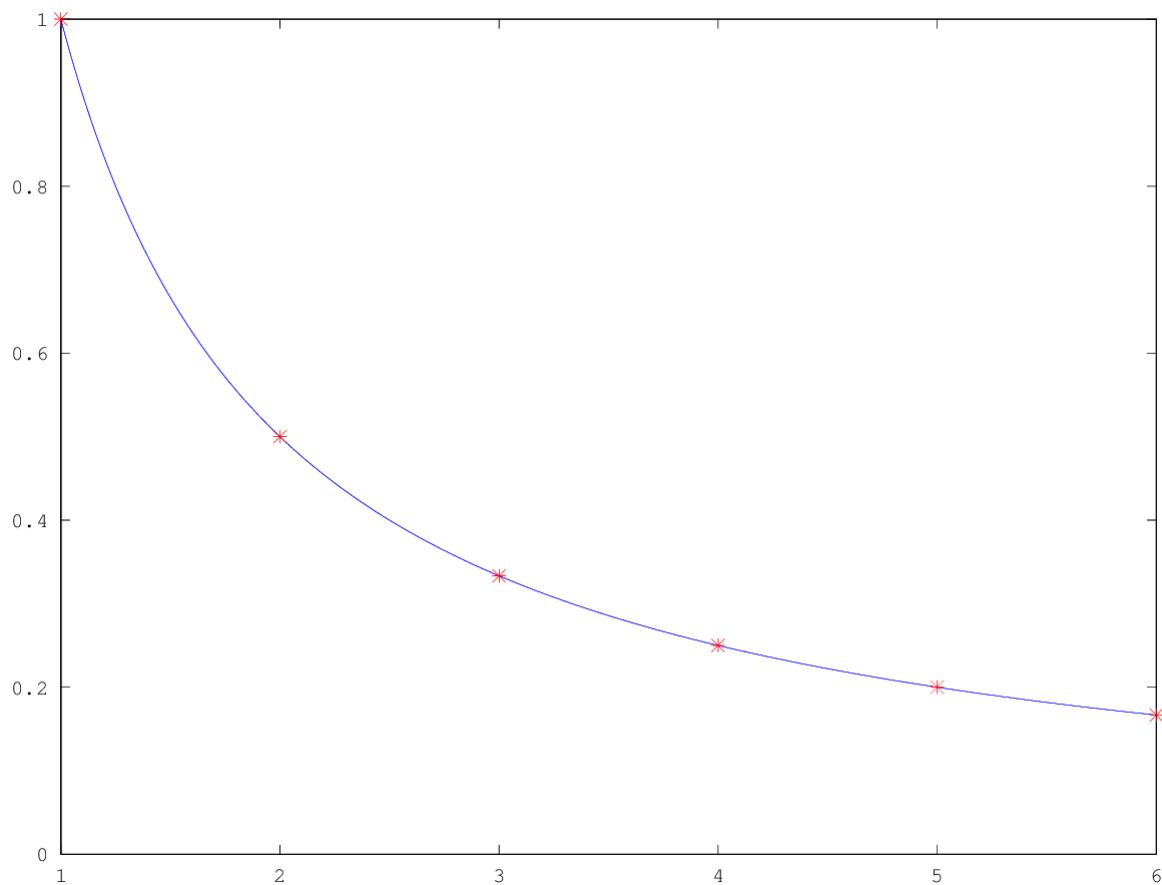
$$\left. \begin{array}{l} 1 = n \\ 1 < n \frac{1}{n} + (1 - n)T \end{array} \right\} = (n)T$$

شاید بهتر باشد که این تابع را این‌طور بنویسیم:

$$\frac{1}{i} \sum_{1=i}^n = (n)T$$

آنگاه داریم:

$$xd \frac{1}{x} \int_1^{1+n} \geq (n)T \geq xd \frac{1}{x} \int_1^n$$



برای این که این موضوع را ببینید به نمودار تابع $y = \frac{1}{x}$ که در این تصویر داده شده دقیق است. 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 $(n)T \in \Theta(n \lg n)$.

به عنوان یک مثال دیگر فرض کنید:

$$i \lg \sum_{1=i}^n = (n)T$$

می‌خواهیم ثابت کنیم که $(n)T \in \Theta(n \lg n)$. داریم:

$$(1 - n \lg)(1 - \frac{n}{2}) < i \lg \sum_{\left\lceil \frac{n}{2} \right\rceil = i}^n < i \lg \sum_{1=i}^n = (n)T$$

از طرفی بدیهیست که $n \lg n \geq (n)T$ ولذا $(n)T \in \Theta(n \lg n)$.

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 نوشتن تابع بر اساس آنها آسان‌تر شود. در اینجا به دو نمونه می‌پردازیم:

$$n \lg + (\bar{n} \sqrt{\bar{n}}) T_2 = (n)T$$

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 $n = {}^m 2$ یا معادلا $n \lg = m$. (آیا این فرض سازگار است؟ چرا؟) حال عبارت بالا را بر اساس متغیر جدید می‌نویسیم:

$$m + (\frac{m}{2})T2 = ({}^m 2)T$$

حال تابع جدید S را به این شکل تعریف می‌کنیم:

$$({}^m 2)T = (m)S$$

با این تعریف می‌توانیم بنویسیم:

$$m + (2/m)S2 = (m)S$$

این یک رابطه‌ی بازگشتی بسیار ساده‌تر است که می‌توانیم با تکنیک‌های قبلی حلش کنیم و به دست بیاوریم که $(n)T = ({}^m 2)T = (m)S$ ($m \lg m$) $O \exists (m)S$ اما $(m)S = (m)S$ ($m \lg m$) $O \exists (n)T$ یا معادلا $(m \lg m)O \exists (n)T$

$$(n \lg \lg n \lg)O \exists (n)T$$

و این تحلیل را تمام می‌کند. دقت کنید که همین روش برای Ω و در نتیجه Θ هم جواب می‌دهد.

حال همین مثال را کمی تغییر می‌دهیم و سعی می‌کنیم این تابع را تحلیل کنیم:

$$n \lg + (\bar{n} \sqrt{}) T3 = (n)T$$

اگر همان تغییر متغیر قبلی را اعمال کنیم به این رابطه می‌رسیم:

$$m + (2/m)S3 = (m)S$$

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 و بخش بعدی به آن می‌پردازد.

قضیه‌ی اصلی

اگر تابع بازگشتی به صورت زیر داشته باشیم:

$$(n)f + \left(\frac{n}{b}\right)Ta = (n)T$$

آنگاه چهار حالت ممکن است رخ دهد:

- وجود داشته باشد عدد $\epsilon < 0$ بطوری که $\epsilon^{+} \frac{a}{b} \log n > 0$. یعنی $n^{\frac{a}{b} \log n}$

به طور چند جمله‌ای از $f(n)$ کمتر باشد. در این صورت خواهیم داشت:

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

- وجود داشته باشد عدد $\epsilon < 0$ بطوری که $(n)f \epsilon^{+} \frac{a}{b} \log n > 0$. یعنی $f(n) < n^{\frac{a}{b} \log n}$

به طور چند جمله‌ای از $n^{\frac{a}{b} \log n}$ کمتر باشد. در این صورت خواهیم داشت:

$$(\frac{a}{b} \log n)^{\frac{a}{b} \log n} \Theta \in (n)T$$

- صورت داریم $((n)f)\Theta \in \frac{a}{b} \log n$ و $n^{\frac{a}{b} \log n} > f(n)$ یعنی $f(n) < n^{\frac{a}{b} \log n}$

$$((n) \lg \frac{a}{b} \log n) \Theta = ((n) \lg(n)f) \Theta \in (n)T$$

- 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 هریک 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 پایان یکی از لیست‌ها انجام می‌دهیم. در نهایت کافی است انتهای لیست دیگر را به انتهای آرایه‌ی مرتب شده اضافه کنیم تا کل عناصر مرتب شوند.

In [7]:

```
def mergesort(A, down, up):
    if len(A) < 2:
        return A
    mid = (up + down)// 2
    #B = A[:mid]
    #C = A[mid:]
    #print(B, C) # Remove comment to see running of algorithm
    B = mergesort(A, down, mid)
    C = mergesort(A, mid + 1, up)
    i = j = 0
    A = []
    while i < len(B) and j < len(C):
        if B[i] <= C[j]:
            A += [B[i]]
            i += 1
        else:
            A += [C[j]]
            j += 1
    A += B[i:] + C[j:]
    #print(A) # Remove comment to see running of algorithm
    return A
```

In [7]:

```
A=[5, -1, 3, 2, -4, 2, 8, 1, 0, -7, 9, 6, 1, 4]
A = mergesort(A)
print(A)
```

```
-----
-
RecursionError                                Traceback (most recent call last)
t)
<ipython-input-7-3a42487e1d09> in <module>()
    1 A=[5, -1, 3, 2, -4, 2, 8, 1, 0, -7, 9, 6, 1, 4]
----> 2 A = mergesort(A, 0, len(A))
    3 print(A)

<ipython-input-5-11907768dc43> in mergesort(A, down, up)
    6     #C = A[mid:]
    7     #print(B, C) # Remove comment to see running of algorithm
----> 8     B = mergesort(A, down, mid)
    9     C = mergesort(A, mid + 1, up)
   10    i = j = 0

... last 1 frames repeated, from the frame below ...

<ipython-input-5-11907768dc43> in mergesort(A, down, up)
    1     B = mergesort(A, down, mid)
    2     C = mergesort(A, mid + 1, up)
    3     i = j = 0
    4     while i <= mid and j <= up:
    5         if B[i] <= C[j]:
    6             A[k] = B[i]
    7             i += 1
    8         else:
```

اگر $T(n)$ را زمان اجرای مرتب‌سازی ادغامی برای یک آرایه به طول n در نظر بگیریم در این صورت داریم:

$$(n)\Theta + \left(\frac{n}{2}\right)T2 = (n)T$$

اگر با قضیهی اصلی بخواهیم به این مساله نگاه کنیم خواهیم داشت:

$$n = {}_2^2\log n = {}_b^a\log n$$

$$(n)\Theta \exists (n)f$$

و در نتیجه:

$$((n)f)\Theta \exists {}_b^a\log n$$

لذا طبق قضیهی اصلی داریم:

$$((n)\lg n)\Theta = ((n)\lg^a n)\Theta \exists (n)T$$

مثال: تابع زیر را با استفاده از قضیه اصلی تحلیل کنید:

$$(n)O + (3/n)T7 = (n)T$$

در این حالت $1 - \frac{7}{3}\log > \epsilon > 0$ برای همه مقادیر $n < \epsilon^{-\frac{7}{3}\log} n = \epsilon^{-\frac{a}{b}\log} n$

برقرار است. پس خواهیم داشت:

$$(1.77n)\Theta \approx (\frac{7}{3}\log n)\Theta = (\frac{a}{b}\log n)\Theta \exists (n)T$$

مثال: تابع زیر را با استفاده از قضیه اصلی تحلیل کنید:

$${}^3n \times 25 + (\frac{n}{2})T8 = (n)T$$

در این حالت $({}^3n \times 25)\Theta \exists \frac{8}{2}\log n = \frac{a}{b}\log n$

لذا طبق قضیه اصلی خواهیم داشت:

$$((n)\lg^3 n)\Theta = ((n)\lg^2 n)\Theta = ((n)\lg^a n)\Theta \exists (n)T$$

مثال: تابع زیر را با استفاده از قضیه اصلی تحلیل کنید:

$$((n)\lg^2 n)O + (2/n)T4 = (n)T$$

با وجود آن که $((n)\lg^2 n)O \exists {}^2n = {}^2\log n = \frac{a}{b}\log n < 0$ وجود ندارد که به ازای آن $\epsilon > 0$ هرچقدر هم مقدار کمتری برای n

انتخاب کنیم بازهم مقداری از n^0 وجود خواهد داشت که برای $n > n_0$ داشته باشیم $\epsilon n < \lg(n)$. به همین دلیل هیچ یک از دو تابع n^2 و $\lg^2(n)$ به طور چند جمله‌ای بیشتر از تابع دیگر نیست، و این دوتابع روند رشد یکسانی هم ندارند، لذا نمی‌توانیم از قضیه‌ی اصلی استفاده کنیم.

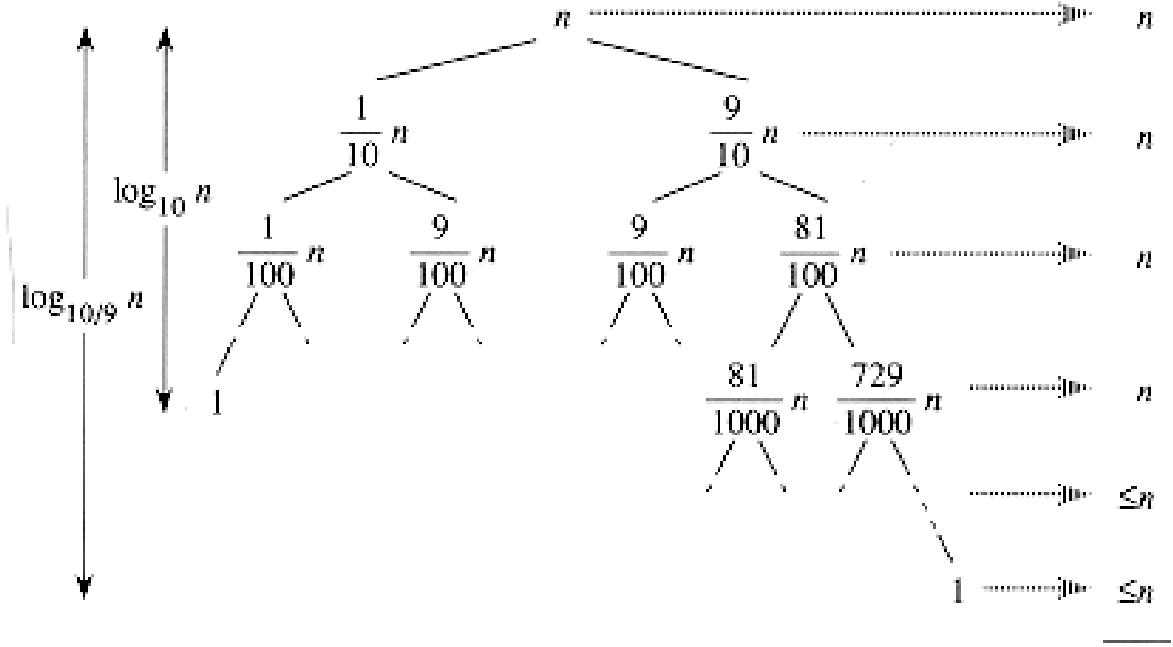
روش درخت بازگشت

بعضی اوقات با رسم یک درخت که روند تابع بازگشتی را در هر سطح نشان دهد می‌توان به سادگی توابع بازگشتی را تحلیل کرد. مثلا فرض کنید بخواهیم تابع زیر را تحلیل کنیم.

$$n + \left(\frac{n}{10}\right)T + \left(\frac{n^9}{10}\right)T = (n)T$$

$$1 = (1)T$$

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 $T\left(\frac{n^9}{10}\right)$ است، لذا می‌توانیم برای ریشه دو راس با اندازه‌های $\frac{n}{10}$ و $\frac{n^9}{10}$ 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:



$\Theta(n \lg n)$

منبع شکل: <http://staff.ustc.edu.cn/~csli>

توجه کنید در هر سطح از سطوح اولیه، چون مجموع اعداد نوشته شده روی راس‌ها n است پس مجموع هزینه‌ی هر سطح از درخت n است.

این درخت نامتوازن است و عمق برگ‌های آن بین $\log_{10/9} n$ تا $\log_{10} n$ متغیر است. لذا هزینه‌ی کل این تابع بین $\log_{10/9} n$ و $\log_{10} n$ است که هر دو مقدار از $\Theta(n \lg n)$ هستند. یعنی این تابع از $\Theta(n \lg n)$ است.

حل چند مثال دیگر

ثابت کنید:

$$(1 + c_n)\Theta = c_n + \dots + c_2 + c_1$$

اثبات:

می‌دانیم

$$\frac{1+c_n}{1+c_2} = {}^c\left(\frac{n}{2}\right)\frac{n}{2} \leq {}^c n + \dots + {}^c 2 + {}^c 1$$

در نتیجه

$$({}^{1+c_n})\Omega = {}^c n + \dots + {}^c 2 + {}^c 1$$

همچنین داریم

$${}^{1+c_n} = ({}^c n)n \geq {}^c n + \dots + {}^c 2 + {}^c 1$$

پس

$$({}^{1+c_n})O = {}^c n + \dots + {}^c 2 + {}^c 1$$

از دو نتیجه به دست آمده در بالا می توان نتیجه گرفت

$$({}^{1+c_n})\Theta = {}^c n + \dots + {}^c 2 + {}^c 1$$

حال روش جدیدی برای اثبات

$$(n\log)\Theta = \frac{1}{i} \sum_{1=i}^n$$

می توان این سری را به صورت زیر نمایش داد

$$\dots + \left(\frac{1}{15} + \dots + \frac{1}{8}\right) + \left(\frac{1}{7} + \frac{1}{6} + \frac{1}{5} + \frac{1}{4}\right) + \left(\frac{1}{3} + \frac{1}{2}\right) + (1)$$

در این نوع دسته بندی، مجموع اعداد هر دسته از 1 کمتر است و از $\frac{1}{2}$ بیشتر است و چون $n \log n$ تا دسته داریم پس:

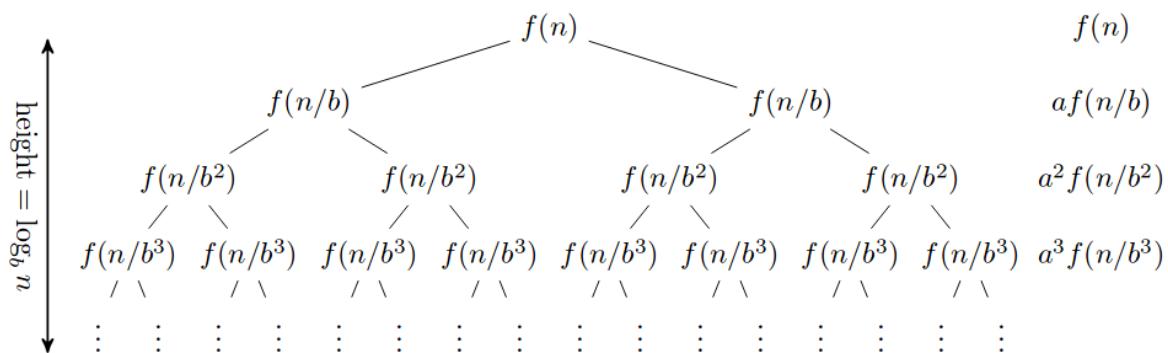
$$(n \log n) \Theta = \frac{1}{i} \sum_{1=i}^n$$

برای مطالعه بیشتر ۱: اثبات قضیه اصلی

رابطه‌ی بازگشتی زیر را در نظر بگیرید:

$$(n)f + \left(\frac{n}{b}\right)Ta = (n)T$$

برای اینکه شهود بیشتر نسبت به این رابطه پیدا کنیم می‌توانیم از نمایش به صورت درخت بازگشت برای حالتی که $a = 2$ است استفاده کنیم.



تعداد راس‌های درخت از مرتبه‌ی $O(a^b \log n)$ است. درنتیجه مجموع هزینه‌ی ثابت راس‌ها نیز از مرتبه‌ی $O(a^b \log n)$ است.

پس می‌توانیم $T(n)$ را به صورت زیر بنویسیم :

$$({}^a b^{\log n})O + \left(\frac{n}{i_b}\right) f^i a \sum_{0=i}^{n_b \log} = (n)T$$

سعی کنید با جایگذاری $f(n)$ در رابطه‌ی بالا حالت‌های مختلف قضیه‌ی اصلی را ثابت کنید.

نکته: اگر لازم داشتید می‌توانید از فرض مشتق پذیر 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:

$$\left. \begin{array}{l} 0 = n \\ 0 < n \frac{(1-n)T(1-n)n+1}{1+n^2} \end{array} \right\} = (n)T$$

می‌خواهیم اثبات کنیم که

$$\left(\frac{(n)gl}{n} \right) \Theta \ni (n)T$$

الف) اثبات می‌کنیم که

$$\left(\frac{(n)gl}{n} \right) O \ni (n)T$$

$$\frac{^2n}{1+^2n} \left[(1-n)T(1-n) + \frac{1}{n} \right] = (n)Tn$$

و $(n)g$ را به این صورت تعریف می‌کنیم: $(n)Tn = (n)g$ بنابراین:

$$\left[(1-n)g + \frac{1}{n} \right] \frac{^2n}{1+^2n} = (n)g$$

$$(1-n)g + \frac{1}{n} > (n)g$$

یعنی

$$\frac{1}{n} + \dots + \frac{1}{2} + 1 > (n)g$$

$$((n)gl)\Theta = \frac{1}{n} + \dots + \frac{1}{2} + 1 = (n)_1 h$$

? $((n)gl)\Theta = (n)_1 h$ چرا

بنابراین:

$$\left(\frac{(n)gl}{n} \right) O \ni (n)T$$

ب) اثبات می‌کنیم که:

$$\left(\frac{(n)gl}{n}\right)\Omega \ni (n)T$$

$$\left[(1-n)T\frac{^2(1-n)}{n} + \frac{1-n}{^2n} \right] \frac{^4n}{1-^4n} = (n)T\frac{^2n}{1+n}$$

تعریف می‌کنیم:

$$(n)T\frac{^2n}{1+n} = (n)g$$

به روابط زیر می‌رسیم:

$$\left[(1-n)g + \frac{1-n}{^2n} \right] \frac{^4n}{1-^4n} = (n)g$$

$$(1-n)g + \frac{1}{^2n} - \frac{1}{n} <$$

$$\left(\frac{1}{^2n} + \dots + \frac{1}{4} + 1\right) - \frac{1}{n} + \dots + \frac{1}{2} + 1 <$$

اگر رابطه‌ی $\frac{1}{2n} + \dots + \frac{1}{4} + 1 = (n)_2 h$ را تعریف کنیم و برای یک ثابت c فرض $c - (n)_1 h < (n)g$ (چرا این فرض درست است؟) در آن صورت $c > (n)_2 h$ کنیم که در نتیجه:

$$(c - (n)_1 h) \left[\frac{1}{2n} + \frac{1}{n} \right] < (n)T$$

$$\frac{c - (n)_1 h}{n} < (n)T$$

$$\left(\frac{(n)gl}{n} \right) \Omega = (n)f$$

طبق الف و ب نتیجه می‌شود: $\left(\frac{(n)gl}{n} \right) \Theta = (n)T$

In []:

بنام خدا

دانشگاه آزاد اسلامی شیراز - رشته مهندسی کامپیوتر

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل سوم، بخش سوم: تحلیل سرشکن

۱

فهرست محتویات

- 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 موافق شویم که هزینه‌ی یک عمل در آن بعضی موقع بسیار بیشتر تراز موقع دیگر است! شمارنده‌ی دودویی یکی از این داده‌ساختارهاست.

شمارنده‌ی دودویی:

فرض کنید یک شمارنده دودویی n بیتی داریم که درابتدا عدد ۰ را نشان می‌دهد. در هر مرحله عددی که این نمایشگر نشان می‌دهد یک 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 ۳ بیتی نشان داده شده است:

مرحله	بیت ۱	بیت ۲	بیت ۳	هزینه
صفر	.	.	.	
یک	۱	.	.	
دو	۰	۱	.	
سه	۰	۰	۱	
چهار	۰	۰	۰	۱
پنج	۰	۰	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!

ابتدا رفتار این شمارنده را به وسیله‌ی توابع زیر شبیه‌سازی می‌کنیم:

In [4]:

```
%matplotlib inline
from matplotlib.pyplot import *
rcParams.update({'font.size': 25, 'font.family': 'serif', 'lines.linewidth':3})
```

In [14]:

```
def increase(counter):
    n = len(counter)
    cost = 0
    for i in range(n-1, -1, -1): # Reverse Loop from n-1 to 0
        cost += 1
        if counter[i] == 1:
            counter[i] = 0
        else: # counter[i] == 0
            counter[i] = 1
            break;
    return cost

def simulate(n, m):
    # n: length of counter, m: number of increments
    counter = [0] * n
    cost = [0] * (m + 1)
    total_cost = [0] * (m + 1)
    average = [0] * (m+1)
    # total_cost[i] = total cost after i increment
    # total_cost[0] = 0
    for i in range(1, m + 1):
        cost[i] = increase(counter)
        total_cost[i] = cost[i] + total_cost[i - 1]
        average[i] = (total_cost[i] * 1.0) / i
        #print("Step ", i, "\tCounter ", counter)
        #print("\tCost ", cost[i], "\tTotal ", total_cost[i])
        #print ("\tAverage ",average[i])
    # Plot
    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 کند)

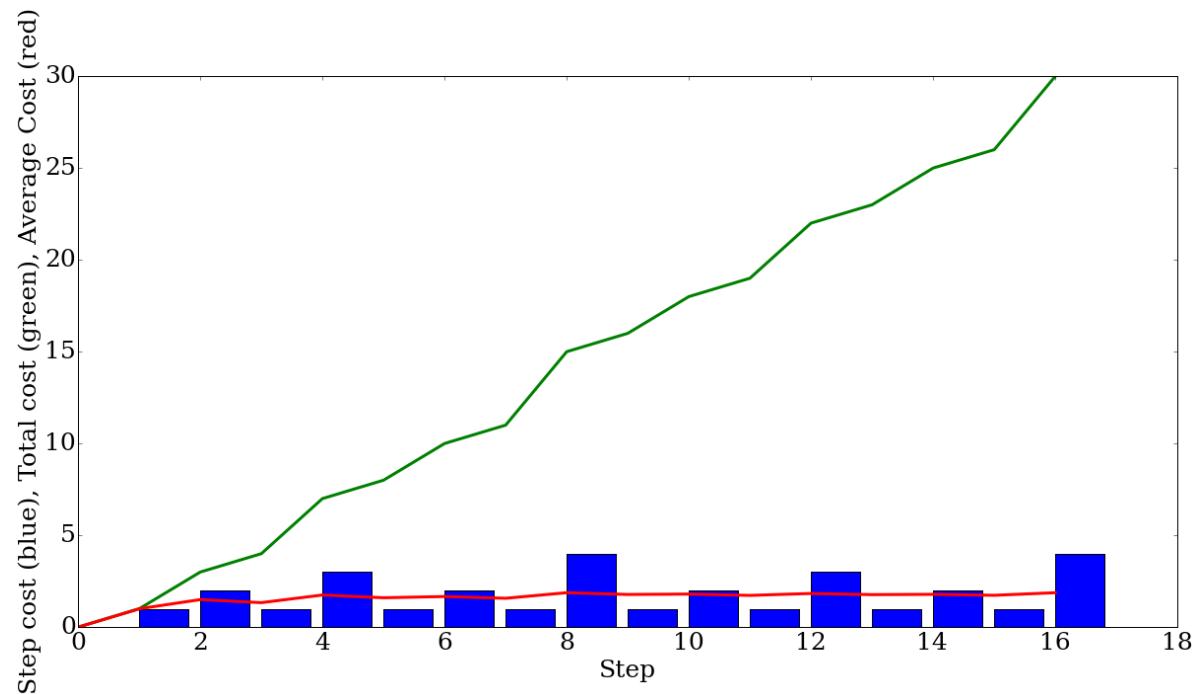
In [15]:

```
def plot_cost(cost, total_cost, average):
    figure(figsize=(20, 10))
    xlabel("Step")
    ylabel("Step cost (blue), Total cost (green), Average Cost (red)")
    x = range(len(total_cost))
    bar(x, cost)
    plot(x, total_cost, 'g')
    plot(x, average, 'r')
```

In [16]:

```
simulate(4, 2**4)
```

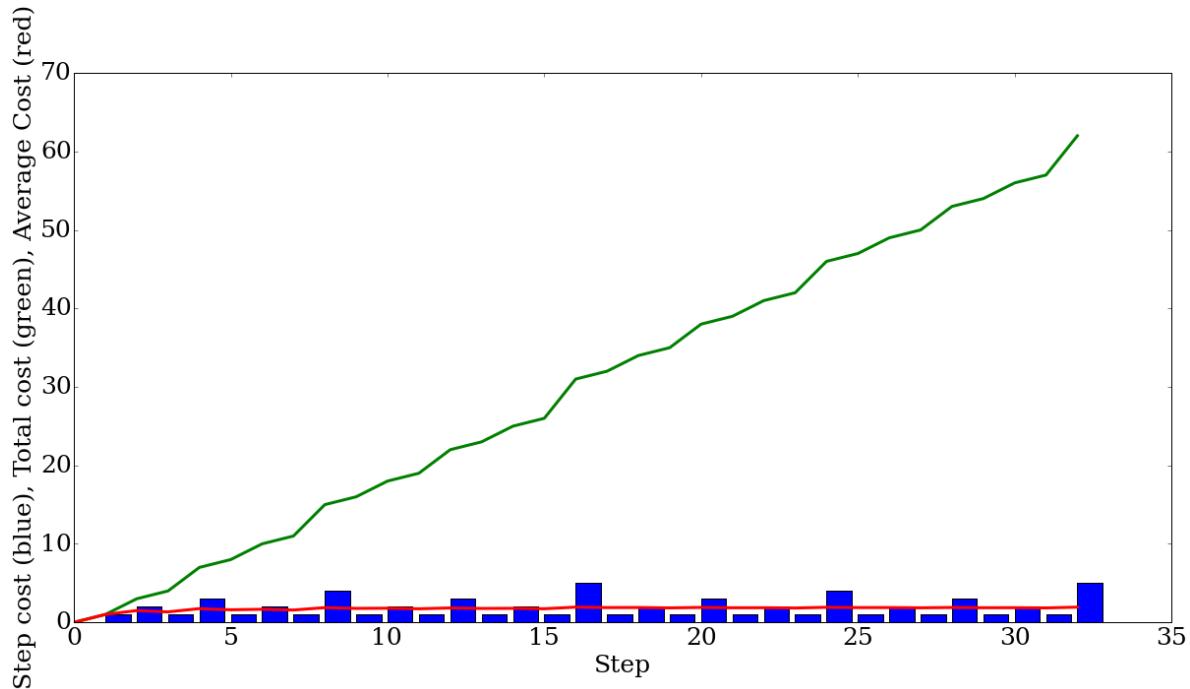
```
[0, 1, 3, 4, 7, 8, 10, 11, 15, 16, 18, 19, 22, 23, 25, 26, 30]
[0, 1.0, 1.5, 1.3333333333333333, 1.75, 1.6, 1.6666666666666667, 1.571428571
4285714, 1.875, 1.7777777777777777, 1.8, 1.7272727272727273, 1.83333333333333
333, 1.7692307692307692, 1.7857142857142858, 1.7333333333333334, 1.875]
```



In [17]:

```
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, 1.333333333333333, 1.75, 1.6, 1.666666666666667, 1.5714285714285714, 1.875, 1.777777777777777, 1.8, 1.72727272727273, 1.833333333333333, 1.7692307692307692, 1.7857142857142858, 1.7333333333333334, 1.9375, 1.823529411764706, 1.888888888888888, 1.8421052631578947, 1.9, 1.8571428571428572, 1.8636363636363635, 1.826086956521739, 1.9166666666666667, 1.88, 1.8846153846153846, 1.8518518518518519, 1.8928571428571428, 1.8620689655172413, 1.8666666666666667, 1.8387096774193548, 1.9375]
```



In []:

```
print()
```

نکته‌ی جالب اینست که قبل از هر عمل پر هزینه در شمارنده‌ی دودویی چندین عمل کم هزینه‌تر داریم. در واقع به ازای هر i ، هر 2^{i-1} مرحله یکبار بیت i -ام را تغییر می‌دهیم! پس برای شمردن از 0 تا m باید $\sum_{i=1}^{\lfloor \lg m \rfloor} \frac{m}{2^i} \leq 2m$ هزینه بپردازیم که بسیار کمتر از $n \times m$ 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 بدھید؟ 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 آوردیم مقایسه بکنیم متوجه تفاوت زیادی می‌شویم. برای ۳۲ بار افزایش یک شمارنده‌ی ۵ بیتی مجموع هزینه ۶۲ است، در حالی که $2m = 5 \times 32 = 160$ و $n \times m = 64$.

تحلیل سرشکن

شمارنده‌ی دودویی مثالی از داده‌ساختارهایی بود که هزینه‌ی یک عمل در آن بعضی موضع بسیار بیشتر تر از موقع دیگر است، همچنین دیدیم که قبل از هر عمل پرهزینه باید تعداد زیادی عمل کم‌هزینه‌تر انجام شود. در رویارویی با داده‌ساختارهای شبیه به شمارنده‌ی دودویی تحلیل بدترین حالت به یک کران بالای درست ولی بدینانه می‌انجامد.

تحلیل سرشکن نوعی از تحلیل برای مواقعی 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. مثلا در شمارنده‌ی دودویی هزینه‌ی میانگین هر عمل در بدترین حالت برابر با $O(1) = \frac{2m}{m}$ است در حالی که هزینه‌ی یک عمل در بدترین حالت از مرتبه‌ی $O(nm)$ 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 موضع مشابه از این ایده‌ها بهتر استفاده کنیم. یک روش که اصطلاحاً روش انبوهه هم گفته می‌شود و در تحلیل شمارنده هم استفاده شد ضمن 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 [1]:

```
def push(x):
    A[top] = x
    top = top + 1
```

برای پیاده سازی تابع حذف آخرین عدد پشته و برگرداندن آن (pop) کافی است

عملیات زیر را انجام دهیم:

In [1]:

```
def pop():
    top = top - 1
    x = A[top]
    return x
```

(در کد بالا فرض کردیم که top مخالف 0 باشد)

با توجه 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 آرایه فعلی تعریف کنیم، اعضای آرایه قبلی را به آرایه جدید منتقل کنیم و کار را در آن ادامه دهیم.

این عملیات هزینه زیادی به دنبال دارد، در نتیجه عملیات اضافه کردنی که منجر به این اتفاق می‌شوند زمان گیر خواهند بود، ولی با توجه به هزینه کم بسیاری از عملیات که ما را به این مرحله رسانده اند، اگر هزینه سرشکن برای هر عملیات را حساب کنیم، شاید هزینه‌ی بسیار کمتری شود. فرض می‌کنیم که هزینه انتقال یک آرایه به طول n به آرایه جدید برابر n است.

حالت اول:

اگر هنگام تعریف آرایه جدید طول آن را یکی بیش از طول آرایه قبلی قرار دهیم، هزینه سرشکن کل عملیات معقول می‌شود؟

خیر. اگر n عملیات اضافه کردن متوالی را در نظر بگیریم، هزینه کپی کردن اعداد به ترتیب برابر $1, 2, \dots, n-1$ خواهد شد، پس کل هزینه برابر $\frac{n \times (n-1)}{2}$ است و به صورت سرشکن هر عملیات $\frac{n-1}{2}$ هزینه دارد.

حالت دوم:

اگر هنگام تعریف آرایه جدید طول آن را دو برابر طول آرایه قبلی قرار دهیم، هزینه سرشکن کل عملیات چه می‌شود؟

در این حالت اگر n عملیات داشته باشیم $1 + 2 + 4 + \dots + 2^i$ عملیات کپی کردن خواهیم داشت که $\log(n) < i$ (اگر همهٔ عملیات از نوع اضافه کردن باشند، i بزرگترین عددی است که 2^i کمتر از n است) در نتیجه این تعداد کمتر از $n \times 2$ عملیات است و اگر هزینهٔ n عملیات دیگر را هم در نظر بگیریم، برای هر عملیات هزینهٔ سرشکن ۳ بدست می‌اید.

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 هزینهٔ سرشکن عملیات، روش حسابداری می‌باشد. برای مثال، مسالهٔ قبل را دوباره در نظر بگیرید، به ازای هر عملیات اضافه کردن، 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 کرده ایم استفاده می‌کنیم. چرا هیچ وقت پول کم نمی‌اوریم؟

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 هزینه انتقال آرایه را بپردازیم. حال می دانستیم که هر بار پس از پرشدن ظرفیت را ۲ برابر می کنیم. پس آخرین باری که ظرفیت پرشده بود و آن را ۲ برابر کرده بودیم، مقدارش $\frac{L}{2}$ بود و اکنون $\frac{L}{2}$ عنصر جدید اضافه کرده ایم.

و در نتیجه در حسابمان حداقل به مقدار L تومان پول داریم.

پس اگر برای هر عملیات ۳ تومان هزینه کنیم، همواره میتوانیم همه هزینه‌ها را پرداخت کنیم. در نتیجه هزینه سرشکن هر عملیات برابر ۳ است.

دقت کنید که در محاسبه هزینه سرشکن، هزینه‌ای که بدست می‌آوریم به ازای هر ترتیبی از عملیات داده شده همچنان حداکثر میانگین هزینه عملیات است و بر خلاف حساب کردن مرتبه زمانی متوسط، ما راجع به ورودی فرضی مانند تصادفی بودن آن‌ها نکردیم، فقط میانگین هزینه عملیات در طول زمان را حساب کردیم.

به نظر شما با استفاده از روش حسابداری چگونه می‌توان مساله شمارنده دودویی را حل کرد؟

تحلیل شمارنده دودویی به روش حسابداری

برای تغییر هر بیت ۰ به ۱، ۱ تومان هزینه می‌کنیم و ۱ 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 ۱ را به ۰ تغییر دهیم، چون قبلاً این بیت از ۰ به ۱ تبیدل شده است ۱ تومان در حسابش دارد، پس از همان پول برای تغییر این بیت استفاده می‌کنیم. هر مرحله می‌تواند تعدادی تغییر بیت از ۱ به ۰ داشته باشد، ولی دقیقاً

یک بیت از ۰ به ۱ تغییر پیدا می‌کند، در نتیجه هزینه‌ای که برای هر مرحله می‌پردازیم برابر ۲ 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 خواصی 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 پولی که در مرحله D_i در حساب داریم را با $\Phi(D_i)$ 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:

$$\Phi(D_0) = 0$$

$$\Phi(D_i) \geq 0 : D_i$$

هزینه‌ای که به حساب واریز می‌شود $= a_i - c_i = \Phi(D_i) - \Phi(D_{i-1})$ که a_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 شروطی که به طور طبیعی قرار دادیم، برای نشان دادن اینکه مجموع a_i ها از مجموع c_i ها بیشتر اند، کافی است. چون در این صورت اگر ما مجموع a_i ها را حساب کنیم یک کران بالا برای هزینه کل که مجموع c_i ها باشد به دست آورده‌ایم. با استفاده از شروط بالا داریم:

$$\sum_{n=0}^n a_i - \sum_{n=0}^n c_i = \Phi(D_n) - \Phi(D_0) = \Phi(D_n) \geq 0$$

بنابراین ثابت شد که اگر یک تابع با خواص بالا روی مراحل تعریف کنیم مجموع a_i ها یک کران بالا برای هزینه کل است.

بیایید در مثال شمارنده باینری از این تابع استفاده کنیم: فرض کنید تعداد ۱ 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 n_i بیت از یک به صفر تغییر می‌کند و ۱ بیت از صفر به یک تغییر می‌کند. بنابراین $1 + n_i = c_i$. حالا a_i را طوری پیدا می‌کنیم که شرط سوم برقرار باشد:

$$\Phi(D_i) - \Phi(D_{i-1}) = -n_i + 1$$

بر اساس شروط تابع: $\Phi(D_i) - \Phi(D_{i-1}) = a_i - c_i = a_i - (n_i + 1)$

نتیجه می‌گیریم: $a_i = 2$

مثال : پشته با حافظه‌ی بهینه

در مسألهٔ پیاده سازی پشته، طول آرایه را L و تعداد اعضای پشته را n در نظر بگیرید، می‌خواهیم L همواره از مرتبه $O(n)$ باشد.

در الگوریتم فعلی اگر 1000 عملیات اضافه کردن و سپس 999 عملیات حذف کردن را انجام دهیم، آرایه‌ای به طول 1024 خواهیم داشت (چرا؟)، که فقط ۱ خانه از آن استفاده شده است و مقدار زیادی حافظهٔ بی‌استفاده گرفته شده. راهکاری برای درست کردن این مشکل ارائه دهید و سپس هزینهٔ سرشکن عملیات را محاسبه کنید.

مثال : شمارنده پر هزینه!

فرض کنید یک شمارنده داریم که هزینه تغییر بیت i ام آن i است. با استفاده از روش انبوهه، حسابداری و تابع پتانسیل مقدار هزینه سرشکن هر عملیات افزایش را حساب کنید. راهنمایی: بسیار شبیه شمارنده معمولی است.

In []:

بنام خدا

دانشگاه آزاد اسلامی - رشته مهندسی کامپیوتر

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل چهارم: داده‌ساختارهای پایه‌ای

فهرست محتویات

- داده ساختارها
- صف
- پشته
- صف دوطرفه
- لیست پیوندی

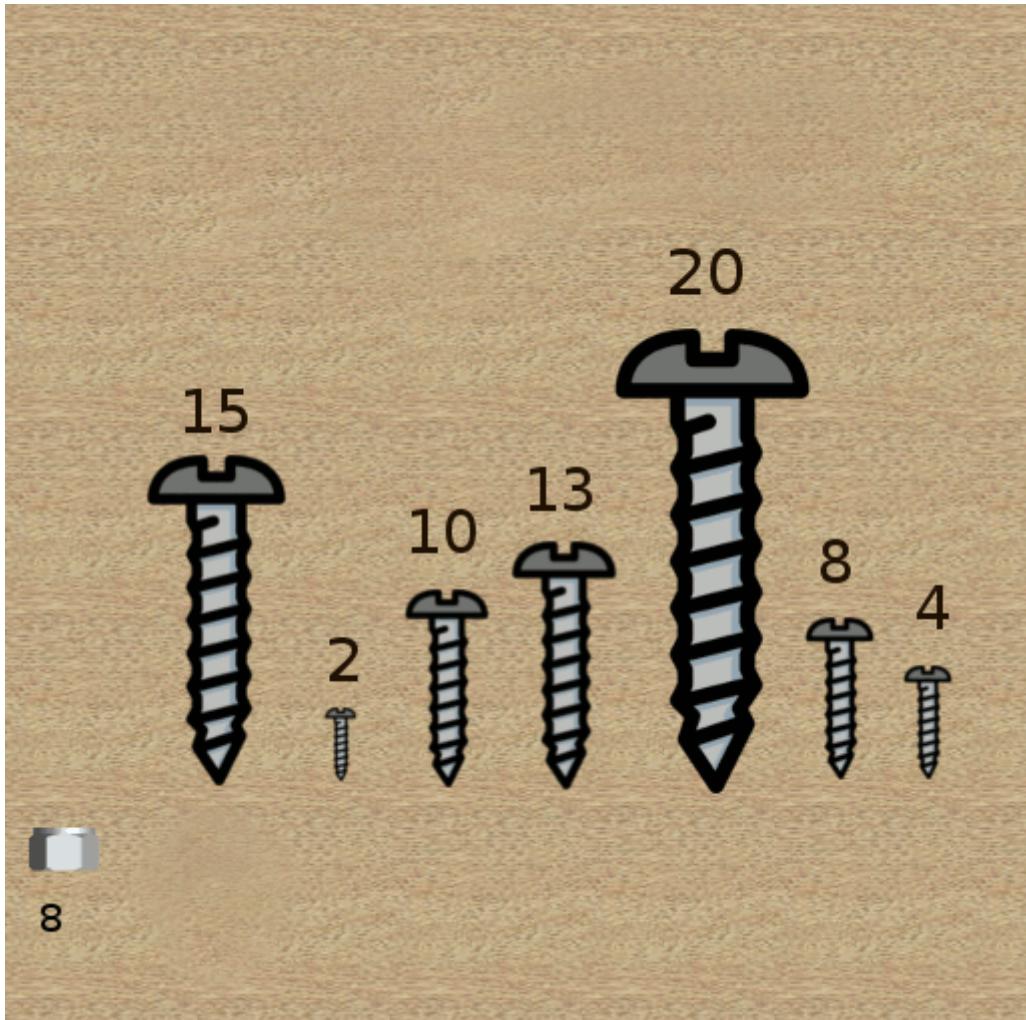
داده ساختارها

داده ساختارها همان طور که از اسم آن‌ها مشخص است روش‌ها و یا به زبانی دیگر ساختارهایی هستند که برای ذخیره‌ی ساختارمند داده‌ها استفاده می‌شوند تا در پاسخ دهی به سوالات ما کارا و سریع باشند و اگر چیزی که این داده ساختار مدلی از آن است در حال تغییر سریع باشد باید در تطابق دادن خود با آن نیز چابک باشند.

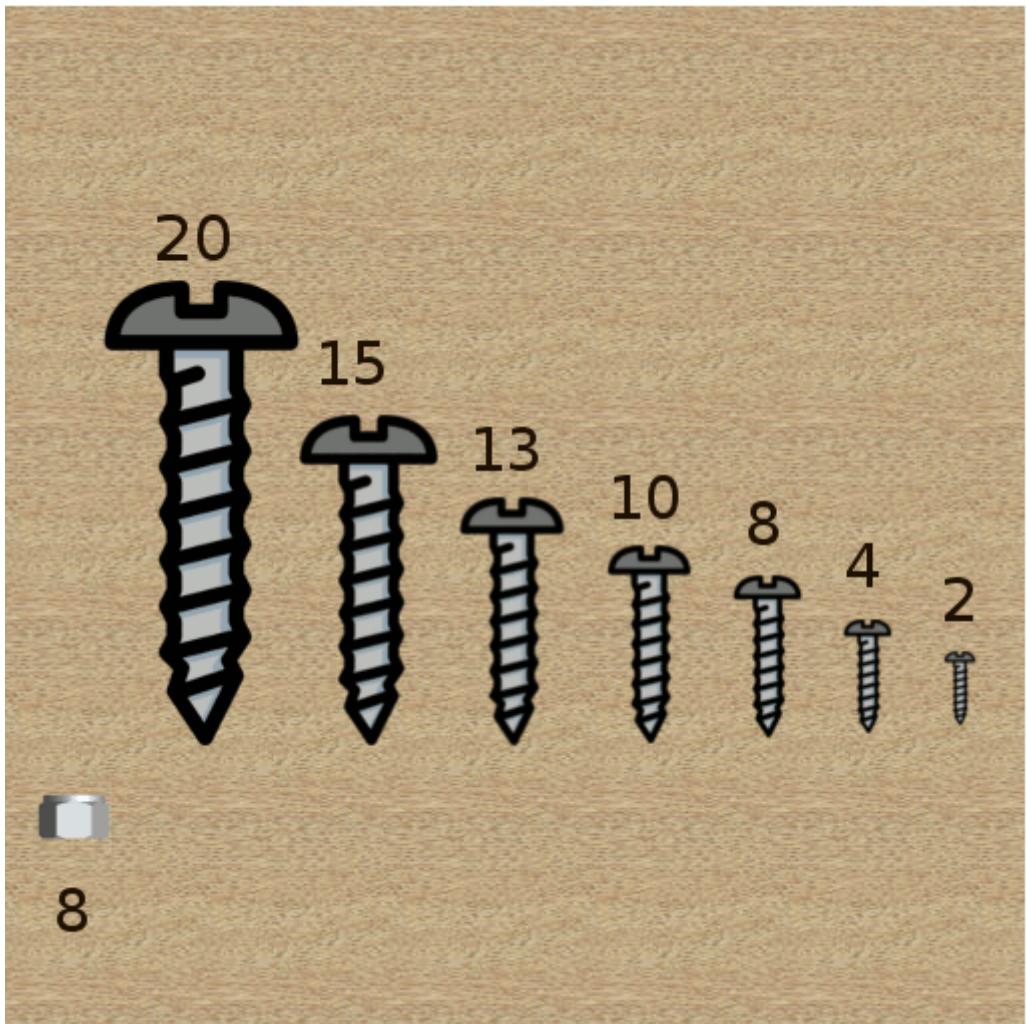
مشخص است که بسته به اینکه سوالاتی که ما از یک سری داده ها داریم چه هستند یا میزان تغییری که در داده ها ممکن است رخ دهد(مثلاً یا حذف شوند یا اضافه شوند) چقدر است(آیا اصولاً حذف و اضافه ای رخ میدهد) یا از چه نوعی است(مثلاً آیا تنها کوچکترین عنصر حذف میشود یا هر عنصری ممکن است حذف شود) باید نحوه ی ذخیره سازی آن ها نیز متفاوت باشد.

ممکن است تعاریف بالا تا حد زیادی گنج به نظر برسند که برای روشن شدن آن ها کافیست به مثال زیر توجه کنید. فرض کنید که آریا یک جعبه از پیچ ها با اندازه های مختلف دارد. هر بار که میلاد به او یک مهره میدهد، او میخواهد بداند که آیا پیچی هم اندازه ی آن مهره دارد یا نه؟ آریا برای ذخیره کردن اندازه ی این پیچ ها از یک آرایه استفاده میکند او برای این کار دو روش دارد:

- اندازه ها را بدون هیچ گونه ترتیب خاصی در آرایه بنویسد. مزیت این روش این است که اگر آریا یک پیچ جدید بخرد میتواند اندازه ی این پیچ را به راحتی در آخر آرایه ی خود اضافه کند اما هرگاه با یک مهره مواجه میشود باید با یک حلقه تمامی آرایه ی خود را چک کند که آیا پیچی هم اندازه ی این مهره دارد یا نه.



- اندازه‌ی پیچ‌ها را به صورت مرتب شده در آرایه ذخیره کند. اگر به مهره‌ی جدیدی بخورد کند میتواند به راحتی و با استفاده از جست و جوی دودویی بسیار سریعتر و کاراتر از این که تمامی اعضای آرایه را ببیند و تنها با دیدن معدودی از اعضای آرایه بفهمد که آیا مهره‌ای هم اندازه‌ی آن پیچ دارد یا خیر. اما نقطه ضعف این روش در این است که اگر آرایا یک پیچ جدید بخورد اضافه کردن این پیچ جدید به آرایه بدون آن که ساختارمندی آن (مرتب بودن اعضاء) به هم بخورد دشوار و زمان بر است



پشته، صف و لیست پیوندی

به نظر شما چگونه یک سری داده را که توالی در آن ها مهم است را ذخیره کنیم؟ در نظر بگیرید که داده ها یک به یک در حال وارد شدن به یا خارج شدن از برنامه هستند و باید آن ها را در داده ساختاری ذخیره کنیم که توالی آن ها را حفظ کند. در این جلسه داده ساختارهایی را بررسی میکنیم که یک توالی از عناصر را ذخیره می کنند و هر کدام برای نوع خاصی از اضافه شدن عناصر به خودشان یا حذف شدن عناصر بهینه شده اند.

صف

صف داده ساختاری است که عنصر جدید فقط در انتهای آن اضافه می شود و حذف عنصر تنها از اول آن اتفاق می افتد. مانند یک صفت نانوایی که هر کسی وارد می شود به انتهای صفت می رود و هر کسی بعد از گرفتن نان از ابتدای صفت خارج می شود.

صف باید از دستورات زیر پشتیبانی کند:

- `enqueue(x)`: عنصر x را در انتهای صفت درج می کند.
- `dequeue()`: اولین عنصر را حذف می کند و آن را باز می گرداند.
- `front()`: اولین عنصر صفت را باز می گرداند.
- `size()`: تعداد عناصر موجود در صفت را باز می گرداند.
- `is_empty()`: خالی بودن یا نبودن صفت را مشخص می کند.
- `is_full()`: پر بودن یا نبودن صفت را مشخص می کند.

توجه: تمامی این اعمال در $O(1)$ انجام می شوند.

پیاده سازی صفت معمولاً به کمک آرایه یا لیست پیوندی که در ادامه با آن آشنا خواهید شد انجام می شود.

پیاده سازی صفت به کمک آرایه

برای پیاده سازی صفت به کمک آرایه ابتدا یک آرایه و دو متغیر که یکی ابتدای صفت (`first`) و دیگری تعداد عناصری که در صفت قرار دارند (`num`) را نشان می دهد، تعریف می کنیم.

هنگامی که عنصر جدیدی اضافه می شود آن را در انتهای صفت قرار می دهیم و سپس مقدار `num` را یکی افزایش می دهیم. برای حذف کردن عنصر ابتدایی صفت نیز مقدار `num` را یکی کم می کنیم و همچنین مقدار `first` را یکی افزایش می دهیم. (توجه: البته در توضیح بالا باید در نظر داشته باشیم که از آرایه به صورت دوری استفاده می

کنیم برای مثال اگر سایز آرایه برابر ۵ باشد، عنصر بعد از عنصر پنجم، عنصر اول است.)

In [1]:

```
class Queue:
    # max_size: size of Q
    # Q: Array
    def __init__(self, max_size):
        self.max_size = max_size
        self.Q = [0] * max_size
        self.num = 0
        self.first = 0

    def enqueue(self, item):
        if self.num >= self.max_size:
            raise Exception("Queue overflow")
        self.Q[(self.num + self.first) % self.max_size] = item
        self.num += 1

    def dequeue(self):
        if self.num == 0:
            raise Exception("Queue empty")
        item = self.Q[self.first]
        self.first = (self.first + 1) % self.max_size
        self.num -= 1
        return item

    def front(self):
        if self.num == 0:
            raise Exception("Queue empty")
        return self.Q[self.first]

    def is_empty(self):
        return self.num == 0

    def size(self):
        return self.num

    def is_full(self):
        return self.num >= self.max_size
```

In [2]:

#Example

```
q=Queue(10) # (front of queue)[](back of queue)
q.enqueue("ra'na") # ["ra'na"]
q.enqueue("vez") # ["ra'na", "vez"]
q.enqueue("Arya") # ["ra'na", "vez", "Arya"]
print("queue size is: ",q.size())
print(q.dequeue(), "left the queue") # ["vez", "Arya"]
print("front of queue is:",q.front())
q.enqueue("milda") # ["vez", "Arya", "milda"]
q.dequeue() # ["Arya", "milda"]
q.dequeue() # ["milda"]
q.dequeue() # []
print("It was a queue")
```

```
('queue size is: ', 3)
("ra'na", 'left the queue')
('front of queue is:', 'vez')
It was a queue
```

تجسم مثال بالا:



تمرين

به پیاده سازی بالا از صفت این قابلیت را اضافه کنید که که عنصر α ام صفت را خروجی دهد.

پشته

در بعضی موارد عناصر تنها از یک طرف اضافه می شوند و از همان طرف نیز خارج می شوند. به عنوان مثال ظرف های کشیفی را در نظر بگیرید که بر روی هم قرار دارند و قصد شستن آن ها را دارید در هر مرحله اگر بخواهید ظرفی را بردارید بالاترین آن ها را انتخاب می کنید و آن را می شویید و اگر ظرف کشیفی را بخواهید به آن ها اضافه کنید در بالای تمام آن ها قرار می دهید. در واقع در این موارد عنصری که دیرتر از همه اضافه شده است زودتر از همه خارج میگردد. به داده ساختاری که این گونه موارد را مدل می کند، پشته می گویند.

یک پشته باید از توابع زیر پشتیبانی کند:

- $push(x)$: x را به بالای پشته اضافه می کند.
- $pop()$: عنصر بالای پشته را حذف می کند و آن را بازمی گرداند.
- $top()$: عنصر بالای پشته را باز می گرداند.
- $size()$: تعداد عناصر موجود در پشته را باز می گرداند.
- $is_empty()$: خالی بودن پشته را مشخص می کند. پ
- $is_full()$: پر بودن پشته را مشخص می کند.

تمام اعمال در $O(1)$ انجام می شوند.

پیاده سازی پشته نیز به کمک آرایه یا لیست پیوندی انجام میشود.

پیاده سازی پشته به کمک آرایه

برای پیاده سازی پشته با آرایه از یک متغیر *num* استفاده می کنیم که همیشه به عنصر بالای پشته اشاره می کند.

In [3]:

```
class Stack:
    def __init__(self, max_size):
        self.max_size = max_size          # Size of stack
        self.S = [0] * max_size          # Stack array
        self.num = 0                      # Number of elements in Stack

    def push(self, item):
        if self.num >= self.max_size:
            raise Exception("Stack overflow")
        self.S[self.num] = item
        self.num += 1

    def pop(self):
        if self.num == 0:
            raise Exception("Stack empty")
        self.num -= 1
        return self.S[self.num]

    def top(self):
        if self.num == 0:
            raise Exception("Stack empty")
        return self.S[self.num-1]

    def size(self):
        return self.num

    def is_full(self):
        return self.num >= self.max_size

    def is_empty(self):
        return self.num == 0
```

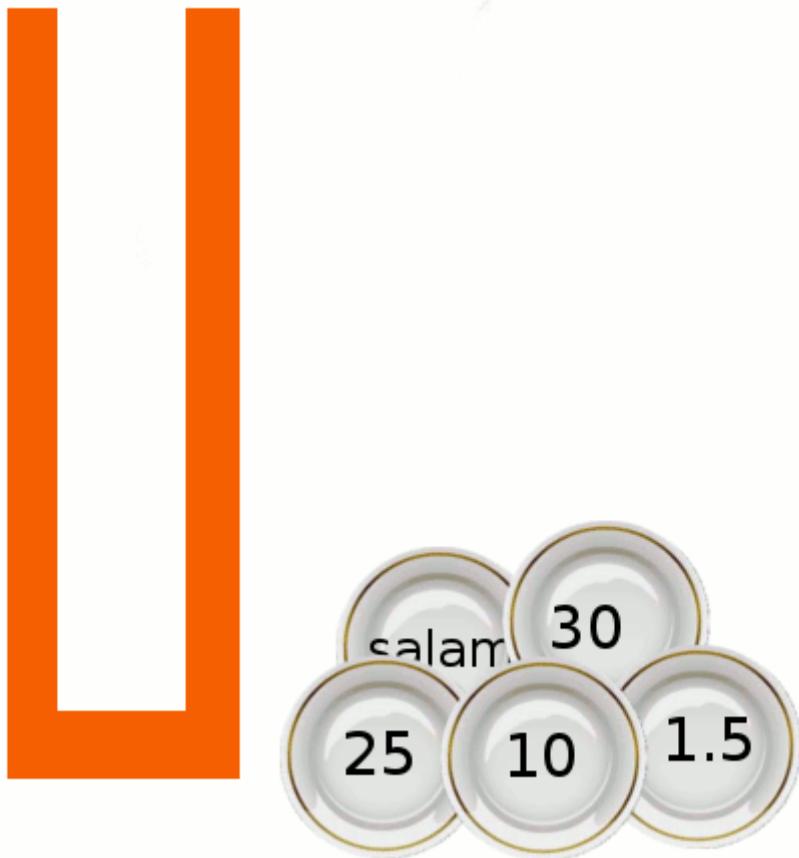
In [4]:

#Example

```
st=Stack(10) # (top of stack) []
st.push(10) # [10]
st.push(30) # [30, 10]
st.push(1.5) # [1.5, 30, 10]
print(st.pop()," is popped") # [30, 10]
st.push("salam") # ["salam", 30, 10]
st.pop() # [30, 10]
print("Top of stack: ",st.top())
st.pop() # [10]
st.pop() # []
print("Stack is empty: ",st.is_empty())
```

```
(1.5, ' is popped')
('Top of stack: ', 30)
('Stack is empty: ', True)
```

جسم مثال بالا:



صف دو طرفه

صف عادی محدودیت هایی برای مدلسازی دارد. برای مثال فرض کنید که داریم صف نانوایی را مدل میکنیم اما نانوا آشناهاش را در ابتدای صف قرار می دهد! 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 می کند و سپس خودش یواشکی! به آخر صف اضافه می شود). در واقع صف دو طرفه قابلیت‌های پشته و صف را به صورت همزمان پشتیبانی می کند.

یک صف دو طرفه باید از توابع زیر پشتیبانی کند:

- $push_front(x)$: عنصر x را در ابتدای صف درج می کند.
- $push_back(x)$: عنصر x را در انتهای صف درج می کند.
- $pop_front()$: عنصر ابتدای صف را حذف می کند و ان را باز می گرداند.
- $pop_back()$: عنصر انتهای صف را حذف و آن را باز می گرداند.
- $front()$: عنصر ابتدای صف را باز می گرداند.
- $back()$: عنصر انتهای صف را باز می گرداند.
- $size()$: تعداد عناصر موجود در صف را باز می گرداند.
- $is_empty()$: خالی بودن یا نبودن صف را مشخص می کند.
- $is_full()$: پر بودن یا نبودن صف را مشخص می کند.

تمامی اعمال در $O(1)$ انجام می‌شوند

پیاده سازی صف دو طرفه نیز به کمک آرایه یا لیست پیوندی انجام می‌شود.

پیاده سازی صف دو طرفه به کمک آرایه

پیاده سازی صفت دو طرفه با آرایه همانند پیاده سازی صفت عادی با استفاده از آرایه است. تنها تفاوت ها این است که هنگام اضافه شدن عنصر در ابتدای صفت باید مقدار متغیر *first* را یکی کاهش دهیم و *num* را یکی افزایش دهیم. در هنگام حذف عنصر از انتهای صفت باید تنها مقدار *num* را یکی کاهش دهیم.

In [5]:

```
class DoubleEndedQueue:
    def __init__(self, max_size):
        self.max_size = max_size
        self.Q = [0] * max_size
        self.num = 0
        self.first = 0

    def push_back(self, item): # Like one-way queue
        if self.num >= self.max_size:
            raise Exception("Queue overflow")
        self.Q[(self.num + self.first) % self.max_size] = item
        self.num += 1

    def push_front(self, item): # New
        if self.num >= self.max_size:
            raise Exception("Queue overflow")
        self.first = (self.first - 1) % self.max_size
        self.Q[self.first] = item
        self.num += 1

    def pop_front(self): # Like one-way queue
        if self.num == 0:
            raise Exception("Queue empty")
        item = self.Q[self.first]
        self.first = (self.first + 1) % self.max_size
        self.num -= 1
        return item

    def front(self): # Like one-way queue
        if self.num == 0:
            raise Exception("Queue empty")
        return self.Q[self.first]

    def pop_back(self): # New
        if self.num == 0:
            raise Exception("Queue empty")
        self.num -= 1
        return self.Q[(self.num + self.first) % self.max_size]

    def back(self): # New
        if self.num == 0:
            raise Exception("Queue empty")
        return self.Q[(self.num + self.first - 1) % self.max_size]

    def is_empty(self):
        return self.num == 0

    def size(self):
        return self.num

    def is_full(self):
        return self.num >= self.max_size
```

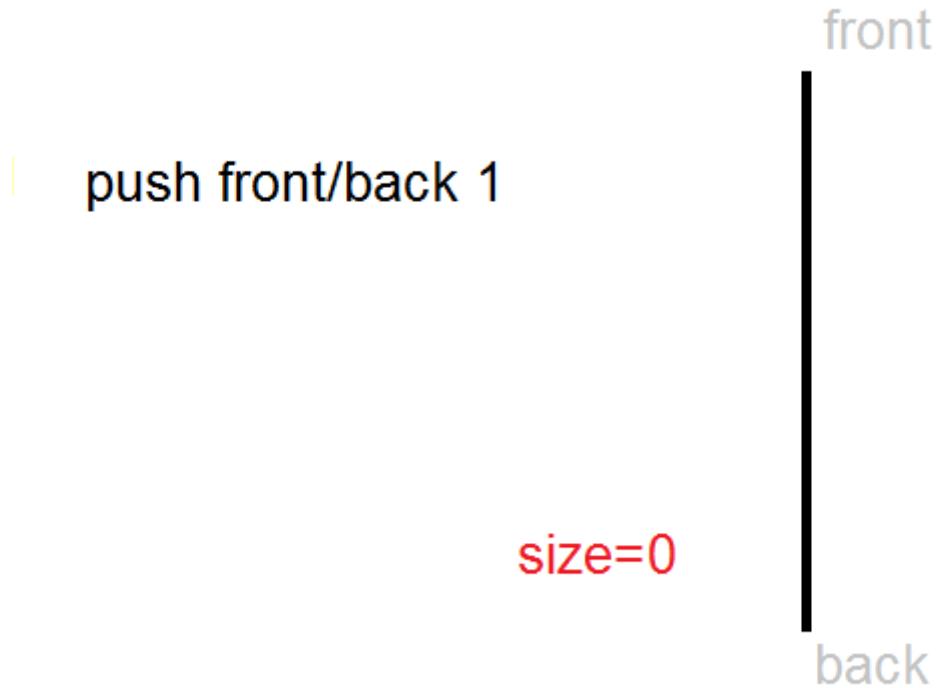
In [6]:

#Example

```
dq=DoubleEndedQueue(7) # (front of queue)[](back of queue)
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() # []
print("queue size is: ",dq.size())
```

```
('queue size is: ', 3)
('back of queue is: ', 0)
('front of queue is: ', 1)
('queue size is: ', 0)
```

جسم مثال بالا:



لیست پیوندی

لیست پیوندی داده ساختاری بسیار پایه‌ای برای نگهداری توالی هاست. در لیست پیوندی هر عنصر تنها از عنصر بعدی خود مطلع است. البته در لیست پیوندی دو طرفه 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 بازمی‌گرداند: $delete(x)$
- اولین عنصر با $data$ مساوی با val را باز می‌گرداند: $find(val)$
- عنصر ind ام لیست را باز می‌گرداند: $get(ind)$
- تعداد عناصر موجود در لیست را باز می‌گرداند: $size()$
- خالی بودن یا نبودن لیست را مشخص می‌کند: $is_empty()$

به جدول زیر توجه کنید:

آرایه	لیست پیوندی
درج در مکان مشخص و حذف	$O(1) + \text{search time}$
دسترسی به عنصر i	$O(n)$

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 دهند و تعداد آن‌ها از $O(n)$ است. اما برای درج یک عنصر مثل x در یک مکان مشخص از لیست پیوندی مانند درج در

جایگاه i ام، کافیست که عنصر بعدی $1 - i$ 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.

پیاده سازی لیست به کمک کلاس یک کلاس برای *node* های لیستمان می گیریم و در آن برای هر *node* اطلاعات *next* را در *data* نگه می داریم. همچنین *node* بعد و قبل از آن را به ترتیب در *next* و *prev* آن قرار می دهیم. حال به کمک کلاس *list* که در آن *head* به عنوان *prev* آن قرار می دهیم. قبل از اولین عنصر و بعد از آخرین عنصر قرار دارد لیست پیوندی دوسویه را پیاده سازی می کنیم.

در واقع یک *node* با مقدار *head* *None* است که وظیفه اش نگه داشتن ابتدا و انتهایی صفت پیوندی ماست.

In [7]:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class List:
    def __init__(self):
        self.head = Node(None)
        self.head.next = self.head
        self.head.prev = self.head
        self.n = 0

    def get(self, ind):
        if ind >= self.size():
            raise Exception('Out of list')
        x = self.head.next
        for i in range(ind):
            x=x.next
        return x

    def insert_after(self, x, data):
        y = Node(data)
        self.n += 1
        y.prev = x
        y.next = x.next
        x.next = y
        y.next.prev = y
        return y

    def delete(self, x):
        if self.size() == 0:
            raise Exception('List is empty')
        self.n -= 1
        x.prev.next = x.next
        x.next.prev = x.prev
        return x

    def find(self, val):
        x = self.head.next
        for i in range(self.size()):
            if x.data == val:
                return x
            x=x.next
        return None

    def size(self):
        return self.n

    def is_empty(self):
        return self.n==0
```

In [8]:

#Example

```
list=List() # head
list.insert_after(list.head,"milad") # head <-> milad
list.insert_after(list.get(0),"Arya") # head <-> milad <-> Arya
list.insert_after(list.find("Arya").prev,"jabbar") # head <-> milad <-> jabbar <-> Arya
list.delete(list.find("jabbar")) # head <-> milad <-> Arya
print("Current size of list is",list.size())

('Current size of list is', 2)
```

جسم مثال بالا:



list = List()

Type *Markdown* and *LaTeX*: α^2

In []:

In []:

به نام خدا

دانشگاه آزاد اسلامی - مهندسی کامپیوتر

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل پنجم، بخش اول: ذخیره‌سازی و پیمایش درخت

فهرست محتویات

• مقدمه

• تعریف در درخت‌ها

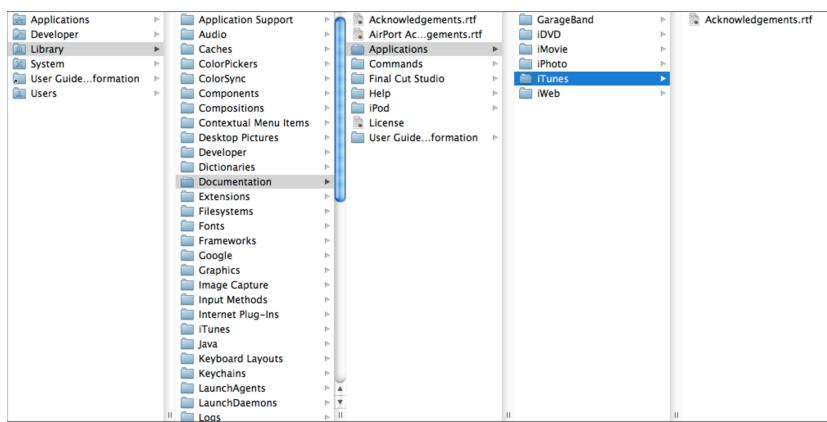
• ذخیره‌سازی درخت

• پیمایش درخت

مقدمه

آرایه‌ها، لیست‌های پیوندی، صفات و پشت‌های داده‌ساختارهایی 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. درخت‌ها بیانگر ساختارهایی سلسله مراتبی هستند که در بسیاری از مسائل به آن‌ها نیاز داریم. به طور مثال یکی از این

کاربردها ذخیره‌سازی ساختار پوشه‌ها در کامپیوتر است که از ساختاری سلسله مراتبی پیروی می‌کند.



همچنین می‌توان درخت‌ها را در ساختارهایی کارآمدتر برای انجام عمل‌های خاص روی ساختارهای خطی نیز به کار گرفت.

تعاریف در درخت‌ها

در زبان نظریه گراف‌ها درخت، گراف همبند بدون دور است و درخت ریشه‌دار درختی است که در آن یک رأس خاص به عنوان ریشه انتخاب شده است و ساختاری سلسله مراتبی به آن بخشیده است. موارد زیر را برای درخت‌های ریشه‌دار تعریف می‌کنیم: (اغلب اسامی به تقلید از درخت‌های تبارشناصی انتخاب شده‌اند).

- عمق یک رأس: فاصله‌ی رأس تا ریشه را عمق آن می‌نامیم.
- ارتفاع درخت: ماکسیمم عمق در بین همه‌ی رأس‌ها را ارتفاع درخت می‌نامیم.
- پدر و فرزند: وقتی دو رأس مجاور باشند، رأسی که به ریشه نزدیک‌تر است را پدر دیگری می‌نامیم و رأس دورتر را فرزند رأس نزدیک‌تر.
- اجداد: اجداد یک رأس همه‌ی رئوسی هستند که روی مسیر آن رأس به ریشه قرار دارند. منظور از جد Λ یک رأس جدی است که فاصله‌اش تا آن رأس Λ است.
- نوادگان: نوادگان یک رأس همه‌ی رئوسی هستند که این رأس جدشان است. زیردرخت یک رأس مجموعه‌ی همه‌ی نوادگان آن رأس است.

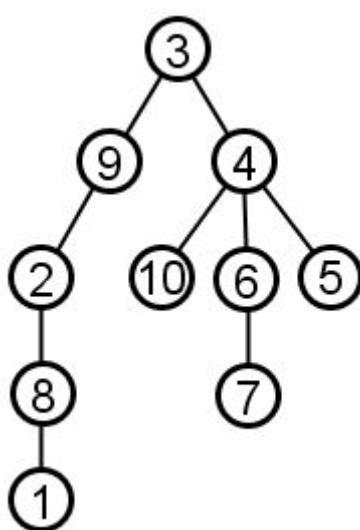
- برگ: رأسی که هیچ فرزندی نداشته باشد (این اصطلاح از درختان واقعی گرفته شده است).

ذخیره سازی درخت

راههای گوناگونی برای ذخیره سازی یک درخت وجود دارد. ساده‌ترین راه برای ذخیره سازی یک درخت آن است که رأس‌های آن را شماره زده 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.

ذخیره کردن پدرهای رئوس گرچه برای تعیین یکتایی یک درخت کافی است، اما پیمایش آن را بسیار دشوار و زمان‌بر می‌کند. راهکار بهتر برای ذخیره سازی این است که برای هر رأس شماره‌ی پدرش و یک لیست از شماره‌های فرزندانش را ذخیره کنیم.

برای مثال یک ذخیره سازی 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 راس بدست آمده است

In [1]:

```
vertices = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
parent = {
    3: 3,
    9: 3,
    4: 3,
    2: 9,
    10: 4,
    6: 4,
    5: 4,
    7: 6,
    8: 2,
    1: 8
}

# Calculations to create children lists of vertices
children = dict()
for vertex in vertices:
    children[vertex] = list()
for vertex in vertices:
    if vertex != parent[vertex]:
        children[parent[vertex]].append(vertex)

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. برای استفاده‌ی

بهینه‌تر از حافظه می‌توان برای هر رأس تنها ۳ اشاره‌گر انتخاب کرد:

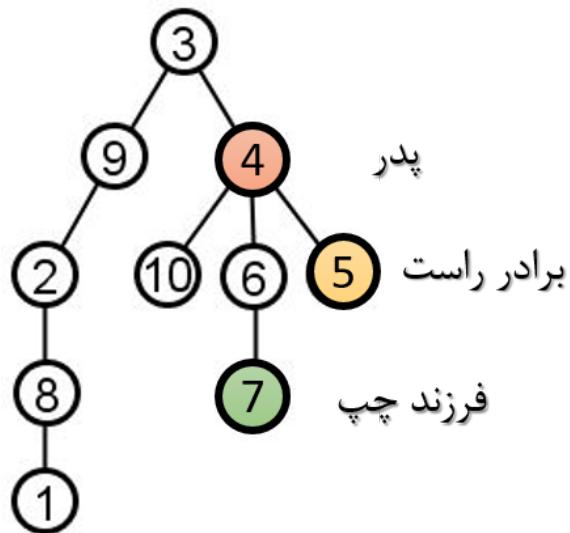
- left_child که به چپ‌ترین بچه‌ی یک رأس اشاره می‌کند

- right_sibling که به رأس بعدی در سمت راست خود اشاره می‌کند(در واقع به

- برادر سمت راست این راس)

- parent که به پدر خود اشاره می‌کند

در شکل زیر این سه اشاره گر برای راس شماره ۶ مشخص شده‌اند.



در قطعه کد زیر، پیاده‌سازی یک راس با استفاده از تعاریف فوق در قالب کلاس TreeNode آمده است.

In [2]:

```

class TreeNode:
    def __init__(self, label):
        self.parent = None
        self.left_child = None
        self.right_sibling = None
        self.label = label

    def __str__(self):
        return "TreeNode(%s)" % str(self.label)

```

در قطعه کد زیر، درختی با استفاده از رأس‌های بالا (کلاس TreeNode)، تعریف می‌کنیم.

In [3]:

```
class Tree:
    def __init__(self):
        self.root = None

    def assign_root(self, label):
        assert self.root is None
        self.root = TreeNode(label)

    def is_empty(self):
        return self.root is None

    def add_new_node_1(self, parent, label):
        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):
        new_node = TreeNode(label)
        new_node.parent = parent
        if parent.left_child is None:
            parent.left_child = new_node
        else:
            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):
        return self.add_new_node_2(parent, label)

    def find_in_subtree(self, label, node):
        if node.label == label:
            return node

        child = node.left_child
        while child is not None:
            result = self.find_in_subtree(label, child)
            if result is not None:
                return result
            child = child.right_sibling
        return None

    def find_by_label(self, label):
        if self.is_empty():
            return None
        return self.find_in_subtree(label, self.root)

    def add_new_node_by_label(self, parent_label, label):
        self.add_new_node(self.find_by_label(parent_label), label)

    def get_subtree_size(self, node):
        if node is None:
            return 0

        count = 1
```

```

child = node.left_child
while child is not None:
    count += self.get_subtree_size(child)
    child = child.right_sibling

return count

def get_size(self):
    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_۱` و `add_new_node_۲`). فرق این دو پیاده‌سازی مختلف در چیست؟
- تابع `find_in_subtree` چگونه عمل می‌کند؟
- چگونه می‌توان اندازه زیر درخت فرزند چپ ریشه را به دست آورد؟

In [7]:

```
tree = Tree()
tree.assign_root(3)
tree.add_new_node_by_label(3, 9)
tree.add_new_node_by_label(9, 2)
tree.add_new_node_by_label(3, 4)
tree.add_new_node_by_label(4, 10)
tree.add_new_node_by_label(4, 6)
tree.add_new_node_by_label(4, 5)
tree.add_new_node_by_label(2, 8)
tree.add_new_node_by_label(6, 7)
tree.add_new_node_by_label(8, 1)

print(tree.find_by_label(2))
print(tree.get_size())
```

TreeNode(2)

10

پیمایش درخت

فرض کنید که یک درخت ریشه‌دار داریم و می‌خواهیم با حرکت روی الگوهای درخت، همه‌ی گرهای آن را هر کدام یک‌بار ملاقات کنیم. به این کار پیمایش درخت می‌گویند. پیمایش درخت‌ها و ترتیب حاصل از نوع پیمایش در بسیاری از مسائل کاربرد دارند. 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 از چپ به راست، پیمایش می‌شود. حاصل پیمایش *preorder* درخت آمده در شکل ابتدای محتوا، پس از اجرای کد زیر قابل مشاهده است.

In [5]:

```
def pre_order(tree, node):
    order_list = list()
    if node is None:
        return order_list

    order_list.append(node.label)
    child = node.left_child
    while child is not None:
        order_list.extend(tree.pre_order(child))
        child = child.right_sibling

    return order_list

Tree.pre_order = pre_order

print(tree.pre_order(tree.root))
```

[3, 9, 2, 8, 1, 4, 10, 6, 7, 5]

نوع دوم پیمایش، پیمایش پس‌ترتیب یا *postorder* است که در آن ابتدا زیر درخت فرزندان به ترتیب از چپ به راست به صورت بازگشتی پیمایش می‌شود و در نهایت ریشه درخت پیمایش می‌شود.

پیمایش *postorder* درخت آمده در شکل ابتدای محتوا، پس از اجرای کد زیر قابل مشاهده است.

In [3]:

```
def post_order(tree, node):
    order_list = list()
    if node is None:
        return order_list

    child = node.left_child
    while child is not None:
        order_list.extend(tree.post_order(child))
        child = child.right_sibling
    order_list.append(node.label)

    return order_list

Tree.post_order = post_order

print(tree.post_order(tree.root))
```

```
NameError                                 Traceback (most recent call last)
<ipython-input-3-9a8552003a0a> in <module>()
      12     return order_list
      13
--> 14 Tree.post_order = post_order
      15
      16 print(tree.post_order(tree.root))

NameError: name 'Tree' is not defined
```

سومین نوع پیمایش، پیمایش میان ترتیب یا *inorder* است که در آن ابتدا زیر درخت فرزند چپ، پس از آن ریشه و در نهایت زیر درخت بقیه فرزندان به ترتیب از چپ به راست به شکل بازگشتی پیمایش می‌شود.

پیمایش *inorder* درخت آمده در شکل ابتدای محتوا، پس از اجرای کد زیر قابل مشاهده است.

In [8]:

```
def in_order(tree, node):
    order_list = list()
    if node is None:
        return order_list

    child = node.left_child
    order_list.extend(tree.in_order(child))
    order_list.append(node.label)
    while child is not None:
        child = child.right_sibling
        order_list.extend(tree.in_order(child))

    return order_list

Tree.in_order = in_order

print(tree.in_order(tree.root))
```

[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 یک گراف، اندازه‌ی زیر درخت فرزند چپ ریشه را برگرداند.

In [2]:

```
from src.tests.tester import tester

def root_left_subtree_size(preorder_list, inorder_list):
    return 0
    #return number_of_node_in_left_subtree_of_root

tester("root_left_subtree_size", root_left_subtree_size)

#example:
#function_call: root_left_subtree_size([3, 9, 2, 8, 1, 4, 10, 6, 7, 5], [1, 8, 2, 9, 3, 10,
#return value: 4
```

Your code get wrong answer in 10 test(s) from 10 test(s).

خودآزمایی ۲

تابع `left_subtree_size` را طوری کامل کنید که با گرفتن لیست پیمایش شده‌ی یک گراف و همچنین لیبل یک راس آن، اندازه‌ی زیر درخت فرزند چپ راس با آن لیبل را برگرداند.

In [5]:

```
def left_subtree_size(preorder_list, inorder_list, node_label):
    return 0
#return number_of_node_in_left_subtree_of_root

tester("left_subtree_size", left_subtree_size)
#example:
#function_call: left_subtree_size([3, 9, 2, 8, 1, 4, 10, 6, 7, 5], [1, 8, 2, 9, 3, 10, 4, 7
#return value: 2
```

Your code get wrong answer in 6 test(s) from 10 test(s).

خودآزمایی ۳

تابع *change_order* را طوری کامل کنید که با گرفتن لیست پیمایش شده‌ی یک گراف، لیست پیمایش *postorder* آن را برگرداند. در این سوال فرض شده است که هر راس درخت حداکثر دو فرزند دارد! (چرا این فرض برای حل سوال لازم است؟) ***توجه کنید که ترتیب *postorder* مثال آورده شده در کد زیر با ترتیب *postorder* ای که برای شکل بالا به دست آوردیم متفاوت شد. چون در آن درجه‌ی بعضی رئوس بیشتر از ۲ است.

In [4]:

```
def change_order(preorder_list, inorder_list):
    return []
#return number_of_node_in_left_subtree_of_root

tester("change_order", change_order)
#example:
#function_call: left_subtree_size([3, 9, 2, 8, 1, 4, 10, 6, 7, 5], [1, 8, 2, 9, 3, 10, 4, 7
#return value: [1, 8, 2, 9, 10, 7, 5, 6, 4, 3]
```

Your code get wrong answer in 10 test(s) from 10 test(s).

In []:

بنام خدا

دانشگاه آزاد اسلامی شیراز - مهندسی کامپیوتر

داده‌ساختارها و الگوریتم‌ها

ترم دوم سال تحصیلی 1400-1401

فصل پنجم، بخش دوم: درخت دودویی جست‌وجو

فهرست محتویات ¶

فهرست محتویات

مقدمه ■

جستجو ■

درج ■

حذف ■

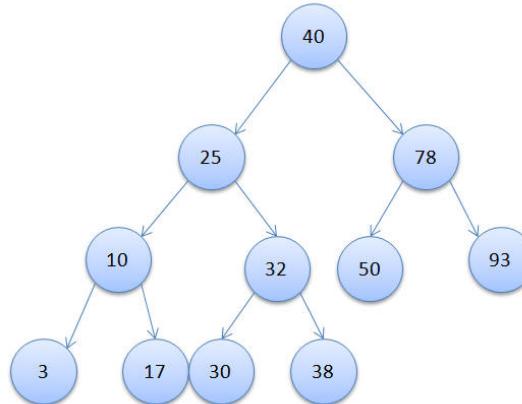
میانگین ارتفاع درخت جست‌وجو ■

پایین‌تیرین حد مشترک ■

مقدمه

درخت دودویی جست‌وجو (به انگلیسی: Binary Search Tree) یا به اختصار د.د.ج) درختی ریشه‌دار و دودویی است که به ازای هر راس مانند V ، مقادیر تمامی راس‌های زیر درخت بچه‌ی

سمت چپ آن از مقدار راس ۷ کوچک‌تر و مقادیر تمامی راس‌های زیردرخت بچه‌ی سمت راستش از ۷ بزرگ‌تر است.



هر راس درون داده‌ج دارای یک برچسب، اشاره‌گر به بچه چپ، اشاره‌گر به بچه راست و اشاره‌گر به پدرش است.

به غیر از راس ریشه، بقیه راس‌ها حتماً پدر خواهند داشت.

In [2]:

```
class Node:
    def __init__(self, label, parent):
        self.label = label
        self.parent = parent
        self.leftChild = None
        self.rightChild = None

    def __str__(self): # Returns the path from root to node
        if self.parent:
            return str(self.parent) + " " + str(self.label)
        else:
            return str(self.label)
```

برای کار با این ساختمان داده، ۳ عمل متصور هستیم:

- جستجو
- درج
- حذف

(می‌توانید در اینجا (<https://visualgo.net/bst>) عملیات‌های گفته شده را به صورت تعاملی مشاهده کنید).

جستجو

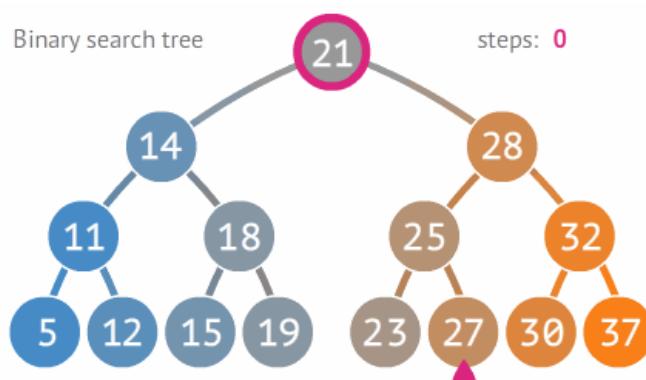
برای جستجو یک عنصر در د.د.ج، با شروع از ریشه، به ازای هر راسی که درون آن هستیم،

مقداری که می‌خواهیم پیدا کنیم را با مقدار راس 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 راسی که داخلش هستیم، جستجو را در زیردرخت سمت

راست یا چپ این راس ادامه می‌دهیم.



In [3]:

```
def search(node, label):
    if node is None or node.label == label:
        return node
    elif node.label > label:
        return search(node.leftChild, label)
    else:
        return search(node.rightChild, label)
```

در صورتی که جستجو موفقیت آمیز باشد، تابع به ما راس با مقدار مورد نظر را برمی‌گرداند اما در صورتی که جستجو موفقیت آمیز نباشد، `None` برگردانده می‌شود.

اگر از ریشه شروع کنیم و به سمت بچه‌ی چپ هر راس حرکت کنیم تا به راسی مانند V برسیم که بچه‌ی چپ نداشته باشد، راس V مقدار کمینه را در بین راس‌های د.ج خواهد داشت. به طور مشابه اگر به جای حرکت به سمت بچه‌ی چپ، به سمت بچه‌ی راست حرکت کنیم راس بیشینه را خواهیم داشت.

In [4]:

```
def findMin(node):
    if node.leftChild is None:
        return node
    else:
        return findMin(node.leftChild)

def findMax(node):
    if node.rightChild is None:
        return node
    else:
        return findMax(node.rightChild)
```

درج

برای درج یک عنصر، رویه بازگشتی زیر را در نظر می‌گیریم: با شروع از ریشه، راس V که در حال حاضر در آن هستیم را در نظر می‌گیریم. مقداری که

می‌خواهیم درج کنیم را با برچسب راس V مقایسه می‌کنیم. اگر از برچسب راس V کوچک‌تر بود، پس جایش در زیردرخت سمت چپ V است و اگر مقدارش از برچسب راس 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 باید به زیردرخت بچه سمت راست V برویم ولی V بچه‌ی سمت راست ندارد، در این صورت یک راس با مقدار مورد نظرمان ایجاد می‌کنیم و آن را بچه‌ی سمت راست V قرار می‌دهیم.

www.penjee.com

In [5]:

```
def insert(node, value):
    if (value < node.label):
        if (node.leftChild != None):
            insert(node.leftChild, value)
        else:
            node.leftChild = Node(value, node)
    else:
        if (node.rightChild != None):
            insert(node.rightChild, value)
        else:
            node.rightChild = Node(value, node)
```

نکته: اگر د.د.ج را به صورت میان‌ترتیب پیمایش کنیم، اعداد را به صورت مرتب شده بدست می‌آوریم. چرا؟

In [6]:

```
def inOrderPrint(node):
    if node is None:
        return
    inOrderPrint(node.leftChild)
    print(node.label)
    inOrderPrint(node.rightChild)
root = Node(5, None)
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)
print ("Searching value 8 in tree:", str(search(root, 8)))
print ("Path to the min node in tree:", findMin(root))
print ("Path to the max node in tree:", findMax(root))
```

InOrder print of the tree:

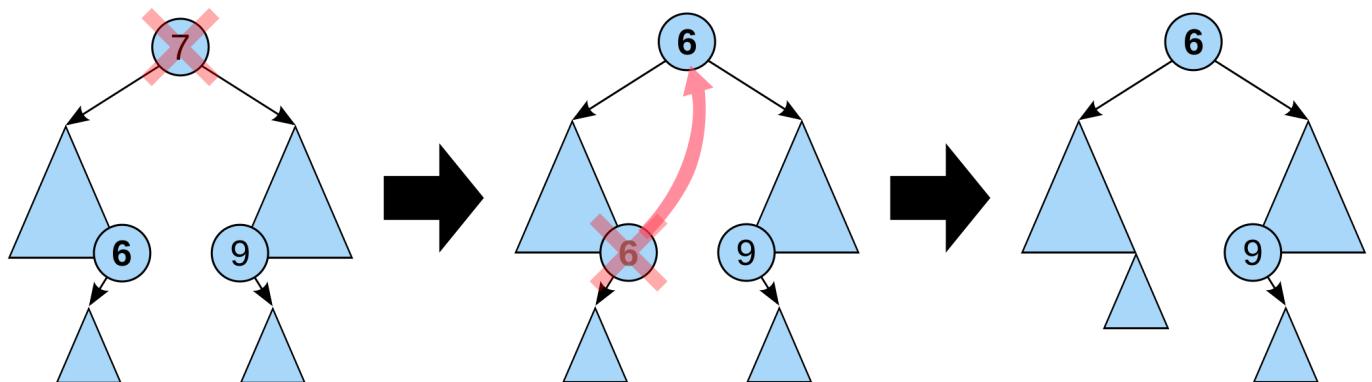
```
1
2
3
4
5
6
7
8
9
10
('Searching value 8 in tree:', '5 9 8')
('Path to the min node in tree:', <__main__.Node instance at 0x10bc78b48>)
('Path to the max node in tree:', <__main__.Node instance at 0x10bc78488>)
```

حذف

برای حذف یک مقدار از درخت، ابتدا راس V دارای این مقدار را پیدا می‌کنیم. در این مرحله ۳
حالت پیش می‌آید:

- راس V بچه نداشته باشد: آن را حذف می‌کنیم.
- راس V یک بچه داشته باشد: V را حذف می‌کنیم و بچه‌اش را جایش قرار می‌دهیم.

- راس ۷ دو بچه داشته باشد: در این صورت راس مینیمم در زیر درخت سمت راست ۷ یا راس ماکزیمم در زیردرخت سمت چپ ۷ را، به نام u را پیدا می‌کنیم. دقت کنید هر کدام از این راس‌ها را که به جای ۷ قرار دهیم و ۷ را حذف کنیم، هنوز شرط لازم برای د.د.ج بودن درخت برقرار است. برچسب راس u را در ۷ قرار می‌دهیم و u را حذف می‌کنیم. از آنجا که u راس مینیمم یا ماکزیمم یک زیردرخت است، حداکثر یک بچه خواهد داشت و می‌شود طبق ۲ حالت اول این راس را حذف کرد.



In [7]:

```
def replaceNodeInParent(oldNode, newNode):
    if root == oldNode:
        root = newNode
    if oldNode.parent: #oldNode is not root
        if oldNode == oldNode.parent.leftChild:
            oldNode.parent.leftChild = newNode
        else:
            oldNode.parent.rightChild = newNode

    if newNode:
        newNode.parent = oldNode.parent

def delete(node, value):
    if node is None:
        return

    if node.label < value:
        delete(node.rightChild, value)
    elif node.label > value:
        delete(node.leftChild, value)
    else: # node to be deleted is found
        if node.leftChild and node.rightChild: #has 2 children
            newNode = findMin(node.rightChild)
            node.label = newNode.label
            delete(newNode, newNode.label) #delete the new node
        elif node.leftChild:
            replaceNodeInParent(node, node.leftChild)
        else:
            replaceNodeInParent(node, node.rightChild)
            # This works when node.rightChild is also None
```

In [7]:

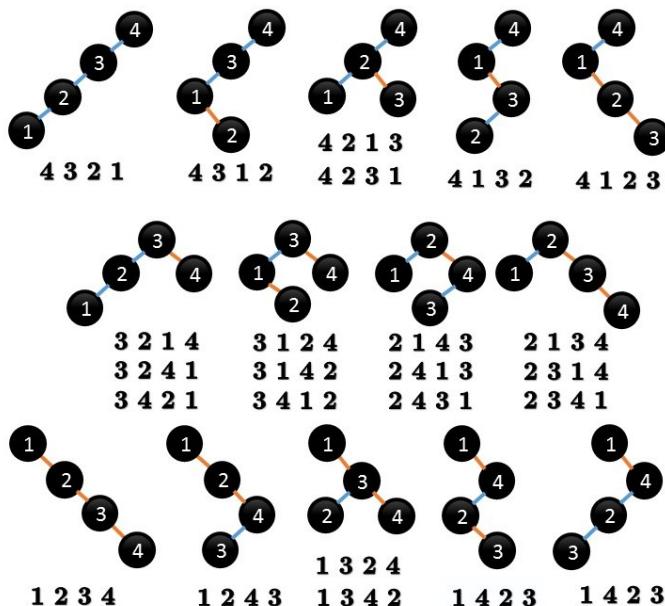
```
root = Node(2, None)
insert(root, 1)
insert(root, 3)
inOrderPrint(root)
delete(root, 2)
inOrderPrint(root)
```

```
1
2
3
1
3
```

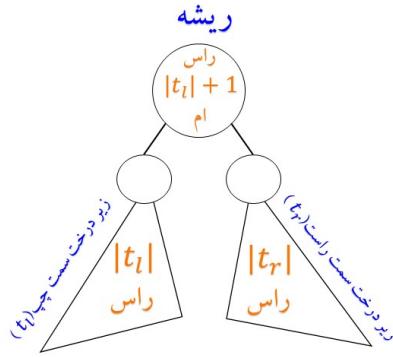
میانگین ارتفاع درخت جستجو

در اینجا به دادن شهود کلی بسنده می‌کنیم. (اثبات ریاضی دقیق آن را می‌توانید در کتاب دکتر قدسی مشاهده فرمایید.)

بدین منظور، تمامی جایگشت‌هایی ممکن را که می‌خواهیم در د.د.ج درج کنیم را در نظر می‌گیرم. توجه کنید عملیات درج هر جایگشتی هنگامی که به ترتیب و از ابتدای آن صورت بگیرد به دقیقاً یک د.د.ج یکتا می‌انجامد، به عنوان مثال زیر تمامی د.د.ج‌های حاصل از درج اعداد ۱ تا ۴ را نشان می‌دهد:



همان طور که مشاهده می‌کنید، هر چه قدر د.د.ج 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 د.د.ج t را با p_t نشان می‌دهیم، و همچنین زیر درخت 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 د.د.ج متعادل باشد، جزء انتخاب آن بسیار بزرگ‌تر از حالتی خواهد بود که این دو از هم فاصله داشته باشند. بنابراین هر چه درخت متعادل‌تر باشد، تعداد جایشگت بیشتری با آن متناظر خواهد شد.

پس به صورت شهودی در حالت میانگین، با یک درخت متعادل، که مرتبه پیچیدگی ارتفاع آن از $\lg n$ است مواجه خواهیم شد

در نمودار زیر هم که حاصل محاسبه میانیگن ارتفاع درخت برای حالت‌های مختلف است می‌توانید ببینید که این مقدار عددی بین $2^{\lg n}$ و $3^{\lg n}$ است!

In [8]:

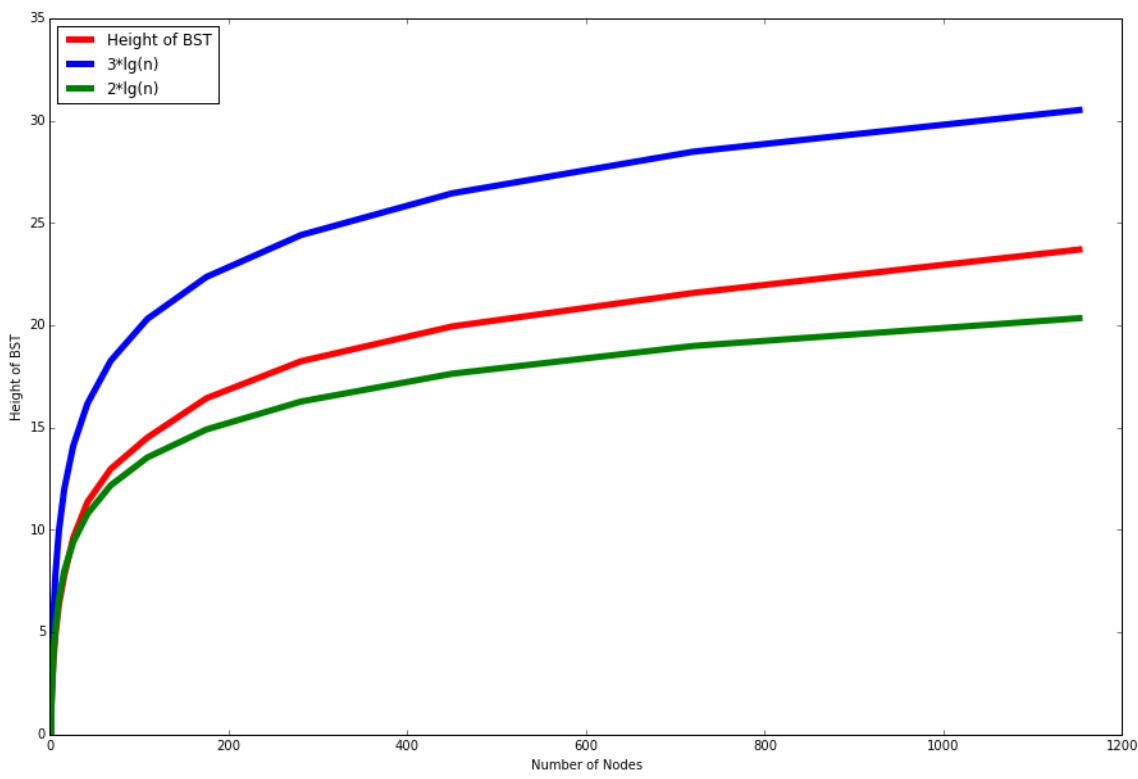
```
%matplotlib inline
from matplotlib.pyplot import *
import random

def find_Height(node, height):
    if(node == None):
        return height
    return max(find_Height(node.leftChild,height+1.0),find_Height(node.rightChild,height+1.0))

def get_avg_height(a):
    max_Sample_Size = 160
    avg = 0.0;
    for i in range(max_Sample_Size):
        numbers = [[random.randrange(16000) for k in range(j)] for j in range(a)]
        root = Node(numbers[0],None)
        for j in range(1,len(numbers)):
            insert(root,numbers[j])
        avg = avg + find_Height(root,0.0)
    return avg/float(max_Sample_Size)

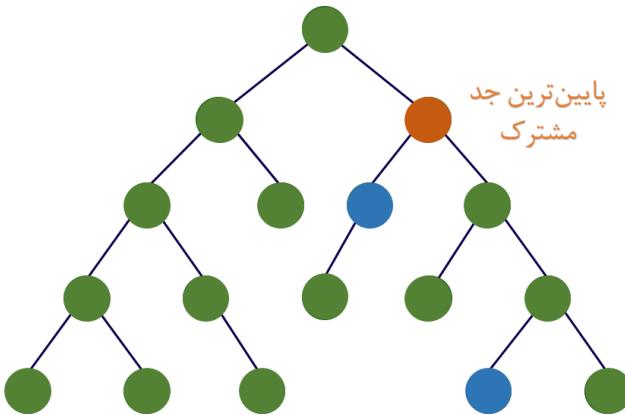
import math

def plot2(N):
    figure(figsize=(15, 10))
    xlabel("Number of Nodes")
    ylabel("Height of BST")
    y = [get_avg_height(a) for a in N]
    plot(N, y, 'r', label='Height of BST', linewidth=5)
    y = [3*math.log(a,2) for a in N]
    plot(N, y, 'b', label='3*lg(n)', linewidth=5)
    y = [2*math.log(a,2) for a in N]
    plot(N, y, 'g', label='2*lg(n)', linewidth=5)
    legend(loc=2)
N = np.power(1.6,range(1,16)).astype(int)
plot2(N)
```



پایین ترین جد مشترک

در یک درخت ریشه‌دار پایین ترین جد مشترک (LCA) دو گره را دورترین راس از ریشه تعریف می‌کنیم که جد هر دو گره باشد.



در اینجا به بررسی الگوریتم حل این مسئله برای یک د.د.ج می‌پردازیم.

همان‌طور که می‌دانیم تمامی گره‌ها در د.ج 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:

از ریشه شروع کرده و مقدار آن را با دو عدد داده شده مقایسه می‌کنیم. اگر از هردو آن‌ها کوچک‌تر بود به سراغ بچه سمت راست آن می‌رویم و اگر از هردوی آن‌ها بزرگ‌تر بود به سراغ بچه سمت چپ آن می‌رویم. این الگوریتم بازگشتی را آنقدر ادامه می‌دهیم تا به گره‌ای بررسیم که مقدار آن بین دو عدد داده شده مسئله باشد.

In [9]:

```
def lca(root, x, y):
    if root.label > x and root.label > y:
        lca(root.leftChild, x, y)
    elif root.label < x and root.label < y:
        lca(root.rightChild, x, y)
    else:
        print(root.label)
```

In [10]:

```
root = None
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)
lca(root, 29, 72)
```

به نام خدا

دانشگاه آزاد اسلامی شیراز - مهندسی کامپیوتر

داده‌ساختار‌ها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل پنجم، بخش چهارم: هرم

فهرست محتویات

• مقدمه

• معرفی هرم

• عملیات‌های هرم

• پیاده‌سازی هرم

• روش‌های ساختن هرم

• کاربردهای هرم

مقدمه

یکی از مهمترین داده ساختارهایی که در این درس فرامی‌گیریم، هرم است. در این نوشه ابتدا این داده ساختار را معرفی می‌کنیم، سپس با پیاده‌سازی‌ها و کاربردهایش آشنا خواهیم شد.

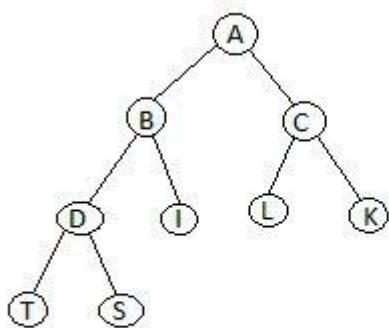
معرفی هرم

اورژانس یک بیمارستان را در نظر بگیرید که همیشه میخواهد به اورژانسی ترین بیمار رسیدگی کند. برای این کار باید بتواند به صورت سریع اورژانسی ترین بیمار را شناسایی کند. همچنین باید بتواند افرادی جدیدی که وارد می‌شوند را سریعاً در محل خود در صف قرار دهد.

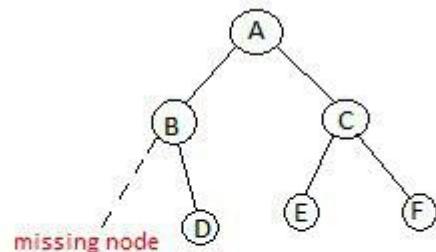
تمرین ۱:

کار خواسته شده را با آرایه مرتب شده انجام دهید و پیچیدگی زمانی عملیاتهای گفته شده را بیابید. (جواب در [جدول ۱](#) آمده است)

هرم داده ساختاری است که امکانات گفته شده در بالا را برای ما فراهم می‌کند. هرم یک درخت ریشه‌دار کامل است که در آن رئوس به گونه‌ای قرار گرفته اند که ارزش هر راس از ارزش بچه‌هایش بیشتر (یا کمتر) 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 گرفته اند.



Complete Binary Tree



In-Complete Binary Tree

شکل ۱: تفاوت درخت کامل و ناکامل

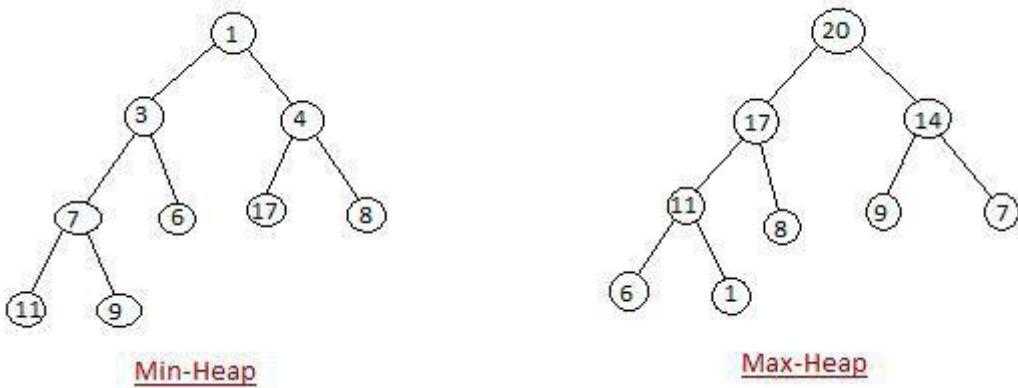
تذکر:

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 و هرم فیبوناچی را بررسی کنید.
با توجه به این که هرم درختی دودویی و کامل است، میتوان نتیجه گرفت که ارتفاع آن همیشه برابر $\log n$ است.

هرمهای بر حسب رابطه بین رئوس با فرزندانشان دو نوع هستند:

- هرم بیشینه: ارزش هر راس از ارزش بچه‌هایش بیشتر است.
- هرم کمینه: ارزش هر راس از ارزش بچه‌هایش کمتر است.



شکل ۲: مثال هرم کمینه و بیشینه

عملیات‌های هرم

هرم بیشینه از عملیات زیر پشتیبانی می‌کند (عملیات هرم کمینه مشابه است):

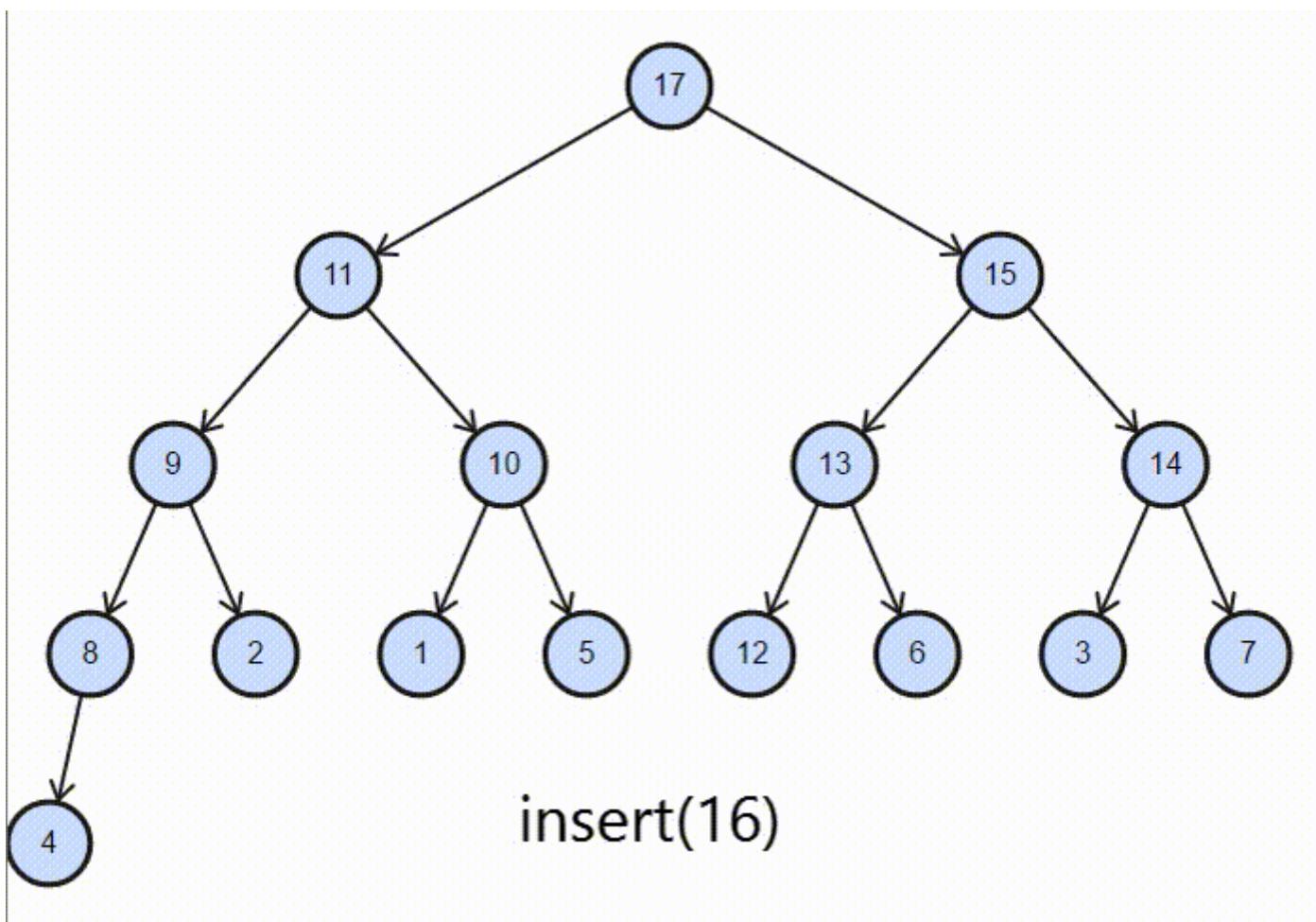
- درج
- پیدا کردن عضو بیشینه
- حذف عضو بیشینه

درج در هرم:

برای درج یک عضو در هرم، ابتدا آن را در اولین مکان خالی قرار می‌دهیم (با حفظ کامل بودن درخت) سپس با تعدادی عمل جابجایی (bubble-up) عضو جدید را به مکان درستش منتقل می‌کنیم.

در bubble-up عضو مورد نظر را با پدر خود مقایسه می‌کنیم و در صورتی که از آن بیشتر بود، جای آن دو را 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 حداکثر به اندازه ارتفاع درخت یا همان $\log n$ طول خواهد کشید. پس پیچیدگی درج از $O(\log n)$ است.



شکل ۳: درج در هرم

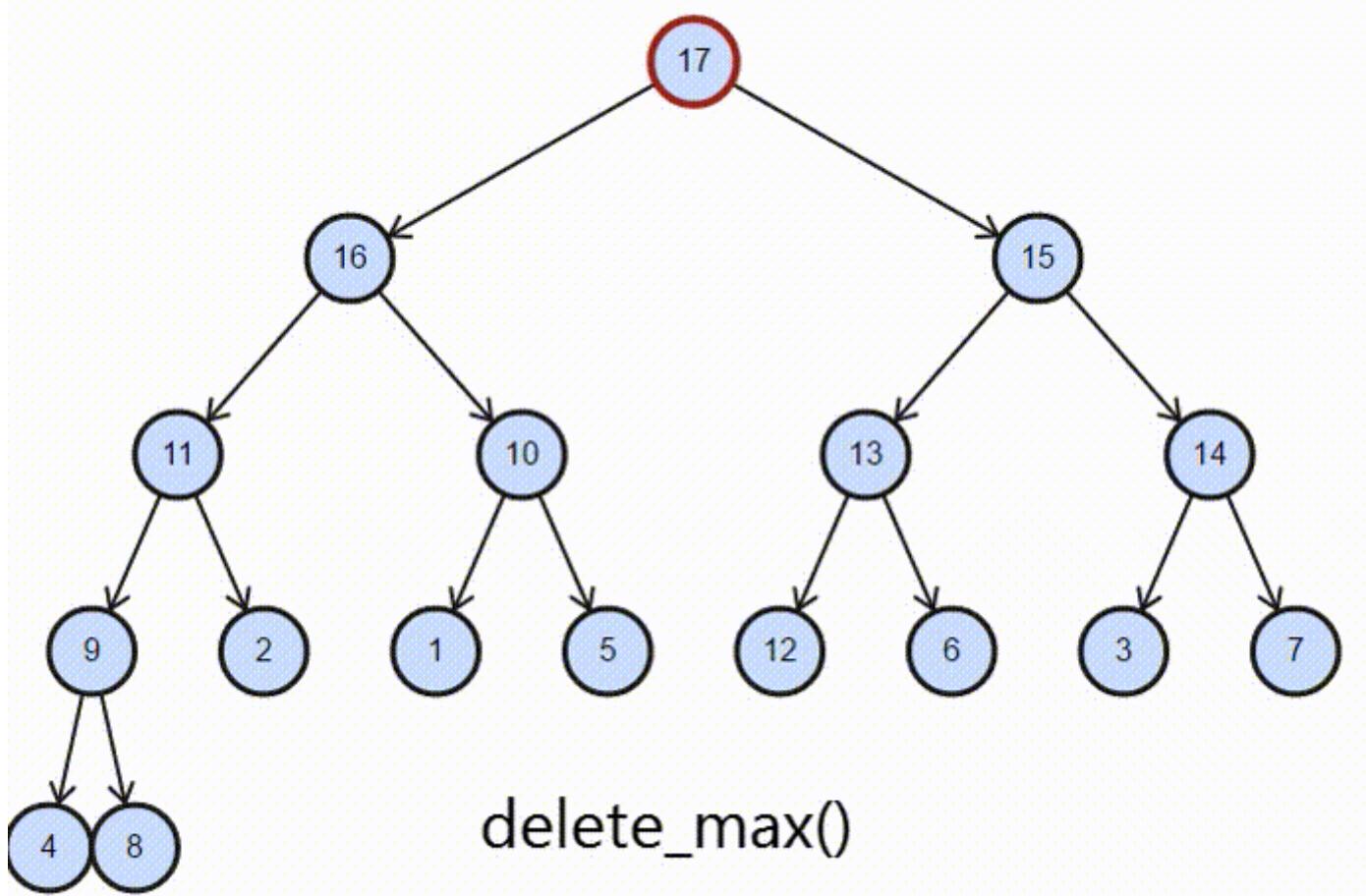
ریشه هرم همیشه عضو بیشینه را در خود نگه می‌دارد پس میتوان در $O(1)$ به آن دسترسی پیدا کرد.

حذف بیشینه در هرم:

برای حذف عضو بیشینه در هرم، ریشه را از درخت حذف میکنیم و آخرین عضو هرم (راست ترین ترین برگ در پایین ترین عمق) را در جایگاه ریشه قرار می‌دهیم. سپس با تعدادی عمل جابجایی (bubble-down) این عضو را به مکان درست منتقل می‌کنیم.

در bubble-down عضو مورد نظر را با بزرگترین فرزند خود مقایسه می‌کنیم و در صورتی که از آن کمتر بود، جای آن دو با هم عوض می‌کنیم. این کار را آنقدر ادامه می‌دهیم تا عضو مورد نظر از هر دو فرزند خود کوچکتر شود، یا آنکه خودش برگ درخت شود.

bubble-down چون هر عمل جابجایی عضو جدید را یک سطح در درخت پایین‌تر می‌برد، حداقل به اندازه ارتفاع درخت یا همان $\log n$ طول خواهد کشید. پس پیچیدگی حذف از $O(\log n)$ است.



شکل ۴: حذف ماکزیمم در هرم

جدول ۱ پیچیدگی زمانی اعمال هرم را در مقایسه با آرایه مرتب شده آورده است.

هرم	آرایه مرتب شده	عمل مرتب شده
$O(\log n)$	$O(n)$	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 ($i=1, 2, \dots, n$)، فرزندان چپ و راستش (در صورت وجود) به ترتیب در خانه‌های $2i+1$ و $2i$ قرار می‌گیرند. به وضوح پدر یک راس در خانه $\left\lfloor \frac{i}{2} \right\rfloor$ خواهد بود. با این روش، پیاده‌سازی هرم با حافظه $O(n)$ امکان‌پذیر است.

تذکر:

در بسیاری از پیاده‌سازی‌ها همچون پیاده‌سازی بالا، برای سهولت کار با اندیس‌ها، خانه صفر آرایه را خالی می‌گذارند. می‌توانید به عنوان تمرین، هرم را با آرایه‌ای که از خانه صفر پر می‌شود پیاده‌سازی کنید.

In [43]:

```
def parent(i):
    return i//2

def left_child(i):
    return 2*i

def right_child(i):
    return 2*i+1

class Max_Heap:
    def __init__(self):
        self.heap = [0] #the zero'th is redundant

    def size(self):
        return len(self.heap)-1

    def bubble_down(self,ind):
        while left_child(ind) <= self.size():
            newInd = ind
            if self.heap[left_child(ind)] > self.heap[ind]:
                newInd = left_child(ind)
            if right_child(ind) <= self.size() and self.heap[right_child(ind)] > self.heap[
                newInd]:
                newInd = right_child(ind)
            if ind == newInd:
                break
            self.heap[ind], self.heap[newInd] = self.heap[newInd], self.heap[ind]
            ind = newInd

    def bubble_up(self,ind):
        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 insert(self, item):
        self.heap.append(item)
        self.bubble_up(self.size())

    def get_max(self) :
        if self.size() == 0 :
            raise Exception("Heap is empty")
        return self.heap[1]

    def del_max(self) :
        if self.size() == 0 :
            raise Exception("Heap is empty")
        MAX = self.heap[1]
        self.heap[1] = self.heap[-1]
        del(self.heap[-1])
        self.bubble_down(1)
        return MAX

    def build_heap_with_bubble_up(self, L):
        self.heap = [0] + L
        for i in range(self.size()+1) :
            self.bubble_up(i)

    def build_heap_with_bubble_down(self,L):
        self.heap = [0] + L
        for i in range(self.size(),0,-1) :
```

```
    self.bubble_down(i)

def clear(self):
    self.heap=[0]
```

In [49]:

```
heap = Max_Heap()
heap.build_heap_with_bubble_down([1,10,9,4,7,11,2,3,6,5])
print(heap.heap)
print (heap.del_max())
heap.insert(8)
print(heap.heap)
print (heap.del_max())
print (heap.del_max())
print (heap.del_max())
print (heap.del_max())
print(heap.heap)
print (heap.size())
```

```
[0, 11, 10, 9, 6, 7, 1, 2, 3, 4, 5]
11
[0, 10, 8, 9, 6, 7, 1, 2, 3, 4, 5]
10
9
8
7
[0, 6, 4, 5, 3, 2, 1]
6
```

روش‌های ساختن 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:

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. دقیق کنید این روش مانند آن است که عناصر را یکی یکی در یک هرم خالی درج کنیم.

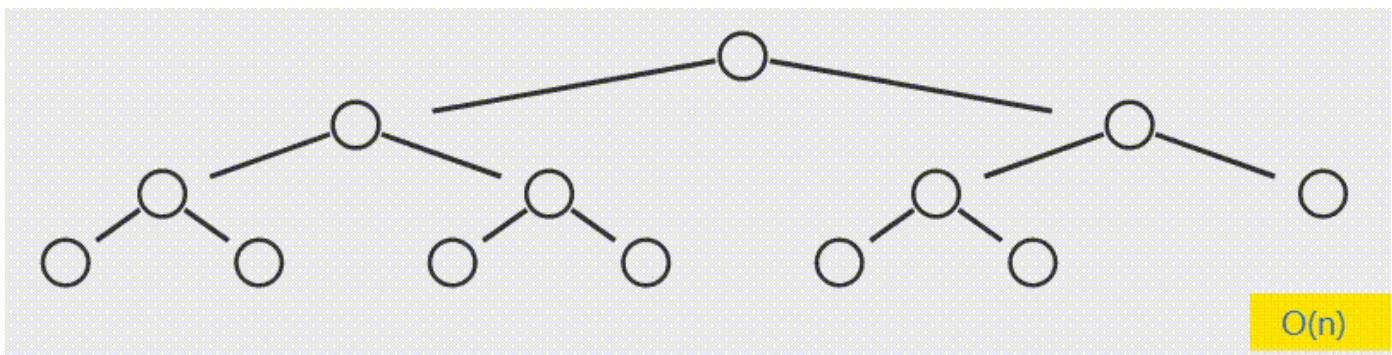
شکل ۵: ساخت هرم با bubble-up (اعداد قرمز اندیس عناصر هستند)

چون در هر bubble_up حداکثر به اندازه فاصله راس شروع تا ریشه، جابجایی انجام می‌شود، تعداد کل جابجایی‌ها در بدترین حالت برابر خواهد بود با

$$\begin{aligned}
\sum_{i=1}^{\log n} i \cdot 2^i &= 1 \times 2 + 2 \times 2^2 + \dots + \log n \times 2^{\log n} \\
&= 2 + 2^2 + 2^3 + \dots + 2^{\log n} \\
&\quad + 2^2 + 2^3 + \dots + 2^{\log n} \\
&\quad + 2^3 + \dots + 2^{\log n} \\
&\quad \dots \\
&\quad + 2^{\log n} \\
- 2) + (2^{\log n+1} - 2^2) + (2^{\log n+1} - 2^3) + \dots + (2^{\log n+1} - 2^{\log n}) \\
\cdot^{\log n+1} \cdot \log n - \sum_{i=1}^{\log n} 2^i &= 2n \cdot \log n - (2^{\log n+1} - 2) \in O(n \log n)
\end{aligned}$$

روش دوم:

با شروع از آخر لیست ، 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 راس شروع تا پایین ترین سطح درخت، عمل جابجایی انجام میشود. پس تعداد کل جابجایی ها حداکثر برابر خواهد بود با:

$$\sum_{i=1}^{\log n} (\log n - i) \cdot 2^i = \log n \sum_{i=1}^{\log n} 2^i - \sum_{i=1}^{\log n} i \cdot 2^i$$

$$\log n \cdot 2^{\log n + 1} - (\log n \cdot 2^{\log n + 1} - (2^{\log n + 1} - 2)) = 2n - 2 \in O(n)$$

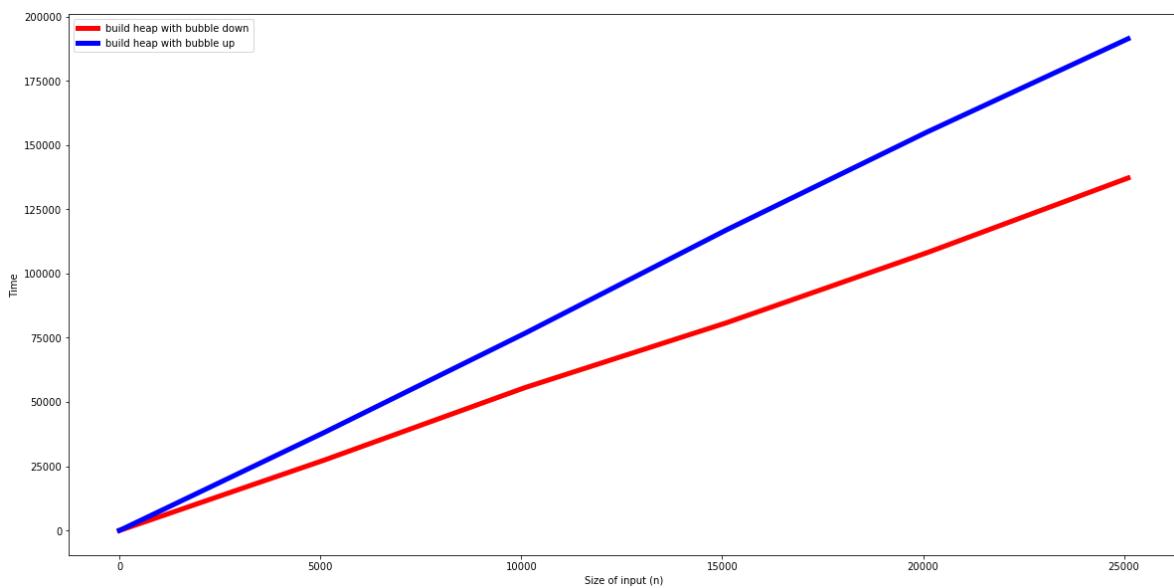
کد هر دو روش در قسمت قبل آمده است. در نمودار زیر زمان اجرا دو روش گفته شده با هم مقایسه شده است:

In [42]:

```
%matplotlib inline
from matplotlib.pyplot import *
import timeit
def get_time(f, input):
    start = timeit.default_timer()
    f(input)
    stop = timeit.default_timer()
    return (stop - start) * 1000 * 1000 #microseconds
def get_avg_time(f, input):
    res = 1 << 30 #inf
    s = 10
    r = 5
    for i in range(s):
        sum = 0
        for j in range(r):
            sum += get_time(f, input)
        res = min(res, sum / float(r))
    return res

def plot1(N, numbers):
    figure(figsize=(20, 10))
    xlabel("Size of input (n)")
    ylabel("Time")
    y = [get_avg_time(heap.build_heap_with_bubble_up, a) for a in numbers]
    plot(N, y, 'r', label='build heap with bubble down', linewidth=5)
    y = [get_avg_time(heap.build_heap_with_bubble_down, a) for a in numbers]
    plot(N, y, 'b', label='build heap with bubble up', linewidth=5)
    legend(loc=2)

from random import randrange
max_N = 30000
N = [1, 5] + list(range(100, max_N, 5000))
numbers = [[randrange(100000) for j in range(i)] for i in N]
plot1(N, numbers)
```



کاربردهای هرم

صف اولویت

صف اولویت داده ساختاری 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. یک روش پیاده‌سازی صف اولویت با استفاده از هرم است. البته این داده ساختار را می‌توان با داده ساختارها ای دیگری نظیر صف و لیست پیوندی هم پیاده‌سازی کرد. برای مطالعه بیشتر می‌توانید به [این لینک](#) (https://en.wikipedia.org/wiki/Priority_queue) مراجعه کنید.

استفاده از هرم در مسائل

از هرم در بسیاری از سولات و الگوریتم‌های معروف استفاده می‌شود. چند نمونه از این الگوریتم‌ها عبارت‌اند از:

- الگوریتم دایکسترا(Dijkstra) برای یافتن مسیر با طول کمینه
- الگوریتم پریم(Prime) برای یافتن زیر درخت کمینه
- الگوریتم‌های یافتن عناصر خاص مانند میانه و k -مین عضو کوچک

خودآزمایی ۱:

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 اعداد و تعداد لیست‌ها هستند.

In [77]:

```
from src.tests.tester import tester
```

In [1]:

```
def merge_lists(list_of_sorted_lists):
    sorted_list = []

    #merge the lists

    return sorted_list

tester("merge lists", merge_lists)
```

```
NameError                                 Traceback (most recent call last)
<ipython-input-1-a9b49462e456> in <module>
      6     return sorted_list
      7
----> 8 tester("merge lists", merge_lists)

NameError: name 'tester' is not defined
```

مرتبسازی هرمی

یکی از کاربردهای مهم هرم، مرتبسازی هرمی است که در فصل بعد به آن اشاره می‌شود.

بنام خدا

دانشگاه آزاد اسلامی شیراز - دانشکده مهندسی

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل ششم، بخش اول: مرتب‌سازی هرمی

فهرست محتویات

- مقدمه
- یادآوری هرم
- مرتب‌سازی هرمی

In []:

In []:

مقدمه

در بخش پنجم با داده ساختار هرم آشنا شده اید. در اینجا به بررسی یکی از مهم‌ترین کاربرهای هرم، یعنی مرتب‌سازی هرمی می‌پردازیم. در این روش اعدادی که

می خواهیم مرتب کنیم را درون یک هرم (مثلا هرم بیشینه) قرار می دهیم. سپس کافیست مقدار بیشینه را از هرم بخوانیم و آن را ثبت و از هرم حذف ش کنیم و این کار را آنقدر ادامه می دهیم تا هرم خالی شود.

یادآوری هرم

هرم داده ساختاری است که با استفاده از آرایه پیاده سازی می شود و عمل درج، حذف بزرگ ترین (کوچک ترین) عنصر و افزایش و کاهش کلید در آن در زمان $O(\log n)$ انجام می گیرد. برای توضیحات بیشتر به قسمت چهارم بخش درخت ها مراجعه کنید. برای الگوریتم مرتب سازی هرمی از توابع تعریف شده در آن قسمت استفاده کرده ایم.

مرتبسازی هرمی

مرتب سازی هرمی به شرح زیر است: در ابتدا آرایه i ورودی را با رویه ساخت هرم که در بخش هرم ها ذکر شد، به صورت یک هرم بیشینه در می آوریم. پس از آن بزرگترین عنصر آرایه را مشابه با عمل حذف بزرگترین عنصر که در بخش 5-4 شرح داده شد، با عنصر آخر آرایه جایه جا می کنیم. سپس عناصر باقی مانده را به صورت هرم در می آوریم. با $1 - n$ بار انجام این عمل در نهایت آرایه به صورت مرتب شده در می آید. همانطور که در بخش هرم گفته شد، پیچیدگی ساخت هرم از $O(n)$ است. همچنین هزینه i عمل $O(\log i)$ از $MaxHeapify(1, i)$ است بنابراین هزینه i کل مرتب سازی 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؟
2. آیا می‌توان این مرتبسازی را به صورت stable پیاده‌سازی کرد؟

خودآزمایی ۱ : به کمک هرم کمینه و به روش مرتبسازی هرمی، لیست ورودی را مرتب کنید و آن را برگردانید.

In [59]:

```
from src.tests.tester import tester
```

In [60]:

```
class min_Heap:  
    def __init__(self):  
        pass  
  
    def insert(a):  
        pass  
  
    def get_min():  
        pass  
  
    def del_min():  
        pass
```

In [90]:

```
def heap_sort(list_of_numbers):  
  
    #sort the given list with heap sort and return the sorted list  
    list_of_numbers.sort()  
  
    return list_of_numbers  
  
tester("heap sort", heap_sort)
```

Your code passes all 3 test(s).

بنام خدا

دانشگاه آزاد اسلامی شیراز - دانشکده مهندسی

داده‌ساختارها و الگوریتم‌ها

ترم دوم سال تحصیلی 1400-1401

فصل ششم، بخش دوم: درخت تصمیم، حد پایین مرتب‌سازی

مقایسه‌ای

فهرست محتویات

- مقدمه
- حد پایین
- تلاشی برای بدست آوردن حد پایین مرتب‌سازی مقایسه‌ای
- درخت تصمیم
- حد پایین برای الگوریتم‌های مرتب‌سازی مقایسه‌ای در حالت میانگین

In []:

مقدمه

در این بخش از درس، به بررسی مفهوم درخت تصمیم و یافتن یک کران پایین برای تعداد مقایسه‌ها در الگوریتم‌های مرتب‌سازی مقایسه‌ای می‌پردازیم.

In []:

حد پایین

در ابتدا ببینیم که محاسبه‌ی حد پایین برای یک کار چه معنایی دارد؟ تا اینجای درس ما با این آشنا شده‌ایم که الگوریتمی به ما داده‌شود 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 بلد 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 به دست آورده‌ایم و در واقع مستقل از نحوه‌ی پیاده‌سازی الگوریتم می‌گوییم تمامی الگوریتم‌هایی که آن کار مشخص را انجام می‌دهند از $\Theta(f(n))$ هستند. البته این بیان کمی نادرست زیرا دقیقاً باید مشخص شود که پیچیدگی زمانی یک الگوریتم برای انجام یک کار را چه تعریف می‌کنیم.

برای گرفتن شهود بیشتر یک مثال ساده را بررسی می‌کنیم:

- مساله: 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 و گرنه اگر خروجی الگوریتم همان عضو باشد ممکن است آن عضو، عضوی مینیمم باشد یا اگر خروجی آن الگوریتم آن عضو نباشد، ممکن است آن عضو، خود عضو ماکسیمم باشد. در واقع می‌توان گفت که ما از آن عضو هیچ اطلاعاتی جمع آوری نکرده‌ایم. که این سبب می‌شود الگوریتم درست کار نکند. پس می‌توان گفت لاقل باید $\lceil n/2 \rceil$ 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 راس 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. آیا این گراف می‌تواند ناهمبند باشد؟

تلاشی برای بدست آوردن حد پایین مرتب‌سازی مقایسه‌ای

مرتب‌سازی از مهم‌ترین مسائل دنیای 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 پاسخ می‌دهند، حد پایین دارند؟

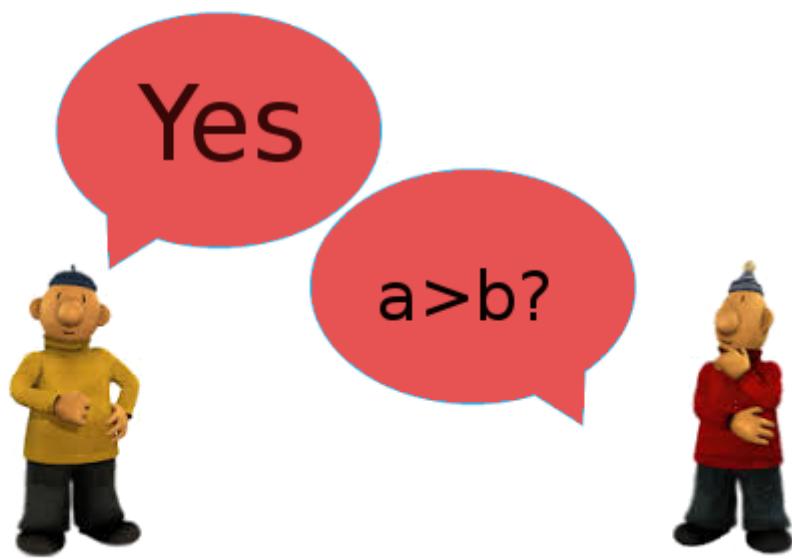
تلاشی برای پیدا کردن یک حد پایین:

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 a, b, c است. ابتدا تمامی دنباله‌های ممکن مرتب شده را در نظر می‌گیرد (همانند تصویر زیر) تا سپس بتواند دنباله‌ی مرتب شده‌ی 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 می‌شنود:



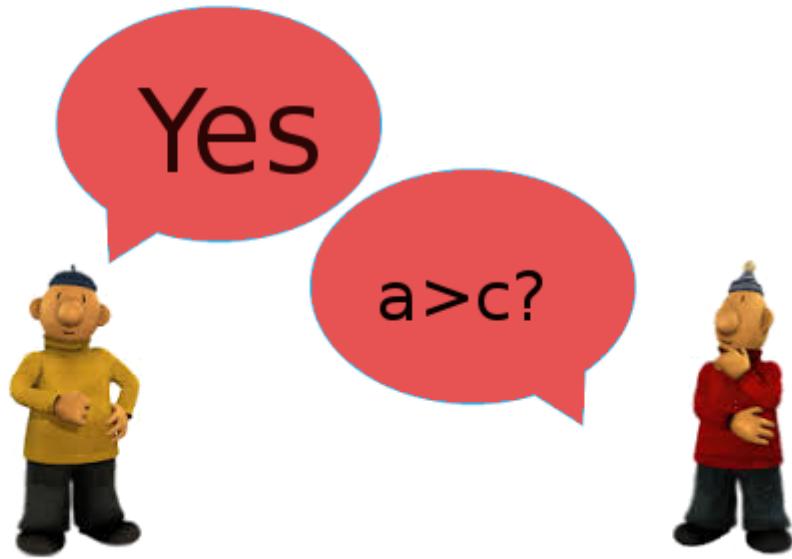
و حالا مت حالاتی که این شرط را ندارند از ذهنش را حذف می کند:

b,a,c
b,c,a
c,b,a



فکرک: آیا مت میتوانست سوالی را بپرسد که حالات بیشتری را برای او حذف کند؟

حالا مت از پت میپرسد که آیا $c > a$ و باز هم جواب مثبت میشنود:



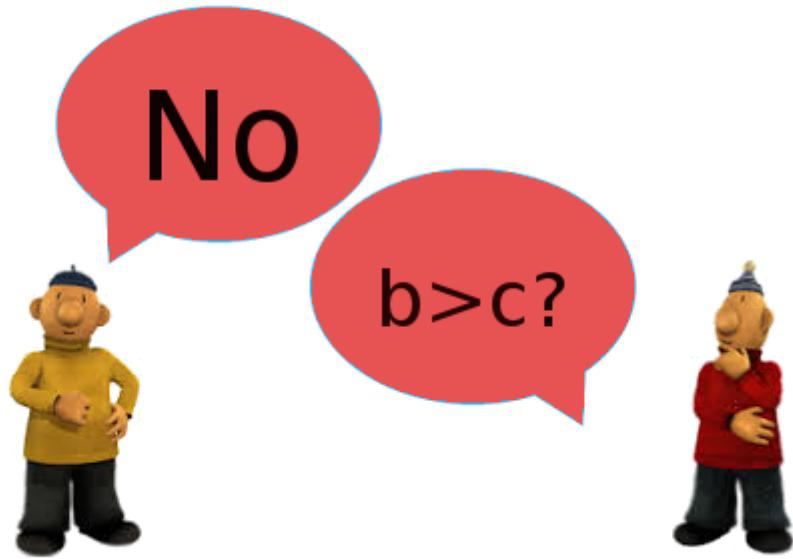
فکر ک: آیا مت می توانست سوال هوشمندانه تری بپرسد؟

و حالا مت حالاتی را که در این شرایط صدق نمی کنند از ذهنی حذف می کند:

b,c,a
c,b,a



حالا مت با یک پرسش می‌تواند دنباله‌ی مرتب شده را بیابد. او از پت می‌پرسد که آیا $c > b$ و جواب منفی می‌گیرد:



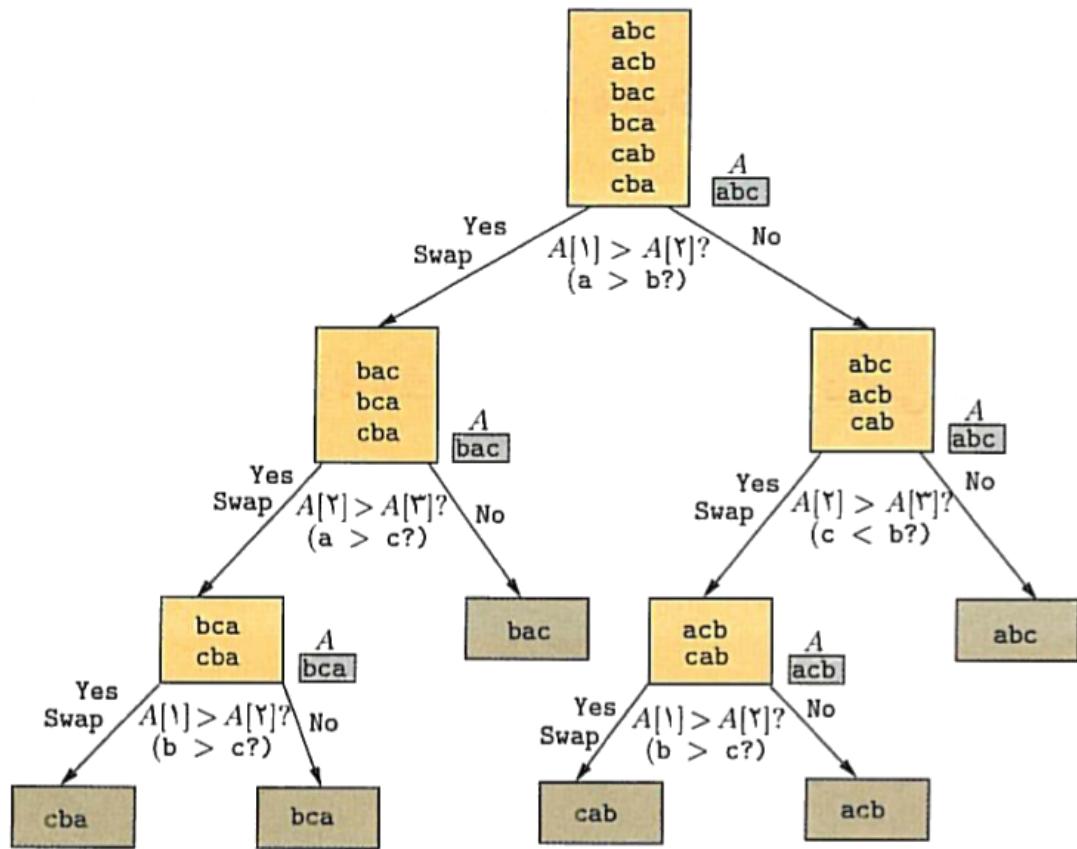
و حالا مت رشته‌ی مرتب شده را می‌داند:

b,c,a



درخت تصمیم

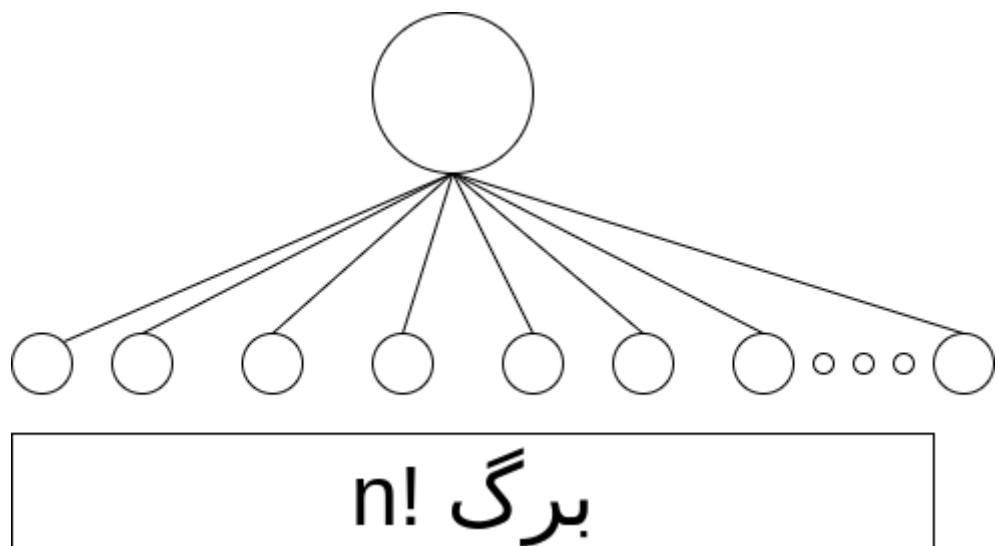
درخت تصمیم در واقع تنها یک مدل شهودی برای بررسی عملکرد بعضی الگوریتم‌های است. شهود ساده‌ی پشت مفهوم «درخت تصمیم» این است که یک الگوریتم غیر احتمالاتی بر اساس داده‌هایی که تا کنون به دست آورده ممکن است چند تصمیم مختلف بگیرد. برای مثال فرض کنید که مت در مثال بالا برای فهمیدن رشته‌ی پت از درخت تصمیم زیر استفاده می‌کند. دقت کنید که اتفاقی که در بالا افتاد بخشی از درخت تصمیم مت بود!



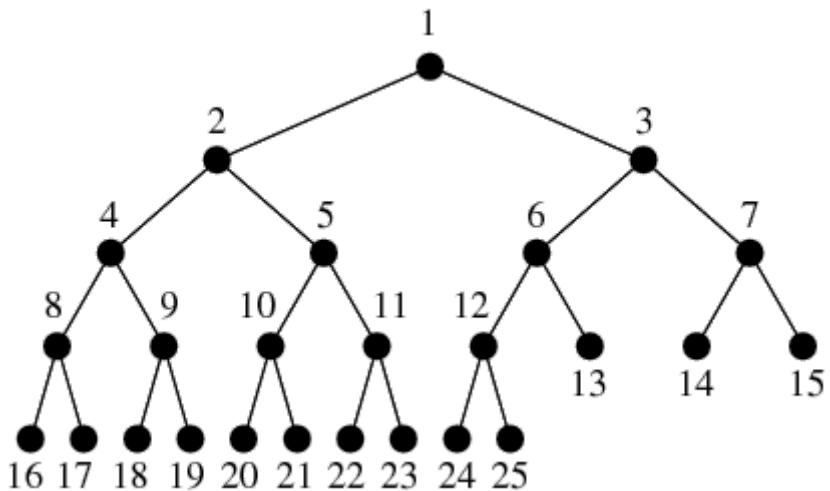
(عکس بالا از کتاب «داده‌ساختارها و مبانی الگوریتم‌ها» نوشته‌ی دکتر محمد قدسی گرفته شده است.) برای هر الگوریتم مرتب‌سازی غیر احتمالاتی می‌توان برای هر n چنین درختی را رسم کرد. دقیق کنید که این درخت قطعاً $n!$ تا برگ دارد زیرا در هر راس مجموعه‌ی جواب‌هایی را می‌بینید که ممکن است جواب باشند و هنوز طبق اطلاعات گرفته شده ردنشده‌اند. 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 (عمر 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 $\Theta(n \log n)$ دارد، توانسته‌ایم حد پایینی برای الگوریتم‌های مرتب‌سازی مقایسه‌ای بیابیم. بباید کمی فکر کنیم. آیا ممکن است که ثابت کنیم که درختی که $n!$ برگ دارد، محدودیتی به این صورت داشته باشد که عمق آن اجبارا از مقداری بیش‌تر باشد؟ تصویر زیر این ادعا را رد می‌کند:



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 یا «بله» است یا «خیر». در واقع شکل درخت‌های تصمیم به صورت زیر است:



و در واقع درخت تصمیم در هر راس تنها می‌تواند به دو شاخه تقسیم شود. در این زمینه لم زیر را داریم:

لم 1: یک درخت دودویی با ارتفاع h حداقل 2^h برگ دارد.

اثبات این لم ساده‌است. تنها کافیست بر روی h اسقرا بزنید.

حال از آنجا که درخت تصمیم یک درخت دودویی است و $n!$ برگ دارد اگر ارتفاع یا عمق آن را بگیریم خواهیم داشت :

$$2^h \geq n! \rightarrow h \geq \lceil \log(n!) \rceil$$

و از قبل می‌دانیم که

$$\log(n!) = \Theta(n \log(n))$$

پس داریم که h از $\Omega(n \log(n))$ است.

تبیک! ما حد پایینی برای عملکرد تمامی الگوریتم‌های مقایسه‌ای غیر احتمالاتی در بدترین حالت یافتیم.

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 باقی خواهد ماند. 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 رشته‌ی ما جزو آن مجموعه‌ی بزرگتر باشد پس لاقل یک ورودی وجود دارد که به ازای آن با هر بار پرسش اندازه‌ی مجموعه‌ی حالات 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 یافته‌ی اما باید ببینیم که عملکرد الگوریتم‌ها در حالت میانگین هم حد پایینی دارد یا خیر؟ در واقع میخواهیم ببینیم 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، $E(X)$ را میخواهیم که نتیجه میدهد باید تعداد مقایسه‌ها را به ازای هر حالت از $n!$ حالت ممکن بدست 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.

حال لم زیر را داریم:

لم ۲: در هر درخت دودویی با m برگ، مجموع عمق برگ‌ها دست کم $m \log(m)$ است.

اثبات: اثبات این حکم به وسیله‌ی استقرای انجام می‌شود و سپس کافیست درخت را به دو قسمت چپ و راست تقسیم کنید و فرض کنید که i برگ در سمت چپ و درخت سمت راست i برگ دارد. و حال میتوانیم ببینیم مجموع عمق برگ‌ها در کجا کمینه می‌شود. اثبات دقیق این لم را می‌توانید در کتاب «داده‌ساختارها و مبانی الگوریتم‌ها»^{۳۱۱} دکتر قدسی در صفحه‌ی ۳۱۱ مشاهده کنید.

با پذیرفتن این لم داریم که الگوریتم‌های مرتب‌سازی مقایسه‌ای در حالت میانگین هم عملکردی از $\Omega(\log(n!))$ دارند زیرا:

$$\sum_{l_i \text{ in leaves}} h(l_i) \geq n! \log(n!) \rightarrow \frac{\sum_{l_i \text{ in leaves}} h(l_i)}{n!} \geq \log(n!)$$

توجه!

دقت کنید که درخت تصمیم ابزاری قدرتمند برای اثبات حد پایین برای الگوریتم هاست 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 با طول n داشته باشیم و بخواهیم تنها با انجام تعدادی مقایسه بین این دو دنباله آن‌ها را با هم ادغام کنیم با استفاده از درخت تصمیم ثابت می‌شود که به لااقل $\Omega(\log_2 \binom{m}{n+m})$ مقایسه نیاز داریم. آیا می‌توانید این موضوع را ثابت کنید؟

In []:

بنام خدا

دانشگاه آزاد اسلامی شیراز - دانشکده مهندسی

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل ششم، بخش سوم: مرتب‌سازی سریع

فهرست محتویات

- مقدمه
- فرآیند الگوریتم
- افراز
- تقسیم و حل
- حالات بد مرتب‌سازی سریع
- حالات خوب مرتب‌سازی سریع

مقدمه

ابتدا می‌خواهیم بررسی کنیم که چرا از الگوریتم مرتب‌سازی سریع استفاده می‌شود. شما تا کنون با مرتب‌سازی ادغامی آشنا شده‌اید و فهمیدید که مرتبه زمانی اجرای آن از $O(n \log n)$ است و همان طور که در مبحث درخت تصمیم فرا گرفته‌اید، این بهترین

مرتبه زمانی ممکن 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:

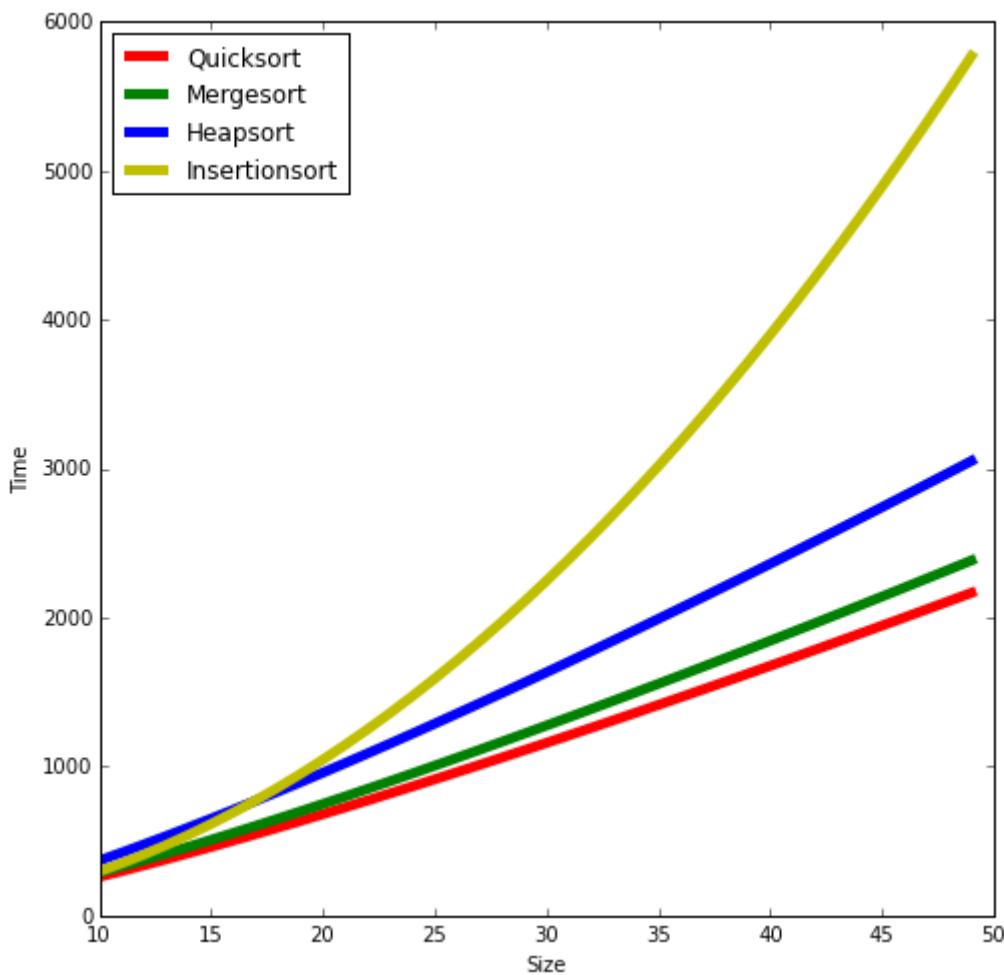
1. مرتب‌سازی سریع، یک الگوریتم درجا است، به همین دلیل حافظه کمتری مصرف می‌کند، هچمنین در زمان پیاده‌سازی نیاز کمتری به دسترسی به RAM دارد.
2. در حالت میانگین، ضرایب ثابت آن کمتر است، شما می‌توانید در نمودار زیر مقایسه زمان اجرای چند الگوریتم را در حالت میانگین ببینید.
3. پیاده‌سازی آن آسان است.

In [1]:

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

import math

def plot2(N):
    figure(figsize=(8, 8))
    xlabel("Size")
    ylabel("Time")
    y = [11.667*(n+1)*math.log(n) - 1.74*n - 18.74 for n in N]
    plot(N, y, 'r', label='Quicksort', linewidth=5)
    y = [12.5*n*math.log(n) for n in N]
    plot(N, y, 'g', label='Mergesort', linewidth=5)
    y = [16*n*math.log(n)+0.01*n for n in N]
    plot(N, y, 'b', label='Heapsort', linewidth=5)
    y = [2.25*pow(n,2)+7.75*n-3*math.log(n) for n in N]
    plot(N, y, 'y', label='Insertionsort', linewidth=5)
    legend(loc=2)
from random import randrange
max_N = 50
N = range(10, max_N, 1)
plot2(N)
```



فرآیند الگوریتم

برای بررسی مرتب‌سازی سریع برای آرایه $A[p..r]$ دو گام کلی زیر را داریم:

۱. افزایش آرایه به سه بخش (ممکن است بخشی‌ها خالی باشند) $A[p..q-1]$ و $A[q+1..r]$ و $A[q]$ افزایش می‌شود به طوری که هر عنصر $A[q]$ کمتر یا مساوی $A[q+1..r]$ باشد. عنصر $A[p..q-1]$ را عنصر محور این آرایه می‌نامیم.
۲. تقسیم و حل: هر یک از بخش‌های $A[q+1..r]$ و $A[p..q-1]$ را با فراخوانی تابع بازگشتی مرتب‌سازی سریع، مرتب می‌کنیم. بنابراین در نهایت آرایه اصلی نیز مرتب خواهد بود.

افراز

گیف زیر روند افزار در مرتب سازی سریع را نشان می دهد.



قطعه کد زیر تابع افزار (partition) در مرتب سازی سریع است. برای بررسی درستی این تابع به این نکته توجه کنید که در طول اجرای حلقه خط چهارم، 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 پایان پذیر است، چون در هر مرحله از اجرای حلقه، فاصله اشاره‌گرها حداقل یکی کم می‌شود.

In [12]:

```
import random
def swap(x,y):
    x, y = y, x

def partition(A, left, right):
    leftPointer = left
    rightPointer = right-1
    swap(A[right], A[random.randint(left,right)])
    pivot = A[right]
    while True:
        while leftPointer<=rightPointer and A[leftPointer] <= pivot :
            leftPointer=leftPointer+1
        while rightPointer >= leftPointer and A[rightPointer] > pivot :
            rightPointer=rightPointer-1
        if leftPointer >= rightPointer :
            break
        else :
            A[leftPointer],A[rightPointer] = A[rightPointer], A[leftPointer]
            A[leftPointer], A[right] = A[right], A[leftPointer]

    return leftPointer, pivot
```

تقسیم و حل

روند بازگشتی مرتب سازی سریع(تقسیم و حل) در گیف زیر نمایان است.



تابع بازگشتی مرتب سازی سریع را در قطعه کد زیر مشاهده می کنید. درستی این تابع را می توان با استقرا روی طول آرایه ثابت کرد. به این صورت که فرض می کنیم، هر آرایه ای با طول کمتر از n را می توان با این تابع مرتب کرد، حال برای آرایه با طول n توسط محور انتخاب شده ، آرایه را به دو تکه تقسیم می کنیم و هر تکه را به صورت بازگشتی مرتب می کنیم. حالت پایه الگوریتم نیز آرایه هایی با طول یک و یا صفر است که در شرط خط پنجم بررسی شده است.

In [13]:

```
A=[19,97,22,50,93,31,72,11,46]
```

```
def quickSort(left, right):
    if right <= left:
        return
    index, pivot = partition(A, left, right)
    print("left=",left,"right=",right,"pivot=",pivot,"index=",index)
    print(A)
    print("")
    quickSort(left, index-1)
    quickSort(index+1, right)
# global A
quickSort(0,8)
```

```
left= 0 right= 8 pivot= 46 index= 4
[19, 11, 22, 31, 46, 50, 72, 97, 93]
```

```
left= 0 right= 3 pivot= 31 index= 3
[19, 11, 22, 31, 46, 50, 72, 97, 93]
```

```
left= 0 right= 2 pivot= 22 index= 2
[19, 11, 22, 31, 46, 50, 72, 97, 93]
```

```
left= 0 right= 1 pivot= 11 index= 0
[11, 19, 22, 31, 46, 50, 72, 97, 93]
```

```
left= 5 right= 8 pivot= 93 index= 7
[11, 19, 22, 31, 46, 50, 72, 93, 97]
```

```
left= 5 right= 6 pivot= 72 index= 6
[11, 19, 22, 31, 46, 50, 72, 93, 97]
```

حالات بد مرتب سازی سریع

گرچه در ابتدای متن اشاره کردیم که حالت میانگین مرتب سازی سریع بهتر از سایر الگوریتم های مبتنی بر مقایسه است، اما حالات خاصی نیز وجود دارند که مرتبه زمانی اجرای الگوریتم از

$\Theta(n^2)$ می‌شود.

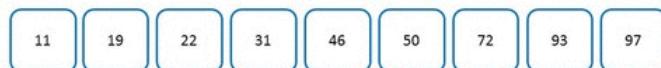
این حالات هنگامی رخ می‌دهند که افزار متوازن نباشد، بدین معنا که یکی از بخش‌های حاصل از افزار k عضوی باشد که k عدد ثابتی است که باعث می‌شود حل زیربخش دیگر، که $n - k$ عضو دارد، تقریباً به اندازه حالت ابتدایی مشکل باشد.

یک مثال بارز از این موضوع، آرایه مرتب شده‌است، که همانطور که در شکل زیر مشاهده می‌کنید، یکی از بخش‌ها همواره صفر عضوی است، بنابراین برای رابطه بازگشتی زمان اجرا خواهیم داشت:

$$T(n) = T(n - 1) + T(0) + \theta(n)$$

$$= T(n - 1) + \theta(n)$$

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 مرتب سازی درون‌گرا(introsort) که ترکیبی از الگوریتم‌های مرتب سازی تصادفی و هرمی است. این الگوریتم با فراخوانی مرتب‌سازی سریع شروع می‌شود و هنگامی که بخش بازگشتی مرتب‌سازی سریع به عمق مشخصی (متناوب با $\log n$) رسید، الگوریتم مرتب‌سازی هرمی را فراخوانی می‌کند.

حالات خوب مرتب سازی 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 دوبخش 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 $\lceil(n - 1)/2\rceil$ و $\lfloor(n - 1)/2\rfloor$ است که رابطه بازگشتی زمان اجرای زیر را نتیجه می‌دهد:

$$T(n) = 2T((n - 1)/2) + \theta(n)$$

که به راحتی می‌توان نشان داد از $\theta(n \log n)$ است.

In []:

بِنَامِ خَدَا

دانشگاه آزاد اسلامی شیراز - دانشکده مهندسی

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل هشتم، بخش اول: مرتبه‌آماری

فهرست محتویات

- الگوریتم پیدا کردن k امین مرتبه آماری
- انتخاب سریع
- میانه میانه‌ها

In []:

الگوریتم پیدا کردن k امین مرتبه آماری

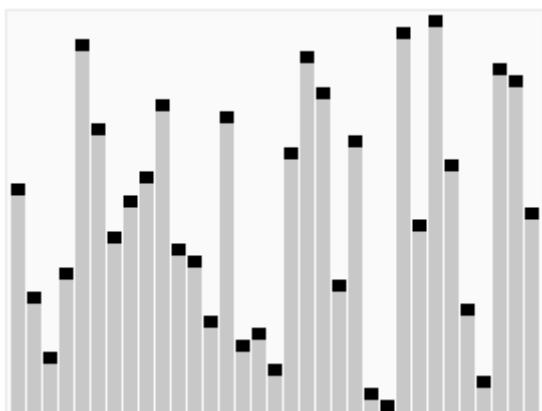
مساله‌ای که به آن می‌پردازیم، یافتن k امین کوچک‌ترین عدد در یک لیست (k امین مرتبه‌ی آماری) است.

اولین راهی که برای این کار به ذهن می‌رسد، مرتب کردن آرایه و انتخاب عنصر k ام آن است که پیچیدگی زمانی $O(n \log n)$ خواهد داشت.

ما در این دفترچه سعی می‌کنیم الگوریتم هایی با زمان اجرای بهتر برای این مسئله پیدا کنیم.

انتخاب سریع

انتخاب سریع از رویکرد یکسانی با مرتب‌سازی سریع بهره می‌برد، عضوی را به عنوان عنصر محوری انتخاب کرده و داده‌ها را بر پایه آن به دو قسمت تقسیم می‌کند (دو قسمت کمتر و بیشتر از عنصر محوری). سپس با توجه به طول دو قسمت و k 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 عناصرهای محوری انتخاب شده می‌باشد. اگر عناصرهای محوری انتخاب شده مناسب باشند، به این معنا که تعداد داده‌ها را در هر مرحله با نسبت معینی کاهش دهند، تعداد داده‌ها به صورت نمایی کاهش می‌یابد و در نتیجه کارکرد الگوریتم به صورت خطی می‌شود. اگر عناصرهای محوری نامناسب به صورت پیاپی انتخاب شوند، مانند حالتی که در هر مرحله تنها یک عضو از داده‌ها کاسته شود، به بدترین عملکرد الگوریتم با $O(n^2)$ منجر خواهد شد. در بخش بعد برای پیدا کردن محور مناسب الگوریتم میانه میانه‌ها را معرفی می‌کنیم.

تمرین ۱ : ثابت کنید پیچیدگی زمانی الگوریتم بالا به طور متوسط خطی است.
راهنمایی :

$$T(n) \leq cn$$
$$T(n) \leq an + \frac{1}{n} \sum_1^n T(\max(i, n - i - 1))$$

تمرین ۲ : چرا به طور متوسط پیچیدگی زمانی این الگوریتم از $O(n)$ است در حالی که در مرتبسازی سریع $O(n \log n)$ است؟

در ادامه پیاده‌سازی از الگوریتم بالا ارائه می‌دهیم. خوب هست که جست‌وجو کنید و شبه کدهای 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.

In [1]:

```
import random

def random_select(a, k):
    assert len(a) > k

    v = a[random.randint(0, len(a) - 1)]

    # less than, equal, greater than x
    lt = [x for x in a if x < v]
    eq = [x for x in a if x == v]
    gt = [x for x in a if x > v]

    if len(lt) > k:
        return random_select(lt, k)
    elif len(eq) + len(lt) > k:
        return v
    else:
        return random_select(gt, k - len(eq) - len(lt))

a = [5, 6, 0, 1, 10, 12, -1, 8, 100]

print(random_select(a, 0))
print(random_select(a, 2))
print(random_select(a, 5))
print(random_select(a, 8))
```

```
-1
1
8
100
```

میانه میانه‌ها

این الگوریتم برای پیدا کردن میانه‌ی آرایه و یا به طور کلی تر n امین عنصر یک آرایه می‌باشد.

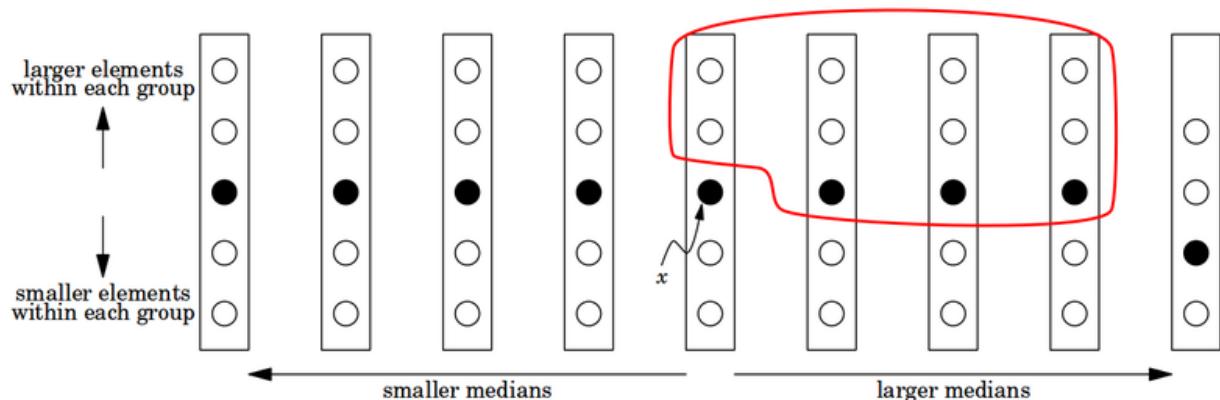
الگوریتم به صورت زیر می‌باشد:

- آرایه را به $n/5$ دسته‌ی ۵ تایی تقسیم می‌کنیم.
- هر دسته را مرتب کرده و میانه‌ی آن را پیدا می‌کنیم.
- به صورت بازگشته میانه‌های $n/5$ دسته‌ی یاد شده را انتخاب می‌کنیم.

- فرض کنید x میانه ای باشد که در مرحله قبل بدست آورده ایم. حال می توانیم آرایه را براساس x به دو دسته اعداد کوچکتر از x و اعداد بزرگتر از x تقسیم کنیم و با توجه به اندازه دسته ای اول و مقدار n عدد مورد نظر را در دسته مناسب به صورت بازگشته به کمک الگوریتم گفته شده پیدا کنیم.

فرض کنید x میانه ای باشد که در مرحله ۳ بدست آورده ایم. حال تمام دسته هایی که میانه آن ها از x کمتر است را در نظر بگیرید تعداد آنها $n/10$ است.

در هر یک از این دسته ها حداقل دو عضو کوچکتر از میانه خود دسته به همراه میانه انتخاب کرده از آن دسته مقداری کمتر از x دارند که تعداد آن ها $3n/10$ است. به طور مشابه می توان ثابت کرد حداقل $3n/10$ عنصر بزرگتر از x در آرایه وجود دارد.



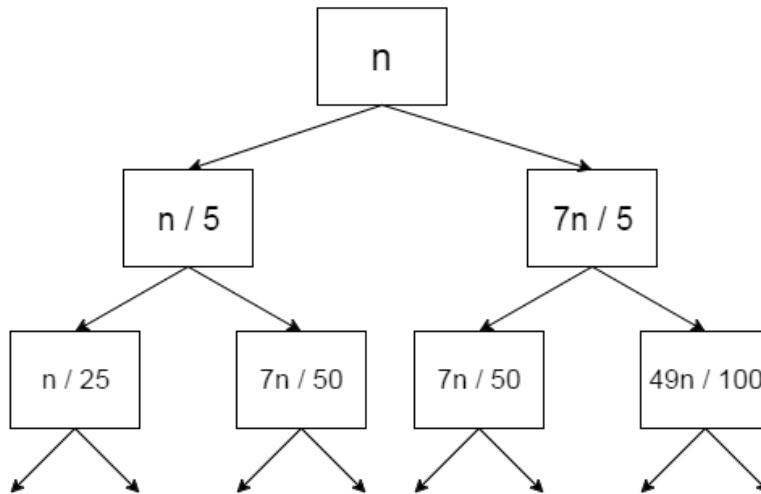
شکل ۲: پیچیدگی زمانی الگوریتم میانه میانه ها

با توجه به اینکه دسته اعداد کوچک تر از x و دسته اعداد بزرگتر از x حداقل $7n/10$ عضو دارند مرتبه زمانی الگوریتم فوق در بدترین حالت از فرمول زیر بدست می آید :

$$T(n) \leq T(n/5) + T(7n/10) + c \cdot n$$

که $T(n/5)$ زمان لازم برای پیدا کردن میانه میانه هاست و $T(7n/10)$ زمان لازم برای حل مساله در مرحله ای بعد بازگشته است.

از فرمول بالا می توان با استقرا دریافت که $T(n)$ از مرتبه زمانی $\theta(n)$ است.
همچنین با استفاده از درخت زیر مشاهده می شود که با طی کردن هر مرحله جمع رئوس نسبت به مرحله‌ی قبل $9/10$ شده است و در نهایت مرتبه‌ی زبانی الگوریتم $\theta(n)$ خواهد بود.



شکل ۳: درخت پیچیدگی زمانی الگوریتم میانه میانه‌ها

تمرین ۳: در حالت دسته‌های ۳ و ۷ تایی تابع $T(n)$ را تعیین کنید و پیچیدگی زمانی آن را تعیین کنید.

راهنمایی :

$$T_3(n) = T_3(n/3) + T_3(4n/6) + c \cdot n = \theta(n \log n)$$

$$T_7(n) = T_7(n/7) + T_7(10n/14) + c \cdot n = \theta(n)$$

در ادامه پیاده‌سازی از این الگوریتم ارائه می‌دهیم.

In [3]:

```
def median_ofmedians(A, i):  
  
    #divide A into sublists of Len 5  
    sublists = [A[j:j+5] for j in range(0, len(A), 5)]  
    medians = [sorted(sublist)[len(sublist)//2] for sublist in sublists]  
    if len(medians) <= 5:  
        pivot = sorted(medians)[len(medians)//2]  
    else:  
        #the pivot is the median of the medians  
        pivot = median_ofmedians(medians, len(medians)//2)  
  
    #partitioning step  
    low = [j for j in A if j < pivot]  
    high = [j for j in A if j > pivot]  
  
    k = len(low)  
    if i < k:  
        return median_ofmedians(low,i)  
    elif i > k:  
        return median_ofmedians(high,i-k-1)  
    else: #pivot = k  
        return pivot  
  
A = [1,2,3,4,5,1000,8,9,99]  
B = [1,2,3,4,5,6]  
print (median_ofmedians(A, 0))  
print (median_ofmedians(A,7))  
print (median_ofmedians(B,4))
```

1
99
5

برای اطمینان پیدا کردن از عملکرد خطی حتی در بدترین حالت ، می توان از الگوریتم بالا برای تعیین عنصر محوری استفاده کرد. البته سربار محاسباتی تعیین عنصر محوری در این روش زیاد است و در عمل از این الگوریتم استفاده نمی شود. به همین دلیل می توان از انتخاب درونگرا (<https://en.wikipedia.org/wiki/Introselect>) استفاده کرد که با ترکیب انتخاب سریع با میانه میانه ها، هم در حالت میانگین و هم در بدترین حالت، عملکردی خطی را نتیجه می دهد.

همچنین با همین دیدگاه می توان روش مرتب سازی جدیدی به نام مرتب سازی درونگرا (<http://www.geeksforgeeks.org/know-your-sorting-algorithm-/>). را معرفی کرد که سریع ترین الگوریتم موجود است.

مرتب‌سازی درونگرا یک مرتب‌سازی ترکیبی است که از روش‌های مرتب‌سازی سریع، مرتب‌سازی هرمی و مرتب‌سازی درجی استفاده می‌کند.

روش این الگوریتم به این صورت است که در ابتدا با مرتب‌سازی سریع شروع می‌کند و اگر عمق بازگشتی آن از یک حد مشخصی گذشت 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. و همچنین هر گاه که تعداد اجزای مرتب‌سازی کم باشد از مرتب‌سازی درجی استفاده می‌کند:

- اگر عمق اجرای الگوریتم (عمق فراخوانی‌های تابع بازگشتی) حد بیشینه برای عمق را رد کند سراغ مرتب‌سازی هرمی می‌رود. حد بیشینه را برابر با $\log(N) * 2$ تعریف می‌کنیم.
- اگر اندازه‌ی قطعه‌ای که قرار است مرتب کنیم به حدی کوچک باشد باشد که استفاده از مرتب‌سازی درجی بهینه‌تر باشد از آن استفاده می‌کنیم و این حد را برابر با ۱۶ تعریف می‌کنیم.
- اگر هیچ کدام از این دو مورد رخ ندهد همان مرتب‌سازی سریع را ادامه خواهیم داد.

In []:

بنام خدا

دانشگاه آزاد اسلامی شیراز - دانشکده مهندسی

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل هشتم، بخش دوم: جست و جوی دودویی و کران بالا و

پایین

فهرست محتویات

- 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

In []:

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؟!



در این موقع مخصوصا اگر به عنوان نفرات اول به محل گرفتن کارت رسیده باشد معمولا 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. چند وضعیت مختلف در این موقع ممکن است رخ دهد. 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. اما اگر افراد لایقی دست‌اندرکار برگزاری همایش باشند مسئولین آن قبل از 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 بويي از الگوريتم بردء باشد به گشتن تک به تک کارت‌ها

بسنده نمی‌کند و سعی می‌کند از تلاش تیم برگزاری در مرتب‌کردن کارت‌ها بهره بگیرد.
شما اگر مسئول ثبت نام 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 (چه طور؟). 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.

جست و جوی دودویی

مسئله‌ای که در بالا دیدیم نمونه‌ای از یک مسئله‌ی جست و جو بود در دنباله‌ای از داده‌های مرتب شده. 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 یا عدم وجود آن در دنباله را گزارش کنیم که به این روش جست‌وجوی خطی 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 جست‌وجو بینجامد. برای این

کار می‌توانیم راحت‌ترین دسته بندی یعنی تقسیم به دو دسته‌ی مساوی را برای پیش‌برد جست‌وجو انتخاب کرد. در این صورت اگر عضو میانه دنباله را mid بنامیم، کافی است mid را با x مقایسه کنیم. در صورتی که x از mid بزرگ‌تر بود، باید در دسته‌ی اول یعنی از ابتدا تا جایگاه mid دنباله، به جست‌وجوی x بپردازیم و در غیر این صورت باید در دسته‌ی دوم یعنی از جایگاه mid تا انتهای دنباله به دنبال x بگردیم. حال طول دنباله‌ای که در آن به دنبال x می‌گردیم نصف شده است و می‌توانیم به شکل بازگشته 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 $(l, r]$ باشد (ابتدا بازه‌ی جواب شامل کل آرایه است).
- در هر مرحله mid را برابر $2(l + r)/2$ قرار می‌دهیم.
- اگر x کوچک‌تر از mid بود، جواب را در بازه $(l, mid]$ جست و جو می‌کنیم.
- در غیر اینصورت 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 جست‌وجو در یک دنباله جست‌وجوی دودویی می‌گویند. مقایسه این روش و جست و جوی خطی در شکل زیر قابل مشاهده است.

Binary search

steps: 0



Sequential search

steps: 0



www.penjee.com

در قطعه کد زیر، مسأله‌ی وجود عدد x در آرایه‌ی a را به روش جستجوی دودویی پیاده‌سازی می‌کنیم.

In [1]:

```
def binary_search(a, x):
    begin, end = 0, len(a)

    # current interval = [begin, end) -> len = end - begin
    while end - begin > 1: # it must be splitable
        mid = (begin + end) // 2

        if a[mid] > x:
            end = mid
        else: # x >= a[mid]
            begin = mid

    return a[begin] == x

a = [1, 3, 3, 5, 7, 9, 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 آرایه‌ی مرتب شده، وجود یا عدم وجود یک عنصر را اطلاع می‌داد؛ حال می‌خواهیم یک مسئله‌ی کلی‌تر را بررسی کنیم:

مسئله: آرایه‌ی صعودی a به همراه عدد دلخواه x داده شده است؛

آخرین (بزرگ‌ترین) اندیسی را پیدا کنید (مثل i) که $x < a[i]$. به این مساله lower bound نیز می‌گویند.

روش حل این مساله مشابه جست‌وجوی 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 جست‌وجو کرد.

البته این صرفا یک ایده و شهود برای حل مساله است و به جزئیات آن هم باید دقت شود؛ جزئیاتی همچون:

- 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 $x < a[mid]$ باشد، جواب در بازه‌ی $[mid + 1, n - 1]$ از آرایه است یا $[1 - 1, mid]$ ؟

پیشنهاد می‌شود کمی به این موارد فکر کنید زیرا بی‌دقی در این موارد عموماً باعث ایجاد الگوریتم‌های پایان‌ناپذیر، اشتباه و یا ناخوانا می‌شود.

روش حل عمومی

سعی می‌کنیم یک روش حل برای این جور سوال‌ها به دست آوریم. الگوریتم اصلی مبتنی بر [اصل ناوردایی](https://fa.wikipedia.org/wiki/اصل_ناوردایی) 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 بسته-باز تعریف کردیم. این کار به نسبت انتخاب بازه به صورت بسته-بسته 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 تک عضوی می‌شود و چون شرط ناوردایی در همه‌ی مراحل برقرار بوده، آن‌تک عنصر باقی‌مانده قطعاً جواب مساله است. البته حالتی که مساله کلاً 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 اول 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. ثانیا، 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 $1 + start < end$. با توجه به فرض $1 + start < end$ می‌توان ثابت می‌گیریم: $mid = \lfloor \frac{start+end}{2} \rfloor$ کرد $mid \in [start, end]$. پس این اندیس قطعاً در داخل بازه‌ی جواب است. حالا عنصر $a[mid]$ را با x مقایسه می‌کنیم. دو حالت پیش می‌آید:

1. $x \geq a[mid]$: 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

2. $x < a[mid]$: 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.

بنابراین شرط ناوردایی برقرار است و جواب همواره در بازه‌ی کاهش‌یافته قرار دارد؛ دقیقاً 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:

$$mid < end \Rightarrow end - start > mid - start .1$$

$$start \leq mid \Rightarrow end - start > end - mid .2$$

پس در هر دو حالت طول بازه‌ی جدید از بازه‌ی قدیمی کمتر است و کاهش به درستی انجام شده‌است. بنابراین هر دو شرط ناورداخی 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 [7]:

```
def lower_bound(a, x, start, end):
    if start == end-1:
        return start
    mid = end + start >> 1 # Division by 2 which is compatible with both python 2 & 3

    if a[mid] >= x:
        return lower_bound(a, x, start, mid)
    else:
        return lower_bound(a, x, mid, end)

a = [2, 3, 3, 5, 8, 8, 8, 10]
n = len(a)

index = lower_bound(a, 5, 0, n)
print(index, a[index])

print(lower_bound(a, 8, 0, n))
print(lower_bound(a, 1000, 0, n))
print(lower_bound(a, 0, 0, n))
```

2 3
3
7
0

دقت کنید که خروجی تابع `lower_bound` یک اندیس است نه مقدار آن عضو از آرایه. در مثال اول با توجه به وجود داشتن دو عدد ۳ در آرایه، عددی که اندیس بزرگتری دارد(اندیس ۲) جواب مساله است در سایر مثال‌ها صرفاً اندیس را چاپ کردیم؛ نکته مهم در مثال آخر 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. پس باید این حالت را جداگانه بررسی کنیم و یک جواب قراردادی ارائه دهیم؛ در زیر همان کد بالا به صورت غیر بازگشتی و با رعایت مورد آخر زده شده است که در صورت عدم وجود جواب ۱ - خروجی می‌دهد.

In [8]:

```
def lower_bound(a, x):
    if a[0] >= x:
        return -1

    start = 0
    end = len(a)

    while end > start + 1:
        mid = start + end >> 1

        if a[mid] >= x:
            end = mid
        else:
            start = mid

    return (start, a[start])

print(lower_bound([1, 3, 3, 3, 4], 3))
print(lower_bound([9, 10, 11], 8))
```

(0, 1)
-1

In [2]:

```
print(3 + 5 >> 1)
```

4

تابع بالا در صورت وجود جواب یک زوج مرتب از اندیس و عنصر متاظر آن در آرایه خروجی می‌دهد. در حالت کلی اگر یک الگوریتم بازگشتی را بتوانید به صورت

غیربازگشته 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 $x > a[i]$

راهنمایی: این بار بازه‌ی جواب را به صورت باز-بسته بگیرید، یعنی در ابتدا بازه را برابر با $[1 - n, 1 - 1)$ در نظر بگیرید و در هر مرحله سعی کنید بازه را به همین صورت نگه دارید.

هم‌چنین اگر جواب موجود نیست 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 [9]:

```
from src.tests.tester import tester

def upper_bound(a, x):
    # Implement your algorithm here
    pass

tester("upper_bound", upper_bound)
```

Your code get wrong answer in 3 test(s) from 3 test(s).

خودآزمایی ۲: آرایه نزولی a به همراه عدد x داده شده است. بزرگترین اندیسی را پیدا کنید (مثل i) که $x > a[i]$. فرمت خروجی مثل تمرین قبل است.

In [10]:

```
def magic_function(a, x):
    # Implement your algorithm here
    pass

tester("magic_function", magic_function)
```

Your code get wrong answer in 2 test(s) from 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 مساله نیامده است؛ این مثال را در نظر بگیرید:

یک آرایه‌ی 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 ۱, ۹, ۸, ۴ باشد و بخواهیم سه عدد را انتخاب کنیم. چهار حالت برای انتخاب این سه عدد وجود دارد:

۱. اعداد ۹, ۸, ۱ انتخاب شوند که کمترین اختلاف برابر است با $1 = 9 - 8$
۲. اعداد ۹, ۴, ۱ انتخاب شوند که کمترین اختلاف برابر است با $3 = 9 - 6$
۳. اعداد ۴, ۸, ۱ انتخاب شوند که کمترین اختلاف برابر است با $3 = 8 - 5$
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 مشکلی دارد ولی حل آن به یک نکته‌ی اساسی بستگی دارد: تغییر در زاویه‌ی نگاه به مساله؛ مساله را به یک [مساله‌ی تصمیم](#) تبدیل می‌کنیم که ساده‌تر از مساله‌ی اصلی است:

یک آرایه به‌همراه اعداد m و x داده شده است؛ آیا می‌توان m عدد را از آن طوری انتخاب کرده که کمینه‌ی اختلاف بین این m عدد حداقل x باشد؟ یا به بیان دیگر

اختلاف هر دو عدد حداقل x باشد؟

حل این مساله ساده است؛ اگر جواب مساله «بله» باشد، حداقل یک حالت وجود دارد که عدد مینیمم آرایه جزو m عدد انتخاب شده است (چرا؟). 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 انتخاب می‌کنیم و به همین طریق پیش می‌رویم. اصولاً کاندیدا های مجاز برای کمترین اختلاف، اعدادی هستند که در ترتیب مرتب شده‌ی آرایه پشت سر هم هستند (و بینشان عددی 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 [1]:

```
def check(a, m, x):  # a is supposed to be sorted
    ans = [a[0]]  # minimum of the array
    for i in range(1, len(a)):
        if a[i] - ans[-1] >= x:
            ans.append(a[i])
    if len(ans) >= m:
        return ans[0:m]  # return m elements that satisfies the criteria
    return None  # otherwise, return nothing
```

هزینه زمانی اجرای تابع `check` از $\Theta(n)$ است. حالا به این نکته‌ی مهم دقت می‌کنیم که اگر این الگوریتم برای عدد x خروجی «بله» بدهد، خروجی اش برای $1 - x$ و برای تمامی اعداد کمتر از x هم «بله» خواهد بود و اگر برای عدد y خروجی «خیر» بدهد، برای تمامی اعداد بزرگتر از y هم خروجی «خیر» می‌دهد. 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!. 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 \hat{z} است.
حال چگونه \hat{z} را بیابیم؟ مساله‌ی کران پایین را به یاد 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 اعضاء را به صفر تبدیل می‌کنیم و هدف، یافتن
اندیس آخرین ۱ است.

پس مساله‌ی یافتن \hat{z} 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 ماکسیمم 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 [3]:

```
def check(a, m, x): # a is supposed to be sorted
    ans = [a[0]] # minimum of the array
    for i in range(1, len(a)):
        if a[i] - ans[-1] >= x:
            ans.append(a[i])
    if len(ans) >= m:
        return ans[0:m] # return m elements that satisfies the criteria
    return None # otherwise, return nothing

def f(a, m):
    a.sort()
    start = 0
    end = a[-1] - a[0] + 1
    ans = []

    while end > start + 1:
        mid = start + end >> 1

        out = check(a, m, mid)
        if out is not None:
            ans = out
            start = mid
        else:
            end = mid

    return (start, ans)

f([1, 8, 9, 4], 3)
```

Out[3]:

(3, [1, 4, 8])

همان‌طور که مشاهده می‌شود خروجی تابع بالا یک زوج مرتب شامل z و یک روش انتخاب m عدد است که کمینه اختلافشان z شود. هزینه‌ی زمانی این الگوریتم از $\Theta(n \log(\max a - \min a))$ است.

بنام خدا

دانشگاه آزاد اسلامی شیراز - دانشکده مهندسی

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل نهم: توابع درهمسازی (قسمت اول)

فهرست محتویات

- مقدمه
- معرفی تابع درهمسازی
- برخی توابع درهمسازی
- برخورد و روش‌های مقابله با آن

مقدمه

یک مسئله ساده:

فرض کنید مجموعه از اعداد در بازه صفر تا صد داریم و می‌خواهیم داده ساختاری برای نگهداری آنها طراحی کنیم که امکان انجام عملیات‌های Insert ، Find و Delete را در زمان $O(1)$ داشته باشد.

راه حل: از یک آرایه به طول صد استفاده می‌کنیم. اگر خانه i ام 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 متناظر و اضافه کردن اعداد با یک کردن آن انجام می‌شود.

یک مسئله سخت‌تر: فرض کنید این بار اعداد در بازه صفر تا 10^{12} هستند، اما می‌دانیم حداقل 10^3 عدد در مجموعه وجود خواهد داشته باشند. آیا راه قبلی قابل استفاده است؟

معرفی تابع درهمسازی

تابع درهمسازی، تابعی مانند $L \rightarrow D : h$ است که اعضای مجموعه D (که اندازه دلخواه دارند) را به مجموعه با اندازه ثابت L نگاشت می‌دهد. معمولاً این توابع به گونه‌ای هستند که اندازه D بسیار بزرگتر از اندازه L 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 به این صورت که به هر رشته، تعداد حروف آن رشته است 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.

توابع درهمساز دامنه کاربردهای گسترده‌ای در مهندسی کامپیوتر دارند از جمله:

- داده ساختارهایی چون جدول درهمسازی
- سیستم‌های رمزنگاری

- مبادلات اینترنتی
- سیستم‌های تعیین هویت دیجیتال مانند تشخیص اثر انگشت
- بررسی صحت فایل‌ها

حل مسائلی مشابه مساله قبلی هم از کاربردهای پایه‌ای در همسازی است. راه حل این سوال را در ادامه می‌آوریم:

راه حل: فرض کنید در مساله قسمت قبل تابعی داشته باشیم که اعداد بین 0 و 10^{12} را به اعداد بین 0 تا 10^4 نگاشت کند، 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 معمولا همینطور است). حال با کمک این تابع، می‌توانیم ایده مساله اول را به مساله بزرگتر تعمیم دهیم به گونه‌ای که برای درج، یافتن و حذف یک عنصر مانند x کافیست به خانه (x) از آرایه h از 10^4 عضوی رجوع کنیم.

قطعه کد زیر پیاده‌سازی این مسئله را آورده است:

In [4]:

```
hash_size = 10**4

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

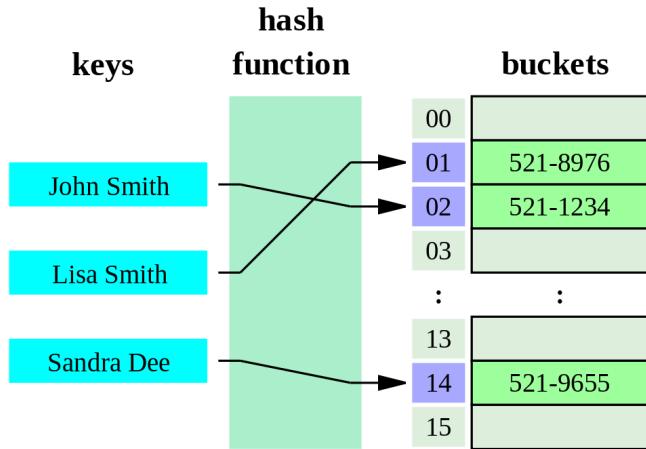
    def remove(self, x):
        self.table[hash_function(x)] = 0

s = my_set()
s.insert(10)
s.insert(245)
print(s.exists(124))
s.insert(8653)
s.insert(12543252)
print(s.exists(10))
s.remove(10)
print(s.exists(10))
print(s.exists(12543252))
print(s.exists(12543251))
print(s.exists(3252))
```

```
False
True
False
True
False
True
```

یک کاربرد: جدول درهمسازی

همانطور که گفته شد، یکی از کاربردهای بسیار مهم درهمسازی در داده ساختاری به اسم جدول درهمسازی است که تعمیمی از مثال زده شده در قسمت قبل است. این داده ساختار 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 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 مجموعه D عددهای با اندازه‌های مختلف هستند).

روش باقیمانده تقسیم

در این روش به هر عدد باقیمانده تقسیمش بر یک عدد ثابت مانند m را نسبت می‌دهیم. این دقیقاً همان کاری است که در مثال قبلی انجام دادیم.

$$h(x) = x \bmod m$$

انتخاب مقدار مناسبه برای m از اهمیت بالایی برخوردار است. به عنوان مثال اگر m توانی از دو باشد خروجی تابع درهمساز برابر $\lg(m)$ بیت کم ارزش m می‌شود و در نتیجه بقیه بیت‌های عدد در خروجی تابع درهمساز نقشی ندارند. به طور کلی بهتر است m یک عدد اول و دور از توان‌های دو باشد.

روش ضرب

یک استراتژی دیگر روش ضرب است که دو مرحله دارد. ابتدا داده‌ی ورودی ضرب در یک عدد ثابت A که $1 < A < 0$ شده و قسمت اعشاری آن گرفته می‌شود. سپس این قسمت اعشاری در عدد صحیح m ضرب شده و قسمت صحیح آن گرفته می‌شود.

$$h(x) = \lfloor m(xA \bmod 1) \rfloor$$

که منظور از باقی‌مانده به ۱ همان قسمت اعشاری عدد بوده. خوبی این است که دیگر انتخاب عدد m اینقدر مهم نیست. معمولاً در این روش m را توانی از ۲ گرفته و A را به صورت عدد گویای $\frac{s}{2^w}$ می‌گیریم که s یک عدد صحیح در بازه‌ی $0 < s < 2^w$ است.

روش همگانی - Universal hashing

می‌توان برای درهمسازی استفاده کرد، بالاخره امکان دارد اعداد ورودی به صورتی انتخاب شوند که همه به یک خانه نگاشته شوند.

یک روش برای مقابله با این مشکل این است که تعداد زیادی تابع درهمسازی آماده 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 گفته می‌شود.

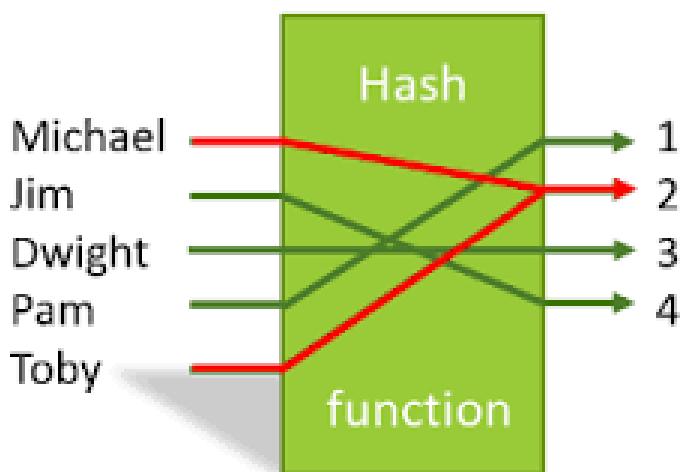
برای مثال اگر از روش ضرب استفاده می‌کنیم، می‌توانیم تعداد زیادی زوج عدد m, A داشته باشیم و در هنگام اجرای برنامه یکی را انتخاب کرده و باقی کارها را براساس آن انجام دهیم.

ثابت می‌شود که این روش به طور متوسط حتی روی بدترین ورودی ممکن روش کارآمدی است.

برخورد و روش‌های مقابله با آن

بهوضوحنمیتوانیم یک نگاشت یک به یک مجموعه با کاردینالیتی C_1 به مجموعه‌ای با کاردینالیتی C_2 داشته باشیم که $C_1 > C_2$. پس حتماً وقتی از تابع درهمساز h برای نگاشت مجموعه D به مجموعه L که $|D| > |L|$ استفاده می‌کنیم، حتماً عناصر متمایزی از D وجود دارند که یک عنصر واحد در L نگاشت شده‌اند.

به عنوان مثال، تابع درهمسازی ارائه شده در قسمت قبل، هر دو عدد ۲۰۹۸۷ و ۱۰۹۸۷ را به عدد ۹۸۷ 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 یا **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 [18]:

```
s.insert(10987)
print(s.exists(20987))
# Collision! 10987 % 10^4 = 20987 % 10^4 = 987
```

True

روشهای مقابله

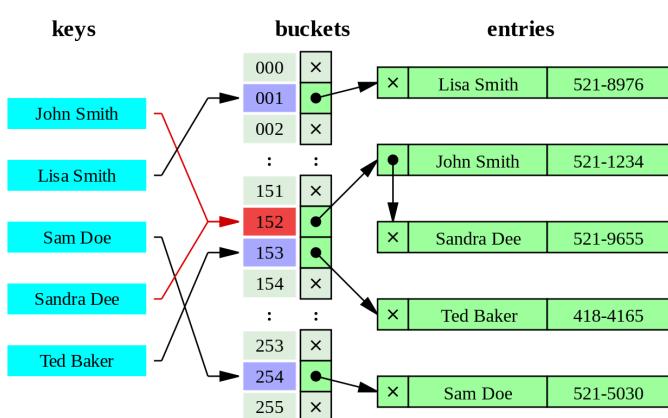
تا به حال سعی می‌کردیم تا توابعی پیدا کنیم که احتمال برخورد را کم کنیم. اما در هر صورت اگر بخواهیم تعداد زیادی عدد را در یک جدول با اندازه‌ی کوچک ذخیره کنیم، حتماً برخورد خواهیم داشت. برای حل این مشکل روش‌های زیادی وجود دارد از جمله:

- زنجیره‌سازی
- آدرس‌دهی آزاد

که در ادامه این دو روش را توضیح می‌دهیم.

زنجیره‌سازی

در این روش، هر خانه‌ی جدول را یک لیست پیوندی 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.



شکل ۳: زنجیره‌سازی در جدول درهمسازی

مشکل این روش داینامیک بودن سایز داده ساختار است.

اگر طول یک لیست پیوندی خیلی زیاد شد می‌توانیم از هش و یا درخت دودویی جست‌وجو به جای لیست پیوندی استفاده کنیم.

این روش شاید به نظر کارا نیاید اما دقت کنید که فرض کرده بودیم قرار نیست تعداد زیادی برخورد داشته باشیم.

کد این جدول درهمسازی تعیین یافته در ادامه آمده است:

In [6]:

```
class my_chained_set:
    table = [[] for _ in range(hash_size)]

    def insert(self, x):
        if not self.exists(x):
            self.table[hash_function(x)].append(x) # O(1)

    def exists(self, x):
        return x in self.table[hash_function(x)] # O(size of List)

    def remove(self, x):
        self.table[hash_function(x)].remove(x) # O(size of List)

s = my_chained_set()
s.insert(10)
s.insert(245)
print(s.exists(10))

s.insert(10987)
print(s.exists(987)) # right answer!
print(s.exists(10987))
```

True
False
True

آدرس دهی باز

در این روش، بر عکس روش قبل، اعداد را در همان خانه‌های جدول نگهداری می‌کنیم، با این تفاوت که اگر به خانه‌ی اصلی رفتیم و پر بود، به سراغ خانه‌ی (مشخص) دیگری می‌رویم. مثلاً 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

امکان 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.

البته در این روش اگر خود خانه خالی بود، حتماً به سراغ خانه‌ی بعدی نمی‌رویم، بلکه تابع درهمساز ما باید دو ورودی بگیرد $(i, h(x, i))$ و ابتدا ما به خانه‌ی $(0, h(x, 0))$ می‌رویم. اگر پر بود به سراغ $(1, h(x, 1))$ و $(2, h(x, 2))$ 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 : آیا الگوریتم گفته شده برای حالتی که یک یا چند عنصر را از داده ساختار حذف کرده‌ایم هم جواب می‌دهد؟ اگر به مشکل برミخورد راهی برای حل این مشکل ارائه دهید.

برای ساختن این تابع معمولاً یکی از سه روش زیر را استفاده می‌کنیم:

۱ - کاوش خطی

این روش تقریباً همان روش رفتن به خانه‌ی بعدی است. روشی که در آن ترتیب بررسی ثابت و معمولاً با قدم‌های به اندازه ۱ می‌باشد. به این صورت که اگر اعضای جدول اعداد صفر تا m باشند، تابع درهمساز جدید را بر اساس تابع درهمساز قبلی به صورت زیر می‌سازیم:

$$h(x, i) = (h'(x) + i) \bmod m$$

اما در این روش معمولاً مشکلی به نام دسته‌بندی اولیه (Primary Clustering) پیش می‌آید که یعنی دنباله‌های پشت‌سر هم طولانی ساخته شده و در نتیجه زمان جستجو افزایش می‌یابد.

امکان وجود برخورد همچنین در این روش وجود دارد.

۲ - کاوش مربعی

روشی که در آن ترتیب بررسی تابعی از درجه دوم می‌باشد. در این روش فرم تابع مورد استفاده به صورت زیر خواهد بود

$$h(x, i) = (h'(x) + c_1 i + c_2 i^2) \bmod m$$

اما باز هم اگر دو دسته تابع در هم‌ساز اولیه‌شان یکی باشد باز هم دنباله‌شان یکسان خواهد بود. به این مشکل نیز دسته‌بندی ثانوی (Secondary Clustering) می‌گویند.

۳ - درهم‌سازی دوتایی

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:

$$h(x, i) = (h_1(x) + i h_2(x)) \bmod m$$

می‌توان گفت به طور تقریبی هیچ برخوردی در این روش اتفاق نمی‌افتد.

بنام خدا

دانشگاه آزاد اسلامی شیراز - دانشکده مهندسی

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل نهم: تابع درهم‌سازی (قسمت دوم)

فهرست محتویات

- تابع درهم سازی برای رشته ها
- آرایه با سایز پویا
- پیاده سازی آرایه پویا در زبان های مختلف
- dict در پایتون
- اهمیت تابع درهم سازی

تابع درهم سازی برای رشته ها
مثال:

داده ساختاری طراحی کنید که عملیات زیر را انجام دهد.
رشته s را به داده ساختار اضافه کن. اگر قبل این رشته اضافه شده بود
اطلاع بده.

باید یک تابع hash برای یک رشته تعریف کنیم یکی از موسوم ترین روش hash یک رشته، در نظر گرفتن یک رشته مثل S به صورت یک عدد |S| بیتی است. و استفاده از روش باقی مانده تقسیم روی آن است.

In [1]:

```
P = 447
mod = 10**9 + 7
def h_string(s):
    result = 0
    for c in s:
        result = (result * P + ord(c)) % mod
        # ord(c): get integer of character c
    return result
```

در پایتون تقریبا به همین روش تابع hash پیاده سازی شده است:

In [3]:

```
print(hash("Shiraz!"),hash("shiraz"))
```

-3289647372329968000 -5252761008007385126

تابع $h(s)$ عدد زیر را برمی‌گرداند.

$$s = c_1 c_2 \dots c_k$$

$$h(s) = c_1 * P^{k-1} + c_2 * P^{k-2} + \dots + c_{k-1} * P + c_k$$

حال با استفاده از این تابع درهم سازی و استفاده از یکی از روش‌های مقابله با برخورد می‌توان مسئله را حل کرد.
 (البته احتمال برخورد آن قدر کم است که استفاده از داده ساختار BST هم کافی است.)

در روش اول از «زنجیره سازی» استفاده می‌کنیم.
 زمان اجرای این روش از $O(nk)$ است.

In [3]:

```
# Chained hashing used in previous session
hash_size = 10**4

def h(s):
    return h_string(s) % hash_size

class my_chained_set:
    def __init__(self):
        self.table = [[] for _ in range(hash_size)]

    def insert(self, x):
        if not self.exists(x):
            self.table[h(x)].append(x) # O(1)

    def exists(self, x):
        return x in self.table[h(x)] # O(size of list)

    def remove(self, x):
        self.table[h(x)].remove(x) # O(size of list)
```

In [4]:

```
from __future__ import print_function # for python2 users
#first solution : O(n*k)
htable1 = my_chained_set()
def add1(s):
    print(s, end=' ')
    if htable1.exists(s): # O(1)
        print("exists")
    else:
        print("inserted")
        htable1.insert(s) # O(1)

add1("Ali")
add1("vezvayi")
add1("Ali")
```

```
Ali inserted
vezvayi inserted
Ali exists
```

در روش دوم از set در پایتون استفاده می کنیم که خود از روش «آدرس دهی باز» استفاده می کند.

زمان اجرای این روش از $O(nk)$ است.

In [5]:

```
#second solution : O(n*k)
htable2 = set()
def add2(s):
    print(s, end=' ')
    if h(s) in htable2:    # O(1)
        print("exists")
    else:
        print("inserted")
        htable2.add(h(s)) # O(1)

# this works because the probability of collision is very low

add2("Ali")
add2("alipour")
add2("Ali")
```

```
Ali inserted
alipour inserted
Ali exists
```

روش بالا در $n \leq 10 * 5^k$ کار می کند چرا که احتمال برخورد خیلی پایین است.

$$(mod = 10^9 + 7)$$

در روش سوم می توان از BST استفاده کرد که زمان اجرایی $O(nk \lg n)$ را خواهد داشت.

تمرین

تابع درهم سازی پیاده سازی کنید که از قاعده زیر پیروی کند:
یک عدد ۱۶ بیتی در نظر بگیرید که در ابتدا ۰ است. سپس به ازای هر کاراکتر آن را یک بار شیفت دوری راست داده و سپس آن را با عدد ASCII آن کاراکتر جمع کند.

In [8]:

```
def test_hash(s):
    result = 0
    for ch in s:
        #insert code here!

    return result

print(test_hash("Hash") == 104)
```

True

آرایه با سایز پویا

اگر شما بخواهید آرایه ای که سایز متغیری داشته باشد، چه می کنید؟ در واقع ما به یک داده ساختار نیاز داریم که عملیات زیر را بتواند انجام دهد.

	Linked list	Array	Dynamic array
Indexing	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Insertion/deletion at end	$\Theta(1)$	N/A	$\Theta(1)$ amortize
Insertion/deletion in middle	$\Theta(1)$	N/A	$\Theta(n)$
Wasted space (average)	$\Theta(n)$	0	$\Theta(n)$
Space	$\Theta(n)$	n	$\Theta(n)$

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 می شود؟

یک روش پیشنهادی تغییر اندازه آرایه های پویا به مقدار زیادتر است؛ مثلا زمانی که آرایه پر شد، اندازه آن را دو برابر می کنیم. چرا با اینکار درج کردن به طور سرشکن از $O(n)$ است؟

A :

	0	
--	---	--

In [6]:

```
#We use Lists here, even though a List is a dynamic array
class DynamicArray:
    def __init__(self):
        self.a = [None] # capacity = len(self.a)
        self.size = 0
    def insert(self, x):
        if self.size == len(self.a):
            new = [None] * 2*len(self.a)
            for i in range(len(self.a)):
                new[i] = self.a[i]
            self.a = new
        self.a[self.size] = x
        self.size += 1
```

In []:

```
da = DynamicArray()  
print(da.size)  
print(da.a)
```

```
0  
[None]
```

In [22]:

```
da.insert(1)  
print(da.size)  
print(da.a)
```

```
1  
[1]
```

In [23]:

```
da.insert(2)  
print(da.size)  
print(da.a)
```

```
2  
[1, 2]
```

In [24]:

```
da.insert(3)  
print(da.size)  
print(da.a)
```

```
3  
[1, 2, 3, None]
```

In [25]:

```
da.insert(4); da.insert(5)  
print(da.size)  
print(da.a)
```

```
5  
[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 مواقعي ظرفیت آن را کم کرد.

اگر وقتی ظرفیت آرایه دو برابر سایز آرایه باشد، اندازه آن را کاهش دهیم، مشکل رفع

می شود؟

چه راه حلی پیشنهاد می دهید؟

پیاده سازی Dynamic Array در زبان های مختلف

ضریب رشد پیاده سازی

Java ArrayList	1.5 (3/2)
Python PyListObject (list)	1.125 (9/8)
G++ 5.2.0 vector	2
Facebook folly/FBVector	1.5 (3/2)

dict در پایتون

داده ساختار dict یک نوع دیکشنری است.

برخلاف رشته‌ها، لیست‌ها اندیس داده‌ها در دیکشنری به دلخواه برنامه‌نویس مشخص می‌شود (که این اندیس‌ها کلید - key گفته می‌شود) این کلید‌ها می‌توانند از هر جنسی باشند و لزومی ندارد که عدد صحیح باشند.

In [26]:

```
grade = {20:0, 3.14:"20"}  
grade["eshagh"] = 10  
grade["bagher"] = 1  
grade["salim"] = "???"  
print(grade)  
  
{20: 0, 3.14: '20', 'eshagh': 10, 'bagher': 1, 'salim': '???'}
```

دیکشنری در پایتون به صورت «آدرس دهی باز» پیاده سازی شده است.

در ادامه به نحوه پیاده سازی دیکشنری در پایتون می‌پردازیم.

In [27]:

```
d = {}
```

دیکشنری در واقع یک لیست است.
یک دیکشنری خالی، یک لیست با طول 8 است.

Idx	Hash	Key	Value
000			
001			
010			
011			
100			
101			
110			
111			

هش هر کلید تولید می شود تا بتوان آن را در لیست ذخیره کرد.
می توان هش استفاده شده توسط پایتون را با استفاده از تابع hash مشاهده کرد.

In [29]:

```
# gives binary of a number
def bits(n):
    n += 2**32
    return bin(n)[-32:] # remove '0b'
print(bits(2), end='\n\n')

print(bits(hash("John Snow")), "John Snow")
print(bits(hash("Grades are not important")), "Grades are not important")
print(bits(hash((1, 2, 3))), (1, 2, 3)) # tuples are also hashable
```

00000000000000000000000000000010

01100011001110100110100011101001 John Snow
100100010100100010100001011010111 Grades are not important
1110100101101111111001101001111 (1, 2, 3)

هش دو رشتہ مشابه می تواند تفاوت زیادی داشته باشد.

In [30]:

```
k1 = bits(hash('Fail'))
k2 = bits(hash('Faal'))
diff = ('^' if a!=b else ' ') for a,b in zip(k1, k2)) # points out unequal bits
print(k1)
print(k2)
print(''.join(diff)) # diff is not a string per se
```

```
1110110101001010101010101100010
10100100001100001010101000100100
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
```

اندیس یک کلید برابر n بیت سمت راس هش آن کلید است.

در ابتدا $n = 3$ است. $2^n = \text{size_of_hash_table}$

In [31]:

```
d["ali"] = 6
b = bits(hash("ali"))
print(b)
print(b[-3:]) # Last 3 bits = 8 combinations
```

```
11000010001100001000101010110100
100
```

Idx	Hash	Key	Value
000			
001			
010			
011			
100			
101			
110			
111	= ...00000111	'ali'	9

In [32]:

```
d["reza"] = 9
b = bits(hash("reza"))
print(b)
print(b[-3:]) # Last 3 bits = 8 combinations
```

```
01000000110010011101111010001011
011
```

Idx	Hash	Key	Value
000			
001			
010			
011			
100	= ...11001100	'reza'	9
101			
110			
111	= ...00000111	'ali'	6

In [33]:

```
d["nasim"] = 8
b = bits(hash("nasim"))
print(b)
print(b[-3:]) # Last 3 bits = 8 combinations
```

```
01111010011100110101110111011001
001
```

Idx	Hash	Key	Value
000			
001			
010			
011			
100	= ...11001100	'reza'	9
101	= ...11010101	'nasim'	8
110			
111	= ...00000111	'ali'	6

In [2]:

```
d = {'ali': 6, 'reza': 9, 'nasim': 8}
print(d.keys())
d = {'nasim': 8, 'reza': 9, 'ali': 6}
print(d.keys())
```

```
dict_keys(['ali', 'reza', 'nasim'])
dict_keys(['nasim', 'reza', 'ali'])
```

Idx	Hash	Key	Value
000			
001			
010			
011			
100	= ...11001100	'reza'	9
101	= ...11010101	'nasim'	8
110			
111	= ...00000111	'ali'	6

Idx	Hash	Key	Value
000			
001			
010			
011			
100	= ...11001100	'reza'	9
101	= ...11010101	'nasim'	8
110			
111	= ...00000111	'ali'	6

چرا این دو دیکشنری با اینکه ترتیب متفاوتی دارند، در پایتون ۲ خروجی یکسانی دارند؟ چرا در پایتون ۳ به این صورت نیست؟

با چاپ کردن دیکشنری در پایتون ۲ ترتیب خروجی همان ترتیب جدول درهم سازی است. (نه ترتیبی که مقدارها به دیکشنری اضافه شده اند)
اما در پایتون ۳ اینطور نیست!

برخورد در dict

اگر برخورد داشته باشیم باید چه کنیم؟

می توان از کاوش خطی استفاده کرد که از نظر زمانی اصلاً ایده آل نیست، چرا که دنباله های پشت سر هم در جدول ساخته می شود و در طول زمان جدول کند می شود.
در پایتون براساس $n - 32$ بیت سمت چپ که از آن ها استفاده نشده یک کاوش (که تقریباً رندوم است) ایجاد می شود. (به دلیل پیچیدگی این کاوش، آن را بررسی نمی کنیم)

In [35]:

```
d = {}
```

In [36]:

```
d["arya"] = 666
b = bits(hash("arya"))
print(b)
print(b[-3:]) # Last 3 bits = 8 combinations
```

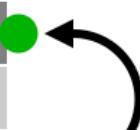
```
0110001100010101011000000001010
010
```

Idx	Hash	Key	Value
000			
001			
010			
011	= ...01111011	'arya'	666
100			
101			
110			
111			

In [37]:

```
d["john"] = 858
b = bits(hash("john"))
print(b)
print(b[-3:]) # Last 3 bits = 8 combinations
```

```
01111001100011111100111110011111
111
```

Idx		Hash	Key	Value	
000					
001	≠	...00111 011	'john'	858	 
010					
011	=	...01111 011	'arya'	666	 
100					
101					
110					
111					

In [38]:

```
d["snow"] = 1
b = bits(hash("snow"))
print(b)
print(b[-3:]) # last 3 bits = 8 combinations
```

00111100110101111000000000000000011
011

Idx		Hash	Key	Value
000				
001	≠	...00111 011	'john'	858
010				
011	=	...01111 011	'arya'	666
100				
101	=	...10101 101	'snow'	1
110				
111				

In [39]:

```
d["alive"] = 2
b = bits(hash("alive"))
print(b)
print(b[-3:]) # last 3 bits = 8 combinations
```

0000000000110011001101100100000
000

Idx		Hash	Key	Value
000				
001	≠	...00111011	'john'	858
010				
011	=	...01111011	'arya'	666
100				
101	=	...10101101	'snow'	1
110	=	...10001110	'alive'	2
111				

In [40]:

```
d["dead"] = 3
b = bits(hash("dead"))
print(b)
print(b[-3:]) # Last 3 bits = 8 combinations
```

01101000001010000111111101110111
111

Idx		Hash	Key	Value
000	≠	...10001110	'dead'	3
001	≠	...00111011	'john'	858
010				
011	=	...01111011	'arya'	666
100				
101	=	...10101101	'snow'	1
110	=	...10001110	'alive'	2
111				

In [2]:

```
d = {'arya': 666, 'snow': 1, 'john': 858, 'alive': 2, 'dead': 3}
print(d.keys())
e = {'snow': 1, 'john': 858, 'arya': 666, 'dead': 3, 'alive': 2}
print(e.keys())
print(d == e) # ???
```

dict_keys(['arya', 'snow', 'john', 'alive', 'dead'])
dict_keys(['snow', 'john', 'arya', 'dead', 'alive'])
True

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	≠ ...00111011	'john'	858
010			
011	= ...01111011	'arya'	666
100			
101	= ...10101101	'snow'	1
110	= ...10001110	'alive'	2
111			

Idx	Hash	Key	Value
000			
001			
010	≠ ...10001110	'alive'	2
011	= ...00111011	'john'	858
100			
101	= ...10101101	'snow'	1
110	= ...10001110	'dead'	3
111	≠ ...01111011	'arya'	666

چرا این دو دیکشنری با اینکه مقادیر یکسانی دارند، ترتیب متفاوتی دارند؟

جست و جو در dict

جست و جو هم شبیه به به درج کردن عمل می کند.
تا زمانی که سطر خالی پیدا نکرده یا کلید مورد نظر را پیدا نکرده به جست و جو ادامه می دهد.

In [42]:

```
b = bits(hash("fire"))
print(b)
print(b[-3:]) # Last 3 bits = 8 combinations
print("fire" in d)
```

000010010110011001100010100111101

101

False

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	≠ ...00111011	'john'	858
010			
011	= ...01111011	'arya'	666
100			
101	= ...10101101	'snow'	1
110	= ...10001110	'alive'	2
111			

حذف کردن در dict

برای حذف کردن یک کلید باید چه کرد؟
آیا خالی کردن سطر کلید مورد نظر کافی است؟

حذف "snow"

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	≠ ...00111011	'john'	858
010			
011	= ...01111011	'arya'	666
100			
101	= ...10101101	'snow'	1
110	= ...10001110	'alive'	2
111			

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	≠ ...00111011	'john'	858
010			
011	= ...01111011	'arya'	666
100			
101			
110	= ...10001110	'alive'	2
111			

جست و جو "dead"

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	≠ ...00111011	'john'	858
010			
011	= ...01111011	'arya'	666
100			
101	= ...10101101	'snow'	1
110	= ...10001110	'alive'	2
111			

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	≠ ...00111011	'john'	858
010			
011	= ...01111011	'arya'	666
100			
101			
110	= ...10001110	'alive'	2
111			

اگر فقط سطر را خالی کنیم، با جست و جو "dead" به نتیجه می‌رسیم که "dead" در جدول وجود ندارد، که غلط است.
چه پیشنهادی برای رفع این مشکل دارید؟

می‌توان به جای خالی کردن سطر مقدار "dummy" را در آن قرار دهیم.

In [43]:

```
d = {'arya': 666, 'john': 858, 'snow':1, 'alive': 2, 'dead':3}
del d["snow"]
print("dead" in d)
```

True

حذف "dead" و جست و جو "snow"

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	≠ ...00111011	'john'	858
010			
011	= ...01111011	'arya'	666
100			
101	!	<dummy>	
110	= ...10001110	'alive'	2
111			

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	≠ ...00111011	'john'	858
010			
011	= ...01111011	'arya'	666
100			
101	!	<dummy>	
110	= ...10001110	'alive'	2
111			

با اینکار در مرحله جست و جو "dead" وقتی به سطر 101 می رسیم، با توجه به اینکه hash آن خالی است ولی این سطر مقدار دارد از روی آن می گذریم.

In [44]:

```
d = {'arya': 666, 'john': 858, 'snow':1, 'alive': 2, 'dead':3}
del d["snow"], d['john'], d['arya'], d['alive']
print("dead" in d)
```

True

جست و جو

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	!	<dummy>	
010			
011	!	<dummy>	
100			
101	!	<dummy>	
110	!	<dummy>	
111			

Idx	Hash	Key	Value
000	≠ ...10001110	'dead'	3
001	!	<dummy>	
010			
011	!	<dummy>	
100			
101	!	<dummy>	
110	!	<dummy>	
111			

پویا بودن سایز dict

وقتی دیکشنری تقریبا پر شود چه مشکلی پیش می آید؟
برای حل این مشکل چه راه حلی پیشنهاد می کنید؟

دیکشنری در پایتون خود مانند یک آرایه پویا عملی می کند.
در پایتون وقتی 2/3 جدول پر شده یک دیکشنری جدید ساخته می شود و تمام
اعضای دیکشنری داخل دیکشنری جدید ریخته می شوند.
سایز دیکشنری جدید در پایتون به صورت است.

`When < 50k entries, size * = 4`

`When > 50k entries, size * = 2`

این کار برای صرفه جویی در حافظه در مقدار های بالا است.

اثبات کنید جدول درهم سازی با الگوریتم گفته شده (معروف به درهم سازی پویا) همچنان عملکردی از $O(1)$ دارد.

فایل words یک دیکشنری حاوی تمام کلمات انگلیسی است.

از آن برای اضافه کردن کلمات به dict استفاده می کنیم.

In [45]:

```
wordfile = open('files/words')
text = wordfile.read()
words = [ w for w in text.split()
          if w == w.lower() and len(w) < 6 ]
# words: words which are lower case and
#         have length less than 6
words[:5]
```

Out[45]:

```
['a', 'abaci', 'aback', 'abaft', 'abase']
```

In [46]:

```
d = dict.fromkeys(words[:5])
# collision rate 40%
# but now 2/3 full - on verge of resizing!
```

Idx	=	Hash	Key	Value
000	=	...11100000	'a'	None
001	=	...00100001	'aback'	None
010				
011	=	...00100011	'abaci'	None
100	≠	...00011111	'abase'	None
101				
110				
111	≠	...01110001	'abaft'	None

In [47]:

```
d["izadi"] = None  
# Resizes x4 to 32, collision rate drops to 0%
```

00000	-	_011000111100000	'a'	None
00001	-	_110010100100001	'aback'	None
00010				
00011	-	_110010100100011	'abaci'	None
00100				
00101				
00110				
00111				
01000				
01001	-	_101000011101010	'izadi'	None
01011				
01100				
01101				
01110				
10000				
10001	-	_001101001110001	'abaft'	None
10010				
10011				
10100				
10101				
10110				
10111				
11000				
11001				
11010				
11011				
11100				
11101				
11110	-	_000100100011110	'abase'	None
11111				

In [48]:

```
d = dict.fromkeys(words[:21])  
# 2/3 full again - collision rate 29%
```

00000	-	_011000111100000	'a'	None
00001	-	_110010100100001	'aback'	None
00010	-	_110011100000010	'abet'	None
00011	-	_110010100100011	'abaci'	None
00100	-	_000001011100100	'abets'	None
00101				
00110	#	_011101110110000	'ably'	None
00111	#	_10111111100010001	'beam'	None
01000	-	_001110010101000	'abbés'	None
01001				
01010				
01011				
01100	-	_011101110101000	'able'	None
01101	#	_110011100010010	'abed'	None
01110	-	_1100100010110	'abbey'	None
01111	-	_110110100101111	'abler'	None
10000	#	_001110011001110	'abbé'	None
10001	-	_0011100100111001	'abaf'	None
10010	-	_000100100010010	'abash'	None
10011				
10100				
10101	-	_111111110010101	'abbot'	None
10110	-	_100001011010110	'abate'	None
10111	#	_111100010000000	'abode'	None
11000				
11001				
11010				
11011				
11100				
11101	-	_001111111111101	'abhor'	None
11110	#	_000101010101110	'abide'	None
11111	-	_000100100011111	'abase'	None

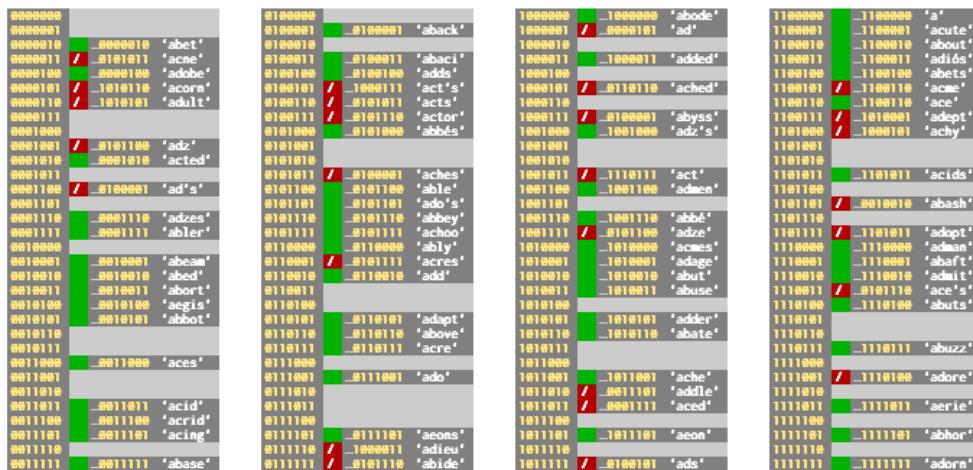
In [49]:

```
d['abode'] = None  
# Resizes x4 to 128, collision rate drops to 9%
```



In [50]:

```
d = dict.fromkeys(words[:85])  
# 2/3 full again - collision rate 33%
```



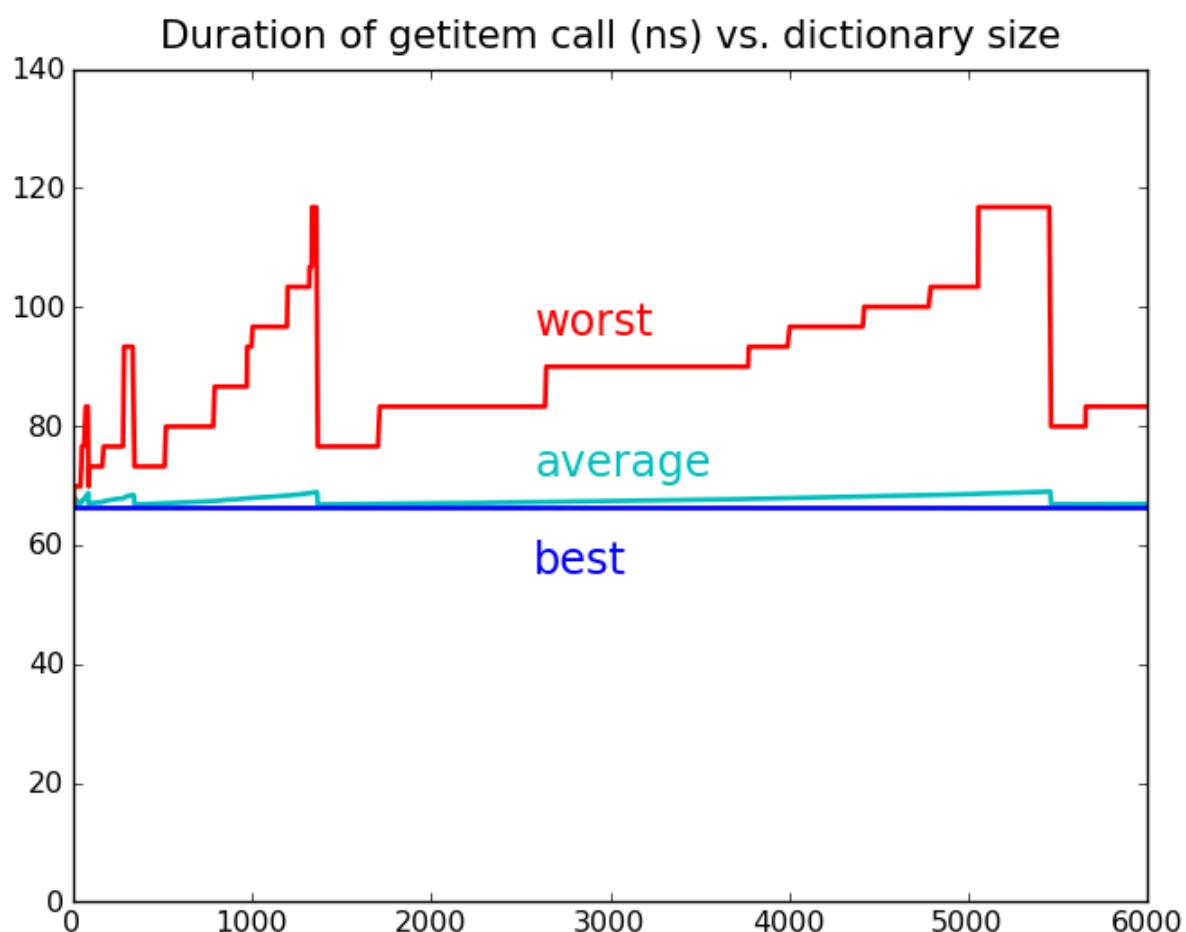
تمرین

اسم و شماره دانشجویی خودتان را به دیکشنری اضافه کنید. سپس آن را جست و جو کرده و حذف نمایید.

In [4]:

```
#insert code here!
```

نمودار زیر را برای زمان دسترسی به اعضا می بینید.
با افزایش اندازه دیکشنری، تعداد برخورد های میانگین از یک ضریب ثابتی بیشتر نمی شود.



اهمیت تابع درهم سازی

اگر تابع درهم سازی به خوبی پیاده سازی نشده باشد، چه مشکلی پیش می آید؟

در پایتون اعداد به طور خیلی ساده هش می شوند به طوری که

$$\text{hash}(x) = x$$

و با دانستن این اطلاعات می توان تعداد زیادی برخورد تولید کرد.

In [51]:

```
print(hash(5))
```

5

In [52]:

```
threes = {3: 1, 3+8: 2, 3+16: 3, 3+24: 4, 3+32: 5}
```

Idx	Hash	Key	Value
000	≠ ...00001011	11	2
001	≠ ...00010011	19	3
010			
011	= ...00000011	3	1
100			
101			
110	≠ ...00011011	27	4
111	≠ ...00100011	35	5

→ **x** → **x**

در جاوا 1.1 برای صرفه جویی در زمان از تابع زیر استفاده می شد:

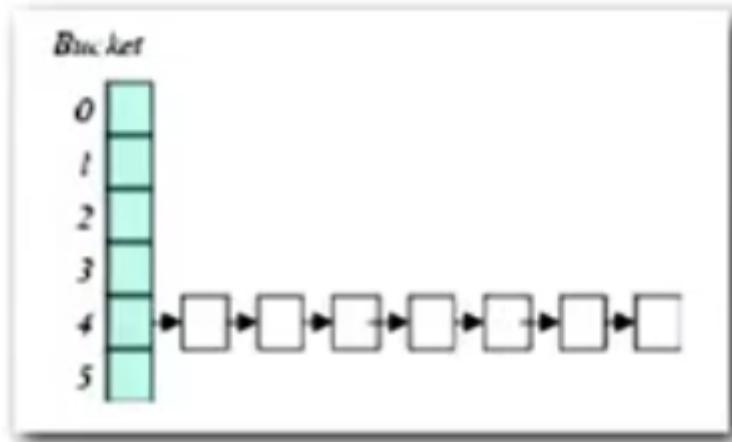
```
public int hashCode()
{
    int hash = 0
    int skip = Math.max(1, length() / 8);
    // skip every 8 characters
    for (int i = 0; i < length(); i += skip)
        hash = s[i] + (37 * hash);
    return hash
}
```

با اینکار رشته های زیر های یکسانی خواهند داشت.

```
http://www.cs.princeton.edu/introcs/13loop>Hello.java  
http://www.cs.princeton.edu/introcs/13loop>Hello.class  
http://www.cs.princeton.edu/introcs/13loop>Hello.html  
http://www.cs.princeton.edu/introcs/12type/index.html
```

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

اگر از روش زنجیره سازی استفاده شده باشد، همه ی این رشته ها در یک زنجیره ذخیره می شوند.



با دانستن تابع هش می توان حمله های زیاد به سرور ها کرد (معمولاً حمله های (Denial of Service

مثلا در کرنل linux 2.4.20 اگر نام فایل ها به صورت خاصی ذخیره می شدند، سیستم crash می کرد.

بنام خدا

دانشگاه آزاد اسلامی شیراز - دانشکده مهندسی

داده‌ساختارها و الگوریتم‌ها

ترم اول سال تحصیلی 1400-1401

فصل نهم: توابع درهمسازی (قسمت سوم)

فهرست محتویات

- مقدمه
- مثال: شبکه‌ی اجتماعی کتابخوان‌ها
- درهمسازی میانی
- شبیه‌ترين کاربر (نزدیک ترین همسایه)
- درهمسازی حساس به شبات
- شبیه‌ترين کتاب!

مقدمه

روش‌های درهمسازی برای کاهش ابعاد داده به منظور مقایسه‌ی راحت‌تر و سریع‌تر آن به وجود آمده‌اند. در دو بخش قبلی توابع درهمسازی را بررسی کردیم که به احتمال بالایی برای هر دو داده‌ی متفاوت hash 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 یا صفحه‌های وب دقیقاً یکسان یا ذخیره‌سازی امن رمزها کارآمد هستند.

در این بخش توابع درهمسازی را بررسی می‌کنیم که برای داده‌های نزدیک (نسبت به یک تعریف مشخص از فاصله یا شباهت)، به احتمالاً بالایی hash یکسان تولید می‌کنند. این توابع در کاربردهایی مثل پیدا کردن صفحه‌های وب با محتوای نزدیک به هم یا پیدا کردن صاحب اثر انگشت یا پیشنهاد دادن دوست به کاربران در شبکه‌های اجتماعی کارآمد هستند!

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 کاربردی 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؟).

یکی از تعاریفی که در عمل خوب کار می‌کند شاخص Jaccard است که شباهت دو مجموعه‌ی 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 کرده‌اند تقسیم بر تعداد کل کتاب‌هایی که خوانده‌اند است.

تمرین: برای دو زیرمجموعه‌ی m عضوی تصادفی از اعداد ۱ تا n امید ریاضی شاخص Jaccard را محاسبه کنید.

In [1]:

```
n = 4    # number of users
m = 13   # number of books

data = [[1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
        [1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1],
        ]

def jaccard(A, B):
    cap = 0.0
    cup = 0.0
    for i in range(0, len(A)):
        cap = cap + (A[i] and B[i])
        cup = cup + (A[i] or B[i])
    return cap / cup

print(jaccard(data[0], data[2]))
print(jaccard(data[0], data[1]))
```

0.25
0.4

چالش‌ها

ذخیره کردن اطلاعات کاربران در داده‌ساختارهای ساده مثل آرایه‌ی صفر و ۱ که در بخش قبل به آن اشاره کردیم در عمل ممکن است بسیار پر هزینه باشد. برای مثال اگر ۱ میلیون کاربر و ۱۰ هزار کتاب در سیستم داشته باشیم، حجم اطلاعات نمایش جدولی حدود 40 ترابایت می‌شود که بارگزاری آن به صورت یکجا در حافظه‌ی اصلی

کامپیوترهای معمولی (یا حتی کامپیوترهای بسیار قدرتمند) ممکن نیست. جدای از مشکل حافظه، محاسبه‌ی شباهت Jaccard دو کاربر از مرتبه‌ی $O(m)$ زمان لازم دارد و به همین ترتیب پیدا کردن پرشباخت‌ترین کاربر برای هر کاربر مشخص شده کار بسیار زمان‌بری است و از مرتبه‌ی $O(nm)$ 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 ذخیره‌ی اطلاعات و هم زمان کمتری برای پاسخ دادن به درخواست‌ها نیاز داشته باشد.

جواب تقریبی

پیدا کردن جواب دقیق در زمان و فضای مناسب برای مسئله‌ی شبیه‌ترین فرد یا نزدیک‌ترین نقطه (در ابعاد بالا)، بسیار سخت است و مدت زیادی بدون جواب باقی‌مانده.

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؟

درهمسازی میانی

ایده‌ی اصلی درهمسازی میانی استفاده از تعدادی تابع مستقل درهمساز مثل (x) h_i است که به هر مجموعه یک عدد صحیح نسبت می‌دهد، به طوری که به ازای هر $A \neq B$ احتمال تساوی $(B) = h_i(A)$ برابر با شباهت A و B در شاخص Jaccard باشد. اگر تعداد کافی از این تابع‌های درهمساز داشته باشیم می‌توانیم به جای هر A ، مقدار $(A)_i$ را نگهداری کنیم. با توجه به قانون اعداد بزرگ، با

افزایش تعداد $h_i(A)$ ها برای هر دو مجموعه‌ی مختلف نسبت تعداد hash های برابر به کل تعداد hash ها بسیار نزدیک به شباهت Jaccard آنها خواهد بود.

درهمسازی میانی در مثال شبکه اجتماعی کتابخوان‌ها

فرض کنید کتاب‌ها را با شماره‌های 1 تا m شماره‌گذاری کرده‌ایم. مجموعه‌ی کتاب‌هایی که یک کاربر مطالعه کرده‌است را A در نظر بگیرید، یک تابع مناسب برای درهمسازی میانی تابع زیر است:

$$h(A) = \min_{x \in A} x$$

یعنی کمترین شماره بین شماره‌ی کتاب‌هایی که این کاربر مطالعه کرده. 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). ثابت کنید احتمال $h(A) = h(B)$ برابر است با $\frac{|A \cap B|}{|A \cup B|}$.

In [2]:

```
def min_hash(A):
    for i in range(0, len(A)):
        if A[i] == 1:
            return i
    return -1

print min_hash(data[0])
print min_hash(data[1])
print min_hash(data[2])
```

0

1

0

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.

پس برای محاسبه‌ی hash اطلاعات کاربران، می‌توانیم در ابتدا تعدادی جایگشت تصادفی انتخاب کنیم و نسبت به هر کدام تابع $h(A)$ را محاسبه کنیم.

In [3]:

```
import numpy

def min_hash(A, p):
    for i in range(0, len(A)):
        if A[p[i]] == 1:
            return i
    return -1

def approximated_jaccard(A, B):
    both = 0
    for i in range(0, len(A)):
        both = both + (A[i] == B[i])
    return 1.0 * both / len(A)

P = [numpy.random.permutation(n),
      numpy.random.permutation(n),
      numpy.random.permutation(n),
      numpy.random.permutation(n),
      numpy.random.permutation(n)] # random permutations

hashed_data = []

for i in range(0, n):
    res = []
    for j in range(0, len(P)):
        res.append(min_hash(data[i], P[j]))
    hashed_data.append(res)

print hashed_data

sum_error = 0
for i in range(0, n):
    for j in range(0, n):
        print(i, j, abs(jaccard(data[i], data[j]) - approximated_jaccard(hashed_data[i], hashed_data[j])))
        sum_error += abs(jaccard(data[i], data[j]) - approximated_jaccard(hashed_data[i], hashed_data[j]))
print('average error = %.2f' % (sum_error / n / n))
```

```
[[0, 1, 0, 1, 0], [1, 3, 3, 3, 0], [0, 1, 0, 1, 1], [3, 0, 1, 0, 3]]
(0, 0, 0.0)
(0, 1, 0.2)
(0, 2, 0.55)
(0, 3, 0.1111111111111111)
(1, 0, 0.2)
(1, 1, 0.0)
(1, 2, 0.0)
(1, 3, 0.0)
(2, 0, 0.55)
(2, 1, 0.0)
(2, 2, 0.0)
(2, 3, 0.0)
(3, 0, 0.1111111111111111)
(3, 1, 0.0)
(3, 2, 0.0)
(3, 3, 0.0)
average error = 0.11
```

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 تصادفی ای را پیدا کنید که میانگین خطأ با در نظر گرفتن داده‌های hash شده به جای داده‌های اصلی حداقل ۰.۰۵ شود.

In []:

```
import random

n = 500
m = 1000

random_data = []
for user in range(0, n):
    d = []
    for book in range(0, m):
        r = random.uniform(0, 1)
        if r <= 0.005:
            d.append(1)
        else:
            d.append(0)
    random_data.append(d)

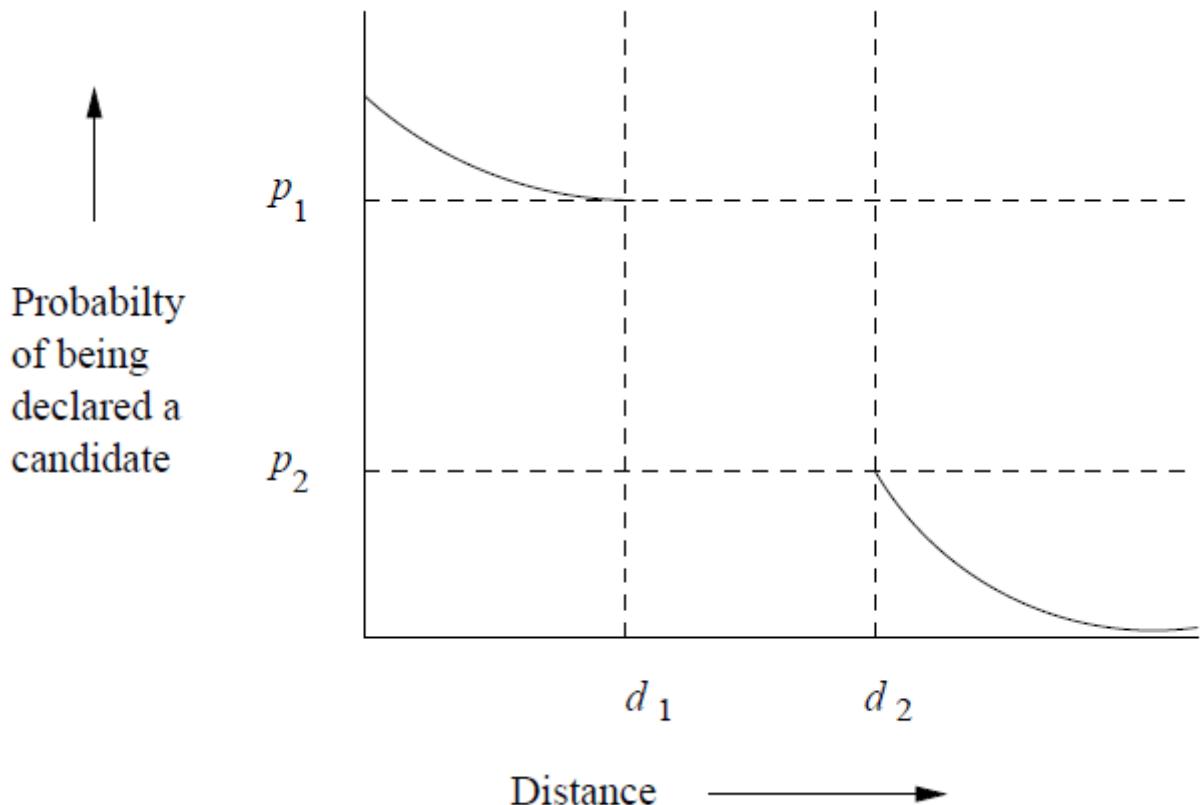
# complete this code and find appropriate number of random permutations!
```

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؟

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 ولی هنوز پیدا کردن نزدیک‌ترین کاربر به یک کاربر دلخواه دست کم از مرتبه‌ی $O(n)$ است و هنوز زمانبر است.

یک ایده چند بار استفاده از روش درهمسازی میانی است! با این کار به احتمال بالا، بعد از چند مرحله کاربرهایی که به هم شبیه هستند در یک دسته قرار می‌گیرند.

In [4]:

```
# TODO:  
# Example needed here.
```

شبیه‌ترین کتاب!

در بخش‌های قبل این درسنامه سعی کردیم نزدیک‌ترین کاربر به هر 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 شbahت دو کتاب یا دو رشته‌ی متنی چیست؟ تا اینجا فقط با معیار شbahت Jaccard آشنا شدیم. آیا می‌توانیم برای تعیین شbahت دو کتاب یا دو رشته از معیار Jaccard استفاده کنیم؟

کیسه‌ی کلمات!

می‌توانیم از روی هر متن یک مجموعه‌ی نظری شامل کلمات آن متن بسازیم. با توجه به این که مجموعه‌ی کلماتی که در دو متن متفاوت با محتوای شبیه به هم استفاده می‌شوند اشترآکات بیشتری از مجموعه کلمات دو متن با موضوعات متفاوت دارند، شاخص Jaccard مجموعه‌های نظری دو متن معیار قابل قبولی از شابهت 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 و دو تا با موضوع فیلم‌سازی را مقایسه می‌کنیم.

In [20]:

```
# 0.txt, 1.txt about LSH
# 2.txt, 3.txt about Filmmaking

all_words = set()
book_words = []
data = []

for i in range(0, 4):
    with open('./src/%d.txt' % i) as f:
        set_of_words = set([word for line in f for word in line.replace('.', ' ').split()])
    book_words.append(set_of_words)
    all_words = all_words.union(set_of_words)

for i in range(0, 4):
    d = []
    for w in all_words:
        if w in book_words[i]:
            d.append(1)
        else:
            d.append(0)
    data.append(d)

for i in range(0, 4):
    sim = []
    for j in range(0, 4):
        sim.append(jaccard(data[i], data[j]))
    print("%.2f" * len(sim) % tuple(sim))
```

1.00 0.10 0.08 0.09
0.10 1.00 0.06 0.05
0.08 0.06 1.00 0.10
0.09 0.05 0.10 1.00

نتایج خیلی خوب نیست! اگر کلمات پر تکرار و کلمات کم تکرار وزن متفاوتی داشته باشند نتایج به مراتب بهتر می‌شود. کد قسمت قبل را جوری تغییر دهید که نسبت به تکرار کلمات حساس باشد.

همچنین می‌توانیم با حذف کلمات و علاوه‌ی تکراری و بدون ارزش معنایی مثل حروف ربط یا نقطه به نتیجه‌ی بهتری بررسیم.

گروه کلمات به جای تک کلمه

یک ایده برای افزایش دقت شاخص Jaccard اضافه کردن همهٔ ترکیب‌های حاصل از دو یا چند کلمهٔ مجاور به جای کلمات تکی است. می‌توانید این بهبود عملکرد را توجیه کنید؟

منابع

1. [Mining of Massive Dataasets by Anand Rajaraman and Jeffrey D. Ullman](http://www.mmds.org/) (<http://www.mmds.org/>).
2. [Nearest Neighbors in High-dimensional Spaces by Piotr Indyk](https://people.csail.mit.edu/indyk/39.ps) (<https://people.csail.mit.edu/indyk/39.ps>).

In []: