تمرین دوم طراحی الگوریتم‌ها

اشکان شکیبا (۹۹۳۱۰۳۰)

سوال اول

مرتب‌سازی‌های پایدار:

merge sort, insertion sort, bubble sort, counting sort, radix sort

توضیح الگوریتم‌ها:

Merge sort:

Input: [3d, 2c, 2b, 1a] Output: [1a, 2c, 2b, 3d]

در هنگام مرج زیرآرایه‌ها، اگر دو عضو برابر باشند، عضو لیست چپ (که در ورودی نیز زودتر آمده) انتخاب و افزوده می‌شود.

Insertion sort:

Input: [3d, 2c, 2b, 1a] Output: [1a, 2c, 2b, 3d]

در هنگام پیمایش اعضا در حلقه، اگر دو عضو برابر باشند، به دلیل شرط key < a[i]، عضو دوم پس از عضو اول قرار می‌گیرد.

Bubble sort:

Input: [3d, 2c, 2b, 1a] Output: [1a, 2c, 2b, 3d]

دلیل پایداری الگوریتم این است که اعضای مجاور با هم مقایسه می‌شوند و در صورت برابری، جابجا نمی‌شوند.

Counting sort:

Input: [3d, 2c, 2b, 1a] Output: [1a, 2c, 2b, 3d]

بروزرسانی آرایه count به ترتیب اعضا انجام می‌شود و دو عضو برابر با همان ترتیب شمرده می‌شوند.

Radix sort:

Input: [2, 112, 43, 58] Output: [2, 43, 58, 112]

چون این الگوریتم رقم به رقم مرتب می‌کند و برای این موضوع از الگوریتم‌های counting استفاده می‌کند، پایدار است.

همچنین bucket sort نیز می‌تواند پایدار باشد، بسته به اینکه در مرتب‌سازی باکت‌ها از الگوریتمی پایدار استفاده کنیم.

سوال دوم

پیچیدگی زمانی counting sort بسته به بازه و محدوده اعداد و تنوع آنها دارد.

بهترین حالت این است که بازه اعداد محدود باشد. حالت میانگین این است که بازه تقریبا محدودی از اعداد باشد. بدترین حالت هم این است که باز گسترده و زیادی از اعداد باشد.

در quick sort بهترین حالت این است که زمانی که بخش‌بندی به بهترین شکل انجام شود. حالت میانگین این است که به طور میانگین چگونه بخش‌بندی انجام شود. بدترین حالت هم زمانی‌ست که بخش‌بندی به درستی انجام شود.

در heap sort پیچیدگی زمانی هر حالتی برابر O(n logn) است. که n برای درست کردن max heap است و logn برای root

	بهترین حالت	حالت میانگین	بدترین حالت	پیچیدگی مکانی
counting sort	O(n+k)	O(n+k)	O(n+k)	O(k)
quick sort	O(n logn)	O(n logn)	$O(n^2)$	O(n)
heap sort	O(n logn)	O(n logn)	O(n logn)	O(1)

سوال سوم

در ابتدا می‌توان الگوریتم‌های غیربهینه مثل insertion, bubble و... را از گزینه‌ها حذف کرد.

merge sort برای داده‌های با تعداد زیاد مناسب است ولی از نظر حجم حافظه مصرفی مناسب نیست. اما به لحاظ زمانی پیچیدگی بهترین و بدترین حالت آن O(n logn) است.

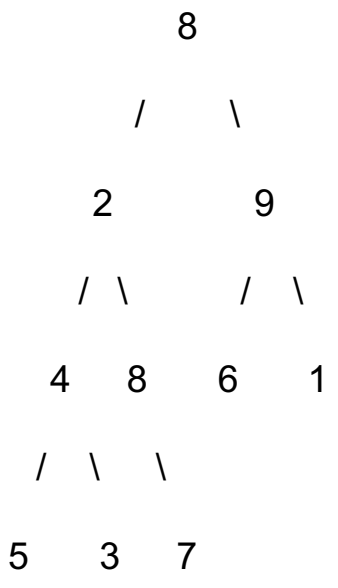
quick sort برای داده‌های با تعداد زیاد مناسب است و درجا است در نتیجه از نظر پیچیدگی مکانی نیز بسیار مناسب است. میانگین پیچیدگی زمانی آن O(n logn) است. بهترین حالت O(n) و بدترین حالت آن $O(n^2)$ است.

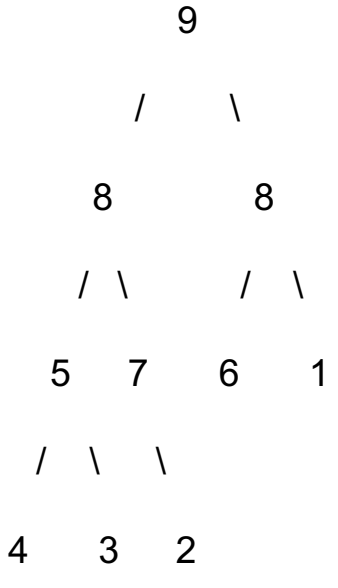
heap sort برای هر تعدادی مناسب است ولی برای تعداد داده‌های زیاد مناسب‌تر است و چون درجا است از نظر پیچیدگی مکانی نیز بسیار مناسب است. بهترین و بدترین حالت آن O(n logn) است.

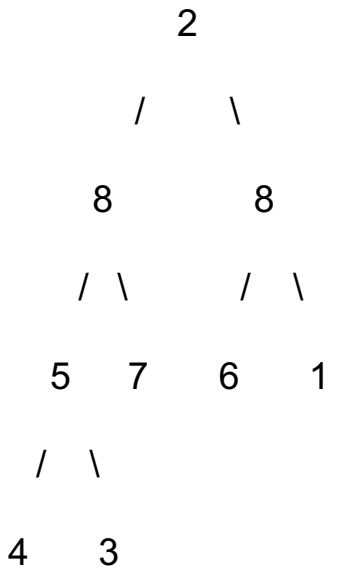
counting sort برای حالتی که تنوع داده‌ها کم باشد مناسب است ولی از نظر پیچیدگی مکانی بهینه نیست.

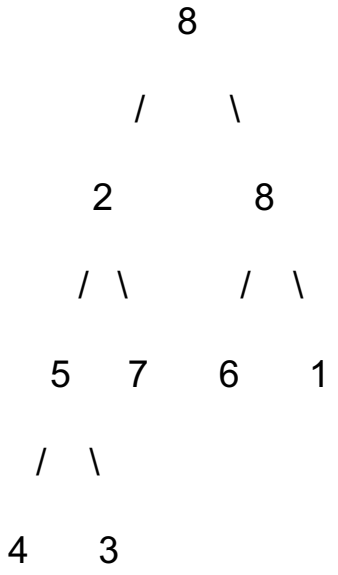
سوال چهارم

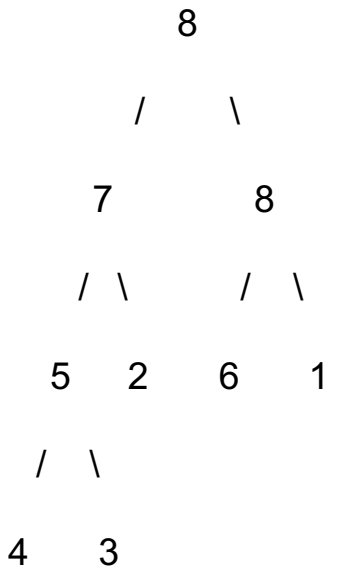
ابتدا با heap sort یک max heap خواهیم داشت.

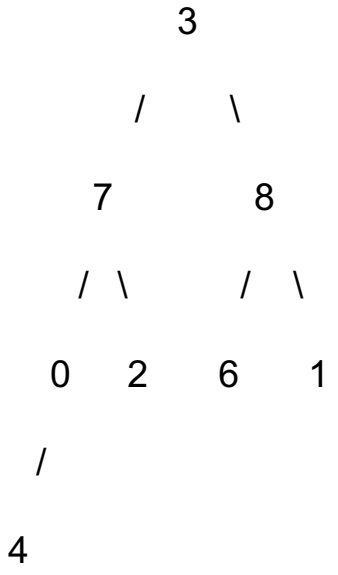


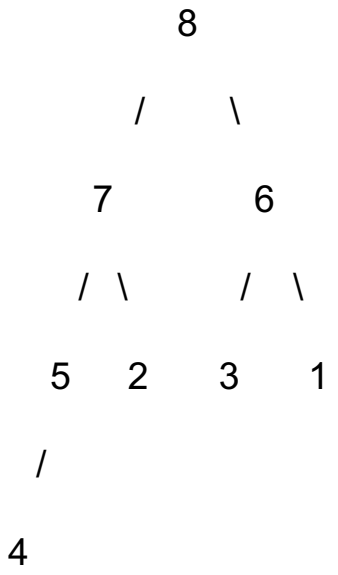


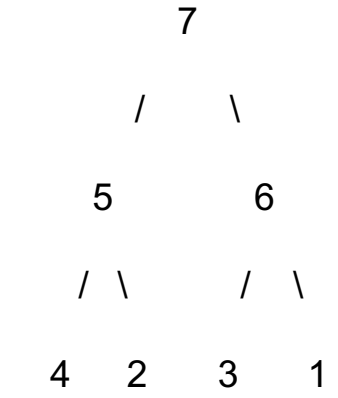

2
8 8
5 7 6 1
4 3

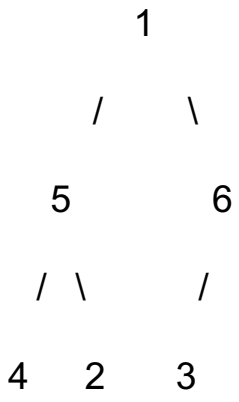

8
2 8
5 7 6 1
4 3

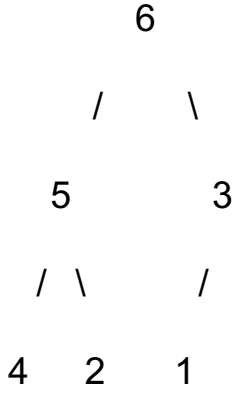

8
7 8
5 2 6 1
4 3

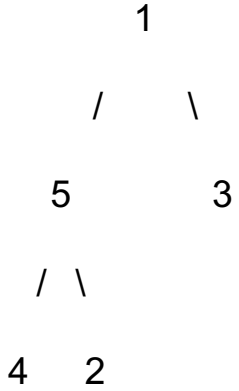

3
7 8
0 2 6 1
4

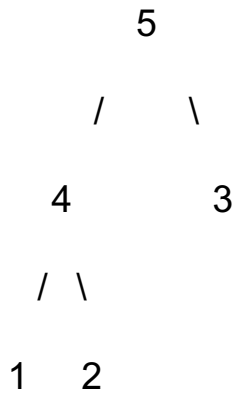

8
7
6
5
2
3
1
4

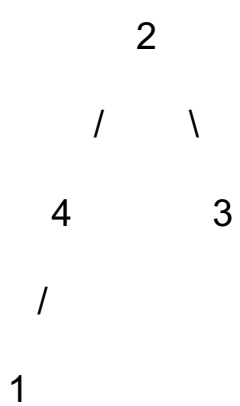

7
5
6
4
2
3
1

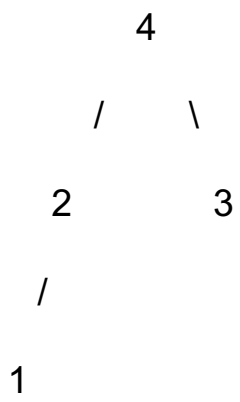

1
/ \
5 6
/ \ /
4 2 3

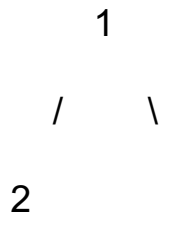

6
/ \
5 3
/ \ /
4 2 1

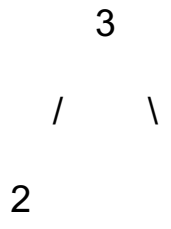

1
/ \
5 3
/ \
4 2

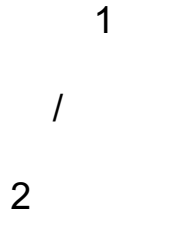

5
/ \
4 3
/ \
1 2

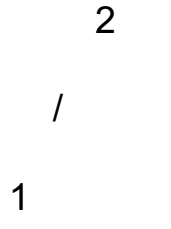

2
/ \
4 3
/
1


4
/ \
2 3
/
1


1
/ \
2 3


3
/ \
2 1


1
/
2


2
/
1

1

1, 2, 3, 4, 5, 6, 7, 8, 8, 9

اکنون با quick sort داریم:

ابتدا خانه اول را pivot در نظر می‌گیریم.

8, 2, 9, 4, 8, 6, 1, 5, 3, 7

حال بررسی می‌کند تا به ۹ برسد. پس از آن ۹ را با تمام عناصر مقایسه کرده و به انتهای آرایه منتقل می‌کند.

7, 2, 4, 8, 6, 1, 5, 3, 8, 9

۲ و ۴ تغییری نمی‌کنند اما به ۸ که می‌رسد آن را بقیه عناصر مقایسه می‌کند.

3, 2, 4, 6, 1, 5, 7, 8, 8, 9

۲ تغییری نمی‌کند اما ۴ و ۶ و ۵ چون بزرگترند جابجا می‌شوند.

1, 2, 3, 6, 4, 5, 7, 8, 8, 9

چون ۱ و ۲ و ۳ از همه کوچکترند تغییری نمی‌کنند.

۶ که پیوت است با ۵ جابجا شده و سپس ۵ پیوت می‌شود و با ۴ جابجا می‌شود.

1, 2, 3, 4, 5, 6, 7, 8, 8, 9

سوال پنجم

```
def npqs(array, low, high, n):
    if low < high:
        pivots = list()
        for i from 0 to n-1:
            pivots.add(partition(array, low+((i+1)*(high-
low)//(n+1)), low, high)
        npqs(arr, low, pivots[0], n)
        for i from 0 to n-2:
            npqs(array, pivots[i]+1, pivots[i+1], n)
        npqs(array, pivots[n-2]+1, high, n)
def partition(array, p, low, high):
    pivot = array[p]
    swap(p, high)
    st = low
    for i from low to high:
        if array[i] <= pivot:
            swap(i, st)
            st += 1
```

در رابطه با پیچیدگی الگوریتم، اگر تقسیم‌بندی به صورت یکنواخت باشد پیچیدگی O(n logn) خواهیم داشت ولی اگر به شکل نامتوازن صورت گیرد پیچیدگی $O(n^2)$ خواهیم داشت.

پیچیدگی میانگین O(n logn) است.

سوال ششم

```
array = list()

size = input()

for i from 0 to size:

    array.add(input())

radix_sort(array)

for i from 0 to size:

    if i+1 < size:

        tmp = array[i]

        array[i] = array[i+1]

        array[i+1] = tmp
```

در ابتدا با radix sort آرایه مرتب می‌شود. سپس از عضو صفرم رد شده و عناصر nم و i+1م جابجا می‌شوند.

پیچیدگی زمانی radix sort O(d*(n+k)) است. همچنین یک بار در انتها آرایه را با O(n) پیمایش می‌کنیم و در کل O(n) می‌شود.

پیچیدگی مکانی radix sort O(n+k) است.

سوال هفتم

```
def find(array):
    radix_sort(array)
    c = 0
    for i from 0 to len(array):
        if array[i] != array[i-1]:
            if c%2 == 0:
                print(c)
                break
            else:
                c = 0
        else:
            c += 1
```

ابتدا با radix sort اعداد مرتب می‌شوند. سپس با پیمایش بر روی اعضای آرایه و شمارش دفعات تکرار هر عدد، هر گاه به عدد مورد نظر رسیدیم آن را در خروجی نمایش می‌دهیم.

پیچیدگی زمانی radix sort O(d*(n+k)) است و پس از آن هم یک بار آرایه پیمایش می‌شود که در کل مرتبه زمانی O(n) خواهد بود.

سوال هشتم

```
def is_rama(n):
    s = False
    for i from 1 to sqrt(n):
        for j from i+1 to sqrt(n):
            sum = (i^3) + (j^3)
            if sum == n:
                if not s:
                    s = True
                else:
                    # is Rama
                    return True
    # not Rama
    return False
```

دو حلقه تودرتو وجود دارند که با به توان رساندن و جمع آنها شرایطشان بررسی می‌شود. شرط پایان حلقه‌ها این است که توان بزرگترین عدد به عدد ورودی برسد.

اگر یک جفت عدد دارای شرایط مذکور پیدا شوند، متغیر s به True تغییر می‌کند و اگر جفت دوم پیدا شود یعنی عدد رامانوجان است.

با توجه به دو حلقه تودرتویی که داریم، پیچیدگی زمانی الگوریتم $O(n^2)$ است.