[redacted] بخشنده مهربان [redacted]

پاسخ تمرین شماره ۲

نازنین صبری

۲۰ دی ماه ۱۳۹۵

۱. برای مرتب سازی استک به طور بازگشتی می‌توانیم به این صورت عمل کنیم:

```
def sortedInsert(Stack S, element)
        if stack is empty OR element > top element:
                push(S, elem)
        else:
                temp = pop(S)
                sortedInsert(S, element)
                push(S, temp)


def sortStack(stack S):
        if stack is not empty:
                temp = pop(S)
                sortStack(S)
                sortedInsert(S, temp)
```

زمان اجرای این الگوریتم برابر است با: $O(n^3)$

۲. الف) ابتدا ند (node) جدید را ایجاد کرده و عدد (داده) متناظر را در آن می‌ریزیم، پوینتر به این ند (newNode) را نگه می‌داریم. برای اضافه کردن این ند جدید به لیست پیوندی موجود به ۲ حالت می‌توان رسید:

— لیست پیوندی خالی باشد:

در این صورت کافی است که next این ند به خودش اشاره کند و head که ابتدای لینک لیست را نشان می‌دهد نیز به همین عنصر اشاره کند

```
newNode->next = newNode
head = newNode
```

— لیست پیوندی خالی نباشد:

فرض می‌کنیم که لیست به طور صعودی مرتب شده است و هدف ما نیز این است که پس از اضافه کردن ند جدید هم‌چنان صعودی بماند (برای لیست پیوندی نزولی هم به روش مشابه می‌توان عمل کرد) برای اینکه بفهمیم عنصر به کجای لینک لیست اضافه می‌شود باید روی لینک لیست حرکت کنیم و هر جا که عنصر کوچکتر یا مساوی محتوای ند ما بود و عنصر بعدی بزرگتر از آن بود متوقف شویم و محل اضافه شدن بین ۲ ند خواهد بود، شبه کد پیدا کردن محل اضافه کردن ند جدید به صورت زیر خواهد بود:

```
current = head
next = head->next
while( !( current->value<=newNode->value  &&  next->value>
    newNode->value ) ){
        current = next
        next = next-> next
}
```

حال این محل اضافه کردن یکی از ۲ حالت زیر را خواهد داشت:

* عنصر جدید باید قبل از head اضافه شود:

این حالت در صورتی اتفاق می‌افتد که داده‌ی ما از همه‌ی داده‌های موجود در لیست پیوندی کوچکتر باشد. در این حالت ابتدا باید اخرین عنصر حلقه را پیدا کنیم، next آن را تغییر دهیم، سپس باید next در newNode را تغییر دهیم تا به عنصر اول لیست در حال حاظر اشاره کند و بعد خود head را برابر با اشاره‌گر به این ند جدید قرار دهیم، شبه کد آن به شکل زیر خواهد شد:

```
current = head
//finding the last node in the loop
while (current->next != head):
        current = current ->next

//when the last node is found change the next of the last node
current -> next = newNode

//change the next of newNode
newNode -> next = head

//change head
head = newNode
```

* عنصر جدید باید جایی بعد از head اضافه شود:

در این صورت زمانی که کد یافتن محل اضافه شدن را اجرا کردیم (شبه کد آن در بالا آماده بود) به ما اشاره گر به عنصر قبلی و بعدی عنصر جدید را داده است پس کافی است با کمک آن‌ها next ها را تغییر داده و عنصر جدید را به وسط دو عنصر دیگر اضافه کنیم.

```
// current points to the previous node and next to the node
    after the one we are inserting
current -> next = newNode
newNode -> next = next
```

ب) برای حل این سوال ۲ روش پشنهاد می‌دهیم که پیچیدگی زمانی هر دو $O(n)$ است:

** روش اول: استفاده از یک flag که در صورت عبور از ند مقدار آن true شود

در این روش نیاز داریم که ساختار کلی لیست پیوندی را تغییر دهیدم به نحوی که هر ند علاوه بر مقدار و اشاره‌گر به ند بعدی یک متغیر visited داشته باشد.

```
struct Node{
        int value;
        struct Node* next;
        bool visited;
}
```

در ابتدا مقدار این متغیر را برای تمام ند ها برابر با false قرار می‌دهیم سپس از ابتدای لیست پیوندی (از head) شروع به حرکت کرده و از هر ند که عنور می‌کنیم مقدار visited آن را برابر با true قرار می‌دهیم، در این صورت اگر به ندی برسیم که مقدار visited آن true باشد به این معنی است که قبلا دیده شده است و این به این معنی است که لینک لیست دارای حلقه است.

** روش دوم: استفاده از ۲ اشاره‌گر برای حرکت روی لیست

اشاره‌گر اول یکی یکی روی لیست حرکت کند و اشاره‌گر دوم دو تا دو تا، اگر این ۲ اشاره‌گر در جایی به هم برسند (با هم یکی شوند) به این معنی است که حلقه وجود دارد و در غیر این صورت به این معنی است که حلقه‌ای وجود ندارد. این الگوریتم با نام Floyd's Cycle-Finding Algorithm شناخته شده است، شما می‌توانید با جست و جوی این نام اطلاعات بیشتری را درباره‌ی آن پیدا کنید.

کد این راه حل به شکل زیر خواهد بود.

```
struct Node  *slow_p = head, *fast_p = head;

//while none of the pointers are NULL
while (slow_p && fast_p && fast_p->next )
{
    slow_p = slow_p->next;
    fast_p  = fast_p->next->next;
    if (slow_p == fast_p)
    {
        printf("Found Loop");
        return 1;
    }
}
return 0;
```

ج) ۲ عدد باینری که به عنوان ورودی به ما داده می‌شوند را A و B و حاصل جمع آن‌ها را C در نظر گرفتیم. (تمام خانه‌های C را به صفر مقدار دهی اولیه می‌کنیم)
طبق شبه کد نشان داده شده در پایین جمع بیت‌های متناظر از A و B و C را در متغیری می‌ریزیم. (این بیت در C در صورتی یک است که carry جمع بیت‌های قبلی ۱ بوده باشد) این حاصل جمع یکی از ۴ عدد ۰، ۱، ۲ و یا ۳ است که چون قرار است C نیز نمایش باینری باشد پس این حاصل برابر با مقدار این خانه از C و carry جمع است که همان مقدار خانه‌ی بعدی از C خواهد بود.

```
vector<int> C (n+1, 0);
for(int i=0; i<n; i++){
        sum = A[i] + B[i] + C[i];
        C[i] = sum%2;
        C[i+1] = int(sum/2);
}
return C;
```

۳. برای حل این سوال از قواعد هم‌نهشتی بر ۳ استفاده می‌کنیم. ابتدا آرایه را به طور نزولی مرتب می‌کنیم سپس جمع عناصر را حساب می‌کنیم اگر باقیمانده‌ی تقسیم حاصل جمع بر ۳ برابر با ۰ بود کافی است آرایه را از ابتدا تا انتها چاپ کنیم (چون نزولی مرتب شده است پس بزرگ‌ترین عدد ممکن چاپ خواهد شد). اگر این باقیمانده صفر نبود باید یک یا ۲ عنصر را حذف کنیم با بخشپذیر شود، عمل حذف را اینگونه انجام می‌دهیم:
- اگر باقیمانده بر ۳ برابر با ۱ بود:
باید ۱ عدد با باقیمانده‌ی ۱ بر ۳ و یا ۲ عدد با باقیمانده‌ی ۲ بر ۳ را حذف کنیم. چون می‌خواهیم بزرگ‌ترین عدد ممکن را چاپ کنیم پس باید سعی کنیم کوچکترین اعدادی که شرایط گفته شده را دارند حذف کنیم. پس از انتها به ابتدا روی آرایه حرکت می‌کنیم و index اولین عناصری که شرایط گفته شده را دارند ذخیره می‌کنیم. اگر ۱ عدد با باقیمانده‌ی ۱ بر ۳ پیدا شد همان عدد را حذف کرده و از حلقه خارج می‌شویم چون قطعا کم شدن ۱ رقم در مقایسه با کم شدن ۲ رقم عدد بزرگ‌تری تولید می‌کند.
- اگر باقیمانده بر ۳ برابر با ۲ بود:
باید ۱ عدد با باقیمانده‌ی ۲ بر ۳ و یا ۲ عدد با باقیمانده‌ی ۱ بر ۳ را حذف کنیم. روش انجام این کار هم مشابه بالا است.
اگر موفق به حذف به طوری که شرط بخشپذیری درست شود نشویم عبارت not possible را برمی‌گردانیم.
زمان اجرا با توجه به الگوریتم مرتب سازی استفاده شده می‌تواند متفاوت باشد ولی اگر بخواهیم فقط هزینه‌ی زمانی بخش پیدا کردن عدد را بیان کنیم، این هزینه‌ی زمانی O(n) خواهد بود

۴. آ) پاسخ: 3 * 1 + 2 - 9

برای کسب اطلاعات بیشتر درباره‌ی نحوه‌ی تبدیل عبارات postfix به infix به لینک زیر مراجعه کنید:

Infix, Postfix and Prefix.

Postfix to Infix Conversion.

ب) با الگوریتم زیر و با استفاده از استک این تبدیل را انجام می‌دهیم:

۱- متغیر جدیدی تعریف می‌کنیم که عبارت postfix را در آن ذخیره کنیم. آن را result می‌نامیم، هم چنین یک استک خالی تعریف می‌کنیم. (string result = "")

۲- عبارت infix را کاراکتر به کاراکتر از چپ به راست می‌خوانیم.

۳- اگر کاراکتر خوانده شده یک عملوند بود (عدد بود) آن را به result اضافه می‌کنیم.

۴- اگر کارکتر خوانده شده یک عملگر است:

۴-۱ : اگر یک عملگر است که اولویت اجرای آن از اولویت اجرای عملگر ابتدای استک بیشتر بود یا استک خالی بود این عملگر را در استک push می‌کنیم

۴ -۲ : در غیر این صورت عملگرهای موجود در استک را یکی یکی pop می‌کنیم و هر عملگری که pop می‌شود را به result اضافه می‌کنیم تا به جایی برسیم که اولویت اجرای این عملگر از عملگر روی استک بیشتر باشد یا استک خالی شده باشد، در اینجا عملگر جدید را push می‌کنیم.

۵- اگر کاراکتر خوانده شده (است آن را در استک push می‌کنیم.

۶- اگر کارارکتر خوانده شده) است، محتوای استک را pop کرده و به result اضافه می‌کنیم تا به (برسیم.

۷- مراحل ۲ تا ۶ را ادامه می‌دهیم تا کل عبارت ورودی خوانده شود.

۸- اگر استک خالی نبود محتوای آن را pop کرده و به result اضافه می‌کنیم

هزینه‌ی زمانی: O(n*m) m<=n دلیل ضرب شدن در m حلقه‌ی while داخلی توصیف شده در بالا است

۵. از ساختار داده‌ای به نام Dequeue که نوعی صف است استفاده می‌کنیم. تفاوت آن با صف عادی این است که عناصر می‌توانند هم به سر و هم به ته آن اضافه شوند و یا حذف شوند. صفی به طول k تعریف می‌کنیم که در آن عنصر هدف هر ریز مجموعه را در آن‌ها نگه می‌داریم. عنصر هدف هر زیر مجموعه عضوی از مجموعه است که در زیر مجموعه k تایی ما موجود است و از تمام اعضای آن زیر مجموعه بزرگ‌تر است. در اسن صف ترتیب نزولی برای اعضا حفظ می‌کنیم.

با شروع از زیر مجموعه‌ی k تایی اول بزرگ‌ترین عضو آن را به سر صف اضافه می‌کنیم.

عنصر سر صف را چاپ می‌کنیم چون عنصر سر صف بزرگ‌ترین عنصر زیر مجموعه‌ی قبلی است. سپس به سراغ اولین عنصر بعد از زیر مجموعه k تایی اول می‌رویم. (i=k)

* تمامی عناصری از صف را که در زیر مجموعه‌ی جدید نیستند از سر صف خارج می‌کنیم. اعضای باقیمانده صف اگر از عنصری از مجموعه که روی آن هستیم کوچکتر باشند آن‌ها را از ته مجموعه حذف می‌کنیم تا جایی که دیگر کوچکتر نباشند یا صف خالی شده باشد و عنصری که روی آن هستیم را به انتهای صف اضافه می‌کنیم.

هر بار i را یک واحد زیاد می‌کنیم و * را تکرار می‌کنیم تا به انتهای آرایه برسیم.

۶. می‌توانیم با استفاده از ساختار داده صف به حل این سوال بپردازیم. به طوری که دور موجود را در آن زخیره کنیم. به این شکل که پمپ بنزین‌ها را با شروع از پمپ بنزین اول در صف enqueue می‌کنیم تا یا به انتهای شهر برسیم و دور کامل شود و یا به جایی برسیم که بنزین موجود در باک منفی شود در این صورت

عناصر را dequeue می‌کنیم تا به جایی برسیم که میزان بنزین مثبت شود و یا صف خالی شود.
شبه کد زیر به طور دقیق‌تر راه حل را نشان می‌دهد:

```
#we have a queue called Q and an array of gas stations (stations)
    which have two variables: distance and petrol
def find_Tour():
        i = 0
        curr_petrol = 0
        Q.enqueue(stations[i])
        #I will use Q.front to see the content of the firls
            element of the queue
        # and I will use Q.rear to refer to the last element in
            the queue
        curr_petrol = curr_petrol + Q.front.petrol - Q.front.
            distance

        while (curr_petrol <0 || !(Q.front == Q.rear) ):

                while curr_petrol <0 and !(Q.front == Q.rear):

                        curr_petrol = curr_petrol - (Q.front.
                            petrol - Q.front.distance)
                        Q.dequeue()

                        #if the first station is being considered
                            as an starting point again, it means
                            that all stations have been checked
                            and there is no possible way to make a
                             tour around the city
                        if Q.front==station[0]:
                                return -1;

                i += 1
                if i >= len(stations):
                        i = 0
                curr_petron = curr_petrol + (Q.rear.petrol - Q.
                    rear.distance)
                Q.enqueue(stations[i])
```

۷. آ) توابع پایه‌ی یک stack ۲ تابع push و pop هستند پس باید نشان دهیم که هر کدام از این کارها با استفاده از ۲ صف چگونه انجام خواهند شد.

۱ - در این حالت می‌خواهیم push بهینه باشد یعنی با $O(1)$ انجام شود.

:push

صف ۱ را به عنوان صف اصلی و صف ۲ را بع عنوان کمکی در نظر می‌گیریم. برای این کار عنصر را در صف اول وارد می‌کنیم (enqueue می‌کنیم).

:pop

هر عنصری که به صف ۱ وارد می‌شود به انتهای آن اضافه می‌شود در نتیجه اخرین عنصری که وارد صف ۱ شده اخیرین عنصر این صف و اولین عنصری که وارد شده اولین عضو صف است و از آن جایی که استک FILO (First In, Last Out) است پس زمانی که می‌خواهیم عمل pop را انجام دهیم باید عنصر اخر صف ۱ را به عنوان خروجی بدهیم، پس اینگونه عمل می‌کنیم:

تا زمانی که صف ۱ بیش از ۱ عضو دارد از آن dequeue کرده و در صف ۲ enqueue می‌کنیم هنگامی که به اخرین عنصر رسیدیم آن را dequeue کرده و به عنوان پاسخ pop بازمی‌گردانیم و سپس نام ۲ صف را جابجا می‌کنیم. (تا مجددا صف خالی صف ۲ و صف اصلی صف ۱ شود).

شبه کد مربوطه در زیر آماده است:

```
#our queues: Queue1, Queue2
def push(x):
        Queue1.enqueue(x)

def pop():
        while len(Queue1)>1:
                y = Queue1.dequeue()
                Queue2.enqueue(y)
        result = Queue1.dequeue()
        Queue1 = Queue2
        Queue2 = EMPTY_QUEUE
        return result
```

۲ - در این حالت می‌خواهیم pop بهینه باشد یعنی با $O(1)$ انجام شود.

:pop

از صف ۱ dequeue می‌کنیم.

:push

عنصر جدید را در صف ۲ می‌ریزیم سپس تمامی عناصر موجود در صف ۱ را به ترتیب dequeue کرده و در صف ۲ enqueue می‌کنیم و سپس نام ۲ صف را جابجا می‌کنیم.

شبه کد مربوطه در زیر آماده است:

```
#our queues: Queue1, Queue2
def push(x):
        Queue2.enqueue(x)
        while len(Queue1)>0:
                y = Queue1.dequeue()
                Queue2.enqueue(y)
```

```
        Queue1 = Queue2
        Queue2 = EMPTY_QUEUE
        return result

def pop():
        x = Queue1.dequeue()
        return x
```

ب) توابع پایه‌ی یک queue ۲ تابع enqueue و dequeue هستند پس باید نشان دهیم که هر کدام از این کارها با استفاده از لیست پیوندی چگونه انجام خواهند شد.

برای اینکه بتوانیم با $O(1)$ به اولین و اخرین عنصر لیست پیوندی دسترسی داشته باشیم باید ۲ اشاره‌گر یکی به ابتدا و دیگری به انتهای لیست نگه‌داری کنیم. (این ۲ اشاره‌گر را front و rear می‌نامیم)

enqueue:

این تابع ۱ عنصر به انتهای لیست اضافه می‌کند و rear را تغییر می‌دهد.

dequeue:

این تابع ۱ عنصر از ابتدای لیست را خارج کرده و front را تغییر می‌دهد.

نحوه‌ی تغییرات را با استفاده از شبه کد زیر نشان می‌دهیم.

```
class Q:
        Node front, rear
#from now on we assume that the Q linked list is
    available in all functions
#front points to the first item
#rear points to the last item
def enqueue(Node newNode):
        #if the linked list is empty then the front
            and the rear will both become this new
            node
        if Q.rear==NULL:
                Q.front = Q.rear = newNode
                return

        #change rear
        newNode.next = Q.rear
        Q.rear = newNode

def dequeue():
        #if the linked list is empty then there is
            nothing to dequeue
        if Q.front==NULL:
                return NULL
```

```
Node temp = Q.front
Q.front = Q.front.next

if Q.front==NULL:
        Q.rear = NULL

return temp
```