تمرینات فصل یک

نام: امیرحسام بهمن‌خواه نام: مریم رضائی نام: محمدحسین فرخی

1) فرض کنید هر گره داخلی درخت دودویی T دو فرزند داشته باشد. اگر تعداد گره‌ها n و ارتفاع h باشد، ثابت کنید:

أ. تعداد گره‌های خارجی T، حداقل $h+1$ و حداکثر 2^h خواهد بود؛

ب. تعداد گره‌های داخلی T، حداقل h و حداکثر 2^h-1 خواهد بود؛

ج. تعداد کل گره‌های T، حداقل $2h+1$ و حداکثر $2^{h+1}-1$ خواهد بود؛

د. ارتفاع T، حداقل $log\ (n+1)\ -1$ و حداکثر $\frac{(n-1)}{2}$ خواهد بود.

جواب:

1 – الف) از آنجایی که ارتفاع برابر با طول بلندترین مسیر از root درخت به یک leaf آن است، مقادیر h به ترتیب برای تعداد یک گره به بالا برابر با 0، 1، 2 تا بی‌نهایت می‌باشد (درخت T نمی‌تواند خالی باشد). برای درخت دودویی T که هر گره‌ی آن یا دو فرزند داشته باشد یا هیچ، به نسبت هر h با مشاهده داریم:

- ✓ **ارتفاع صفر:** تنها گره‌ی موجود هم ریشه است و هم برگ، پس تعداد گره‌ی خارجی $e = 1$.
- ✓ **ارتفاع یک:** اگر ریشه هر دو فرزند را داشته باشد آنگاه آن‌ها برگ‌ها خواهند بود و $e = 2$.
- ✓ **ارتفاع دو:** اگر یکی از دو فرزندِ ریشه دو فرزند داشته باشد $e = 3$ و اگر هر دو داشته باشند $e = 4$.
- ✓ **ارتفاع سه:** کافی‌ست یکی از دو فرزندِ ریشه دارای دو برگ و تنها یکی از آن دو برگ دارای دو برگ دیگر باشد تا با کمترین تعداد گره به $h = 3$ برسیم که آنگاه $e = 4$ می‌باشد. ماکسیموم تعداد گره در این ارتفاع هم زمانی اتفاق می‌افتد که اگر دو فرزندِ ریشه دارای دو برگ باشند و از چهار برگ حاصل هر دو باز دارای دو فرزند باشد، در این صورت $e = 8$ خواهد بود.

در این صورت دو دنباله اعداد برای مینیموم و ماکسیموم تعداد گره در هر ارتفاع وجود دارد:

minimum external nodes: $1, 2, 3, 4, \ldots \rightarrow m + 1 \qquad m \geq 0$

maximum external nodes: $1, 2, 4, 8, \ldots \rightarrow 2^m \qquad m \geq 0$

می‌بینیم $m = h$ پس رابطه تعداد گره‌های خارجی با ارتفاع در min و max تعداد گره‌ی کلی برابر است با:

$$e_{min} = h + 1 \qquad e_{max} = 2^h$$

****

1 – ب) مانند قسمت قبل سوال، کمترین و بیشترین تعداد گره‌های خارجی (یعنی گره‌هایی که دارای فرزند هستند) را برای هر ارتفاع به صورت دو دنباله داریم که می‌توانیم جمله‌ی عمومی آن‌ها را بیابیم:

minimum internal nodes: $0, 1, 2, 3, \dots \rightarrow m \qquad m \geq 0$

maximum internal nodes: $0, 1, 3, 7, \dots \rightarrow 2^m - 1 \qquad m \geq 0$

می‌بینیم $m = h$ پس رابطه تعداد گره‌های داخلی با ارتفاع در min و max تعداد گره‌ی کلی برابر است با:

$$i_{min} = h \qquad i_{max} = 2^h - 1$$

****

1 – ج) به همین ترتیب برای تعداد min و max کل گره‌های درخت T در ارتفاع‌های مختلف (از 0 تا h) داریم:

$$\textit{minimum nodes} = e_{min} + i_{min} = (h + 1) + h = 2h + 1$$

$$\textit{maximum nodes} = e_{max} + i_{max} = 2^h + (2^h - 1) = 2^{h+1} - 1$$

****

1 – د) کمترین ارتفاع ممکن برای درخت دودویی T با بیشترین n زمانی رخ می‌دهد که تمام گره‌ها تا جای ممکن دارای دو فرزند باشند و بیشترین ارتفاع ممکن برای کمترین n زمانی که گره‌ها به صورت رشته‌وار تنها از یک سوی درخت پیش روند. پس برای $h \geq 0$ مقدار n دو دنباله نتیجه می‌دهد:

height:	0,	1,	2,	3,	…
maximum nodes:	1,	3,	7,	15,	…
minimum nodes:	1,	3,	5,	7,	…

در قسمت ج سوال برای n_{min} رابطه‌ای با h یافتیم؛ از آنجا که h_{max} برای n_{min} در حالت گفته شده رخ می‌دهد و n_{min} با h رابطه‌ای درجه یک دارد، می‌توانیم به راحتی از آن رابطه به نتیجه برسیم:

$$2h_{max} + 1 = n_{min} \rightarrow h_{max} = \frac{n-1}{2}$$

و برای n_{max} با h_{min} بر این اساس داریم:

$$2^{h+1} - 1 = n_{max} \rightarrow h_{min} = \log_2(n+1) - 1$$

این رابطه را می‌توان جداگانه با اثبات ریاضی نیز به دست آورد. بدین صورت که اگر درخت دودویی T همواره در پرترین حالت خود باشد (یعنی تمامی گره‌های آن دارای دو فرزند باشند) با بررسی تعداد گره‌های دارای depth یا عمق یکسان، مقادیر 2^0 (عمق صفر، یعنی فقط خود ریشه)، 2^1، 2^2، ...، 2^{h-1}، 2^h (زمانی که عمق رده گره‌های مورد نظر برابر با ارتفاع درخت است) را میابیم که جمع تمامی آن‌ها مقدار ماکسیموم n را برای کوچکترین مقدار ممکن h تعیین می‌کند؛ یعنی:

$n_{max} = 2^h + 2^{h-1} + \ldots + 2^2 + 2^1 + 2^0 \rightarrow n_{max} + 1 = 2^h + 2^{h-1} + \ldots + 2^2 + 2^1 + 2^1$

$\rightarrow n_{max} + 1 = 2^h + 2^{h-1} + \ldots + 2^2 + 2^2 \rightarrow n_{max} + 1 = 2^h + 2^{h-1} + \ldots + 2^3$

$\rightarrow n_{max} + 1 = 2^{h+1} \rightarrow h_{min} = \log_2(n+1) - 1$

نکته‌ی قابل توجه این است که این جمله‌ی عمومی مخصوص حالات خاصی‌ست که برای کمترین میزان h بیشترین مقدار n رخ دهد، یعنی با قرار دادن مقادیر دنباله‌ی maximum nodes در آن، کوچکترین ارتفاع ممکن را پیدا می‌کنیم:

$n = 15 \rightarrow \log_2(15+1) - 1 = 3$

اما اگر مقدار عضو حالات خاص نباشد (یعنی برای h_{min} بیشترین مقدار n نباشد) و تنها برای یک n ممکن بخواهیم کمترین ارتفاع را بیابیم، به مشکل برمی‌خوریم:

$n = 13 \rightarrow \log_2(13+1) - 1 = 2.8$

از آنجا که ارتفاع تنها عددی صحیح می‌تواند باشد، برای یافتن رابطه‌ای که برای تمامی مقادیر فرد n (که تعداد گره‌های ممکن این درخت دودویی هستند) کوچکترین ارتفاع را به ما بدهد، از استقرا کمک گرفته و برای هر n فرد حاصل رابطه را یافته و همزمان با استفاده از اعداد و بازه‌های حاصل از آن‌ها درون دنباله‌ی maximum nodes، ارتفاع مینیموم اصل برای آن n را مقایسه می‌کنیم:

$n = 1 \rightarrow \log_2(1+1) - 1 = 0$ $\quad (h_{min} = 0)$

$n = 3 \rightarrow \log_2(3+1) - 1 = 1$ $\quad (h_{min} = 1)$

$n = 5 \rightarrow \log_2(5+1) - 1 \approx 1.6$ $\quad (h_{min} = 2)$

$n = 7 \rightarrow \log_2(7+1) - 1 = 2$ $\quad (h_{min} = 2)$

$n = 9 \rightarrow \log_2(9+1) - 1 \approx 2.3$ $\quad (h_{min} = 3)$

$n = 13 \rightarrow \log_2(13+1) - 1 \approx 2.8$ $\quad (h_{min} = 3)$

$n = 15 \rightarrow \log_2(15+1) - 1 = 3$ $\quad (h_{min} = 3)$

بنابراین با گرفتن سقف مقدار log جمله‌ی عمومی کامل را پیدا می‌کنیم:

$h_{min} = \lceil \log_2 n + 1 \rceil - 1$

---

2. الف) یک صف را با دو پشته شبیه‌سازی کنید. یعنی با کمک دو پشته و طبق سیاست درج و حذف از پشته، الگوریتمهای $Enqueue(Q, e)$ و $Dequeue(Q)$ را برای درج و حذف داده‌ها از صف طراحی کنید.

ب) یک پشته را با دو صف شبیه‌سازی کنید. یعنی با کمک دو صف و طبق سیاست درج و حذف از صف، الگوریتم‌های $Push(S, e)$ و $Pop(S)$ را برای درج و حذف داده‌ها از پشته طراحی کنید.

جواب:

2 – الف) برای شبیه‌سازی یک صف با دو پشته با این اطمینان که داده همواره در front اضافه شود و از rear حذف شود، اگر top پشته اول را معادل front صف بدانیم، عملیات enqueue به سادگی با استفاده از push پشته انجام می‌شود زیرا در پشته همواره داده‌ها به top اضافه می‌شود.

Algorithm: Enqueue(Q, e)

// stimulates *enqueue()* of queue using *push()* from an unlimited stack

// Input: *Q* which is stack *S1*, element *e* to insert in *front* of queue

// Output: element *e* is inserted on top of *S1*

S1.push(e)

اما برای عملیات dequeue از آنجا که داده در صف باید در rear اضافه شود و در پشته به top و در این حالت top پشته معادل front صف است، با کمک یک پشته‌ی خالی برعکس کردن ترتیب داده‌ها را انجام می‌دهیم.

Algorithm: Dequeue(Q)

// stimulates *dequeue()* of queue using *push()* and *pop()* in unlimited stacks

// Input: *Q* which is mainly stack *S1*

// Output: element *e* is removed from *rear* of *S1* and returned

if S1.isEmpty() **then**

    raise error

else then

    while S1.isEmpty() **is False do**

        S2.push(S1.pop()) // pushing everything from *S1* into empty *S2*

    e ← S2.pop()

    while S2.isEmpty() **is False do**

        S1.push(S2.pop()) // pushing elements of *S2* back into empty *S1*

    return e

2 – ب) برای شبیه‌سازی یک پشته با دو صف باید عملیات‌های push و pop یک پشته را با عملیات‌های enqueue و dequeue دو صف جایگزین کنیم. اگر front اولین صف را معادل top پشته در نظر بگیریم، از آنجا که در صف داده‌ها به rear اضافه می‌شوند نه front، برای اطمینان از اضافه شدن داده به top یک پشته در push، از هر دو عملیات enqueue و dequeue صف‌ها باید استفاده شود که دومین صف کمکی‌ست و برای برعکس کردن اولین صف به کار می‌رود.

Algorithm: Push(S, e)

// stimulates *push()* operation of stack with *enqueue()* and *dequeue()* operations of queue, assuming size of queue is unlimited

// Input: *S* which is mainly queue *Q1*, element *e* to put in the *front* of *Q1*

// Output: element *e* is pushed on *top* of *S*

if Q1.isEmpty() **then**

Q1.enqueue(e)

else then

Q2.enqueue(e)

while Q1.isEmpty() **is False do**

Q2.enqueue(Q1.dequeue())   // putting elements of *Q1* into *Q2*

while Q2.isEmpty() **is False do**

Q1.enqueue(Q2.dequeue())   // reversing *Q2* and filling *Q1*

برای pop فقط از dequeue استفاده می‌شود زیرا در صف داده از front حذف می‌شود که در اینجا معادل top پشته است.

Algorithm: Pop(S)

// stimulates *pop()* operation of stack with *dequeue()* operation of queue

// Input: *S* which is the queue *Q1*

// Output: element *e* is removed from the *rear* of *Q1* and returned

if Q1.isEmpty() **then**

raise error

else then

e ← Q1.deqeueue()   // removes element and assigns it to e

return e

3) ساختارداده‌ای طراحی کنید که با آن بتوان، هر سه عملیات $Push$، Pop و $Find - Min$ را در زمانی ثابت انجام داد. (منظور از «زمان ثابت» این است که سرعت انجام عملیات وابسته به تعداد عناصر نباشد.) تعاریف این سه عملیات عبارتند از:

- **$Push(S, e)$: عنصری را در انتهای ساختارداده درج می‌کند.**
- **$Pop(S)$: عنصری را از انتهای ساختارداده حذف می‌کند.**
- **$Find - Min(S)$: کوچکترین عنصر را در ساختارداده برمی‌گرداند (بدون آنکه آن را حذف کند).**

در واقع، در اینجا ما به دنبال طراحی الگوریتم‌هایی برای انجام سریع سه عملیات $Push$، Pop و $Find - Min$ هستیم. و قسمت عمده طراحی این الگوریتم‌ها، چیزی نیست جز طراحی ساختارداده‌ای برای چینش مناسب داده‌ها.

<u>جواب:</u>

3) از آنجایی که می‌خواهیم عملیات‌های Push و Pop و FindMin مقدار زمان یکسانی مصرف کنند، باید پشته را به شکلی طراحی کنیم که در زمان قرار دادن داده‌ها در این ساختار داده، مقدار مینیموم همواره در top قرار بگیرد تا تفاوتی با دو عملیات دیگر نکند و دسترسی به آن آسان باشد. برای این کار به دو پشته نیاز داریم؛ یکی به عنوان پشته‌ی اصلی برای ذخیره‌ی داده‌ها و دیگری به عنوان پشته‌ی کمکی که در آن مقادیر مینیموم را برای هر سری انجام عملیات‌های Push و Pop در بالا قرار دهیم.

Algorithm: Push(S, e)

// inserts element on top of main stack and if it’s the minimum element, also on top of an auxiliary stack, assuming size of stacks is unlimited

// Input: stack *S* which is *S1* and *S2*, with element *e* to push into stack

// Output: extra element is added to stack

S1.push(e)

if S2.isEmpty() **then**

    S2.push(e)

else then

    temp ← S2.peek()   // returns top of stack without removing element

    if e < temp **then**

        S2.push(e)

    else then

        S2.push(temp)

برای عملیات Pop تنها کافی‌ست از هر دو پشته داده‌ی top را حذف کنیم. در این حالت داده‌ی top پشته‌ی دوم مقداری را نشان می‌دهد که قبل از اضافه شدن عضو حذف شده از پشته‌ی اول، مقدار مینیموم پشته بوده‌است.

Algorithm: Pop(S)

// removes element from top of both stacks

// Input: *S* which is mainly *S1* with *S2* as an auxiliary stack

// Output: top element of *S1* and *S2* is removed and one is returned

e ← S1.pop()

S2.pop()

return e

حال با وجود داشتن پشته‌ی دوم، عملیات FindMin به راحتی با کمک عملیات top انجام می‌شود.

Algorithm: FindMin(S)

// returns top element of auxiliary stack which is always the minimum

// Input: *S* which is mainly *S1* with *S2* as an auxiliary stack

// Output: returns minimum without removing element

e ← S2.peek()   // returns top of stack without removing element

return e

---

4. الف) الگوریتمی برای معکوس کردن یک لیست پیوندی یک‌طرفه طراحی کنید. الگوریتم شما علاوه بر حافظه لازم برای نمایش خود لیست پیوندی، مجاز به استفاده از تنها تعداد ثابت و اندکی واحد حافظه خواهد بود.

ب) لیست پیوندی دوطرفه‌ای را در نظر بگیرید که تعداد گره‌های آن عددی فرد باشد و علاوه بر آن، دو گره مصنوعیِ شروع ($header$) و پایان ($trailer$) نیز داشته باشد. الگوریتمی طراحی کنید که با آن بتوان گره میانی لیست را یافت. (شما در الگوریتم خود، مجاز به استفاده از شمارنده نیستید.)

جواب:

4 – الف) برای حل این سوال نیاز به حلقه‌ای‌ست تا با آن عملیات تکرار را انجام دهیم و از آنجا که می‌توانیم از تعداد ثابت و اندکی واحد حافظه استفاده کنیم، این اجازه را داریم که دو متغیر موقت را به وجود آوریم. بدین صورت از ابتدای لیست پیوندی یک‌طرفه شروع کرده و گره‌ها را هر بار جابه‌جا می‌کنیم که به قول هم‌گروهی فصیحمان «واسه‌ی دفعه‌ی بعد این بشه قبلیِ بعدی».

```
Algorithm: Reverse(L)
    // reverses a linked list
    // Input: linked list L
    // Output: L reversed
    tempPrev ← NULL  // initializing temporary variable needed in loop
    while current ≠ NULL do   // current is the current element of linked list
                              // and starts from the head AKA first elemnt
        tempNext ← next(current)   // next(current) is the next element in list
                                   // tempNext is a temporary variable
        next(current) ← tempPrev
        tempPrev ← current
        current ← tempNext
```

4 – ب) می‌دانیم تعداد گره‌های لیست فرد است، پس همواره گره‌ی میانی وجود دارد. از طرفی header و trailer لیست پیوندی را داریم که شروع و پایان هستند. با استفاده از عملگرهای next و previous از دو طرف لیست شروع به حرکت کرده تا به میان آن برسیم. ما اجازه استفاده از این عملگرها را داریم زیرا لیست پیوندی دو طرفه است و از هر دو سو می‌توانیم در آن حرکت کنیم.

```
Algorithm: FindMiddle(L)
    // finds middle element of a doubly linked list
    // Input: list L with header and trailer
    // Output: returns middle element of L
    start ← next(header)
    end ← previous(trailer)
    while start ≠ end do
        start ← next(start)
        end ← previous(end)
    return start   // or return end as they are now the same
```

5. فرض کنید A مجموعه‌ای متناهی باشد و f تابعی باشد از A به A که $f: A \longrightarrow A$

الف) الگوریتمی طراحی کنید که با آن بتوان تعیین کرد که آیا چنین توابع f ای، «یک- به- یک» هستند یا خیر.

ب) الگوریتمی طراحی کنید که با آن بتوان بزرگترین زیرمجموعه $S \subseteq A$ را به گونه‌ای که تابع $f: S \longrightarrow S$ «یک- به- یک» باشد، تعیین کرد.

<u>جواب:</u>

5 – الف) قصد ما طراحی الگوریتمی‌ست که یک به یک بودن توابع f از A به A را بسنجد، پس برای ورودی باید مجموعه‌ی متناهی A و ضابطه‌ی تابع را بگیرد. در این الگوریتم برای انجام عمل سنجش، به روی اعضای دامنه ضابطه‌ی تابع اعمال شده و مقادیر حاصل در پشته‌ای خالی ذخیره می‌شوند که در اصل برد تابع است.

این روش هم از نظر زمان و هم حافظه بهینه‌تر از سنجش تک تک اعضای دامنه و برد است زیرا پشته کمترین میزان حافظه را مصرف می‌کند و علاوه بر این با این الگوریتم حتماً نیاز به بررسی تمام اعضای دامنه و برد نیست چون به محض برخورد با مقداری که قبلاً در لیست قرار داده شده بوده به راحتی می‌توان نتیجه‌گیری کرد که از دو x متفاوت یک y یکسان به دست آمده‌است و تابع یک به یک نیست.

```
Algorithm: IsInjective(f, A)
    // checks if function from finite set A to A is injective
    // Input: function f and the finite set A
    // Output: if function is injective, True is returned, else False is
    S ← stack()   // new empty stack data structure is created
                  // because stack takes up less space
    for x in A do
        if S.exists(f(x)) then   // exists function can be defined for stack
                                 // using two stacks, one to temporarily store data
            return False
        else then
            S.push(f(x))
    return True   // when S (which is range) is completed without repetition
```

5 – ب) برای یافتن بزرگترین زیرمجموعه‌ی A به نام S که تابع $f\colon S \to S$ یک به یک باشد می‌توانیم از مجموعه‌ی A برای برد، مقادیر تکراری را حذف کنیم یا زیرمجموعه‌ی جدیدی تشکیل دهیم و به بزرگترین حالت ممکن آن را برسانیم. در هر دو راه حل بایستی تمامی مقادیر بررسی شوند، اما از آنجا که ایجاد یک پشته برای ذخیره روشی بهینه‌تر از جستجو و تغییر در یک لیست پیوندی‌ست که به علت سنگینی سرعت کار با آن کندتر از یک پشته است، روش دوم را در این الگوریتم به کار می‌بریم. پس مانند قسمت ب پشته‌ی R را برای ذخیره‌ی داده‌های حاصل از اعمال تابع f بر اعضای دامنه ایجاد کرده و علاوه بر آن پشته‌ای به نام S نیز برای ذخیره xهایی از دامنه که شرط یک به یکی را بر هم نمی‌زنند.

```
Algorithm: LargestSubset(f, A)
    // finds S the largest subset of A where f: S → S is injective
    // Input: function f and finite set A
    // Output: subset S
    R ← stack()   // intended as range of f
    S ← stack()
    for x in A do
        if R.exists(f(x)) is False then
            R.push(f(x))
            S.push(x)
    return S
```

---

6. یک رشته‌بیتی[1]، دنباله‌ای است از صفر یا چند بیت؛ برای مثال، 011001 یک رشته‌بیتی با طول 6 است (به رشته‌ای که طول آن صفر باشد، رشته تهی گفته می‌شود و با λ نمایش داده می‌شود). معکوس[2] یک رشته‌بیتی، با معکوس کردن ترتیب بیت‌های آن رشته به دست میآید؛ برای مثال، معکوس رشته 011001 عبارت است از رشته 100110 .

الف) معکوس یک رشته‌بیتی را به طور بازگشتی تعریف کنید. (فرض کنید w یک رشته و w^R معکوس آن باشد.)

ب) مبتنی بر تعریف بازگشتی، یک الگوریتم بازگشتی برای تعیین معکوس یک رشته‌بیتی طراحی کنید.

پ) درستی الگوریتم بازگشتی خود را ثابت کنید.

<u>جواب:</u>

[1] bit string
[2] reversal

6 – الف) اولین نکته‌ی قابل توجه در مورد سوال این است که منظور از برعکس کردن رشته‌ی بیتی آینه کردن آن است و نه تغییر 0ها به 1 و 1ها به 0 آن‌طور که هم‌گروهی نابغه‌مان برداشت کرد و ساعت‌ها به روی آن وقت گذاشت و با ما بحث کرد و القاب شایسته صدایمان کرد، اصرار بر این‌که نه، کاملاً برداشت ایشان درست است.

حال با درک صحیح سوال و فرض اینکه W یک رشته‌بیتی و W^R معکوس آن است، الگوریتم بازگشتی برای تبدیل W به W^R ابتدا طول رشته را بررسی می‌کند و در صورت برابر بودن آن با 1 بیت، خود رشته را به عنوان W^R باز می‌گرداند. اما در صورت بیشتر بودن طول رشته بیتی، الگوریتم بازگشتی به شکلی تعریف می‌شود که عضو آخر رشته را حذف کرده و در ابتدای یک رشته‌ی جدید قرار می‌دهد، به صورتی که ادامه‌ی آن رشته‌ی جدید دوباره به طور بازگشتی الگوریتم را برای رشته‌ی تغییر یافته فراخوانی می‌کند. بدین صورت رشته تا جایی تغییر یافته و اعضایش حذف می‌شوند که طول آن برابر با 1 بیت شود و فراخوانی متوقف شود.

****

6 – ب) در طراحی الگوریتم تعریف شده در قسمت الف سوال، از ساختار داده‌ی پشته برای ذخیره‌سازی رشته بیتی استفاده می‌کنیم زیرا از نظر حافظه بهینه‌ترین است و می‌توانیم عملیات سریع pop را به کار ببریم.

Algorithm: ReverseBit(W)

// reverses bit string using recursion method

// Input: bit string *W*

// Output: bit string W^R

S ← stack(W)

if length(S) = 1 **then** // *length* can be defined for *S* using auxiliary stack

return S.pop()

else then

return S.pop() + ReverseBit(S) // combining two strings

6 – پ) برای اثبات از استنتاج استقرائی استفاده می‌کنیم، به طوری که درستی الگوریتم بازگشتی را ابتدا برای base case یا وضعیت پایه یعنی برابر بودن طول رشته با 1 بررسی می‌کنیم ($n = 1$)، سپس صحت inductive case یا وضعیت استقرایی را نشان می‌دهیم ($n > 1$) و اثبات کامل خواهد بود.

- ✓ **وضعیت پایه (*base case*)**: رشته‌ی '0' را که طولی برابر یک دارد در نظر می‌گیریم. با اعمال الگوریتم بر آن، شرط اول برقرار شده و خروجی خودِ رشته برگردانده می‌شود. پس الگوریتم برای base case درست است.

- ✓ **وضعیت استقرایی (*inductive case*)**: رشته‌ی ’110‘ را که طولی بزرگ‌تر از یک بیت دارد وارد الگوریتم کرده و می‌بینیم ابتدا ’0‘ حذف شده و در رشته‌ی نهایی قرار داده می‌شود، سپس ’11‘ دوباره وارد الگوریتم شده و ’1‘ راستی باز حذف و اضافه می‌شود و در انتها ’1‘ مانده باز وارد الگوریتم شده و با روبه‌رویی با شرط برابر بودن طول رشته با یک، خود آن بازگردانده می‌شود. با ردگیری کردن مسیر به عقب مشاهده می‌کنیم رشته‌بیتی ’011‘ شکل گرفته و در نهایت از الگوریتم خارج می‌شود. به همین شکل برای رشته‌بیتی نمونه به نام B با طول n، عضو nام هر بار حذف شده و طول رشته به $n - 1$ تغییر میابد و این عمل آنقدر تکرار می‌شود که طول B برابر با 1 بیت شده و recursionها به پایان برسد. پس الگوریتم برای inductive case نیز درست است.

حال با توجه به این‌که n همواره کوچک‌تر از تعداد دفعات فراخوانی recursion است و الگوریتم برای دو حالت بالا اثبات شده‌است، با استقرا به این نتیجه می‌رسیم که خروجی در هر صورت خواسته‌های الگوریتم را به جا می‌آورد یعنی برای ورودی رشته‌بیتی W با طول n الگوریتم W^R را به دست می‌آورد که حتماً بر عکس شده‌ی W می‌باشد و اثبات کامل است.

---

7. الف) با این فرض که $a_1, a_2, \cdots, a_n$ اعداد صحیح مثبت باشند، درستی رابطه بازگشتی زیر را نشان دهید.

$$gcd(a_1, a_2, \cdots, a_n) = gcd\big(a_1, gcd(a_2, a_3, \cdots, a_n)\big)$$

ب) با مبنا قرار دادن رابطه بازگشتی، یک الگوریتم بازگشتی کارا برای محاسبه بزرگترین مقسوم علیه مشترک n عدد صحیح مثبت طراحی کنید.

پ) اکنون یک الگوریتم بازگشتی طراحی کنید که با آن بتوان، بزرگترین مقسوم علیه مشترک n عدد صحیح مثبت را به شکل ترکیب خطی آن اعداد نوشت. به بیان دقیق‌تر، الگوریتم شما باید اعداد $a_1, a_2, \cdots, a_n$ را به عنوان ورودی بگیرد و به عنوان خروجی، اعداد صحیح $x_1, x_2, \cdots, x_n$ را بیابد به قسمی که رابطه زیر برقرار باشد.

$$gcd(a_1, a_2, \cdots, a_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n$$

<u>جواب:</u>

7 – الف) برای اثبات با استقرا نیاز به بررسی دو حالت داریم: base case یا وضعیت پایه و inductive case یا وضعیت استقرایی. بر این اساس در اثبات این رابطه‌ی بازگشتی داریم:

- ✓ **وضعیت پایه (*base case*)**: پایه‌ترین حالت این رابطه‌ی بازگشتی زمانی‌ست که $n = 1$:

$gcd(a_1, 0) = gcd(a_1, gcd(0, 0))$

که $gcd(0, 0) = 0$ و بدیهی‌ست که رابطه بالا برقرار است.

- ✓ **وضعیت استقرایی (*inductive case*)**: حال این رابطه را با $n = 2$ بررسی می‌کنیم:

$gcd(a_1, a_2) = gcd(a_1, gcd(a_2, 0))$

از آنجا که بنا بر الگوریتم اقلیدس می‌دانیم $gcd(a_2, 0) = a_2$ می‌باشد، صحت رابطه بالا نیز قابل مشاهده است.

اما برای $n = 3$ رابطه‌ی بازگشتی به شکل روبه‌روست: $gcd(a_1, a_2, a_3) = gcd(a_1, gcd(a_2, a_3))$

به قصد اثبات این رابطه فرض می‌کنیم a_1 و a_2 و a_3 سه عدد طبیعی هستند که نسبت به هم اول نیستند.

اگر $gcd(a_2, a_3) = b$، آنگاه دو حالت وجود دارد:

1) اگر b مقسوم‌علیه a_1 باشد بر اساس فرض پیشین، مقسوم‌علیه a_2 و a_3 نیز است. حال اگر فرض کنیم b بزرگترین مقسوم‌علیه مشترک a_1و a_2 و a_3 نیست و عددی به عنوان c مقسوم‌علیه مشترک این سه است که $c > b$، آنگاه c مقسوم‌علیه مشترک a_2 و a_3 نیز هست و بنا بر الگوریتم اقلیدس c باید مقسوم‌علیه b نیز باشد (زیرا b بزرگترین مقسوم‌علیه a_2 و a_3 است) که در این صورت $c \leq b$ و این خلاف فرض $c > b$ می‌باشد. پس c وجود ندارد و همان b بزرگترین مقسوم‌علیه مشترک a_1و a_2 و a_3 است.

2) اگر b مقسوم‌علیه a_1 نباشد از آنجا که a_1و a_2 و a_3 نسبت به هم اول نیستند، باید مقسوم‌علیه مشترکی داشته باشند. این عدد که مقسوم‌علیه هر سه a_1و a_2 و a_3 است منطقاً مقسوم‌علیه a_2 و a_3 نیز هست و بر اساس الگوریتم اقلیدس، مقسوم‌علیه b هم می‌باشد. پس این عدد یک مقسوم‌علیه مشترک a_1 و b نیز بوده و بنابراین این دو نسبت به هم اول نبوده و دارای بزرگترین مقسوم‌علیه مشترکی هستند که آن را فرضاً c می‌نامیم؛ حال این c منطقاً مقسوم‌علیه مشترک a_2 و a_3 نیز هست و بنابراین یک مقسوم‌علیه مشترک هر سه عدد a_1و a_2 و a_3 می‌باشد. اگر فرض کنیم c بزرگترین مقسوم‌علیه مشترکشان نیست، پس عددی دیگر به نام d وجود دارد که $d > c$ و مقسوم‌علیه مشترک هر سه عدد ابتدایی‌ست. به مانند قبل، از آنجا که d مقسوم‌علیه a_2 و a_3 است بر اساس الگوریتم اقلیدس مقسوم‌علیه b (که GCD آن دو عدد است) نیز می‌باشد و در نتیجه مقسوم‌علیه مشترک a_1 و b است و باز بنا بر الگوریتم اقلیدس، مقسوم‌علیهی برای GCD این عدد (یعنی c) هم بوده و پس $d \leq c$ که این نتیجه، فرض پیشین را نقض می‌کند. بنابراین چنین dای وجود ندارد و c بزرگترین مقسوم‌علیه a_1و a_2 و a_3 می‌باشد و اثبات کامل است.

دیدیم که رابطه‌ی بازگشتی برای سه ورودی صحت دارد. بر این پایه می‌توانیم برای هر n بزرگتر همواره اثبات را تکرار کنیم بدین صورت که فرضاً اگر $n = 4$ آنگاه:

$$gcd(a_1, a_2, a_3, a_4) = gcd(a_1, gcd(a_2, a_3, a_4)) = gcd(a_1, gcd(a_2, gcd (a_3, a_4)))$$

در نتیجه بنا بر استنتاج استقرائی رابطه‌ی بازگشتی برای n عدد ثابت می‌شود.

****

7– ب) به کمک رابطه بازگشتی اثبات شده و همچنین ساختار داده‌ی بهینه و کم حجم پشته برای ذخیره‌ی n داده، الگوریتم بازگشتی را به این صورت داریم که هر بار با pop کردن عضو اول، پشته‌ی تغییر یافته با یک عضو کمتر را دوباره در تابع قرار می‌دهیم؛ این تکرار تا جایی ادامه پیدا می‌کند که پشته خالی شود و آنگاه در خروجی صفر را باز می‌گردانیم زیرا بنا بر الگوریتم اقلیدس gcd(a, 0) = a و ما قصد رسیدن به آن را داریم تا در آخر با اعضای حذف شده از پشته، به ترتیب از داخل به خارج recursion برگشته و با مقادیر GCD را محاسبه می‌کنیم.

Algorithm: GCDN(numbers)

// finds GCD of *n* numbers using recursion method

// Input: *n* integers

// Output: greatest common divisor of the *n* integers

S ← stack(numbers)

if S.isEmpty() **then**

    return 0   // gcd(0, 0) = 0

else then

    x ← S.pop()

    y ← GCDN(S)

    if y = 0 **then**

        return x

    else then

        while y ≠ 0 **do**

            c ← x mod y

            x ← y

            y ← c

        return x

7– پ) برای به دست آوردن بزرگترین مقسوم‌علیه مشترک n عدد به شکل ترکیب خطی، با استفاده از قسمت ب GCD اعداد را محاسبه کرده و همرا با اعداد به عنوان ورودی می‌گیریم تا با انجام تقسیم‌های مکررِ آن مقدار بر اعداد ورودیِ الگوریتم و سپس تعداد اعداد، مقادیر x را پیدا کنیم (علت تقسیم بر تعداد، رعایت نسبت است). با مرتباً ذخیره کردن همه‌ی مقادیر در یک رشته، ترکیب خطی را بنا بر الگوی زیر یافته و باز می‌گردانیم.

$$gcd(a_1, a_2, \cdots, a_n) = a_1x_1 + a_2x_2 + \cdots + \ a_nx_n$$

علت ورودی گرفتن GCD و n این است که با حذف اعضا از پشته این مقادیر تغییر کرده و نمی‌توانیم در هر recursive call آن‌ها را مجددا محاسبه کنیم. مقدار n در خود الگوریتم برای بار اول حساب می‌شود و در دفعات دیگر ورودی داده می‌شود پس کاربر نیاز ندارد آن را وارد کند.

Algorithm: GCDString(gcd, numbers, n = 0)

// finds GCD of *n* numbers as a linear combination

// Input: array of *n* integers

// Output: greatest common divisor of the *n* integers as a linear combination

if n = 0 **then** // the user does not need to enter this

n ← length(numbers)

A ← stack(numbers)

if length(A) = 1 **then**

a ← A.peek() // returns top element of stack without removal

x ← (gcd / a) / n

text ← string(x) + “ × ” + string(a)

return text

else then

a ← A.pop()

x ← (gcd / a) / n

text ← string(x) + “ × ” + string(a) + “ + ”

return text + GCDString(gcd, numbers, n)

8. عدد صحیح m را (که کوچکتر از $n!$ باشد) می‌توان به شکل عبارت $a_{n-1}(n-1)! + \cdots + a_2 2! + a_1 1!$ نوشت. در این عبارت، هر a_i ، عددی صحیح است با قید $0 \le a_i \le i$. به این نحوه نمایش یک عدد صحیح، بسط کانتوری[3] آن عدد گفته می‌شود.

الف) بسط کانتوری اعداد 5 ، 84 و 1000 را بیابید.

ب) الگوریتمی طراحی کنید که از روی بسط کانتوری یک عدد صحیح، خود عدد صحیح را تعیین کند.

پ) الگوریتمی طراحی کنید که با آن بتوان بسط کانتوری یک عدد صحیح را یافت.

<u>جواب:</u>

8 – الف) برای محاسبه بسط کانتوری اعداد به این گونه عمل می‌کنیم که بزرگترین فاکتوریل ممکن که از عدد مورد نظر کوچک‌تر باشد را در نظر گرفته و عدد را بر آن تقسیم می‌کنیم تا خارج قسمت و باقی‌مانده را به دست آوریم سپس این عمل را تا جایی برای باقی‌مانده (حتی باقی‌مانده‌ی صفر) تکرار می‌کنیم که به $1!$ برسیم، سپس ضرب هر خارج قسمت در فاکتوریل مربوطه را با هم جمع می‌کنیم که این بسط m است و با خود آن برابر است. برای نوشتن بسط کانتوری اعداد 5 و 84 و 1000 به همین شکل داریم:

$5 = 2! \times 2 + 1! \times 1$

$84 = 4! \times 3 + 3! \times 2 + 2! \times 0 + 1! \times 0$

$1000 = 6! \times 1 + 5! \times 2 + 4! \times 1 + 3! \times 2 + 2! \times 2 + 1! \times 0$

8 – ب) برای نوشتن الگوریتم ReverseCantor بنا بر روش شرح داده شده در قسمت الف، می‌توانیم با گرفتن پشته‌ی مقادیر خارج قسمت‌ها به عنوان ورودی، بسط را از آخر به اول برگردیم تا مقدار باقی‌مانده (که همان‌طور که دیدیم جمع جملاتِ قبلی بسط است) به جای کوچک شدن همواره بزرگتر شود تا با خود عدد صحیح m برابر شود.

```
Algorithm: ReverseCantor(S)

    // finds an integer from its Cantor expansion
    // Input: stack of every a AKA the quotients in Cantor expansion
    // Output: returns integer m

    r ← 0   // the remainder that will get larger and closer to m with each iteration
    i ← 1
    while S.isEmpty() is False do
        r ← S.pop() × i! + r   // S.pop() is a
        i ← i + 1
    return r
```

---

[3] Cantor

8 – پ) با توجه به توضیحات داده شده در مورد روش به دست آوردن بسط کانتوری یک عدد صحیح در قسمت الف سوال، برای یافتن بسط زیر الگوریتمی طراحی می‌کنیم که با گرفتن عدد صحیح، عملیات شرح داده شده را بر آن انجام دهد و در خروجی یک پشته از مقادیر a بازگرداند. علت انتخاب پشته برای ساختار داده‌ی این الگوریتم کم حجم‌تر و سریع‌تر بودن آن است.

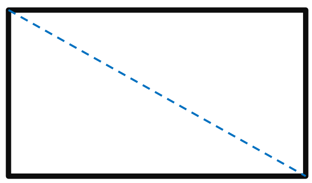
$$m = a_{n-1}(n-1)! + \cdots + a_2 2! + a_1 1!$$

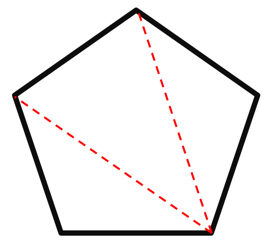
Algorithm: CantorExpansion(m)

```
    // finds the Cantor expansion of integer
    // Input: integer m
    // Output: returns stack of every a in Cantor expansion
    S ← stack()
    n ← 1
    while n! < m do
        n ← n + 1
    for i ← 0 to (n – 1) do
        a ← m div i!
        r ← m mod i!
        S.push(a)
    return S
```

9. گاهی کار با یک چندضلعی، مستلزم این است که آن را به تکه‌هایی بشکنیم. یک شیوه معمول برای تجزیه یک چندضعلی، این است که با رسم حداکثر تعداد قطرهای نامتقاطعِ چندضعلی، آن را به تعدادی مثلث تجزیه کنیم. به این عملیات، مثلث‌بندی[4] چندضلعی گفته می‌شود. هر چندضعلی را میتوان به حداقل یک طریق، مثلث‌بندی کرد.

برای مثال، یک مستطیل را میتوان با رسم یکی از دو قطر آن، به دو مثلث تجزیه کرد. و یک پنجضلعی را می‌توان با رسم دو قطر نامتقاطع آن، به سه مثلث تجزیه کرد.





الف) با این فرض که $n \geq 3$ باشد، ثابت کنید هر چندضلعی ساده P با n ضلع، به هر شیوه‌ای که به تعدادی مثلث تجزیه شود، شکل حاصل شامل $n-2$ مثلث و $n-3$ قطر خواهد بود.

ب) یک الگوریتم بازگشتی برای مثلث‌بندی هر چندضلعی ساده P طراحی کنید.

جواب:

$9-$ الف) راس فرضی a را در یک n ضلعی برای شروع انتخاب می‌کنیم. سپس از آن راس تمامی قطرهای ممکن را رسم می‌کنیم. از آنجایی که خود راس و دو راس مجاور در این فرایند شرکت نداشته‌اند، تعداد قطرهای رسم شده برابر با $n-3$ می‌شود. قابل به ذکر است که اگر هر قطر ممکن دیگری از هر راس چندضلعی رسم شود، حتما با حداقل یک قطر رسم شده تقاطع خوهد داشت. در نتیجه تعداد حداکثر قطرهای ممکن همان $n-3$ خواهد بود. بدین صورت تعداد مثلث‌ها منطقاً برابر با $\frac{1}{3}$ تعداد اضلاع با احتساب اشتراکات است و از آنجا که هر قطر رسم شده در تشکیل 2 مثلث شرکت دارد نتیجه می‌گیریم:

$$Triangles = \frac{2 \times (n-3) + n}{3} = \frac{3n-6}{3} = n-2$$

****

[4] triangulation

9 – ب) برای طراحی الگوریتم مثلث بندی بازگشتی مخصوص به مثلث های محدب (ساده) لیست نام رئوس را به عنوان ورودی دریافت می‌کنیم (رئوس در لیست به صورت مرتب شده فرض می‌شوند، یعنی هر راس با راس بعدی در لیست ضلعی تشکیل می‌دهد و راس آخر و اول نیز با یک ضلع به هم متصل هستند). اگر تعداد رئوس برابر 3 بود خود لیست را به عنوان خروجی تحویل می‌دهیم (زیرا خود یک مثلث است) و اگر طول لیست بیشتر از 3 بود عضوهای اول تا سوم را (که اولین مثلث تشکیل شده هستند) در یک متغیر ذخیره می‌کنیم، سپس عضو دوم لیست را حذف می‌کنیم و متغیر و لیست جدید را به شکل بازگشتی در تابع در return قرار می‌دهیم و عملیات را در هر فراخوانی تکرار می‌کنیم تا جایی که تعداد اعضای لیست 3 شوند.

Algorithm: Triangulation (L)

```
    // triangulates a simple polygon
    // Input: ordered list L of polygon's points
    // Output: names of triangles' points, triangles divided by comma
    L ← list()
    if length(L) > 3 then
        T ← string(L[1]) + string(L[2]) + string(L[3])   //combining strings
        L.remove(2)   // removes the 3th element of the list
        return   T + ", "  + Triangulation(L)
    else then
        T ← string(L[1]) + string(L[2]) + string(L[3])
        return T
```

10. ویراستار کتاب «تاریخ علم جهان» می‌خواهد این را بداند که در چه دوره زمانی، بیشترین تعداد از دانشمندان برجسته زنده بوده‌اند. منظور از «دانشمندان برجسته» افرادی است که تاریخ تولد و تاریخ مرگ آن‌ها در کتاب ذکر شده باشد. (ذکری از دانشمندان زنده در کتاب به میان نیامده است.) الگوریتمی طراحی کنید که نمایه[5] کتاب را به عنوان ورودی بگیرد و به عنوان خروجی، دوره زمانی مورد نظر و نام دانشمندان زنده در آن دوره را برگرداند. سطرهای نمایه کتاب، به ترتیب الفبایی مرتب شده‌اند و هر یک از آن‌ها، سال تولد و سال مرگ یک فرد را مشخص می‌کند. در موردی که شخص A در همان سالی که شخص B متولد شده باشد، از دنیا رفته باشد، مرگ شخص A را قبل از تولد شخص B به حساب آورید.

<u>جواب:</u>

این الگوریتم با تشکیل یک دیکشنری برای هر فرد کار می‌کند که در آن تمام بازه‌های ممکن در زمان زندگی وی و تعداد دانشمندان زنده در هر بازه ذخیره می‌شود.

هر نفر می‌تواند در بازه‌های 1 تا n ساله از تولد تا قبل مرگ خود به عنوان دانشمند زنده شمرده شود، در نتیجه ما برای یافتن بازه‌ای که بیشترین تعداد دانشمندان زنده را دارد، تعداد افراد زنده در این بازه ها را برای دانشمندان حساب می‌کنیم به این جهت که از بین آنها بازه مورد نظر را پیدا کنیم. برای این کار در ابتدای تابع یک دیکشنری ایجاد کرده تا در آن، بازه و لیست افراد زنده در آن بازه را در هر مرحله ذخیره کنیم. سپس در یک حلقه بازه‌های مختلف زندگی هر فرد را برای محاسبه بیشترین افراد زنده در آن بازه‌ها سنجیده و در آخر با مقایسه اعداد به دست آمده، بازه و افراد زنده در آن بازه را باز می‌گردانیم.

[5] index

Algorithm: SearchBook(index)

```
// finds a period of time with the largest number of simultaneously living scientists
// Input: the index of book, shaped like → {name : (birthyear, deathyear)}
// Output:

mainDict ← dict(index)   // intended to save {time period : list of names}
for name in mainDict do
    sciL ← list()   // intended to save list of simultaneously living scientists
    // dictionary[key] returns value
    birth ← (mainDict[name])[1]    // 1st element of (birthyear, deathyear)
    death ← (mainDict[name])[2]    // 2nd element of (birthyear, deathyear)
    for year ← birth to death do
        period ← list(birth, birth + year)
        subL ← list[]
        subL.add(period)    // first element of list is time period, then names
        for scientist in index do
            if (mainDict[scientist])[1] < birth + year and (mainDict [scientist])[2] > birth then
                subL.add(scientist)
         sciL.add(subL)
    max ← 0
    for sub in sciL do
        if length(sub)  > max then
            max ← length(sub)
            maxList ← sub
    maxTime ← maxList[1]
    maxList ← maxList[2:]
    mainDict.add(maxTime: maxList)
// we suppose dictionary.value(1) returns first value and dictionary.key(1) returns first key
maxLength ← length(mainDict.value(1))
maxPeriod ← mainDict.key(1)
for period in mainDict do
    test ← length(mainDict[period])
    if test > maxLength then
        maxLength ← test
        maxPeriod ← period
return maxPeriod, mainDict[maxPeriod]
```