فصل ٦

نگاهی مختصر بر ارائه دو سمینار

١.٦ LOOP INVARIANT

LOOP INVARIANT چیست ؟

این که ما بدانیم درهر حلقه چه اتفاقی می افتد کار بسیار مشکلی است .
حلقه های بی پایان یا حلقه هایی که بدون رسیدن به هدف مورد نظر پایان می پذیرند، یک مشکل رایج در برنامه نویسی کامپیوتر هستند.در این زمینه این Loop Invariant است که به ما کمک می کند.
در واقع ما از Loop Invariant ها استفاده می نماییم تا بدانیم آیا یک الگوریتم درست کار می کند یا خیر.
Loop Invariant یک یا گاهی چند عبارت فرما لیته است که رابطه بین متغیر ها در برنامه‌ی مارا بیان می کند که قبل از اجرای حلقه، هر زمان داخل حلقه و همچنین در انتهای حلقه درست باقی می ماند.

در این قسمت سعی بر آن است که مفاهیم اولیه LOOP INVARINT و طرز استفاده از آن به صورت خیلی مختصر توضیح داده شود و در انتها نیز چند مثال مختلف از استفاده از آن آورده شده است .

در این قسمت یک الگوی کلی استفاده از Loop Invariant را مشاهده

می نمایید:

```
    ...
// the Loop Invariant must be true here
while ( TEST CONDITION ) {
        //top of the loop
        ...
        //bottom of the loop
        //the Loop Invariant must be true here
}
// Termination + Loop Invariant ⇒ Goal
...
```

وقتی که مقادیر متغیر ها در حلقه تغییر کنند درستی Loop Invariant تغییری نمی کند. برای مشخص نمودن Loop Invariant باید به این نکته توجه نمود که عبارات بسیاری وجود دارند که قبل و بعد از هر تکرار حلقه ثابت و درست باقی می مانند اما عبارتی Loop Invarint است که با کار حلقه مرتبط است و از آن مهم تر این که با پایان یافتن حلقه درستی Loop Invariant ما را به نتیجه دلخواه و مورد انتظار از آن حلقه می رساند. در این قسمت چند اصطلاح را مشخص می نماییم :

Pre-condition:آن چیزی است که باید قبل از اجرای حلقه درست باشد.

Post-condition: آن چیزی است که بعد از اجرای کامل حلقه درست باقی می ماند و به ازای شرط خروج از حلقه در Loop Invariant به دست می آید که همان نتیجه مورد انتظار از حلقه است.

Loop Variant: شرطی است که اجرای حلقه را کنترل می کند و در واقع خروج از حلقه‌یا ادامه‌ی حلقه را کنترل می نماید.

برای چک کردن کار یک حلقه باید نکات زیر را مورد توجه قرار دهیم :

۱ـ *مطمئن شویم* Pre-condition *قبل از اینکه حلقه آغاز شود درست است .*

۲ـ *نشان دهیم* Loop Invariant *برای* Pre-condition *درست است.*

۳ـ *نشان دهیم اجرای* Loop Variant *اثر جانبی بر درستی* Loop Invariant *ندارد.*

۴ـ *نشان دهیم* Loop Invariant *بعد از هر اجرای حلقه درست است.*

۵ـ نشان دهیم‌به مجرد پایان حلقه Loop Invariant *دلالت بر درستی* Post-condition *دارد.*

٦ـ نشان دهیم هر تکرار حلقه Loop Variant *راافزایش یا کاهش می دهد.*

برای نشان دادن درستی مورد چهارم باید نشان دهیم‌اگر loop Invariant قبل از یک تکرار حلقه درست است قبل از تکرار بعدی نیز درست می ماند.

در این قسمت باید به نکته جالبی توجه نماییم و آن این است که اجرای مراحل ۴ـ۱ مشابه استقرای ریاضی است .در واقع با نشان دادن مورد ۲ پایه استقرا را بنا نهاده ایم وبا نشان دادن مورد ۴ گام های استقرا را پیموده ایم با این تفاوت که مورد ۵ باعث می شود که ما یک نوع استقرای محدود داشته باشیم .در واقع با پایان یافتن حلقه استقرا متوقف می گردد.
حال با ذکر چند مثال انجام مراحل فوق را نشان می دهیم :

۱)الگوریتمی که مجموع اعداد صحیح از 1 تاn را محاسبه می نماید:

```
1. int sum=0;
2. int k=0;
3. while(k < n){
4.       k++;
5.       sum+=k;
6.            }
```

....

$precondition : sum = ۰, k = ۰$

$postcondition : sum = \sum_{i=۱}^{n} i$

$loop\ \ invariant : sum_k = \sum_{i=۱}^{k} i$

INITIALIZATION:

همان طور که مشاهده می گردد loop invariant برای precondition درست است :

$$k = 0 \Rightarrow sum = \sum_{i=1}^{0} i = 0 = sum$$

MAINTENANCE:

حال نشان می دهیم اگر loop invariant برای مرحله j ام از تکرار حلقه درست باشد برای مرحله j+1 ام از تکرار حلقه نیز درست است :

$$sum_j = \sum_{j=1}^{k_{j+1}} j \ , \ \ k_{j+1} = k_j + 1$$

$$sum_{j+1} = sum_j + k_{j+1} = (\sum_{j=1}^{k_j} j) + k_{j+1} = (\sum_{j=1}^{k_j} j) + k_j + 1 = \sum_{j=1}^{k_{j+1}} j$$

TERMINATION:

حال شرط خروج از حلقه را چک می نماییم که به ازای آن loop invariant عبارت postcondition را به ما می دهد.شرط خروج از حلقه $k = n$ می باشدکه به ازای آن داریم :

$$sum = \sum_{i=1}^{n} i \equiv postcondition$$

۲)الگوریتمی که فاکتوریل عددی صحیح وبزرگتر از صفر را محاسبه می نمایید:

```
1. int factorial(n){
2.     i =1;
3.     fact =1;
4.     while(i ! = n){
5.       i++;
6.       fact=fact×i; }
7.      return fact;
8.              }
```

....

$precondition : n \geq 1$

$loop\ \ invariant : fact = i\,!$

$postcondition : fact = n\,!$

....

INITIALIZATION:

$i = ۱ \Rightarrow fact = ۱! = ۱ \Rightarrow fact = i\,!$

MAINTENANCE:

$fact' = j'!\ ,\ \ j = j' + ۱\ ,\ \ fact = fact' \times j$

$\Rightarrow fact = j'! \times j = j' \times (j' + ۱) = (j' + ۱)! = j! \Rightarrow fact = j!$

TERMINATION:

$i = n\ \ ,\ \ fact = i! \Rightarrow fact = n! \equiv postcondition$

نکته : این الگوریتم ،الگوریتمی است که در آن اهمیت شرط precondition را به خوبی نشان می دهد،زیرا اگر شرط$n \geq ۱$ را در نظر نگرفته وn را برابر صفر بگیریم حلقه بی نهایت بار تکرار خواهد شدو ما را به نتیجه دلخواه نخواهد رساند.

۳)الگوریتمی که بزرگترین مقسوم علیه مشترک بین دو عدد صحیح بزرگتر از صفر را بر می گرداند:

```
int gcd(int m ,int n){
      int mprime = m;
      int nprime = n;
      while(mprime ! =nprime){
        if(mprime > nprime)
          mprime - = nprime;
        else
          nprime - = mprime;}
      return mprime;
                    }
```

....

$precondition : m, n \in Z^{+}$

$loop \;\; invariant: gcd[m,n] = gcd[mprime, nprime]$

$postcondition: gcd[m,n] = mprime$

....

INITIALIZATION:

$mprime = m \quad nprime = n \Rightarrow gcd[m,n] = gcd[mprime, nprime]$

MAINTENANCE:

$gcd[m,n] = gcd[mprime_i, nprime_i]$

- $if(mprime_i > nprime_i):$
  $mprime_{i+1} = mprime_i - nprime_i, \quad nprime_{i+1} = nprime_i$
  $\Rightarrow gcd[mprime_{i+1}, nprime_{i+1}] = gcd[mprime_i - nprime_i, nprime_i] =$
  $gcd[mprime_i, nprime_i] = gcd[m,n]$

- $else:$
  $nprime_{i+1} = nprime_i - mprime_i, \quad mprime_{i+1} = mprime_i$
  $\Rightarrow gcd[mprime_{i+1}, nprime_{i+1}] = gcd[mprime_i, nprime_i - mprime_i] =$
  $gcd[mprime_i, nprime_i] = gcd[m,n]$

TERMINATION:

$mprime = nprime \Rightarrow gcd[m,n] = gcd[mprime, nprime] =$
$gcd[mprime, mprime] = mprime \equiv postcondition$

۴)الگوریتمی که k جمله‌ی اول بسط تیلور e^n را محاسبه می نماید:

1. double TaylorExp(double n ,int k){

```
double result =1;
int count =0;
int denom=1;
while (count<k){
  count ++;
  denom×=count;
  result+=pow(n,count)/denom;
                    }
return result;
                                    }
```

....

$precondition : n \in Z \quad k \in N, k > ۰$

$loop\ \ invariant : result = ۱ + n + \frac{n^{۲}}{۲!} + \cdots + \frac{n^{count}}{count!} \quad , denom = count!$

$postcondition : ۱ + n + \frac{n^{۲}}{۲!} + \cdots + \frac{n^{K}}{K!} = result$

....

INITIALIZATION:

$count = ۰, \quad denom = ۱, \quad result = ۱ \Rightarrow \frac{n^{۰}}{۰!} = ۱ = result$

MAINTENANCE:

$result_i = ۱ + n + \frac{n^{۲}}{۲!} + \cdots + \frac{n^{count_i}}{count_i!} \quad , denom_i = count_i!$

$\Rightarrow denom_{i+۱} = denom_i \times count_{i+۱} = (count_i!) \times count_{i+۱} = (count_{i+۱})!,$

$result_{i+۱} = result_i + \frac{n^{count_{i+۱}}}{(count_{i+۱})!} = ۱ + n + \frac{n^{۲}}{۲!} + \cdots + \frac{n^{count_i}}{count_i!} + \frac{n^{count_{i+۱}}}{(count_{i+۱})!}$

TERMINATION:

$count = k \Rightarrow result = ۱ + n + \frac{n^{۲}}{۲!} + \cdots + \frac{n^{K}}{K!} \equiv postcondition$

۵)با استفاده از loop invariant نشان دهیدالگوریتم زیر جمع دو عدد طبیعی را انجام می دهد:

function *add(y,z)*

comment *return* $y + z$*, where* $y,z \in \mathbf{N}$

1. $x := 0$; $c := 0$; $d := 1$;
2. **while**$(y > 0) \vee (z > 0) \vee (c > 0)$**do**
3. $a := y \bmod 2$;
   $b := z \bmod 2$;
4. **if** $a \oplus b \oplus c$ **then** $x := x + d$;
5. $c := (a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$;
6. $d := 2\,d$; $y := \lfloor y/2 \rfloor$;
   $z := \lfloor z/2 \rfloor$;
7. **return**(x)

....

loop invariant : $(y_j + z_j + c_j)d_j + x_j = y_0 + z_0$

اگر y و z اولیه را y_0 و z_0 در نظر بگیریم قبل از شروع حلقه داریم :

$x_0 = 0, c_0 = 0, d_0 = 1 \Rightarrow$

$(y_j + z_j + c_j)d_j + x_j = (y_0 + z_0 + 0) \times 1 + 0 = y_0 + z_0$

حال فرض می نماییم loop invariant برای مرحله‌ی j ام برقرار است یعنی :

$(y_j + z_j + c_j)d_j + x_j = y_0 + z_0$

همچنین طبق روال حلقه داریم :

$a_{j+1} = y_j \bmod 2 \quad , b_{j+1} = z_j \bmod 2$

$y_{j+1} = \lfloor y_j/2 \rfloor \quad , z_{j+1} = \lfloor z_j/2 \rfloor \quad , d_{j+1} = 2 \times d_j$

همچنین همان طور که در خط پنجم مشاهده می شود c_{j+1} که در این مرحله‌از a_{j+1} و b_{j+1} و c_j به دست می آید تنها وقتی یک است که دو تا از a_{j+1} و b_{j+1} و c_j یک باشند،پس می توان c_{j+1} را به صورت زیر نوشت :

$c_{j+۱} = \lfloor (a_{j+۱} + b_{j+۱} + c_j) \ / \ 2 \ \rfloor$

به همین صورت هم مطابق خط چهارم وقتی x_j با d_j جمع شده و $x_{j+۱}$ را به وجود می آورند که عبارت $a_{j+۱} \oplus b_{j+۱} \oplus c_j$ یک باشد پس می توان $x_{j+۱}$ را به صورت زیر تعریف نمود:

$x_{j+۱} = x_j + d_j((a_{j+۱} + b_{j+۱} + c_j) \bmod 2)$

برای‌ادامه اثبات نیاز به یک رابطه مهم دیگر که آن را رابطه (*) می نامیم وبه صورت زیر است نیز داریم :

$2\lfloor \ n \ /2 \rfloor + (\ n \bmod 2) = n \quad \forall n \in \mathbf{N} \qquad\qquad (*)$

حال طرف اول تساوی loop invariant را نوشته وداریم :

$$(y_{j+۱} + z_{j+۱} + c_{j+۱})d_{j+۱} + x_{j+۱} =$$
$$(\lfloor y_j/۲ \rfloor + \lfloor z_j/۲ \rfloor + \lfloor (y_j \bmod ۲ + z_j \bmod ۲ + c_j)/۲ \rfloor) \times ۲d_j + x_j +$$
$$d_j\,((y_j \bmod ۲ + z_j \bmod ۲ + c_j) \bmod ۲) = (\lfloor y_j/۲ \rfloor + \lfloor z_j/۲ \rfloor) \times ۲d_j$$
$$+x_j + (\lfloor (y_j \bmod ۲ + z_j \bmod ۲ + c_j)/۲ \rfloor) \times ۲d_j +$$
$$d_j\,((y_j \bmod ۲ + z_j \bmod ۲ + c_j) \bmod ۲) \underline{\underline{(*)}} (\lfloor y_j/۲ \rfloor + \lfloor z_j/۲ \rfloor) \times ۲d_j$$
$$+x_j + d_j\,(y_j \bmod ۲ + z_j \bmod ۲ + c_j) = \lfloor y_j/۲ \rfloor \times ۲d_j +$$
$$d_j\,(y_j \bmod ۲) + \lfloor z_j/۲ \rfloor \times ۲d_j + d_j\,(z_j \bmod ۲) + c_j \times d_j + x_j \underline{\underline{(*)}}$$
$$(y_j + z_j + c_j)\,d_j + x_j = y_۰ + z_۰$$

حال زمانی را در نظر می گیریم که حلقه خاتمه پیدا می کند،اگر بعد از k تکرار حلقه خاتمه یابد طبق loop invarint داریم :

$(y_k + z_k + c_k)d_k + x_k = y_۰ + z_۰$

همچنین چون هر بار y و z در هر تکرار به مقادیر $\lfloor y/2 \rfloor$ و $\lfloor z/2 \rfloor$ کاهش می یابند زمان خروج از حلقه یعنی بعد از k امین تکرار داریم :

$y_k = z_k = c_k = ۰ \Rightarrow x_k = y_۰ + z_۰$

پس مشاهده می شود که وقتی الگوریتم به پایان می رسد مقدار x ای که بر گردانده می شود برابر مجموع y و z اولیه‌ی ماست .

۶)با استفاده از loop invariant نشان دهیدالگوریتم زیر ضرب دو عدد طبیعی را انجام می دهد:

function *multiply(y,z)*

comment *return y z , where y , z* $\in$ **N**

```
x :=0
while (z >0)do
  x :=x + y × (z mod 2);
  y := 2 y;    z :=⌊ z/2 ⌋;
return(x)
```

....

loop invariant : $x_j + y_j \times z_j = y_\circ \times z_\circ$

مطابق مثال قبل مقادیر اولیه y و z را $y_\circ$ و $z_\circ$ در نظرمی گیریم ، درابتدا $x_\circ = \circ$ بوده و داریم :

$x_\circ = \circ \Rightarrow x_j + y_j \times z_j = x_\circ + y_\circ \times z_\circ = y_\circ \times z_\circ$

حال فرض می نماییم loop invariant برای مرحله‌ی j ام برقرار است یعنی :

$x_j + y_j \times z_j = y_\circ \times z_\circ$

همچنین طبق روال حلقه داریم :

$x_{j+۱} = x_j + y_j(z_j \bmod 2) \quad y_{j+۱} = 2\ y_j \quad z_{j+۱} = \lfloor z_j/2 \rfloor$

پس داریم :

$x_{j+۱} + y_{j+۱} \times z_{j+۱} = x_j + y_j(z_j \bmod 2) + 2y_j(\lfloor z_j/2 \rfloor) =$
$x_j + y_j((z_j \bmod 2) + 2\lfloor z_j/2^{\rfloor))} \overset{(*)}{=} x_j + y_j \times z_j = y_\circ \times z_\circ$

ودر انتها که حلقه پایان می یابد مقدار z برابر صفر خواهد شد که اگر این اتفاق در k امین تکرار حلقه رخ دهد داریم :

$$z_k = 0, x_k + y_k \times z_k = y_0 z_0 \Rightarrow x_k + y_k \times z_k = x_k + y_k \times 0 = x_k = y_0 \times z_0$$

پس مشاهده شد که وقتی الگوریتم به پایان می رسد مقدار x ای که برگردانده می شودبرابر ضرب مقادیر اولیه‌ی y و z خواهد بود و این همان نتیجه مطلوب و مورد انتظار ماست .

۷)با استفاده از loop invariant نشان دهیدالگوریتم زیر خارج قسمت و باقی مانده تقسیم دو عدد طبیعی را بر می گرداند:

function *divide(y,z)*

comment *return q, $r \in \mathbf{N}$ such that $y = qz+r$ and $r < z$, where y , $z \in \mathbf{N}$*

r :=y; q :=0; w :=z;

while $w \leq y$ **do** *w :=2 w;*

while $w > z$ **do**

  q :=2 q; w := ⌊ w /2 ⌋ ;

  if $w \leq r$ **then**

    r :=r - w; q :=q+1;

return *(q , r)*

....

loop invariant : $q_j w_j + r_j = y_0 , r_j < w_j$

مقدار اولیه‌یy را y_0 در نظر می گیریم ،قبل از ورود به حلقه $q = 0$ و $r = y_0$ است پس داریم :

$$r = y_0 \quad , q = 0 \Rightarrow q_j w_j + r_j = 0 + y_0 = y_0$$

حال فرض می کنیم loop invariant برای مرحله‌ی j ام برقرار باشد،یعنی :

$q_j w_j + r_j = y_\circ , r_j < w_j$

حال برای تعیین وضعیت متغیر ها در تکرار بعدی لازم است قدری روی دو حلقه الگوریتم تامل نماییم،قبل از شروع حلقه اول w را برابر z در نظر گرفتیم ،در حلقه اول تا موقعی که شرط $w \leq y$ برقرار است w را دو برابر می نماید ،پس در انتهای این حلقه wای به دست می آید که‌از y بزرگتر است وهمچنین عددی زوج می باشد،حال وارد حلقه دوم می شویم ،در این حلقه تا موقعی که $w > z$ است w نصف شده و کف آن محاسبه می گردد،واضح است که همواره $\lfloor w/۲ \rfloor$ عددی زوج است و این به این خاطر است که در ابتدای امر ما$z = w$ در نظر گرفتیم ،اگر z زوج باشد w نیز زوج بوده و با هر بار دو برابر شدن نیز زوج باقی می ماند وبا هر بار نصف شدن نیز می توانیم علامت کف را نادیده بگیریم ،حال اگر مقدارz فرد باشد بعد از انتهای حلقه اول w ی زوجی در اختیار داریم ، اما در حلقه دوم نیز این w به دست آمده هر بار نصف شده وکف آن محاسبه می گردد و اگر بخواهد مقدار فردی داشته باشد برابر با خود z خواهد بود و این در حا لی است که شرط برقراری حلقه $w > z$ است ،پس در هر بار تکرار در این حلقه مقدار $\lfloor w/۲ \rfloor$ برابر با$w/۲$ می باشد.
حال در تکرار $j + ۱$ ام داریم : $q_{j+۱} = ۲q_j \quad , w_{j+۱} = \lfloor w_j/۲ \rfloor$
حال باید دو حالت را بررسی نماییم :

(a) اگر شرط خط پنجم برقرار نباشد داریم :

$$w_{j+۱} > r_j \Rightarrow \begin{cases} ۱: \quad r_{j+۱} = r_j \quad , q_{j+۱} w_{j+۱} + r_{j+۱} = \\ \quad ۲q_j \lfloor w_j/۲ \rfloor + r_j = ۲q_j (w_j/۲) + r_j = q_j w_j + r_j = y_\circ \\ ۲: \quad w_{j+۱} > r_j \Rightarrow r_j < w_{j+۱} \quad , r_{j+۱} = r_j \\ \quad \Rightarrow r_{j+۱} < w_{j+۱} \end{cases}$$

(b) اگر شرط خط پنجم برقرار باشد داریم :

$$w_{j+1} < r_j \Rightarrow \begin{cases} 1: & r_{j+1} = r_j - w_{j+1} = r_j - \lfloor w_j/2 \rfloor, q_{j+1} = \\ & q_{j+1} + 1 = 2q_j + 1 \Rightarrow q_{j+1}w_{j+1} + r_{j+1} = \\ & (2q_j + 1)\lfloor w_j/2 \rfloor + r_j - \lfloor w_j/2 \rfloor = \\ & q_j w_j + \lfloor w_j/2 \rfloor + r_j - \lfloor w_j/2 \rfloor = y_0 \\ & \\ 2: & w_{j+1} < r_j, r_{j+1} = r_j - w_{j+1} \quad (1) \end{cases}$$

حال باداشتن (۱) باید اثبات نماییم $r_{j+1} < w_{j+1}$ ،برای این کار از برهان خلف استفاده نموده و فرض می نماییم $r_{j+1} \geq w_{j+1}$ پس داریم :

$$r_j - w_{j+1} \geq w_{j+1} \Rightarrow r_j \geq 2w_{j+1} \Rightarrow r_j \geq 2\lfloor w_j/2 \rfloor \Rightarrow r_j \geq w_j$$

که این نتیجه با فرض ما که همان برقرار بودن loop invariant برای مرحله‌ی j ام بود تناقض داشته و داریم :$r_{j+1} < w_{j+1}$ ، به این ترتیب قسمت دوم loop invariant نیز برای مرحله‌ی j+1 ام نیز برقرار است .

در انتها نیز وقتی از حلقه‌ی دوم خارج شده والگوریتم بعد از k تکرار به پایان می رسد داریم $w_k = z_0$.

پس طبق loop invariant داریم : $r_k < z_0 \quad , q_k z_0 + r_k = y_0$ پس الگوریتم خارج قسمت وباقی مانده‌ی تقسیم دو عدد صحیح را برای ما برگرداند.

۸)با استفاده از loop invariant نشان دهیدالگوریتم زیر عددی حقیقی را به توان عددی طبیعی می رساند:

function *power(y,z)*

comment *return* y^z, *where* $y \in \mathbf{R}, z \in \mathbf{N}$

1. $x := 1;$
2. **while** $z > 0$ **do**
3. **if** z *is odd* **then** $x := x.y;$
4. $z := \lfloor z/2 \rfloor;$
5. $y := y^2;$
6. **return** (x)

loop invariant : $x_j {y_j}^{z_j} = {y_0}^{z_0}$

مقادیر اولیه y و z را y_0 و z_0 در نظرمی گیریم ، در ابتدا $x_0 = 1$ بوده وطبق loop invariant داریم :

$x_j {y_j}^{z_j} = {y_0}^{z_0}$

حال بر اساس loop invariant برای مرحله j ام و طبق روال حلقه برای مرحله j+1 ام داریم :

$x_j {y_j}^{z_j} = {y_0}^{z_0},$

$$\begin{cases} \text{اگر z فرد باشد} \Rightarrow x_{j+1} = x_j y_j, \quad z_{j+1} = \lfloor z_j/2 \rfloor, \quad y_{j+1} = {y_j}^2 \\ \text{اگر z زوج باشد} \Rightarrow x_{j+1} = x_j, \quad z_{j+1} = \lfloor z_j/2 \rfloor, \quad y_{j+1} = {y_j}^2 \end{cases}$$

پس داریم :

$$\begin{cases} \text{اگر z فرد باشد} : \quad x_{j+1} {y_{j+1}}^{z_{j+1}} = x_j y_j \times (y_j)^{2 \times \lfloor z_j/2 \rfloor} = \\ \quad x_j y_j \times (y_j)^{2 \times (z_j - 1)/2} = x_j {y_j}^{z_j} = {y_0}^{z_0} \\ \\ \text{اگر z زوج باشد} : \quad x_{j+1} {y_{j+1}}^{z_{j+1}} = x_j \times (y_j)^{2 \times \lfloor z_j/2 \rfloor} = \\ \quad x_j \times (y_j)^{2 \times (z_j)/2} = x_j {y_j}^{z_j} = {y_0}^{z_0} \end{cases}$$

درانتهای کار نیز حلقه موقعی به پایان می رسد که در یک مرحله مانند مرحله k ام داشته باشیمz=0 پس داریم :

$x_k \times {y_k}^{z_k} = {y_0}^{z_0}, z_k = 0 \Rightarrow x_k = {y_0}^{z_0}$

بدین ترتیب x ای که درانتهای الگوریتم به ما تحویل داده می شود y اولیه به توان z اولیه است ،پس این الگوریتم نیز نتیجه‌ی مطلوب را به ما داد.

۹)با استفاده از loop invariant نشان دهیدالگوریتم زیر مقادیر موجود در آرایه $A[1 \cdots n]$ رابا یکدیگر جمع می نماید:

function *sum(A)*

comment $return \sum_{i=1}^{n} A[i]$

1. *s :=0;*
2. **for** *i := 1* **to** *n* **do**
3. *s :=s+A[i]*
4. **return** *(s)*

....

$$\textbf{loop invariant}: s_j = \sum_{i=1}^{j} A[i]$$

قبل از ورود به حلقه مجموع عناصر آرایه برابر صفر است و ما نیز طبق loop invariant داریم :

$$j = 0 \Rightarrow s = s_0 = \sum_{i=1}^{0} A[i] = 0$$

حال فرض می کنیم loop invarint برای مرحله‌ی j ام برقرار باشد ،برای مرحله‌ی j+1 ام داریم :

$$s_{j+1} = s_j + A[j+1] = (\sum_{i=1}^{j} A[i]) + A[j+1] = \sum_{i=1}^{j+1} A[i]$$

در هنگام خروج از حلقه نیز داریم :

$$j = n \Rightarrow s = s_n = \sum_{i=1}^{n} A[i]$$

که همان نتیجه‌ی مورد انتظار ما از الگوریتم است .

۰۱) با استفاده از loop invariant نشان دهید الگوریتم زیر مقدار چند جمله‌ای $a_n x^n + a_{n-1} x^{n-1} + \cdots a_1 x + a_0$ را با استفاده از روش Horner محاسبه می کند که در آن ضرایب در آرایه A[0..n] ذخیره شده‌اند:

$A[i] = a_i \quad for \quad all \quad 0 \le i \le n$

function *Horner(A , n)*

comment *return* $\sum_{i=0}^{n} A[i].x^i$

1. *v := 0*
2. **for** *i := n* **downto** *0* **do**
3. *v := A[i]+v.x*
4. **return** *(v)*

....

$$\textbf{loop invariant} : v_j = \sum_{i=j}^{n} A[i] x^{i-j}$$

باید توجه نمود که این الگوریتم از حلقه‌ی for کاهشی استفاده می نماید ،لذا ما loop invariant رابه صورت بالا در نظر گرفته و قبل از ورود به حلقه داریم j=n+1 پس طبق loop invariant داریم :

$$j = n+1 \Rightarrow v = \sum_{i=n+1}^{n} A[i] x^{i-(n+1)} = 0$$

حال اگرloop invariant برای مرحله‌ی j ام برقرار باشد ،در مرحله‌ی بعدی داریم :

$$v_j = A[j] + v_{j+1}.x \Rightarrow v_{j+1} = \frac{v_j - A[j]}{x} = \frac{\sum_{i=j}^{n} A[i]x^{i-j} - A[j]}{x} = \frac{\sum_{i=j+1}^{n} A[i]x^{i-j}}{x} =$$

$$\sum_{i=j+1}^{n} A[i] x^{i-j-1} = \sum_{i=j+1}^{n} A[i] x^{i-(j+1)}$$

در هنگام خروج از حلقه j = 0 شده و داریم : $j = 0 \Rightarrow v = \sum_{i=0}^{n} A[i] x^i$ که همان نتیجه‌ی مورد انتظارماست .

۲.٦ آنالیز استهلاکی (Amortized Analysis)

یکی از روشهای آنالیز یک مجموعه از عملیات ، آنالیز استهلاکی است . در آنالیز استهلاکی زمان اجرای یک مجموعه از عملیات ساختمان داده ای روی تمام عملیات اجرا شده سرشکن می شود . از این روش برای نشان دادن این نکته استفاده می کنیم که هزینه متوسط هر عمل کوچک است حتی اگر یک عمل درون یک مجموعه هزینه زیادی داشته باشد . اگر چه ما در مورد میانگین و متوسط صحبت می کنیم اما آنالیز استهلاکی با آنالیز در حالت میانگین تفاوت دارد و روش های آماری را شامل نمی شود . یک آنالیز استهلاکی زمان اجرای متوسط هر عمل در بدترین حالت را ضمانت می کند .

انوع آنالیز استهلاکی

سه روش از پر کاربردترین روش های آنالیز استهلاکی عبارتند از:

- آنالیز تجمعی (Aggregate Analysis)
- روش حسابی (Accenting Method)
- روش پتانسیل (Potential Method)

۱.۲.٦ آنالیز تجمعی (Aggregate Analysis)

در آنالیز تجمعی نشان می دهیم به ازای تمام n ها،یک مجموعه از n عملیات در بدترین حالت ،مجموعاً $T(n)$ را می گیرد . بنابر این در بدترین حالت ، هزینه متوسط یا هزینه استهلاکی هر عملیات $\frac{T(n)}{n}$ است . توجه کنید که این هزینه متوسط برای هر عملیات صادق است ، حتی وقتی که چندین نوع از عملیات در مجموعه موجود هستند . اما در دو روش بعدی ممکن است هزینه های متوسط متفاوتی را به عملیات های مختلف نسبت دهیم . این روش اگر چه ساده است اما دقت دو روش بعدی را ندارد . در عمل روش های حسابی و پتانسیل یک هزینه استهلاکی مخصوص به هر عمل اختصاص می دهند .

مثال ۱ : در اولین مثال از آنالیز تجمعی ، ساختمان داده پشته [1] را آنالیز می کنیم .
دو عمل اصلی پشته که از $O(1)$ هستند عبارتست از: $\text{push}(S,x)$: عنصر x را وارد پشته S میکند.
$\text{pop}(S)$: بالاترین عنصر پشته S را برداشته ، آن را برمی گرداند.

از آنجایی که هر کدام از این عملیات ها در زمان $O(1)$ اجرا می شوند فرض می کنیم هزینه هر کدام ۱ باشد . بنابراین هزینه نهایی یک مجموعه از n عملیات push و pop ، n است .
حال ما یک عمل $\text{multipop}(S,k)$ را به پشته اضافه می کنیم که k عنصر را از بالای پشته S برمیدارد . و یااگر پشته S ، کمتر از k عنصر داشته باشد ، آن را خالی می کند .
در شبه کد زیر تابع Stack-Empty مقدار TRUE می گیرد اگر هیچ عنصری در پشته موجود نباشد ، در غیر این صورت مقدار FALSE را برمی گرداند .

```
   Multipop(S, k)
1. while   not   Stack-Empty(S)   and   k ≠ 0
2.         do pop(S)
3.         k ← k − 1
```

در اینجا یک مجموعه از n عملیات push و pop و multipop را روی یک پشته که در ابتدا خالیست آنالیزمی کنیم ، دراین مجموعه در بدترین حالت اگر سایز پشته حداکثر n باشد هزینه عمل multipop از $O(n)$ است . در بدترین حالت زمان اجرای هر عملیات در پشته از $O(n)$ است بنابراین هزینه کل $O(n^2)$ می شود .
اگر چه این تحلیل درست است اما نتیجه به دست آمده با توجه به هزینه در بدترین حالت محکم نیست . بااستفاده از روش آنالیز تجمعی میتوانیم با توجه به مجموعه شامل n عملیات یک کران بالایی بهتر بدست بیاوریم . در حقیقت ، اگر چه عمل multipop به تنهایی هزینه زیادی می برد اما هر مجموعه از n عملیات push و pop و multipop روی یک پشته در ابتدای خالی حداکثر ، هزینه $O(n)$ را دارد .
چون هر عنصر به ازای هر بار ورود به پشته فقط می تواند یکبار از پشته برداشته شود . بنابراین ، تعداد فراخوانی تابع pop (با در نظر گرفتن multipop) ، روی یک پشته غیر تهی به تعداد فراخوانی تابع push است که حداکثر برابر n است . به

[1] stack

ازای هر n ، هر مجموعه از n push ، pop و multipop زمان کل $O(n)$ را می گیرد . هزینه میانگین یا سرشکن هر عمل $\frac{O(n)}{n} = O(1)$ است . در آنالیز تجمعی ، هزینه استهلاکی برای هر عمل در واقع همان هزینه متوسط ان عمل است . بنابراین در این مثال هزینه استهلاکی هر ۳ عمل پشته $O(1)$ است .
دوباره تأکید می شود که اگر چه ما هزینه متوسط را محاسبه کرده ایم اما از روش های آماری استفاده نکرده ایم .

مثال ۲ : مثال دیگر از این روش مسأله شمارنده دودویی k-bit افزایشی[۲] است . یک آرایه $A[0..k-1]$ به طول length[A]=k به عنوان شمارنده است . عدد دودویی x که در شمارنده ذخیره می شود دارای کمترین ارزش در $A[0]$ و بیشترین ارزش در $A[k-1]$ است. عمل افزایش توسط تابع زیر صورت می گیرد :

```
INCREMENT(A)
i ← 0
while   i < length[A]   and   A[i] = 1
        do   A[i] ← 0
             i ← i + 1
if   i < length[A]
        then   A[i] ← 1
```

اجرای INCREMENT به تنهایی در بدترین حالت (زمانی که همه بیت ها ۱ باشند)زمان $\Theta(k)$ را می گیرد بنابرین یک رشته از n عمل INCREMENT در بدترین حالت روی یک شمارنده که در ابتدا صفر است ، زمان $O(nk)$ را می گیرد .
با توجه به این که در هر فراخوانی همه بیت ها تغییر نمی کنند می توانیم کران دقیقتری ارائه دهیم به طوری که در بدترین حالت اجرای یک رشته از n عملیات INCREMENT از $O(n)$ شود . $A[0]$ در هر بار فراخوانی تابع تغییر می کند ، $A[1]$ در n بار اجرای تابع هر $\lfloor \frac{n}{2} \rfloor$ بار تغییر می کند و $A[2]$ هر $\lfloor \frac{n}{4} \rfloor$ بار تغییر می کند .
به طور کلی ، برای $i = 0, 1, 2, \cdots, \lfloor \log_2^n \rfloor$ ، بیت $A[i]$ ، هر $\lfloor \frac{n}{2^i} \rfloor$ بار تغییر می کند .
برای $i > \lfloor \log_2^n \rfloor$ ، بیت $A[i]$ هرگز تغییر نمی کند .

---
۲ Incrementing a Binary Counter

تعداد کل تغییر ها در مجموعه برابر است با:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

بنابراین هزینه متوسط هر عملیات $\frac{O(2n)}{n} = O(1)$ می شود .

۲.۲.٦ روش حسابی (Accounting Method)

در روش حسابی از آنالیز استهلاکی ، به عملیات های مختلف هزینه های متفاوتی اختصاص داده می شود که این شارژ گاهی ممکن است از هزینه واقعی عمل کمتر یا بیشتر باشد. هزینه ای که به عنوان شارژ روی یک عمل ذخیره می شود را هزینه استهلاک گوییم . هنگامی که هزینه استهلاکی بیشتر از هزینه واقعی عمل باشد ، اختلاف به عنوان اعتبار آن عمل به خصوص در ساختمان داده در نظر گرفته می شود . اعتبار می تواند بعداً در پرداخت های بعدی برای عملیات هایی که هزینه استهلاکشان کمتر از هزینه واقعی است استفاده شود . این روش با روش تجمعی بسیار متفاوت است .

ابتدا باید هزینه استهلاکی هر عملیات بدقت مشخص شود . اگر ما می خواهیم از روش استهلاکی برای اثبات اینکه هزینه متوسط در بدترین حالت هر عملیات کوچک است ، استفاده کنیم باید هزینه استهلاکی کلی یک کران بالایی برای هزینه واقعی کل باشد .

اگر هزینه واقعی عمل i ام را با C_i و هزینه استهلاکی عمل i ام را با $\hat{C}_i$ مشخص کنیم ، داریم :

$$\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i$$

اعتبار نهایی ذخیره شده در ساختمان داده اختلاف بین هزینه واقعی و استهلاکی است .

$$\sum_{i=1}^{n} \hat{C}_i - \sum_{i=1}^{n} C_i$$

این مقدار باید همیشه غیر منفی باشد . اگر اعتبار نهایی منفی شد آنگاه هزینه استهلاکی نهایی زیر هزینه واقعی کل قرار می گیرد و در نتیجه هزینه استهلاکی کل یک کران بالا برای هزینه واقعی نخواهد بود . بنابراین باید مراقب باشیم اعتبار نهایی

در ساختمان داده منفی نشود .
مثال ۱: مثال پشته را در نظر بگیرید یادآوری می کنیم که هزینه واقعی عملیات به صورت زیر است :

Push	: 1
Pop	: 2
Multipop	: 3

فرض کنید مقادیر استهلاکی برای هر عمل بصورت زیر باشد .

Push	: 2
Pop	: 0
Multipop	: 0

توجه کنید اگر چه هزینه واقعی multipop متغیر است اما هزینه استهلاکی تابع صفر است وقتی یک عنصر را وارد پشته می کنیم ۱ واحد (به اندازه هزینه واقعی) از هزینه استهلاکی برای انجام این کار می پردازیم و یک واحد باقی مانده در شئ ذخیره می شود .
این اعتبار ذخیره شده روی شئ در واقع هزینه برداشتن شئ از پشته است . وقتی تابع pop فراخوانی می شود هزینه برداشتن عنصر از این اعتبار ذخیره شده تامین می شود . بنابر این ما همیشه اعتبار کافی برای انجام عمل pop یا multipop روی شئ را داریم . تا زمانی که عنصری در پشته است اعتبار هیچ گاه منفی نمی شود .
برای هر مجموعه از n عمل push و pop و multipop هزینه استهلاک $O(n)$ است که کران بالا برای هزینه واقعی نیز هست .
مثال ۲: مثال شمارنده دودویی افزایشی را بررسی می کنیم . قبلاً دیدیم که زمان اجرای این تابع متناسب با تعداد بیت هایی است که تغییر می کنند . هزینه استهلاکی تغییر یک از صفر به یک را ۲ در نظر می گیریم . هنگامی که یک بیت ۱ می شود یک واحد پرداخت می شود و واحد دیگر اعتبار برای زمانی که بیت را به صفر تغییر دهیم ذخیره می شود . در هر زمانی از اجرا ، هر ۱ در شمارنده یک واحد اعتبار ذخیره دارد بنابراین برای صفر کردن آن ، اعتبار لازم موجود است و نیازی به اعتبار جدید نیست و چون تعداد یک ها در شمارنده هیچ گاه منفی نیست پس اعتبار نهایی نیز هیچ گاه منفی نمی شود . بنابراین هزینه سرشکن $O(n)$ است که کران بالا برای هزینه واقعی است .

۳.۲.٦ روش پتانسیل (Potential Method)

سومین روش آنالیز سرشکنی ، روش پتانسیل است که بر خلاف روش های دیگر که روی یک شئ مشخص در یک ساختمان داده کار می کردند ، روش پتانسیل روی ساختمان داده ها کامل کار می کند . فرق آن با روش حسابی اینست که مقدار اضافی ذخیره شده یا همان سود در اینجا به عنوان انرژی پتانسیل تعریف می شود .

اگر تعداد اجراهای ساختمان داده از ۱ تا n باشد و $D_\circ$ ساختمان داده اولیه باشد برای هر $i = ۱, ۲, \cdots, n$ تعریف می کنیم :

$C_i =$ زمان واقعی انجام عملیات i ام

$D_i =$ ساختمان داده بعد از انجام عملیات i ام

$\Phi =$(Potential Function) تابع پتانسیل که هر ساختمان داده D_i را به اعداد حقیقی می برد

$\Phi : D_i \rightarrow RealNumber$

$\hat{C}'_i = C_i + \Phi(D_i) - \Phi(D_{i-۱})$ هزینه سرشکنی در مرحله i ام

هزینه استهلاکی کل برای n عملیات:

$$\sum_{i=۱}^{n} \hat{C}_i = \sum_{i=۱}^{n} [C_i + \Phi(D_i) - \Phi(D_{i-۱})] = \sum_{i=۱}^{n} C_i + \sum_{i=۱}^{n} \Phi(D_i) - \Phi(D_{i-۱})$$

$$= \sum_{i=۱}^{n} C_i + \Phi(D_n) - \Phi(D_\circ)$$

اگرفرض کنیم $\Phi(D_n) > \Phi(D_\circ)$ بنابراین هزینه استهلاکی کل کران بالایی برای هزینه واقعی کل است . چون ما نمی دانیم چند عملیات ممکن است انجام شود . بنابراین اگر فرض کنیم برای هر i :

$$\left.\begin{matrix} \Phi(D_i) \geq \Phi(D_\circ) \\ \Phi(D_\circ) = \circ \end{matrix}\right\} \Rightarrow \Phi(D_i) \geq \circ$$

یعنی تابع پتانسیل یک تابع غیر منفی است . بنابراین تغییرات پتانسیل برابر است با:

$\Phi(D_i) - \Phi(\circ) > \circ \qquad \text{for all i}$

هزینه استهلاکی بدست آمده در اینجا به انتخاب تابع پتانسیل وابسته است . به ازای تابع پتانسیل‌های مختلف هزینه های استهلاکی مختلف خواهیم داشت . بنابراین مهمترین کار ما در روش پتانسیل انتخاب بهترین تابع پتانسیل است .

مثال ۱ : مثال پشته را بررسی می کنیم . تعریف می کنیم :

$\Phi(D_i) =$ تعداد عناصر داخل پشته

$D_{\circ} =$ پشته خالی

$\Phi(D_{\circ}) = \circ$

چون عناصر داخل یک آرایه هیچ گاه منفی نمی شود پس

$\Phi(D_i) \geq \circ = \Phi(D_{\circ})$

فرض می کنیم در مرحله $i-۱$ ، پشته s عنصر داشته باشد

$\Phi(D_{i-۱}) = s$

اگر push درمرحله i ام اجرا شود:

$\Phi(D_i) - \Phi(D_{i-۱}) = (s+۱) - s = ۱$

$\hat{C}_i = C_i + \Phi(D_i) - \phi(D_{i-۱}) = ۱ + ۱ = ۲ \rightarrow O(۱)$

اگر تابع pop در مرحله i ام انجام شود:

$\Phi(D_i) - \Phi(D_{i-۱}) = (s-۱) - s = -۱$

$\hat{C}_i = C_i + \Phi(D_i) - \phi(D_{i-۱}) = ۱ - ۱ = \circ \rightarrow O(۱)$

اگر تابع multipop اجرا شود:

$k' = min(s,k)\ \Phi(D_i) - \Phi(D_{i-۱}) = (s-k') - s = -k'$

$\hat{C}_i = C_i + \Phi(D_i) - \phi(D_{i-۱}) = k' - k' = \circ \rightarrow O(۱)$

بنابراین هزینه استهلاکی کل از $O(n)$ است .

مثال ۲: در شمارنده دودویی داریم :

$\Phi(D_i) = b_i$ تعداد یک ها در مرحله i ام

واضح است که

$\Phi(D_i) \geq \circ$

$t_i =$ تعداد یک هایی که به صفر تبدیل می شوند در مرحله i ام

$C_i = t_i + ۱$

$$\left.\begin{array}{l} if \quad b_i = 0 \Rightarrow b_{i-1} = k = t_i \\ if \quad b_i > 0 \Rightarrow b_i = b_{i-1} - t_i + 1 \end{array}\right\} \Rightarrow b_i \leq b_{i-1} - t_i + 1$$

$$\Phi(D_i) - \Phi(D_{i-1}) \leq b_{i-1} - t_i + 1 - b_{i-1} = -t_i + 1$$

$$C'_i = C_i + \Phi(D_i) - \Phi(D_{i-1}) \leq t_i + 1 - t_i + 1 = 2 \Rightarrow C'_i \leq 2$$

$$\sum_{i=1}^{n} C'_i \leq 2n \rightarrow O(n)$$