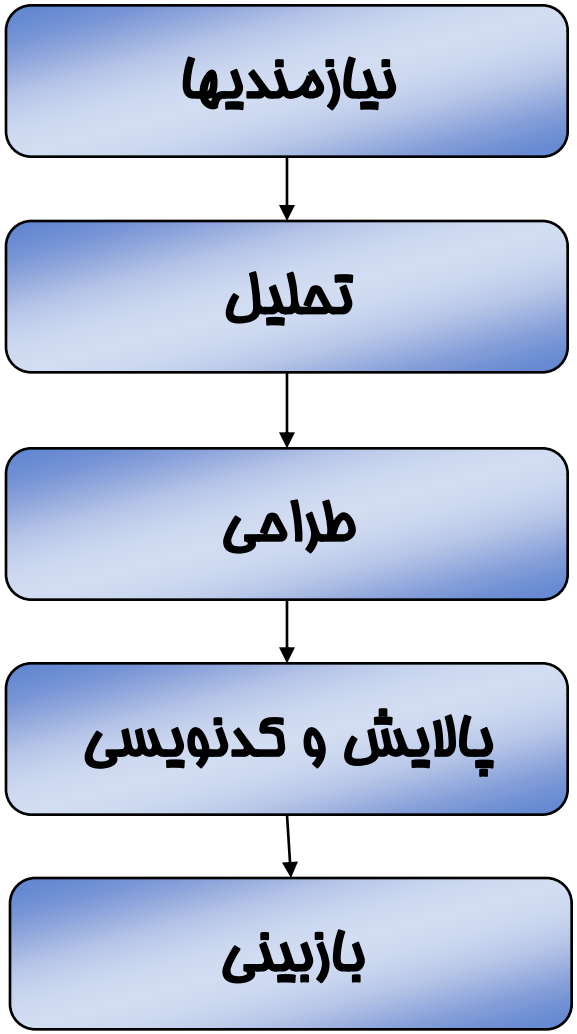
پیچیدگی زمانی

نمودار سیکل زندگی نرم افزار



الگوریتم

الگوریتم مجموعه ای از دستورالعمل ها است که

اگر دنبال شوند، موجب انجام کار خاصی می گردد

شرایط الگوریتم

❖ **ورودی:** یک الگوریتم می تواند هیچ یا چندین کمیت ورودی داشته باشدکه از محیط خارج تامین می شود.

❖ **خروجی:** الگوریتم بایستی حداقل یک کمیت به عنوان خروجی داشته باشد.

❖ **قطعیت:** هر دستورالعمل باید واضح و بدون ابهام باشد.

❖ **محدودیت:** اگر دستوذالعمل های یک الگوریتم را دنبال کنیم ،برای تمام حالات ، الگوریتم باید پس از طی مراحل محدودی خاتمه یابد.

❖ **کارایی:** هر دستورالعمل باید به گونه ای باشد که با استفاده از قلم و کاغذ بتوان آن را با دست نیز اجرا نمود.

الگوریتم مرتب سازی انتخابی

```
for(i=0 ; i<n ; i++)
{
  Examine list [ i ] to list [n-1] and suppose that the
  smallest  integer is at list [min];
  Interchange list [ i ] and list [min];
}
```

الگوریتم های بازگشتی

تابع چیزی است که توسط تابع دیگر فراخوانده می شود .
توابع می توانند خودشان را صدا بزنند (بازگشت مستقیم)
یا
می توانند توابعی که تابع فراخواننده
را صدا میزنند(بازگشتی غیرمستقیم) را احضار نمایند.

الگوریتم جستجوی دودویی

```
int binsearch ( int  list [ ] ،int  searchnum  ،int  left  ،int  right )
{
/* search  list [0] <=  list [1] <=  … <=list [ n-1 ]  for  searchnum  Return
its  position  if  found . Otherwise  return  -1  */
        int  middle ;
        if (left <= right )  {
                middle = ( left + right ) / 2 ;
                switch  ( COMPARE ( list [ middle ]  ،searchnum ))  {
                        case -1 :  return
                           binsearch (list ،searchnum ،middle +1 ،right ) ;
                        case 0 :  return  middle ;
                        case 1 :  return
                             binsearch ( list ،searchnum ،left ،middle -1 ) ;
                }
        }
        return  -1 ;
}
```

آرایه، ساختار و نوع داده

آرایه مجموعه ای از عناصر از یک نوع داده می باشد

ساختار مجموعه ای از عناصر است که لزومی ندارد داده های آن یکسان باشد

نوع داده مجموعه ای از انواع داده مقصد (object) و عملکردهایی است که بر روی این نوع داده ها عمل می کنند

نوع داده ای مجرد

- نوع داده مجرد یا انتزاعی (ADT) یک نوع داده است که شامل مشخصات داده ها و اعمالی که بر روی آنها انجام می شود.

- جهت جداسازی پیاده سازی و نمایش داده ها از یکدیگر.

نوع داده ای مجرد

توابع یک نوع داده به گروه های زیر تقسیم می شوند:

۱- ایجادکننده/ سازنده

۲- تبدیل کنندگان

۳- مشاهده کنندگان / گزارش کنندگان

تحلیل نحوه اجرای برنامه

معیارهای محک زدن یک برنامه:

- آیا برنامه اهداف اصلی کاری را که می خواهیم، انجام می دهد؟
- آیا برنامه درست کار می کند؟
- آیا برنامه مستند سازی شده است تا نحوه استفاده و طرز کار با آن مشخص شود؟
- آیا برنامه برای ایجاد واحدهای منطقی ، به طور موثر از توابع استفاده می کند؟
- آیا کد گذاری خوانا است؟
- آیا برنامه از حافظه اصلی و کمکی به طور موثری استفاده می کند؟
- آیا زمان اجرای برنامه برای هدف شما قابل قبول است؟

این قسمت مربوط به تخمین های حافظه و زمان مورد نیاز بوده و مستقل از ماشین است . این قسمت را تحلیل نحوه اجرای برنامه می نامیم.

پیچیدگی فضای یک برنامه

میزان حافظه یا پیچیدگی فضای یک برنامه مقدارحافظه مورد نیازبرای اجرای کامل یک برنامه است.

میزان یا پیچیدگی زمان یک برنامه مقدار زمان کامپیوتر است که برای اجرای کامل برنامه لازم است.

میزان حافظه

فضای مورد نیاز یک برنامه شامل موارد زیر است :

نیازمندیهای فضای ثابت

این مطلب به فضای مورد نیازی که به تعداد و اندازه ورودی و خروجی بستگی ندارد، اشاره دارد.

نیازمندیهای فضای متغیر

این مورد شامل فضای مورد نیاز متغیرهای ساخت یافته ای است که اندازه آن بستگی به نمونه I از مساله ای که حل می شود، دارد.

زمان T(P) برنامه

زمان T(P) برنامه عبارتست ازمجموع زمان کامپایل و
زمان اجرای برنامه.
زمان کامپایل مشابه اجزای فضای ثابت است زیرابه
خصیصه های نمونه بستگی ندارد.

$T_p(n)$ به صورت زیر تعریف می شود:

$$T_p(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$$

مرحله برنامه

یک مرحله برنامه ، قسمت با معنی برنامه است
(از لحاظ معنایی یا نحوی) که زمان اجرای آن
مستقل از خصیصه های نمونه باشد

پیچیدگی زمانی

زمان اجرای هر دستور * تعداد دستورات اجرایی =زمان اجرای برنامه

❖ فرض می کنیم زمان اجرای همه دستورات یکسان و برابر زمان واحد (۱) می باشد.

تعداد دستورات اجرایی =زمان اجرای برنامه

پیچیدگی زمانی

❖ تعریفات، } و { جز مراحل اجرایی برنامه حساب نمی شود

```
int x;
```

❖ اگر تعریف متغیر همراه با مقداردهی باشد یک مرحله حساب می شود

```
int x=20;
```

❖ بقیه دستورات یک مرحله حساب می شود

```
x=x+y;
printf(“%d”,x);
```

❖ در حلقه ها باید طول حلقه و تعداد دفعات تکرار در نظر گرفته شود

پیچیدگی زمانی

❖ مثال

```
for(i=1;i<=n;i++)
      sum=sum+a[i];
```

❖ مثال

```
for(i=1;i<n;i++)
      sum=sum+a[i];
```

❖ مثال

```
for(i=0;i<=n;i++)
      sum=sum+a[i];
```

❖ مثال

```
for(i=0;i<n;i++)
      sum=sum+a[i];
```

پیچیدگی زمانی

❖ مثال

```
for(i=1;i<=n;i++)
      for(j=1;j<=m;j++)
            printf(“ali”);
```

❖ مثال

```
for(i=1;i<n;i++)
     {
      for(j=1;j<=m;j++)
              x+=j;
      y+=i;
     }
```

پیچیدگی زمانی

الگوریتم	تعداد مراحل اجرایی
void add (int a []MAX-SIZE…)	0
{	0
int i،j;	0
for(i=0;i<rows ; i++)	rows+1
for[j=0 ; j<cols ; j++)	rows.cols
c[i][j]=a[i][j]+b[i][j];	rows.cols
}	0
T	2rows.cols+rows+1

پیچیدگی زمانی

الگوریتم	تعداد مراحل اجرایی

void fun1(int m,int a[])
{
  int i,sum;
  for(i=0;i<m;i++)
  sum+=a[i];
  printf(“%d”,sum);
}	

T	

پیچیدگی زمانی

الگوریتم	تعداد مراحل اجرایی

void fun2(int n, int n)
{
int i,j,p=1;
for(i=1;i<=m;i++)
for(j=1;j<=n;j++)
p=i*j;
printf(“%d”,p);	

T	

پیچیدگی زمانی

❖ در حلقه های تو در تو که طول حلقه به همدیگر وابسته باشد به جای ضرب از سیگما استفاده می کنیم.

```
for(i=1;i<=n;i++)

    for(j=1;j<i;j++)
```

نماد O

انگیزه برای تعیین تعداد مراحل، قابلیت مقایسه پیچیدگی دو برنامه که یک تابع را اجرا می کنند و پیش بینی رشد و افزایش زمان اجرا با تغییر صفات نمونه می باشد.

f(n)=O(g(n)) است اگر و فقط اگر به ازای مقادیر ثابتی از c،n0،f(n) برای تمامی مقادیز n کمتر یا مساوی g(n) باشد (n ≥ n0)

نماد O

- $O(1)$: زمان محاسبه ثابتی را نشان می دهد
- $O(n)$: یک تابع خطی نامیده می شود
- $O(n^2)$: تابع درجه دو نامیده می شود

قضیه :

$f(n) = a_m n^m + \ldots + a_1 n + n_0$ باشد، بنابرا $f(n) = O(n^m)$ خواهد بود

نماد Ω

تعریف:[امگا]: f(n)=Ω(g(n)) می باشد، اگر و فقط اگر به ازای مقادیر ثابت مثبت c و n0 ، f(n)≥cg(n) باشد(برای تمام مقادیر n به شرطی که n≥n0 باشد)

قضیه :

اگر $f(n) = a_m n^m + \ldots + a_1 n + a_0$ **باشد، و** $a_m > 0$

بنابراین $f(n) = \Omega(n^m)$ **خواهد بود**

نماد θ

تعريف تتا[Theta] :$f(n) = \theta(g(n))$ مي باشد، اگر و فقط اگر به ازاي مقادير ثابت c_1 و c_2 و n_0، $c_1g(n) <= f(n) <= c_2g(n)$ وجود داشته باشد (براي تمام مقادير$n>=n_0$)

نشانه گذاری تتا از دو نشانه گذاری ذکر شده " big oh " و امگا دقیق تر می باشد . $f(n) = \Theta(g(n))$ می باشد اگر و فقط اگر g(n) هم به عنوان کرانه بالا و هم به عنوان کرانه پایین در f(n) باشد.

نماد θ

قضیه :

اگر $f(n) = a_m n^m + \ldots + a_1 n + a_0$ **باشد، و** $a_m > 0$

بنابراین $f(n) = \Theta(n^m)$ **خواهد بود**

پیچیدگی زمانی

❖ در جاهایی که بخواهیم پیچیدگی زمانی برنامه را بر اساس نماد O، Ω یا θ بنویسیم:

• اگر تعداد مراحل اجرایی را داشته باشیم برابر با بالاترین توان بدون ضریب خواهد بود.

• در غیر اینصورت فقط مرتبه حلقه هایی که بالاترین تودرتویی و مرتبه را داشته باشند حساب می کنیم.

❖ ترتیب مرتبه پیچیدگی ها

$\theta(\lg n)\ \theta(n)\ \theta(n \lg n)\ \theta(n^2)\ \theta(n^j)\ \theta(n^k)\ \theta(a^n)\ \theta(b^n)\ \theta(n!)$

الگوریتم	پیچیدگی زمانی

Void add (int a []MAX-SIZE…)
{
j;،int I
For(i=0;i<rows ; i++)
For[j=0 ; j<cols ; j++)
C[i][j]=a[i][j]+b[i][j];
}
	0
	0
	0
	Θ(rows)
	Θ(rows.cols)
	Θ(rows.cols)
	0

Total	Θ(rows.cols)

پیچیدگی زمانی

❖ **مثال**

```
for(i=0;i<n;i++)
    {
     a[i]=a[i+1];
     for(j=1;j<n;j++)
           for(k=1;k<n;k++)
                 a[i]=a[i-1];
      printf("ok\n");
     }
```

پیچیدگی زمانی

❖ در حلقه هایی که گام حلقه به صورت ضرب یا تقسیم می باشد، پیچیدگی زمانی بر حسب log نوشته می شود.

❖ **مثال**

```
for(i=1;i<=n;i*=2)
```

❖ **مثال**

```
for(i=n;i>1;i/=3)
```

پیچیدگی زمانی

❖ مثال

```
for(i=1;i<n;i*=2)
    { j=1;
      while(j<=n)
          j*=3;
    }
```

❖ مثال

```
k=0;
for(i=1;i<=n;i++)
  {
   for(j=1;j<=m;j++)
         k=k+1;
   j=1;
   while(j<n)
         {k=k+1;
          j*=2;}
  }
```

پیچیدگی زمانی توابع بازگشتی

❖ برای محاسبه پیچیدگی زمانی توابع بازگشتی، ابتدا یک رابطه بازگشتی برای تابع می نویسیم بعد تابع را حل می کنیم.

```
int fact(int n)
{
 if(n<=1)
      return 1;
else
      return (n*fact(n-1));
}
```

پیچیدگی زمانی توابع بازگشتی

```
float rsum(float list[], int n)
{
 if(n)
       return (rsum(list,n-1)+list[n]);
return list[0];
}
```