1.

```
main()
{
 int i ,Counter ,Tafazol[3000] ,Number[3000] ;
 bool shart=true;

 printf("*** Enter your numbers ***\n");
 scanf("%d",&Number[1]);
 Counter=Number[1];

 for(i=0;i<Counter;i++)
  scanf("%d",&Number[i]);
 for(i=1;i<Counter;i++)
 {
  Tafazol[i]=Number[i]-Number[i+1];
  if(Tafazol[i]<0)
   Tafazol[i]=Tafazol[i]*(-1);
 }
 for(i=1;i<Counter;i++)
 {
  if(Tafazol[i]<Tafazol[i+1] || Tafazol[i]-
Tafazol[i+1]!=1)
   shart=false;
  break;
 }
 if(shart==true)
  printf("Jolly\n");
 else if(shart==false)
  printf("Not Jolly\n");
}<pre class="brush: cpp;" style="direction:ltr;">
/*Variance*/
/*9122360017*/
# include <stdio.h>
main()
{
 int i;
 int x[15];
 double Avg,Var,Siqma=0,Sum=0;
 printf("*** Enter 15 numbers ***\n");
  for(i=0;i<15;i++)
  {
   scanf("%d",&x[i]);
   Sum=Sum+x[i];
  }
 Avg=Sum/15;
  for(i=0;i<15;i++)
   Siqma=Siqma+((x[i]-Avg)*(x[i]-Avg));
 Var=Siqma/14;
```

```
 printf("Variance= %f",Var);
}
</pre
```

چند تا نوع عدد صحیح هم تعریف کردم با اندازه های ۱۲۸ و ۲۵۶ و ۵۱۲ و ۱۰۲۴و ۲۰۴۸ بیت که می توانید بجای اینها از یک تکه از حافظه مثل آرایه ای از بیتها ، با اندازه ی دلخواه خودتان هم استفاده کنید و توابع جمع و تفریق و ضرب و تقسیم و عملیات بیتی همه با گرفتن اشاره گر کار میکنند و سایز عدد را هم می پرسند. پس با هر اندازه ای می توانید محاسبات انجام بدهید ولی باید دو تا عددی که مثلا با هم جمع یا ضرب میشوند از نظر سایز حافظه اندازه ی هم باشند.

با استفاده از روش تقسیم و حل می‌توان روشی بهینه‌تر از ضرب عادی چندجمله‌ای‌ها برای آنها تعریف کرد. در این روش چندجمله‌ای‌ها به دو قسمت تقسیم شده و با استفاده از یک سری روابط، ضرب و جمع شده و نتیجه نهایی را می‌دهند. از همین روش با اندکی تغییر برای ضرب اعداد بسیار بزرگ هم می‌توان استفاده کرد که با اعمال آن، مرتبه ضرب از O(n2)ا به O(n1.58) کاهش پیدا می‌کند.

۲. اگر زیرمسئله به اندازه کافی کوچک باشد از این روش برای حل آن استفاده نمی‌کنیم. تعیین این اندازه - که به آن مقدار آستانه گویند. زمانی که مسئله را به چند زیرمسئله تقسیم می‌کنیم، اگر تقسیم طوری باشد که هر زیرمسئله خودش نزدیک به n ورودی داشته باشد، الگوریتم کارا نخواهد بود. نمونه چنین مسائلی محاسبه بازگشتی جمله nام دنباله فیبوناتچی است.

۴. هدف از تحلیل پیچیدگی الگوریتم‌ها، عمدتاً تخمین زمان اجرا یا حافظه‌ی مصرفی الگوریتم می‌باشد. زمان اجرای الگوریتم معیار مهمی در هر الگوریتم می‌باشد که به ما نشان می‌دهد الگوریتم به ازای یک ورودی مشخص چه مقدار طول می‌کشد تا به پایان برسد. پیاده‌سازی الگوریتم‌ها و اجرای آن در کامپیوتر برای اندازه‌گیری میزان زمان اجرا، کار دشوار و پیچیده‌ای است و پارامترهایی که شاید چندان برای ما مهم نباشند، مانند سرعت پردازنده کامپیوتر، زبان برنامه نویسی و نحوه پیاده‌سازی الگوریتم در آن تاثیر دارند. بنابراین باید از روشی استفاده کنیم که مستقل از موارد یاد شده باشد. این روش لزوماً دقیق نیست اما شهود خوبی نسبت به زمان اجرا به ما می‌دهد و به ما اجازه می‌دهد الگوریتم‌های مختلف را با یکدیگر مقایسه کنیم. روش رایج در تحلیل پیچیدگی الگوریتم‌ها تحلیل مجانبی (asymptotic) می‌باشد. در این روش، زمان اجرا بر حسب اندازه ورودی محاسبه می‌شود و عوامل ثابت نادیده گرفته می‌شوند. منظور از اندازه ورودی میزان فضای لازم برای ذخیره سازی ورودی می‌باشد. مثلاً اگر ورودی الگوریتم، لیستی از اعداد به طول nباشد، اندازه‌ی ورودی nاست، مستقل از این که مقدار هر یک از خانه‌های آرایه چقدر باشد.

اکثر الگوریتم‌ها به ازای ورودی‌های مختلف رفتار متفاوتی از خود نشان می‌دهند و مقدار ورودی، مستقل از اندازه آن، زمان اجرا را تحت تاثیر قرار می‌دهد. مسئله‌ای که به وجود می‌آید این است که زمان اجرای الگوریتم باید به ازای کدام ورودی مورد بررسی قرار بگیرد. گزینه‌های متفاوتی مانند بهترین ورودی، ورودی میانگین یا بدترین ورودی وجود دارند. بهترین ورودی گزینه مناسبی

نیست زیرا رفتار الگوریتم در بهترین ورودی نسبت به حالت عادی تفاوت دارد و در تحلیل الگوریتم دانستن این که زمان اجرا حداقل چه مقداری می‌باشد چندان مفید نیست. محاسبه ورودی میانگین عمدتاً پیچیدگی بیشتری نسبت به حالت بهترین یا بدترین دارد، زیرا باید تمامی حالات ورودی، احتمال وقوع آن و زمان اجرا به ازای هر یک از آن حالات را در نظر گرفت که خود می‌تواند محاسبات سنگینی داشته باشد. در اکثر مواقع الگوریتم به ازای بدترین ورودی (worst case) تحلیل می‌شود. در این حالت می‌دانیم که زمان اجرا حداکثر چه مقداری می‌باشد و تحلیل‌ها را بر اساس آن انجام می‌دهیم.

همانطور که گفته شد، برای تحلیل الگوریتم‌ها از عوامل ثابت چشم‌پوشی می‌کنیم. برای بهتر انجام دادن این کار به نمادگذاری ویژه‌ای نیازمندیم. می‌گوییم تابع $f(n)$ از $O(g(n))$ است، اگر ثابت‌های c و N وجود داشته باشند، به گونه‌ای که برای $n \geq N$ داشته باشیم. $f(n) \leq cg(n)$. به صورت «اُ» یا «اُی بزرگ» تلفظ می‌شود. به عبارت دیگر برای nهای به قدر کافی بزرگ، تابع $g(n)$ از چند برابر تابع $f(n)$ بزرگ‌تر نیست. (در اینجا «چند» همان ضریب ثابت c است.) نماد O تابع را تنها از سمت بالا محدود می‌کند بنابراین تابع $f(n)=5n^2+100n+4$ هم از $O(n^2)$ و هم از $O(n^3)$ می‌باشد. استفاده از نماد O برای تحلیل الگوریتم‌ها روش معقولی می‌باشد، زیرا تفاوت زمان اجرا به ازای ورودی‌هایی با اندازه بزرگ صورت می‌گیرند و نماد O سعی می‌کند تا میزان رشد توابع بر اساس اندازه ورودی (و نه چیزی بیشتر) را نشان بدهد.

۳.

```
_CUTOFF = ۱۵۳۶;


def multiply(x, y):
   if x.bit_length() <= _CUTOFF or y.bit_length() <= _CUTOFF:  # Base case
      return x * y;


   else:
      n = max(x.bit_length(), y.bit_length())
      half = (n + 32) / 64 * 32
```

```
mask = (1 <<half) - 1
xlow = x & mask
ylow = y & mask
xhigh = x>> half
yhigh = y>> half

a = multiply(xhigh, yhigh)
b = multiply(xlow + xhigh, ylow + yhigh)
c = multiply(xlow, ylow)
d = b - a - c
return (((a <<half) + d) <<half) + c
```