سوال اول

```c
#include <stdio.h>
#include <math.h>

int main() {
    double a, b, c, result1, result2, result3;

    // PI number is already defined in math library as M_PI
    a = -5;
    b = M_PI / 6;
    c = M_PI / 5;
    result1 = exp(log2(fabs(a)) + sin(b) + tan(c));

    a = 3.5;
    b = 2;
    c = 9;
    //log() in math.h library is base e and we know ln is log with base e
    result2 = a * (b + a) * b * log(c);

    a = 5;
    b = 6;
    result3 = pow((1 / pow(a, 0.3) + (2 / pow(b, 2.3))), 4.5);

    printf("%.2lf\n", result1);
    printf("%.2lf\n", result2);
    printf("%.2lf", result3);

    return 0;
}
```

سوال دوم

الف)

شکل درست کد به صورت زیر است.

```
#include <stdio.h>

int main(){
   int c = 10;
   int a = 3; int b =2,d; //any number except 0 is correct
   double x2Result = a * 2;
   d = a / b;
   printf("c is now %d", c);
   printf("%lf", x2Result);
   return (0);
}
```

۱. در خط اول Main باید به صورت main باشد و این ارور از نوع linker است.

۲. در خط دوم int c نباید کامنت باشد و ارور از نوع compile است زیرا در خط ششم از متغیر تعریف نشده استفاده شده است.(syntax ارور هم جواب درستیه)

۳. در خط سوم نام متغیر با عدد نمی تواند شروع شود و ارور از نوع syntax است.

۴. در خط پنجم b=0 باعث میشود تقسیم بر صفر جواب محاسبه نشود و ارور از نوع runtime است.

۵. در خط هفتم در انتهای خط سمیکالن قرار داده نشده و این ارور از نوع syntax است.

۶. در خط هشتم return string قرار داده شده در صورتی که خروجی تابع باید از نوع int باشد که سبب ارور منطقی (Logical) می‌شود.

ب) ابتدا مقادیر متغیر ها را بررسی میکنیم.

```
int a = 0,b;
int i, j, k, l;
char x;
double c, d;
```

a -= -5 - 5; ⇒ a -= -10 ⇒ a += 10 ⇒ a = 10 (int)

b = -3 - - (- 3) ; ⇒ b = -3 - 3 ⇒ b = -6 (int)

c = a + 7; ⇒ c = 10 + 7 ⇒ c = 17 (cast to double)

d = b + 4.0; ⇒ d = -6 + 4.0 ⇒ d = -2.0

x = a + b + 65; ⇒ x = 10 + (-6) + 65 ⇒ x = 69 (cast to char) ⇒ x = 'E'

i = j = k = l = 1;

i *= (k += (2 * (l -= (3 / j--)))); ⇒ l -= 3 ⇒ l = -2 ⇒ k += 2 * (-2) ⇒ k = -3 ⇒ i *= -3 ⇒ i = -3

And at last j = 0

و در نهایت خروجی به شرح زیر است:

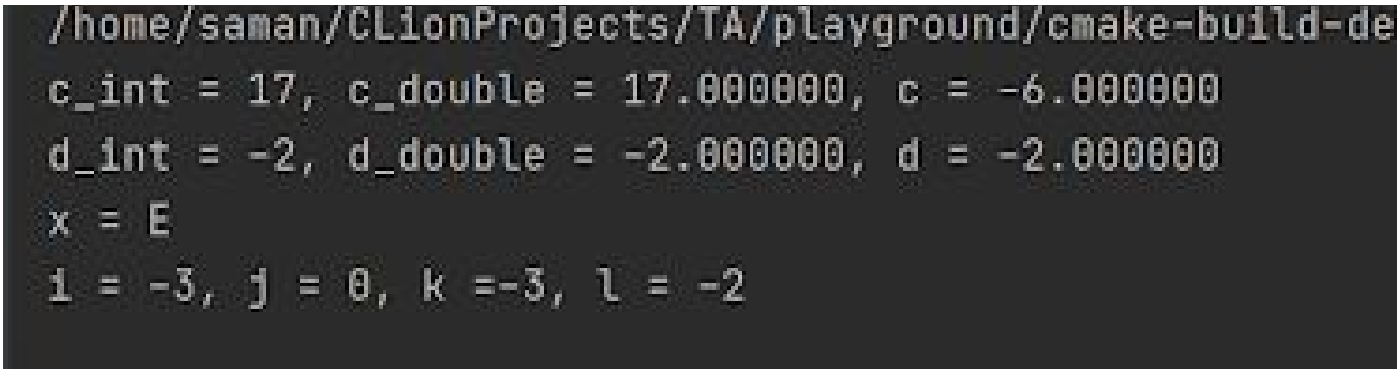
c_int = 17, c_double = 17.000000, c = 0.000000

مورد آخر با این وجود که انتظار می‌رفت مقدار ۱۷ را خروجی دهد، به دلیل cast شدن ناصحیح مقداری نامشخص خروجی می‌دهد. (خروجی با توجه به عملکرد کامپایلر در سیستم‌عامل‌های متفاوت یکسان نمی‌باشد)

d_int = -2,d_double = -2.000000, d = -2.000000

x = E

i = -3, j = 0, k = -3, l = -2



سوال سوم

Pseudocode

Read p

Read d

Set i to 1

While d * i % p > p / 2, do:

    i = i + 1

Set answer to d * i

Write answer

Code

```c
#include <stdio.h>
int main()
{
    int p,d, answer;
    scanf("%d", &p);
    scanf("%d", &d);

    int i = 1;
    while((d * i) % p > p / 2)
        i++;

    answer = d * i;

    printf("%d", answer);
    return 0;
}
```

سوال چهارم

روش اول:

Pseudocode

```
count <- 0
read n
for i from 1 to n:
        if isPrime(i):
                count <- count + 1
print count

isPrime(n){
        if n <= 1:
                return false
        if n = 2:
                return true
        for i from 2 to n-1:
                if n mod i is 0:
                        return false
        return true
}
```

در این شبه کد بخش isPrime با گرفتن یک عدد به ما اعلام می‌کند که آیا آن عدد اول است یا خیر.
اگر عدد کوچکتر یا مساوی ۱ باشد ممکن نیست اول باشد در نتیجه مقدار غلط را خروجی می‌دهیم. درصورتی هم که عدد دو باشد عدد اول است در نتیجه مقدار صحیح را خروجی می‌دهیم. از این پس با استفاده از یک حلقه، عدد n را بر تمامی اعداد بین ۲ تا n-1 تقسیم می‌کنیم و اگر باقی‌مانده برابر صفر شده (mod مانند ٪ باقی‌مانده تقسیم را خروجی می‌دهد) به این معناست که عدد اول نمی‌باشد در نتیجه غلط را خروجی می‌دهیم. اگر تا انتهای حلقه عدد n بر هیچ کدام بخش‌پذیر نبود نتیجه می‌شود که عدد اول است و مقدار صحیح خروجی داده می‌شود.

Code

```c
#include <stdio.h>
#include <stdlib.h>

//it's a function to check if number n is prime or not.
//returns 1 if it's prime and 0 if it's not.
int isPrime(int n) {
    if (n <= 1)
        return 0;
    if (n == 2)
        return 1;

    int i;
    //we know n does not have any divisor after n/2 so we don't need
to
    // check numbers after that. We could also check until radical(n).
    for (i = 2; i <= n / 2; i++) {
        if (n % i == 0)
            return 0;
    }
    //if we reach here and does not return 0 in for loop it means
    //n is prime so  we return 1
    return 1;
}

int main() {
    int count = 0;
    int n, i;
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        if (isPrime(i) == 1) {
            count++;
        }
    }
    printf("%d", count);
    return 0;
}
```

روش دوم:

روش دیگر بدست آوردن اعداد اول استفاده از الگوریتم غربال اراتستن می‌باشد. در این روش با استفاده از یک آرایه مضارب اعداد اول را خط می‌زنیم(در ابتدا فرض می‌شود تمامی اعداد اول هستند) تا جایی که تنها اعداد اول کوچکتر از n خط نخورده باقی بمانند:

Pseudocode

```
Prime[1...n]
Prime[1] = false
Counter = 2
While (counter <= n)
        Prime[counter] = true
        Counter += 1
lastPrime = 2
While (lastPrime <= √n)
        if (Prime[lastPrime] = true)
                Coefficient = 2
                While (Coefficient * lastPrime <= n)
                        isNotPrime = Coefficient * lastPrime
                        Prime[isNotPrime] = false
                        Coefficient += 1
        lastPrime += 1
Count = 0
for i from 1 to n:
        if (Prime[i] = true)
                Print i
                Count += 1
Print count
```

Code

```c
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

int main() {
    int n;
    scanf("%d", &n);

    bool prime[n + 1];
    prime[1] = false;
    for (int i = 2; i <= n; i++)
        prime[i] = true;

    int lastPrime = 2;
    while (lastPrime <= sqrt(n)) {
        if (prime[lastPrime] == true) {
            int coefficient = 2;
            while (coefficient * lastPrime <= n) {
                int isNotPrime = coefficient * lastPrime;
                prime[isNotPrime] = false;
                coefficient++;
            }
        }
        lastPrime++;
    }

    int count = 0;
    for (int i = 1; i <= n; ++i)
        if (prime[i] == true) {
            printf("%d ", i);
            count++;
        }

    printf("\n%d\n", count);
    return 0;
}
```

سوال پنجم

Pseudocode

First solution:

```
i = 1
j = 1
While (i < n)
        While (j < 200)
                If (votes[j] = i)
                        Result[i] += 1
                j += 1
        i += 1
i = 1
While (i < n)
        Print Result[i]
        i += 1
```

Second(and better) solution:

```
j = 1
While (j < 200)
        Result[votes[j]] += 1
        j += 1
i = 1
While (i < n)
        Print Result[i]
        i += 1
```

Code

```c
#include <stdio.h>

void calculateVotes(int votes[], int results[]) {
    for (int i = 0; i < 200; i++)
        results[votes[i]]++;
}

int main() {
    int n;
    scanf("%d", &n);

    int votes[200], results[n];

    for (int i = 0; i < 200; ++i) {
        scanf("%d", &votes[i]);
    }

    calculateVotes(votes, results);
    for (int i = 0; i < n; ++i) {
        printf("Result[%d]: %d\n", i, results[i]);
    }

    return 0;
}
```

سوال ششم

```
Algorithm: function(n,sum):
  if(n!=0){
    m = n mod 10
    sum = sum * 10 + m
    function(n/10, sum)
  }
  return sum
```

Function یک تابع بازگشتی است که عدد n را با ارقام برعکس برمی گرداند.

نحوه ی اجرای تابع به صورت زیر است:

- یکان n را با گرفتن باقیمانده ی آن به 10 به دست می آورد و در m ذخیره می‌کند
- با ضرب sum در 10 و اضافه کردن m، یکان عدد n را به انتهای sum اضافه می کند
- سپس تابع را برای n تغییر یافته که با تقسیم کردن بر ۱۰ رقم یکانش حذف شده است و sum جدید صدا می زند.

و این مراحل را تا زمانی که ارقام n به پایان برسد (n برابر با صفر شود) ادامه می دهد. و در نهایت sum را که اکنون شامل ارقام عدد ورودی ولی به صورت معکوس است برمی گرداند.

بخش امتیازی:

1200 تبدیل به 21 می شود، در حالی که برای وارد کردن رمز به 0021 نیاز داریم. در ادامه با سه روش این مشکل را حل خواهیم کرد:

- با استفاده از تعداد ارقام ورودی و خروجی تابع
- با استفاده از آرایه
- با کمک رشته

روش اول:

یکی از روش‌های حل این مشکل با استفاده از مقایسه کردن تعداد ارقام عدد ورودی و عدد خروجی می‌باشد:

Pseudocode

```
function(n,sum):
   if(n!=0){
      m = n mod 10
      sum = sum * 10 + m
      function(n/10, sum)
   }
   return sum

main():
        Read n
        copyOfN = n
        numOfDigitsN = 0
        While (copyOfN > 0):
                numOfDigitsN += 1
                copyOfN = copyOfN / 10
        Reversed = function(n, 0)
        copyOfReversed = Reversed
        numOfDigitsReversed = 0
        While (copyOfReversed > 0):
                numOfDigitsReversed += 1
                copyOfReversed = copyOfReversed / 10
        While (numOfDigitsN > numOfDigitsReversed)
                Print 0
                numOfDigitsN -= 1
        Print Reversed
```

Code

```c
#include <stdio.h>

int function(int n, int sum) {
    if (n != 0) {
        int m = n % 10;
        sum = sum * 10 + m;
        sum = function(n / 10, sum);
    }
    return sum;
}

int main() {
    int n;
    scanf("%d", &n);

    int copyOfN = n;
    int numOfDigitsN = 0;
    while (copyOfN > 0) {
        numOfDigitsN += 1;
        copyOfN = copyOfN / 10;
    }

    int Reversed = function(n, 0);
    int copyOfReversed = Reversed;
    int numOfDigitsReversed = 0;
    while (copyOfReversed > 0) {
        numOfDigitsReversed += 1;
        copyOfReversed = copyOfReversed / 10;
    }

    while (numOfDigitsN > numOfDigitsReversed) {
        printf("%d", 0);
        numOfDigitsN -= 1;
    }
    printf("%d", Reversed);

    return 0;
}
```

روش دوم:

در این روش ارقام استخراج شده را در یک آرایه ذخیره می‌کنیم و در نهایت آرایه را برعکس چاپ می‌کنیم:

Pseudocode

```
function(n, digits[]):
        Counter = 0
        while(n!=0){
        m = n mod 10
        Digits[Counter] = m
                n = n / 10
                Counter = Counter+1
        }
main():
        Read n
        copyOfN = n
        numOfDigits = 0
        While (copyOfN > 0):
                numOfDigits += 1
                copyOfN = copyOfN / 10
        digits[1...numOfDigits]
        function(n, digits)
        While (numOfDigits > 0)
                Print digits[numOfDigits]
                numOfDigits = numOfDigits - 1
```

Code

```c
#include <stdio.h>

void function(int n, int digits[]) {
    int counter = 0;
    while (n != 0) {
        int m = n % 10;
        digits[counter] = m;
        n /= 10;
        counter++;
    }
}

int main() {
    int n;
    scanf("%d", &n);

    int copyOfN = n;
    int numOfDigits = 0;
    while (copyOfN > 0) {
        numOfDigits += 1;
        copyOfN = copyOfN / 10;
    }

    int digits[numOfDigits];
    function(n, digits);

    for (int i = 0; i < numOfDigits; ++i)
        printf("%d", digits[i]);

    return 0;
}
```

روش سوم:

می‌توان برای ذخیره‌ی نتیجه از string به جای integer استفاده کرد تا leading zero را از دست ندهیم. در این صورت sum یک رشته است که در ابتدا خالی و برابر "" است.

Pseudocode

```
Algorithm: function(n,sum):
        if(n!=0){
                m = n mod 10
                sum = sum + int_to_string(m)
                function(n/10, sum)
        }
        return sum

main():
        Read n
        Reversed = “”
        function(n, reversed)
        Print reversed
```

Code

```c
#include <stdio.h>
#include <string.h>

void function(int n, char *sum) {
    int m;
    char strDigit[2];
    if (n != 0) {
        m = n % 10;
        sprintf(strDigit, "%d", m);
        strcat(sum, strDigit);
        function(n / 10, sum);
    }
}


int main() {
    int n;
    scanf("%d", &n);

    int copyOfN = n;
    int numOfDigits = 0;
    while (copyOfN > 0) {
        numOfDigits += 1;
        copyOfN = copyOfN / 10;
    }

    char reversed[numOfDigits + 1];
    function(n, reversed);

    printf("%s", reversed);

    return 0;
}
```

سوال هفتم

روش اول:

در روش پایین مشاهده می‌شود برای رسیدن به پاسخ از سه حلقه تودرتو استفاده می‌شود. یعنی اگر در هر حلقه n پیمایش انجام دهیم در کل n^3 پیمایش خواهیم داشت. در این صورت زمانی که n عددی بزرگ باشد الگوریتم ما بهینه نیست. برای مثال اگر n = 1000000 باشد باید 10^{18} عملیات انجام داد!

Pseudocode

```
read n
count <- 0
for a from 1 to n/2:
   for b from a to n/2:
      for c from b to n/2:
         if a + b + c == n and a + b > c and a + c > b and b + c > a:
            count <- count + 1
            print a, b, c
print count
```

روش بالا را می‌توان بهینه‌تر نیز انجام داد، برای مثال می‌توانستیم در هر حلقه از 1 تا n را پیمایش کنیم و سپس در حلقه سوم شرط ها را چک کنیم و پاسخ را چاپ کنیم. اما در اینجا حلقه اول از ۱ آغاز شده و حلقه دوم از a و حلقه سوم از b. با این کار اعداد که در حلقه سوم باید چک کنیم شرط a =< b =< c را دارا هستند و طبیعتا تعداد پیمایش ما از n^3 کمتر خواهد شد. علاوه بر این در هرکدام از حلقه ها به جای n، تا n/2 را پیمایش کردیم زیرا می‌دانیم ضلع یک مثلث نمی‌تواند بیشتر از نصف محیط باشد. با این کار هم از تعداد پیمایش هایی که باید انجام دهیم مقدار قابل توجهی کاسته می‌شود.

Code

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
   int n, a, b, c, count = 0;
   scanf("%d", &n);
   //a triangle side can't be longer than n/2.
   //b starts from a and c starts from b, so always c >= b >= a
   //and we should not be worried about repeated results
   for (int a = 1; a <= n / 2; a++) {
       for (int b = a; b <= n / 2; b++) {
           for (int c = b; c <= n / 2; c++) {
               //checking triangle conditions
               if (a + b + c == n && a + b > c && a + c > b && b + c > a) {
                   count++;
                   printf("{%d, %d, %d}\n", a, b, c);
               }
           }

       }
   }

   printf("number of all triangles with perimeter of n: %d", count);

   return 0;
}
```

روش دوم (امتیازی):

برای قسمت امتیازی می‌توان بهینه‌سازی دیگری هم انجام داد که در کد فوق قابل مشاهده است. می‌دانیم اگر دو ضلع یک مثلث و محیط آن را داشته باشیم، ضلع سوم از کم کردن جمع دو ضلع دیگر از محیط به دست می‌آید. بنابراین به جای قرار دارن حلقه سوم می‌توانیم مقدار ضلع سوم را خودمان با یک تفریق به‌دست آوریم و سپس شرط اینکه سه ضلع a, b, c تشکیل مثلث می‌دهند را بررسی کنیم. با این کار یکی از تعداد حلقه های استفاده شده کم شد که همین موضوع بر عملکرد کد ما بسیار تاثیر گذار است.

Pseudocode

```
read n
count <- 0
for a from 1 to n/2:
    for b from a to n/2:
    c <- n - a - b
    if b <= c and a + b > c and a + c > b and b + c > a:
    count <- count + 1
    print a, b, c
Print count
```

Code

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n, a, b, c, count = 0;
    scanf("%d", &n);
    //a triangle side can't be longer than n/2
    for (a = 1; a <= n/2; a++){
        for(b = a; b <= n/2; b++){
             //c is (perimeter of a triangle - sum of other sides)
            c = n - a - b;
            //check triangle conditions.
            if(b <= c && a + b > c && a + c > b && b + c > a){
                count++;
                printf("{%d, %d, %d}\n", a, b, c);
            }
        }
    }
    printf("number of all triangles with perimeter of n: %d",count);

    return 0;
}
```

روش سوم:

روش های دیگری هم برای پیدا کردن این تعداد با یک حلقه for و بدون حلقه وجود دارد که شامل محاسبات ریاضیاتی می‌شود:

With one loop:

```
answer <- 0
for a from 1 to [n/3]:
   answer <- answer + [(n-3*a)/2] - max(0 , [n/2]-2a+1) + 1
print answer
```

No loop:

```
bound <- [([n/2]+1)/2]
answer <- (bound) * (bound + 1) - [n/2] * bound +  [n/3] - bound
k <- [[n/3] / 2]
answer <- answer + (n+1)*k - 3*k*(k+1)
if [n / 3] mod 2 is 1:
   answer <- answer + [(n - 3*(2*k+1)) / 2]
Print answer
```

این دو روش شامل محاسبات نسبتا پیچیده ریاضی هستند و تنها برای اطلاعات بیشتر شما آورده شده‌است. انتظار پیاده سازی آن توسط شما نمی‌رود.

سوال هشتم

اعدادی که دارای خاصیت ذکر شده‌اند همه به صورت $2^n - 1$ می‌باشند. ادامه حل بسیار شبیه به حل سوال ۴ می‌باشد:

Pseudocode

```
count <- 0
read n
i = 2
While i <= n:
    if isPrime(i-1):
        Print i-1
        count <- count + 1
    i = i*2
print count

isPrime(n){
    if n <= 1:
        return false
    for i from 2 to √n:
        if n mod i is 0:
            return false
    return true
}
```

Code

```c
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

bool isPrime(int n) {
   if (n <= 1)
       return false;

   for (int i = 2; i <= sqrt(n); i++)
       if (n % i == 0)
           return false;

   return true;
}

int main() {
   int count = 0;
   int n, i;
   scanf("%d", &n);
   for (i = 2; i <= n; i *= 2) {
       if (isPrime(i - 1)) {
           printf("%d ", i - 1);
           count++;
       }
   }

   printf("\n%d", count);
   return 0;
}
```

سوال نهم

در ابتدا همه‌ی نسخه ها مشکوک به شروع خرابی هستند. در هر مرحله نیمی از نسخه‌ها را از نسخه‌های مشکوک حذف می کنیم. ابتدا نسخه‌ای که در نیمه قرار دارد یعنی 512 امین نسخه را چک می کنیم. اگر سالم بود، نیمه ی اول را از لیست مشکوک‌ها خارج می کنیم و در بازه ی 513 تا 1024 به دنبال اولین نسخه ی خراب می‌گردیم. اگر خراب بود، نیمه‌ی دوم را از لیست مشکوک‌ها خارج می کنیم و از 0 تا 512 به دنبال اولین نسخه‌ی خراب می گردیم.
این کار را ادامه می دهیم و دقیقا پس از ده بار بررسی، لیست نسخه های مشکوک به یک نسخه محدود می‌شود. برای این کار در هر مرحله باید بازه مشکوک را ذخیره کنیم:

Pseudocode

```
find_first_mistake(start, end):
    If start = end:
        return start

    middle = (start + end) / 2
    if middle = okay:
        return find_first_mistake(middle+1, end)
    else:
        return find_first_mistake(start, middle)
```

سوال بالا مثالی از الگوریتم بسیار معروفی به نام binary search می‌باشد. می‌توانید از طریق این لینک اطلاعات بیشتری در این باره کسب کنید.