Indian Institute of Technology, Kharagpur

Department of Computer Science and Engineering

End-Semester Examination, Spring 2015-16: SOLUTIONS

Software Engineering (CS 20006)

Students: 130 **Date**: 25-Apr-16 (AN)

Full marks: 100 **Time**: 3 hours

Instructions:

1. Marks for every question is shown with the question.
2. No clarification to any of the questions will be provided. If you have any doubt, please make suitable assumptions and proceed. State your assumptions clearly. While making assumptions, be careful that you do not contradict any explicitly stated fact in the question.

---

1. You need to design a **Hire Car Management System (HCMS)** for Bhraman Rent-a-Car – a company that passengers use for renting cars with drivers when they need to travel from one city to another. **HCMS** should conform to the following:

Requirements for Hire Car Management System (HCMS)

(a) Bhraman Rent-a-Car has four types of cars maintained in respective *Unit*s:

Unit	Comfort	Passengers	Luggage	Rate
Hatchback	Low	3	Small	Lowest
SUV	Medium	5	Small	Medium
		3	Medium	
Sedan	High	3	Small	Same as *SUV*
Limousine	Luxury	3	Large	Most expensive

Note:

- The *Vehicle*s in a *Unit* are all of the same type (though the model and make may differ). Every *Vehicle* is known by its MAKE, MODEL, and COLOR and is identified by its REGISTRATION NUMBER.

(b) The following types of *Passenger*s use cars from Bhraman Rent-a-Car:

i. *Weekly*. Who use the service every week. They use round-trips that are pre-booked every week on specific days (at given times) unless explicitly canceled. Bhraman Rent-a-Car guarantees these bookings every week.

ii. *Regular*. Who use the service regularly but may not be on a specific periodicity.

iii. *Occasional*. Who use the service sporadically.

Note:

- Every *Passenger* needs to be registered with **HCMS**. *Weekly* and *Regular Passenger*s enjoy 10% discount on their bookings. Every *Passenger* is known by HER / HIS NAME, MOBILE NUMBER, DEFAULT CHOICE OF TYPE OF CAR and DEFAULT PICKUP LOCATION and is identified by a PASSENGER ID.
- *Passenger*s are not pre-designated with the above types. **HCMS** assigns the types to them from their use pattern of hiring services. Naturally, the assignment of types can change every month based on a strategy that the designer of **HCMS** (you!) can decide.

(c) Most *Trip*s as taken up by passengers from Bhraman Rent-a-Car are for plying between Kharagpur and Kolkata. Four types of *Trip*s can be booked:

i. *Round Trip*. Going from Kharagpur to Kolkata and back with a maximum waiting time of 12 hours. Extra retention charge to be paid for additional waiting time between trips.

ii. *Drop Trip*. Going from Kharagpur to Kolkata.

iii. *Pickup Trip*. Going from Kolkata to Kharagpur.

iv. *Custom Trip*. Trips to any destination other than Kolkata must start from Kharagpur and also end back in Kharagpur.

Note:

- Every *Round Trip* is charged on the kilometers traveled at a rate suitable for a type of car.
- Further, every *Drop Trip* (*Pickup Trip*) is charged for double the kilometers traveled unless adjusted against another *Pickup Trip* (*Drop Trip*).
- A vehicle retention charge (that depends on the type of car) is added for a night stay and for an additional day. Retention beyond one day is not allowed.
- Toll fees are charged on actual for every trip.

(d) The *Staff* structure of Bhraman Rent-a-Car is as follows:

i. *Owner*. The *Owner* owns the business and manages *Bhraman Rent-a-Car*. He is responsible to set the per kilometer rate and the rates for retention for different types of cars.

ii. *Manager*. The *Manager* is responsible for the operations. He receives the request for a trip from the passenger and allocates the same suitably to the *Overseer* of the unit. *Overseer*s report to the *Manager*. The *Manager* reports to the *Owner*.

iii. *Overseer*s. Every *Overseer* is responsible for a unit of cars and reports to the *Manager*. He manages the trips allocated for his unit.

iv. *Driver*s. Every *Driver* can drive a car of one designated type (of a unit). He reports to the *Overseer* of the corresponding unit. Every *Driver* is assigned a trip by his *Overseer* and is responsible to ferry the passenger on the trip.

Note:

- Every *Staff* is identified by an EMPLOYEE CODE, and has NAME, ADDRESS, and MOBILE NUMBER. In addition, the *Owner*, the *Manager*, and *Overseer*s have EMAIL; DRIVERS have DRIVING LICENSE NUMBER with valid duration and given car type; and the *Owner* has PAN NO.
- *Driver*s are contractual and paid on trip basis – Rs. 600/= for first 8 hours, Rs. 120/= for every extra hour (maximum up to 4 hours) and Rs. 300/= for overnight stay. Other *Staff*s (with the exception of the *Owner*) draw monthly salary.

(e) The *Booking*s are handled as follows:

- To Request a *Booking* for a *Trip*, a *Passenger* needs to specify TRIP TYPE, TYPE OF CAR, DATE AND TIME OF FORWARD JOURNEY, DESTINATION (for *Custom Trip*), DATE AND TIME OF RETURN JOURNEY (if applicable), ADDRESS FOR PICKUP, NAME OF PASSENGER, and MOBILE NUMBER of passenger. Some of these may be optional if those can be obtained from the profile of the *Passenger*.
- A *Booking* request for a *Trip* is first analyzed by the *Manager*. If the requested type of car is available (for the requested duration), he Reserves the same with the corresponding *Unit* and Confirms the booking to the *Passenger*. If no vehicle of the requested type of car is available, *Manager* sends a Change request to the the *Passenger* specifying the availability of the types of cars (on the requested trip date and time). The *Passenger* would select the type of car from the suggestions and Accept the change request or Cancel the booking. If the change request is accepted, the *Manager* Reserves the type of car and Confirms the booking to the *Passenger*.
- Once a type of car is reserved, the *Overseer* of the *Unit*, Assigns a *Vehicle* and a *Driver* for the *Trip* and generates a *Duty Slip* with details of *Passenger*, *Trip*, *Driver* and *Vehicle*. The *Duty Slip* is sent to the *Driver* and an SMS is sent to the *Passenger* with the mobile number of the *Driver*. The *Vehicle* and the *Driver* assigned for a *Trip* needs to be blocked from other *Booking* requests for the duration of the *Trip* with a buffer of one-hour ahead of the *Trip* and two-hours beyond expected completion.

- The *Driver* Reports to the passenger half-an-hour before the start of the *Trip* at the designated place of pickup. The *Driver* Conducts the trip as planned and on completion (when he has dropped the passenger to the final destination) gets the *Duty Slip* signed by her / him after filling in the place of drop, date / time and kilometers traveled. He then sends the *Duty Slip* (along with toll tokens) to his *Overseer*. This completes a *Trip*. On receipt of signed *Duty Slip*, the *Overseer Close*s the *Booking*. With this the *Driver* and the *Vehicle* are released and become available for further allocation.
- The *Passenger* may Modify or Cancel any booking for a *Trip* till four hours before the start of the trip. On cancellation, the reserved vehicle and assigned driver may need to be released (made available) as the case may be. On a request to modification, the process of cancellation and re-booking would be performed by the respective staffs. A modification request that cannot be honored is canceled.
- A set of *Round Trip*s are pre-booked every week for *Weekly* passengers. These bookings are managed by the respective *Overseers*. A *Weekly* passenger may Cancel such a pre-bookings one week in advance.

(f) The *Payment*s are handled as follows:

- On the closure of a *Booking* the *Manager* proceeds to Raise the invoice for it to the *Passenger* from the *Duty Slip* using the kilometer and retention rates as set already by the *Owner*.
- The *Owner* Collects the payment on an invoice and also makes payments to the *Driver*.

Read the above specifications carefully to generate the SA/SD documents for **HCMS**. Your documents need to cover the following:

(a) Identify the use-cases and design suitable Use-Case Diagrams for **HCMS**. Highlight the relationships among the actors and the use-cases. [**5**]

(b) Design Class Diagrams for *Staff*. Show the attributes and operations with their associated properties. Highlight specialization hierarchies, if any. [**5**]

(c) Complete the Class Diagram of **HCMS** showing all other classes (in addition to Question 1b) by their respective brief Diagrams (with name and key attributes). For the entire collection of classes (that is, including *Staff*) show the associations, aggregations / compositions, generalization / specialization, and abstract / concrete etc. [**10**]

(d) Show the State-chart Diagram for a *Booking* and a *Passenger*. [**5**]

(e) Design Sequence Diagrams for the actions in **HCMS** relating to *Booking*. [**15**]

(f) Prepare a test plan for **HCMS** to perform black-box tests for *Booking*. Clearly mark the scenarios for Unit Testing. (*You need not prepare any test case*). [**10**]

2. The *Secant Method* is used to find the roots of a continuous real-valued function $f(x)$ within an interval $[a, b], a < b$. That is, to find solutions of $f(x) = 0$ such that $a \leq x \leq b$. It is an iterative method that uses a succession of roots of secant lines to better approximate a root of a function f. It is defined by the recurrence:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} = \frac{x_{n-2}f(x_{n-1}) - x_{n-1}f(x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}$$

It requires two initial values, x_0 and x_1, which should ideally be chosen to lie close to the root to start the process of iteration and continues till $f(x_n)$ gets arbitrarily close to 0. That is, $|f(x) - 0| < \epsilon$, where ϵ is a very small positive value like 10^{-6}. To ensure that the process always terminates, the method also uses an upper bound (n) on the number of iterations.

For example, if $f(x) = x^2 - 4$, $x_0 = 1.6$, $x_1 = 2.3$, $\epsilon = 0.0000001$, and $n = 4$, the successive roots by the Secant method computes as: $x_2 = 1.969231$, $x_3 = 1.997838$, $x_4 = 2.000017$, $x_5 = 2.000000$. Naturally, $x = x_5 = 2.000000$ is the final solution as the iterations converge.

You are given the following template-code for `SecantSolver` function to solve for an equation `f(x) = 0` by Secant Method starting from initial estimates `x0` and `x1`. You need to prepare tests for the code.

```
/*01:*/ template <typename T>
/*02:*/ void SecantSolver(
/*03:*/     T f,                // Function f to solve for f(x) = 0
/*04:*/     double x0,          // First initial iteration
/*05:*/     double x1,          // Second initial iteration
/*06:*/     int nIter,          // Maximum number of iterations
/*07:*/     double epsilon) {   // Precision
/*08:*/
/*09:*/     double x_n_minus_1 = x1, x_n_minus_2 = x0;
/*10:*/     int count = 1;
/*11:*/     double x;
/*12:*/
/*13:*/     do {
/*14:*/         x = (x_n_minus_2*f(x_n_minus_1) - x_n_minus_1*f(x_n_minus_2)) /
/*15:*/             (f(x_n_minus_1) - f(x_n_minus_2));
/*16:*/         x_n_minus_2 = x_n_minus_1;
/*17:*/         x_n_minus_1 = x;
/*18:*/         cout << "Iteration No: " << count << " x = " << x << endl;
/*19:*/         ++count;
/*20:*/         if (count == nIter)
/*21:*/             break;
/*22:*/     }
/*23:*/     while (fabs(f(x)) > epsilon);
/*24:*/     cout << "The solution is: " << x << endl;
/*25:*/ }
```

(a) Analyze the problem and the code to prepare for designing test plan.

i. Construct the Control Flow Graph for `SecantSolver` using the line numbers as shown. **[3]**

Answer:
The CFG is shown in Figure 1. Start Node = 9. End Nodes = {24}. Since the assignment statement in lines #14 continues to line #15, we mark the corresponding node as 14(15) or simply as 14.

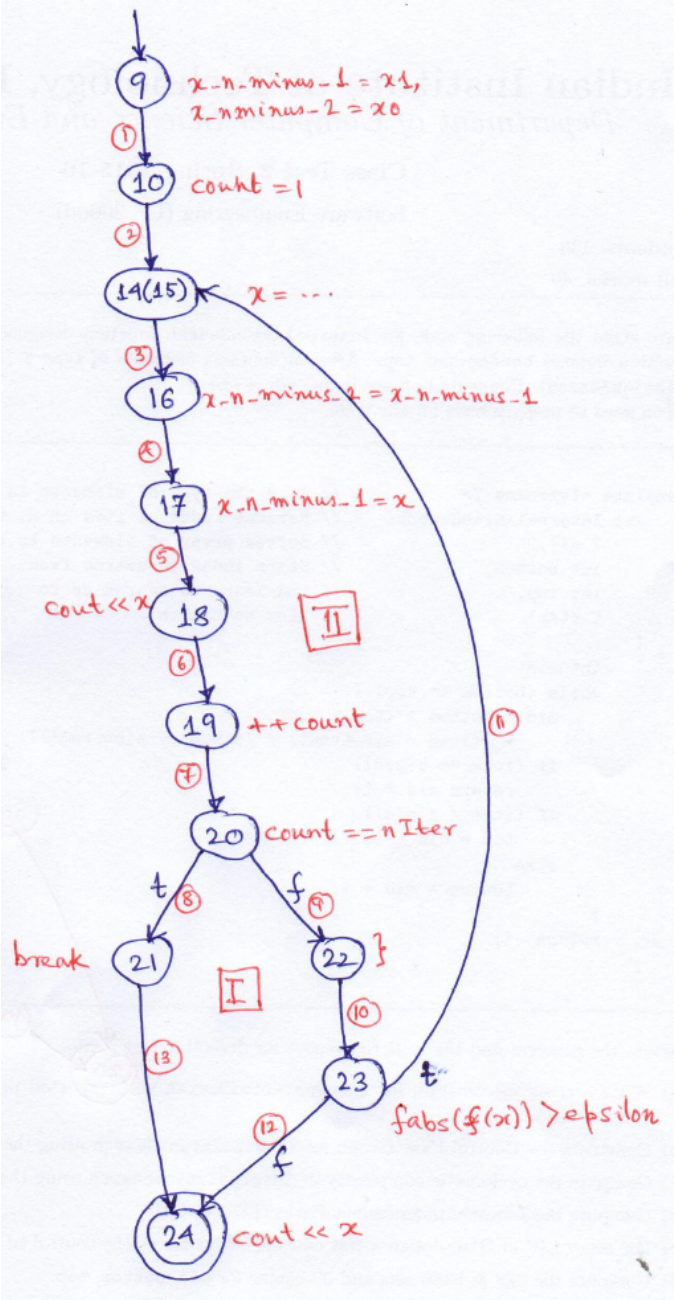


Figure 1: Control Flow Graph

ii. Compute the Cyclomatic Complexity of `SecantSolver` using the CFG in Q2(a)i. [**1**]

Answer: N = 12. { 9, 10, 14(15), 16, 17, 18, 19, 20, 21, 22, 23, 24}, E = 13, V(G) = E – N + 2 = 3 (two bounded regions marked in CFG).

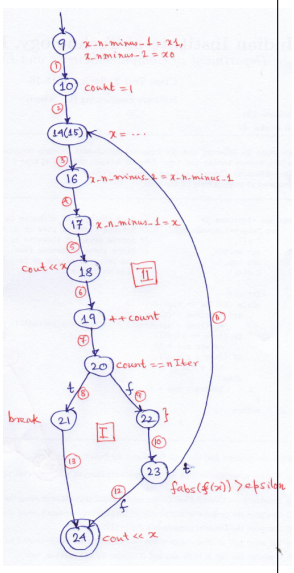
iii. Compute the Linearly Independent Paths (LIP) in Q2(a)i. [**3**]

Answer:

LIP 1 : 9-10-14-16-17-18-19-20-21-24

LIP 2 : 9-10-14-16-17-18-19-20-(22)-23-24

LIP 3 : 9-10-14-16-17-18-19-20-(22)-23-14-16-17-18-19-20-21-24



Note for marks: The solution is not unique. However, all cases must get covered. The numbering for LIPs will depend on the case.

iv. For every LIP in Q2(a)iii, design a test case each for `f(x) = x * x - 4` that forces the control to trace the LIP. [**4**]

Answer:

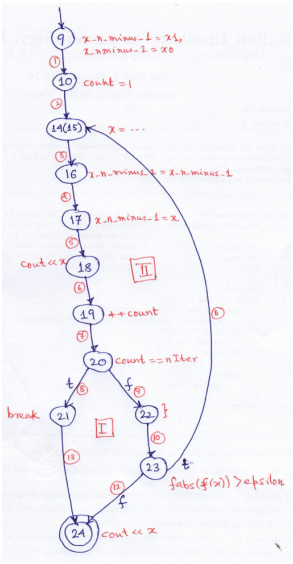
Test	x0	x1	nIter	epsilon	Path	LIP	Iterations	Result / Remarks
T1	1.999	2.001	10	0.100000	9-10-14-16-17-18-19-20-(22)-23-24	LIP 2	1	x = 2
T2	1.999	2.001	1	0.010000	9-10-14-16-17-18-19-20-21-24	LIP 1	1	x = 2
T3	1.8	2.2	10	0.1	9-10-14-16-17-18-19-20-(22)-23-24	LIP 2	1	x = 1.99
T4	1.8	2.2	10	0.01	9-10-14-16-17-18-19-20-(22)-23-14-16-17-18-19-20-(22)-23-24	Mixed	2	x = 1.99952
T5	0.5	4.0	2	0.001000	9-10-14-16-17-18-19-20-(22)-23-14-16-17-18-19-20-21-24	LIP 3	2	x = 1.75

Note for marks: 1 mark each for test case for LIP 1 and LIP 2. 2 marks for test case for LIP 3. Mixed case will be counted for LIP 3. Naturally, all 5 cases as above are not needed. T1 (or T3), T2, and T5 (or T4) will fetch full credit.

v. Compute the `DEF` & `USES` sets for `x_n_minus_1`, `x_n_minus_2`, `x`. **[2]**

Answer:

	`x_n_minus_1`	`x_n_minus_2`	`x`
DEF	{9, 17}	{9, 16}	14(15)
USE	{14(15)-16}	{14(15)}	{17-18-23}, {17-18-23-24}

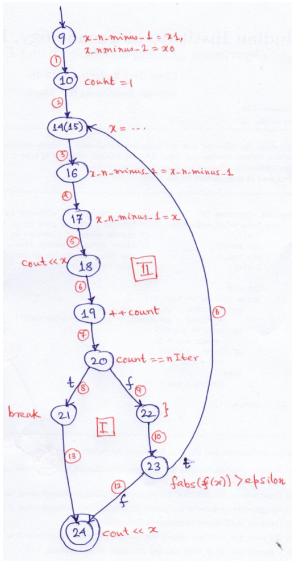


Note for marks: 1 each for DEF and USES sets. Full credit in each case for getting the set correct for at least two variables. Else half mark.

vi. Identify the `DU`-chains in Q2(a)v that are covered by test cases in Q2(a)iv. **[2]**

Answer:

Variable	DU	Test case
`x_n_minus_1`	9 → {14(15)-16}	T1, T2, T3, T4, T5
`x_n_minus_1`	17 → {14(15)-16}	T4, T5
`x_n_minus_2`	9 → {14(15)}	T1, T2, T3, T4, T5
`x_n_minus_2`	16 → {14(15)}	T4, T5
`x`	14(15) → {17-18-23}	T1, T3, T4, T5
`x`	14(15) → {17-18-23-24}	T1, T3, T4



Note for marks: Full credit if the answer is correct for any two variables. Half mark if it is correct for one DU and wrong for the other.

(b) Design a testplan for `SecantSolver` covering the following scenarios.

i. Compile-Time Tests

A. Traits of `T` [**4**]

Answer: The required traits are:
- Functor `double operator()(double)` (Line 23) or
- Function Pointer `double (*)(double)` (Line 23)

Note for marks: Half mark for getting one trait correct. `float` *is acceptable in place of* `double`*.*

B. Candidate data types (`T`) for elements [**2**]

Answer:
- `typedef double(*funcPtr)(double)`
- UDT functor specialized from `unary_function<double, double>` or UDT functor having `double operator()(double)`

Note for marking: Half mark for getting one data type correct. `float` *is acceptable in place of* `double`*.*

C. Black Box Tests

- Equivalence Classes – of functions, of values [**2**]

Answer:
– Types of functions – Polynomial Functions, Transcendental Functions
– Based on number of roots – 0, 1 (Linear), 2 (Quadratic), 3 (Cubic), ..., Infinite

- Corner Cases [**2**]

Answer:
– `root` $<$ `x0`
– `x0` $\geq$ `root` and `root` $\leq$ `x1`
– `root` $>$ `x1`
– `f(x0) == f(x1)`
– Unsolvable function

D. White Box Tests

- Coverage Tests [**3**]

Answer:
– Line Coverage
– Branch Coverage
– Path Coverage

E. Performance Tests [**2**]

Answer:
- Precision Tests – Correctness of solution, Choice of ϵ, ...
- Stability Tests – Stability of computation

Note: The above scenarios are indicative but not exhaustive. You may include more scenarios if you need.

(c) Using the testplan in Q2b design test cases for following scenarios.

i. Compile-Time Tests

A. Traits of `T` [**4**]

Note: These test cases are applications!

Answer:

```
// Type of function pointer for the solver
typedef double(*funcPtr)(double);

double F(double x) { return(x*x - 4); }

// Types of functor for the solver

// UDT functor specialized from unary_function<double, double>
struct my_function : public unary_function<double, double> {
   my_function() {}
   double operator()(double x) { return x*x - 4; }
};

// UDT functor having double operator()(double)
struct my_functor {
   my_functor() {}
   double operator()(double x) { return x*x - 4; }
};

int main()
{
   FILE *fp = fopen("Output.txt", "w");
   fclose(fp);

   double a = 0.5, b = 4.0, e = 0.000000001;
   int count = 1, n = 100;

   SecantSolver<funcPtr>(F, a, b, n, e);

   my_function G;
   SecantSolver<my_function>(G, a, b, n, e);

   my_functor H;
   SecantSolver<my_functor>(H, a, b, n, e);

   return 0;
}
```

The compilation should fail for other variants.

ii. Run-Time Tests

A. Black Box Tests

- Corner Cases [**2**]

Answer:

- root $<$ x0: `f(x) = x*x - 4, x0 = 1.5, x1 = 1.8, n = 10, epsilon = 0.00001`
- x0 $\geq$ root and root $\leq$ x1: `f(x) = x*x - 4, x0 = 1.9, x1 = 2.1, n = 10, epsilon = 0.00001`
- root $>$ x1: `f(x) = x*x - 4, x0 = 2.5, x1 = 3.5, n = 10, epsilon = 0.00001`
- `f(x0) == f(x1)`: `f(x) = x*x - 4, x0 = -4.0, x1 = 4.0, n = 10, epsilon = 0.00001`
- Unsolvable function: `f(x) = x*x + 4, x0 = -5, x1 = +5, n = 100, epsilon = 0.00001`

B. White Box Tests

- Coverage Tests [**2**]

Answer:
LIP Tests in Q 2(a)iv suffice. Refer respectively.

- Line Coverage (Test: Any of ((T1 or T3) and T2), T5)
- Branch Coverage (Test: Any of ((T1 or T3) and T2), T5)
- Path Coverage (Test: All of (T1 or T3), T2, T5)

C. Performance Tests

Answer:

- Precision Tests – Correctness of solution, Choice of ϵ, ... [**1**]
  Over the following tests, the solution should get closer and closer to 2.0.
  `f(x) = x*x - 4, x0 = 1.5, x1 = 2.5, n = 100, epsilon = 0.1`
  `f(x) = x*x - 4, x0 = 1.5, x1 = 2.5, n = 100, epsilon = 0.01`
  `f(x) = x*x - 4, x0 = 1.5, x1 = 2.5, n = 100, epsilon = 0.001`
  `f(x) = x*x - 4, x0 = 1.5, x1 = 2.5, n = 100, epsilon = 0.0001`
  `f(x) = x*x - 4, x0 = 1.5, x1 = 2.5, n = 100, epsilon = 0.00001`
- Stability Tests – Stability of computation [**1**]
  `f(x) = x*x - 4, x0 = -4.001, x1 = 4.0001, n = 10, epsilon = 0.00001` converges, but
  `f(x) = x*x - 4, x0 = -4.0, x1 = 4.0, n = 10, epsilon = 0.00001` fails.

Note: These test cases include expression for function `f(x)` and values for parameters of `SecantSolver` function. Many scenarios and hence test cases may overlap. That is, one test case may test more than one scenarios. In order to avoid duplication and minimize effort, you should uniquely number every test case, and make a cross reference in the scenario in the test plan as to which test case/s would cover this scenario. If some test already designed also works for a later scenario, you may just refer to it.

3. Write the output from the following code: [**0.5*20=10**]

```
#include <vector> #include <algorithm> #include <iostream> #include <cmath>
using namespace std;

struct number { int val; number() : val(0) {}
    int operator()() { return (val & 0x2) ? -2 * val++ : val++; } };

bool IsOdd(int i) { return (i % 2) == 1; }

bool IsNeg(int i) { return i < 0; }

struct compare : public binary_function<int, int, bool> {
    bool operator()(int x, int y) { return abs(x) > abs(y); } };

struct sum_of_square : public unary_function<double, void> { int sum; unsigned int count;
    sum_of_square() : sum(0), count(0) {} void operator()(int i) { sum += i * i; ++count; } };

int main() {
    vector<int> myvector(8);

    generate(myvector.begin(), myvector.end(), number());
    cout << "Filled Vector is:" << endl;
    for (vector<int>::const_iterator it = myvector.begin(); it != myvector.end(); ++it)
        cout << *it << " ";
    cout << endl << endl;

    int nOdd = count_if(myvector.begin(), myvector.end(), IsOdd);
    cout << "Vector contains " << nOdd << " odd values" << endl << endl;

    int nNeg = count_if(myvector.begin(), myvector.end(), IsNeg);
    cout << "Vector contains " << nNeg << " negative values" << endl << endl;

    sort(myvector.begin(), myvector.end(), compare());
    cout << "Sorted Vector is:" << endl;
    for (vector<int>::const_iterator it = myvector.begin(); it != myvector.end(); ++it)
        cout << *it << " ";
    cout << endl << endl;

    sum_of_square result = for_each(myvector.begin() + 2, myvector.end(), sum_of_square());
    cout << "SoP of " << result.count << " numbers is " << result.sum << endl << endl;

    return 0;
}
```

Answer:

```
Filled Vector is:
0 1 -4 -6 4 5 -12 -14

Vector contains 2 odd values

Vector contains 4 negative values

Sorted Vector is:
-14 -12 -6 5 -4 4 1 0

SoP of 6 numbers is 94
```

Note for marking: It is possible that someone does not get the `return (val & 0x2) ? -2 * val++ : val++;` *logic right and fills up the vector with wrong elements. Naturally, marks are to be deducted based on the number of mismatches. However, after that, the vector will be assumed to have been filled with these 'wrong' numbers and the remaining answers will be evaluated based on that. This will ensure that a student is not unduly penalized all through the question for getting one part wrong.*

Hint for Q3:

A brief documentation on STL (the parts used in the question) is reproduced here for quick reference.

Standard Template Library: Algorithms

The header `<algorithm>` defines a collection of functions especially designed to be used on ranges of elements.

A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the STL containers. Notice though, that algorithms operate through iterators directly on the values, not affecting in any way the structure of any possible container (it never affects the size or storage allocation of the container).

Select functions are:

`for_each`: *Apply function to range*

Applies function fn to each of the elements in the range [first,last).

```
template <class InputIterator, class Function>
   Function for_each (InputIterator first, InputIterator last, Function fn);
```

`generate`: *Generate values for range with function*

Assigns the value returned by successive calls to gen to the elements in the range [first,last).

```
template <class ForwardIterator, class Generator>
  void generate (ForwardIterator first, ForwardIterator last, Generator gen);
```

`count_if`: *Return number of elements in range satisfying condition*

Returns the number of elements in the range [first,last) for which pred is true.

```
template <class InputIterator, class UnaryPredicate>
  typename iterator_traits<InputIterator>::difference_type
    count_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

`sort`: *Sort elements in range*

Sorts the elements in the range [first,last) into ascending order. The elements are compared using `comp`. Equivalent elements are not guaranteed to keep their original relative order.

```
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```