# April 2019

# Fundamental IT Engineer Examination (Afternoon)

**Questions must be answered in accordance with the following:**

| Question Nos. | Q1 – Q6 | Q7 , Q8 |
|---|---|---|
| Question Selection | Compulsory | Select 1 of 2 |
| Examination Time | 13:30 - 16:00 (150 minutes) | |

**Instructions:**

1. Use a pencil. If you need to change an answer, erase your previous answer completely and neatly. Wipe away any eraser debris.

2. Mark your examinee information and test answers in accordance with the instructions below. Your answer will not be graded if you do not mark properly. Do not mark or write on the answer sheet outside of the prescribed places.

   (1) **Examinee Number**
   Write your examinee number in the space provided, and mark the appropriate space below each digit.

   (2) **Date of Birth**
   Write your date of birth (in numbers) exactly as it is printed on your examination admission card, and mark the appropriate space below each digit.

   (3) **Question Selection**
   For **Q7** and **Q8**, mark the ⓢ of the question you select to answer in the "Selection Column" on your answer sheet.

   (4) **Answers**
   Mark your answers as shown in the following sample question.

   [Sample Question]
   In which month is the spring Fundamental IT Engineer Examination conducted?

   Answer group
   a) March b) April c) May d) June

   Since the correct answer is " b) April ", mark your answer sheet as follows:

   [Sample Answer]

| | |
|---|---|
| Sample | ⓐ ● ⓒ ⓓ ⓔ ⓕ ⓖ ⓗ ⓘ ⓙ |

**Do not open the exam booklet until instructed to do so.**
**Inquiries about the exam questions will not be answered.**

## Notations used in the pseudo-language

In questions that use pseudo-language, the following notations are used unless otherwise stated:

[Declaration, comment, and process]

| | Notation | Description |
|---|---|---|
| | *type*: *var1*, ..., *array1*[], ... | Declares variables *var1*, ... , and/or arrays *array1*[], ... , by data *type* such as INT and CHAR. |
| | FUNCTION: *function*(*type*: *arg1*, ...) | Declares a *function* and its arguments *arg1*, ... . |
| | /* comment */ | Describes a comment. |
| Process | *variable* ← *expression* ; | Assigns the value of the *expression* to the *variable*. |
| Process | *function* (*arg1*, ...) ; | Calls the *function* by passing / receiving the arguments *arg1*, ... . |
| Process | IF (*condition*) { *process1* } ELSE { *process2* } | Indicates the selection process. If the *condition* is true, then *process1* is executed. If the *condition* is false, then *process2* is executed, when the optional ELSE clause is present. |
| Process | WHILE (*condition*) { *process* } | Indicates the "WHILE" iteration process. While the *condition* is true, the *process* is executed repeatedly. |
| Process | DO { *process* } WHILE (*condition*); | Indicates the "DO-WHILE" iteration process. The *process* is executed once, and then while the *condition* is true, the *process* is executed repeatedly. |
| Process | FOR (*init* ; *condition* ; *incr*) { *process* } | Indicates the "FOR" iteration process. While the *condition* is true, the *process* is executed repeatedly. At the start of the first iteration, the process *init* is executed before testing the *condition*. At the end of each iteration, the process *incr* is executed before testing the *condition*. |

[Logical constants]

true, false

[Operators and their precedence]

| Type of operation | Unary | Arithmetic | | Relational | Logical | |
|---|---|---|---|---|---|---|
| Operators | +, −, not | ×, ÷, % | +, − | >, <, ≥, ≤, =, ≠ | and | or |
| Precedence | High ◄ | | | | ► | Low |

Note: With division of integers, an integer quotient is returned as a result.
The "%" operator indicates a remainder operation.

Questions **Q1** through **Q6** are all **compulsory**. Answer every question.

**Q1.** Read the following description of a hybrid encryption schema, and then answer Subquestion.

In a public-key cryptosystem, hybrid encryption uses both public-key and symmetric-key encryption. In hybrid encryption schema, public-key encryption is used for key encapsulation and symmetric-key encryption for data encapsulation. The key used for data encapsulation and encapsulated by public-key encryption is called the session key.
A hybrid encryption schema is shown below.

[Sender's processes]
(1) Generate session key `sk` for encryption. Session key `sk` can be generated randomly.
(2) Encrypt plain message `m` using session key `sk` and symmetric-key encryption function `SENC`.
(3) Encrypt session key `sk` using public-key encryption function `PENC`.
(4) Transmit encrypted message `c1` and encrypted session key `c2` to the receiver.

[Receiver's processes]
(1) Receive encrypted message `c1` and encrypted session key `c2`.
(2) Decrypt the received encrypted session key `c2`.
(3) Decrypt the received encrypted message `c1`. The resulting decrypted message is plain message `m`.

Figure 1 shows the hybrid encryption schema, where a rectangle indicates a function and a parallelogram indicates a variable.

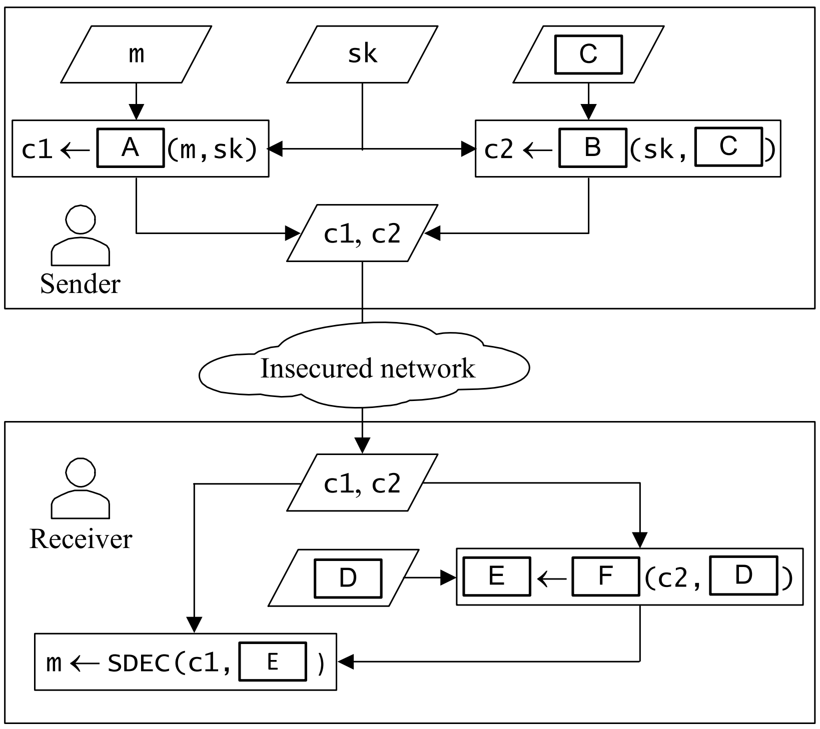


Figure 1 Hybrid encryption schema

The following notations are used in Figure 1.

| Notation | Description |
|---|---|
| Functions | |
| PENC | Public-key encryption function. |
| PDEC | Public-key decryption function (inverse of PENC). |
| SENC | Symmetric-key encryption function. |
| SDEC | Symmetric-key decryption function (inverse of SENC). |
| Variables | |
| m | Plain message |
| c1, c2 | Encrypted messages |
| ssk | Sender's private key |
| spk | Sender's public key |
| rsk | Receiver's private key |
| rpk | Receiver's public key |
| sk | Session key |

## Subquestion

From the answer groups below, select the correct answers to be inserted in each blank $\boxed{\qquad\quad}$ in the above figure.

Answer group for A, B and F

a) `PDEC`
b) `PENC`
c) `SDEC`
d) `SENC`

Answer group for C, D and E

a) `rpk`
b) `rsk`
c) `sk`
d) `spk`
e) `ssk`

**Q2.** Read the following description of virtual memory and paging, and then answer Subquestions 1 and 2.

Virtual memory is a technique that allows processes to use more memory than the system is physically equipped for. In this technique, each process has its own virtual memory consisting of fixed-sized fragments called *pages*, while the physical memory is managed by fragments of equal size called *page frames*. Some pages that are being used or were recently used are placed into page frames in the physical memory, and the other pages are placed in auxiliary storage such as a hard disk. Mappings between pages and page frames are managed by the operating system. When a process demands a page that is not in physical memory, a page fault occurs and the operating system loads the demanded page from the auxiliary storage to a page frame. This strategy of memory management is known as demand paging.

Figure 1 shows the concept of demand paging.

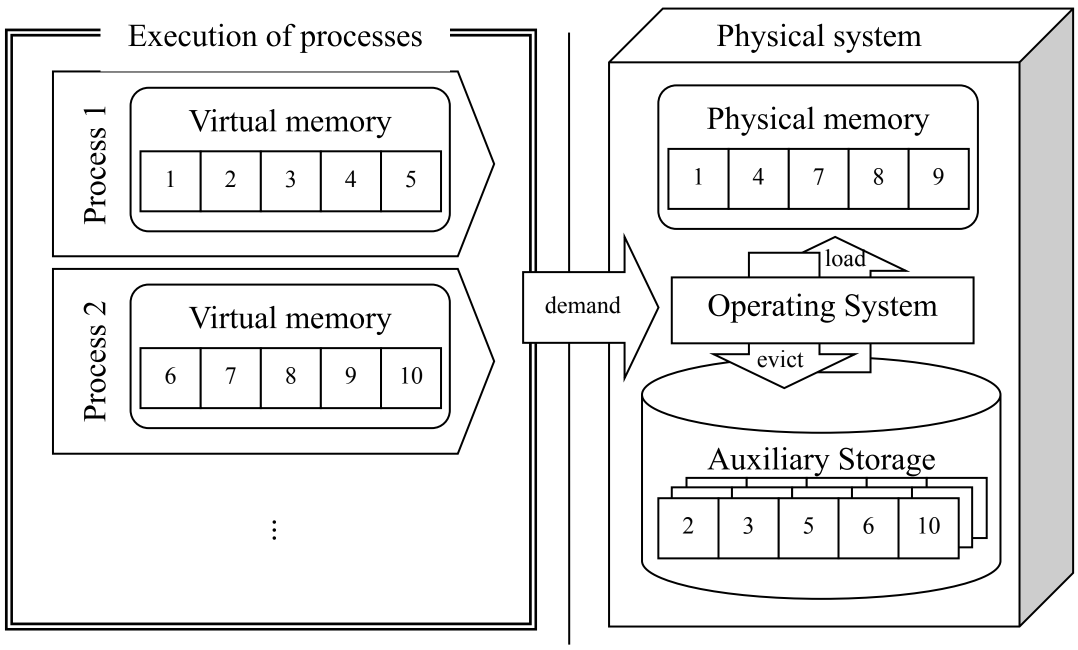


Figure 1 Concept of demand paging

When a page is not in physical memory, it takes additional time to load the page from auxiliary storage. Therefore, the frequent occurrence of page faults can severely degrade performance.
The expected data access time $T$ to access a data in a page can be formulated as follows:

$$T = ((1 - R_f) \times T_m) + (R_f \times T_a)$$

where:

$R_f$ : page fault rate
$T_m$ : data access time when the page is in physical memory
$T_a$ : data access time when the page is in auxiliary storage

Consider a system where $T_m$ is 200 nanoseconds, and $T_a$ is 4 milliseconds. Table 1 shows examples of page fault rates and their expected data access times. Assuming that there are enough empty page frames in the physical memory.

Table 1 Examples of page fault rates and their expected data access times

| Page fault rate | Expected data access time (*) |
|---|---|
| 1% | 40.2 microseconds |
| 0.1% | $\boxed{\text{A}}$ microseconds |
| 0.01% | $\boxed{\text{B}}$ nanoseconds |

Note: (*) Rounded to 3 significant figures.

### Subquestion 1

From the answer group below, select the correct answer to be inserted in each blank $\boxed{\phantom{AAAA}}$ in Table 1.

Answer group

| | | |
|---|---|---|
| a) 4.02 | b) 4.20 | c) 6.00 |
| d) 40.2 | e) 42.0 | f) 60.0 |
| g) 402 | h) 420 | i) 600 |

**Subquestion 2**

From the answer groups below, select the correct answer to be inserted in each blank $\boxed{\quad\quad}$ in the following description.

[Page replacement]

When a page fault occurs and no empty page frame is available, the operating system evicts the content of a page frame to the auxiliary storage to make room for the demanded page. The page frame that is evicted is called the victim.

Several algorithms can be used to determine the page frame to be chosen as victim. Two algorithms are considered here: first-in, first-out (FIFO), and the least recently used (LRU) algorithm.

In the FIFO algorithm, the page frame that has held the same page for the longest period is chosen as victim. In the LRU algorithm, the page frame that has held a page unused for the longest period is chosen as victim.

Figure 2 illustrates an example of system behavior for each of the above two algorithms. In this system, there are 3 page frames X, Y and Z in the physical memory, and the process running on the system demands 5 pages (pages 1 through 5) in order 1 2 3 4 1 2 5 1 2 3 4 5. The pages loaded with page faults are underlined.

FIFO algorithm

| Demand for the pages | | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page frames | X | $\underline{1}$ | 1 | 1 | $\underline{4}$ | 4 | 4 | $\underline{5}$ | 5 | 5 | 5 | 5 | 5 |
| | Y | | $\underline{2}$ | 2 | 2 | $\underline{1}$ | 1 | 1 | 1 | 1 | $\underline{3}$ | 3 | 3 |
| | Z | | | $\underline{3}$ | 3 | 3 | $\underline{2}$ | 2 | 2 | 2 | 2 | $\underline{4}$ | 4 |

LRU algorithm

| Demand for the pages | | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4▼ | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page frames | X | $\underline{1}$ | 1 | 1 | $\underline{4}$ | 4 | 4 | $\underline{5}$ | 5 | | | | |
| | Y | | $\underline{2}$ | 2 | 2 | $\underline{1}$ | 1 | 1 | 1 | | | | |
| | Z | | | $\underline{3}$ | 3 | 3 | $\underline{2}$ | 2 | 2 | | | | |

Note: The shaded parts are intentionally left blank.

Figure 2 Example of system behavior for each algorithm

In the FIFO algorithm, 9 page faults occur in Figure 2. In the LRU algorithm, at the second demand for page 4 (marked by "▼"), the page in page frame $\boxed{\quad C \quad}$ is replaced. In total, when using the LRU algorithm, $\boxed{\quad D \quad}$ page faults occur in Figure 2.

Figure 3 illustrates another example of system behavior for each of the two algorithms. Figure 3 is identical to Figure 2 except for an additional page frame W.

FIFO algorithm

| Demand for the pages | | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page frames | W | $\underline{1}$ | 1 | 1 | 1 | 1 | 1 | $\underline{5}$ | 5 | | | | |
| | X | | $\underline{2}$ | 2 | 2 | 2 | 2 | 2 | $\underline{1}$ | | | | |
| | Y | | | $\underline{3}$ | 3 | 3 | 3 | 3 | 3 | | | | |
| | Z | | | | $\underline{4}$ | 4 | 4 | 4 | 4 | | | | |

LRU algorithm

| Demand for the pages | | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page frames | W | $\underline{1}$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | $\underline{5}$ |
| | X | | $\underline{2}$ | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | Y | | | $\underline{3}$ | 3 | 3 | 3 | $\underline{5}$ | 5 | 5 | 5 | $\underline{4}$ | 4 |
| | Z | | | | $\underline{4}$ | 4 | 4 | 4 | 4 | 4 | $\underline{3}$ | 3 | 3 |

Figure 2 Another example of system behavior for each algorithm

In Figure 3, $\boxed{\text{E}}$ page faults occur in the FIFO algorithm, and 8 page faults occur in the LRU algorithm.

Answer group for C

| | | |
|---|---|---|
| a) X | b) Y | c) Z |

Answer group for D and E

| | | |
|---|---|---|
| a) 5 | b) 6 | c) 7 |
| d) 8 | e) 9 | f) 10 |

**Q3.** Read the following description of a database for managing the courses and grades of university students, and then answer Subquestions 1 through 3.

A university maintains a database in order to keep track of information on student profiles, courses and sections offered, and grades obtained.

The database is composed of four tables. The structure of the table and sample data for each table are shown below. The primary keys are underlined.

(1) Student Table

| <u>StudentNo</u> | Name | Department | Class | Program |
|---|---|---|---|---|
| 1599221 | Steve Kam | CS | Sophomore | BSC |
| 1599222 | Mathew Ken | CS | Sophomore | BSC |
| 1599223 | Allen Strew | CS | Sophomore | BSC |
| 1599224 | Stephen Ford | CS | Sophomore | BSC |

(2) Course Table

| <u>CourseNo</u> | CourseName | CreditHours | Department |
|---|---|---|---|
| CS173 | Discrete Mathematics | 3 | CS |
| CS225 | DataStructures | 3 | CS |
| CS311 | Database Systems | 3 | CS |
| CS377 | Algorithms | 3 | CS |

(3) Section Table

| <u>SectionNo</u> | CourseNo | Instructor | Semester | Year |
|---|---|---|---|---|
| 1112 | CS 311 | John | Summer | 2018 |
| 1113 | CS 173 | Clark | Fall | 2018 |
| 1114 | CS 377 | Abraham | Fall | 2018 |
| 1115 | CS 225 | Clark | Spring | 2019 |
| 1116 | CS 311 | John | Summer | 2019 |

(4) GradeReport Table

| <u>StudentNo</u> | <u>SectionNo</u> | Grade |
|---|---|---|
| 1599221 | 1112 | B |
| 1599221 | 1113 | C |
| 1599222 | 1113 | B |
| 1599222 | 1115 | A |
| 1599223 | 1116 | (null) |
| 1599224 | 1112 | A |
| 1599224 | 1114 | A |

A course has one or more sections. SectionNo uniquely identities each section.
Grade shows the result the student obtained in the given section. It is a one-character code, such as A(excellent), B(good), C(fair), …, or "null" if the section is not completed.
All students register at least one section, that is, every StudentNo in Student table appears in GradeReport table.

## Subquestion 1

From the answer groups below, select the correct answer to be inserted in each blank $\boxed{\phantom{xxxx}}$ in the following SQL statement.

The SQL statement SQL1 outputs the student number, name, and department of students who got As in all courses they completed.

```
-- SQL1 --
SELECT StudentNo, Name, Department
FROM Student S
WHERE [    A    ]
       (SELECT StudentNo
        FROM GradeReport
        WHERE StudentNo = S.StudentNo AND [    B    ])
```

From the sample data of each table shown in the description, SQL1 outputs the following result:

| StudentNo | Name | Department |
|---|---|---|
| 1599224 | Stephen Ford | CS |

Answer group for A

a) `EXISTS`
b) `IN`
c) `NOT EXISTS`
d) `NOT IN`

Answer group for B

a) `Grade != 'A'`
b) `Grade < 'A'`
c) `Grade = 'A'`

**Subquestion 2**

From the answer group below, select the correct answer to be inserted in each blank $\boxed{\phantom{XXXX}}$ in Figure 1.

In SQL, UNION, INTERSECT and EXCEPT are binary operations on relations.
Assuming that there are two relations R and S,

(1) R UNION S results in a relation that contains the tuple/records that are either in R, or in S, or in both.

(2) R INTERSECT S results in a relation that contains the tuple/records that are in both R and S.

(3) R EXCEPT S results in a relation that contains the tuple/records that are in R but not in S.

Figure 1 shows an example of the binary operations

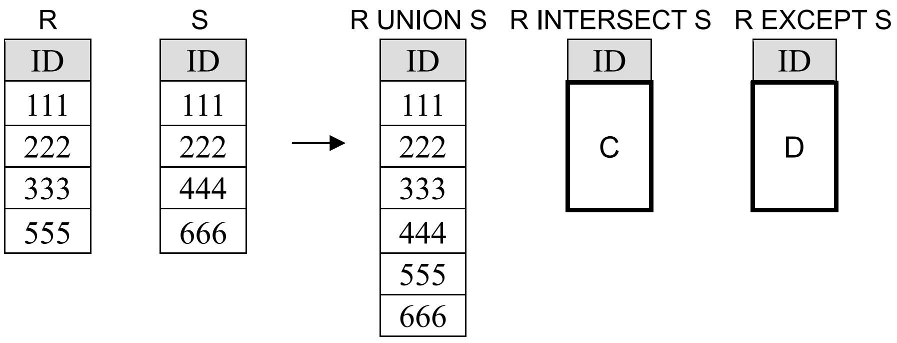


Figure 1 Example of binary operations

Answer group for C and D

a)

| |
|---|
| 111 |
| 222 |

b)

| |
|---|
| 333 |
| 444 |
| 555 |
| 666 |

c)

| |
|---|
| 333 |
| 555 |

d)

| |
|---|
| 444 |
| 666 |

**Subquestion 3**

From the answer groups below, select the correct answer to be inserted in each blank [ ] in the following SQL statement.

The SQL statement SQL2 outputs students who completed all courses taught by the instructor named Clark in 2018 and 2019.

```
-- SQL2 --
SELECT StudentNo, Name, Department
FROM Student S
WHERE [    E    ] (
        (SELECT SectionNo
         FROM Section
         WHERE Instructor = 'Clark'
           AND (Year = 2018 OR Year = 2019))
        [    F    ]
        (SELECT SectionNo
         FROM GradeReport G
         WHERE G.StudentNo = S.StudentNo)  )
```

From the sample data of each table shown in the description, SQL2 outputs the following result:

| StudentNo | Name | Department |
|---|---|---|
| 1599222 | Mathew Ken | CS |

Answer group for E

a) `EXISTS` b) `IN`
c) `NOT EXISTS` d) `NOT IN`

Answer group for F

a) `EXCEPT` b) `INTERSECT` c) `UNION`

**Q4.** Read the following description of Hamming code, and then answer Subquestions 1 and 2.

Hamming code is extra information to be added to data to detect/correct errors, if any, during transmission.
For example, Hamming code C(7, 4) implies that while transmitting 4 data-bits, the sender computes three additional parity bits, assembles them with the data to convert it to 7 code-bits and transmits these code-bits to the receiver. The receiver computes the three parity bits from the received 7 code-bits and checks for errors. If no error has occurred in transmission, the receiver correctly receives 4 data-bits from the code-bits. If even a single bit has been changed in transmission, the value of the three parity bits will indicate the position of the error, which can then be corrected.

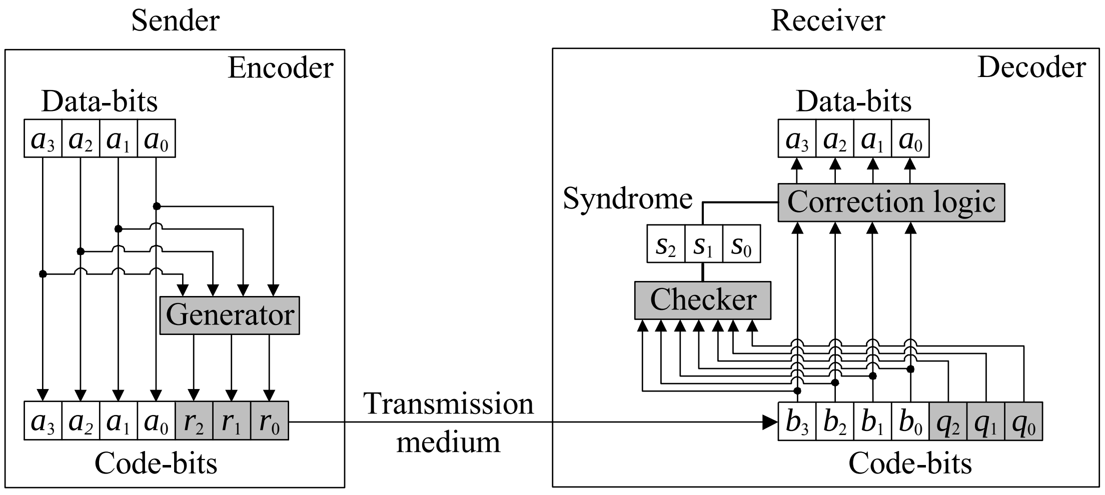


Figure 1 The structure of encoder and decoder for Hamming code

Figure 1 shows the structure of the encoder at a sender site and the decoder at a receiver site for Hamming code C(7, 4). At the sender site, three parity bits $r_2$, $r_1$, and $r_0$ are generated from 4 data-bits $a_3$, $a_2$, $a_1$, and $a_0$, and appended to the data-bits before transmission. The parity bits $r_2$, $r_1$ and $r_0$ are generated by using the equations below:

$$r_2 = a_3 \oplus a_1 \oplus a_0$$
$$r_1 = a_3 \oplus a_2 \oplus a_1$$
$$r_0 = a_2 \oplus a_1 \oplus a_0$$

where symbol "$\oplus$" designates the XOR (exclusive-OR) operation.

At the receiver site, upon receiving 7 code-bits $b_3$, $b_2$, $b_1$, $b_0$, $q_2$, $q_1$ and $q_0$, the checker generates 3 syndrome-bits $s_2$, $s_1$ and $s_0$ by using the equations given below. The syndrome-bits identify the syndrome and the location of a single bit error.

$$\begin{aligned} s_2 &= b_3 \oplus b_1 \oplus b_0 \oplus q_2 \\ s_1 &= b_3 \oplus b_2 \oplus b_1 \oplus q_1 \\ s_0 &= b_2 \oplus b_1 \oplus b_0 \oplus q_0 \end{aligned}$$

The pattern of the 3 syndrome-bits identifies an error, if any, in the received code-bits as well as the location of the error.

Table 1 shows the correspondence of the values of syndrome-bits with the location of the error. For example, if $q_0$ is in error, $s_0$ is the only bit affected and the syndrome-bits are 001.

Table 1 Correspondence of syndrome-bits with error location

| Syndrome-bits ($s_2$, $s_1$ , $s_0$) | Error location |
|---|---|
| 000 | No error |
| 001 | $q_0$ |
| … | … |
| 111 | $b_1$ |

### Subquestion 1

From the answer groups below, select the correct answer to be inserted in each blank $\boxed{\phantom{xxxx}}$ in the following description.

The data-bits 1001 become the code-bits $\boxed{\text{A}}$. When the receiver receives the code-bits, the checker generates the syndrome-bits $\boxed{\text{B}}$, and identifies the data-bits as 1001.

The data-bits 0110 become the code-bits 0110100. However, if the receiver receives the code-bits 1110100 because of a transmission error, the checker generates syndrome-bits $\boxed{\text{C}}$.

Answer group for A

a) 1001000 b) 1001011

c) 1001100 d) 1001110

Answer group for B and C

a) 000 b) 010 c) 011

d) 100 e) 110 f) 111

**Subquestion 2**

From the answer groups below, select the correct answer to be inserted in each blank $\boxed{\phantom{XXXXX}}$ in the following description.

When the checker at the receiver generates syndrome-bits 110, the checker realizes that there is an error in $\boxed{\text{D}}$ bit.

Hamming code C(7, 4) cannot correct two-bit errors. Assuming that a sender generates code-bits 1101000 from data-bits 1101, and sends them to a receiver. However, a two-bit error has incurred on the code-bits during the transmission, and the receiver receives code-bits 0001000. Then the checker generates syndrome-bits 101, and realizes that there is an error in bit $\boxed{\text{E}}$. Finally, the receiver receives incorrect data-bits $\boxed{\text{F}}$ after error correction.

Answer group for D and E

a) $b_0$ b) $b_1$

c) $b_2$ d) $b_3$

Answer group for F

a) 0000 b) 0011 c) 0101

d) 1001 e) 1101 f) 1110

**Q5.** Read the following description of the software design of a stock trading application, and then answer Subquestions 1 and 2.

Stock trading consists of traders selling and buying stock items throughout the day. Each trader has several stock items in different quantities and cash for buying. The system allows short selling, where a trader has a negative cash amount or negative stock quantity during the day but it should be positive by the end of the day. This happens when traders borrow stocks to sell or cash from other traders during the day and return it at day's end.
A program is written to update the trader record (Figure 1) and report all negative stock quantities or negative cash amounts (Figure 2) at the end of the day.

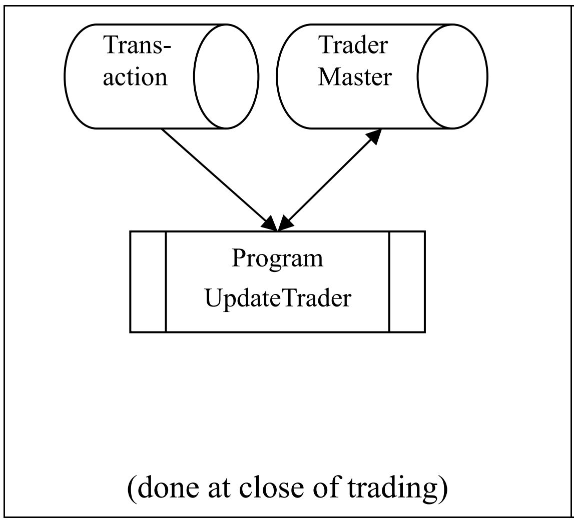


Figure 1 Update trader record

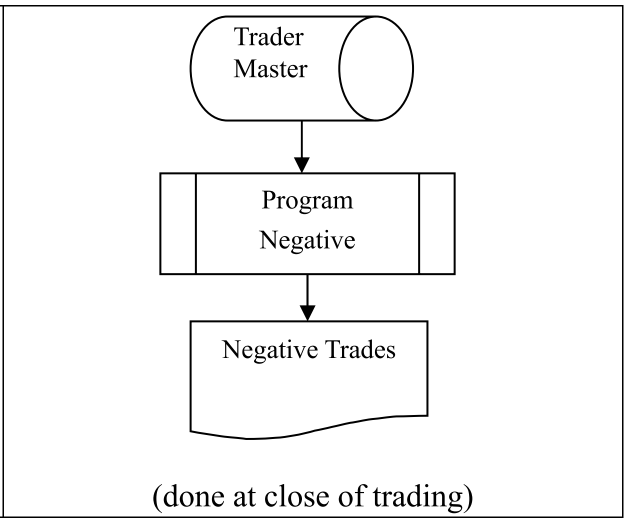


Figure 2 Report all negative trades

[Program Description for UpdateTrader]
Once trading is closed, the program updates TraderMaster file using Transaction file.
(1) The sequential file Transaction contains a chronological record of each trade (stock items bought or sold) for the given day.

File: Transaction ( TDate, TTime, TID, TItemID, TQty, TPrice, TBuyID, TSellID )

| Field | Description | Field | Description |
|---|---|---|---|
| TDate | Date of transaction | TQty | Quantity bought/sold |
| TTime | Time of transaction | TPrice | Unit price of stock item |
| TID | Unique transaction ID | TBuyID | Trader ID of buyer |
| TItemID | ID of stock item traded | TSellID | Trader ID of seller |

(2) The indexed file TraderMaster (buyer and seller) contains stocks held by traders. The file is indexed by MID and MItemID. Each record contains each stock item a trader has (or owns). Cash is considered an item. The file is updated after trading close.

File: TraderMaster (MID, MItemID, MQty, MPrice)

| Field | Description |
|---|---|
| MID | Unique ID of trader |
| MItemID | Unique ID of stock item<br>When item is cash, value is “000” |
| MQty | Quantity of stock item held by the trader<br>When item is cash, value is 1 |
| MPrice | Unit price of stock item (the latest price)<br>When item is cash, value shows the balance |

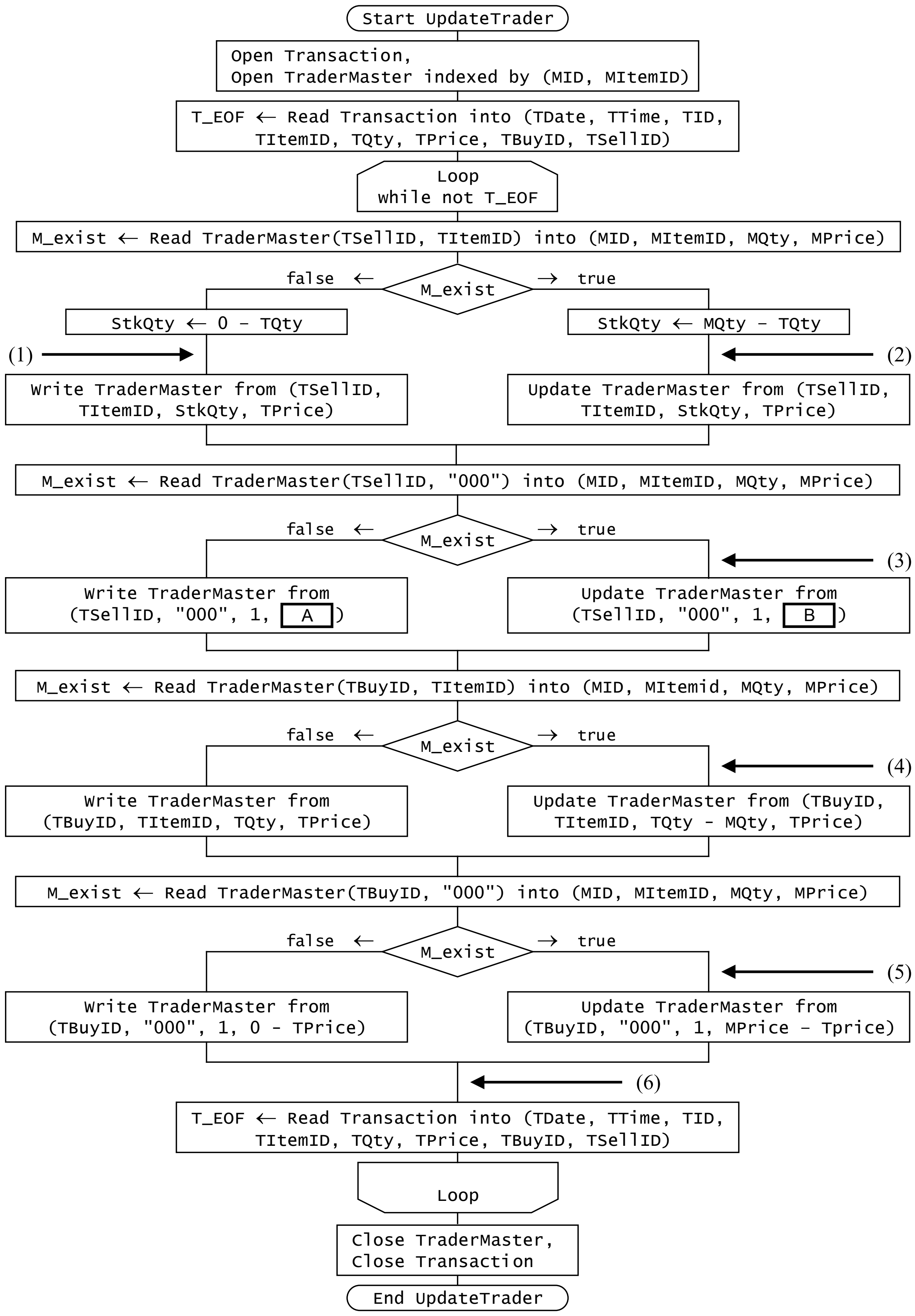
[Program Description for Negative]

(1) The program prints all Traders with a negative cash position or negative stock quantity at close of trading. This program is executed after UpdateTrader.

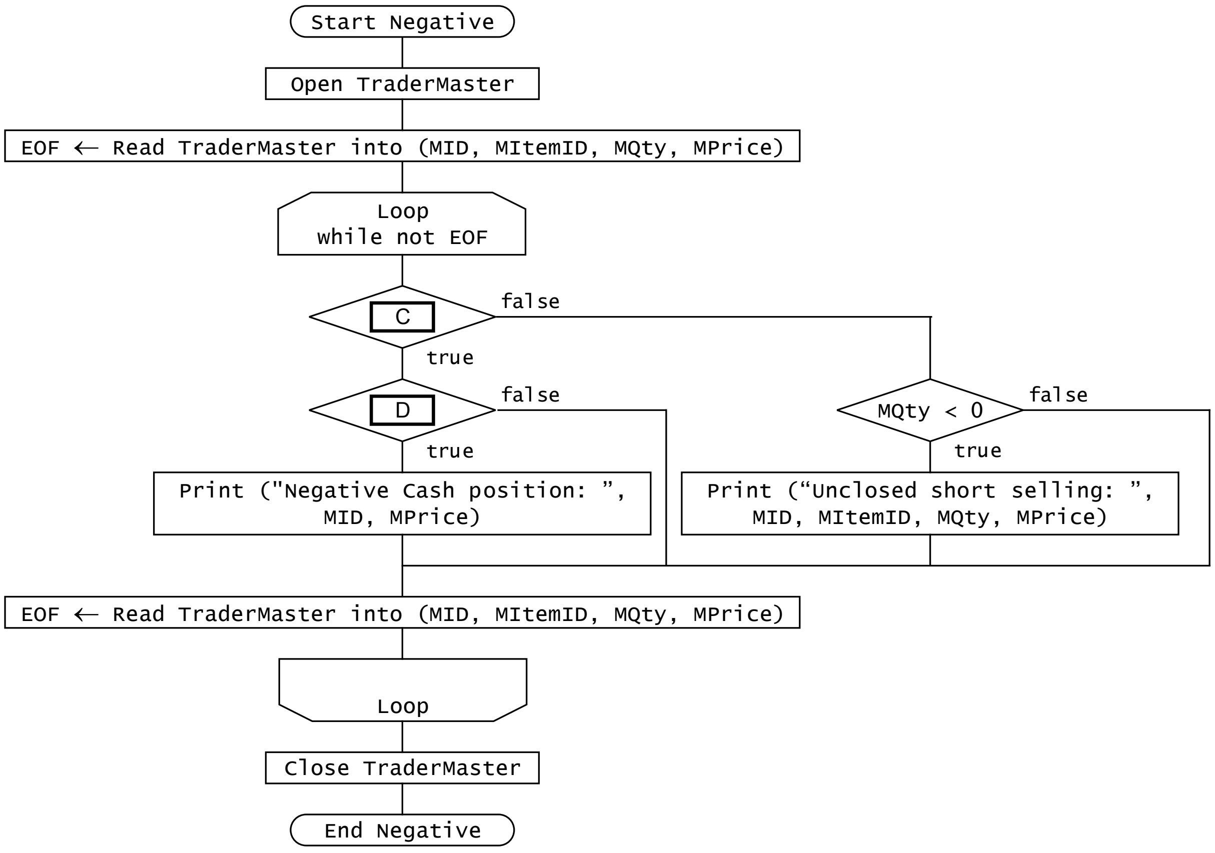
In the programs UpdateTrader and Negative, the following statements are used:

- Open *filename* [indexed by (*index1*, *index2*, …)]
  Open *filename* opens *filename* for sequential reading. Open *filename* indexed by (*index1*, *index2*, …) opens *filename* for indexed reading with the field list as index.
- Read *filename* into (*field1*, *field2*, …)
  Read one record from *filename*. The fields of the record are stored in the variables *field1*, *field2*, … . When the end of the file is reached, the statement returns `true`.
- Read *filename* (*index1*, *index2*, …) into (*field1*, *field2*, …)
  The file *filename* is searched for in the records with index equal to *index1*, *index2*, … . The fields of the record are stored in variables *field1*, *field2*, … . The statement returns `true` when the specified record exists.
- Write *filename* from (*value1*, *value2*, …)
  Write one record into *filename*. The fields of the record are constructed from the values *value1*, *value2*, … .
- Update *filename* from (*value1*, *value2*, …)
  Update the given record into *filename*. The fields of the record are constructed from values *value1*, *value2*, … .
- Close *filename*
  Close *filename*.

[Program UpdateTrader]


Start UpdateTrader
Open Transaction,
Open TraderMaster indexed by (MID, MItemID)
T_EOF ← Read Transaction into (TDate, TTime, TID, TItemID, TQty, TPrice, TBuyID, TSellID)
Loop while not T_EOF
M_exist ← Read TraderMaster(TSellID, TItemID) into (MID, MItemID, MQty, MPrice)
false ← M_exist → true
StkQty ← 0 – TQty
StkQty ← MQty – TQty
(1)
(2)
Write TraderMaster from (TSellID, TItemID, StkQty, TPrice)
Update TraderMaster from (TSellID, TItemID, StkQty, TPrice)
M_exist ← Read TraderMaster(TSellID, "000") into (MID, MItemID, MQty, MPrice)
false ← M_exist → true
(3)
Write TraderMaster from (TSellID, "000", 1, A)
Update TraderMaster from (TSellID, "000", 1, B)
M_exist ← Read TraderMaster(TBuyID, TItemID) into (MID, MItemid, MQty, MPrice)
false ← M_exist → true
(4)
Write TraderMaster from (TBuyID, TItemID, TQty, TPrice)
Update TraderMaster from (TBuyID, TItemID, TQty - MQty, TPrice)
M_exist ← Read TraderMaster(TBuyID, "000") into (MID, MItemID, MQty, MPrice)
false ← M_exist → true
(5)
Write TraderMaster from (TBuyID, "000", 1, 0 - TPrice)
Update TraderMaster from (TBuyID, "000", 1, MPrice – Tprice)
(6)
T_EOF ← Read Transaction into (TDate, TTime, TID, TItemID, TQty, TPrice, TBuyID, TSellID)
Loop
Close TraderMaster,
Close Transaction
End UpdateTrader

[Program Negative]


Start Negative
Open TraderMaster
EOF ← Read TraderMaster into (MID, MItemID, MQty, MPrice)
Loop
while not EOF
C
false
true
D
false
true
MQty < 0
false
true
Print ("Negative Cash position: ", MID, MPrice)
Print ("Unclosed short selling: ", MID, MItemID, MQty, MPrice)
EOF ← Read TraderMaster into (MID, MItemID, MQty, MPrice)
Loop
Close TraderMaster
End Negative

## Subquestion 1

From the answer groups below, select the correct answer to be inserted in each blank $\boxed{\phantom{xxxx}}$ in the program.

Answer group for A and B

| | |
|---|---|
| a) `- MPrice` | b) `- TPrice` |
| c) `MPrice` | d) `TPrice` |
| e) `MPrice + TPrice` | f) `MPrice - TPrice` |

Answer group for C and D

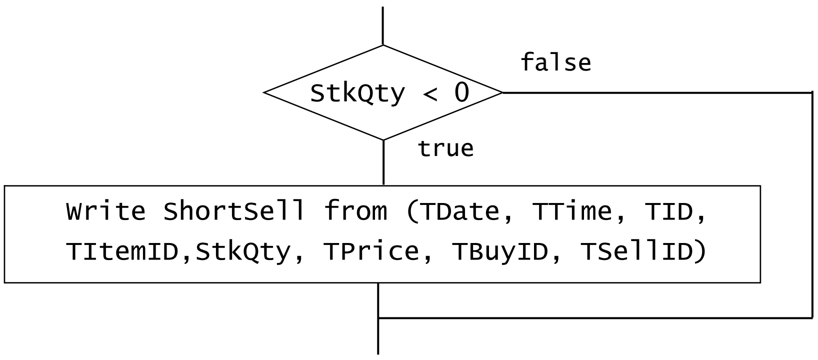
| | |
|---|---|
| a) `MItemID = "000"` | b) `MPrice < 0` |
| c) `MPrice ≥ 0` | d) `MQty < 0` |
| e) `MQty ≥ 0` | |

## Subquestion 2

From the answer group below, select the correct answer to be inserted in the blank $\boxed{\quad\quad}$ in the following description.

To monitor short selling, a sequential file ShortSell is used in the program UpdateTrader. The file ShortSell contains all transactions that were sold short or in negative quantity. The fields of the file ShortSell are identical to those of the file Transaction, except TQty contains a negative value for quantity in the former file.

To monitor short selling, the following process should be inserted immediately before the statement pointed to by $\boxed{\quad E \quad}$. Here, the file ShortSell is opened and closed properly in the program.


StkQty < 0
false
true
Write ShortSell from (TDate, TTime, TID, TItemID,StkQty, TPrice, TBuyID, TSellID)

Answer group for E

a) (1) b) (2) c) (3)

d) (4) e) (5) f) (6)

**Q6.** Read the following description of a program to solve the subarray sum problem, and then answer Subquestions 1 through 3.

(See the top of this booklet for the notations used in the pseudo-language.)

Given an array of positive integers and a target sum, the problem is to find all subarrays whose sum is equal to the target sum. A subarray is a part of the given array composed of contiguous elements. For example, when the target sum is 14, the shaded subarray in Figure 1 satisfies the given condition. The 2-point method is a more efficient way to solve this problem than the naïve method.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 3 | 2 | 5 | 1 | 1 | 7 | 3 |

Figure 1 Example of an array

[Program Description]

(1) The subprogram `sSumNaive` checks whether all possible sums of subarrays are equal to the target sum. If subarrays that satisfy this condition are found, it prints their starting and ending indices. Otherwise, it prints the message “No subarray found”.

(2) The subprogram `sSum2point` uses two pointers: starting and ending indices that represent a subarray. In the initialization step, the starting and ending indices start from 1, and the current sum is the first element of the array.

(i) If the current sum of the subarray is greater than the target sum and the starting index is smaller than the ending index, removes starting elements from the current sum and moves the starting index rightward until it becomes less than or equal to the target sum.

(ii) If the current sum is equal to the target sum, it outputs the two pointers.

(iii) The next element of the current subarray is added to the current sum, and the ending index moves rightward by 1. The subprogram stops after the ending index exceeds the last index of the array. Otherwise, it goes to step (i).

(iv) If the target sum is not found in any subarray, it prints the message “No subarray found”.

(3) Index of the array starts at 1.

(4) The subprogram `print( )` displays the input parameter in a new line.

(5) Table 1 shows the variables used in the program.

Table 1: Variables used in the program

| Variable | Description |
|---|---|
| `N` | The size of the input array. `N` is not less than 1. |
| `S` | The target sum |
| `A[]` | The input array. The values shown in Figure 1 are set to the array elements by the program. |
| `start` | The starting index of the current subarray |
| `end` | The ending index of the current subarray |
| `sum` | The current sum of the current subarray |

[Program]

```
GLOBAL: INT: N ← 8, S ← 14
GLOBAL: INT: A[N] ← {6, 3, 2, 5, 1, 1, 7, 3}

SUBPROGRAM: SSumNaive() {
   INT: sum, start, end, found ← 0
   FOR (start ← 1; start ≤ N; start ← start + 1) {
      sum ← 0;
      FOR (end ← $\boxed{\text{A}}$; sum ≤ S and end ≤ N; end ← end + 1) {
         sum ← sum + A[end];    /* α */
         IF (S = sum) {
            print(start + ", " + end);
            found ← 1;
         }
      }
   }
   IF ($\boxed{\text{B}}$) {
      print("No subarray found");
   }
}
```

```
SUBPROGRAM: SSum2point() {
   INT: sum ← $\boxed{C}$, start ← 1, end, found ← 0
   FOR (end ← 1; end ≤ N; end ← end + 1) {
      WHILE (S < sum and start < end) {
         sum ← sum - A[start];     /* β */
         start ← start + 1;
      }
      IF (S = sum) {
         print(start + ", " + end);
         found ← 1;
      }
      IF (end < N) {
         sum ← sum + A[$\boxed{D}$];
      }
   }
   IF ($\boxed{B}$) {
      print("No subarray found");
   }
}
```

## Subquestion 1

From the answer groups below, select the correct answer to be inserted in each blank $\boxed{\phantom{xxxxx}}$ in the above program.

Answer group for A and D

| | |
|---|---|
| a) `0` | b) `1` |
| c) `end` | d) `end + 1` |
| e) `start` | f) `start + 1` |

Answer group for B

| | |
|---|---|
| a) `found = 0` | b) `found = 1` |
| c) `found = S` | d) `found > 1` |

Answer group for C

| | |
|---|---|
| a) `0` | b) `A[1]` |
| c) `A[1] + A[2]` | d) `S` |

## Subquestion 2

From the answer groups below, select the correct answer to be inserted in each blank $\boxed{\phantom{XXXX}}$ in the following description.

When subprogram `sSumNaive` is called, the line `/* α */` is executed $\boxed{\text{E}}$ times.
When subprogram `sSum2point` is called, the line `/* β */` is executed $\boxed{\text{F}}$ times.

Answer group for E

| | |
|---|---|
| a) 8 | b) 28 |
| c) 29 | d) 30 |
| e) 36 | f) 56 |

Answer group for F

| | |
|---|---|
| a) 1 | b) 2 |
| c) 3 | d) 4 |
| e) 5 | f) 6 |
| g) 7 | h) 8 |

## Subquestion 3

From the answer group below, select the correct combination of the results of execution of subprograms `sSumNaive` and `sSum2point` when array `A[]` contains zero or a negative value. Here, "O" indicates that the result of execution is always correct, and "X" indicates that the result is incorrect in some cases.

Answer group

| | Subprogram `sSumNaive` | | Subprogram `sSum2point` | |
|---|---|---|---|---|
| | `A[]` contains zero value | `A[]` contains negative value | `A[]` contains zero value | `A[]` contains negative value |
| a) | O | O | O | O |
| b) | O | O | O | X |
| c) | O | O | X | X |
| d) | O | X | O | X |
| e) | O | X | X | X |
| f) | X | X | X | X |

Concerning questions **Q7** and **Q8**, **select one** of the two questions.
Then, mark the Ⓢ in the selection area on the answer sheet, and answer the question.
If two questions are selected, only the first question will be graded.

**Q7.** Read the following description of a C program and the program itself, and then answer Subquestions 1 through 3.

In mathematics, a perfect number is a positive integer whose sum of divisors is double of the integer itself. For example, 6 is a perfect number because the sum of its divisors (1, 2, 3 and 6) is 12, which is the double of 6.

[Program Description]
The program outputs perfect numbers between 1 and 100 to the standard output.
The function `divSum` calculates the sum of divisors for integer `num`.

[Program]
```
#include <stdio.h>

int divSum(int num) {
   int result = 0;
   int i;
   for (i = 1; i * i <= num; i++) {
      if (num % i == 0) {
         result += [   A   ];
         if (i != (num / i)) {
            result += [   B   ];
         }
      }
   }
   return result;
}

/* α */
int main() {
   int i;    /* β */
   printf("Perfect numbers that lie between 1 and 100:");
   for (i = 1; i <= 100; i++) {
      if (divSum(i) == 2 * i) {
         printf(" %d", i);
      }
   }
   printf("\n");
   /* γ */
   return 0;
}
```

**Subquestion 1**

From the answer group below, select the correct answer to be inserted in each blank $\boxed{\phantom{xxxxxx}}$ in the above program.

Answer group for A and B

| | |
|---|---|
| a) `i` | b) `num` |
| c) `num % i` | d) `num / i` |
| e) `result + i` | f) `result + i % num` |
| g) `result + i / num` | |

**Subquestion 2**

From the answer groups below, select the correct answer to be inserted in each blank $\boxed{\phantom{xxxxxx}}$ in the following description.

The concept of the perfect number can be extended to the (m, k)-perfect number. An integer `num` is an (m, k)-perfect number when the equality below is true, where `m` and `k` are positive integers.

$$\underbrace{\texttt{divSum(divSum(...divSum(num)...))}}_{\text{Applied m times}} \texttt{ = k * num}$$

For example, 16 is a (2, 2)-perfect number because `divSum(16) = 1 + 2 + 4 + 8 + 16 = 31` and `divSum(31) = 1 + 31 = 32`. Therefore, the equality `divSum(divSum(16)) = 2 * 16` is true.

In order to output (m, k)-perfect numbers as well as perfect numbers, the above program is modified as follows:

(1) Replace the line marked `/* α */` with the new function `divSumM`.
It applies `divSum` `m` times for a positive integer `num`.

```
int divSumM(int m, int num) {
   if (m > 1) {
      return $\boxed{\text{C}}$;
   } else {
      return divSum(num);  /* δ */ }
}
```

(2) Replace the line marked `/* β */` with the following statement. This adds declarations of variables `m` and `k`.

```
int i, m, k;
```

(3) Replace the line marked `/* γ */` with the following statements that calculate (m, k)-perfect numbers.

```
printf("Enter two positive integers: ");
scanf("%d %d", [    D    ]);
printf("(%d, %d)-perfect numbers that lie between 1 and 100:",
       m, k);
for (i = 1; i <= 100; i++) {
   if (divSumM(m, i) == k * i) {
      printf(" %d", i);
   }
}
printf("\n");
```

From the answer groups below, select the correct answer to be inserted in each blank [          ] in the above description.

Answer group for C

| | |
|---|---|
| a) `divSum(m)` | b) `divSum(m - 1)` |
| c) `divSum(num)` | d) `divSum(num - 1)` |
| e) `divSumM(m - 1, divSum(num))` | f) `divSumM(m - 1, divSum(num - 1))` |
| g) `divSumM(m - 1, num)` | h) `divSumM(m - 1, num - 1)` |

Answer group for D

| | | |
|---|---|---|
| a) `m, k` | b) `*m, *k` | c) `*&m, *&k` |
| d) `&m, &k` | e) `&*m, &*k` | |

## Subquestion 3

From the answer group below, select the correct answer to be inserted in the blank $\boxed{\phantom{xxxxxx}}$ in the following description.

The modified program is executed. The program produces the following output:

```
Perfect numbers that lie between 1 and 100: 6 28
Enter two positive integers: 3 6
(3, 6)-perfect numbers that lie between 1 and 100: 98
```

In this execution, the line marked `/* δ */` is executed $\boxed{\text{ E }}$ times.

Answer group

| | |
|---|---|
| a) 100 | b) 200 |
| c) 300 | d) 400 |
| e) 600 | f) 700 |

**Q8.** Read the following description of Java programs and the programs themselves, and then answer Subquestions 1 and 2.

The Java programs represent the POSIX-style tree structure of a file system consisting of nodes. Each node represents a file or directory. Directories may contain other files and directories, creating the tree structure. The root node is a special node representing the topmost directory. All other files and directories are organized under the root node. The programs are also capable of searching nodes with matching conditions.

(1) The `Node` class represents a node that is either a file or a directory. The class contains the following attributes:

(i) `name`: The name of this `Node`, e.g., "`MyLog.log`". The name must not contain any `'/'` (slash) characters.

(ii) `extension`: If this `Node` represents a file, it must have the extension in the node name. The extension starts with a `'.'` (dot) character and indicates the type of this file, such as "`.txt`" for text files. If there is more than one `'.'` in the name, the last `'.'` designates the extension. If this `Node` represents a directory, this attribute is `null`.

(iii) `fullPath`: The absolute path of this `Node`. Node names are separated by `'/'`, e.g., "`/home/user1/Desktop/note.txt`".

(iv) `parent`: The parent directory containing this `Node`. All nodes must have their parents, except for the root node.

(v) `children`: A list of nodes contained in this `Node`. Node names contained in the same directory node must be unique. If this `Node` is a file, this attribute is `null`.

(vi) The `ROOT` class (static) field represents the root node.

(2) The `ICondition` interface represents conditions to search for nodes. The `isSatisfied` method returns `true` if the specified `Node` meets the conditions, or `false` otherwise.

(3) The `NameCondition` class implements the `ICondition` interface.

(i) The `name` attribute is the node name that the caller wants to find.

(ii) The `isSatisfied` method returns `true` if the name of the specified `Node` is equal to the value of the `name` attribute, or `false` otherwise.

(4) The `Search` class contains a static method named `searchByName` that searches all nodes under the specified node to find nodes the names of which are equal to the `name` parameter.

(5) It is assumed that all constructers and methods are called with the correct parameters.

The following are string manipulation methods of the `String` class used in this question.

(1) `String substring(int beginIndex)`

Returns a string that is a substring of this string. The substring begins with the character at the index specified by `beginIndex` (inclusive) and extends to the end of this string.

(2) `int indexOf(int ch)`

Returns the index of the first occurrence of the character specified by `ch`, or -1 if the character does not occur.

(3) `int lastIndexOf(int ch)`

Returns the index of the last occurrence of the character specified by `ch`, or -1 if the character does not occur.

The following output is generated when the `main` method of the `Search` class is executed.

```
Some nodes info:
 "/"
 "/var/log/program.home.log" .log
 "/home/user1/document/note.txt" .txt
Result of searching nodes:
 ["/home/user1/document/note.txt" .txt]
```

[Program 1]

```
import java.util.ArrayList;
import java.util.List;

public class Node {
   public static final Node ROOT = new Node("", null, true);

   private final String name;
   private final Node parent;
   private final String extension;
   private final List<Node> children;
   private final String fullPath;

   private Node(String name, Node parent, boolean directory) {
      this.name = name;
      this.parent = parent;
      if (directory) {
         extension = null;
         children = new ArrayList<>();
      } else {
         extension = name.substring([   A   ]);
         children = null;
      }
      if (parent == null || parent == ROOT) {
         fullPath = "/" + name;
      } else {
         fullPath = [   B   ] + "/" + name;
      }
   }
```

```
    public static Node create(String name, Node parent,
                              boolean directory) {
       Node node = new Node(name, parent, directory);
       parent.children.add(node);
       return node;
    }

    @Override
    public String toString() {
       return "\"" + fullPath + "\""
              + (extension == null ? "" : " " + extension);
    }

    public String getName() { return name; }

    public String getExtension() { return extension; }

    public String getFullPath() { return fullPath; }

    public Node getParent() { return parent; }

    public boolean isDirectory() { return children != null; }

    public List<Node> getChildren() {
       return isDirectory() ? new ArrayList<>(children) : null;
    }
 }
```

[Program 2]

```
 public interface ICondition {
    boolean isSatisfied(Node node);
 }
```

[Program 3]

```
 public class NameCondition implements ICondition {
    private final String name;

    public NameCondition(String name) {
       this.name = name;
    }

    public String getName() { return name; }

    @Override
    public boolean isSatisfied(Node node) {
        return node.getName().equals(getName());
    }
 }
```

[Program 4]

```
import java.util.ArrayList;
import java.util.List;

public class Search {
   private static List<Node> searchList(Node root,
                                        ICondition condition) {
      List<Node> result = new ArrayList<>();
      doSearchList(root, condition, result);
      return result;
   }

   private static void doSearchList(Node current, ICondition condition,
                                    List<Node> result) {
      if (condition.isSatisfied(current)) {
         result.add(current);
      }
      if ([  C  ]) {
         for (Node child : current.getChildren()) {
            doSearchList(child, condition, result);
         }
      }
   }

   public static List<Node> searchByName(Node root, String name) {
      ICondition condition = [  D  ];
      return [  E  ];
   }

   public static void main(String[] args) {
      Node home = Node.create("home", Node.ROOT, true);
      Node user1 = Node.create("user1", home, true);
      Node document = Node.create("document", user1, true);
      Node note = Node.create("note.txt", document, false);
      Node var = Node.create("var", Node.ROOT, true);
      Node tmp = Node.create("tmp", var, true);
      Node log = Node.create("log", var, true);
      Node logfile = Node.create("program.home.log", log, false);

      System.out.printf("Some nodes info:%n %s%n %s%n %s%n%n",
            Node.ROOT, logfile, note);
      System.out.printf("Result of searching nodes:%n %s%n",
            searchByName(Node.ROOT, "note.txt"));
   }
}
```

### Subquestion 1

From the answer groups below, select the correct answer to be inserted in each blank $\boxed{\qquad\qquad}$ in Program 1 and Program 4.

Answer group for A

- a) `name.indexOf('.')`
- b) `name.indexOf('.') + 1`
- c) `name.indexOf('.') - 1`
- d) `name.lastIndexOf('.')`
- e) `name.lastIndexOf('.') + 1`
- f) `name.lastIndexOf('.') - 1`

Answer group for B

- a) `"/" + parent.fullPath`
- b) `"/" + parent.name`
- c) `fullPath`
- d) `name`
- e) `parent.fullPath`
- f) `parent.name`

Answer group for C

- a) `current == Node.ROOT`
- b) `current.getChildren() == null`
- c) `current.getChildren().size() > 0`
- d) `current.isDirectory()`
- e) `result == null`
- f) `result.size() == 0`

Answer group for D

- a) `ICondition(name)`
- b) `NameCondition(name)`
- c) `new ICondition(name)`
- d) `new NameCondition(name)`

Answer group for E

- a) `doSearchList(root, condition, new ArrayList<>())`
- b) `doSearchList(root.getChildren(), condition, new ArrayList<>())`
- c) `doSearchList(root.getParent(), condition, new ArrayList<>())`
- d) `searchList(root, condition)`
- e) `searchList(root.getChildren(), condition)`
- f) `searchList(root.getParent(), condition)`

**Subquestion 2**

From the answer groups below, select the correct answer to be inserted into each blank $\boxed{\phantom{xxxxx}}$ in Program 5.

Typically, users may want to find files or directories whose names partially match the specified string rather than exact names. For example, when searching for "`home`" with the same `Node` instances created in Program 4, only two nodes, file `program.home.log` and directory `home`, match the partial name condition because their names contain the string "`home`". Program 5 is the implementation of a new class named `PartialNameCondition` for the partial name matching condition.

[Program 5]

```
public class PartialNameCondition extends NameCondition {
   public PartialNameCondition(String name) {
      [   F   ];
   }

   @Override
   public boolean isSatisfied(Node node) {
      return [   G   ];
   }
}
```

Answer group for F

a) `return`
b) `super()`
c) `super(name)`
d) `this(name)`
e) `this.name = name`

Answer group for G

a) `getFullPath().contains(node.getFullPath())`
b) `getName().contains(node.getName())`
c) `node.getFullPath().contains(getFullPath())`
d) `node.getFullPath().contains(getName())`
e) `node.getName().contains(getFullPath())`
f) `node.getName().contains(getName())`