0. Get Ready

Background: Welcome one and all! It's time to dive in to part two! Let's get this party started!

A Reminder About Contest User IDs:

The challenges described within use CC##### as a placeholder to describe instances where you should insert your contest user ID. Each country and geographic region participating in the contest will have their own unique prefix and each contestant will have their own unique numeric identifier. During this contest, wherever you see "CC#####", please change this to your specific country code and user ID number or the challenges will not work as designed.

For example: If a challenge tells you to change CC##### to your user ID, and you are participating in the China contest with an ID of CN99999, replace CC##### with CN99999.

The list of user ID prefixes for each country's particular user ID format are as follows:

- BeNeLux: BE#####
- Brazil: BR#####
- Central & Eastern Europe: HU#####
- China: CN#####
- DACH: ZC#####
- France: FR#####
- India: IN#####
- Japan: JP#####
- Mexico: MX#####
- Spain: ES#####
- Spanish South America: AR#####
- United Kingdom & Ireland: GB#####
- United States, Canada, & Puerto Rico: US#####
- Mainframe Contest Learning System: AU#####

Your challenge: Prepare your environment for part two.

Issue the following command from the ISPF Command Shell command line, option 6 from the ISPF Primary Option Menu:
SUBMIT 'Y2015.PUBLIC.JCL(P2)'

Don't forget the quotes! This job will take a minute to complete, so feel free to space out for a moment - this is the last chance your brain will have to relax as long as you're working on Part 2. If you have any questions about the status of your submission, shoot us an email at zskills@us.ibm.com.

Now go see what goodies we've placed for you. Go to the DSLIST panel (ISPF option 3.4) and enter your user ID in the Dsname Level field. Hit ENTER, and you'll see a list of several data sets (including your P1.OUTPUT data set). Got 'em? Good.

You'll notice that you now have a data set called 'CC#####.P2.OUTPUT' (where CC##### is your assigned user ID). This data set is very important -- it will eventually contain all of your work to be graded for Part 2. From here on out, we will refer to this data set as OUTPUT or P2.OUTPUT. You'll need to keep its full name in mind to complete many of the Part 2 challenges successfully.

**Note on starting over:** If you mess up any of your CC##### data set members during Part 2, you'll be able to find read-only copies for almost all of them in the Y2015.** data sets. For example, if you have a member called CC#####.DATA(ZSYSTEMS) that you'd like to replace, you can find a fresh copy at Y2015.DATA(ZSYSTEMS). To make a new copy, just create a new member with the name you want, then run this command from the command line inside the blank member:

```
COPY 'Y2015.qualifier(membename)'
```

Voila! Alternately, you can delete the contents of an existing member by using the line command D#### (where #### is the number of lines you want to delete - using D9999 will quickly clear out all but the longest members), and then issuing the copy command from the command line once the member is blank. Throughout Part 2, whenever you need to access a data set that does NOT start with your CC##### user ID, such as the Y2015.** data sets, always open the data set using Browse, rather than Edit.

Now you know how to quickly create and copy members, and knowing is half the battle. You've got everything you need to become a bonafide mainframe master, right? Yes! Let's get to it.

Back to Top (Collapse All)

1. ISPF and the ISPF Editor

**Background:** When you first log into z/OS using a 3270 Emulator, the z/OS Time Sharing Option (TSO) is started. TSO is a command line interface, or CLI, and is not very user friendly.

We've configured the z/OS system to automatically start the Interactive System Productivity Facility (ISPF). ISPF includes a full screen editor to edit source code and view data. You can also create and submit a Job Control Language (JCL) program to execute work and view the results.

**Your challenge:** Learn ISPF Panel Navigation & ISPF Editor

The ISPF Primary Option Menu is a tree like structure where entering one option can take you to a deeper view with additional navigation options. For example, selecting (Option
3) **Utilities** will present another list of options like *(Option 4) Dslist.*

The Dslist panel is useful for displaying a list of data set names from a pattern matching a given argument.

It’s possible to jump directly to a desired panel. For example, entering **3.4** from the ISPF Primary Option Menu will jump you directly to the Dslist panel.

Also, you can jump from any panel to another by prefixing the menu path with an equals sign (=). For example, entering =**3.4** from anywhere in ISPF will navigate directly to the Dslist panel. The prefix = designates the ISPF Primary Option Menu as the starting panel, then the following **3.4** navigates to the target Data Set List Utility panel.

The 3.4 Data Set List Utility panel has two different locations where you can enter commands. A command line that runs the length of the panel, and a command column next to any data sets being displayed.

Example:

```
Menu  Options  View  Utilities  Compilers  Help
-------------------------------------------------------------------------------
DSLIST - Data Sets Matching US09999                                Row 1 of 10
Command ===>                                                  Scroll ===> PAGE
-------------------------------------------------------------------------------
Command - Enter "/" to select action                  Message           Volume
-------------------------------------------------------------------------------
   US09999                    *ALIAS
   US09999.DATA                VPWRKA
   US09999.JCL                  VPWRKB
   US09999.LOAD                 VPWRKC
   US09999.P1.OUTPUT            VPWRKJ
   US09999.P2.OUTPUT            VPWRKI
   US09999.SOURCE               VPWRKE
   US09999.STATUS               VPWRKB
   US09999.SOW1.ISPF.ISPPROF   VPWRKC
   US09999.SOW1.SPFL0G1.LIST   VPWRKF
******************************************************************************
```

The command column lets you issue an action against a specific data set name in the list.

Entering a forward slash (/) displays a list of actions and commands that can be issued.

The most common command column actions are:

- e - edit
- b - browse

Let’s take a moment to illustrate the differences between Sequential and Partitioned Data Sets.

- A Sequential data set will display data as a result of edit or browse action.
- A Partitioned Data Set is used to group related members of data.
- A Partitioned Data Set will display member names as a result of edit or browse action.

Enter **s** to the left of any member name to select the member and display the member data.

Now, for the real challenge in this section!
From the DSLIST view of all datasets beginning with your ID, enter **e** in the command column next to the data set **CC#####.DATA**.
Next, enter \texttt{s} to left of the member named \texttt{EDITOR}.

The \texttt{CC####.DATA(EDITOR)} member is opened in the ISPF editor and it contains information and instructions necessary to complete the rest of this challenge. Follow the instructions you see in the \texttt{EDITOR} member now.

\textbf{Note on starting over:} If you need to start over from the beginning, type in the following two commands and then press enter:

- In the command column for line 1, enter \texttt{d999}. You will be overtyping the \texttt{000001}.
- In the primary command line, enter \texttt{copy 'y2015.data(editor)'}

Once you’ve successfully completed this challenge, you will have a new member in your \texttt{P2.OUTPUT} dataset called \texttt{$001$}. It will contain two lines of text from the \texttt{EDITOR} member.

\textbf{2. Data Representation}

\textbf{Background:} Fundamentals of understanding digital data includes knowledge of binary, hexadecimal, EBCDIC, and ASCII data encoding formats. Being able to manipulate data by encoding it into different formats will give you a deeper understanding of computing. It’s particularly important to understand data conversion and representation in z/OS as z/OS is a primarily EBCDIC operating system, whereas Windows and Linux primarily use ASCII.

There is a famous saying often quoted by computer scientists:

\textit{"There are 10 types of people in the world: those who understand binary, and those who don’t.

If this quote seems confusing, then this challenge can help you understand it.}

If you are already familiar with binary, EBCDIC, ASCII and hexadecimal, feel free to skip down \textbf{"Your challenge:"} below. If these concepts are vague to you, read on.

All data in a computer (characters, strings, machine instructions, memory addresses, etc.) are a sequence of bits (binary digits) where each individual bit is either a 1 or 0. We call this binary. Since it’s very difficult for a human to read binary, often data is represented in hexadecimal. Hexadecimal is simply a set of 16 unique digits that map to a group of 4 binary digits.

Here is a table of decimal, binary, and hexadecimal equivalents:

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
</tbody>
</table>
More terminology and equivalents:

- bit = 1 binary digit (0 or 1)
- byte = 8 bits
- character = 8 bits = 1 byte (generally)
- string = sequence of characters, like a word or sentence

With a byte, it's possible to have 256 unique representations. There is a simple formula for determining how many unique values that can be represented given a number of bits: \( 2^x \) where \( x \) is the number of bits being used. Given that hexadecimal uses 4 bits, our equation \( 2^4=16 \) holds. If we have a byte, \( 2^8=256 \).

There are a number of different character encoding schemes. The two that have been around since the epoch are ASCII and EBCDIC. Unicode is a third standard that is very popular, but we will not be discussing it here.

In ASCII and EBCDIC, characters are encoded as a single byte. Since hexadecimal only gives us representation of 4 bits, we need to use two hexadecimal digits to represent a byte, or a single ASCII/EBCDIC character. Here is a partial table of decimal, byte, and hexadecimal equivalents, you can see how each set of four bits map to a single hex digit:

<table>
<thead>
<tr>
<th>Dec</th>
<th>Byte</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0000 0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0000 0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0000 0011</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>16</td>
<td>0001 0000</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>26</td>
<td>0001 1010</td>
<td>1A</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>188</td>
<td>1011 1100</td>
<td>BC</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>221</td>
<td>1101 1101</td>
<td>DD</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>255</td>
<td>1111 1111</td>
<td>FF</td>
</tr>
</tbody>
</table>

Additional information on character encoding can be found at the following links:

- **Wikipedia: Computer number format**
  1. Table 1: Binary to Octal
  2. Table 2: Number of values for a bit string
  3. Table 3: Comparison of Values in Different Bases
- **Wikipedia: ASCII**
  1. ASCII control code chart
  2. ASCII printable code chart
- **Wikipedia: EBCDIC**
  1. EBCDIC chart
- **Simotime: ASCII to EBCDIC**
  1. Excellent description that also includes much more at this topic
Take note how the ASCII and EBCDIC characters have different decimal values for the same characters. For example:

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Bin</th>
<th>EBCDIC</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>50</td>
<td>0101 0000</td>
<td>&amp;</td>
<td>P</td>
</tr>
</tbody>
</table>

The reason ASCII and EBCDIC are different is because the standards both grew out of different organizations at the same time. It's similar to how one can perform calculus using Newtonian or Leibniz notation! Two different ways to represent the same idea.

**Your challenge:** You will be changing data between ASCII and EBCDIC encoding using hexadecimal value representation in the ISPF editor.

You now will have the opportunity to visually observe the differences between ASCII and EBCDIC.

Using the ISPF Editor, open the data set member CC##.DATA(EBCDIC) in edit mode. Observe the following things:

- line 2 - lower case english alphabet
- line 3 - upper case english alphabet
- line 6 - strange looking
- line 7 - strange looking
- line 10 - decimal numbers
- line 13 - nothing that is printable on screen

Enter the following command in the ISPF primary command line:

```
source ascii
```

Observe the following things:

- line 2 - strange looking (stored in EBCDIC)
- line 3 - strange looking (stored in EBCDIC)
- line 6 - lower case english alphabet (stored in ASCII)
- line 7 - upper case english alphabet (stored in ASCII)
- line 10 - nothing that is printable on screen
- line 13 - decimal number (stored in ASCII)

The reason for all the @ signs when the editor is set to **source ascii** is because an EBCDIC space, x'40' is interpreted by ASCII as @.

Enter the following command in the ISPF primary command line:

```
reset
```

This results in EBCDIC (the default) viewing of the data.

You just observed the difference between ASCII and EBCDIC character representation.

Enter the following command in the ISPF primary command line:

```
hex on
```
Observe that two additional lines are now inserted after every line of text. These additional lines provide the hex value for each character. For example, line 000002 and the additional lines now contain:

abcd
448888
001234
|||d is hexadecimal 84 ... x'84' EBCDIC
|||c is hexadecimal 83 ... x'83' EBCDIC
|b is hexadecimal 82 ... x'82' EBCDIC
a is hexadecimal 81 ... x'81' EBCDIC

A space in EBCDIC is hex 40.

Enter the following command in the ISPF primary command line to hide the hex values:

    hex off

Press F3 to exit member EBCDIC now.

For your reference: CC####.DATA(TABLE) contains an ASCII to EBCDIC conversion table.

Open CC####.DATA(WHOAMI) for editing. This member contains five questions with five answers on lines 8, 17, 24, 31, and 39. However, the 5 answers are not currently human readable. The remainder of this challenge is to make the answers human readable by altering the hex values for each character in the answers.

Enter the following command in the line command area for line number 0000008 to turn on hex display for that line only:

    hx

The hx line command will display line 8 hex values below the line as follows:

EDIT US09997.DATA(WHOAMI)
Command ===>
****** ******************************************* Top of Data *******************************************
000001 #1-- Who am I --------------------------------------
000002 I am a self taught mathematician who is best known as an author
000003 of didactic works on mathematics, such as Philosophy and Fun
000004 of Algebra, and as the wife of fellow mathematician. Some have
000005 said that my contributions to math in the 1800's is at the
000006 heart of the digital revolution.
000007
000008 I am )Øµ {µØ°Øµµ {°°°Ø
C4894D89A4CA898AA4C9998444444444444444444444444444444444444444444444
9014000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

Note that the 2nd row of hex values have a bunch of zeros! That's not right. Here are the correct values for both rows. Over type the values on the 2nd row to get the right answer to display:

C4894D89A4CA898AA4C9998
Turn off the hex display for line 8 by entering the `hx` command again in the line command area. Then enter the `hx` line command on line number 000017.

Again, you'll need to set the hex digits for this line to the following:

```
C4894D994D9AA89
901401650762353
```

Turn off hex for line 17, then turn on hex display for line 24. The hex values you'll need here are:

```
C4894C89A4D89889
90140255038647265
```

Turn off hex for line 24, then turn it on for line 31. The hex values you'll need here are:

```
C4894D894E8999A99
90140255038647265
```

Turn off hex for line 31, then turn it on for line 39. The hex values you'll need here are:

```
C4894E89A4C898
90140595303596
```

Turn off hex for line number 39.

Now you have the answers to the questions and only need to perform one final step to complete this challenge.

Enter the following line command on line number 1:

```
c99
```

Then enter the following command in the ISPF primary command line:

```
replace p2.output($002)
```

Press F3 to save and exit. You can view the `CC####.P2.OUTPUT($002)` member to ensure you've copied the data correctly.

**Background:** z/OS and the mainframe are at the heart of many large organizations. Mainframe computing is so critical to today's economy that workers skilled in it's usage are in high demand. As a result, those skilled workers can demand a high pay!
In this challenge, you will learn some productivity tools in the ISPF editor. Did you know that it is possible to overlay a line or a block of lines with data from another line in just a few keystrokes? Let's learn how!

**Your challenge**: Edit a PDS member using the ISPF editor copy and overlay line commands.

We discussed overtyping line numbers with commands in the previous challenge. Feel free to review that section for more information on how to overtype the line number with a command.

Open the member CC#####.DATA(ZJOBS) for editing in the ISPF editor. Then perform the following actions, in order. Note that each of these commands are entered by overtyping the line number (in the line command area).

1. overtype 000001 with c
2. overtype 000006 with o
3. press enter
4. overtype 000002 with c
5. overtype 000006 with o
6. press enter
7. overtype 000002 with c
8. overtype 000018 with o
9. press enter
10. overtype 000002 with c
11. overtype 000034 with o
12. press enter
13. overtype 000003 with c
14. overtype 000008 with oo
15. overtype 000015 with oo
16. press enter
17. overtype 000003 with c
18. overtype 000020 with oo
19. overtype 000032 with oo
20. press enter
21. overtype 000001 with d5
22. press enter

After performing the actions, the first line in zjobs should read:

**Little systems, little money earning potential**

To complete the challenge, enter c999 on the line 000001 line command and enter replace p2.output($003) in the primary command line. Then press F3 to exit.

You can view your output in the P2.OUTPUT($003) member.

**4. Complete z Picture**

**Background**: Since editing text is so important, we have set up another challenge for you in the ISPF editor. There are so many times you will need to be able to edit text effectively, editing source code, altering data, generating documentation, etc. and we wanted to show you another nifty trick in the ISPF editor.
Surely you are already familiar with the find and replace all functionality in modern editors. Well ISPF has it too: The primary command line CHANGE ALL command!

Your challenge: Become more familiar with the ISPF Editor and use the change all, c all, primary command to reveal a hidden picture.

Open the member CC#####.DATA(ZSYSTEMS) for editing.

```plaintext
DSLIST - Data Sets Matching US09997
Command ===> 
Command - Enter "/" to select action

US09997
e US09997.DATA

EDIT US09997.DATA
Command ===> 
Name Prompt
----------- -----------
------- EBCDIC
------- EDITOR
------- MSG
------- TABLE
------- WHOAMI
------- ZJOBS
s ZSYSTEMS

Issue the primary command hex on to turn hexadecimal view on for every line.

Command ===> hex on

Observe all the hex 00 values, for example characters 13 through 29 on line one.

Turn hexadecimal view off by issuing the primary command hex off.

Command ===> hex off

To begin to reveal the hidden picture, change all occurrences of hexadecimal 00 to z by issuing the following primary command (don't forget the single quotes around '00'):

```
    c x'00' z all
```

Next, change all occurrences of # to Mainframe by issuing the following primary command:

```
    c # Mainframe all
```

To complete the challenge, save the current contents of the editor to data set member...
CC#####.P2.OUTPUT($004) by entering C999 in the line command area of line number 000001, and then entering the following primary command:

```
replace p2.output($004)
```

If you need to start over with the original contents of the ZSYSTEMS member, evaluate the previous challenges in part two and copy from 'Y2015.DATA(ZSYSTEMS)'. Alternatively, you can issue the primary command CANCEL to exit the ISPF editor without saving, then reopen the member and try again.

Back to Top (Collapse All)

5. What is JCL?

**Background:** Mainframe computers generally run on backend processes, and it's important to queue work in such a way that it will run exactly when you want it to and how you want it to. For example, nightly inventory processing for a large retail company needs to run at specific times of the day and only when certain data is ready for processing.

Computers generally perform work in one of two different ways: interactively in the foreground or non-interactively in the background. This background processing is also known as "batch processing." All of your work from Part 1 was considered foreground work. You can do a lot of stuff in the foreground, but when a program takes a lot of time to process, you can't do anything else with your interactive session while it's running. The solution is to submit long-running programs to process in the background. Then you can continue to work interactively and z/OS will let you know when your long-running program has completed.

In mainframe-speak, a "job" is a collection of different programs that work together and are ran in the background. In order to cause work to run in batch (in other words, to "submit a job"), you need to instruct z/OS through JCL (Job Control Language). You "submit" JCL to JES (Job Entry Subsystem), the system allocates the necessary resources for your job, then executes the work when the resources are available. Understanding JCL is a very important skill to have, so you'll be seeing more of it throughout the rest of Parts 2 and 3. Plus, IBM customers are always impressed by applicants who know something about JCL, so you can brag on your resume! Woot!

JCL is very rich, and it's very different from any other programming language. Most systems and applications programmers will find a piece of existing JCL code that does something very close to what they want to do, and then make a few little changes to it so that it fits their needs. You're going to do the same thing in this challenge and debug a faulty JCL program.

JCL:

- is relatively simple and an important piece of z/OS.
- statements are read, interpreted, and executed by the z/OS JES.
- instructs z/OS to execute at least one program.
- can be started in the foreground or submitted to the background
- statements begin with // starting in column 1.

JCL has 3 major JCL statement types:

1. JOB: gives the system a name for the work to be done.
2. EXEC: gives the system a program name to find, load, and execute.
3. DD: associates an internal program file name with an operating system controlled
resource, such as a data set name.

Example:

```
//SOMENAME JOB
// EXEC PGM=SOMEPROG
//SOMEFILE DD operands-identifying-system-physical-resource
```

Where:

- SOMENAME is a user selected name for work to be done by z/OS.
- SOMEPROG is the program name to be found, loaded, and executed.
- SOMEFILE is a file name in the program source code.

DD operands associate the SOMEFILE with system physical resource such as an MVS data set or unix file. The DD (Data Definition) statement is one of the primary benefits of JCL as it eliminates the need to change the program source code to access different resources. The same program can be used in different ways by altering the JCL used to execute it.

**Your challenge:** Locate and fix errors in a JCL program.

From any primary command line in ISPF, submit some JCL for execution by entering the following command:

```
tso submit jcl(sortjcl)
```

Well, that wasn’t too exciting... Let’s go see what happened!

To view the output from the job, switch to the SDSF, System Display and Search Facility, panel by entering the following ISPF primary command:

```
=sd
```

Set the SDSF filter to view only jobs owned by you by entering the following SDSF primary command, where CC##### is your ID:

```
owner CC#####
```

Next, set the SDSF filter to view all JCL jobs with any jobname (* is a wildcard used to match anything). Issue the following SDSF primary command:

```
prefix *
```

Next, open the panel to view the status of all job output. This panel will only display results as a result of filter processing. Enter the primary SDSF command:

```
st
```

From SDSF st (status) panel, locate the line that contains the SORTJCL and move your cursor to the NP column for that line (use the [TAB] key like a pro!). Type the character S in
the NP column for this line, then press Enter.

The output of the SORTJCL job you submitted earlier is now displayed.

**OOPS:** SORTJCL failed to process due to a JCL error that is identified at the bottom of the output. Observe the error is on JCL line 3 and also note how SDSF conveniently prints the contents of your JCL along with lines numbers.

**Hint:** JCL statements must ALWAYS BE IN UPPERCASE. Exceptions to this rule only apply when dealing with Unix files.

Let's fix this problem in the JCL source code. Enter `=3.4` in the SDSF command line to jump to the ISPF 3.4 panel.

Edit the CN#####.JCL(SORTJCL) member and correct the error within.

**Bonus hint:** Enter the ISPF editor primary command `hilite jcl` to turn on syntax highlighting. Now you're really looking like a z/OS pro!

Once you think you've found and corrected the error, you can submit the JCL code directly from the editor. In the ISPF editor primary command line enter:

```
sub ; =sd ; st
```

The provided command performs the following actions, in order:

1. Submit the modified JCL
2. Save the changes and jump to SDSF panel
3. Display the SDSF st (status) panel

A new SORTJCL output will be visible now. You can tell which one is the most recently ran job by evaluating the JOB##### number. The JOB##### number that is greater is the most recently ran job.

Enter `s` to the left of the new SORTJCL output. If you correctly identified and fixed the original problem, you will now notice a new problem in the output.

Observe the system message in the output:

```
CSV003I REQUESTED MODULE SORTPGM NOT FOUND
```

Go ahead and fix this problem now.

**Hint:** The system was unable to find a program named SORTPGM. "SORT" is the program name known by the system. Change the unknown program name to the known program name in your JCL, submit it, then view the output. Wash, rinse, and repeat this process until you see a return code (RC) of 00 in the job output. If you look closely at the output when getting RC 00, you'll notice that the JCL executed a second program that writes a record into P2.OUTPUT($005).

Challenge complete!

**Back to Top (Collapse All)**
**6. Common Business Oriented Language, COBOL**

**Background:** COBOL is a relatively simple programming language in which statements are written in a very natural language, making it easier to understand. COBOL is widely used in various industries to process critical data.

Decades of performance improvements to COBOL applications have enabled extreme levels of efficiency in data processing. Knowing how to program in COBOL makes it possible to understand critical business processing.

COBOL is quite an attention getter on a resume. Java and C/C++ are very common languages understood by many people, but COBOL is a programming language that is super-important, in high demand, and not currently in vogue. Let’s put our hipster hats on and learn some COBOL!

**Your challenge:** Decrypt a message using COBOL.

We used a COBOL program to encrypt a message. The source code for this program is located in CC#####.SOURCE(SCRAMBLE) and you’ll find the encrypted message in CC#####.DATA(MSG).

Browse view the data set member CC#####.DATA(MSG) to see encrypted message.

By making a minor change to the COBOL source code that encrypted the message, you will decrypt the message.

**README:**

The SCRAMBLE source code has values for STRING-IN and STRING-OUT. Swapping the values of STRING-IN and STRING-OUT and re-running the program will generate the unencrypted message from the encrypted message.

Edit CC#####.SOURCE(SCRAMBLE) and modify the source code using above information.

COBOL is a compiled language, which means that the source code needs to be passed through a compiler, generating an executable module as the output. We’ve conviently provided some JCL that:

1. Compiles your CC#####.SOURCE(SCRAMBLE) member.
2. Executes the newly compiled SCRAMBLE program

Edit CC#####.JCL(SCRAMBLE) and observe that:

1. The SYSIN DD resource is the encrypted message data set member.
2. The SYSOUT DD resource is your P2.OUTPUT($006) member. If it doesn’t exist when the job is submitted, z/OS will automatically create it!

Go ahead and submit the JCL. View the output in CC#####.P2.OUTPUT($006).

If there is a message in english then congratulations, you’ve successfully completed this challenge. Feel free to view the output of the job in SDSF or move on to the next challenge now.

Back to Top (Collapse All)
7. Guess the Address

**Background:** A significant amount of COBOL is at the heart of many large enterprise operations. Learning to read COBOL is an easy and valuable skill to acquire. In this challenge you’ll be exposed to more COBOL.

**Your challenge:** Read the source code for a COBOL program to determine an address number.

ID.SOURCE(GUESS) is a COBOL program that computes the same address number whenever executed. The program requires you to guess the correct address number, which is somewhere between 0 and 4095.

To compile the GUESS COBOL source code, from the ISPF primary command line, enter:

```
tso submit jcl(guess)
```

To execute the COBOL GUESS program interactively, jump to the TSO command shell (=6) and enter the following command:

```
guess
```

A routine (named "guess") allocates the input and output to the screen then executes the guess program.

You can waste your time trying to guess the number, but there's an easier way! Use the source, Luke!

View the GUESS COBOL source code to determine the correct address number. The source code is located in ID.SOURCE(GUESS). Don’t forget about the ISPF editor primary command **HILITE AUTO**.

Once you know what the correct number is, execute the guess routine via =6 and enter the correct number. If you’ve ‘guessed’ correctly, the GUESS COBOL program writes a special message to P2.OUTPUT($007). Feel free to take a look and then move on to the next challenge.

**An aside:** Base index displacement addressing is a common scheme used in assembler language programming. This knowledge might come in handy later...

---

8. 5 Major Programming Languages

**Background:** After learning one programming language, the next programming language is easier to learn. Each additional programming language becomes easier to learn as a result of the programming fundamentals previously learned and applied. However, the patterns and techniques used in procedural programming languages are different than those used in object oriented programming languages.

**Your challenge:** Read and modify a variety of procedural and object oriented programming languages.
The following major programming languages are used by large enterprises. Exposure to the syntax and usage of these languages will benefit you greatly.

- COBOL
- Assembler
- Java
- C
- C++

For each of these languages, you will be modifying the source code as instructed, compiling the source into an executable program, and running the program.

From the ISPF primary command line, enter:

```
tso submit jcl(pgmcomp)
```

Submitting this JCL compiles five separate sources into executable programs, one for each language in the list above.

Wait for JES to send you a message about the job ending, then enter the following command on the ISPF primary command line, replacing CC##### with your ID:

```
=sd ; prefix * ; owner your_ID ; st
```

This command sequence performs the following actions:

1. go to SDSF Primary Option Panel
2. change SDSF view filters to all jobnames that you own
3. display the status of all jobs owned by your ID

Locate the job associated with job name PGMCOMP and enter s in the command column to the left. The output should include the following:

<table>
<thead>
<tr>
<th>STEPNAME</th>
<th>PROCSTEP</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASM</td>
<td>C</td>
<td>00</td>
</tr>
<tr>
<td>ASM</td>
<td>L</td>
<td>00</td>
</tr>
<tr>
<td>COBOL</td>
<td>COBOL</td>
<td>00</td>
</tr>
<tr>
<td>COBOL</td>
<td>LKED</td>
<td>00</td>
</tr>
<tr>
<td>CPP</td>
<td>COMPILE</td>
<td>00</td>
</tr>
<tr>
<td>CPP</td>
<td>BIND</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>COMPILE</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>BIND</td>
<td>00</td>
</tr>
<tr>
<td>JAVA_COPY</td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>JAVA</td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

The RC column currently contains all 00 condition codes. This means each step of the JCL job completed with no errors and each program compiled successfully.

Enter the following command in the ISPF primary command line:

```
tso submit jcl(pgmrun)
```

Wait for JES to send you a message about the job ending, then enter the following ISPF
primary command, replacing CC##### with your ID:

=sd ; prefix * ; owner your_ID ; st

Again, SDSF lists your jobs, one of which is PGMRUN. Enter s next to the PGMRUN job to view its output.

<table>
<thead>
<tr>
<th>STEPNAME</th>
<th>PROCSTEP</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ASM</td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>-COBOL</td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>-CPP</td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>-C</td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>-JAVA</td>
<td>JAVA JVM</td>
<td>00</td>
</tr>
<tr>
<td>-COPY</td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

The PGMRUN JCL executes the five unique programs in batch (in the background) and collects all the output for each program. Each program’s output is written to your P2.OUTPUT($008) member.

Take a moment and browse each of the following members in your CC#####.SOURCE data set:

- PGMCOBOL - COBOL Hello World!
- PGMASM - Assembler Hello World!
- PGMJAVA - JAVA Hello World!
- PGMC - C Hello World!
- PGMCPP - C++ Hello World!

The following members in your CC#####.JCL data set contain the JCL to compile and execute each of the above programs:

- PGMCOMP - Compile each Hello World! program source
- PGMRUN - Execute each Hello World! program

Now things get interesting. Here’s another list of members in your SOURCE data set:

- ZC
- ZCOBOL
- ZASM
- ZJAVA

And, here’s a list of members in your CC#####.JCL data set:

- ZCOMP - Compile each program source
- ZRUN - Run each read program and write the output

Each program in the list above will read a line, modify 1 part of the line, then pass the line to another program. The next program will read the incoming line, modify a different part of the line, then pass the line to another program until the line is passed through all the programs.

The starting line is:

z## ---- processing memory, ---- 5.2GHz CPUs, ---- GB/sec I/O throughput
The order of the execution of the five programs is important, and is as follows:

1. The C program reads the starting line given above and modifies part of the line.
2. The COBOL program reads the C program line output and modifies part of the line.
3. The Assembler program reads the COBOL program line output and modifies part of the line.
4. The Java program reads the Assembler program line output and modifies part of the line.
5. **Note:** The C++ (ZCPP) program is not used in this challenge.

The following ISPF primary command will compile all the programs and create executable modules:

```
tso submit jcl(zcomp)
```

Use SDSF to check for successful compile return codes on all programs before attempting to run them. Again, be sure that CC#####.JCL(ZCOMP) completes successfully before continuing.

The following ISPF primary command will execute all the programs:

```
tso submit jcl(zrun)
```

Again, wait for JES to inform you the job completed, and ensure that SDSF reports for good return codes (RC 00) for each step in the job.

Browse (b) P2.OUTPUT($008) to see line modified by the programs. The member should contain the following line:

```
z12  3TB processing memory, 101 5.2GHz CPUs, 384 GB/sec I/O throughput
```

That's great, what a beast machine! However, we are on a whole new level of next-gen hardware, the z13! The line needs to read:

```
z13 10TB processing memory, 141 5.2GHz CPUs, 832 GB/sec I/O throughput
```

**Note:** You may notice some periods (.) in the output. This is okay - do not remove them!

The real challenge begins! Modify each program such that the above line describing the z13 is written to P2.OUTPUT($008). Don’t worry, this is easy! Edit the source code for each program, compile the newly modified programs (don’t forget to check for good return codes), then execute the programs. Repeat this process until you are successful.

Remember that you compile the programs with

```
tso submit jcl(zcomp)
```

And execute the programs with
tso submit jcl(zrun)

FYI: We've hidden some non-printable characters in the output string to discourage cheating in this challenge. If you notice some dots where there should be spaces, just disregard them!

Back to Top (Collapse All)

9. z/OS Operator Commands
Background: Just like any other operating system, z Operating System (z/OS) has an operator interface where you can issue commands to gather important information from and control the system. This challenge will give you the opportunity to become familiar with how to enter z/OS operator commands, capture the output, then write the output into a partitioned data set (PDS) member.

Your challenge: Execute a few z/OS Display operation commands, then print the output of the commands to a member in the P2.OUTPUT PDS.

From the ISPF primary command line, enter the following command:

```
=sd ; ulog
```

You will be taken to the ISPF user log (ulog) panel. This is a subset of the entire system log (syslog) where only messages generated in response to commands you enter are displayed.

On the command input line, enter the following operator commands, pressing Enter after each command:

```
/d iplinfo
/d m=cpu
/d parmlib
```

Next, you will need to issue operator commands to

1. Open the P2.OUTPUT($009) member for output.
2. Print the output in the ulog to the member.
3. Close the P2.OUTPUT($009) member.

On the command input line, enter the following operator commands, pressing Enter after each command:

```
print odsn p2.output($009)
print
print close
```

Now exit the SDSF user log and return to the ISPF data set browser by entering the following command:

```
=3.4
```

Browse (b) your P2.OUTPUT($009) PDS member. The same output you saw in SDSF should be saved there. The command output is truncated to 80 characters per line because the parameters of the PDS force it to be. This is okay!

If any of the command output appears incorrect, just start the challenge from the beginning and SDSF will overwrite your P2.OUTPUT($009) member with the new command output.

Back to Top (Collapse All)
10. Unix and z/OS: A Match Made in POK

**Background:** z/OS includes a POSIX compliant Unix, build into the operating system. The history of z/OS is very long and rich, and we won't bore you a ton of details here. However, it is very often useful to know how we got to this point, so we can have a better understanding of the system and help us to know what's possible in the future.

z/OS began it's existence way back in 1964. Back then it was called OS/360 and was the flagship operating system for the most important computing platform on the planet: System/360 (a.k.a the mainframe). A progression of updates to the OS occurred over many decades, which included many name changes. OS/360 begat MFT begat MVT begat OS/VS1&2 begat MVS begat OS/390 begat z/OS! There were a bunch of variations in addition to this list, but each subsequent new version always maintained backwards compatibility with the previous versions. This is why you can still find programs written in the 1970's running on mainframes today! In the early 1990's, a version of MVS was released called MVS/ESA Open Edition. The release included about 1 million new lines of code, which provided a fully POSIX compliant API shell, utilities, and an extended user interface. Unix System Services (USS) is what we call this facility today in z/OS, and it's very important.

Wikipedia has some really great articles on this topic, and we will leave it as an exercise for the reader to investigate further.

You might be forgiven to think of z/OS as two different operating systems combined into one. There's the older, MVS 'side' that contains TSO, ISPF, SDSF, and other things you've already experienced. And the newer, USS 'side' that is very much a Unix environment. This powerful combination of technology enables modern interfaces, to include web and mobile.

**Your challenge:** Become familiar with z/OS Unix System Services by entering commands, navigating the filesystem, and redirecting output from your commands to a Unix file. Finally, you will copy the Unix file to a P2.OUTPU T PDS member. You might have future challenges that assume you know how to enter a z/OS UNIX services shell command, so don't forget that you can come back here and reference this section.

From the ISPF primary command line, proceed to the z/OS Unix system services shell prompt by entering the following command:

```
tso omvs
```

After a few moments, you will be presented with a prompt that looks like `CC#####:/z /cc#####:`

Since you've already been introduced to some Linux commands in part one, you can try using some of them here. Input the following Unix shell commands, one at a time, pressing enter after each one:
So, you can see, POSIX compliant environments (Unix and Linux) are really very much alike and once you know one, you can get around pretty well in any other one.

To complete this challenge, you need to issue the `uname -aI` command and redirect the output to a file. Then, copy the file to your MVS P2.OUTPUT($010) PDS member.

Once you've completed the copy successfully, or want to back out to MVS and take a look at your P2.OUTPUT data set, issue the command:

```
exit
```

When the following message appears, simply press enter to end OMVS session and return to ISPF:

```
>>> FSUM2331 The session has ended. Press <Enter> to end OMVS.
```

**11. Put on your Poker Face**

**Background:** The mainframe supports many programming languages, including C, C++, Java, and COBOL. You may already have experience with one or more of these languages in a non-mainframe environment! The mechanics of creating and running a program on the mainframe are the same as on a PC: design the program, write the source code, compile using a compiler written for the environment in which you'll run the program, link/bind the program so it can run in your environment, and run the program.

**Your challenge:** Compile and run a program on the mainframe.

Navigate to your CC#####.SOURCE partitioned data set and check out the member named POKER, which contains source code for the program you'll be compiling. The program reads random pairs of poker hands from an input file, analyzes them, and determines which player should be declared the winner of each round. Take a moment to
review the comments and code in this member.

A common way to accomplish the compile, link/bind, and run steps of developing a program on the mainframe is to submit some JCL. Back out and open your CC####.JCL data set, where you will find some JCL that has already been prepared for these tasks.

Edit the member called POKERCMP, which compiles and binds the program. Change the jobname to CCMP####, where #### is the last four characters of your userid. Now submit the job.

**Hint:** The HILITE command allows the ISPF editor to color-code data based on context when you are Editing or Viewing a member. The editor is capable of coloring keywords, symbols, comments and operands. Simply invoke HILITE at the command line, and the editor will take care of the rest. It recognizes just about every major language out there, and if you don't want to specify the language every time, you can simply type HILITE AUTO and let the editor figure it out on its own. HILITE OFF returns to the unhighlighted view. Enabling HILITE in your ISPF editor is very handy and will make editing code easier!

You should see a MAXCC of 0 when this JCL completes, which you can verify by checking your output in SDSF. The end result of the steps contained in POKERCMP are that an executable member (similar to a .exe file on your PC) was created and stored in your CC#####.LOAD data set. You can pop over there and verify that it's present if you want, but you'll find if you attempt to view the contents of LOAD(POKER) that it's not a readable file.

Now, edit the CC#####.JCL member called POKRUN13, which will run your new program. Change the jobname to CRUN####, where #### is the last four characters of your userid, and submit the job.

Oh man! A JCL ERROR message will be returned to the screen. What gives? Check your output in SDSF and fix the error.

**Hint:** Plug the message ID from JESYSMSG into the LookAt Facility for some help (the message ID starts with IEF). The system you are currently using is z/OS, release level V2R1. The Application Programmer Response provides the information needed to correct this JCL problem.

**Another hint:** Pay attention to the comments in both JCL members!

Correct the error and re-submit the POKRUN13 JCL to (this time successfully!) run your program. You will be rewarded with a MAXCC of 0 when your JCL completes, and a shiny new $013 member in your P2.OUTPUT data set containing the output from the poker program.

Important note: We are providing multiple POKRUN## JCL members for this and subsequent challenges. This is to ensure that we can evaluate your output from each challenge. As such, if you accidentally submit POKRUN13 after working on the next challenges, your P2.OUTPUT($013) member will get clobbered with incorrect output. Take care!

Once you verify that you got this new data set member exists and contains appropriate output, that's it! You just debugged a JCL job. Life is good. Let’s keep going.

Back to Top [Collapse All]

12. Know When to Hold 'em, Know When to Fold
Background: Oftentimes, you’ll find yourself working with a program and realize that you need to change the program; either there is a problem with it, or you want to add some new functionality. The process is simple: change your source code, recompile the program (and fix any errors introduced to your source code, recompile, and repeat a few times), and run the program again to test your changes.

Your challenge: Make a modification to the C program that analyzes poker hands.

You’ve already taken a look at the C source code member in CC#####.SOURCE(POKER), and you’ve already seen what it does. Now you’re going to make a few modifications.

Never programmed in C before? That’s okay! You will discover that the syntax of C is similar to other modern languages like C++ and Java, and there are examples of everything you need to do already present in the program: the program already reads a file, does some processing on the data, and writes some output.

Hint: Don’t ignore code comments! There are some really great clues there as well.

Make the following changes to the program:

1. Have the program print two header lines at the top of the output.  
   The first line should say John Doe’s Poker Program. Replace “John Doe” with your name.  
   The second line is simply blank.

2. Instead of having the output contain a message for each poker hand analyzed, the output needs to contain a short report detailing the number of hands won by each player and the percentage of the total. You need to find a way to prevent the program from printing the player wins messages and tally up the wins for each player. Some math is required to calculate the win percentage as well.  
   The output from your C program should contain the two lines mentioned, and the following text. Note that we have adjusted the numbers here so they will not match the correct output -- copy and paste is not an option.

   Summary:  
   Total games: 30  
   Player 1 total wins: 13 Percentage 43%  
   Player 2 total wins: 17 Percentage 57%

There are no card combinations in the input file that would result in a tie between the two players, and all input is valid. All hands contain five cards, and there are no invalid characters or repeated cards.

The percentages must be rounded down to whole numbers. C’s division operator (/) does this for you automatically.

Hint: If you haven’t already used the HILITE command to let the ISPF editor color-code your source code, you’re missing out!

Once you have your source code changed, submit POKERCMP to recompile it, and when that has run successfully, submit JCL(POKRUN14) to execute the program with your changes. The JCL creates a new member in P2.OUTPUT named $014.

Important note: There are differnet POKRUN## JCL members for each of these poker
challenges. We've done this to preserve the output from each challenge in different
P2.OUTPUT members. If you accidentally run JCL(POKRUN13) at this point, your
P2.OUTPUT($013) member will be overwritten with the wrong data. Take care!

Once you get everything compiled and successfully run, check the $014 member in your
P2.OUTPUT data set and make sure it looks ok. Does it? Great - that means you’re done
with this challenge. On to the next!

13. Luck be a Mainframe

**Background:** Just like any other operating system, z/OS provides a means to protect data
from unauthorized access. It’s called the Resource Access Control Facility, or RACF
(pronounced RACK-F). Specifically, RACF allows the mainframe to:

- identify and verify system users
- identify, classify, and protect system resources
- authorize the users who need access to the resources you’ve protected
- control the means of access to these resources
- log and report unauthorized attempts at gaining access to the system and to the
protected resources
- administer security to meet your installation’s security goals

Good stuff, and it’s extremely important to know how to protect your data. Let’s get some
learning on.

**Your challenge:** Authorize another user to access your data sets and run your poker
program on your behalf.

Head on over to your CC#####.JCL data set and edit the member called POKERACF. This
JCL requests that your POKER program be run by surrogate user ZUSERID. You might
notice that some of the mechanics of this JCL are different than that of POKERRUN, but
that’s ok! Change the #'s in the jobname to the last 4 digits of your user ID. Then on line 7
change **MYUSERID** to your actual user id. Submit the JCL.

Since your job ran with the parameter user=ZUSERID, you won’t get notified when it
completes. It shouldn’t take long, so head on over to SDSF to check out the results. Wait,
where are the results? Since the JCL was run by the surrogate user ZUSERID, ZUSERID is
the owner. In the =SD;ST panel, enter the primary line command:

```
set display
```

This command causes an additional line of information to show following the command
line. Take a look at the OWNER= field. It’s your user ID. This means that SDSF is filtering all
job output to only show jobs owned by your ID. To change the output to show jobs owned
by ZUSERID, issue the following primary line command:

```
OWNER ZUSERID
```

Next, enter the following command, replacing **RACF####** with the jobname you assigned
in the JCL:

```
```
Enter an S in the NP column next to your job to view the results.

The first thing you should notice is that the job did NOT run successfully. You should see an ICH408I message letting you know that the job encountered “insufficient access authority” when it attempted to run, and you should see that the first step in your JCL, listed as stepname "caller1," did not get an RC=0, it got an RC=FLUSH, meaning the step was never actually run.

Congratulations, you’ve just had your first negative encounter with RACF! If you look more carefully at the “insufficient access authority” message, you see that user zuserid had the intent to read the data set called userid.load (where userid is your user id), but the access that was allowed was none. In order for this job to run successfully, zuserid must have authority to read your LOAD data set. That’s where RACF comes in. Using RACF, you can create discrete profiles to protect your data sets and then permit individuals or groups to use your data sets.

RACF includes a command called LISTDSD which allows you to display information about how an individual data set is protected, so you can check your work as you go and wait to re-attempt to submit your POKERACF JCL until everything looks right. Ready?

Navigate to the ISPF Command Shell (option =6) to execute RACF commands.

A detailed description of the RACF commands you will need is available in this section of the z/OS Security Server RACF General User’s Guide.

Using the commands you find in the RACF General User’s Guide, create a discrete profile for your data set (do not create a generic profile – generic profiling is beyond the scope of this challenge!) and set the appropriate access levels. The UACC (universal access) for any profile you create should be set to NONE – after all, you don’t want to give every user on the system access to your data set! Be sure to use the LISTDSD command to check your work when you’re finished!

You may have to create multiple discrete profiles and give ZUSERID different permissions to various data sets that you own. Read the RACF ICH408I messages carefully in your job output.

**Hint:** The data sets you are creating discrete profiles for are CATALOGED data sets.

Once everything’s looking good, re-run the POKERACF JCL and check out your output in SDSF. When everything is set up correctly, you should see no ICH408I messages in your output, you should see that stepname “caller1” executed with an RC=0, and most importantly, you should have a new piece of output in your P2.OUTPUT data set called $015 which should look similar to member $014 from the previous challenge. The results will not be exactly the same, as the input data for the poker hands is different this time around. Got it? Wahoo!

**One last thing:** Don’t forget to reset the OWNER specification in SDSF so that next time you need to check some job output in SDSF you see jobs from your own queue instead of ZUSERID's!

Back to Top (Collapse All)

14. Lay All your Cards on the Table
Background: You’ve gotten your feet wet with some C programming, now let’s go for a swim in the ocean.

Your challenge (1 of 2): The first part of this challenge is to add more analysis on the input data. In addition to determining the winner of each round, let’s take a look at what hands are encountered each round and tally them up.

In the game of poker, each hand is ranked, from highest to lowest, in the following way:

- Royal Flush: Ten, Jack, Queen, King, Ace, in same suit.
- Straight Flush: All cards are consecutive values of the same suit.
- Four of a Kind: Four cards of the same value.
- Full House: Three of a kind and a pair.
- Flush: All cards of the same suit.
- Straight: All cards are consecutive values.
- Three of a Kind: Three cards of the same value.
- Two pairs: Two different pairs.
- One pair: Two cards of the same value.
- High Card: None of the above.

Your goal is to generate a section to be included in your output report following the "Summary" section which indicates the total number of hands analyzed and how often each hand type occurred. The new section must be formatted as shown below. Note: The data displayed here is incorrect, we’ve modified it slightly to discourage copy and paste!

Hand Occurrences:
Total Hands Analyzed: 60
Royal Flushes: 1 (1%)
Straight Flushes: 4 (4%)
Four of a Kinds: 2 (2%)
Full Houses: 2 (2%)
Flushes: 4 (4%)
Straights: 5 (5%)
Three of a Kinds: 6 (6%)
Two Pairs: 19 (19%)
One Pairs: 32 (35%)
High Cards: 20 (20%)

Warning: Again, we’ve provided you with a unique runtime JCL member for this challenge. You’ll find it at CC#####JCL(POKRUN16). Use this JCL to generate the appropriate output data set member. As before, your compile JCL remains the same.

Hint 1: The source code already includes logic to determine the type of hand (flush, 4 pair, etc...). The only missing hand type is called a Royal Flush. You need to create some logic to differentiate a royal flush from a straight flush.

Hint 2: A royal flush always has an ace high card, in addition to being a straight flush. Take a careful look at the scoreHand subroutine and spend some time understanding what’s really happening in the program.

Take your time adding this additional functionality. It’s good practice to add a few lines of code to a working program, and then compile and run the program to make sure the few changes doesn’t break anything. Once you have added the new logic to the POKER program and things are looking good, move on to the next part to this challenge.

Your challenge (2 of 2): The second part of this challenge is to change the game up a
little bit. In poker, the more players in the game, the more dynamic it gets. Let's deal in another player!

The input data set for the POKER program actually contains fifteen "cards" per record. Currently, the program reads in only the first ten cards. Five for player one and five for player two, then discards the remainder of the record. Your challenge is to read in the last five cards per record as player three, perform the same hand analysis as was done for the first two players, then determine the winner of the three players.

Important: Make sure you update the Summary and Hand Occurrences sections to include player three's win statistics and hand type occurrences.

That's it! You're nearly there! Don't forget that you can get an original copy of the POKER source code by copying it from Y2015.SOURCE(POKER). Once you’ve successfully completed all the steps in this challenge, your P2.OUTPUT($016) member will look very much like the following (again we’ve changed the numbers on you):

John Doe's Poker Program

Summary:
Total games: 30
Player 1 total wins: 12 Percentage 40%
Player 2 total wins: 10 Percentage 33%
Player 3 total wins: 8 Percentage 26%

Hand Occurrences:
Total Hands Analyzed: 90
Royal Flushes: 3 (3%)
Straight Flushes: 2 (2%)
Four of a Kinds: 2 (2%)
Full Houses: 2 (2%)
Flushes: 3 (3%)
straights: 5 (5%)
Three of a Kinds: 6 (6%)
Two Pairs: 8 (9%)
One Pairs: 32 (35%)
High Cards: 26 (28%)

Once you’re done with both parts of the challenge and get your program to successfully run, feel free to update your resume with the title "Mainframe Master" and move on to the final step!

Back to Top (Collapse All)

15. That's all, Folks!

Congratulations! If you made it this far, you have nearly finished with the “Part 2: Practical Experience” section of the contest, a most impressive accomplishment. If you followed all the steps correctly, you should have 14 members in your P2.OUTPUT data set:

1. $001
2. $002
3. $003
4. $004
5. $005
Review your work carefully! In order to win the Master the Mainframe Prize Pack, you must be among the first 75 contestants to have every challenge from this part of the contest completed 100% correctly.

When you are confident in your answers, submit them for grading by entering the following command on any command line:

```
TSO SCOREP2
```

Hopefully you’ll see a message indicating you successfully completed part two! If not, take a look at your STATUS data set for messages indicating missing or incorrect part two data set members.

Regardless of whether or not you successfully finish among the first 75 contestants, you really have demonstrated some significant mainframe skills to get this far, and you should be proud of your work. Employers will want to hear about your experience! Be sure to let them know about it on your resume. Don’t forget to peruse the mainframe jobs posted at Systemzjobs.com.

If you’d like more mainframe education, but your college, university or high school doesn’t currently offer any, talk to your educators about the IBM Academic Initiative. IBM offers free course materials and mainframe access for students and professors. See the Academic Initiative System z program for more information.

Really excellent work finishing Part Two! We love it when you talk about the Master the Mainframe Contest on social media sites. Tell the world how smart you are!

The “first one to the finish” element of the contest concludes here. For Part 3, contestants have until December 31 to finish their work. Only students who are very serious about putting in many hours on the mainframe over the next few months should attempt Part 3. If you do go on, you’ll have the option to try using Rational Developer for System z. Cool! For details, see the Part 3 instructions.

**Note: the dates in the paragraph above are for the US and Canada contest only, please review your local contest information for appropriate information.**

**Thanks for competing in Part 2 of the Master the Mainframe Contest, and good luck if you choose to compete in Part 3!**

Back to Top (Collapse All)