# Welcome to Assignment 2!

Approximate time to completion: 6 to 8 hours.

Total Marks: 70

*You must complete Assignment 1 before beginning Assignment 2.*

The objective of this assignment is to get you acquainted with some of the more practical aspects of systems programming and application development on the mainframe. These challenges are modeled after tasks that are commonly performed in z/OS shops.   
  
The instructions are not nearly as detailed as Assignment 1, you might have to use some additional resources, along with your new-found mainframe expertise, to complete the challenges. Though several of these challenges are difficult, they have been designed so that you'll know whether or not you've completed them correctly before you proceed. Take your time, check your answers and read the instructions carefully!

Table of Contents

[Welcome to Assignment 2! 1](#_Toc350157454)

[1. Mainframe Documentation 2](#_Toc350157455)

[2. Setting Up Your Assignment Two Data Sets (5 Marks) 3](#_Toc350157456)

[3. JCL and SDSF (5 Marks) 5](#_Toc350157457)

[4. Fix a JCL Error (or Two) (10 Marks) 8](#_Toc350157458)

[5. System Commands (12 marks) 9](#_Toc350157459)

[6. REXX Commands Menu (10 Marks) 11](#_Toc350157460)

[7. Customize your own ISPF panel (5 Marks) 13](#_Toc350157461)

[8. C Programming: Abundantly Lucky Numbers and Mersenne Primes (10 Marks) 15](#_Toc350157462)

[9. C Programming and JCL (10 Marks) 18](#_Toc350157463)

[10. Mainframe Adventures with JAVA! (10 Marks) 20](#_Toc350157464)

[11. Submitting Assignment 2 (3 Marks) 23](#_Toc350157465)

## 1. Mainframe Documentation

If you get stuck during Assignment 2, take a look at these mainframe resources:

* [*Introduction to the New Mainframe*](http://www.redbooks.ibm.com/redbooks/pdfs/sg246366.pdf): A college textbook; great for introductory and overview information about the mainframe.
* [z/OS V1R13.0 information center](http://publib.boulder.ibm.com/infocenter/zos/v1r13/index.jsp): Contains all of IBM's official technical documentation on z/OS, with hundreds upon hundreds of searchable books and documents. Everything you'd ever need to know about mainframes can be found here.
* [*ABCs of System Programming, Volume 9*](http://www.redbooks.ibm.com/redbooks/pdfs/sg246989.pdf): Written by system users, this is a great resource for systems programmers.
* [Message LookAt Facility](http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/): If you are curious about a message issued by the system, you can look up a detailed explanation here.
* [Mainframe blog](http://mainframe.typepad.com): A mainframe blog that gives you a chance to interact with other mainframe enthusiasts.
* [z/OS Hot Topics newsletters](http://www.ibm.com/servers/eserver/zseries/zos/bkserv/hot_topics.html): Keep up on the latest technical news about z/OS.

## 2. Setting Up Your Assignment Two Data Sets (5 Marks)

We now need to create (and copy) the data sets you will use to complete Assignment 2.

Issue the following command from the ISPF Command Shell command line, found at option 6 from the ISPF Primary Option Menu:

SUBMIT 'KC02315.ASSIGN2.JCL(SETUP)'

Don't forget the quotes! This job may take a minute to complete.

Now let's go take a look at the data sets that were created. 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 your data sets. There should be 8 new ones that have been created for Assignment 2, including:

'***KC03###***.ASSIGN2.CLIST'

'***KC03###***.ASSIGN2.JCL'

'***KC03###***.ASSIGN2.LOAD'

'***KC03###***.ASSIGN2.OUTPUT'

'***KC03###***.ASSIGN2.PANELS'

'***KC03###***.ASSIGN2.REXX'

'***KC03###***.ASSIGN2.SOURCE'

and

'***KC03###***.ASSIGN2.WORK'

where ***KC03###*** is your Marist userid.

The data set called '*KC03###*.*ASSIGN2.WORK*' (where *KC03###* is your Marist userid) is very important, it will eventually contain all of your work to be handed in for Assignment 2. From here on out, we will refer to this data set as *ASSIGN2.WORK*.

On the Command Line enter =2 to go directly to the Edit Entry Panel. In the Name field under the "Other Partitioned, Sequential or VSAM Data Set, or z/OS UNIX file:" title, enter:

**ASSIGN2.WORK(NEWMEMBR)**

This will automatically open a new member (named NEWMEMBR) for you to edit. Enter "*KC03###* WAS HERE 2021" on line 1 and a comment telling me what you think of mainframes (be nice now) on line 2. Remember the KC03### is your Marist userid. Now F3 back out, and you'll see a message indicating "Member NEWMEMBR saved"   
  
**Note on starting over:** If you mess up any of your *KC03###* data set members during Assignment 2, you'll be able to find read-only copies for almost all of them in the KC02315.ASSIGN2 data sets. For example, if you have a member called *KC03###.ASSIGN2.*JCL(BADJCL) that you'd like to replace, you can find a fresh copy at 'KC02315.ASSIGN2.JCL(BADJCL). To make a new copy just use the Move/Copy Utility (3.3) to copy 'KC02315.ASSIGN2.*qualifier*(*membername*)' to your data set. Don't forget to place a forward slash (/) on the Replace like-named members option on the second copy screen.

## 3. JCL and SDSF (5 Marks)

**Background:** In Assignment 1, you ran a simple [REXX](http://en.wikipedia.org/wiki/Rexx) program from the ISPF Command Shell (=6). Now we are going to run another REXX program, only this time in batch. In z/OS, you have two different ways to do work: interactively in the foreground or non-interactively in the background (aka "batch processing"). All of your work in Assignment 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 the mainframe world, a "job" is a collection of different programs that work together and runs in the background. In order to cause work to run in batch (that is, to submit a job), you need to instruct z/OS through JCL (Job Control Language). You submit your JCL to JES (Job Entry Subsystem) and 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 Assignment 2.

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.

**Your challenge:** Submit JCL to execute a REXX exec as a batch job.   
  
First, edit member RUNREXX in *KC03###.ASSIGN2.*JCL. This member contains a job that is very close to what you need. Inside that member, edit the job name so that it ends with the last three hexadecimal digits of your user ID. For example, change it to REXXD11 if your ID is KC03D11. (REXX### is the jobname. The jobname is the very first thing in the JCL job, and it identifies the job to the system.) ***From here on out, whenever you see three # signs in a row, you can assume that you need to replace them with the last three hexadecimal digits of your user ID.***  
On the command line, enter **SUB** (short for submit).   
  
This will send your JCL, which you can also call your job, to JES for processing. JES manages all jobs submitted to the z/OS system. JES will put your job on a queue (called an "initiator"), which helps decide when to submit your job for execution, and helps allocate some resources needed for the job. When your job runs, JES manages output from your job (and all other jobs that are submitted) in a special set of data sets called the "spool". Press Enter to see the results of your submission. See the text MAXCC=0 inside the system generated message? That means that your job ran with a return code of 0, which is good news. Press Enter again to dismiss the message and return to your job.  
  
To check the status of your job, enter the following on the command line:  
  
**tso status REXX###**   
  
which should tell you that your job is on the output queue.

To display the output produced when your job ran we use the Spool Display and Search Facility (SDSF).   
  
To get to SDSF on the Marist system, enter **SD** from the ISPF Primary Option Menu or =SD from any ISPF command line. From the SDSF Primary Option Menu, enter the following command:   
  
**OWNER *KC03###***

This filters the output so that you only see your own jobs. Then enter the following SDSF command:   
  
**ST**   
  
ST, or STATUS, will display the status of all tasks owned by OWNER *KC03###*.

***Note:*** If you've already set the PREFIX option to KC03###\* for the labs just reset it by entering prefix without any parameters (i.e. PREFIX) .   
  
View your output, along with some system generated output, by entering a ? to the left your jobname. The jobname specified in your JCL job was REXX*###*. Enter S (select) to the left of each piece to see output of your job.   
  
F3 will exit each piece of the output and a second F3 will return to the panel that lists the entire REXX*###* job in the JES output queue.   
  
Enter an S to the left of REXX*###*.   
  
What you'll see now is all your output in one place. Entering **?** to the left of the jobname separates the output into logical sections (these sections are called DDNAMES). Entering S to the left of the jobname will display all the logical sections concatenated as a continuous stream of output. To view all of the output, you'll need to use F7 and F8 to scroll up and down, and F10 and F11 to scroll left and right.   
  
At the bottom of the output (in the OUTDD section), you'll see a list of all users who are currently logged into the Marist system.   
  
After viewing the output of your job, you need to write out all of the output to a member of your *ASSIGN2.WORK* data set called RUNREXX.   
  
To do this, press F3 to return to the status queue, then navigate to the left of job REXX*###* and enter the command XDC. On the next menu, enter the Data set name (specify *ASSIGN2.WORK)*, the Member to use (RUNREXX) and Disposition (OLD). You can accept all the other defaults. After pressing enter, you should see the message "PRINT CLOSED" on the top right-hand corner of the status display.  
  
Now you'll find a new member in your *ASSIGN2.WORK* data set. Back out of SDSF and go have a look.   
  
If you weren't successful and want to purge your job output (or have successfully completed this challenge and want to clear the output so it doesn't distract you or get in your way), go to the SDSF Status Display menu and enter P to the left of any jobs that you want to purge. Press enter and watch them vanish. You'll also find this useful in later challenges.   
  
**Note on purging:** Marist purge output automatically every 24 hours, or as needed to keep the system tidy. If you run a job and want to make sure you can refer to the JCL output later, use the XDC command to copy it to one of your *KC03###* data sets.   
  
**Note on DDNames:** You may have noticed the use of the term DDName above, and since you'll be working more with DDNames later, let's take just a moment to explain them. DDName, or Data Definition Name, is the eight-character designation after the // of the DD statement. The DD statement is a job control statement describing a data set associated with a specific job step.

## 4. Fix a JCL Error (or Two) (10 Marks)

**Background:** As a systems programmer, you will be doing lots of work with JCL. If you make a simple JCL error, such as forgetting a comma or putting a character in column 72, your job will, in most cases, end very quickly, and the system will inform you of a JCL error.

**Your challenge:** Fix simple JCL errors.

First, edit member BADJCL in *KC03###*.ASSIGN2.JCL and change the jobname (BJCL*###*), substituting the last three hexadecimal digits of your user ID for the pound signs. Submit the job. You'll get a JCL error.   
  
Check your output in SDSF and fix the error(s).   
  
**Hint:** Plug the message ID from JESYSMSG into the [LookAt facility](http://www-03.ibm.com/systems/z/os/zos/bkserv/lookat/index.html) for some help (the message ID starts with IEF). The system you are currently using is z/OS, release level V1R12.   
  
**Another 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.   
  
After you fix the error(s) and get the job to run correctly with no JCL errors, go back to SDSF to check your held output. Once you verify that your job ran successfully, XDC all of the output from the job into your *ASSIGN2.WORK* data set under the member name FIXEDJCL. Again, be sure that you XDC from beside the JOBNAME (not a DDNAME).  
  
**Note:** If you look at your ASSIGN2.WORK data set it should now contain a member named EBCDIC (you've done the challenge correctly) which contain a sort list of characters. The purpose of this JCL is to read in a data member which contains printable characters collated (sorted) in the ASCII environment. It runs the SORT utility on the data member and displays the output.

That's it, you completed this challenge.

## 5. System Commands (12 marks)

**Background:** In this challenge, you will learn how to investigate the system configuration used by the Marist system.   
  
**Lookahead warning:** Take a moment to peek ahead to the next challenge (REXX Commands Menu), in which you will use the commands you discover here to complete the challenge there. I recommend writing down the list of the commands you use in this challenge because you really need them later on.  
  
**Your challenge:** Your next challenge is to complete a quiz named SYSCMDS.

**Hint:** Look at member QUIZCMDS in your ASSIGN2.HINTS data set for a list of system commands (in the correct order) that will help with this challenge.   
  
  
To bring up the quiz, enter the following command:   
  
**TSO EXEC 'KC02315.ASSIGN2.QUIZ(SYSCMDS)'**   
  
The quiz consists of twelve multiple choice questions about the system. The answers can be found by issuing commands on the mainframe.   
  
To aid you in this challenge, it's time you learned how to use an indispensable tool: screen splitting. Move your cursor to the topmost line of the ISPF panel and press the F2 key (this can be done from any panel). You will notice that a brand new ISPF Primary Option Menu has appeared, and there is a row of dots across the top. You just started a second screen, which you can use just like you would use a normal screen. To toggle between your two screens, use the F9 key. You might find this newfound knowledge helpful when you're comparing data sets or copying output from one member into another. When you're done with your second screen, just hit F3 until you back all the way out of it. Your second screen will disappear, leaving you with only your original screen.   
  
Once you've got your screen split, press F9 to toggle to the ISPF Primary Option Menu panel. From here, enter SD.ULOG to enter the user log of SDSF. From the command line on this panel, type **/D M**. This will display some information about the system configuration. The answer to the first question in the quiz can be found in this display. Go ahead and hit F9 to toggle back to the quiz and enter your answer.   
  
A tool you will still find useful is the [LookAt website](http://www-03.ibm.com/systems/z/os/zos/bkserv/lookat/index.html), where documentation for any z/OS message can be found.   
  
The commands needed to answer the rest of the quiz questions can be found in [MVS System Commands, chapter 4: DISPLAY](http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.ieag100/display.htm). In the ULOG environment, any command must start with a forward slash (/) as you used before for the D M command.   
  
When you have answered all the questions, enter the following command to check your work:   
  
**TSO EXEC 'KC02315.REXX(CHKQUIZ)'**   
  
This will run a REXX program that that checks your answers. When it completes, check out your *ASSIGN2.WORK* data set for a member called SYSCMDS (your answers are there too in a member called SYSCMDA). The SYSCMDS member contains a note indicating your score. Didn't get them all right? Go ahead and re-take the quiz! You can take the quiz and check your answers as many times as necessary until you've got a perfect score.

You now know a little bit more about how the Marist z/OS system is configured, and you know how to issue system commands.

## 6. REXX Commands Menu (10 Marks)

**Background:** In this task, you'll be learning a little bit about [REXX](http://en.wikipedia.org/wiki/REXX). REXX, or Restructured eXtended eXecutor, is a language that is particularly suitable for command procedures, application front ends, user-defined, prototyping, and personal computing. Individual users can write programs for their own needs. REXX is a general purpose programming language with the usual structured-programming instructions - IF, SELECT, DO WHILE, LEAVE, etc. - and a number of useful built-in functions.

**Your challenge:** Your next challenge is to modify a REXX exec that presents yourself with a panel-driven interface for issuing system commands and viewing their output.   
  
Navigate to your *KC03###*.ASSIGN2.CLIST data set and edit the MYPANELS member. Replace ***all*** instances of *###* with the last 3 hexadecimal digits of your user id and save your changes.   
  
Use F3 to back out of ISPF until you get to the TSO Ready prompt. There, enter the following command:   
  
**EX '*KC03###.ASSIGN2.*CLIST(MYPANELS)'**   
  
This activates your personalized clist, or command list. The change you made tells the system where to look first when a panel (window, in PC-lingo) is requested. By putting your *KC03###.ASSIGN2.*PANELS data set at the top of the list, you've instructed the system where the panel that supports your REXX application lives.   
  
Now, navigate to your *KC03###.ASSIGN2*.REXX data set and enter the word **exec** in the prefix area to the left of the REXXSYSC member. You should be presented with a panel containing a list of commands (well, not quite yet). Your challenge is to complete this "command list skeleton" to contain the commands you discovered in the previous challenge. Remember I told you to write down your commands? Aren't you glad you listened?  
  
So how does REXXSYSC work? Go ahead and F3 out of the application then put an E next to REXXSYSC in the data member list. The source code has lots of comments explaining what it does. Modify the source code so that when the exec is run it uses the commands you want it to. F3 out of the source and exec it again.   
  
Hmmm, it looks like you didn't make any changes! Go ahead and enter option #1. After just a second you should see the output you would see in SD.ULOG if you tried the command you just entered for menu option 1. Your REXX application has been updated, but now you need to modify the panel that sits in front of it.   
  
Navigate to your *KC03###*..ASSIGN2.PANELS data set and take a look at SYSCPAN. This is the source code for the panel that is displayed by your REXX exec. Go ahead and modify it by once again entering your list of system commands. Note that this is just the "display" of your application, but spelling is very important. In order to get this challenge 100% correct, you MUST double check your work and make sure that the display matches the command that is issued when each option is chosen, and each command is free of typographical errors. The commands may be abbreviated where the system allows (for example, D IPLINFO can be used instead of DISPLAY IPLINFO).   
  
All done? Enter =X on the command line to exit ISPF. When you get to the TSO READY prompt, enter ISPF to restart ISPF (to pick up the changes you just made), and return to your KC03###.ASSIGN2.REXX data set. Try your application again. Everything look okay? I recommend testing each and every one of your commands to verify that all are issued correctly.

You should notice that after issuing each command and perusing its output, hitting F3 returns you to the command menu.   
When you're done testing, hit F3 from the menu to exit the app. Copy your REXXSYSC source to *ASSIGN2.WORK* as member name REXXSYSC and copy your SYSCPAN panel source to *ASSIGN2.WORK* as member name SYSCPAN. Good job, challenge complete!

## 7. Customize your own ISPF panel (5 Marks)

**Background:** You can add a new icon on the desktop of your PC, and you can customize your environment in z/OS. The panel interface of ISPF can be completely rewritten to fit your needs. Most mainframe shops do this as a first step of their customization. In fact, the z/OS system that you are using right now already has a degree of customization to its panels.  
  
**Your challenge:** Create your own customized ISPF panel.   
  
Navigate to your *KC03###*.ASSIGN2.PANELS data set and create a new member called ISR@PRIM. This is the name of the panel that displays the ISPF Primary Option Menu, which is what you see immediately after logging on to the system. Once you've created the new member, enter **COPY** on the command line and, on the next panel, enter  
  
**'KC02315.ASSIGN2.CUSTPANL(ISR@PRIM)'**   
  
to copy our customized panel to your PANELS data set.   
  
Recall that in the REXX challenge you modified *KC03###*.ASSIGN2.CLIST(MYPANELS) so that your panels located in *KC03###*.ASSIGN2.PANELS take precedence. Now that you have your own version of ISR@PRIM in your PANELS data set it will be displayed in place of the ISPF Primary Option Menu.   
  
Drop down to the TSO READY prompt (=x in ISPF is the shortcut) and enter this command, substituting your user ID for *KC03###*:   
  
**ex '*KC03###*.ASSIGN2.CLIST(MYPANELS)'**   
  
After the command executes, you should be presented with a very slimmed-down version of the ISPF Primary Option Menu.   
  
**Note:** If at any time you need to get back to the original ISPF main panel (as you might if you break something in the course of completing this challenge), you can do so by exiting from your ISPF session to TSO READY, then entering LOGON *KC03###* from the TSO READY prompt and following the steps you already know from there.   
  
So now let's set it up to execute your system commands REXX exec directly from your new ISR@PRIM panel.   
  
Take a look in your *KC03###.ASSIGN2*.PANELS data set and peek at ISR@PRIM. It currently has examples of how to invoke programs and display panels. The other function that can be called from a panel is to run a command. Take a look at this page in the [ISPF User's Guide, Volume 2](http://publib.boulder.ibm.com/infocenter/zos/v1r13/index.jsp?topic=/com.ibm.zos.r13.f54u200/ispzu29048.htm) for information about functions you can call from an ISPF panel.   
  
Your REXX exec should be invoked when the **R - REXX** option is specified from your slimmed down ISPF Primary Option Menu. Make sure that you've included the code to run the exec as well as the code to display the option on the panel!

Hint: To run a REXX exec from a ISPF panel specify:

**CMD(rexx\_name) LANG(CREX) MODE(LINE)**  
  
F3 back to the TSO Ready prompt and restart ISPF. Does your new option show up? Great! Try it out.   
  
If it doesn't work, that's because you have to tell ISPF where your REXX exec lives. This is done similarly to how you tell ISPF where your panels live. This should be done in the SYSEXEC concatenation in your MYPANELS CLIST. (Note: If it did work, you updated MYPANELS correctly the first time).   
  
Once you've got this step done you will be able to re-execute your CLIST from TSO ready and then successfully launch your REXX exec from your slimmed down ISPF Primary Option Menu.  
  
When you've verified that everything works as it should, copy your ISR@PRIM to *ASSIGN2.WORK* as member name ISR@PRIM.

## 8. C Programming: Abundantly Lucky Numbers and Mersenne Primes (10 Marks)

**Background:** The mainframe supports many programming languages, including C and C++. Let's have some fun with C programming.  
  
**Your challenge:** Modify a C program to behave a bit differently, then use JCL to compile and run it.

**Hint:** Look at member CROUTINE in your ASSIGN2.HINTS data set for some routines to help with this challenge.   
  
Often in the mainframe world (and in the computer world in general), you'll be provided with a functional application and you'll be asked to modify it to serve a new purpose. That is the case here.

First, we'll compile and run the program as-is:

1. Take a look at *KC03###*.ASSIGN2.SOURCE(LUCKSRC). This is the C code that you'll be working with. If you haven't been using the HILITE AUTO command to this point, you might want to start using it now.
2. Now take a look at *KC03###*.ASSIGN2.JCL(CCMP). This is the JCL code that compiles the code in LUCKSRC. Change the jobname to CCMP*###*, where *###* is the last three hexidecimal digits of your ID, and submit CCMP to compile the source code. A successful JCL run will produce an executable module in *KC03###*..ASSIGN2.LOAD called LUCKY. In SDSF, JESMSGLG will show RC 00 for both the COMPILE step and the BIND step if everything was successful.
3. Once your code is compiled, take a look at *KC03###.ASSIGN2.*JCL(CRUN). This is the JCL that runs the program you've just compiled. Change the jobname to CRUN*###*, where *###* is - you guessed it - the last three hexadecimal digits of your ID, and submit CRUN to run the program. In SDSF, make sure the RC of the EXECUTE step is 0. Further indication of success will be contained in your SYSPRINT DDNAME, where lucky numbers should be shown.

Let's talk about math for a minute. The program you just ran displayed [lucky numbers](http://en.wikipedia.org/wiki/Lucky_number). In number theory, a lucky number is a natural number in a set which is generated by the following process: first, write out all odd numbers: 1, 3, 5, 7, 9, 11, 13 ... The first number greater than (>) 1 is 3, so strike out every third number from the list, leaving you with this: 1, 3, 7, 9, 13, 15, 19 ... The first number >3 is 7, so strike out every seventh number leaving you with this: 1, 3, 7, 9, 13, 15, 21, 25, 31 ... Repeat this process indefinitely to produce the list of lucky numbers. Lucky for you, the algorithm to accomplish this process is already included in your program.   
  
Now you're going to change things up a bit.   
  
**Important note:** You will see some pre-existing code in LUCKSRC that doesn't appear to have anything to do with the current task. We'll probably use it in the next challenge so just leave it there!   
  
**Pro tip:** As you use the F7 and F8 keys to scroll through your source, you may be annoyed that you always skip around a full screen at a time. You can change the scroll amount (in the top-right corner) to CSR so that when you scroll, you only scroll so that the cursor is at the top (or bottom) of the screen.

1. First, modify the program so that only lucky numbers that are also abundant are printed. An abundant number is a number n for which the sum of the divisors of n is greater than 2\*n (2 times n). Example: the divisors of 12 are 1, 2, 3, 4, 6, and 12. The sum of these is 28. Since 28 is greater than 2\*12, or 24, the number 12 is abundant.   
     
   The first few abundant numbers are 12, 18, 20, and 24 and the first lucky number that is also abundant is 1575.
2. Next, modify the program again to also print [Mersenne primes](http://en.wikipedia.org/wiki/Mersenne_prime). A Mersenne prime is a Mersenne number that is also prime. A Mersenne number is a positive integer that is one less than a power of two, and a prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. The first few Mersenne primes are 3, 7, 31, and 127. (1 is a Mersenne number, but it is not a prime!)  
     
   **Hint:** the isPower2 function in *KC03###.ASSIGN2.*SOURCE(LUCKSRC) should make this a breeze!
3. Now, change the text that is output by the program to more closely match the new function of the program. That is, change the words "is lucky!" to "is abundantly lucky" if the number is abundant and lucky or "is a Mersenne prime!" if the number is a Mersenne prime. No numbers in the range you're checking are both abundantly lucky and Mersenne primes.
4. Finally, update the source to include two headers lines in your output:   
   - Your Name's Abundant and Lucky Numbers and Mersenne Primes   
   - A blank line
5. Once you've made your changes, submit CCMP to re-compile your program and, once that is successful, submit CRUN to run your program.

When your program runs successfully, SYSPRINT should contain abundantly lucky numbers and Mersenne primes. You'll see that the output is all clumped together, with Mersenne primes and abundant and lucky numbers all over the place! We'll fix that up in the next challenge. For now, your output should look exactly like this (as long as your name's Tim - and yes, that blank line needs to be there!):   
  
Tim's Abundant and Lucky Numbers and Mersenne Primes   
  
3 is a Mersenne prime!   
7 is a Mersenne prime!   
31 is a Mersenne prime!   
127 is a Mersenne prime!   
1575 is abundantly lucky!   
2835 is abundantly lucky!   
3465 is abundantly lucky!   
4095 is abundantly lucky!   
4725 is abundantly lucky!   
6435 is abundantly lucky!   
7245 is abundantly lucky!   
8085 is abundantly lucky!   
8191 is a Mersenne prime!   
9135 is abundantly lucky!   
9555 is abundantly lucky!   
12285 is abundantly lucky!   
  
Once you're finished and your program runs successfully, copy the entire job output to your *ASSIGN2.WORK* data set using XDC, naming it LUCKOUT. You may notice, when viewing LUCKOUT in *ASSIGN2.WORK*, that some lines (including the blank line that you added) include a zero, one, or dash in the first column. This is expected - you can leave them there.

## 9. C Programming and JCL (10 Marks)

**Background:** Typically, mainframers will export their JCL output into data set for future reference. This is especially true when writing C programs. In this challenge, you will redirect output from the last challenge into a data set for easy viewing.

**Your challenge:** Modify your C programming challenge source and make changes to your JCL to direct the output to a member in a data set.

**Hint:** Look at member IOCODE in your ASSIGN2.HINTS data set for some code to help with this challenge.  
  
You may have noticed in the previous challenge that the program produced a new member in your *KC03###.ASSIGN2.*OUTPUT data set called EXAMPLE. If you take another look at your CRUN JCL in your *KC03###*.ASSIGN2.JCL data set you'll see the DD statement that produced it. It's called OUTPUT1.   
  
Take a look back in *KC03###.ASSIGN2.*SOURCE(LUCKSRC) and search for OUTPUT1. You'll see that DDNAME OUTPUT1 is opened in the source code as luckyFile. If you search on luckyFile you'll see that the file is declared, opened, written to, and closed. Simple as that! Pay special attention to the manner in which the file is written to, and how it differs from the printf statements you used to write your abundantly lucky and Mersenne prime output, which were simply directed to standard output.   
  
Your assignment is to direct the output from this program to two new members in your *ASSIGN2.WORK* data set. For this challenge, all abundantly lucky numbers should be written to *ASSIGN2.WORK*(ABULUCKY) and all Mersenne primes should be written to *ASSIGN2.WORK*(MERPRIME).   
  
Let's review the steps needed to complete this challenge:

1. Declare two new files in your source code.
2. Open the two new files, and terminate the program if there is an error opening either of them. Take a look at how this is done for luckyFile. Take note of the two new DDNames you'll need to include in your JCL! You may assign whatever name you would like to them.
3. Change the printf statements that currently write abundantly lucky numbers and Mersenne primes to fwrite statements. Be sure to copy the line to fill the buffer with the data you want to write and the line to write the contents of the buffer to the correct output file!
4. Create 2 header lines for each of the new files:   
     
   - Abundant and Lucky Member
   * Your Name's Abundant and Lucky Numbers
   * A blank line

- Mersenne Prime Member

* + Your Name's Mersenne Primes!
  + A blank line

1. Run CCMP to compile the changes to your source code. When finished, submit your CRUN job for processing.
2. Modify your CRUN JCL: you need to include DD statements for the two new DDNames that you added to your source code.

When you've got it working, navigate to your *ASSIGN2.WORK* data set, where you should see two new members: ABULUCKY and MERPRIME. The content of each should be exactly as follows (if your name is Tim):  
  
Tim's Abundant and Lucky Numbers   
  
1575 is abundantly lucky!   
2835 is abundantly lucky!   
3465 is abundantly lucky!   
4095 is abundantly lucky!   
4725 is abundantly lucky!   
6435 is abundantly lucky!   
7245 is abundantly lucky!   
8085 is abundantly lucky!   
9135 is abundantly lucky!   
9555 is abundantly lucky!   
12285 is abundantly lucky!   
  
-----------------------------------------   
  
Tim's Mersenne Primes!   
  
3 is a Mersenne prime!   
7 is a Mersenne prime!   
31 is a Mersenne prime!   
127 is a Mersenne prime!   
8191 is a Mersenne prime!   
  
Once you've verified that your program produces the correct output, copy your source code to your *ASSIGN2.WORK* data set as member LUCKSRC and copy your CRUN JCL to your *ASSIGN2.WORK* data set as member name LUCKRUN (leave the LUCKOUT output you copied to *ASSIGN2.WORK* for the previous challenge as-is. This output contains your abundantly lucky and Mersenne primes jumbled up in SYSPRINT).

## 10. Mainframe Adventures with JAVA! (10 Marks)

**Background:** The z/OS environment includes a [POSIX](http://en.wikipedia.org/wiki/POSIX)-compliant UNIX operating system as a core component. The inclusion of UNIX offers enhanced application programming and communication flexibility on the mainframe. One great use of this component is compiling and running JAVA programs interactively, though JAVA can also be run in batch through JCL.

**Your challenge:** Compile and execute a JAVA program interactively through OMVS.

**Hint:** Look at member JAVA in your ASSIGN2.HINTS data set for some code to help with this challenge.  
  
Several methods exist for interactively working with z/OS UNIX services and the z/OS UNIX shell prompt. In this challenge, you will do so using the built-in OMVS shell available through the ISPF command prompt. "OMVS" stands for "Open MVS" which was the original name given to z/OS UNIX when IBM was making many improvements to the mainframe operating system to enable it to run UNIX applications.   
  
To get to the OMVS shell, choose option 6 from the ISPF Primary Menu. At the command line, type OMVS and press enter. Wait for the *KC03###*:/u/*KC03###*:> prompt. Notice that your JAVA classpath is displayed /u/Java6\_64/J6.0\_64), as well as your working directory (u/*KC03###*). If you have experience with JAVA and/or UNIX, these terms should be familiar, if not, don't worry about it.   
  
From the UNIX command prompt enter:   
  
**cp '//ASSIGN2.source(Anagram)' Anagram.java**

This is one of the times in the mainframe world where case matters; make sure your big letters and little letters are where they need to be!   
  
This command copies a bit of source code from your MVS data set *KC03###.ASSIGN2.*SOURCE to your zFS file system working directory. Why does it need to be copied, you ask? z/OS supports MVS data sets (all the data sets, members, records, etc. that you have been working with so far, as well as other types). Two of the other types are specifically for z/OS Unix: HFS and zFS, whose contents are the hierarchical file systems used by z/OS Unix. What you've just done is copy a file to a z/OS Unix hierarchical zFS. Type in the **ls** command to list the files in your working directory. Anagram.java will be there. Go ahead and open it for editing:   
  
**oedit Anagram.java**   
  
When you've finished peeking, press F3 to return to OMVS. Now, compile the code:   
  
**javac Anagram.java**   
  
Assuming you got everything entered right, your UNIX command prompt should come back to you, waiting for the next command.

If you receive the following error: javac: FSUM7351 not found, try entering:

**export PATH=/usr/lpp/java/J7.0\_64/bin:$PATH**

and re-issue the **javac Anagram.java .**  
  
If you see the following error, you'll have to exit TSO and up your virtual storage:   
  
**JVMJ9VM011W Unable to load j9jit29: EDC5204E Not enough storage to load DLL module.**

**JVMJ9GC020E -Xms too large for heap**

**JVMJ9VM015W Initialization error for library j9gc29(2): Failed to initialize**

**Error: Could not create the Java Virtual Machine.**

To up your virtual storage, exit ISPF, then from TSO READY prompt enter LOGON (which will logoff your userid), then logon again - but this time before you hit enter, change the value in the "Size" field to 1000000 (which represents 1000000KB, 1000MB or 1GB of virtual storage).   
  
When you have successfully compiled the program, **ls** will show you that Anagram.class now exists in your working directory. Go ahead and run it:   
  
**java Anagram '//KC02315.ASSIGN2.DATA(ANAGRAM)'**

This command invokes the Java program you just compiled, passing in one input argument, which is a path to a member in a data set. This member will be read, one record at a time, and processed by the program. The program will attempt to determine if the two strings present in each record are anagrams, that is the letters found in the first can be rearranged to produce the second, using each letter exactly once. The result of each is displayed right in OMVS.   
  
Notice any discrepancies in the output this program displays? Some records that appear to be anagrams are not properly labeled! Your challenge, then, is to correct this program. You must make sure your program does the following:

* Ignore case: Case and Aces are anagrams, though the first has a capital C and the second has a capital A
* Ignore punctuation: the following characters should be ignored: . , ' ? ; !

Got it? Once you've modified your source code, repeat the compile step and run it again. Things looking better? **Hint:** If your program is working correctly, it will report that every record in the input file contains a pair of anagrams.   
  
Once you have things working, copy your modified source code to your *ASSIGN2.WORK* data set:   
 **cp Anagram.java '//*ASSIGN2.WORK*(ANAGRAM)'**   
  
Your final task is to save a copy of your program's output:

**java Anagram '//KC02315.ASSIGN2.DATA(ANAGRAM)' >> javaout**

(This sends the program output to a file in the local zFS file system.)

**cp javaout '//*ASSIGN2.WORK*(JAVAOUT)'**

(This copies the file from the zFS file system back over to the MVS file system, specifically to your *ASSIGN2.WORK* data set. Ignore any truncation warnings you see - it's okay if the longest lines get cut off.)

When you've finished up in OMVS, type **exit** then hit enter to be returned back to ISPF. Navigate to your *ASSIGN2.WORK* data set and verify that you've got two new members there: ANAGRAM contains your modified Java source code and JAVAOUT contains a copy of the output of your Java program.

If you changed your virtual storage to 1000000, log out of TSO, and log in again: but this time set the "Size" field back to 32786.

***Congratulations you have completed Assignment 2!***

## 11. Submitting Assignment 2 (3 Marks)

If you followed all the steps correctly, you should have 16 members in your *KC03###*.*ASSIGN2.WORK* data set:

1. ABULUCKY
2. ANAGRAM
3. EBCDIC
4. FIXEDJCL
5. ISR@PRIM
6. JAVAOUT
7. LUCKOUT
8. LUCKRUN
9. LUCKSRC
10. MERPRIME
11. NEWMEMBR
12. REXXSYSC
13. RUNREXX
14. SYSCMDA
15. SYSCMDS
16. SYSCPAN

Follow the instructions below to submit your assignment.

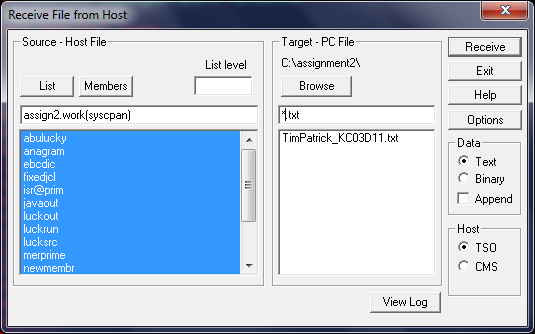
Create a new folder on your C: drive named; **assignment2**. In the folder add a new empty text file named; ***yourname\_yourid*.TXT** (e.g. TimPatrick\_KC03D11.TXT).

1. As you worked through each challenge of assignment 2, you added members to a PDS named ***yourid.ASSIGN2.WORK***. It should have 16 members when you finish everything.
2. When you are finished the steps of assignment 2, I want you to use the “Tranfer” menu of the Vista TN3270 Emulator to download the contents of this PDS to your c**:\assignment2** folder. See the example below for instructions on downloading a PDS from a mainframe to your PC using the Vista TN3270 Emulator.
3. Zip all the files in your c**:\assignment2** folder into a file called ***your\_name\_assignment#2.zip***, and submit the zip file to me via Blackboard.

**Downloading a mainframe PDS to your PC using the Vista TN3270 Emulator.**

Here’s an example of downloading a mainframe PDS to your PC.

1. Logon to TSO on the Marist mainframe. Note: The transfer is best done from native TSO, so exit ISPF if you are currently in it.
2. Select the “Receive from Host” menu item from the “Transfer” menu on your Vista TN3270 Emulator.
3. In the “Source – From Host” entry field, enter: 'yourid.ASSIGN2.WORK' and click the “members” button.
4. Select all the members in the list that appears by clicking on the first member in the list and then clicking on the last member of the list while holding down the shift button. All members should be highlighted.
5. Under “Target – PC File”, use the “Browse” button to find and select c:\assignment2\yourname\_yourid.txt. In the entry field change yourname\_yourid.txt to an asterisk (\*), make sure “Text” data and “TSO” host are selected and finally click on the “Receive” button. The transfer will take a minute or two.

**