1. **Does one algorithm perform better than the other? For a given reference-string size, what is the optimal number of page frames?**
   1. LRU generally performs better than FIFO.
   2. If there are less page frames than the reference string size, there will almost never be page faults.
2. **System uses: 32-bit addresses, 8kB per page, 128MB total memory**
   1. 32-bit address means 2­32 bytes. 8K bytes (213 bytes) per page means there is 219 addressable pages.
   2. Each page takes up 8K (213) bytes of memory. The system requires 13 bits to maintain the displacement. d = 13
   3. There is 219 total pages, meaning the system requires 19 bits to maintain the page number. p = 19
3. **Pure paging system uses 3 levels of page tables with 64-bit addresses. Virtual Address is *v = (p, m, t, d)* where *(p, m, t)* is the page number and *d* is the displacement into the page. np = number of bits to store p, nm is the number of bits to store m, and n­t is the number of bits to store t.**
   1. *Assume np = nm = n­t = 18.*
      1. Each level of the table is 18 bits long. A table, therefore, points to 218 (256K) pages. 64-bit addresses mean 8 bytes per address. Thus, the minimum size is 256 8 = 2048K bytes = 2Mbytes per table.
      2. Page number is (np + nm + n­t) = 54 bites. Each address is 64 bits meaning the displacement is 10 bits. Page size is 210 bytes or 1Kbyte.
   2. *Assume np = nm = n­t = 14.*
      1. Each level of the table is 14 bits long. A table, therefore, points to 214 (16K) pages. 64-bit addresses mean 8 bytes per address. Thus, the minimum size is 16 8 = 128Kbytes per table.
      2. Page number is (np + nm + n­t) = 42 bites. Each address is 64 bits meaning the displacement is 22 bits. Page size is 222 bytes or 4096K bytes or 4Mbytes.
   3. Large table sizes allow more information to be stored on each page but require more memory to implement. Small table sizes require small amounts of memory but can’t store as much information
4. **Discuss how fragmentation manifests itself in each of the following types of virtual memory systems.**
   1. *Segmentation-* fragmentation can occur when there is not enough contiguous memory available to the operating system to fully access the segment of memory
   2. *Paging-* process allocation requires an exact integer value when determining the number of pages to create, while processes rarely require an exact number of pages. As a result, fragmentation will almost always occur with the last page in the set, when the program does not actually require the full last page in order to run
   3. *Combined segmentation/paging-* this has the lowest amount of fragmentation since it can only occur during the segment transfer between the primary and secondary storages. Additionally, segments do not have to be contiguous so the memory. Will almost always not be fragmented.
5. Since the runtime almost does not change beyond the dial values, the process must only require about 5 page-frames in order to fully execute.

*/\*\*  
 \* David Jefts  
 \* Operating Systems - CS420  
 \*  
 \* Least Recently Used (L.R.U.) algorithm for page replacement  
 \* Simulates a page replacement function in an operating system  
 \*  
 \* This class is automatically called using the Test class file  
 \* Usage:  
 \* LRU lru = new LRU(numberOfPageFrames);  
 \* lru.insert(pageFrameNumber);  
 \*/***import** java.util.ArrayList;  
  
**public class** LRU **extends** ReplacementAlgorithm {  
 **int**[] **uses** = **new int**[PageGenerator.*getRANGE*() + 1];  
   
 **public** LRU(**int** pageFrameCount) {  
 **super**(pageFrameCount);  
 }  
   
 @Override  
 **public void** insert(**int** pageNumber) {  
 **if**(pageNumber < 0) **return**; *//kill method if pageNumber is invalid* PageGenerator gen = **new** PageGenerator(**pageFrameCount**);  
 **int**[] ref = gen.getReferenceString();  
 **uses**[pageNumber]++;  
   
 *//search entire list for the new pageNumber* **boolean** inPage = **false**;  
 **for**(**int** i = 0; i < ref.**length**; i++) {  
 **if**(pageNumber == ref[i]) {  
 inPage = **true**;  
 **break**;  
 }  
 }  
   
 *//insert the page if not already existing in ref, then increase pageFaultCount* **if**(!inPage) {  
 actualInsert(pageNumber, ref);  
 }  
 }  
   
 **public void** actualInsert(**int** pageNumber, **int**[] ref) {  
 System.***out***.println(**"LRU inserting page "** + pageNumber);  
   
 *//find all pages with least num uses* **int** minUses = 0; *//least calls on a given page* ArrayList<Integer> pageReplace = **new** ArrayList<>(); *//list of pages with low number of calls* **for**(**int** i = 0; i < **uses**.**length**; i++) {  
 **if**(**uses**[i] <= minUses) {  
 pageReplace.add(i);  
 minUses = **uses**[i];  
 }  
 }  
   
 *//clean up pageReplace* **int** x = minUses;  
 pageReplace.removeIf(i -> i < x);  
   
 *//replace ref[pageReplace] with pageNumber* **for**(**int** i = 0; i < ref.**length**; i++) {  
 **if**(pageReplace.contains(ref[i])) {  
 ref[i] = pageNumber;  
 **break**;  
 }  
 }  
 **pageFaultCount**++;  
 }  
}

I added the getRANGE method so that I could create a counter array with a length equal to the maximum possible value, in case RANGE is changed during testing.

*/\*\*  
 \* David Jefts  
 \* Operating Systems - CS420  
 \*  
 \* First In First Out (F.I.F.O.) algorithm for page replacement  
 \* Simulates a page replacement function in an operating system  
 \*  
 \* This class is automatically called using the Test class file  
 \* Usage:  
 \* FIFO fifo = new FIFO(numberOfPageFrames);  
 \* fifo.insert(pageFrameNumber);  
 \*/***public class** FIFO **extends** ReplacementAlgorithm {  
   
 **public static void** main(String[] args) {  
 FIFO fifo = **new** FIFO(7);  
 fifo.insert(4);  
 }  
   
 **public** FIFO(**int** pageFrameCount) {  
 **super**(pageFrameCount);  
 }  
   
 @Override  
 **public void** insert(**int** pageNumber) {  
 **if**(pageNumber < 0) **return**; *//kill method if pageNumber is invalid* PageGenerator gen = **new** PageGenerator(**pageFrameCount**);  
 **int**[] ref = gen.getReferenceString();  
   
 *//search entire list for the new pageNumber* **boolean** inPage = **false**;  
 **for**(**int** i = 0; i < ref.**length**; i++) {  
 **if**(pageNumber == ref[i]) {  
 inPage = **true**;  
 }  
 }  
   
 *//insert the page if not already existing in ref, then increase pageFaultCount* **if**(!inPage) {  
 actualInsert(pageNumber, ref);  
 }  
 }  
   
 **public void** actualInsert(**int** pageNumber, **int**[] ref) {  
 System.***out***.println(**"FIFO inserting page "** + pageNumber);  
 **for**(**int** i = 0; i < ref.**length** - 1; i++) {  
 ref[i] = ref[i + 1];  
 }  
 ref[ref.**length** - 1] = pageNumber;  
 **pageFaultCount**++;  
 }  
}