CSE 520 Project3
Problem 1
i) in which scenario, LRU will perform usually better
When a cache miss happens, LRU policy replaces the least recently used cache blocks. So if the most recently used cache blocks can be re-referenced immediately after they are referenced the first time, LRU can work efficiently. LRU performs well when the program has particular data acess pattern. For example, rencency-friendly pattern and some mixed pattern. Besides, LRU performs well when the program has small data working set which fit in cache.
In summary, LRU works well with workloads with high data locality.

ii) scenarios where LRU will be inefficient
LRU works badly with applications whose rereferences only occur in the distant future. Such applications correspond to situations where the application working set is larger than the available cache or when a burst of references to non-temporal data discards the active working set from the cache. In both scenarios, LRU inefficiently utilizes the cache since newly inserted blocks have no temporal locality after insertion.

iii) how LRU is implemented in gem5. Note: You will find the cache replacement policy implementation in the following file in the gem5 source directory – /src/mem/cache/tags/fa_lru.cc

In gem5, the basic class of all policy uses a doubly linked list as an MRU(most recently used) chain, which has most recently used block at the head of the list and least recently used the block at the tail.
In the LRU policy file, three basic policies are implemented. First, when a new cache block is loaded, it is inserted in the list and set its last touch tick as the current tick. Second, when a cache hit happens, reset the last touch tick as the current tick. Third, when a cache miss happens, find the block with the smallest time tick which is the least recent use one and mark it as the "victim". Find the corresponding address in memory and write back its content to memory. Then update the "victim" to the desired block that causes the cache miss and inserts it to the list.

Problem 2
In which scenarios RRIP
– i) will perform like ideal replacement policy (i.e. performs very well) for given benchmark
RRIP uses M-bits per cache block to store one of 2^M possible Rereference Prediction Values. RRIP dynamically learns rereference information for each block in the cache access pattern. RRIP works better than LRU when accesses have a distant re-reference interval. Since RRIP is scan-resistance, we can assume that RRIP works better compared to LRU towards particular access pattern such as thrashing access pattern and some mixed pattern contain scan. So based on the code of benchmarks. We can assume that basicmath, FFT, and Qsort perform better with SRRIP.

ii) will be inefficient?
SPPIP works inefficiently when the working set is small and accesses have an immediate re-reference interval. So we can assume that SRRIP works inefficiently with recency-friendly

access pattern or some mixed pattern which contains immediate re-reference as the major part. And SRRIP performs poorly as the effective size of the cache is reduced due to high hysteresis. Besides, large RRPVs can be resistant to long scans, they can result in inefficient cache utilization when a cache block receives its last hit and the RRPV becomes zero. RRIP inefficiently utilizes the cache when the re-reference interval of all blocks is larger than the available cache. In such scenarios, RRIP causes cache thrashing and results in no cache hits.

iii) Explain logistics behind implementing 2-bit SRRIP block replacement policy in gem5. Please give an overview and describe that what will be the changes required in which source files.
The basic process of SRRIP-HP is:
When Cache Hit:
(i) set RRPV of the block to ‘0'
When Cache Miss:
(i) search for first ‘3' from left
(ii) if ‘3' found go to step (v)
(iii) increment all RRPVs
(iv) goto step (i)
This process is similar to the LRU replacement policy. The following changes should be implemented:
In LRU, we record the last access time for each block but in SRRIP, we need to record the corresponding 2-bit RRPV value for each block;
When a cache miss happens, LRU finds the smallest time tick and mark the corresponding block as "victim". But in SRRIP, we need to find the first "3" in cache blocks and mark it as "victim";
When no "3" found, the steps (iii) and (iv) should also be implemented.;
When we invalidate a block, LRU set the last touch tick as "0" to invalidate. But in SRRIP, we set the RRPV value to "3";
When we add a new block into the cache which is the call of "reset" function, LRU set the set the last touch tick as the current tick. But in SRRIP, we set the RRPV value to "2";
When a cache hit happens, LRU set the last touch tick as the current tick for corresponding block. SRRIP set the RRPV value as "0".
Modifications made on source file:
1. lru_rp.hh
a. Add a member "rrpv" as int type in struct LRUReplData : ReplacementData;
b. The constructor of LRUReplData initializes "rrpv" as "3";
2. lru_rp.cc
a. Function "invalidate": set "rrpv" as "3" for the given block;
b. Function "touch" (cache hit): set "rrpv" as "0" for the given block;
c. Function "reset" (load new block): set "rrpv" as "2" for the given block;
d. Function "getvitim" (cache miss): inumerate every block in cache to find the first "3" from left. If no "3" found, increase all "rrpv" of the blocks by 1 and restart this process.

Problem 3
1、LRU

	Execution time	Dcache miss rate	Icache miss rate	L2 Cache miss rate
dijkstra_small	125930108500	0.213919	0.032056	0.041918
dijkstra_large	534056362500	0.241900	0.016466	0.052903
basicmath_small	857348890500	0.063775	0.047154	0.295190
basicmath_large	9598601660500	0.058866	0.084389	0.061833
fft	231845477500	0.036073	0.065515	0.004953
qsort_small	52190265500	0.051248	0.065470	0.133840

2 .SRRIP

	Execution time	Dcache miss rate	Icache miss rate	L2 Cache miss rate
dijkstra_small	126123614500	0.214443	0.032056	0.042328
dijkstra_large	535418514500	0.242733	0.016466	0.053524
basicmath_small	791430861500	0.063875	0.047154	0.235513
basicmath_large	9223844254500	0.056395	0.084389	0.046179
fft	231493352500	0.034426	0.065515	0.004996
qsort_small	51978605500	0.050323	0.065470	0.132776

* The data marked green represents better performance.

From the table, we can conclude that in general, a lower miss rate will lead to a less execution time. For both policy, if it can cause a lower miss rate, the corresponding execution time will also be shorter.

From the data, we can tell that Dijkstra algorithm has a better performance when the cache replacement policy is LRU. The data access pattern for this algorithm is mixed access pattern. Because the algorithm accesses the adjacency list with an immediate re-reference interval. But the access to the matrix of nodes satisfies streaming access pattern. We can compute the size of the adjacency list from the original code of Dijkstra. For "dijkstra_small", the size of the matrix of nodes is 1.2kB which is much the same as L1 dcache size. So the dcache miss rate is less than SRRIP's miss rate. This also result in lower L2 cache miss rate. The icache miss rate does not influenced by the replacement policy so it has no difference. LRU works very well if the frequently accessed part of working set fits the cache size. Overall the algorithm has a better performance when using LRU policy.

"basicmath", "fft" and "qsort" all have a better performance with SRRIP. Based on the code of these three benchmarks, "fft" accords with streaming access pattern and its pattern can be also conclude as scan. Since SRRIP is scan-resistance and it can perform much better than LRU when working with scan. Besides, via the code, we can compute the size of the working set. The size of the working set is 4096*4(bytes) which is much bigger than the cache size. With this factor, SRRIP can not optimize this case too much because when the

working set and cache size differ greatly, both replacemnet policy can not work very efficiently.
"qsort" and "basicmath" both accord with thrashing access pattern. These two kinds of patterns all perform well when the policy is SRRIP because they all access data in a distant interval. Besides, for thrashing access pattern, two policies have close performance because working set is accessed in a particular order and no re-reference exist in this pattern. “qsort” can also be conclude as scan. So as said above, SRRIP is scan-resistance, so it can perform better than LRU in this case. We can compute this working set as 128*10000(bytes) which ia much bigger than the cache size as well. So in this case, SRRIP can not optimize this case too much. While the working set of “basicmath_small” has very small size which is smaller than cache size. So for both policies, the miss rate can maintain a low level.