SANTA CLARA UNIVERSITY

Caching Lab

Rick Sullivan

November 20, 2014

1 LEAST RECENTLY USED

The least recently used algorithm maintains an ordered list of pages currently in memory. My source code can be found in the Appendix section 5.1.3. My algorithm iterates through the list of pages, looking for a match of the requested page ID.
If the page is found, it is removed from that position in the list, and reinserted at the head of the list. If it is not found, however, we remove the last page in the list, because that must be the least recently used of all pages in memory. The new page is then likewise inserted at the head of the list.
The disadvantage of LRU lies in the fact that you must iterate through the list of pages; traversal through a linked-list or deuque is O(n) and will be slow on large lists. As shown on the example in class (Figure 1.1), LRU can avoid a large number of cache misses because it accommodates the behavior of real-world programs.

TESTING LRU Using the input from in class, LRU results in only 11 cache misses when using a page table size of four. In my automated testing, I only found that LRU generated slightly lower hit rates than my other tested algorithms. This is likely due to the fact that my test inputs were still very random, so semi-intelligent page selection does not help all that much. The most interesting test case is my third test, where one page is requested 25% of the time. LRU consistently has a miss rate trailing only LFU. LFU performs exceptionally well on this test only because it is almost guaranteed to never kick a popular page out of memory.

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 1 3 4 1

Figure 1.1: Input given in the class slides to differentiate page replacement algorithms.

2 Least Frequently Used

The least frequently used algorithm maintains counts of how many times each page has been accessed. It does not maintain any ordering of the pages themselves. At the time of an interrupt, LFU updates the count of how many times each page has been accessed. It then removes the page with the lowest access count when a page needs to be replaced. My LFU implementation can be found in section 5.1.4.

For my implementation, I created a `pageTableScan` function that increments the counter attributes of each page that has been referenced in the time since the function was last called. Because simulating an interrupt is relatively complicated, I simply update the page counters whenever we have a page fault.

When a page fault occurs, after updating page reference counters, I then traverse the list, looking for the Page object with the smallest count value. This least-frequently referenced page is then replaced with the new page.

This implementation requires list traversal for both reference updating and for finding the least frequently used page. This could likely be improved by maintaining pages in a binary tree or min heap. However, LFU by itself is undesirable because it has no method of removing pages that have a high reference count. If a page is referenced many times in a short period of time, it may never actually be removed from memory, even if it is never referenced again.

Testing LFU As expected, LFU is unexceptional in all tests except for my third test, where one page is accessed 25% of the time. Because LFU is unlikely to ever remove a frequently-used page from memory, that page will remain in memory throughout the test. This is unrealistic, as real-world programs will access pages in a certain range repeatedly, but then move on to other locations.

Using the class slide input (Figure 1.1), we see that LFU registers 13 page faults; LFU is slightly less successful than LRU, although this example is created to show the strength of LRU.

3 Second Chance

The second chance algorithm is very similar to a plain FIFO queue implementation. However, as the name implies, this algorithm will consider a page at least twice before it is replaced. This is accomplished by maintaining a circular list. Each page then maintains a flag to indicate whether or not it has been referenced. On traversing the list, the page replacement algorithm will replace any page that has not been referenced. Any referenced page it considers has its referenced flag set to false, as well.

For my implementation, I used a vector data structure as a circular list. I maintain an index integer; as soon as the index overflows the boundary of the array, it is reset to zero, and the program continues. New pages are inserted wherever the first non-referenced page is found. My implementation has a linear search time for pages in the page table. Page replacement itself is also linear in time.

TESTING SECOND CHANCE Like LRU, the second chance algorithm does not distinguish itself in my generated semi-random tests. Using the slides' example (Figure 1.1), second chance misses 12 page requests. This is in-between the results for LRU and LFU.

4 RANDOM

Randomly selecting pages for replacement has clear benefits and disadvantages. It does not require any overhead involved in tracking if or how many times a page has been referenced. However, it is also not able to make intelligent assumptions about the behavior of real-world programs. Many complex page replacement algorithms fall back on a random choice in certain cases because of its simplicity.
My implementation of random page replacement is very simple: I generate an index within the bounds of the page table's size, and evict the corresponding page.

TESTING SECOND CHANCE This random algorithm is on par with all other algorithms if page accesses are uniformly accessed, which is to be expected. However, it does not improve when the data is increasingly biased with a triangular distribution and my "One Page" example. At those points, more directed algorithms show cache-hit improvements, while the random algorithm does not.
Using the input from Figure 1.1, this algorithm's number of page faults fluctuates between 11 and 15. If a workflow is difficult to predict, the random algorithm could be useful because of its lack of overhead.

5 IMPROVING PERFORMANCE

Each of my implementations for this lab used a vector or a deque: data structures that offer O(n) searching for elements. Time complexities could be improved by using a hash map to store the pages in memory. This would allow us to lookup page IDs in constant time. This does come with a space requirement tradeoff, however (although it is still O(n)). Another option would be to use a binary tree to maintain pages sorted by their ID. Lookup would then be O(logn).

6 APPENDIX

6.1 SOURCE CODE

This appendix section contains the sources used for implementing each page replacement algorithm. Each algorithm uses the Page class, which is shown in this section. Simple STL data structures were used to implement each algorithm. Stack and array-based algorithms were implemented using std::vector, while queue-based algorithms were implemented using std::deque.

6.1.1 PAGE.H

```
#ifndef __Page_H
#define __Page_H

#include <vector>

class Page {
public:
    int id;
    bool referenced;
    int counter;

    Page();
    Page(int id);
    Page &operator=(const Page &other);
};

/* Naively implementing table as an array for simplicity. */
bool tableContainsPage(const std::vector<Page> &pages, const int pageID)
    ;

/* Used for debugging. */
void printPages(const std::vector<Page> &pages);

#endif
```

6.1.2 PAGE.CPP

```
#include <iostream>
#include <string>
#include <vector>
#include <cstdlib>

#include "Page.h"
```

```
/* Simple class to encapsulate page IDs and referenced bits. */
Page::Page() {
    /* Invalid page ID. */
    this->id = -1;
    /* Invalid counter. */
    this->counter = -1;
    this->referenced = false;
}

Page::Page(int id) {
    this->id = id;
    this->counter = 0;
    this->referenced = true;
}

Page& Page::operator=(const Page &other) {
    if (this == &other)
        return *this;

    this->id = other.id;
    this->counter = other.counter;
    this->referenced = other.referenced;

    return *this;
}

/* Naively implementing table as an array for simplicity. */
bool tableContainsPage(const std::vector<Page> &pages, const int pageID)
    {
    for (int j = 0; j < pages.size(); j++) {
        if (pages[j].id == pageID)
            return true;
    }

    return false;
}

/* Used for debugging. */
void printPages(const std::vector<Page> &pages) {
    std::cout << "Pages:\t";
    for (int i = 0; i < pages.size(); i++) {
        std::cout << pages[i].id << '\t';
    }
    std::cout << std::endl;
}
```

6.1.3 Least Recently Used

```
#include <iostream>
#include <string>
#include <deque>
#include <cstdlib>

#include "Page.h"

/* Simulates the least recently used page replacement algorithm.
 * Uses a deque to maintain the queue of pages, with the most
 * recently used page at the front of the deque.
 * */
void leastRecentlyUsed(int numPFrames) {
    std::deque<Page> pages(numPFrames);

    int pageID;
    while(std::cin >> pageID) {
        bool found = false;
        for (int i = 0; i < pages.size(); i++) {
            if (pages[i].id == pageID) {
                /* If we found the page, move it to the front of the
                   queue. */
                pages.erase(pages.begin() + i);
                Page newPage(pageID);
                pages.push_front(newPage);
                found = true;
                break;
            }
        }

        if (found)
            continue;

        /* Else we have a page fault. */
        std::cout << pageID << std::endl;

        /* If the page table is full, get rid of the last page. */
        if (pages.size() == numPFrames)
            pages.pop_back();

        /* Add to front of queue. */
        Page newPage(pageID);
        pages.push_front(newPage);
```

```
            //std::cout << "Adding page " << newPage.id << std::endl;
            //printPages(pages);
        }
}

int main(int argc, char **argv) {
    /* Should be given a single command-line argument for the number of
       available page frames. */
    if (argc != 2) {
        std::cout << "Provide␣a␣single␣command-line␣argument␣for␣the␣
            number␣of␣page␣frames␣available." << std::endl;
        return 0;
    }

    int numPFrames = atoi(argv[1]);

    leastRecentlyUsed(numPFrames);
}
```

6.1.4 Least Frequently Used

```
#include <iostream>
#include <string>
#include <deque>
#include <cstdlib>
#include <climits>

#include "Page.h"

/* Checks for referenced pages and increments counters of
    referenced pages. */
void pageTableScan(std::vector<Page> &pages) {
    for (int i = 0; i < pages.size(); i++) {
        if (pages[i].referenced) {
            pages[i].referenced = false;
            pages[i].counter++;
        }
    }
}

/* Simulates the least frequently used page replacement algorithm.
 * Uses a vector to maintain the unordered list of pages.
 * */
void leastFrequentlyUsed(int numPFrames) {
    std::vector<Page> pages(numPFrames);

    int pageID;
    while(std::cin >> pageID) {
        bool found = false;
        for (int i = 0; i < pages.size(); i++) {
            if (pages[i].id == pageID) {
                /* If we found the page, set it as referenced. */
                pages[i].referenced = true;
                found = true;
            }
        }

        if (found)
            continue;

        /* Else we have a page fault. */
        std::cout << pageID << std::endl;
        pageTableScan(pages);

        Page newPage(pageID);
        /* If the page table is full, get rid of the LFU page. */
        if (pages.size() == numPFrames) {
            int minCount = INT_MAX;
            int minIndex = 0;
            for (int i = 0; i < pages.size(); i++) {
                if (pages[i].counter < minCount) {
                    minCount = pages[i].counter;
                    minIndex = i;
                }
            }

            pages[minIndex] = newPage;
        } else {
            /* Table is not full. Just add page. */
            pages.push_back(newPage);
        }

        //std::cout << "Adding page " << newPage.id << std::endl;
        //printPages(pages);
    }
}

int main(int argc, char **argv) {
    /* Should be given a single command-line argument for the number of
       available page frames. */
    if (argc != 2) {
        std::cout << "Provide␣a␣single␣command-line␣argument␣for␣the␣
            number␣of␣page␣frames␣available." << std::endl;
        return 0;
    }

    int numPFrames = atoi(argv[1]);

    leastFrequentlyUsed(numPFrames);
}
```

6.1.5 Second Chance

```
#include <iostream>
#include <string>
#include <vector>
#include <cstdlib>

#include "Page.h"

/* Simulates the second-change page replacement algorithm.
 * Uses a vector as a circular queue.
 * */
void secondChance(int numPFrames) {
    std::vector<Page> pages(numPFrames);

    int pageID, i = 0;
    while(std::cin >> pageID) {
        if (tableContainsPage(pages, pageID))
            continue;

        /* Else we have a page fault, so we want to replace a page. */
        std::cout << pageID << std::endl;

        while(pages[i].referenced == true) {
            /* Set referenced value to false. */
            pages[i].referenced = false;
            i = (i+1) % numPFrames;
        }

        /* Iterator is pointing to a non-referenced page. */
        Page newPage(pageID);
        pages[i] = newPage;
        //std::cout << "Adding page " << newPage.id << std::endl;
        //printPages(pages);
    }
}

int main(int argc, char **argv) {
    /* Should be given a single command-line argument for the number of
        available page frames. */
    if (argc != 2) {
        std::cout << "Provide␣a␣single␣command-line␣argument␣for␣the␣
            number␣of␣page␣frames␣available." << std::endl;
        return 0;
    }

    int numPFrames = atoi(argv[1]);

    secondChance(numPFrames);
}
```

6.1.6 Random

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

#include "Page.h"

/* Simulates a random page replacement algorithm.
 * Uses a vector to maintain the unordered list of pages.
 * */
void random(int numPFrames) {
    std::vector<Page> pages(numPFrames);

    int pageID;
    while(std::cin >> pageID) {
        /* Do nothing if page is found. */
        if (tableContainsPage(pages, pageID))
            continue;

        /* Else we have a page fault. */
        std::cout << pageID << std::endl;

        Page newPage(pageID);
        int i = rand() % numPFrames;
        pages[i] = newPage;

        //std::cout << "Adding page " << newPage.id << std::endl;
        //printPages(pages);
    }
}

int main(int argc, char **argv) {
    srand(time(NULL));
    /* Should be given a single command-line argument for the number of
       available page frames. */
    if (argc != 2) {
        std::cout << "Provide a single command-line argument for the 
            number of page frames available." << std::endl;
        return 0;
    }

    int numPFrames = atoi(argv[1]);
    random(numPFrames);
}
```

6.2 Testing

I automated some testing with Python scripts. I created one script to automatically create semi-random input files. The biased input generating script takes two parameters; one that lets you choose the total number of lines in the input file and one that specifies the range of possible page IDs to be generated in the files.
For testing, I tried different combinations for input type choices, along with varying page table sizes. Test outputs are tables with the number of input elements shown on the X-axis and the range of possible IDs on the Y-axis. Different table sizes are shown in separate tables. Each test has the number of misses shown, followed by the percent of requests missed.
As the page table size gets as big as the number of possible IDs in the input, misses fall to only first-time cache misses.

6.2.1 Biased input generation

```python
import sys
import random
import math

NUM_INPUT_LINES = 500
ID_RANGE = 50


def writeInputFiles(numInputLines=NUM_INPUT_LINES, idRange=ID_RANGE):
    f = open("inputs/uniform.txt", 'w')
    for i in range(0, numInputLines):
        r = random.randint(0, idRange - 1)
        f.write(str(r) + '\n')

    f = open("inputs/triangular.txt", 'w')
    for i in range(0, numInputLines):
        r = int(random.triangular(0, idRange - 1))
        f.write(str(r) + '\n')

    f = open("inputs/onePage.txt", 'w')
    for i in range(0, numInputLines):
        r = random.randint(0, idRange - 1)
        # 25% of the time, request the same page
        r2 = random.randint(0, 4)
        if r2 == 0:
            r = 0
        f.write(str(r) + '\n')

writeInputFiles(NUM_INPUT_LINES, ID_RANGE)
```

6.2.2 Test automation and output formatting

```
import os
import subprocess

from createInputs import *

progs = ['lru', 'lfu', 'secondChance', 'random']
inputDir = 'inputs/'
outputDir = 'outputs/'
tableSizes = ['10', '100', '500']
inputSizes = ['500', '1000', '2000']
idRanges = ['50', '200', '1000']
for fileName in os.listdir(inputDir):
    fout = open(outputDir + fileName, 'w')
    for tableSize in tableSizes:
        fout.write('Table␣size:␣' + tableSize + '\n')
        fout.write('X=#␣of␣elements␣in␣input␣files.␣Y=Range␣of␣page␣IDs␣
            in␣input␣files.\n\n')

        counts = []
        for inputSize in inputSizes:
            row = []
            for idRange in idRanges:
                writeInputFiles(int(inputSize), int(idRange))
                element = {}

                for prog in progs:
                    cmd = 'cat␣' + inputDir + fileName + '|␣./' + prog +
                         '␣' + tableSize + '␣|␣wc␣-l'
                    p = subprocess.Popen(cmd, shell=True, stdout=
                        subprocess.PIPE)
                    output, errors = p.communicate()
                    element[prog] = int(output)

                row.append(element)
            counts.append(row)

        colWidth = 12
        for inputSize in inputSizes:
            fout.write('␣' + inputSize.ljust(colWidth) + '␣')
        fout.write('\n')

        for i in range(len(idRanges)):
```

```
    fout.write(idRanges[i] + '\n')
    for prog in progs:
        for j in range(len(inputSizes)):
            numMissed = counts[j][i][prog]
            percentMissed = (int(numMissed) * 100) / int(
                inputSizes[j])
            content = prog[0:3] + ': ' + str(numMissed) + ' ' +
                str(percentMissed) + '%'
            fout.write(' ' + content.ljust(colWidth) + ' |')
        fout.write('\n')
    fout.write('\n')

fout.write('\n')
```

6.2.3 Test results: uniform distribution

```
Table size: 10
X=# of elements in input files. Y=Range of page IDs in input
   files.

 500           1000          2000
50
 lru: 409 81% | lru: 792 79% | lru: 1592 79% |
 lfu: 390 78% | lfu: 799 79% | lfu: 1615 80% |
 sec: 406 81% | sec: 788 78% | sec: 1594 79% |
 ran: 404 80% | ran: 799 79% | ran: 1595 79% |

200
 lru: 472 94% | lru: 948 94% | lru: 1894 94% |
 lfu: 478 95% | lfu: 955 95% | lfu: 1901 95% |
 sec: 471 94% | sec: 948 94% | sec: 1895 94% |
 ran: 468 93% | ran: 955 95% | ran: 1897 94% |

1000
 lru: 494 98% | lru: 989 98% | lru: 1985 99% |
 lfu: 491 98% | lfu: 988 98% | lfu: 1986 99% |
 sec: 494 98% | sec: 989 98% | sec: 1985 99% |
 ran: 492 98% | ran: 990 99% | ran: 1982 99% |


Table size: 100
X=# of elements in input files. Y=Range of page IDs in input
   files.

 500           1000          2000
50
 lru: 50 10%  | lru: 50 5%   | lru: 50 2%   |
 lfu: 50 10%  | lfu: 50 5%   | lfu: 50 2%   |
 sec: 50 10%  | sec: 50 5%   | sec: 50 2%   |
 ran: 66 13%  | ran: 66 6%   | ran: 66 3%   |

200
 lru: 293 58% | lru: 524 52% | lru: 1033 51% |
 lfu: 299 59% | lfu: 521 52% | lfu: 1026 51% |
 sec: 298 59% | sec: 527 52% | sec: 1014 50% |
 ran: 329 65% | ran: 539 53% | ran: 1078 53% |

1000
```

```
 lru: 462 92% | lru: 912 91% | lru: 1815 90% |
 lfu: 451 90% | lfu: 915 91% | lfu: 1783 89% |
 sec: 459 91% | sec: 913 91% | sec: 1820 91% |
 ran: 466 93% | ran: 919 91% | ran: 1825 91% |


Table size: 500
X=# of elements in input files. Y=Range of page IDs in input
   files.

 500           1000          2000
50
 lru: 50 10%  | lru: 50 5%   | lru: 50 2%   |
 lfu: 50 10%  | lfu: 50 5%   | lfu: 50 2%   |
 sec: 50 10%  | sec: 50 5%   | sec: 50 2%   |
 ran: 52 10%  | ran: 52 5%   | ran: 52 2%   |

200
 lru: 185 37% | lru: 199 19% | lru: 200 10% |
 lfu: 185 37% | lfu: 199 19% | lfu: 200 10% |
 sec: 185 37% | sec: 199 19% | sec: 200 10% |
 ran: 205 41% | ran: 240 24% | ran: 249 12% |

1000
 lru: 388 77% | lru: 632 63% | lru: 1160 58% |
 lfu: 388 77% | lfu: 634 63% | lfu: 1153 57% |
 sec: 388 77% | sec: 625 62% | sec: 1158 57% |
 ran: 413 82% | ran: 732 73% | ran: 1296 64% |
```

6.2.4 Test results: triangular distribution

```
Table size: 10
X=# of elements in input files. Y=Range of page IDs in input
   files.

 500           1000          2000
50
 lru: 351 70% | lru: 703 70% | lru: 1485 74% |
 lfu: 365 73% | lfu: 716 71% | lfu: 1507 75% |
 sec: 355 71% | sec: 715 71% | sec: 1494 74% |
 ran: 358 71% | ran: 719 71% | ran: 1499 74% |

200
 lru: 481 96% | lru: 939 93% | lru: 1853 92% |
 lfu: 466 93% | lfu: 922 92% | lfu: 1855 92% |
 sec: 481 96% | sec: 941 94% | sec: 1854 92% |
 ran: 475 95% | ran: 945 94% | ran: 1852 92% |

1000
 lru: 494 98% | lru: 989 98% | lru: 1974 98% |
 lfu: 492 98% | lfu: 981 98% | lfu: 1986 99% |
 sec: 494 98% | sec: 989 98% | sec: 1974 98% |
 ran: 495 99% | ran: 991 99% | ran: 1978 98% |


Table size: 100
X=# of elements in input files. Y=Range of page IDs in input
   files.

 500           1000          2000
50
 lru: 48 9%   | lru: 49 4%   | lru: 49 2%   |
 lfu: 48 9%   | lfu: 49 4%   | lfu: 49 2%   |
 sec: 48 9%   | sec: 49 4%   | sec: 49 2%   |
 ran: 59 11%  | ran: 60 6%   | ran: 60 3%   |

200
 lru: 224 44% | lru: 405 40% | lru: 797 39% |
 lfu: 204 40% | lfu: 415 41% | lfu: 768 38% |
 sec: 220 44% | sec: 426 42% | sec: 857 42% |
 ran: 267 53% | ran: 480 48% | ran: 918 45% |

1000
```

```
 lru: 430 86% | lru: 880 88% | lru: 1749 87% |
 lfu: 450 90% | lfu: 886 88% | lfu: 1752 87% |
 sec: 434 86% | sec: 882 88% | sec: 1751 87% |
 ran: 445 89% | ran: 896 89% | ran: 1776 88% |


Table size: 500
X=# of elements in input files. Y=Range of page IDs in input
   files.

 500           1000          2000
50
 lru: 46 9%   | lru: 49 4%   | lru: 49 2%   |
 lfu: 46 9%   | lfu: 49 4%   | lfu: 49 2%   |
 sec: 46 9%   | sec: 49 4%   | sec: 49 2%   |
 ran: 47 9%   | ran: 51 5%   | ran: 54 2%   |

200
 lru: 159 31% | lru: 178 17% | lru: 192 9%  |
 lfu: 159 31% | lfu: 178 17% | lfu: 192 9%  |
 sec: 159 31% | sec: 178 17% | sec: 192 9%  |
 ran: 174 34% | ran: 208 20% | ran: 247 12% |

1000
 lru: 380 76% | lru: 595 59% | lru: 963 48% |
 lfu: 380 76% | lfu: 598 59% | lfu: 961 48% |
 sec: 380 76% | sec: 597 59% | sec: 994 49% |
 ran: 410 82% | ran: 685 68% | ran: 1180 59% |
```

6.2.5 Test results: one page favored

```
Table size: 10
X=# of elements in input files. Y=Range of page IDs in input
   files.

 500           1000          2000
50
 lru: 339 67% | lru: 630 63% | lru: 1296 64% |
 lfu: 325 65% | lfu: 646 64% | lfu: 1297 64% |
 sec: 355 71% | sec: 667 66% | sec: 1369 68% |
 ran: 343 68% | ran: 664 66% | ran: 1368 68% |

200
 lru: 396 79% | lru: 764 76% | lru: 1536 76% |
 lfu: 378 75% | lfu: 749 74% | lfu: 1491 74% |
 sec: 410 82% | sec: 800 80% | sec: 1604 80% |
 ran: 410 82% | ran: 807 80% | ran: 1619 80% |

1000
 lru: 416 83% | lru: 800 80% | lru: 1609 80% |
 lfu: 405 81% | lfu: 785 78% | lfu: 1566 78% |
 sec: 432 86% | sec: 841 84% | sec: 1680 84% |
 ran: 433 86% | ran: 847 84% | ran: 1685 84% |


Table size: 100
X=# of elements in input files. Y=Range of page IDs in input
   files.

 500           1000          2000
50
 lru: 50 10%  | lru: 50 5%   | lru: 50 2%   |
 lfu: 50 10%  | lfu: 50 5%   | lfu: 50 2%   |
 sec: 50 10%  | sec: 50 5%   | sec: 50 2%   |
 ran: 64 12%  | ran: 66 6%   | ran: 66 3%   |

200
 lru: 238 47% | lru: 451 45% | lru: 832 41% |
 lfu: 234 46% | lfu: 418 41% | lfu: 822 41% |
 sec: 236 47% | sec: 444 44% | sec: 838 41% |
 ran: 268 53% | ran: 478 47% | ran: 867 43% |

1000
```

```
 lru: 366 73% | lru: 735 73% | lru: 1401 70% |
 lfu: 372 74% | lfu: 736 73% | lfu: 1422 71% |
 sec: 372 74% | sec: 742 74% | sec: 1407 70% |
 ran: 373 74% | ran: 737 73% | ran: 1441 72% |


Table size: 500
X=# of elements in input files. Y=Range of page IDs in input
   files.

 500           1000          2000
50
 lru: 50 10%  | lru: 50 5%   | lru: 50 2%   |
 lfu: 50 10%  | lfu: 50 5%   | lfu: 50 2%   |
 sec: 50 10%  | sec: 50 5%   | sec: 50 2%   |
 ran: 52 10%  | ran: 52 5%   | ran: 52 2%   |

200
 lru: 174 34% | lru: 198 19% | lru: 200 10% |
 lfu: 174 34% | lfu: 198 19% | lfu: 200 10% |
 sec: 174 34% | sec: 198 19% | sec: 200 10% |
 ran: 180 36% | ran: 227 22% | ran: 249 12% |

1000
 lru: 343 68% | lru: 560 56% | lru: 988 49% |
 lfu: 343 68% | lfu: 554 55% | lfu: 985 49% |
 sec: 343 68% | sec: 559 55% | sec: 989 49% |
 ran: 356 71% | ran: 620 62% | ran: 1100 55% |
```