Lab8
Scenario 1
1.What combination of parameters is producing the hit rate you observe? (Hint: Your answer should be of the form: "Because parameter A == parameter B")
- (0/16) 0% hit rate b/c step size

2.What is our hit rate if we increase Rep Count arbitrarily? Why?
- Doesn't change the hit rate // Miss count increases (conflict misses)

3.How could we modify our program parameters to maximize our hit rate?
- make step size to 1 -> improvs to 50%

Scenario 2
1. Explain the hit rate in terms of the parameters of the cache and the program.
- 75% hit rate (1 miss, 3 hits)

2. What happens to our hit rate as Rep Count goes to infinity? Why?
- hit rate improves, b/c all the elements are in the cache

3. Suppose we have a program that uses a very large array and during each Rep, we apply a different operator to the elements of our array (e.g. if Rep Count = 1024, we apply 1024 different operations to each of the array elements). How can we restructure our program to achieve a hit rate like that achieved in this scenario? (Assume that the number of operations we apply to each element is very large and that the result for each element can be computed independently of the other elements.) What is this technique called? (Hint)
- We can use blocking to keep data in our cache, so we don’t go to memory as often

Scenario 3:
1. Run the simulation a few times. Note that the hit rate is non-deterministic. Why is this the case? Explain in terms of the cache parameters and our access pattern. ("The program is accessing random indicies" is not a sufficient answer).
- The offset is chosen at a random range between 0 and step size, thus resulting in access of random index

Which Cache parameter can you modify in order to get a constant hit rate? Record the parameter and its value (and be prepared to show your TA a few runs of the simulation). How does this parameter allow us to get a constant hit rate?
- step size is 1, so there is only 1 “random” choice - so it converges to 88%

Exercise 2:

1. Which ordering(s) perform best for 1000-by-1000 matrices?
- jki, kji when i is in the inside

2. Which ordering(s) perform the worst?
kij, ikj

3. How does the way we stride through the matrices with respect to the innermost loop affect performance?
- i is a row therefore, we are taking advantage of spatial locality.

Exercise 3:
Part 1: Changing Array Sizes

```
~/labs/08 $ ./transpose 100 20
Testing naive transpose: 0.006 milliseconds
Testing transpose with blocking: 0.007 milliseconds

(00:54:38 Mon Nov 01 2016 cs61c-ajw@hive1 Linux x86_64)
~/labs/08 $ ./transpose 1000 20
Testing naive transpose: 0.972 milliseconds
Testing transpose with blocking: 1.159 milliseconds

(00:54:47 Mon Nov 01 2016 cs61c-ajw@hive1 Linux x86_64)
~/labs/08 $ ./transpose 2000 20
Testing naive transpose: 5.088 milliseconds
Testing transpose with blocking: 6.212 milliseconds

(00:54:56 Mon Nov 01 2016 cs61c-ajw@hive1 Linux x86_64)
~/labs/08 $ ./transpose 5000 20
Testing naive transpose: 52.51 milliseconds
Testing transpose with blocking: 42.597 milliseconds

(00:55:01 Mon Nov 01 2016 cs61c-ajw@hive1 Linux x86_64)
~/labs/08 $ ./transpose 10000 20
Testing naive transpose: 225.178 milliseconds
Testing transpose with blocking: 157.396 milliseconds
```

1. Fix the blocksize to be 20, and run your code with n equal to 100, 1000, 2000, 5000, and 10000. At what point does cache blocked version of transpose become faster than the non-cache blocked version? Why does cache blocking require the matrix to be a certain size before it outperforms the non-cache blocked code?

- when blocksize is 5000, it becomes faster. because after certain amount of n increases, as number of entries increases it goes down to reduced capacity and conflict misses and hit time increase by larger memory structures.

Part 2: Changing Blocksize

```
~/labs/08 $ ./transpose 10000 50
Testing naive transpose: 219.363 milliseconds
Testing transpose with blocking: 126.913 milliseconds

(00:58:36 Mon Nov 01 2016 cs61c-ajw@hive1 Linux x86_64)
~/labs/08 $ ./transpose 10000 100
Testing naive transpose: 219.313 milliseconds
Testing transpose with blocking: 119.631 milliseconds

(00:58:41 Mon Nov 01 2016 cs61c-ajw@hive1 Linux x86_64)
~/labs/08 $ ./transpose 10000 500
Testing naive transpose: 219.145 milliseconds
Testing transpose with blocking: 105.93 milliseconds

(00:58:50 Mon Nov 01 2016 cs61c-ajw@hive1 Linux x86_64)
~/labs/08 $ ./transpose 10000 1000
Testing naive transpose: 225.106 milliseconds
Testing transpose with blocking: 131.188 milliseconds

(00:58:56 Mon Nov 01 2016 cs61c-ajw@hive1 Linux x86_64)
~/labs/08 $ ./transpose 10000 5000
Testing naive transpose: 219.088 milliseconds
Testing transpose with blocking: 215.837 milliseconds
```

Fix n to be 10000, and run your code with blocksize equal to 50, 100, 500, 1000, 5000. How does performance change as the blocksize increases? Why is this the case?

As block size increases, it decrease the time until block size is 500, then increase again after 1000. It decreases, at the first time(spatial locality) but then it increases because of (conflict misses if too big)