COMP 322: Fundamentals of Parallel Programming (Spring 2019)
Instructors: Instructors: Mack Joyner, Zoran Budimlić
Worksheet 6: due at end of class today

Name: ______________________________ Net ID: _______________

Parallelizing Pascal's Triangle with Futures and Memoization

There are four variants of the Binomial Co-efficients program provided in four different HJlib methods in the next page:

a. Sequential Recursive without Memoization (`chooseRecursiveSeq()`)
b. Parallel Recursive without Memoization (`chooseRecursivePar()`)
c. Sequential Recursive with Memoization (`chooseMemoizedSeq()`)
d. Parallel Recursive with Memoization (`chooseMemoizedPar()`)

Your task is to analyze the WORK, CPL, and Ideal Parallelism for these four versions, for the input N = 4, and K = 2. **Assume that each call to ComputeSum() has COST = 1, and all other operations are free.** Complete all entries in the table:

Variant	Work	CPL	Ideal Parallelism
`chooseRecursiveSeq`			
`chooseRecursivePar`			
`chooseMemoizedSeq`			
`chooseMemoizedPar`			

Note: The work values should be equal for
(i) `chooseRecursiveSeq` and `chooseRecursivePar`
(ii) `chooseMemoizedSeq` and `chooseMemoizedPar`

Do you agree with the following statement: "Parallelization of inefficient algorithms often leads to more ideal parallelism than parallelization of efficient algorithms" in the context of this worksheet?

```java
private static int chooseRecursiveSeq(final int N, final int K) {
    if (N == 0 || K == 0 || N == K) return computeBaseCaseResult();
    final int left = chooseRecursiveSeq(N - 1, K - 1);
    final int right = chooseRecursiveSeq(N - 1, K);
    return computeSum(left, right);
}

private static int chooseRecursivePar(final int N, final int K) {
    if (N == 0 || K == 0 || N == K) return computeBaseCaseResult();
    final HjFuture<Integer> left = future(() -> chooseRecursivePar(N - 1, K - 1));
    final HjFuture<Integer> right = future(() -> chooseRecursivePar(N - 1, K));
    final HjFuture<Integer> resultFuture = future(() -> {
            final Integer leftValue = left.get();
            final Integer rightValue = right.get();
            return computeSum(leftValue, rightValue);
        });
    return resultFuture.get();
}

private static final Map<Pair<Integer, Integer>, Integer> chooseMemoizedSeqCache = new ConcurrentHashMap<>();

private static int chooseMemoizedSeq(final int N, final int K) {
    final Pair<Integer, Integer> key = Pair.factory(N, K);
    if (chooseMemoizedSeqCache.containsKey(key)) {
        final Integer result = chooseMemoizedSeqCache.get(key);
        return result;
    }
    if (N == 0 || K == 0 || N == K) {
        final Integer result = computeBaseCaseResult();
        chooseMemoizedSeqCache.put(key, result);
        return result;
    }
    final int left = chooseMemoizedSeq(N - 1, K - 1);
    final int right = chooseMemoizedSeq(N - 1, K);
    final int result = computeSum(left, right);
    chooseMemoizedSeqCache.put(key, result);
    return result;
}

private static final Map<Pair<Integer, Integer>, HjFuture<Integer>> chooseMemoizedParCache = new ConcurrentHashMap<>(
private static int chooseMemoizedPar(final int N, final int K) {
    final Pair<Integer, Integer> key = Pair.factory(N, K);
    if (chooseMemoizedParCache.containsKey(key)) {
        final HjFuture<Integer> result = chooseMemoizedParCache.get(key);
        return result.get();
    }
    final HjFuture<Integer> resultFuture = future(() -> {
            if (N == 0 || K == 0 || N == K) {
                return computeBaseCaseResult();
            }

            final HjFuture<Integer> left = future(() -> chooseMemoizedPar(N - 1, K - 1));
            final HjFuture<Integer> right = future(() -> chooseMemoizedPar(N - 1, K));

            final Integer leftValue = left.get();
            final Integer rightValue = right.get();
            return computeSum(leftValue, rightValue);
        });
    chooseMemoizedParCache.put(key, resultFuture);
    return resultFuture.get();
}
```