**R-20.7**

Solution:

2 3 4 1 first 4 misses cache-> 2 3 4 1 in least recent descending order

2 cache-> 3 4 1 2

5 cache -> 4 1 2 5 1 miss

1 cache -> 4 2 5 1

3 cache -> 2 5 1 3 1 miss

5 cache -> 2 1 3 5

4 cache -> 1 3 5 4 1 miss

1 cache -> 3 5 4 1

2 cache -> 5 4 1 2 1 miss

3 cache -> 4 1 2 3 1 miss

so there are total 9 misses.

**C-20.1**

Solution: We could use B tree to solve the problem, because the B tree is a special version of (a, b) tree. A and b should obey the Θ(B) when we are searching and updating. Then we should find a proper d. AB tree of order d is an (a, b) tree with a = [d / 2] and d = b. Because a and b is Θ(B), the d is also Θ(B). Because what we are interested is the number of disk accessing which implies that f(b) = c and g(b) = c for some constant c >= 1. For each search or update operation, we just need to perform one single disk transfer. So f(b) and g(b) are both O(1).

**A-20.4**

Solution: Because the operation of storing n key-value pairs in MapReduce framework is like we implement the dictionary in external memory where pairs are stored by the key and dictionary is stored in alphabetical words. We could use the external-memory sorting algorithm such that the input pairs can be put in order by the key. After the whole process, the set is ordered by the key.

The disk transfer times of the algorithm is O() according to the textbook, where B is the size of disk blocks and M is the size of the internal memory.

R-23.13

Solution: According to the textbook, “this worst-case bound is based on the pessimistic assumption that the hash values for the patterns in *L* might all be identical”. So when given the text string is “bbbbbbbbbbbbb” and the pattern string is like “bbb”, the Karp-Rabin algorithm run in Ω(nm) time.

C-23.4

Solution: We could modify the KMP algorithm in the textbook to find the longest prefix of P. Since the KMP algorithm is to find the whole pattern within the text. What we need is the prefix. So the idea is to record the prefix and its length each time.

modifiedKMP(T, P) {

input: text T, pattern P

output: longest prefix and the index

maxLength = 0 //we record the maximum length found so far

currentLength = 0 //we record the current matched length of the prefix

index n = 0 //the longest prefix’s first index

for i = 0 -> T.length – 1 {

if T[i] == P[j] currentLength++, i++, j++; //which means the both elements match, so we //continue to compare next element and //increment current index.

If T[i] != P[j] {

If maxLength < currentLength

maxLength = currentLength

// if these two elements don’t match, we check if current length is longer than the

// maximum, length, and set the current length to be 0.set n to be the current index.

// And move the index according to the KMP

}

}

Return maxLength, n;

}

A-23.2

Solution: We could use trie structure. Let define S as a set of all stop words. Then we could establish a standard trie depending on S. Since we know trie has some properties: Each node of T, except the root, is labeled with a character of S; the ordering of the children of an internal node of T is determined by a canonical ordering of the alphabet Σ; T has s external nodes, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to an external node v of T yields the string of S associated with v.

So each stop word is represented through the root to the external node. Since the search operation and insert operation both cost O(key\_length). And for each word, the length is constant. So the running time is constant.