b)

1.

while (first1 <= last1 && first2 <= last2) {

if (a[first1] <= a[first2]) {

aux[index] = a[first1];

first1++;

}

else {

aux[index] = a[first2];

first2++;

}

index++;

}

As the code shown above, when a[first1]<=a[first2], it will always be first inserted to the aux array, which means the relative order between the same keys won’t be changed since the sub\_array from lo to mid will be always before the sub\_array from mid+1 to hi. Therefore, if there are same keys in both sub\_array, then the keys in the front sub\_array will be always first inserted. So the sorting is stable.

2.

void MergeSort(int a[], int aux[], int lo, int hi) {

if (hi <= lo) return;

int mid = lo + (hi - lo) / 2;

MergeSort(a, aux, lo, mid);

MergeSort(a, aux, mid + 1, hi);

Merge(a, aux, lo, mid, hi);

}

Here I used mergesort. Like what shown above, by recursively calling MergeSort, we can divide the function calling into log2N levels because every time the the size of the array being called is divided by 2.

Also, for each level I use the merge function to merge the every two sub\_array divided by mergesort function.

int first1 = lo;

int last1 = mid;

int first2 = mid + 1;

int last2 = hi;

int index = lo;

while (first1 <= last1 && first2 <= last2) {

if (a[first1] <= a[first2]) {

aux[index] = a[first1];

first1++;

}

else {

aux[index] = a[first2];

first2++;

}

index++;

}

while (first1 <= last1) {

aux[index] = a[first1];

++first1;

++index;

}

while (first2 <= last2) {

aux[index] = a[first2];

++first2;

++index;

}

The main body of merge is shown above, as you can see, the merge only operates from lo to hi and after elements from lo to hi have been evenly sorted into aux[], then we copy aux[]’s elements from lo to hi back to a[] which should only take O(hi-lo+1). Since the total elements of each level is n, it means it the total running time will be O(N) for each level. Then since there are logN levels in total. We get O(NlogN).

The best-case complexity should be same as the worst-case complexity since no matter what kinds of orders of elements in the array are, the merge function will always go through from lo to hi for each sub-array pasted by the MergeSort. Therefore, it means it will always have O(N) for each level of mergesort.

while (first1 <= last1 && first2 <= last2) {

if (a[first1] <= a[first2]) {

aux[index] = a[first1];

first1++;

}

else {

aux[index] = a[first2];

first2++;

}

index++;

}

while (first1 <= last1) {

aux[index] = a[first1];

++first1;

++index;

}

while (first2 <= last2) {

aux[index] = a[first2];

++first2;

++index;

Like the code shown above, the total time of the three while loops will be always constant as hi-lo+1, because it has to make sure every element from a has been evenly sorted into aux.

Therefore, the best-case is same worst-case, which is O(NlogN).

3.Same as 2.