(a)

```
Bubblesort(A, start, end) // A[start…end]
swapped_before = True
   While swapped_before == True:
   swapped_before = False
         for i = start+1 to end //inclusive
            if A[i] < A[i-1] then // compare each pair of adjacent items
                  swap(A[i], A[i-1])
                  swapped_before = True
            i++ // if no swaps were done, the array is sorted
```

(b) best case:
The best case for bubbleSort is an already sorted array, this is because the expression A[i] < A[i-1] is always false, so after one iteration, the algorithm terminates. More formally, the algorithm only runs $n-1$ compares, leading to a time complexity of $\Theta(n)$.

Worst case:
The worst case for bubbleSort with the most amount of swaps is an array in descending order. In the first iteration of Bubblesort, the largest element, in position 1, will end up being moved to the end of the array, resulting in $n-1$ swaps. During the second iteration, the largest element is already at the end of the array, and the second largest element (now at position 1 / start) will end up in the (n-1)th position, resulting in n -1 -1 swaps. This results in the following sum:

$$T(n) = \sum_{i=0}^{n-1}(n-i-1) = n\sum_{i=0}^{n-1}1 - \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1}1$$

$$= n(n-1) - \frac{(n-1)(n)}{2} - n + 1$$

$$= n^2 - n - \frac{n^2-n}{2} - n + 1$$

$$= \frac{n^2}{2} - \frac{3n}{2} + 1$$

therefore, $T(n) = \Theta(n^2)$

Average case:
The average case is also similar to the worst case, since the amount of times that the outer loop will be run depends on n, and each iteration must do n comparisons, and at most n-1 swaps. Therefore, the time complexity for the average case is $O(n^2)$.

(c)

Insertion Sort: Stable sort, it does not change the order of the elements when the keys are the same.

Merge Sort: Stable sort. This is true under the “standard” implementation of merge sort where we prioritise the left-most sub-array over the right-most sub-array during equal comparisons.

Heapsort: Unstable sort: any initial ordering of the keys is broken up by the heapify() function

Bubblesort: Stable sort: As per the pseudocode above, swapping elements only occurs when the i-1th element is strictly bigger than the ith element of the array. If they are the same, nothing changes, so the sort is stable.

(d)

Insertion sort: Adaptive, the complexity of insertion sort is $\Theta(n)$ if the array is already sorted, if parts of the array are sorted, fewer swaps are done

Merge Sort: Not Adaptive. The time complexity of merge sort is always $\Theta(n \log n)$. The number of merge operations does not chang

Heap Sort: Non-Adaptive: every iteration of heap sort removes the largest element from the “root “of the heap and calls the heapify() function. The number of Heapify calls does not change if the array is already sorted.

Bubble sort: Adaptive, the time complexity of bubble sort is $\Theta(n)$ if the array is already sorted. In other words, we can say that the time complexity of bubble sort is $\Omega(n)$. An array that is partly sorted will results in fewer swaps and thus be faster.

PROBLEM 6.2
For full implementation, see the code in main.cpp. this code was inspired by the English wikipedia pseudocode implementation of Bottom Up HeapSort.

(a)

```
#include <algorithm>
template <class T>
void Build_Max_Heap(T* arr, const int size) {
	for (int i = (size-1) / 2; i >= 0; i--)
		Max_Heapify(arr, i, size);
}

template <class T> void Max_Heapify(T* arr, const int i, const int size){
	int l = left(i);
	int r = right(i);
	int largest;
	if (l < size && arr[l] > arr[i])
		largest = l;
	else
		largest = i;
	if (r < size && arr[r] > arr[largest])
		largest = r;
	if (largest != i){
		T temp;
		temp = arr[i];
		arr[i] = arr[largest];
		arr[largest] = temp;
		Max_Heapify<T>(arr, largest, size);
	}
}

void HeapSort(T* arr, int size){
	Build_Max_Heap<T>(arr, size);
	for (int i = size-1; i > 0; i--) {
		T temp = arr[0];
		arr[0] = arr[i];
		arr[i] = temp;
		//cout<<"i is: "<<i<<" value taken from max is: "<<arr[i]<<endl;
		Max_Heapify<T>(arr, 0, i);
	}
}
```

(b)

```
//quickly sinks the topmost element to the bottom, then returns position of sunk element.

template <class T> int FastSink(T* arr, int size){
	int current = 0;
	int l = left(current);
```

```
        int r = right(current);

        while (l < size && r < size){
                if (arr[l] > arr[r]) {
                        current = l;
                }
                else{
                        current = r;
                }
                l = left(current);
                r = right(current);
        }
        //corner case: r is not part of heap but l is
        if (l < size) {
                current = l;
        }
        return current;
}

//shamelessly inspired by wikipedia's bottom down heapsort,
// puts the old root in the correct position
template <class T>void FixHeap(T* arr, int pos, int size){
        int navigator = pos;

        while (arr[0] > arr[navigator]){
                navigator = parent(navigator);
        }
        T temp = arr[navigator];
        arr[navigator] = arr[0];

        while (navigator > 0){
                swap(temp, arr[parent(navigator)]);
                navigator = parent(navigator);

        }
}

template <class T> void NewHeapSort(T* arr, int size){
        Build_Max_Heap<T>(arr, size);
        for ( int i = size-1; i > 0; i--){
                T temp = arr[0];
                arr[0] = arr[i];
                arr[i] = temp;
                int pos = FastSink(arr, i);
                FixHeap(arr, pos, i-1);
        }
}
```