CS101A(H) Homework 5

Brief Overview

This homework consists of four parts:

- **Single Choices (10 pts):** *Scoring per question:*
  - Select the single correct option to receive full credit.
  - Any wrong selection, selecting multiple options, or leaving it blank receives **0 pt**.
- **Multiple Choices (15 pts):** *Scoring per question:*
  - Selecting *all and only* the correct options: **full credit**.
  - Selecting a *proper subset* of the correct options (and *no* incorrect options): **half credit**.
  - Selecting *any* incorrect option or leaving it blank: **0 pt**.

  Note: If the correct set has only one option, the "half credit for a proper subset" case does not apply.
- **Others (8+4+6+12 = 30 pts): Answers must be written in English**. The answer format varies by problem (e.g., short proofs, derivations, explanations, code snippets). The point value for each subproblem is printed next to the problem title. Follow the instructions given in each problem; show key steps when requested.

Your grade for this homework will be computed as

$$\min\Big(\text{Single Choices} + \text{Multiple Choices} + \text{Others},\ 50\ \text{pts}\Big).$$

Notes: Unless otherwise stated, always express your final asymptotic bounds using $\Theta(\cdot)$ notation for tight complexity, not just $O(\cdot)$ or $\Omega(\cdot)$. All answers must be written **inside the provided answer boxes** and **in English**. When submitting, match your solutions to the problems correctly in Gradescope. No late submission will be accepted. Failure to follow these rule may result in partial or full loss of credit.

Usage of AI. The use of AI tools for searching information or obtaining assistance on this homework is strictly prohibited. All solutions must be derived from your own understanding and effort. Submissions will be reviewed carefully, and any indication of reliance on AI-generated content will result in severe penalties.

Academic Integrity. This course upholds the highest standards of academic integrity. Any form of academic dishonesty, including plagiarism, unauthorized collaboration, or the use of prohibited resources such as AI tools, will be treated as a serious violation. Such actions not only undermine your own learning but also violate university policies.

Due Date: Nov 11, 2025, **8:00 PM**

(1.1)	(1.2)	(1.3)	(1.4)	(1.5)
(2.1)	(2.2)	(2.3)	(2.4)	(2.5)

1 Single Choices

1. You have to sort n data with very limited extra memory ($\Theta(1)$ extra memory every case). Choose sort algorithm(s) which is(are) appropriate.

(1) merge sort, (2) insertion sort (3) bubble sort (4) quick sort

A. (1)(2)
B. (1)(2)(3)
C. (2)(3)
D. (2)(3)(4)
E. (2)(4).

2. In merge sort, if the merge function uses $if\ (a1[i1] < a2[i2])$ and otherwise takes the element from the right array. What is the stability of this implementation?

A. Stable.
B. Unstable.
C. Stable only when the two sub-arrays have same number of inversions .
D. Stable only when the two sub-arrays are of equal length.

3. Using half-open intervals $[first, last)$ with $mid = (first + last)/2$, which combination of calls is correct?

A. merge_sort(first, mid); merge_sort(mid + 1, last); merge(first, mid + 1, last)
B. merge_sort(first, mid); merge_sort(mid, last); merge(first, mid, last)
C. merge_sort(first, mid - 1); merge_sort(mid, last); merge(first, mid - 1, last)
D. merge_sort(first, last - 1); merge_sort(last - 1, last); merge(first, last - 1, last)

4. What is the expected number of pairs of inversions in an array of length n?

A. $\Theta(n)$ B. $\Theta(n^2)$ C. $\Theta(n \log n)$ D. $\Theta(\log(n!))$

5. Which of the following is correct?

A. When array A contains distinct elements and is arranged in descending order, the time complexity of quick sort(always choose the last element as pivot) is $\Theta(n^2)$.
B. All sorting algorithms must be $\omega(n)$.
C. Any comparison-based sorting algorithm can be represented by a comparison tree. Worst-case running time can be less than the height of the tree
D. There doesn't exist a sorting algorithm takes $\theta((n+1)!)$ time.

2 Multiple Choices

1. In the lecture we have learned that different sorting algorithms are suitable for different scenarios. Which of the following statements is/are suitable for insertion sort?

A. Each element of the array is close to its final sorted position.

B. The length of the array is small.

C. The sorting must be performed in place.

D. Comparisons are more expensive than writes or swaps.

2. Which of the following sorting algorithm(s) is(are) stable?

A. Insertion sort.

B. Merge sort.

C. Quick sort.

D. Flagged bubble sort.

3. Which of the following situations are true for an array of n random numbers?

A. The number of inversions in this array can be found by applying a recursive algorithm adapted from merge-sort in $\Theta(n \log n)$ time.

B. If it has no more than n inversions, it can be sorted in $O(n)$ time.

C. If the array is $\langle 8, 6, 3, 7, 4 \rangle$, there are 6 inversions.

D. If all elements of the array are the same, the time complexity of quick sort is $\Theta(n \log n)$.

E. Given 2 sorted lists of size m and n respectively, and we want to merge them to one sorted list by mergesort. Then in the worst case, we need $m + n - 1$ comparisons.

4. The time complexity for both insertion sort and flagged bubble sort will be the same if:

A. the input array is reversely sorted.

B. the input array is a list containing n copies of the same number.

C. the input array is already sorted.

D. the input array is a concatenation of two sorted subarrays.

5. Which of the following implementations of quick-sort may improve the average behavior (not only improvement on time complexity) of trivial quick-sort?(Compared to always picking the first element as pivot and using no optimization, data are uniformly distributed.)

A. Always picking the last element.

B. When partitioning the subarray $\langle a_l, \cdots, a_r \rangle$ (assuming $r - l \geqslant 2$), choose the median of $\{a_x, a_y, a_z\}$ as the pivot, where x, y, z are three different indices chosen randomly from $\{l, l + 1, \cdots, r\}$.

C. When partitioning the subarray $\langle a_l, \cdots, a_r \rangle$ (assuming $r - l \geqslant 2$), we first calculate $q = \frac{1}{2}(a_{\max} + a_{\min})$ where $a_{\max}$ and $a_{\min}$ are the maximum and minimum values in the current subarray respectively. Then we traverse the whole subarray to find a_m $s.t.$ $|a_m - q| = \min_{i=l}^{r} |a_i - q|$

and choose a_m as the pivot.

D. When the disordered sub-array is short enough, use insertion-sort instead of quick-sort.

3 Sort Practice (8 pts)

(a) Run Insertion Sort for array $\{1, 6, 2, 7, 8, 3, 5, 4\}$. Write down the array after each outer iteration. (2 pts)

```
for(int k = 1; k < n; k++){
    for(int j = k; j > 0; j--){
        if( array[j- 1] > array[j] )
            swap(array[j- 1], array[j]);
        else
            break;
        }
    print(array);
}
```

(b) From list (diagram) to arrays (fill numbers). (2 pts)

Run flagged bubble Sort for the array in (a). Write down the array **after each** outer iteration.

```
for(int i = n-1; i > 0; i--){
    int max_t = array[0];
    bool sorted = true;
    for(int j = 1; j <= i; j++){
        if(array[j] < max_t){
            array[j-1] = array[j];
            sorted = false;
        }
        else{
            array[j-1] = max_t;
            max_t = array[j];
        }
    }
    array[i] = max_t;
    print(array);
    if (sorted)
        break;
}
```

(c) (2 pts) Run Merge Sort for this array. Write down the array **after** each merge and underline the sub-array being merged. Note that insertion sort are only used when for sub-array of size 2 or less.

(d) (2 pts) Run Quick Sort for this array. Choose the last entry as pivot in this question. Write down the array **after** each quick sort operation(move all entries smaller than the pivot to left and move all entries larger than the pivot to right, then move the pivot. Please refer to the implementation in ppt.) and underline the sub-array. Note that insertion sort are only used when for sub-array of size 2 or less.

4 Insertion Sort using Linked List (4 pts)

We have learnt the insertion sort implementation using array. In this question, you are required to implement insertion sort using single linked list. Since it is not easy to traverse single linked list from back to front, we can traverse from front to back instead if it is needed.

Fill in the blanks to complete the algorithm. Please note that there is at most one statement (ended with ;) in each blank line.

```
struct ListNode
{
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};

ListNode* insertionSort(ListNode* head) {
    //if linked list is empty or only contains 1 element, return directly
    if(!head || !(head->next)){return head;}

    //create dummy node
    ListNode *dummy = new ListNode(-1);
    dummy->next = head;

    //split linked list into sorted list and unsorted list
    ListNode *tail = head; //tail of sorted list
    ListNode *sort = head->next; //head of unsorted list

    //insertion sort
    while(sort)
    {
       if(sort->val < tail->val)
       {
           ListNode *ptr = dummy;
           while(ptr->next->val < sort->val) ptr = ptr->next;
           // Your code HERE!
           // line1
           // line2
           // line3
           // line4
       }
       else
       { //no need to insert
           tail = tail->next;
           sort = sort->next;
       }
    }
```

```
    ListNode *ans = dummy->next;
    delete dummy; dummy = nullptr;
    return ans;
}
```

Fill:

(1) ____

(2) ____

(3) ____

(4) ____

5 Quick Sort (6 pts)

(a) (2 pt)If we use randomized quick-sort (i.e., randomly choosing pivots) to sort the array $[3, 9, 6, 2, 1, 5, 8, 0]$, the probability of 2 and 5 being compared is _____.

(b) (4 pts) Fill in the blanks to complete the algorithm. Please note that there is at most one statement in each blank line.

```
// Median-of-3: ensure a[first] <= a[mid] <= a[last-1],
//then swap the median to the end as pivot
template <typename T>
T pick_pivot_median3(T* a, int first, int last) { // [first, last)
    int mid = first + (last - first) / 2;
    // TODO(1): after three swaps, ensure a[first] <= a[mid] <= a[last-1]
    if (/* TODO(1a) */) std::swap(a[first], a[mid]);
    if (/* TODO(1b) */) std::swap(a[first], a[last - 1]);
    if (/* TODO(1c) */) std::swap(a[mid], a[last - 1]);
    std::swap(a[mid], a[last - 1]);
    return a[last - 1];
}

// Find in [i, last-1) the first index with a[index] >= pivot
template <typename T>
int find_next_ge(const T& pivot, T* a, int i, int last) {
    while (/* TODO(2) */) {
        ++i;
    }
    return i;
}

// Find in (first, j] the last index with a[index] < pivot
template <typename T>
int find_prev_lt(const T& pivot, T* a, int j, int first) {
    while (j > first && !(a[j] < pivot)) {
        --j;
    }
    return j;
}

// Quicksort: sort the half-open range [first, last)
template <typename T>
void quick_sort(T* a, int first, int last) {
    while (last - first > QS_INSERTION_THRESHOLD) {
        // Choose pivot (median moved to a[last-1])
        T pivot = pick_pivot_median3(a, first, last);

        // Left keeps < pivot, right keeps >= pivot
```

```
        int low  = find_next_ge(pivot, a, first, last);
        int high = find_prev_lt(pivot, a, last - 2, first);

        while (low < high) {
            std::swap(a[low], a[high]);
            ++low;
            --high;
            low  = /* TODO(3a) */;
            high = /* TODO(3b) */;
        }

        // Place pivot into its final position;
        std::swap(/* TODO(4a) */,/* TODO(4b) */);
        int pivot_pos = low;

        // Tail recursion elimination: recurse on the smaller side first
        int left_size  = pivot_pos - first;
        int right_size = last - (pivot_pos + 1);

        if (left_size < right_size) {
            if (left_size>1) quick_sort(a, first, pivot_pos);
            first = pivot_pos+1;
        } else {
            if (right_size>1) quick_sort(a, pivot_pos + 1, last);
            last = pivot_pos;
        }
    }

    if (last - first > 1) {
        insertion_sort(a, first, last);
    }
}
```

Fill:

(1a) _____ (1b) _____

(1c) _____

(2) _____

(3a) _____ (3b) _____

(4a) _____ (4b) _____

6 Choosing Appropriate Sorting Algorithms (12 pts)

For each of the following scenarios, choose a sorting algorithm (from either bubble sort, insertion sort, or merge sort) that best applies, and justify your choice. Each sort may be used more than once. If you find that multiple sorts could be appropriate for a scenario, identify their pros and cons, and choose the one that best suits the application. State and justify any assumptions you make. “Best" should be evaluated by asymptotic running time.

(a) Suppose you are given a data structure `D` maintaining an extrinsic order on n items, supporting two standard sequence operations: `D.get` at(i) in worst-case $\Theta(1)$ time and `D.set` at(i, x) in worst-case $\Theta(n \log n)$ time. Choose an algorithm to sort the items in `D` in place best.

(b) Suppose you have a static array A containing pointers to n comparable objects, pairs of which take $\Theta(\log n)$ time to compare. Choose an algorithm to sort the pointers in A so that the pointed-to objects appear in non-decreasing order with minimum time cost.

(c) Suppose you have a sorted array A containing n integers. Now suppose someone performs some log log n swaps between pairs of adjacent items in A so that A is no longer sorted. Choose an algorithm to best re-sort the integers in A.