COMP2119 Introduction to Data Structures and Algorithms
Practice Problem Set 4 - Sorting Algorithms

Release Date: Oct 21, 2024

Please do NOT submit the answer of this part.

Chapter 9 - Sorting (1)

1. Given a list $L = [188, 157, 122, 148, 192, 136, 167, 31, 138, 202, 126]$. Show step-by-step on how the following sorting algorithms sorts L in ascending order. In particular, show the resulting list of numbers after each round of iteration.

   (a) Bubble sort
   (b) Insertion sort
   (c) Selection sort

   Solution:

   (a) 157 122 148 188 136 167 31 138 192 126 202
   122 148 157 136 167 31 138 188 126 192 202
   122 148 136 157 31 138 167 126 188 192 202
   122 136 148 31 138 157 126 167 188 192 202
   122 136 31 138 148 126 157 167 188 192 202
   122 31 136 138 126 148 157 167 188 192 202
   31 122 136 126 138 148 157 167 188 192 202
   31 122 126 136 138 148 157 167 188 192 202
   31 122 126 136 138 148 157 167 188 192 202
   31 122 126 136 138 148 157 167 188 192 202

   (b) 157 188 122 148 192 136 167 31 138 202 126
   122 157 188 148 192 136 167 31 138 202 126
   122 148 157 188 192 136 167 31 138 202 126
   122 148 157 188 192 136 167 31 138 202 126
   122 136 148 157 188 192 167 31 138 202 126
   122 136 148 157 167 188 192 31 138 202 126
   31 122 136 148 157 167 188 192 138 202 126
   31 122 136 138 148 157 167 188 192 202 126
   31 122 136 138 148 157 167 188 192 202 126
   31 122 126 136 138 148 157 167 188 192 202

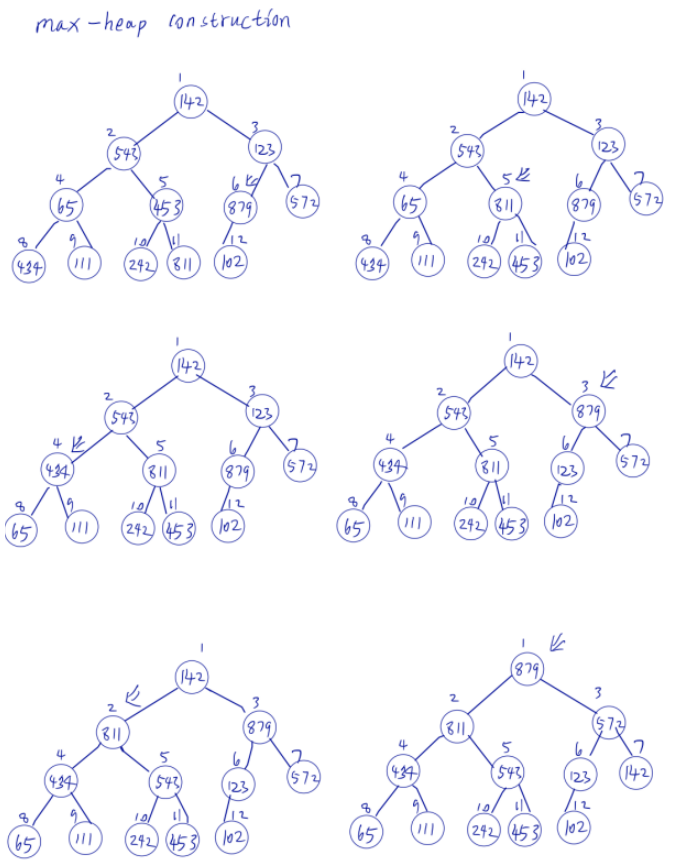
(c) 31 157 122 148 192 136 167 188 138 202 126
31 122 157 148 192 136 167 188 138 202 126
31 122 126 148 192 136 167 188 138 202 157
31 122 126 136 192 148 167 188 138 202 157
31 122 126 136 138 148 167 188 192 202 157
31 122 126 136 138 148 167 188 192 202 157
31 122 126 136 138 148 157 188 192 202 167
31 122 126 136 138 148 157 167 192 202 188
31 122 126 136 138 148 157 167 188 202 192
31 122 126 136 138 148 157 167 188 192 202

2. Given a list $L = [142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102]$. Show step-by-step on how heapsort sorts L in ascending order. In particular, show the max-heap construction process. You also need to show the changes made to the heap during first iteration of the sorting procedure.

Solution:

Step 1: Heapify Procedure

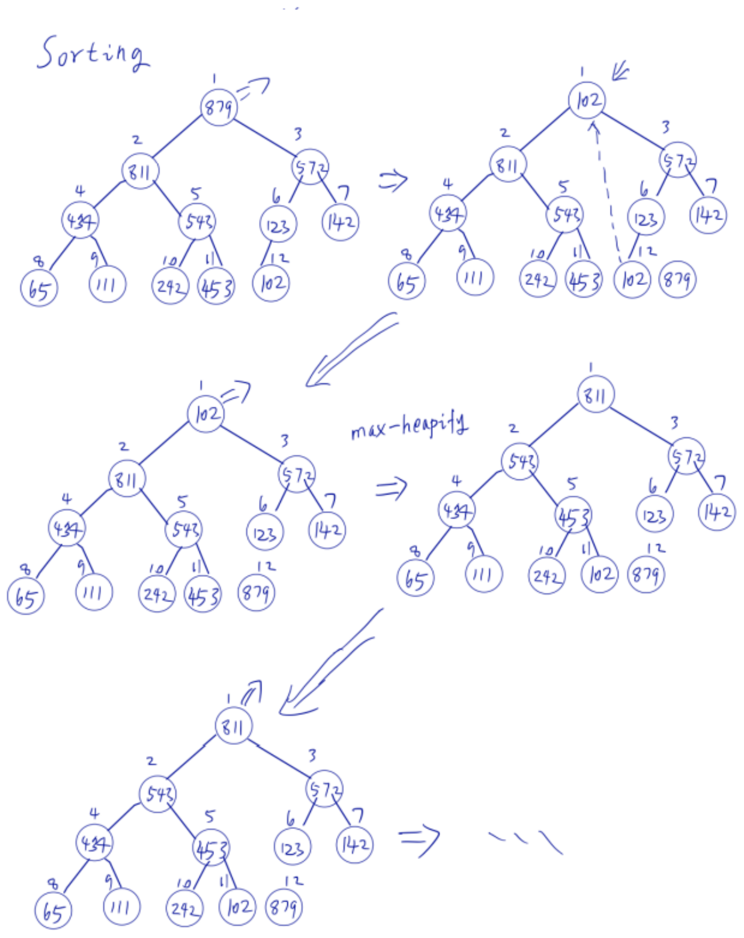
The algorithm first conducts a heapify procedure. In each iteration i (from list index $\lfloor \frac{12}{2} \rfloor$ to index 1), node at index i will compare with its two children nodes and swap with the largest children if the node at index i is smaller than its children. Refer to the figure below for the max-heap construction:



The heapified list is $[879, 811, 572, 434, 543, 123, 142, 65, 111, 242, 453, 102]$.

Step 2: Sorting Procedure

In the sorting procedure, we could remove the top element and place it at the end of the list. After that, the heap will be maintained as max-heap. 102 will be bubbled down. The resultant list becomes [811, 543, 572, 434, 453, 123, 142, 65, 111, 242, 102, **879**]. Refer to the diagram below for illustration:



After that, top node is obtain one by one and sort elements as follows (The sorted elements are bolded):

572, 543, 142, 434, 453, 123, 102, 65, 111, 242, **811, 879**
543, 453, 142, 434, 242, 123, 102, 65, 111, **572, 811, 879**
453, 434, 142, 111, 242, 123, 102, 65, **543, 572, 811, 879**
434, 242, 142, 111, 65, 123, 102, **453, 543, 572, 811, 879**
242, 111, 142, 102, 65, 123, **434, 453, 543, 572, 811, 879**
142, 111, 123, 102, 65, **242, 434, 453, 543, 572, 811, 879**
123, 111, 65, 102, **142, 242, 434, 453, 543, 572, 811, 879**
111, 102, 65, **123, 142, 242, 434, 453, 543, 572, 811, 879**
102, 65, **111, 123, 142, 242, 434, 453, 543, 572, 811, 879**
65, **102, 111, 123, 142, 242, 434, 453, 543, 572, 811, 879**

3. There are $n!$ different arrays of size n containing 1, 2, ..., n. Consider the C++ code below to bubble sort all these arrays.

```
void bubble_sort(int *array, int n) {
    int i, j;
    for (i = 0; i < n - 1; ++i) {
        for (j = 0; j < n - 1; ++j) {
            if (array[j] > array[j + 1]) {
                swap(array[j],array[j+1]);
            }
        }
    }
}
```

Calculate the average number of swaps in terms of n.

Example:
When $n = 2$, there are two arrays: $[1, 2]$ and $[2, 1]$. The total number of swaps is 1, so the average is 0.5.

Solution:

We consider reverse pair: If $i, j \in 1, 2, ..., n$, $i < j$, and $array[i] > array[j]$, we call i and j a reverse pair.

For any array A, assume there are x reverse pair(s). Reverse A to B, such that $A[i] = B[n - 1 - i]$. The number of reverse pair(s) in B is $\frac{n(n-1)}{2-x}$.

Therefore, the average is $\frac{n(n-1)}{4}$.

4. The following C++ program uses a heap to output k smallest elements in an array of length n within $O(n \log k)$ time. Fill in the blanks such that the program can work.

```
void heapify(int arr[], int num, int i) {
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left node
    int r = 2 * i + 2; // right node

    // If left child is larger than root
    if (l < num && arr[l] > arr[largest])
        largest = l;

    if (1.______________________________)
        largest = r;

    if (2.__________) {
        swap(arr[i], arr[largest]);
        heapify(arr, num, largest);
    }
}

void find_k_smallest(int arr[], int n, int k) {
    // Build heap (rearrange array)
    for (int i = k / 2 - 1; i >= 0; i--)
        3.__________________;

    for (int i = k; i < n; i++) {
        if (arr[i] < arr[0]) {
            4._________________;
            5._________________;
        }
    }
}

int main() {
    int n, k;
    cin >> n >> k; // size of the array
    int arr[n];
    for (int i = 0; i < n ; ++i) {
        cin >> arr[i];
    }

    find_k_smallest(arr, n, k);
    cout << k << " smallest elements are " << endl;
    for (int j = 0; j < k ; ++j) {
        cout << arr[j] << " ";
    }
}
```

Solution:

```
1. r < num && arr[r] > arr[largest]
2. largest != i
3. heapify(arr, k, i)
4. swap(arr[0], arr[i])
5. heapify(arr, k, 0)
```

5. You are given an integer b and an almost sorted array $A[1:n]$ of distinct integers, where each element can be misplaced by at most b positions (i.e. the sorted position of the i-th element $A[i]$ is at least $i-b$ and at most $i+b$.).

   (a) Design an algorithm to sort A in $O(nb)$ time and $O(1)$ extra space.
   (Hint: Modify selection sort)

   (b) Design an algorithm to sort A in $O(n \log b)$ time and $O(1)$ extra space.
   (Hint: Modify heap sort)

Solution:

(a) The algorithm modifies selection sort to do the job:

Algorithm 5.1 Sort1

```
function SORT1($A$, $b$)

    for ($i = 1$ to $n$) do
        $m = \infty$

        for ($j = i$ to $i + b$) do                ▷ Find min element among next $b$ positions
            if ($m \leq A[j]$) then
                $m = A[j]$
            end if
        end for

        Swap($A[i]$, $A[m]$)
    end for

    return $A$

end function
```

(b) The algorithm modifies heap sort to do the job:

Algorithm 5.2 Sort2

```
function SORT2($A$, $b$)

    $start = 1$
    while ($start \leq n$) do
        $end$ = Min($n$, $start + 2b - 1$)
        HeapSort(A[$start$:$end$])
        $start = start + b$
    end while

    return $A$

end function
```

Chapter 10 - Sorting (2)

1. Given a list of number $L = [3, 1, 4, 1, 5, 9, 2]$. Describe step-by-step on how the following sorting algorithms sorts L in ascending order. Show the resulting list of numbers after each step.

   (a) Merge sort

   (b) Quick sort

   Solution:

   (a) () represents a sub-set of integers not being sorted yet.
   [] represents a sub-set of integers finished sorting.

   The input can be split into half continually:
   1. $(3, 1, 4, 1, 5, 9, 2)$
   2. $(3, 1, 4)$ $(1, 5, 9, 2)$
   3. $(3, 1)$ (4) $(1, 5)$ $(9, 2)$
   4. (3) (1) (4) (1) (5) (9) (2)

   Now, sort the sub-problems and the solutions are merged back together.

   5. $[1, 3]$ $[4]$ $[1, 5]$ $[2, 9]$
   6. $[1, 3, 4]$ $[1, 2, 5, 9]$
   7. $[1, 1, 2, 3, 4, 5, 9]$

   (b) () represents a sub-set of integers not being sorted yet.
   [] represents a sub-set of integers finished sorting.

   1. $(\mathbf{3}, 1, 4, 1, 5, 9, 2)$, 3 is pivot.
   2. $(\mathbf{2}, 1, 1)$ $[3]$ $(\mathbf{5}, 9, 4)$, 2 is pivot for $(2, 1, 1)$ and 5 is pivot for $(5, 9, 4)$.
   3. $(\mathbf{1}, 1)$ $[2, 3]$ $(\mathbf{4})$ $[5]$ $(\mathbf{9})$, 1 is pivot for $(1, 1)$, 9 is pivot for (9), 4 is pivot for (4).
   4. $[1]$ (1) $[2, 3, 4, 5, 9]$, 1 is pivot for (1).
   5. $[1, 1, 2, 3, 4, 5, 9]$

2. For each of the following statements, state whether it is True or False. For a False statement, point out the incorrectness.

   (a) The worst case time complexity of insertion sort is $\Theta(n^2)$.

   (b) There exits an input array of length n that forces randomized quicksort to run in $\Theta(n^2)$ time in expectation.

   (c) Building a heap out on n elements stored in an array requires only $O(n)$ time.

   (d) Radix sort works in linear time only if the elements to sort are integers in the range $[1, cn]$ for some constant c.

   (e) There exists a comparison-based sorting algorithm for 5 numbers that uses at most 8 comparisons in the worst case.

Solution:

(a) True.

(b) False. For any input, the expected running time of quicksort is $O(n \log n)$, where the expectation is taken over the random choices made by quicksort, independent of the choice of the input.

(c) True.

(d) False. Radix sort also works in linear time if the elements to sort are integers in the range $[1, n^d]$ for any constant d.

(e) True.

3. Consider the following algorithm for sorting n distinct numbers in an array $A[1..n]$.

Algorithm 3.1 TreeSort

```
function TreeSort($A$, $n$)

    initialize a null binary search tree $T$;
    for ($i = 1$; $i \leq n$; $i{+}{+}$) do
        insert $A[i]$ into $T$;
    end for
    perform a tree walk on T to enumerate the numbers in sorted order;

end function
```

(a) What is the average-case complexity (Big-O) of TreeSort? Explain your answer.

(b) What is the worst-case complexity (Big-O) of TreeSort? Explain your answer.

(c) What tree traversal order will you use in Line 5?

(d) What would you do to improve the worst-case complexity of the algorithm?

(e) How would you compare TreeSort against Merge Sort?

Solution:

(a) The average depth of a node in a randomly built BST is $O(\log n)$. Therefore, the average insertion cost of a node in a randomly built BST is $O(\log n)$. Therefore, lines 3-4 runs in $O(n \log n)$ time. The time complexity of applying in-order tree traversal at line 5 is $O(n)$. As a result, the average-case complexity (Big-O) of TreeSort is $O(n \log n)$.

(b) The worst case happens when the tree created is degenerated into a linked list, with only one element at each of the levels in the tree. In this case, inserting the i-th element requires $i - 1$ comparisons. The cost of inserting all n elements is thus $O(n^2)$. As a result, the worst-case time complexity is $O(n^2)$.

(c) Inorder traversal

(d) We can use the rotation operation to make T an AVL tree. The time complexity of the worst-case is still $O(n \log n)$.

(e) This is an open-ended question. Other reasonable answers can be accepted.

- MergeSort has a better worst-case complexity $O(n \log n)$ than TreeSort.
- TreeSort uses much more additional space than Mergesort. Every element is represented by a node in a tree, which requires 3 pointers (left-child, right-child, parent).

4. Modify randomized quick-sort (i.e. The pivot is chosen randomly) to find the k-th smallest integer in an unsorted array $A[1 : n]$ of distinct integers. Your algorithm should run in $O(n)$ time.

Solution:

The algorithm modifies quick-sort to do the job:

Algorithm 4.1 Find the k-th smallest integer by quick-sort

```
function kthSmallest($A[i...j]$, $k$)

    if ($i == j$) then
        return $A[i]$
    end if

    $v$ = RandomPivot($A[i...j]$)
    $m$ = Partition($A[i...j]$, $v$)

    if ($i + k - 1 \leq m$) then
        kthSmallest($A[i...m]$, $k$)
    else
        kthSmallest($A[m + 1...j]$, $k - (m - i + 1)$)
    end if

end function
```

Further Practice from the Textbook:
To gain more practice, here are a list of supplementary problems you may attempt from the course textbook **Introduction to Algorithms, 3rd edition**:

Chapter 9 - Sorting (1)

1. Insertion Sort:
   Exercise 2.1-1, 2.1-2, 2.3-4, 2.3-6, 2-4
2. Selection Sort:
   Exercise 2.2.2
3. Heap Sort:
   Exercise 6.1-1, 6.1-2, 6.1-4, 6.1-5, 6.1-6, 6.3-1, 6.4-1, 6.4-3, 6.3

Chapter 10 - Sorting (2)

1. Merge Sort:
   Exercise 2.3-1, 2.3-7
2. Quick Sort:
   Exercise 7.1-1, 7.1-2, 7.1-3, 7.1-4, 7.2-2, 7.2-3, 7.2-4
3. Counting Sort:
   Exercise 8.2-1, 8.2-4
4. Radix Sort:
   Exercise 8.3-1, 8.3-4