Programming Assignment 2

Department of Computer Science, University of Wisconsin – Whitewater
Theory of Algorithms (CS 433)

Instructions For Submissions

- **This assignment is to be completed individually. If you are stuck with something, consider asking the instructor for help.**
- Submit code via Canvas as a single zip file.
- **Any function with a compilation error will receive a zero, regardless of how much it has been completed.**

---

1 Overview

We are going to:

- **Implement a better partitioning strategy**. Then, use it to **implement randomized quick-select and median-of-medians**

  Implement the `partition` method in `Partition`

  Implement the `select` methods in `SelectionRandom` and `SelectionMofM`

- **Sort an Array of Strings using Radix-Sort**

  Implement the `radixSort` and `countSortOnLowerCaseCharacters` methods in `StringSorter`

- **Implement Huffman coding**

  Implement the `encode`, `buildTree`, and `createTable` methods in `HuffmanEncoder`

  Implement the `decode` method in `HuffmanDecoder`

- **Schedule and Color Intervals**

  Implement the `schedule` and `color` methods in `GreedyIntervals`

The project also contains additional files which you do not need to modify (but need to use). You will use `TestCorrectness` to test your code. For each part, you will get an output that you can match with the output I have given to verify whether your code is correct, or not. Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output. You must also run the TestTime file; that will help you uncover hidden bugs, complexities that are higher than what they should be, memory leaks, and other gaffes not easily detectable with small examples.

2 Before You Start

2.1 C++ Helpful Tips

For C++ programmers, remember to use DYNAMIC ALLOCATION for declaring all arrays. Remember to return an array from a function, you must use dynamic allocation. So, if you want to return an array x having length 10, it must be declared as **int *x = new int[10];** Once the array has served its purpose, de-allocate the memory using **delete[ ] x;** this will prevent a memory leak.

2.2 Priority Queue

You do not need to write code for a priority queue, but you need to use it for Huffman Encoding and Interval Coloring. Your task will be to create a priority queue and use its main operations – `add`, `poll`, `peek`, and `size`. You'll need to use a BinaryTreeNode priority queue, i.e., the items are binary tree nodes (which represent the nodes of the Huffman tree), as well as an integer priority queue (basically a heap). Both need to be minimum priority queues; the comparators for them have been written accordingly. **You should study the usage as in future assignments you may be asked to write your own comparators.**

C++

- To create a priority queue for BinaryTreeNodes, the syntax is:
  PriorityQueue<BinaryTreeNode*, BinaryTreeNodeComparator> pq;
  To create a priority queue for integers, the syntax is: PriorityQueue<int, compare_int> pq;
- To add an item (integer or BinaryTreeNode*) x, the syntax is `pq.add(x);`

Java

- To create a priority queue for BinaryTreeNodes, the syntax is: PriorityQueue<BinaryTreeNode> pq = new PriorityQueue<>(new BinaryTreeNodeComparator());
  To create a priority queue for integers, the syntax is: PriorityQueue<Integer> pq = new PriorityQueue<>(new IntegerComparator());
- To add an item (integer or BinaryTreeNode) x, the syntax is `pq.add(x);`

C#

- To create a priority queue for BinaryTreeNodes, the syntax is: PriorityQueue<BinaryTreeNode> pq = new PriorityQueue<BinaryTreeNode>(new BinaryTreeNodeComparator());
  To create a priority queue for integers, the syntax is:
  PriorityQueue<int> pq = new PriorityQueue<int>(new IntegerComparator());
- To add an item (integer or BinaryTreeNode) x, the syntax is `pq.add(x);`

For all three languages

- To retrieve the item at the top of the priority queue, the syntax is `pq.peek();`
- To retrieve and remove the item at the top of the priority queue, the syntax is `pq.poll();`
- To get the size of the queue, the syntax is `pq.size();`

2.3 Hashtable

You will be required to use hashtables for Huffman Encoding and Huffman Decoding. A hash table entry comprises of a *key* and *value* pair, whereby you can search the hash table for a particular key, and if it exists, retrieve the corresponding values back.

We use two hash tables – `charToEncodingMapping` for Huffman encoding and `encodingToCharMapping` for Huffman decoding. The first is used to get the encoding for each character, and then to encode the string, whereas the latter is used to decode the Huffman-encoded string back; hence, for the former, keys are of type `char` and values are of type `string` and vice-versa for the latter.

For your code, *ht* is either `charToEncodingMapping` or `encodingToCharMapping`, depending on whether you are working on encoding or decoding. Note that the type of *key* and *value* have to be modified accordingly. Refer to `TestCorrectness` file to learn more stuff.

C++. To create a hash table object *ht* whose keys are of type *int* and values are of type *string*, the syntax is unordered_map<int, string> ht. You are not required to create a hash table as they have already been created, but it is good to know about them. What you will need are the following:

- To insert a value v with key k, the syntax is ht[k] = v;
- To check whether or not *ht* contains a key k, the syntax is: if(ht.find(k) != ht.end()). The statement inside the if evaluates to *true* if *ht* contains k.
- To retrieve the value v corresponding to a key k, the syntax is: string v = ht[k];

Java. To create a hash table object *ht* whose keys are of type *int* and values are of type *string*, the syntax is Hashtable<Integer, String> ht = new Hashtable<>(). You are not required to create a hash table as they have already been created, but it is good to know about them. What you will need are the following:

- To insert a value v with key k, the syntax is ht.put(k, v);
- To check whether or not *ht* contains a key k, the syntax is: if(ht.containsKey(k)). The statement inside the if evaluates to *true* if *ht* contains k.
- To retrieve the value v corresponding to a key k, the syntax is: string v = ht.get(k);

C#. To create a hash table object *ht*, the syntax is Hashtable ht = new Hashtable().[1] You are not required to create a hash table as they have already been created, but it is good to know about them. What you will need are the following:

- To insert a value v with a key k, the syntax is ht.Add(k, v);
- To check whether or not *ht* contains a key k, the syntax is: if(ht.ContainsKey(k)). The statement inside the if evaluates to *true* if *ht* contains k.
- To retrieve the value v corresponding to a key k, the syntax is: String v = (String) ht[k]; Note the need to typecast over here due to the absence of generics.

[1] You can use generics if you use a Dictionary instead of Hashtable, which apparently is slower.

2.4 Dynamic Arrays

Here, you will use their C++/Java/C# implementations of dynamic arrays, which are named respectively **vector**, **ArrayList**, and **List**. Typically, this encompasses use of generics, whereby you can create dynamic arrays of any type (and not just integer). However, you will create integer dynamic arrays here; the syntax is pretty self explanatory on how to extend this to other types (such as char, string, or even objects of a class).

- In C++, to create an integer vector use: `vector<int> name;`

  The size is given by `name.size()`.

  To add a number (say 15) at the end of the vector, the syntax is `name.push_back(15)`.

  To remove the last number, the syntax is `name.pop_back()`.

  To access the number at a particular index (say 4), the syntax is `name[4]`.

- In Java, to create an integer ArrayList use: `ArrayList<Integer> name = new ArrayList<>();`

  The size is given by `name.size()`.

  To add a number (say 15) at the end of the array list, the syntax is `name.add(15)`.

  To remove the last number, the syntax is `name.remove(name.size() - 1)`.

  To access the number at a particular index (say 4), the syntax is `name.get(4)`.

- In C#, to create an integer List use: `List<int> name = new List<int>();`

  The size is given by `name.Count`.

  To add a number (say 15) at the end of the array list, the syntax is `name.Add(15)`.

  To remove the last number, the syntax is `name.RemoveAt(name.Count - 1)`.

  To access the number at a particular index (say 4), the syntax is `name[4]`.

3 Improved Partition (20 points)

The quicksort partition technique that you have learnt in class has a major problem. I'll illustrate the problem first. Suppose, you find the median of the array in $O(n)$ time. You use the median as pivot to partition the array. Ideally, this will split the array into two roughly equal halves and ultimately result in an $O(n \log n)$ time quick-sort algorithm. Recall the way the partition algorithm has been designed – all numbers less than equal to the median will be on the left half. So, if you have too many occurrences of the median, then you no longer get the half split, leading ultimately to an $O(n^2)$ time algorithm. For example, say the array is $[1, 30, 5, 5, 3, 5, 5, 5]$; then the median is 5. The split you get is $[1, 5, 5, 5, 3, 5]$ and $[30]$; clearly, this is not a half split (in fact, nowhere near).

The main idea is to fix this is to use two partition indexes based on $pivot$ – $lowerPartitionIndex$ and $upperPartitionIndex$, defined as follows:

- for $i \geq left$ and $i < lowerPartitionIndex$, we will have $array[i] < pivot$, i.e., all numbers before $lowerPartitionIndex$ are strictly smaller than $pivot$
- for $i \geq lowerPartitionIndex$ and $i \leq upperPartitionIndex$, we will have $array[i] = pivot$, i.e., all numbers from $lowerPartitionIndex$ to $upperPartitionIndex$ are equal to $pivot$
- for $i > upperPartitionIndex$ and $i \leq right$, we will have $array[i] > pivot$, i.e., all numbers after $upperPartitionIndex$ are strictly larger than $pivot$

The `partition` method in the `Partition` class will implement this logic, and it will return *lowerPartitionIndex* and *upperPartitionIndex* in array. Based on this, we can modify and write the quick sort code as follows:

```
void quicksortRandom(int[] array, int left, int right) {
    if (left <= right) {
        int pivot = getRandomPivot();
        int[] partitionIndex = partition(array, left, right, pivot);
        int lowerPartitionIndex = partitionIndex[0];
        int upperPartitionIndex = partitionIndex[1];
        quicksortRandom(array, left, lowerPartitionIndex - 1);
        quicksortRandom(array, upperPartitionIndex + 1, right);
    }
}
```

You will find the quick-sort code is provided to you. Your first task will be to fix partition. I'll sketch the idea, but I will leave the details to you. Having said that, you should be able to re-use most of the code from the previous assignment.

The main idea is that once you have partitioned using the strategy discussed in notes, you will carry out another step. Start scanning from $i = left$ and from $j = partitionIndex - 1$. Whenever you find that $array[i] = pivot$ and $array[j] < pivot$, you swap $array[i]$ and $array[j]$, increment i, and decrement j; we repeat this as long as $i < j$. Here's an illustrative figure.

[1, 9, 5, 2, 6, 5, 19, 2, 0, 3, 5, 34, 5, 90, 4, 1, 5, 2, 89, 5, 5, 5]

↓ partition using pivot = 5; I am choosing the last number but it can be anything (including the median). Following the strategy we have seen before:

↓ partitionIndex = 15 (shown in red below)

[1, 1, **5**, 2, 5, 5, 5, 2, 0, 3, 5, 2, 5, 5, **4**, **5**, 90, 34, 89, 19, 6, 9]

↓ Now, we fix the partition by bringing all the 5s (i.e., the pivots) together; I will illustrate one way.

↓ swap the 5 & 4:

[1, 1, 4, 2, **5**, 5, 5, 2, 0, 3, 5, **2**, 5, 5, 5, **5**, 90, 34, 89, 19, 6, 9]

↓ swap the 5 & 2:

[1, 1, 4, 2, 2, **5**, 5, 2, 0, **3**, 5, 5, 5, 5, 5, **5**, 90, 34, 89, 19, 6, 9]

↓ swap the 5 & 3:

[1, 1, 4, 2, 2, 3, **5**, 2, **0**, 5, 5, 5, 5, 5, 5, **5**, 90, 34, 89, 19, 6, 9]

↓ swap the 5 & 0:

[1, 1, 4, 2, 2, 3, 0, 2, **5**, 5, 5, 5, 5, 5, 5, **5**, 90, 34, 89, 19, 6, 9]

lowerPartitionIndex = 8 (shown in green above)
upperPartitionIndex = 15 (shown in red above)

Notice that *upperPartitionIndex* is the same as *partitionIndex*, and *lowerPartitionIndex* =

$upperPartitionIndex - number\ of\ occurrences\ of\ pivot + 1$.

Finally, remember the basic rules about partition (you must not violate them): do not use another array, but only variables, and it must run in linear time, i.e., $O(right - left + 1)$

4 Randomized Selection and Median of Medians (65 points)

For the randomized quick-select algorithm, the code remains essentially the same as previously, but we will slightly change the recursion calls:

- if k is greater than or equal to $(lowerPartitionIndex - left + 1)$ and k is less than or equal to $(upperPartitionIndex - left + 1)$, then $pivot$ is the answer.
- if k is less than $(lowerPartitionIndex - left + 1)$, then recursively find the answer to the left side of $lowerPartitionIndex$
- otherwise, recursively find the answer to the right side of $upperPartitionIndex$; make sure you modify the value of k appropriately for the recursion call.

For the median of medians quick-select algorithm, follow the pseudo-code in the notes, except that in the final portion where you either determine that pivot is the answer or make one of the two recursion calls, you will follow the same strategy as you have done for the randomized quick-select with $lowerPartitionIndex$ and $upperPartitionIndex$.

I will add some more details for clarity:

- The length of median array will $\lceil n/5 \rceil$, where $n = right - left + 1$. This means that if the $right - left + 1 = 20$, then length of median array is 4, and if $right - left + 1 = 21$ upto 25, then length of median array is 5.
- Blocks of the original array are of the from $left$ to $left + 4$, $left + 5$ to $left + 9$, and so on, until the last block ends at $right$.

  Thus, if $left = 10$ and $right = 23$, the blocks are from indexes 10 to 14, 15 to 19, and 20 to 23. Notice that each block is of length 5, except possibly the last one (which may be smaller; here it is of length 4).
- To sort each block of the array, use the insertion sort method given to you with the appropriate parameters. So, to sort a block from index i to index j (both inclusive), you will call the insertion sort method with the array, $left$ set to i, and $right$ set to j. Note that j is either $(i + 4)$ or it is $right$ (for the last block).
- After you have sorted each block using insertion sort, the median of each block is the middle element of the block, i.e., number at the index given by $(length\ of\ block)/2$

5 Sorting Strings of Lower Case English Letters (25 points)

We will sort an array of strings using radix sort; see the Google Drive folder for explanations.

The idea is to essentially treat each character in the string as a number between 1 - 26; thus, a is mapped to 1, b to 2, and so on all the way upto z being mapped to 26. This is achieved by subtracting 96 from the ASCII value of each character. Now, since a string may be shorter than another, we use 0 to treat a blank/null character. Hence, each character is represented in base-27 digit. Thus, if we compare the two strings $abcyx$ and $abcz$, then we treat them as (1)(2)(3)(25)(24)

and (1)(2)(3)(26)(0) respectively; the last 0 is padded essentially to make the strings have the same number of digits in base-27 notation.

We use the same idea as the radix sort algorithm on integers, but with the following changes:

- **radixSort** method:
  - max is the length of the longest string.
  - The loop for going over the digits now conceptually changes to going over the characters in each string. So, the outer while-loop changes to a loop that runs from $e = max - 1$ to $e = 0$; obviously, there is no need to multiply e by 10 anymore.
  - Within the inner for-loop, if e is greater than or equal to the length of $strings[i]$, then we let $digits[i] = 0$ because the string does not contain any character at index e (as it is shorter), else we let $digits[i]$ = the character at index e of $strings[i]$ – 96.
- **countSortOnLowerCaseCharacters** method is essentially the **countSortOnDigits** method:
  - the *digitCount* array should have length 27. Make necessary changes for the loops.
  - the *temp* array has to be a string array.

For C++ programmers, notice that strings array has been passed into the method as a pointer to an array of string pointers. So, when you create the temp array, make sure you follow suit: `string **temp = new string*[n]`. Also, each entry in the strings array is a string pointer; so, if you are trying to carry out an operation on a particular string pointer in this array, remember to use –> (arrow) instead of a . (dot). Finally, use dynamic allocation for all arrays, and remember to de-allocate memory for all arrays using `delete[]` once they have served their purpose. Refer to the `StringMergeSort` class for similar usages.

6 Huffman Coding (60 points)

Let σ (sigma) be the number of distinct characters. Let $alphabet[\ ]$ be an array of size σ which stores the characters. Let $frequencies[\ ]$ be an array of size σ which stores the frequency of each character, i.e., $frequencies[i]$ stores the frequency of $alphabet[i]$, $0 \leq i \leq \sigma - 1$.

You will need to use PriorityQueue and Hashing here; so, if you have not read how to use them, check out Section 2.2 and Section 2.3.

In Java or C#, to create a BinaryTreeNode with character c and value v, the syntax is: BinaryTreeNode node = new BinaryTreeNode(c, v);

In C++, to create a BinaryTreeNode with character c and value v, the syntax is: BinaryTreeNode *node = new BinaryTreeNode(c, v);

Implement the `encode`, `buildTree`, and `createTable` methods using the pseudo-codes.

Encode

- root = BUILDTREE()
- CREATETABLE(root, “”);
- For $i = 0$ to $\sigma - 1$, do the following:
  - let $c = alphabet[i]$ and $str = getEncoding(c)$
  - encodingLength = encodingLength + frequencies[i] $*$ length of str
  - tableSize = tableSize + length of str + 8

BuildTree

- Create a BinaryTreeNode minimum priority queue PQ
- For $i = 0$ to $\sigma - 1$, do the following:
  - create a binary tree node x for the character $alphabet[i]$ and value $frequencies[i]$
  - add x to PQ
- While PQ has more than than 1 node, do the following:
  - get the minimum binary tree node min from PQ and remove it
  - get the second minimum binary tree node $secondMin$ from PQ and remove it
  - create a binary tree node y with the character $'\backslash 0'$ and value the sum of the values of min and $secondMin$
  - assign min to be the left child of y and $secondMin$ to be the right child of y
  - add y to PQ
- return the minimum from PQ

CreateTable

- If node's left child and right child are both null, then insert node's character as *key* and the string encoding as *value* into the hash table `charToEncodingMapping`
- Else do the following:
  - If node's left child $\neq$ null, createTable(node's left child, encoding + "0")
  - If node's right child $\neq$ null, createTable(node's right child, encoding + "1")

Implement the `decode` method using the following pseudo-code.

Decode

- Let $encode =$ "" and $decodedMsg =$ ""
- Let n be the length of the encoded message
- For $i = 0$ to $n - 1$, do the following:
  - encode = encode + the character at index i of $encodedMsg$
  - If the hash table `encodingToCharMapping` contains the key *encode*, then
    - let c be the value for the key *encode* in `encodingToCharMapping`
    - decodedMsg = decodedMsg + c
    - encode = "";
- return decodedMsg

7 Greedy Intervals (30 points)

You will implement the greedy algorithms for scheduling and coloring intervals. For the first one, you will return a list of optimal intervals that are chosen by the algorithm. For the second one, you will return the minimum number of colors needed to paint the intervals such that no two intervals of the same color overlap; the assignment of colors is not necessary (that is rather a straightforward extension). I will sketch the idea of the algorithms, and leave the details to you; see the Google Drive folder for explanations.

Make sure that the 6 text files for interval scheduling and coloring are in the correct path; they are used by `TestCorrectness`.

7.1 Scheduling Intervals

- First you have to sort by the intervals by their end time. You will find the method already written. Study the use of comparators and usage of in-built sorting algorithms. In future assignments, you may be asked to implement them on your own.
- Now, create a list of intervals (basically a dynamic array) called *optimal* that you will return as the set of intervals chosen by the algorithm. Add the first interval in sorted to *optimal*; the first interval is always chosen.
- Scan through the list of sorted intervals.

  If the current interval in the sorted list starts after the end time of the last interval added to the list *optimal*, then add the current interval to *optimal*, else do nothing.
- Finally, return the *optimal* set of intervals.

7.2 Coloring Intervals

- First you have to sort by the intervals by their start time. Use the method already written.
- Create an integer priority queue, which will store the end time of intervals. Add the end time of the first interval to the priority queue. Also, create an integer counter to count the number of colors needed.
- Scan through the list of sorted intervals.

  If the current interval in the sorted list starts after the smallest end time among all intervals in the priority queue, then we can paint this interval with an existing color; so, simply remove the topmost entry in the priority queue (because we will use that color to paint this current interval and its end time needs to be updated). Otherwise, we need to create another color; so, increment the counter.

  Regardless, the priority queue has to be updated with the endtime of the current interval; so, insert the endtime of the current interval into the priority queue.
- Finally, return the counter.