Advanced Data Structure (COP 5536)
Programming Project Report

Mohit Mewara, UFID: 8413-2114, mohitmewara@ufl.edu, Spring 2017

***Abstract*—In this report, we will discuss the implementation of an encoding and decoding system in which a large input file is compressed to an encoded file. The original file is then retrieved from the encoded file with the help of a code table. Huffman coding is used to reduce the data size and the performance of Binary Heap, 4-Way Cache Heap, and Pairing Heap is analyzed for the generation of Huffman tree.**

***Index Terms*—Huffman Coding, Binary Heap, 4-Way Cache Heap, Pairing Heap, Encoding, Decoding**

I. Program Structure

There are in total 9 java files used in the project. The files - **HeapPerformanceCheck**, **encoder**, and **decoder** contain the main method and they can be used to check the performance of different heaps for generation of Huffman Tree, encoding the input file and decoding the encoded file respectively. The other java files are either User-Defined Data-Types or Utility files which are used by the above-mentioned files for successful execution of tasks.

II. Working Environment

Language: java version "1.8.0_121"; Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
Runtime Environment: Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
IDE: Eclipse Java EE IDE - Kepler Service Release 2

III. Compiling Procedure

The zip file contains the java files as well as the makefile. The makefile can be executed on a Linux Machine to compile all the java files by using the below command:

$ make

Once the make command is executed, all the java files defined in the make file are compiled into the corresponding class files using javac compiler. After this we can run the encoder class file by the following command:

$ java encoder <input_file_name>

The encoded.bin file is generated after the above command. Due to the change in format, the size of the file is compressed. Also, another file code_table.txt is generated which contains the unique Huffman code of each input which will be used while decoding the encoded file.

The original input file can be retrieved from the encoded file by using the following command:

$ java decoder <encoded_file > code_table_file >

The decoder.txt file is generated by the above command and it is identical to the original input file with respect to data and size.

IV. BinaryNodes.java

I have defined this node to store the various information about a number given as input. This node is used to implement Binary Heap, 4-Way Cache Heap, and the Huffman Tree. All the variables in this class file are global because we need to access them from a different class for storing/retrieving data.

int number - Stores the number present in the input file
int frequency - Stores the number of times that number is present in input file. When number repeats in the input file, the frequency of that number is incremented.
BinaryNode left – It points to the left child of the node.
BinaryNode right – It points to the right child of the node.
.

V. PairingNode.java

This class represents the node used to implement pairing heap.

BinaryNode node – The data stored in pairing node is of type BinaryNode because we need to store the number, frequency, and links between the input numbers. Also, the root of HuffmanTree created by pairing node is of type BinaryNode.
PairingNode leftChild – The pairing node has only 1 child and it is represented by this variable.
PairingNode next – It represents the next sibling of a node.
PairingNode prev – It represents the previous sibling of a node.

VI. BinaryHeap.java

This class contains a minimum binary heap represented by an array which is initialized when the constructor of the class is invoked. Each entry in this array represents a binary node object. The 0th index of the array is kept blank and the objects are stored from 1st index onwards. The degree of a binary heap is 2, which means that each node can have at-most 2 children.

BinaryNode[] arr – Represents the binary heap.
int count – Represents the number of elements present in the heap.

int capacity – Represents the size of array/heap.
int degree – Represents the degree of a node in heap.

A. public void insert (BinaryNode node)

Parameter: BinaryNode to be inserted.
Return Type: void
Running Time: O (log n)
Description: Inserts the new node as the leaf node. The parent of the new node is checked and if the node is minimum then it is swapped with the parent. The swapping of nodes continues till the node reach its appropriate position or it becomes the root of the heap.

B. public void resizeHeap ()

Parameter: void
Return Type: void
Running Time (Amortized cost): O (n)
Description: Initialized a new array of double size. The objects of the old array are copied in new array and the new array represents the binary heap.

C. public BinaryNode getMin ()

Parameter: void
Return Type: Root(BinaryNode) of the heap.
Running Time: O (1)
Description: This method is used to get the minimum node in the heap. The root is not removed from the heap and only the reference of the root is returned.

D. public BinaryNode extractMin ()

Parameter: void
Return Type: Root(BinaryNode) of the heap.
Running Time: O (log n)
Description: This method removes the root of the binary heap and returns it to the calling method. The last leaf becomes the new root and heapifyDown method is called to again implant the property of min binary heap.

E. public void heapifyDown (int index)

Parameter: The index of the array below which the adjustment of min property must be done.
Return Type: void
Running Time: O (log n)
Description: This method is called whenever the minimum property of the binary heap is violated. All children of the index are checked to ensure that the parent value is less than the child value. In case of violation, the nodes are swapped and the check continues till the min property of binary heap is restored.

VII. FOURCACHEHEAP.JAVA

This class also contains a minimum heap but the degree of 4-Way Cache Heap is 4, which means that each node can have at-most 4 children. It is represented by an array but while populating the data in array, the 0th, 1st and 2nd index is kept blank in the array.

BinaryNode[] arr – Represents the 4-way cache heap.
int count – Represents the number of elements present in the heap.
int capacity – Represents the size of array/heap.
int degree – Represents the degree of a node in heap.

The implementation of 4-way cache heap is similar to the binary heap. It also contains the methods – insert, resizeHeap, getMin, extractMin, and HeapifyDown with the same functionality of binary heap. However, each node in this heap can have at most 4 children and due to this the parent-child indexing in array is changed.

It is used because it optimizes the cache utilization for remove min and max operation.

VIII. PAIRINGHEAP.JAVA

The pairing heap class contains a root node representing the root of the heap. PairingNode is the data structure which is used to store the data of different nodes. The root node of the heap has the minimum value. Whenever the minimum object(root) is extracted from the heap, the children nodes of the root are combined and a new node is selected with minimum value.

PairingNode root – Represents the root of the Pairing Heap.
PairingNode[] elementsArray – Used to store the children of root node when the extractMin() method is invoked.

A. public boolean isEmpty ()

Parameter: void
Return Type: Boolean (True or False)
Running Time: O (1)
Description: It checks whether the pairing heap is empty or not. If the root of the heap is null then it is empty, otherwise it is not empty.

B. public PairingNode insert (BinaryNode node)

Parameter: Binary node which needs to be inserted in the heap. A new PairingNode is created using this node which is inserted in the heap.
Return Type: void
Running Time: O (1)
Description: This method is used to insert a new pairing node in the heap. The new node is compared with the existing root and the root is updated if the new node is lesser than the existing root.

C. public BinaryNode extractMin ()

Parameter: void
Return Type: The data(BinaryNode) of the root is returned.
Running Time (Amortized Cost): O (log n)
Description: The root of the heap is removed and the left child of it with the siblings of child are compared with each other to determine the minimum node. The minimum node becomes the new root and the one of the node becomes child of it. The other nodes become the siblings of child node.

D. public PairingNode compareAndAddNodes (PairingNode first, PairingNode second)

Parameter: Two PairingNodes which needs to be compared.

Return Type: One PairingNode which becomes the parent of the other.
Running Time: O (1)
Description: It is used by the combineSiblings() method to compare two nodes and determine which one is smaller than other. The smaller node becomes the parent and is returned to the calling method.

E. private PairingNode combineSiblings (PairingNode firstSibling)

Parameter: The child of root. It is the leftmost sibling of the nodes present at 1 level below the root.
Return Type: PairingNode - The new root of the heap.
Running Time (Amortized cost): O (log n)
Description: It is invoked when the root node is extracted. The child of the root along with its siblings are detached from the heap and then after comparisons, again combined to make a heap with the least element as the root. The Two-Pass scheme is used for combining siblings.

Pass 1:

- Subtrees are examined from left to right.
- The subtrees are melded from left to right. The subtrees are reduced to half the number.

Pass 2:

- The remaining subtrees are melded one by one from right to left.
- The last subtree remained will be the root of the new pairing heap.

F. private PairingNode[] doubleArray (PairingNode[] array, int index)

Parameter: The PairingNode array and index of the array.
Return Type: A new array of type PairingNode with double size.
Running Time (Amortized cost): O (n)
Description: Checks whether the index is less than the size of existing array. If the index is less, then the same array is retuned, else a new array of double size is returned. The entries of the old array are stored in the new array.

IX. HeapPerformanceCheck.java

This class builds the Huffman tree using Binary Heap, 4-Way Cache Heap, and Pairing heap. The time required for building the tree from 3 heaps is measured in this class.

A. public BinaryNode buildHuffmanBinaryHeap (HashMap<Integer, Integer> hash)

Parameter: A HashMap containing the input number as key and its frequency as value.
Return Type: The root of the HuffmanTree of BinaryNode type is returned.
Running Time: O (n log n)
Description: A Min Binary Heap is built by adding all the numbers present in the HashMap. Then, 2 minimum nodes are extracted from the heap and a single combined node is added in the heap. This process is continued until only one node is present in the heap. The last node represents the root of the HuffmanTree and when this process is completed we are left with the root of tree.

B. public BinaryNode buildHuffmanFourCache (HashMap<Integer, Integer> hash)

Parameter: A HashMap containing the input number as key and its frequency as value.
Return Type: The root of the HuffmanTree of BinaryNode type is returned.
Running Time: O (n log n)
Description: A Min 4-Way Cache Heap is built by adding all the numbers present in the HashMap. The process of finding root of HuffmanTree is same as that of the above method. The last node represents the root of the tree.

C. public BinaryNode buildHuffmanUsingPairingHeap (HashMap<Integer, Integer> hash)

Parameter: A HashMap containing the input number as key and its frequency as value.
Return Type: The root of the HuffmanTree of BinaryNode type is returned.
Running Time (Amortized Cost): O (n log n)
Description: The Pairing Heap is build using all the entries of the HashMap. After this, the 2 minimum nodes are extracted from the heap and a single combined node is inserted into the heap. This process continues till the heap becomes empty. The last node represents the root of the HuffmanTree.

X. InputReader.java

This is a utility class which contains methods for reading the input data file.

A. public Vector<Integer> readInputFile (String fileName)

Parameter: The input filename which is to be read.
Return Type: A Vector of Integer class containing all the numbers present in the input file.
Running Time: O (n)
Description: This method is used to read the input file.

B. public HashMap<Integer, Integer> generateFrequency (Vector<Integer> input)

Parameter: The Vector of Integer class containing all the numbers present in the input file.
Return Type: A HashMap with the number as the key and the frequency as the value.
Running Time: O (n)
Description: The input Vector is iterated and a HashMap is created with the number as the Key and its frequency as the value. When a duplicate entry of the number is detected, the

value corresponding to that number in HashMap is incremented.

C. public HashMap<Integer, String> readCodeTableFile (String fileName)

Parameter: The input filename of the code table which is to be read.
Return Type: A HashMap with a number as the Key and the binary code of type String as the Value.
Running Time: O (n)
Description: The code table written by the encoder is used by the decoder to decode the encoded file. This method is used to read the code table which contains the numbers and the corresponding codes of that number.

XI. ENCODER.JAVA

This class takes the input file as input and then generate Huffman Tree, Huffman Codes for every input, and eventually write encoded.bin and code_table.txt file. Due to compression, the data size of the encoded.bin file is lesser than the original file. The code_table.txt file is used by decoder afterwards to extract the original data from the encoded file.

A. public BinaryNode buildHuffmanFourCache (HashMap<Integer, Integer> hash)

Parameter: A HashMap containing the input number as key and its frequency as value.
Return Type: The root of the HuffmanTree of BinaryNode type is returned.
Running Time: O (n log n)
Description: A Min 4-Way Cache Heap is built by adding all the numbers present in the HashMap. Then, 2 minimum nodes are extracted from the heap and a single combined node is added to the heap. This process is continued until only one node is present in the heap. The last node represents the root of the HuffmanTree and when this process is completed we are left with the root of the tree.

B. public HashMap<Integer, String> generateHuffmanCodes (BinaryNode node)

Parameter: The root of the HuffmanTree is passed as input parameter.
Return Type: The HashMap with the number as the Key and the corresponding unique code of String Type as the Value.
Running Time: O (n)
Description: This method calls an overloaded method recursively to compute the unique code for each number. The numbers are present at the leaf of the tree. The algorithm traverses to the leaf of the tree and writes the code of the number in the HashMap. For a left child traversal, the path string is appended with a 0 and for a right child traversal, the path string is appended with a 1. So, when the recursion reaches the leaf node, we get a string representing a path from the root to the leaf. This path string represents the code of the number and is added in the HashMap.

C. public void codeTableWriter (Map<Integer, String> codeMap)

Parameter: The Map containing the number as the Key and the binary code as the Value.
Return Type: A HashMap with a number as the Key and the binary code of type String as the Value.
Running Time: O (n)
Description: The entries of the map are written in a file named “code_table.txt”.

D. public void encodedBinaryWriter(Map<Integer, String> codeMap, Vector<Integer> input)

Parameter: The Map containing the number as the Key and the binary code as the Value; The Vector containing the input numbers.
Return Type: void.
Running Time: O (n)
Description: This method writes the “encoded.bin” file. The input Vector is iterated and for each number, the code is retrieved from the Map. The code is in binary format but each binary (0 and 1) is saved as a character. So, we define a byte and set each bit of this byte with the binary format string. By doing so, we compressed the data of 8 bytes into a single byte. After all the input is converted into bytes, we save the byte array into the “encoded.bin” file.

XII. DECODER.JAVA

This class takes the code_table.txt file as input to generate Huffman tree. The nodes of the trees are generated by reading the different codes and appending the number at the leaf. Once the Huffman tree is generated, the decoded.txt file is generated using the “encoded.bin” file. The decoded.txt file contains the same message as present in the original input file.

A. public BinaryNode buildDecodedHuffmanTree ()

Parameter: void
Return Type: The root of the HuffmanTree of type BinaryNode.
Running Time: O (n * max. length of characters in binary format for an input), where n is the number of unique numbers.
Description: This method builds a HuffmanTree using the unique codes of each input. The code is traversed from the root and a new node is added if the code path is not present in the tree. Once the path is added to the tree, the number is added to the tree as a leaf. This mechanism is repeated for each input.

B. public void generateDecodedFile (BinaryNode rootNode)

Parameter: The root node of the decoded Huffman tree.
Return Type: void.
Running Time: O (n), where n is the number of bits present in the input file.
Description: This method creates a decoded file which contains the same inputs as were present in the original input

file. First, the "encoded.bin" file is read. Then the bytes of the file are written in a byte array. Each byte of this array contains 8 bits. So, the HuffmanTree is traversed per the values present in the bits and whenever a leaf is detected in the tree, the number is written into the decoded file. These steps are repeating until we reach the end of the file.

XIII. Performance Analysis of Different Heaps

The HuffmanTree is build using the three heaps – Binary Heap, 4-way Cache heap, and the Priority Heap. The tree is constructed for 10 times using each heap and the average time is computed.

The performance of the 4-Way Cache Heap is better most of the times. However, there were some instances when the Binary Heap performed better but a majority of time, 4-Way Cache Heap outperforms Binary and Pairing Heap in terms of execution time.

The experiment is performed over the "sample_input_large.txt" file. The results of each iteration are recorded in the below table after executing the program on CISE storm server.

Iteration Number	**Binary Heap (milliseconds)**	**4-Way Cache Heap (milliseconds)**	**Pairing Heap (milliseconds)**
1	472	414	995
2	402	377	1111
3	525	475	1173
4	435	422	1044
5	519	464	1234
6	415	372	934
7	489	381	945
8	472	447	1120
9	498	418	957
10	412	477	1151
Average Time	**463**	**424**	**1066**

Table 1: The Average Time for Building Huffman Tree using different heaps

From the above experiment, it is concluded that the 4-Way Cache Heap performs better than other heaps most of the times. **Reason:** This is due to the fact that there is less number of cache misses in the 4-Way Cache Heap. While the system read a cache, most of the times the parent and children are present in the same cache. So, when we perform actions like insert or extract minimum, the data is available in the same cache and the overhead for extracting data again from the file system is avoided.

One probable reason for pairing heap being the slowest among all three is the reason that the cache allocation is not optimally used by pairing heap in comparison to binary or 4-way cache.

```
stormx:17% java HeapPerformanceCheck
Time using binary heap (millisecond):491
Time using 4-way heap (millisecond):511
Time using pairing heap (millisecond):1022
stormx:18% java HeapPerformanceCheck
Time using binary heap (millisecond):472
Time using 4-way heap (millisecond):414
Time using pairing heap (millisecond):995
stormx:19% java HeapPerformanceCheck
Time using binary heap (millisecond):402
Time using 4-way heap (millisecond):377
Time using pairing heap (millisecond):1111
stormx:20% java HeapPerformanceCheck
Time using binary heap (millisecond):525
Time using 4-way heap (millisecond):475
Time using pairing heap (millisecond):1173
stormx:21% java HeapPerformanceCheck
Time using binary heap (millisecond):435
Time using 4-way heap (millisecond):422
Time using pairing heap (millisecond):1044
stormx:22% java HeapPerformanceCheck
Time using binary heap (millisecond):519
Time using 4-way heap (millisecond):464
Time using pairing heap (millisecond):1234
stormx:23% java HeapPerformanceCheck
Time using binary heap (millisecond):415
Time using 4-way heap (millisecond):372
Time using pairing heap (millisecond):934
stormx:24% java HeapPerformanceCheck
Time using binary heap (millisecond):489
Time using 4-way heap (millisecond):381
Time using pairing heap (millisecond):945
stormx:25% java HeapPerformanceCheck
Time using binary heap (millisecond):472
Time using 4-way heap (millisecond):447
Time using pairing heap (millisecond):1120
stormx:26% java HeapPerformanceCheck
Time using binary heap (millisecond):498
Time using 4-way heap (millisecond):418
Time using pairing heap (millisecond):957
stormx:27%
stormx:27% java HeapPerformanceCheck
Time using binary heap (millisecond):412
Time using 4-way heap (millisecond):477
Time using pairing heap (millisecond):1151
```

Figure 1: Execution time for Building Huffman Tree using different heaps

XIV. Encoding Algorithm

We have discussed above in the "encoder.java" section about the different methods implemented in the java file. We have used these methods and following steps to implement encoding:

1. The Input file is read and the numbers with their frequency are stored in a hash map. *(Complexity – O (n))*
2. From the above experiment, it is confirmed that the 4-Way Cache Heap performs better. So, we build a

Huffman code tree using this heap. The algorithm for creating the tree is discussed in section VII. *(Complexity – O (n log n))*, where log n is the height of the tree.

3. Once the tree is build, we generate the Huffman codes for each input number. As we know that the numbers are present at the leaf of the tree, we traverse from the root to the leaf and when we lay-off, we create an entry in the hash map with the number as Key and the code as the Value. We keep a record of the path by saving that in a string. When we move to a left child then 0 is appended to the path, and when we move to the right, 1 is appended to the path. *(Complexity – O (n))*
4. Now we have unique codes for each input. So, we write the input and codes to a file named "code_table.txt". *(Complexity – O (n))*
5. The last step of the algorithm is to create an "encoded.bin" file. This is a compressed file when compared with the original data file. Each input is first converted to its code and then for 8 characters, we write only 1 character in the encoded.bin file. We set the 8 bits of a single byte with the value of those 8 characters. *(Complexity – O (n))*

Total Time Complexity – O (n log n),

Where,
n is the number of input numbers.
log n is the height of the Huffman Tree.

The above algorithm is tested on the storm server and the below screenshots were captured. For "sample_input_large.txt" file, the time required to execute encoder.java is 10 seconds.

```
stormx:50% java encoder sample_input_large.txt
Encoded file created: encoded.bin
Code Table file created: code_table.txt
Time for encoding (milliseconds): 10196
```

Figure 2: The output of encoder.java file

```
stormx:51% ls -lrth
total 118M
-rw-------+ 1 mmewara grad 1.8K Apr  1 15:25 FourCacheHeap.java
-rw-------+ 1 mmewara grad  442 Apr  1 15:25 BinaryNode.java
-rw-------+ 1 mmewara grad 3.5K Apr  1 17:41 HeapPerformanceCheck.java
-rw-------+ 1 mmewara grad 1.4K Apr  4 17:06 BinaryHeap.java
-rw-------+ 1 mmewara grad 4.4K Apr  5 03:51 encoder.java
-rw-------+ 1 mmewara grad  259 Apr  5 10:36 PairingNode.java
-rw-------+ 1 mmewara grad 2.6K Apr  5 11:12 PairingHeap.java
-rw-------+ 1 mmewara grad 2.2K Apr  5 13:18 InputReader.java
-rw-------+ 1 mmewara grad 4.3K Apr  5 16:19 decoder.java
-rw-------+ 1 mmewara grad  477 Apr  5 16:48 BinaryNode.class
-rw-------+ 1 mmewara grad 3.0K Apr  5 16:48 InputReader.class
-rw-------+ 1 mmewara grad 1.5K Apr  5 16:48 FourCacheHeap.class
-rw-------+ 1 mmewara grad  772 Apr  5 16:49 decoder$TraverseBit.class
-rw-------+ 1 mmewara grad 3.6K Apr  5 16:49 decoder.class
-rw-------+ 1 mmewara grad  67M Apr  5 16:50 sample_input_large.txt
-rw-------+ 1 mmewara grad 5.4K Apr  5 16:51 encoder.class
-rw-------+ 1 mmewara grad  27M Apr  5 16:51 code_table.txt
-rw-------+ 1 mmewara grad  24M Apr  5 16:51 encoded.bin
```

Figure 3: code_table.txt and encoded.bin generated after execution of encoder.java

XV. Decoding Algorithm

The decoding section decodes the "encoded.bin" file back to the original file with the name "decoded.txt". The methods written in decoder.java are used to decode the file. The decoding algorithm performs the following steps:

1. The "code_table.txt" file is read and saved in a hash map. *(Complexity – O (n))*
2. The codes of each input are read and the decoded Huffman tree is created. The code is traversed from the root and a new node is added if the code path is not present in the tree. Once the path is added to the tree, the number is added to the tree as a leaf. This mechanism is repeated for each input. *(Complexity – O (n * h)), Where,*

   n is the count of numbers in the input,
   h is length of largest bit of code of an input or the largest height of the decoded huffman tree.

3. Once we get the Huffman tree, we read the "encoded.bin" file byte by byte and store that in a byte array. Then we read a byte and for each bit of this byte, we traverse through the decoded Huffman tree. When we reach a leaf, we write that number in the output file – "decoded.txt". This process continues till we reach the end of the file. *(Complexity – O (m))*

Total Time Complexity –
O (8 * m) + *O (n * h)),*
{Because each byte has 8 bits}
= O (m) + O (n * length of largest bit of code of an input)),

Where,
m is the number of bytes' present in the encoded.bin file.
n is the count of numbers in code_table.txt
h is length of largest bit of code of an input or the largest height of the decoded huffman tree

= max {O (m), O (n * h)}

For a small sample file, where the duplication of numbers is very less O (n * h) would be an upper bound but for a large input file with a lot of duplicate numbers, O (m) would be an upper bound. So, the complexity can vary between these two upper bounds.

However, as we are talking about a sample input file with 100 million lines, so in this case **O (m)** would take over the complexity.

Therefore, the complexity is O (m).

The above algorithm is tested on the storm server and the below screenshots were captured. For "sample_input_large.txt" file, the time required to execute decoder.java is 10 seconds.

```
stormx:52% java decoder encoded.bin code_table.txt
Decoded file created: decoded.txt
Time for decoding (milliseconds): 10150
```

Figure 4: The output of decoder.java file

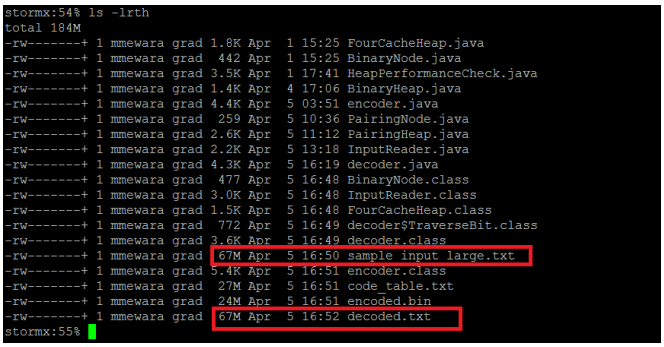

Figure 5: decoded.txt is generated after execution of decoder.java

XVI. Conclusion

The output of decoder matches with the original file and the size of both files are same.

I have also run the code for 100-million-line input file and it worked fine for that too. The runtime for encoder was 120 seconds and the runtime of decoder was 70 seconds. The size of the decoded.txt file generated was same as that of the original file.

Also, I have run the program to ensure that all the exceptions are thrown whenever an input file or other things are missing.

References

[1] Sartaj Sahni's (COP-5536) Lecture 7, 10 and 15

[2] Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.