University of Nevada, Las Vegas Computer Science 477/677 Spring 2025

Assignment 2: Due Saturday February 28, 2026 23:59:59

Follow instructions given by our Graduate Assistant Rakibul Hassan `hassar2@unlv.nevada.edu` on how to turn in the assignment.

Name:_____

You are permitted to work in groups, get help from others, read books, and use the internet.

1. A *binary code* on any alphabet Σ is a function which assigns a binary string to each member of Σ. That code is *prefix-free* if no symbol is assigned a string which is a prefix of the assigned string for any other symbol.

   (i) Huffman's algorithm creates an optimal prefix-free binary code (or prefix code, or Huffman code) for any finite weighted alphabet, where weights are proportional to the expected frequencies of symbols in a message. Construct an optimal prefix-free code for the following weighted alphabet.

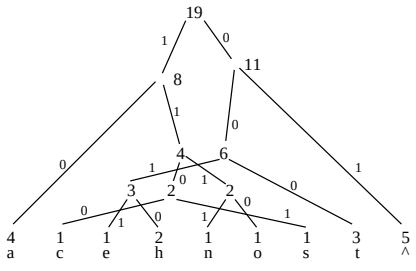
   Huffman's algorithm requires making arbitrary decisions, such as which of two equal weights to combine, or whether to label a branch "0" or "1". Thus there are multiple optimal prefix codes. In the case of the weighted alphabet below, the number of optimal codes is over a thousand.

a	6	
b	4	
c	2	
d	7	
e	13	
f	8	
g	4	
h	9	

   Below is a weighted alphabet with its prefix code, where $\wedge$ denotes blank.

   (iii) Draw a Huffman tree for the code.

a	4	10
c	1	1100
e	1	0011
h	2	0010
n	1	1111
i	1	1110
s	1	1101
t	3	000
∧	5	01



   The only requirement for labeling branches is that the two from each internal node must be labeled 0 and 1. It is not necessary for the left branch to be labeled 0.

(ii) A message is encrypted using that code. Find the original message.

0 0 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 1 1 0 0 1 0 0 1 0 1 0 0 0 0

the cat sat on a hat

2. Given a loop in a program, a *loop invariant* for that loop is a Boolean expression I such that
   1. I is true before the first iteration of the loop.
   2. If I is true before an iteration of the loop, it is true after that iteration.

   It follows logically that the loop invariant is true after the loop terminates. Here is a simple example. The purpose of thie code is to compute write twice its input parameter, n, provided n is non-negative. Loop invariants can be used to verify correctness, as in this simple example.

```
   void double(int n)
    {
(1)  int m = n;
(2)  int p = 0;
(3)  // I holds here
(4)  while(m > 0)
(5)   {
(6)    // If I holds here
(7)    m = m-1;
(8)    p = p+2;
(9)   }
(10)  // then I holds here
(11)  cout << p << endl;
    }
```

   The loop invariant is the statement I: $2n = 2m + p$. We need to verify that I satisfies conditions 1. and 2. above.

   Condition 1: I holds before the loop first iterates, at line (3), since $2m + p = 2n + 0 = 2n$.

   Condition 2: We need to assume I holds at line (6) and then prove it holds at line (9). Let m' and p' be the values of m and p before a given iteration, and let m" and p" be the values of those veriables after that iteration. Thus m" $= m$'-1 and p" $= p$'$+2$. We assume I holds at line (6), namely that $2n = 2m$'$+p$'. Then $2m$"$+p$" $= 2(m$'$-1) + p$'$+2) = 2m$'$+p$'$= 2n$, hence I holds at line (9),

   Finally, we show how we use the loop invariant to prove correctness of the code, namely that the output is $2n$. I holds at line (10), namely $2n = 2m + p = p$. Thus the output is $p = 2n$.

   Find the loop invariant of the loop in the C++ code below, which computes the product nxj provided n is non-negative. Then use that loop invariant to prove correctness of the code.

```
float product(float x, int n)
 {
  int m = n;
  float y = x;
  float z = 0.0;
  while(m > 0)
   {
    if(m%2) z = z+y;
    y = y+y;
    m = m/2:
   }
  cout << z
 }
```

The loop invariant is $xn = ym + z$.

Proof of the Loop Invariant. Condition 1 holds because initially $y = x$, $m = n$, and $z = 0.0$.

To verify condition 2, we let m', y', and z' be the values of the variables m, y, and z at the beginning of an iteration of the loop, and m", y", and z" the values of those variables at the end of that iteration.

We assume that the loop invariant holds at the beginning of the iteration, namely that $xn = y'm' + z'$: we need to prove that the invariant holds at the end of the iteration, namely that $xn = y"m" + z"$.

By the loop condition, we know that $m' > 0$. We consider two cases: m' even and m' odd. If m' is even, then $m'\%2 = 0$, which is interpreted as false, hence $z" = z'$, $y" = 2y'$, and $m" = \frac{1}{2}m'$. Thus, $y"m" + z" = (2y')(\frac{1}{2}m') + z' = y'm' + z' = xn$.

If m' is odd, then $m'\%2 = 1$, which is interpreted as true, hence $z" = z'+y'$, $y" = 2y'$, and $m" = \frac{1}{2}(m'-1)$. Thus, $y"m" + z" = (2y')(\frac{1}{2}(m'-1)) + z' + y' = y'(m'-1) + z' + y' = y'm' - y' + z' + y' = y'm' + z' = xn$.

In either case, the loop invariant hold at the end of the loop.

Proof of Correctness. After the loop exit, the loop invariant holds and $m = 0$. Thus $xn = ym + z = z$ The output is z, which is the correct product of x and n.

3. Merging, mergesort, polyphase mergesort.

Merging is combining two sorted lists into one sorted list. For example, the sorted lists BHMQVWZ and ANPRTUXY can be merged into the sorted list ABHMNPQRTUVWXY.

(a) What is the asymptotic time complextiy of merging two sorted lists each of length n?

Mergesort is a divide and conquer algorithm, and takes $O(n \log n)$ time to sort a list of length n, using the folowing steps.

(i) Divide the items of the list into two sublists.

(ii) Use mergesort to recursively sort each of those sublists.

(iii) Merge the two sublists.

(b) Explain why the time complexity of mergesort is $O(n \log n)$.

(c) Walk through mergesort of the list ALQWVLXYBZTMNPHS.

Polyphase Mergesort is not a modification of mergesort, but it does use merging.

We define a *run* of a list L to be a maximal sorted sublist of L. For example, if $L = XMGHKNQFTSNZ$ the runs of L are $X, M, GHKNQ, FT, S, NZ$. A *phase* of polyphase mergesort combines a list into a new list with fewer runs, using the following steps. Let L_1 be a list.

(i) Partition L_1 into runs.

(ii) Merge the first two runs of L_1 and write the resulting list to L_2.

(iii) Merge the next two runs of L_1 and write the resulting list to L_2.

(iv) Repeat the prevfious steps, merging two runs of L_1 and writing to L_2. If there is one run left over, simply copy it to L_2.

(v) The resulting list will be sorted.

(d) Prove that the time complexity of polyphase mergesort on a list of length n is $O(n \log n)$.

(e) Walk through polyphase mergesort to sort the list BCQWERYIOPNLKHFMTZDW.

4. *Quicksort* is also a divide and conquer sorting algorithm. It is said that quicksort and mergesort are *dual*. In both cases, the original list is partitioned into two lists, each list is recursively sorted, and the lists are then combined to form the final sorted list. It is the manner of partitioning and combining that disitinguishes the algorithms. In mergesort, the list is partitioned quickly and the sublists combined carefully, while in quicksort, the list is partitioned carefully and the sublists combined quickly.

   A list L is sorted using the following steps.

   (i) A *pivot* entry of L is chosen. (The ideal choice of pivot is the median of L, but it is not practical to insist on that choice.)

   (ii) L is partitioned into two lists A and B, where the entries of A are the entries of L whose values are less than the pivot, while the B consists of the entries of L larger than the pivot.

   (iii) A and B are sorted using quicksort.

   (iv) The concatenation A, followed by the pivot, followed by B, is now sorted.

   (a) What is the time complexity of quicksort on a list of length n? Give the worst case answer and the average case answer. For the average case, assume that initially the entries of L are randomly permuted.

   Despite its apparent simplicity, coding quicksort is full of pitfalls. We will discuss those, as well as strategies for avoiding them.

5. Build a treap by inserting eight items in the order given in the following table. The first column is the item, the second its priority. Use min-heap order.

   Draw the treap after each insertion and after each rotation. My answer consists of 12 figures.

W	8
F	9
Q	2
H	5
A	6
B	3
P	7
M	1