University of Nevada, Las Vegas Computer Science 477/677 Spring 2021

Answers to Assignment 3 due Thursday February 25, 2021

Name:____

You are permitted to work in groups, get help from others, read books, and use the internet. Turn in the completed assignment on Canvas, using instructions given to you by the grader, Mr. Nicholas Heerdt, by 11:59 PM February 25.

1. Work Problem 2.4 on page 71 of your textbook.

   The time complexity of Algorithm A satisfies the recurrence
   $T(n) = 5T(n/2) + n$
   The solution is $T(n) = \Theta(n^{\log_2 5}) \approx \Theta(n^{2.32})$

   The time complexity of Algorithm B satisfies the recurrence
   $T(n) = 2T(n-1) + 1$
   The solution is $T(n) = \Theta(2^n)$

   The time complexity of Algorithm C satisfies the recurrence
   $T(n) = 9T(n/3) + n^2$
   The solution is $T(n) = \Theta(n^2 \log n)$

   Thus, the best choice is Algorithm C.

2. Work Problem 2.16 on page 73 of your textook.

   We can use the rule that any integer is less than ∞. We assume the array indices start with 1.

   (a) Compute $A[2^i]$ for $i = 0, 1, \ldots$ until we find the first i for which $A[2^i] \geq x$. We are guaranteed to find such an i, since $A[2^i] = \infty$ for sufficiently large i. Note that $2^i = \Theta(n)$, hence $i = \Theta(\log n)$. This step takes $\Theta(i) = \Theta(\log n)$ time.

   (b) If $A[2^i] = x$, we have found x, else if $i = 0$ and $A[1] > x$, x is not an item of the array. Otherwise, proceed to the next step.

   (c) We know that $A[2^{i-1}] < x < A[2^i]$. Use binary search to determine whether x is an item in that subarray. That search takes $O(i) = O(\log n)$ time.

   Thus, the time complexity of the algorithm is $O(\log n)$.

3. Work Problem 2.20 on page 74 of your textbook.

   We do not use the comparison model. Here is code for a solution. There is a similarity with bucket sort.

```
int minitem = minimum value of x[i]; // O(n) time
int frequency[M+1]; // assume indices are 0, 1, ... M
for(int i = 1; i <= n; i++)
  frequency[A[i]-minitem]++;
```

```
// frequency[j] is the number of copies of j+minitem in A
int i = 0;
for(int m = 0; m <= M; m++)
  for(int f = 0; f < frequency[m]; f++)
    A[++i] = minitem + m;
// frequency[j] copies of j+minitem are written consecutively to A
```

5. Walk through the steps of heapsort for the array UBRYPQSVFMTX. Show the array after each exchange. I'll help you get started:

```
123456789abc Hexadecimal indices of the array implementation of max-heap.
UBRYPQSVFMTX Initial array.
UBRYPXSVFMTQ Start heapification: bottom-up bubbledown.  Bubbledown(Q).
UBRYTXSVFMPQ Bubbledown(P).
UBXYTRSVFMPQ Bubbledown(R).
UYXBTRSVFMPQ Bubbledown(B).
UYXVTRSBFMPQ Bubbledown(B).
YUXVTRSBFMPQ Bubbledown(U).
YVXUTRSBFMPQ Bubbledown(U).  Heap order established.
QVXUTRSBFMPY Phase 1 begins.  Swap Y to final position.  Heap size is 11.
XVQUTRSBFMPY Bubbledown(Q).
XVSUTRQBFMPY Bubbledown(Q).  Heap order restored.  End Phase 1.
PVSUTRQBFMXY Phase 2 begins.  Swap X to final position.  Heap size is 10.
VPSUTRQBFMXY Bubbledown(P).
VUSPTRQBFMXY Bubbledown(P).  Heap order restored.  End Phase 2.
MUSPTRQBFVXY Phase 3 begins.  Swap V to final position.  Heap size is 9.
UMSPTRQBFVXY Bubbledown(M).
UTSPMRQBFVXY Bubbledown(M).  Heap order restored.  End Phase 3.
FTSPMRQBUVXY Phase 4 begins.  Swap U to final position.  Heap size is 8.
TFSPMRQBUVXY Bubbledown(F).
TPSFMRQBUVXY Bubbledown(F).  Heap order restored.  End Phase 4.
BPSFMRQTUVXY Phase 5 begins.  Swap T to final position.  Heap size is 7.
SPBFMRQTUVXY Bubbledown(B).
SPRFMBQTUVXY Bubbledown(B).  Heap order restored.  End Phase 5.
QPRFMBSTUVXY Phase 6 begins.  Swap S to final position.  Heap size is 6.
RPQFMBSTUVXY Bubbledown(Q).  Heap order restored.  End Phase 6.
BPQFMRSTUVXY Phase 7 begins.  Swap R to final position.  Heap size is 5.
QPBFMRSTUVXY Bubbledown(B).  Heap order restored.  End Phase 7.
MPBFQRSTUVXY Phase 8 begins.  Swap Q to final position.  Heap size is 4.
PMBFQRSTUVXY Bubbledown(M).  Heap order restored.  End Phase 8.
FMBPQRSTUVXY Phase 9 begins.  Swap P to final position.  Heap size is 3.
MFBPQRSTUVXY Bubbledown(F).  Heap order restored.  End Phase 9.
BFMPQRSTUVXY Phase 10 begins.  Swap M to final position.  Heap size is 2.
FBMPQRSTUVXY Bubbledown(B).  Heap order restored.  End Phase 10.
BFMPQRSTUVXY Bubbledown(B).  Swap F to final position.  Array is sorted.
```

6. The following function computes the product of two positive integers. Verify that that `p + cd = ab` is a loop invariant for this code, and use the loop invariant to show that `product(a,b)` returns `a*b`.

```
int product(int a, int b)
 {
  assert (a > 0 and b > 0);
  int c = a;
  int d = b;
  int p = 0;
  while(d > 0)
   {
    if(d % 2) p = p+c;
    c = c+c;
    d = d/2;
   }
  return p;
 }
```

Before the first iteration, $c = a$, $b = d$ and $p = 0$, thus $cd + p = ab$, *i.e.* the loop invariant holds.

Consider one iteration of the loop. let a', b' and p' be the values of c, d and p before the iteration, and let c, d, and p the values after the iteration. Suppose the loop invariant holds before the iteration: then $c'd' + p' = ab$. We need to prove the loop invariant holds after the iteration.

There are two cases, d' is odd or even. If d' is odd, then $p = p' + c'$ and $d = \frac{d'-1}{2}$. If d' is even, then $p = p'$ and $d = \frac{d}{2}$. In either case, $c = 2c'$.

If d' is odd, then

$$\begin{aligned} cd + p &= 2c'\frac{d'-1}{2} + p' + c' && (1)\\ &= c'(d'-1) + p' + c' && (2)\\ &= c'd' + p' && (3)\\ &= ab && (4) \end{aligned}$$

On the other hand, if d' is even, then

$$\begin{aligned} cd + p &= 2c'\frac{d'}{2} + p' && (5)\\ &= c'd' + p' && (6)\\ &= ab && (7) \end{aligned}$$

In either case, the loop invariant holds after the iteration.

By the loop invariant, we know that $cd + p = ab$ after all iterations of the loop. By the loop condition, we have $d = 0$, hence $p = ab$. The function returns p, which is ab, the product of the two parameters. Thus the code is correct.

7. The following function computes x^b for a real number x and a positive integer b. What is its loop invariant?

```
float power(float x, int b)
 {
  float y = x;
  int d = b;
  float z = 1.0;
  while(d > 0)
   {
    if(d % 2) z = z*y;
    y = y*y;
    d = d/2;
   }
  return z;
 }
```

The code for power is analogous to the code for product. The loop invariant is also the anolog, namely $y^d z = x^b$.

The loop invariant trivially holds before the first iteration, since $d = b$, $y = x$, and $z = 1$. Suppose the loop invariant holds before one iteration of the loop; we need to prove that it holds after that iteration. Let y', z', d' be the values of y, z, d before the iteration, and y, z, d the values after the iteration. Thus $(y')^{d'} z' = x^b$.

There are two cases – d' odd and d' even. If d' is odd, then $z = z'y'$ and $d = \frac{d'-1}{2}$, while if d' is even, $z = z'$ and $d = \frac{d'}{2}$. In either case, $y = (y')^2$.

Suppose d' is odd. Then

$$\begin{aligned} y^d z &= ((y')^2)^{\frac{d'-1}{2}} y' z' \\ &= (y')^{d'-1} y' z' \\ &= (y')^{d'} z' \\ &= x^b \end{aligned}$$

If d' is even, then

$$\begin{aligned} y^d z &= ((y')^2)^{\frac{d'}{2}} z' \\ &= (y')^{d'} z' \\ &= x^b \end{aligned}$$

In either case, the loop invariant is verified.

To prove correctness of the procedure, we use the fact that when the loop is finished, the loop invariant still holds, and $d = 0$. Thus $x^b = y^d z = z$. The value $z = x^b$ is returned, hence the code is correct.