1 Asymptotic Relationships [15 marks]

Circle the statements that are correct.

1 per box

[✓ $n^2 \in O(n^4)$]	[✓ $n^3 + 100n \in \Theta(n^3)$]	[✓ $\sqrt{n} \in \Omega(\lg n)$]
$n \log n \in \Omega(n^{1.01})$	$\log n \in \Theta\left(\frac{\log_3(n)}{\lg n}\right)$	[✓ $n^{100} \in O(3^n)$]
$n \log^3 n \in \Omega(n^2 \log n)$	[✓ $3n^5 + 17 \in \Theta(17n^5 + 3)$]	$2^n \in \Theta(2^{2n})$
$\log(n!) \in O(n)$	[✓ $n^3 \in \Omega(10n^3 + 100n^2 + 1000n)$]	$\log \log n \in \Theta(\log^2 n)$
[✓ $\sum_{i=1}^{n}(2i+1) \in \Theta(n^2)$]	[✓ $1,000,000 \in O(n)$]	[✓ $2^{2^n} \in \Omega(3^n)$]

Remarks:

- $\log^3 n = (\log n)^3$
- $\log_b a = \dfrac{\log_c a}{\log_c b}$
- You can assume that $\log n = \log_{10} n$ (although it does not really matter).

2 Big-O Proof [10 marks]

For this problem, you must **formally prove** that

$$\text{if } f(n) \in O(h(n)) \text{ and } g(n) \in O(h(n)) \text{ then } f(n) + g(n) \in O(h(n)).$$

Since this is a formal proof, it will be based on the definition of big-O notation (that uses c and n_0).

Hint:

(1) Write down what $f(n) \in O(h(n))$ means according to the definition.

(2) Write down what $g(n) \in O(h(n))$ means according to the definition (you should call constants c and n_0 something slightly different, as it's very unlikely, they are same as in (1)).

(3) Do some algebra to show that $f(n) + g(n) \in O(h(n))$.

Since $f(n) \in O(h(n))$, $f(n) \leq c\,h(n)$ for all $n \geq n_0$ for some $c > 0$ and $n_0 > 0$

Since $g(n) \in O(h(n))$, $g(n) \leq d \cdot h(n)$ for all $n \geq m_0$ for some $d > 0$ and $m_0 > 0$

Therefore $f(n) + g(n) \leq c\,h(n) + d\,h(n)$ for all $n \geq \max\{n_0, m_0\}$

$\Rightarrow f(n) + g(n) \leq \underline{(c+d)}\,h(n)$ for all $n \geq \underline{\max\{n_0, m_0\}}$

+2 for O definition

+2 for using different constants for $f(n)$ and $g(n)$

+4 for correct manipulation of constants

+2 for correct logic

3 Nth Power [10 marks]

The following code computes the n-th power of x using repeated squaring.

```
// Calculcate x^n (n is a non-negative integer)
double Pow(double x,int n) {
  double result = 1.0;
  double p = x;
  int i = n;
  while( i > 0 ) {
    // Invariant: result * p^i = x^n
    if( i%2 == 1 ) result *= p;
    p *= p;
    i = i/2;
  }
  return result;
}
```

1. Describe the worst-case time complexity of the function Pow using Θ-notation. You do not need to prove that your answer is correct.

2

$\Theta(\log n)$

2. Use induction to prove the loop invariant (stated in the code). For the inductive step, consider two cases depending on whether i is even or odd.

6

by induction on number of iterations, w, of the while-loop.

Base ($w=0$)

$\overline{\text{result}} = 1.0$, $p = x$, $i = n$, so $\overline{\text{result}} \cdot p^i = 1.0\, x^n = x^n$ ✓

Ind. Hyp Assume the invariant holds after $w \geq 0$ iterations

We need to show it holds after $w+1$ iterations.

Let $\overline{\text{result}}$, $\overline{p}$, and $\overline{i}$ be the values of
result, p, and i after the $(w+1)^{th}$ iteration.

if i is even $\overline{\text{result}} = \text{result}$, $\overline{p} = p^2$, $\overline{i} = i/2$ so

$\overline{\text{result}} \cdot \overline{p}^{\overline{i}} = \text{result} \cdot (p^2)^{i/2} = \text{result} \cdot p^i = x^n$ (by I.H.)

if i is odd $\overline{\text{result}} = \text{result} \cdot p$, $\overline{p} = p^2$, $\overline{i} = \frac{i-1}{2}$ so

$\overline{\text{result}} \cdot \overline{p}^{\overline{i}} = \text{result} \cdot p \cdot (p^2)^{i-1/2} = \text{result} \cdot p \cdot p^{i-1} = \text{result} \cdot p^i = x^n$

by I.H.

3. Show that if the loop invariant is correct, then the function Pow computes x^n.

2

After the last iteration, $i=0$, so $result \cdot p^i = result \cdot p^0 = result$, which by the loop invariant is x^n.

4 k-Way Mergesort [10 marks]

The **Mergesort3** algorithm is like **Mergesort** except that it splits the input array into three (approximately equal sized) parts, rather than just two. It recursively sorts the three parts and then merges the three parts together into one sorted array. For questions 1, 2, and 3, assume that n is a power of 3 and that **Merge3**(B,C,D) takes $|B| + |C| + |D|$ steps (where $|X|$ means the size of array X).

```
Mergesort3($A[1 \dots n]$)
  if ($n \leq 1$) return $A$
  $B$ = Mergesort3($A[1 \dots n/3]$)
  $C$ = Mergesort3($A[n/3 + 1 \dots (2n)/3]$)
  $D$ = Mergesort3($A[(2n)/3 + 1 \dots n]$)
  return Merge3($B$,$C$,$D$)
```

1. Let $M(n)$ be the number of steps taken by **Mergesort3** on an input of size n. What is a recurrence equation for $M(n)$? (Don't forget the base cases.)

3

$$M(n) = 3M(n/3) + n$$
$$M(1) = 1$$

2. What is $M(n)$ as a function of n? Your solution should not be a recurrence or contain a summation.

3

Claim $M(n) = n(\log_3(n) + 1)$.

Proof (not needed but here it is) (by induction on n)

base $M(1) = 1 = n(\log_3(n) + 1) = 1$ ✓

Suppose $M(n) = n(\log_3 n + 1)$

$$M(3n) = 3M(n) + 3n = 3n(\log_3 n + 1) + 3n$$
$$= 3n\log_3(3n) + 3n$$
$$= 3n(\log_3(3n) + 1)$$ ☒

3. Find the simplest function $g(n)$ such that $M(n) \in \Theta(g(n))$. (You do not need to prove anything.) Is this asymptotic running time (i.e., $\Theta(g(n))$) different from that of regular 2-way **Mergesort** (yes or [✓ no])?

1

$\Theta(n \log n)$ NO

4. We can imagine splitting the input array into k (approximately equal-sized) parts, for any integer k, and recursively sorting those k parts and merging them together into one sorted array. What is the asymptotic running time of this k-way **Mergesort** assuming that k is a constant (not depending on n)? (You do not need to prove anything.) You may assume that n is a power of k.

1

$\Theta(n \log n)$

5. What is the asymptotic running time of n-way **Mergesort**?

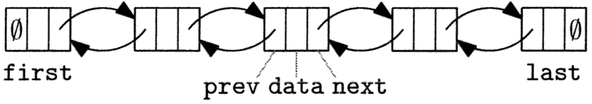
Be careful. You should describe how to perform the n-way **Merge** step of the algorithm. How much time does this step take?

2

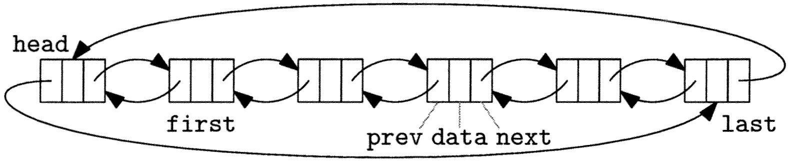
The merge step must combine n sorted lists of size 1 into one sorted list of size n.
We could do this in $\Theta(n^2)$ time by repeatedly finding the min of all size-one lists or we could use a priority queue.
If we implement the priority queue as a heap, we can heapify in $\Theta(n)$ time and deleteMin n times in $\Theta(n \log n)$ time, for a total running time of $\Theta(n \log n)$.

5 Circular Doubly-Linked Lists [6 marks]

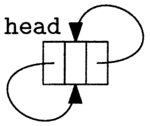
A non-empty regular doubly-linked list has a *first* and *last* node (which might be the same node), where the prev pointer of the first node is NULL and next pointer of the last node is NULL.



A **circular** doubly-linked list is a doubly-linked list that contains one special node called the *head* of the list. The next pointer of the head points to the first node in the doubly-linked list and the prev pointer of the head points to the last node in the doubly-linked list. In addtion, both the prev pointer of the first node and the next pointer of the last node point to the head. The head node acts as a dummy node that "closes the loop" to make the doubly-linked list circular. An **empty** circular doubly-linked list is just a head node that points to itself with both next and prev pointers.



A circular doubly-linked list.



An empty circular doubly-linked list.

```
struct Node { int data; Node * prev; Node * next; };
```

Your job is to write code to join two circular doubly-linked lists to create one circular doubly-linked list. The joined list contains all of the nodes from the first list (accessed via headA) followed by all of the nodes from the second list (accessed via headB) except **it contains only the first head** (*headA), which becomes the head of the joined list. You must not create any new nodes. Your code must run in constant time and space. (1) Don't forget to delete headB. (2) Either list might be empty, but neither headA nor headB are NULL.

```
void join_lists(Node * headA, Node * headB) {
\\ headA and headB are pointers to the head nodes of two
\\ circular doubly-linked lists.
\\ Join the two lists so that *headA is the head of the merged list.

\\ YOUR CODE GOES HERE
+1   if ( headB->next != headB) {
              headA->prev->next = headB->next;
              headB->next->prev = headA->prev;
              headB->prev->next = headA;
              headA->prev = headB->prev;
     }
 +1  delete headB;
}
```

+4 correct pointer manipulation

6 nevePrint [6 marks]

Write a **recursive** C++ function, `nevePrint`, that given a pointer to the first entry of a singly-linked list prints the data in even positions of the list in reverse order. For example, `nevePrint` should print `8 6 4 2` when given `first` → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → ∅.

Keep your code short, simple, and **recursive**.

```
struct Node { int data; Node *next; };
void nevePrint( Node *first ) {
    if ( first == NULL || first->next == NULL ) return;
    nevePrint( first->next->next );
    cout << first->next->data << " ";
}
```

+2 base cases

+1

+3 recursive
correct order of call and cout

7 Short Answers [8 marks]

1. Consider the following incomplete statement:

2

> Assume algorithm foo and algorithm bar solve the same problem (so they take the same set of inputs). If foo has worst-case running time $\boxed{A}(n^2)$ and bar has worst-case running time $\boxed{B}(n^3)$ on inputs of size n, then foo is faster than bar on $\boxed{C}$ inputs of size n for sufficiently large n.

By filling in boxes $\boxed{A}$, $\boxed{B}$, and $\boxed{C}$, we can complete the statement and make it True or False. Assume $\boxed{A}$ and $\boxed{B}$ must be: $\boxed{O}$ or $\boxed{\Omega}$. Assume $\boxed{C}$ is $\boxed{\text{all}}$ or $\boxed{\text{some}}$. List all ways of filling in the boxes that result in the statement being True. Each row that you fill in should specify the three choices that make the statement true.

$\boxed{A}$	$\boxed{B}$	$\boxed{C}$
O	Ω	Some

2. Assume that the ADT Queue is implemented using a circular array with size 6. The current state of this data structure is as follows:

2

0	1	2	3	4	5
b	c	d		x	

`front = 4` and `back = 5`

Note that this means that the queue contains one element "x".

Draw the final state of the data structure after the following sequence of operations:

```
enqueue(a); enqueue(b); enqueue(c); deque(); deque(); enqueue(d);
```

Final state:

0	1	2	3	4	5
b	c	d			

`front = 0` and `back = 3`

2

3. Recall the first version of `fib` function from the lecture:

```
   int fib(int n) {
1.    if (n <= 2) return 1;
2.    int a = fib(n-1);
3.    int b = fib(n-2);
4.    return a+b;
   }
```

At one point in the execution of `fib(5)`, the call stack looks like this:

	Line 1, n = 2, return 1
Fill in the missing line! ⟶	Line 2, n=3, a=fib(2)
	Line 3, n = 5, a = 3, b = fib(3)

2

4. Consider a d-heap with n elements. Circle the correct answers below:

- The height of this d-heap is approximately:

  A. $\lg n$ B. $\lg d$ [✓ C. $\log_d n$] D. $d \lg n$

- Assume the height of this d-heap is H. The `swapDown` function will take at most:

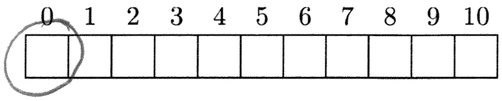
  A. H B. $H \lg d$ [✓ C. dH] D. $d + H$ steps.

8 Two-dimensional Heaps [10 marks]

keys distinct

2

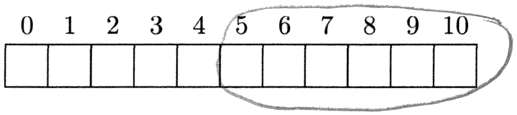
1. The following figure represents an array that stores a binary Min-Heap with 11 elements. I haven't shown the keys for these elements.



Circle the locations where the element with the **minimum** key in the heap might lie.

2

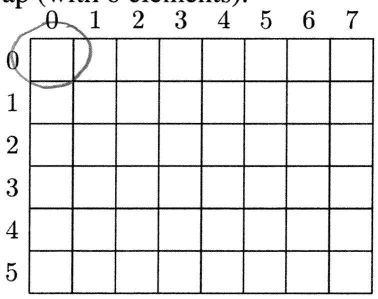
2. The following figure represents an array that stores a binary Min-Heap with 11 elements. I haven't shown the keys for these elements.



Circle the locations where the element with the **maximum** key in the heap might lie.

2

3. The following figure represents a two-dimensional array in which every row and every column is a binary Min-Heap. Notice that each element in the two-dimensional array is a member of two heaps: its column heap (with 6 elements) and its row heap (with 8 elements).

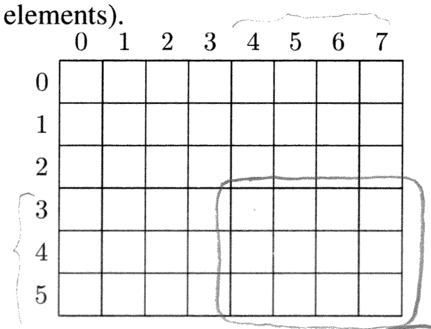


the 2D array

Circle the locations where the element with the **minimum** key in **all** the heaps might lie.

2

4. The figure represents a two-dimensional array in which every row and every column is a binary Min-Heap. Notice that each element in the two-dimensional array is a member of two heaps: its column heap (with 6 elements) and its row heap (with 8 elements).



the 2D array

Circle the locations where the element with the **maximum** key in **all** the heaps might lie.

2

1	2
3	4

5. Is the only two-dimensional array with keys 1, 2, 3, and 4 in which every row and every column is a binary Min-Heap? Yes or No. No.

1	3
2	4

This page intentionally left blank.

If you put solutions here or anywhere other than the blank provided for each solution, you must *clearly* indicate which problem the solution goes with and also indicate where the solution is at the designated area for that problem's solution.