Chapter 1

1. (a)

$$\begin{array}{ccccccc}
1 & \underline{7} & \underline{2} & 6 & 3 & 5 & 4 \\
1 & 2 & \underline{7} & \underline{6} & 3 & 5 & 4 \\
1 & 2 & 6 & \underline{7} & \underline{3} & 5 & 4 \\
1 & 2 & \underline{6} & \underline{3} & 7 & 5 & 4 \\
1 & 2 & 3 & 6 & \underline{7} & \underline{5} & 4 \\
1 & 2 & 3 & \underline{6} & \underline{5} & 7 & 4 \\
1 & 2 & 3 & 5 & 6 & \underline{7} & \underline{4} \\
1 & 2 & 3 & 5 & \underline{6} & \underline{4} & 7 \\
1 & 2 & 3 & \underline{5} & \underline{4} & 6 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}$$

(b)

$$\begin{array}{cccccccc}
1 & 2 & \underline{8} & \underline{7} & 3 & 4 & 6 & 5 \\
1 & 2 & 7 & \underline{8} & \underline{3} & 4 & 6 & 5 \\
1 & 2 & \underline{7} & \underline{3} & 8 & 4 & 6 & 5 \\
1 & 2 & 3 & 7 & \underline{8} & \underline{4} & 6 & 5 \\
1 & 2 & 3 & \underline{7} & \underline{4} & 8 & 6 & 5 \\
1 & 2 & 3 & 4 & 7 & \underline{8} & \underline{6} & 5 \\
1 & 2 & 3 & 4 & \underline{7} & \underline{6} & 8 & 5 \\
1 & 2 & 3 & 4 & 6 & 7 & \underline{8} & \underline{5} \\
1 & 2 & 3 & 4 & 6 & \underline{7} & \underline{5} & 8 \\
1 & 2 & 3 & 4 & \underline{6} & \underline{5} & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{array}$$

3. (a)

$$\begin{array}{ccccccc}
1 & \underline{7} & \underline{2} & 6 & 3 & 5 & 4 \\
1 & 2 & \underline{7} & 6 & \underline{3} & 5 & 4 \\
1 & 2 & 3 & \underline{6} & 7 & 5 & \underline{4} \\
1 & 2 & 3 & 4 & \underline{7} & \underline{5} & 6 \\
1 & 2 & 3 & 4 & 5 & \underline{7} & \underline{6} \\
1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}$$

(b)

$$\begin{array}{cccccccc}
1 & 2 & \underline{8} & 7 & \underline{3} & 4 & 6 & 5 \\
1 & 2 & 3 & \underline{7} & 8 & \underline{4} & 6 & 5 \\
1 & 2 & 3 & 4 & \underline{8} & 7 & 6 & \underline{5} \\
1 & 2 & 3 & 4 & 5 & \underline{7} & \underline{6} & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{array}$$

5. Take $N = 1$ and $c = 13$. Then

$$f(n) = 4n^2 + 9n^3 \leq 4n^3 + 9n^3 = 13n^3 = cn^3$$

for all $n \geq N$. Therefore, by the definition of O, we have $f(n) = O(n^3)$.

Now take $N = 1$ and $c = 1$. Then

$$f(n) = 4n^2 + 9n^3 \geq n^3 = cn^3$$

for all $n \geq N$. Thus, by the definition of Ω, we have $f(n) = \Omega(n^3)$.

By the definition of Θ, since we have $f(n) = O(n^3)$ and $f(n) = \Omega(n^3)$, then $f(n) = \Theta(n^3)$.

7. We have

$$\lim_{n\to\infty} \frac{2017n^4 + 4n^{2017}}{n^{2017}} = \lim_{n\to\infty} \left(\frac{2017n^4}{n^{2017}} + \frac{4n^{2017}}{n^{2017}}\right) = \lim_{n\to\infty} \left(\frac{2017}{n^{2013}} + 4\right) = 4.$$

Therefore, by the limit criterion, we have $2017n^4 + 4n^{2017} = \Theta(n^{2017})$.

9. The following algorithm finds the largest number in a list of n numbers by scanning the list and keeping track of the largest number found so far.

Input: a non-empty list $a_0, a_1, \ldots, a_{n-1}$ of numbers

```
x ← a_0
for i = 1 to n − 1 do
  if a_i > x then
    x ← a_i
  end if
end for
return x
```

Independently of the input, the for-loop runs $n-1$ times, each time performing exactly one comparison. Therefore this algorithm takes $\Theta(n)$ time in the worst case.

11. Here is the trick. To simplify the discussion, suppose that n is even. Let $A[1..n]$ be the array. Split the array into $\frac{n}{2}$ pairs

$$A[1], A[2] \quad | \quad A[3], A[4] \quad | \quad \cdots \quad | \quad A[n-1], A[n].$$

find the minimum and the maximum of each pair by doing one comparison within each pair. Therefore, we have made $\frac{n}{2}$ comparisons so far.

You end up with $\frac{n}{2}$ *small numbers* (one for each pair). The minimum of these $\frac{n}{2}$ small numbers is the minimum you are looking for. Since you have $\frac{n}{2}$ numbers, you can find this minimum with $\frac{n}{2}$ comparisons. Hence, in total, we have made $\frac{n}{2} + \frac{n}{2} = n$ comparisons so far.

You do the same thing for the maximum. That is, you find the maximum of the $\frac{n}{2}$ *large numbers* and you are done. This adds an extra $\frac{n}{2}$ comparisons for a total of $\frac{3}{2}n$ comparisons.

Input: An array $A[1..n]$ of n numbers.
Initialize an array $A_{small}[1..n/2]$
Initialize an array $A_{big}[1..n/2]$
for $i = 1$ **to** $n/2$ **do**
  if $A[2i-1] < A[2i]$ **then**
    $A_{small}[i] \leftarrow A[2i-1]$
    $A_{big}[i] \leftarrow A[2i]$
  else
    $A_{small}[i] \leftarrow A[2i]$
    $A_{big}[i] \leftarrow A[2i-1]$
  end if
end for
$x_{small} \leftarrow A_{small}[1]$
for $i = 2$ **to** $n/2$ **do**
  if $A_{small}[i] < x_{small}$ **then**
    $x_{small} \leftarrow A_{small}[i]$
  end if
end for
$x_{big} \leftarrow A_{big}[1]$
for $i = 2$ **to** $n/2$ **do**
  if $A_{big}[i] > x_{big}$ **then**
    $x_{big} \leftarrow A_{big}[i]$
  end if
end for
return x_{small}, x_{big}

13. (a) **Input:** An array $A[1..n]$ of n distinct integers in the interval $[1, 2019n]$.
Initalize an array $B[1..(2019n)]$ with 0's everywhere.
for $i = 1$ **to** n **do**
  $B[A[i]] \leftarrow 1$
end for
$i \leftarrow 1$
for $j = 1$ **to** $2019n$ **do**
  if $B[j] = 1$ **then**
    $A[i] \leftarrow j$
    $i \leftarrow i + 1$
  end if
end for

This is called *bucket sort*.

It takes $\Theta(2019n) = \Theta(n)$ time to initialize B. Then we scan A once (and modify B). This takes $\Theta(n)$ time. Then we scan B once (and fill A) this takes $\Theta(2019n) = \Theta(n)$ time. In total, it takes $\Theta(n)$ time.

(b) **Input:** An array $A[1..n]$ of n distinct integers in the interval $[1, kn]$.

```
Initalize an array B[1..(kn)] with 0's everywhere.
for i = 1 to n do
  B[A[i]] ← 1
end for
i ← 1
for j = 1 to kn do
  if B[j] = 1 then
    A[i] ← j
    i ← i + 1
  end if
end for
```

The number k is a **fixed** integer. Whatever is the input, k stays the same. Therefore, it takes $\Theta(kn) = \Theta(n)$ time to initialize B. Then we scan A once (and modify B). This takes $\Theta(n)$ time. Then we scan B once (and fill A) this takes $\Theta(kn) = \Theta(n)$ time. In total, it takes $\Theta(n)$ time.

(c) The same technique works with a tiny modification. Do you see how?

(d) If we do not have any upper on the numbers to be sorted, we do not know what should be the size of B. So this technique does not work. Is there another technique we could use to sort A in $O(n)$ time? By the end of the semester, we will see that the answer is no.

15.

$$
\begin{aligned}
T(n) &= \sum_{j=1}^{n/2} \sum_{i=1}^{j} 1 \\
&= \sum_{j=1}^{n/2} j \\
&= \frac{n/2(n/2+1)}{2} \\
&= \frac{n^2}{8} + \frac{n}{4} \\
&= \Theta(n^2)
\end{aligned}
$$

17. How many times do we execute the inner loop? How many times can we do "$\lfloor j/2 \rfloor$" starting from n?

$$1 + \log_2(n)$$

So we get

$$\begin{aligned} T(n) &= \sum_{i=1}^{n}(1+\log_2(n)) \\ &= n(1+\log_2(n)) \\ &= n + n\log_2(n) \\ &= \Theta(n\log_2(n)) \end{aligned}$$

19. We proceed by induction.

    The definition of F_n is recursive and there are *two* base cases in that definition. Therefore, there are *two* base cases to consider in our proof by induction: we have $F_6 = 8 \geq 8 = 2^{6/2}$ and $F_7 = 13 \geq 8\sqrt{2} = 2^{7/2}$.

    For the inductive step, let any natural number $n \geq 8$ be given and assume that $F_{n-2} \geq 2^{(n-2)/2}$ and $F_{n-1} \geq 2^{(n-1)/2}$. We then have

    $$F_n = F_{n-2} + F_{n-1} \geq 2^{(n-2)/2} + 2^{(n-1)/2} \geq 2 \times 2^{(n-2)/2} = 2^{(n-2)/2+1} = 2^{n/2},$$

    which completes the proof.

    The inequality is false for $n < 6$ since

    $$\begin{aligned} F_0 &= 0 < 1 = 2^{0/2}, \\ F_1 &= 1 < \sqrt{2} = 2^{1/2}, \\ F_2 &= 1 < 2 = 2^{2/2}, \\ F_3 &= 2 < 2\sqrt{2} = 2^{3/2}, \\ F_4 &= 3 < 4 = 2^{4/2}, \\ F_5 &= 5 < 4\sqrt{2} = 2^{5/2}. \end{aligned}$$

21. We proceed by induction. For the base cases, we have $0 \leq 0 = 2F_0$ and $1 \leq 2 = 2F_0$.

    For the inductive step, let any natural number $n \geq 2$ be given and assume that $n-2 \leq 2F_{n-2}$ and $n-1 \leq 2F_{n-1}$. We then have

    $$2F_n = 2(F_{n-2} + F_{n-1}) = 2F_{n-2} + 2F_{n-1} \geq (n-2) + (n-1) = 2n - 3 \geq n.$$

23. Let $T(n)$ be the number of bit-operations performed by $fib(n)$. We will show that there is a constant $c \geq 1$ such that $T(n) \leq 2c\,nF_n$ by induction.

    For the base cases, since $fib(0)$ and $fib(1)$ only return a number, they do not execute any bit-operation. So $T(0) = 0 \leq 0 = 2c \cdot 0 \cdot F_0$ and $T(1) = 0 \leq 2c = 2c \cdot 1 \cdot F_1$ for any constant $c \geq 1$.

For the inductive step, let any natural number $n \geq 2$ be given and assume that $T(n-2) \leq 2c\,(n-2)F_{n-2}$ and $T(n-1) \leq 2c\,(n-1)F_{n-1}$. Note that $F_n < 2^n$ (refer to Exercise #22) implies that F_n has at most $O(\log_2(F_n)) = O(\log_2(2^n)) = O(n)$ bits. Hence the sum in the fib algorithm makes $O(n)$ bit-operations.

Therefore, from the algorithm, there is a constant $c \geq 1$ such that

$$\begin{aligned}
T(n) &\leq T(n-2) + T(n-1) + c\,n \\
&\leq 2c\,(n-2)F_{n-2} + 2c\,(n-1)F_{n-1} + c\,n \\
&= 2c\,n(F_{n-2} + F_{n-1}) - 4c\,F_{n-2} - 2c\,F_{n-1} + c\,n \\
&\leq 2c\,nF_n - 2c\,F_{n-2} - 2c\,F_{n-1} + c\,n \\
&= 2c\,nF_n - 2c\,(F_{n-2} + F_{n-1}) + c\,n \\
&= 2c\,nF_n - 2c\,F_n + c\,n \\
&= 2c\,nF_n - c\,(2F_n - n), \\
&\leq 2c\,nF_n && \text{by Exercise \#21.}
\end{aligned}$$

25. We prove that $2^n \neq \Omega(3^n)$, which proves that $2^n \neq \Theta(3^n)$. The proof is by contradiction using the definition of Ω. Suppose there are positive constants N and c such that

$$2^n \geq c\,3^n$$

for all $n \geq N$. Then we get

$$(3/2)^n \leq 1/c \tag{1}$$

for all $n \geq N$. As n increases, $(3/2)^n$ gets arbitrarily large. Therefore, since c is a constant, (1) cannot be true for all $n \geq N$. This is a contradiction.