MATH182 HOMEWORK #2
DUE July 5, 2020

Exercise 1. *Consider the following basic problem. You're given an array A consisting of n integers $A[1], A[2], \ldots, A[n]$. You'd like to output a two-dimensional $n \times n$ array B in which $B[i,j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$, that is, the sum $A[i] + A[i+1] + \cdots + A[j]$, (The value of array entry $B[i,j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what is output for these values.)*

Here is a simple algorithm to solve this problem.

```
1  for $i = 1$ to $n$
2      for $j = i + 1$ to $n$
3          Add up array entries $A[i]$ through $A[j]$
4          Store the result in $B[i,j]$
```

(1) *For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).*
(2) *For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)*
(3) *Although the algorithm you analyzed in parts (a) and (b) is the most natural way to solve the problem – after all, it just iterates through the relevant entries of the array B, filling in a value for each – it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $O(g(n))$, where $\lim_{n\to\infty} g(n)/f(n) = 0$.*

Solution.
(1) We do a complete analysis of the algorithm to directly obtain a tight bound on its run-time. Let $C(n,k) :=$ total cost of line k for input n. Assuming the cost of line 1 is c_1 per run, its total cost is given by

$$C(n,1) = c_1 \cdot \sum_{i=1}^{n+1} 1 = c_1 \cdot (n+1) = \Theta(n)$$

Similarly, for line 2, assuming c_2 to be its cost per run, its total cost is given by

$$C(n,2) = c_2 \cdot \sum_{i=1}^{n+1} \sum_{j=i+1}^{n+1} 1 = c_2 \cdot \sum_{i=1}^{n+1} (n-i) = c_2 \cdot \frac{(n+1)(n-2)}{2} = \Theta(n^2)$$

The analysis of line 3 is slightly more complex. We assume the process of adding up the array entries $A[i]$ through $A[j]$ takes linear time; in particular, each run costs $c_3 \cdot (j-i)$ for some $c_3 > 0$.

We therefore have, for $k := j - i$,

$$\begin{aligned}
C(n,3) &= c_3 \cdot \sum_{i=1}^{n+1} \sum_{j=i+1}^{n+1} (j-i) = c_3 \cdot \sum_{i=1}^{n+1} \sum_{k=1}^{n-i+1} k \\
&= c_3 \cdot \sum_{i=1}^{n+1} \frac{(n-i+1)(n-i+2)}{2} = \frac{c_3}{2} \cdot \sum_{i=1}^{n+1} (n^2 - 2ni + 3n + i^2 - 3i + 2) \\
&= \frac{c_3}{2} \left((n^2+3n+2) \sum_{i=1}^{n+1} 1 - (2n+3) \sum_{i=1}^{n+1} i + \sum_{i=1}^{n+1} i^2 \right) \\
&= \frac{c_3}{2} \left((n^2+3n+2)(n+1) - (2n+3)\frac{(n+1)(n+2)}{2} + \frac{n(n+1)(2n+1)}{6} \right) \\
&= \frac{c_3}{24} \big(12(n^2+3n+2)(n+1) - 6(2n+3)(n+1)(n+2) + 2n(n+1)(2n+1) \big)
\end{aligned}$$

We are only interested in the power of the leading term of this expression. Calculating more precisely only the coefficients of n^3, we see that

$$\begin{aligned}
C(n,3) &= \frac{c_3}{24}(12n^3 - 12n^3 + 4n^3 + k_2 n^2 + k_1 n + k_0) \\
&= \frac{c_3}{24}(4n^3 + k_2 n^2 + k_1 n + k_0) = \Theta(n^3) \qquad \text{(for some } k_0, k_1, k_2 \in \mathbb{Z})
\end{aligned}$$

The analysis for line 4 is similar to that of line 2, and we have $C(n,4) = \Theta(n^2)$. We therefore have the total running time, $T(n)$, for the algorithm given by

$$\begin{aligned}
T(n) &= C(n,1) + C(n,2) + C(n,3) + C(n,4) \\
&= \Theta(n) + \Theta(n^2) + \Theta(n^3) + \Theta(n^2) = \Theta(n^3) \implies T(n) = \Theta(n^3)
\end{aligned}$$

We therefore have $f(n) = n^3$ such that the running time of this algorithm has an upper bound of the form $O(f(n))$.

(2) Since the running time of the algorithm, per the analysis in *(2)*, is $\Theta(n^3)$, the running time of the algorithm for size n is also $\Omega(n^3)$. ■

(3) Any algorithm to generate such a matrix can only, in theory, be $\Theta(n^2)$ at best — even if we were setting all $B[i,j]$ for $i < j$ to the same integer (say, 0), that process alone would take $\Theta(n^2)$. We therefore focus on making the adding part of the algorithm (line 3) better.

We notice that at each iteration of the inner loop, instead of computing the sum of $A[i]$ through $A[j]$ from scratch, we could use the sum of $A[i]$ through $A[j-1]$ (which has already been computed, and is stored in $B[i][j-1]$) and add $A[j]$ to it to get the same result.[1]

```
1  for $i = 1$ to $n$
2          Store $A[i]$ in $B[i][i]$   // for algorithm correctness in case $j = i + 1$
3          for $j = i + 1$ to $n$
4                  Add $B[i][j-1]$ and $A[j]$
5                  Store the result in $B[i][j]$
```

Since the body of the inner loop now runs in constant time rather than linear time, our time complexity improves from $\Theta(n^3)$ to $\Theta(n^2)$. This time is is asymptotically better, since for $g(n) := n^2$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0$$

[1]See Appendix I for C++ code

Exercise 2. *Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.*

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2       for j = A.length downto i + 1
3            if A[j] < A[j − 1]
4                 exchange A[j] with A[j − 1]
```

(1) Let A' denote the output of BUBBLESORT(A). *To prove that* BUBBLESORT *is correct, we need to prove that it terminates and that*

(†) $$A'[1] \le A'[2] \le \cdots \le A'[n],$$

where $n = A.length$. In order to show that BUBBLESORT *actually sorts, what else do we need to prove?*

The next two parts will prove inequality (†).

(2) State precisely a loop invariant for the **for** *loop in lines 2-4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.*

(3) Using the termination condition of the loop invariant proved in part (2), state a loop invariant for the **for** *loop in lines 1-4 that will allow you to prove inequality (†). Your proof should use the structure of the loop invariant proof presented in this chapter.*

(4) What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

Solution.

(1) We additionally need to show that the elements in A' are the same as the elements in A, i.e., the element-sets (with duplicate entries) of A and A' are identical. In particular, if an element k appears n_k times in A, it must appear exactly n_k times in A'. It is, however, trivial to show that the array contains the same elements each iteration — since the only line where the array is modified, line 4, merely swaps two elements, the element-set (with duplicate entries) of the array does not change.

In parts (2) and (3), we assume this additional condition has already been shown.

(2) The loop invariant for the inner **for** loop is as follows:

Loop Invariant: *After line 2 is run, $A[j]$ is the smallest element in the subarray $A[j, \ldots, A.length]$.*

We show this loop invariant holds.

Initialisation: Before the first iteration, we have $j = A.length$. Since the subarray $A[j, \ldots, A.length] = A[A.length]$ is a one-element array, it is (trivially) sorted.

Maintenance: Assume the loop invariant is true before some iteration with $j = j_0 \in [i + 1, A.length]$ for some $i \in [1, A.length]$, i.e., $A[j_0]$ is the smallest element in the subarray $A[j_0, \ldots, A.length]$. In line 3, we compare $A[j_0]$ with $A[j_0 - 1]$. We then have two cases:

Case 1: $A[j_0] < A[j_0 - 1]$. In this case, the **if** condition evaluates true and $A[j_0]$ and $A[j_0 - 1]$ are swapped. Since, by the loop invariant, we know the element now at index $j_0 - 1$ is smaller than every element in $A[j_0 + 1, \ldots, A.length]$, and that additionally, by assumption, $A[j_0 - 1] < A[j_0]$ (after the elements are swapped, we know $A[j_0 - 1]$ is the smallest element in the subarray $A[j_0 - 1, \ldots, A.length]$.

Case 2: $A[j_0 - 1] <= A[j_0]$. In this case, the **if** condition evaluates false and the array is not changed. Since, by the loop invariant, $A[j_0]$ is the smallest element in the subarray $A[j_0, \ldots, A.length]$, and additionally, by assumption, $A[j_0 - 1] <= A[j_0]$, know $A[j_0 - 1]$ is the smallest element in the subarray $A[j_0 - 1, \dots, A.length]$.

The loop invariant is therefore true for $j = j_0 - 1$.

Termination: In the final iteration, we have $j = i < i + 1$ and the loop terminates. From the loop invariant, we know $A[i]$ is the smallest element in the subarray $A[i, \dots, A.length]$.

(3) The loop invariant for the outer **for** loop is as follows:

Loop Invariant: *After line 1 is run, the subarray $A[1, \dots, i]$ is sorted.*

We show this loop invariant holds.

Initialisation: Before the first iteration, we have $i = 1$. Since the subarray $A[1, \dots, i] = A[1]$ is a one-element array, it is (trivially) sorted.

Maintenance: Assume the loop invariant is true before some iteration with $i = i_0 \in [1, A.length - 1]$, i.e., the subarray $A[1, \dots, i_0]$ is sorted. From the loop invariant of the inner loop, after the last iteration of the inner loop, we know $A[i_0]$ is the smallest element in the subarray $A[i_0, \dots, A.length]$. Therefore, since the subarray $A[1, \dots, i_0]$ is sorted (by assumption) and $A[i_0] <= A[i_0 + 1]$, the subarray $A[1, \dots, i_0 + 1]$ is sorted.

The loop invariant is therefore true for $i = i_0 + 1$.

Termination: In the final iteration, we have $i = A.length]$ and the loop terminates. From the loop invariant, we know $A[1, \dots, i] = A[1, \dots, A.length] = A$ is sorted.

$\therefore$ BubbleSort is correct. ■

(3) We do a complete analysis of the algorithm to directly obtain a tight bound on its run-time. Let $C(n, k) :=$ total cost of line k for input n. Assuming the cost of line 1 is c_1 per run, its total cost is given by

$$C(n,1) = c_1 \cdot \sum_{i=1}^{n+1} 1 = c_1 \cdot (n+1) = \Theta(n)$$

Similarly, for line 2, assuming c_2 to be its cost per run, its total cost is given by

$$C(n,2) = c_2 \cdot \sum_{i=1}^{n+1} \sum_{j=i+1}^{n+1} 1 = c_2 \cdot \sum_{i=1}^{n+1} (n - i) = c_2 \cdot \frac{(n+1)(n-2)}{2} = \Theta(n^2)$$

Line 3 runs once for each time line 2 runs, and therefore also has time complexity $\Theta(n^2)$. Line 4 runs at most once for each time Line 2 runs, and therefore has time complexity $O(n^2)$.

We therefore have the total running time, $T(n)$, for the algorithm given by

$$\begin{aligned} T(n) &= C(n,1) + C(n,2) + C(n,3) + C(n,4) \\ &= \Theta(n) + \Theta(n^2) + \Theta(n^2) + O(n^2) = \Theta(n^2) \implies T(n) = \Theta(n^2) \end{aligned}$$

The worst-case running time of bubblesort is therefore $O(n^2)$, similar to the worst-case running time of insertion sort, which is also $O(n^2)$. However, the best-case running time of insertion sort, $\Omega(n)$, is better than the best-case running time of bubblesort, $\Omega(n^2)$.

Exercise 3. *Let $A[1 \,..\, n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A.*

(1) List the five inversion of the array $\langle 2, 3, 8, 6, 1 \rangle$.

(2) What array with elements from the set $\{1, 2, \dots, n\}$ has the most inversions? How many does it have?

(3) What is the relationship between the running time of insertion sort and the number of inversion in the input array? Justify your answer.

*(4) Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint*: Modify merge sort.)*

Solution.

(1) The set of inversions of the array is $\{(1,5),(2,5),(3,4),(3,5),(4,5)\}$.

(2) The reverse-sorted array $\{n, n-1, \ldots, 2, 1\}$ has the most inversions, since for each index i, $A[i] > A[j]$ for all $j > i$. Consequently, the first element n makes $n-1$ inversions, the second element $n-2$ makes $n-3$ inversions, and so on until the last element, 1, which makes zero inversions.

The total number of inversions is therefore given by

$$\sum_{i=1}^{n}(i-1) = \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2}$$

(3) From the running time analysis of insertion sort covered in class, we know that for each inversion in the array, the algorithm needs to do more work to sort it. In particular, for the following implementation of insertion sort:

```
1  for $j = 2$ to $A.length$
2      $key = A[j]$
3      $i = j - 1$
4      while $i > 0$ and $A[i] > key$
5          $A[i+1] = A[i]$
6          $i = i - 1$
7      $A[i+1] = key$
```

If we let t_j be the number of times the **while** loop is fully run, t_j is determined by the number of i's ($i < j$) such that $A[i] > A[j]$. In other words, t_j is the number of inversions (a, b) of the array such that $b = j$.

For an array that's close to sorted, the number of inversions is small and the time complexity of the algorithm is determined primarily by the main **for** loop, and is therefore $\Theta(n)$. For arrays with more inversions, the running time is best-approximated by $\Theta(n^2)$. The running time of insertion sort is therefore directly proportional to the number of inversions in the input array.

Exercise 4. *Most computers can perform the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving more quickly than computing remainders. This problem investigates the* ***binary gcd algorithm****, which avoids the remainder computations used in Euclid's algorithm.*

(1) Prove that if a and b are both even, then $\gcd(a,b) = 2 \cdot \gcd(a/2, b/2)$.
(2) Prove that if a is odd and b is even, then $\gcd(a,b) = \gcd(a, b/2)$.
(3) Prove that if a and b are both odd, then $\gcd(a,b) = \gcd((a-b)/2, b)$.
(4) Design an efficient binary gcd algorithm for input integers a and b, where $a \geq b$, that runs in $O(\lg a)$ times. Assume that each subtraction, parity test, and halving takes unit time.

Solution.

(1) We define $P(s)$ to be the prime factorisation of $|s|$ for some integer s. Since any divisor of s can be represented as the product over some subset of $P(s)$, we have

$$\{d \in \mathbb{N} : d|s\} = \left\{ \prod_{f \in A} f : A \subset P(s) \right\}$$

Therefore, by definition of $\gcd(a,b)$, we have

$$\gcd(a,b) = \max\{d \in \mathbb{N} : d|a \text{ and } d|b\} = \prod_{f \in P(a) \cap P(b)} f$$

Let a and b be even. Since 2 divides both a and b (by assumption), $2 \in P(a) \cap P(b)$. Since, by definition of gcd, we know $\gcd(a,b)$ is the product over $P(a) \cap P(b)$, $\gcd(a,b)$ is must be divisible by 2.

Now consider $a/2$ and $b/2$. The prime factorisations of $a/2$ and $b/2$ are, by definition, exactly $P(a)\backslash\{2\}$ and $P(b)\backslash\{2\}$ respectively. We therefore have $P(a/2) \cap P(b/2) = (P(a) \cap P(b))\backslash\{2\}$. Since $P(a)$ and $P(b)$ both contain 2 (by assumption on evenness of a and b), we therefore have

$$P(a/2) \cap P(b/2) = (P(a) \cap P(b))\backslash\{2\} \implies (P(a/2) \cap P(b/2)) \cup \{2\} = P(a) \cap P(b)$$

We therefore know $\gcd(a,b)$ is twice the product over the intersection $P(a/2) \cap P(b/2)$, i.e., $\gcd(a,b) = 2\gcd(a/2, b/2)$. ■

(2) Let a be odd and b be even. We have, by assumption $2 \notin P(a)$ and $2 \in P(b)$. We therefore have $2 \notin P(a) \cap P(b) \implies P(a) \cap P(b) = P(a) \cap (P(b)\backslash\{2\}) = P(a) \cap P(b/2)$. Taking the product over each side, by definition of gcd, we have $\gcd(a,b) = \gcd(a, b/2)$. ■

(3) Let $d \in CD(a,b)$ be arbitrary. By definition of $CD(a,b)$, there exist $q_1, q_2 \in \mathbb{Z}\backslash\{0\}$ such that $a = q_1 d$ and $b = q_2 d$. We therefore have $a - b = (q_1 - q_2)d$. The case where $q_1 = q_2$ is trivial, since then $a = b$ and $\gcd(a,b) = |b| = \gcd(0, b) = \gcd(\frac{a-b}{2}, b)$ (if $b \neq 0$).

We consider the case where $q_1 \neq q_2$. We therefore have $q_3 := q_1 - q_2 \in \mathbb{Z}\backslash\{0\}$ such that $a - b = q_3 d$. Therefore, $d \in CD(a,b)$ implies $d \in CD(a-b, b)$.

Conversely, for arbitrary $d \in CD(a-b, b)$, there exists $q_1, q_2 \in \mathbb{Z}\backslash\{0\}$ such that $a - b = q_1 d$ and $b = q_2 d$. We therefore have $a = (q_1 + q_2)d$. The case where $q_1 = -q_2$ is not possible, since then a must be zero and not odd. We therefore know there exists $q_3 := q_1 + q_2 \in \mathbb{Z}\backslash\{2\}$ such that $a = q_3 d$. Therefore, $d \in CD(a-b, b)$ implies $d \in CD(a,b)$.

We therefore have $CD(a,b) = CD(a-b, b)$ and, consequently, $\gcd(a,b) = \gcd(a-b, b)$. Since $a - b$ is even and b is odd, from (2), we have $\gcd(a-b, b) = \gcd(\frac{a-b}{2}, b)$.

Hence shown $\gcd(a,b) = \gcd(\frac{a-b}{2}, b)$. ■

(4) See below pseudocode for BinaryGCD, which takes in two integers a and b (assumed $a \geq b \geq 0$) and returns their GCD:

```
BinaryGCD($a, b$)
1   if $b == 0$
2       return $a$
3   else if $a$ and $b$ are even  // case (1): $a$ and $b$ are both even
4       return $2 * $ BinaryGCD($a/2, b/2$)
5   else if only $a$ is even  // case (2): one of $a$ and $b$ is even
6       if $a/2 >= b$
7           return BinaryGCD($a/2, b$)
8       else return BinaryGCD($b, a/2$)
9   else if only $b$ is even  // case (2)
10      return BinaryGCD($a, b/2$)
11  else  // case (3): $a$ and $b$ are both odd
12      if $(a-b)/2 >= b$
13          return BinaryGCD($(a-b)/2, b$)
14      else return BinaryGCD($b, (a-b)/2$)
```

At each call of BinaryGCD, we check which of cases (1), (2) and (3) (from Exercise 4) apply, and reduce a and/or b accordingly, while also maintaining the inequality $a \geq b \geq 0$. The algorithm terminates when b has been fully reduced (i.e., $b = 0$) since then we have $\gcd(a, b) = \gcd(a, 0) = a$.[2]

Exercise 5. *Let*

$$p(n) \;=\; \sum_{i=0}^{d} a_i n^i,$$

where $a_d > 0$, be a degree d polynomial in n, and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties:

(1) If $k \geq d$, then $p(n) = O(n^k)$.
(2) If $k \leq d$, then $p(n) = \Omega(n^k)$.
(3) If $k = d$, then $p(n) = \Theta(n^k)$.
(4) If $k > d$, then $p(n) = o(n^k)$.
(5) If $k < d$, then $p(n) = \omega(n^k)$.

Solution.
(1) By definition of O-notation, $p(n) = O(n^k)$ if there exist positive constants c, n_0 such that $0 \leq p(n) \leq cn^k$ for all $n \geq n_0$. Let $n_0 := 1$, $a_M := \max\{|a_i|\}_{i=1}^{d} \geq a_i$ for $i = 0, 1, \dots, d$. Furthermore, for $n \geq 1$, we have $n^k \geq n^d \geq n^i$ for $i = 0, 1, \dots, d$ (by assumption $k \geq d$). We therefore have for all $n \geq 1$, $i = 0, 1, \dots, d$

$$n^k \geq n^i,\ a_M \geq a_i \implies a_M n^k \geq a_i n^i \implies \sum_{i=0}^{d} a_M n^k \geq \sum_{i=0}^{d} a_i n^i = p(n) \implies a_M(d+1)n^k \geq p(n)$$

Therefore, for $c := (d+1)a_M$, $n_0 = 1$, we have $p(n) \leq cn^k$ for all $n \geq n_0$, and, consequently, $p(n) = O(n)$. ■

(2) By definition of Ω-notation, $p(n) = \Omega(n^k)$ if there exist positive constants c, n_0 such that $0 \leq cn^k \leq p(n)$ for all $n \geq n_0$.

Let $c := \frac{a_d}{2}$, $a_M := \max\{|a_i|\}_{i=1}^{d} \geq a_i$ for $i = 0, 1, \dots, d$. We then define $n_0 := 2d \left\lceil \frac{a_M}{a_d} \right\rceil$. Note that since $a_M \geq a_d$ (by definition), we have $\left\lceil \frac{a_M}{a_d} \right\rceil \geq 1$, and consequently, $n_0 = 2d \left\lceil \frac{a_M}{a_d} \right\rceil \geq 2d \geq 2$.

[2]See Appendix II for C++ code

We then have, for $n \geq n_0$

$$\begin{aligned}
p(n) &= \sum_{i=0}^{d} a_i n^i = a_d \sum_{i=0}^{d} \frac{a_i}{a_d} n^i = a_d \left(\sum_{i=0}^{d-1} \frac{a_i}{a_d} n^i + n^d \right) \\
&= a_d \left(\sum_{i=0}^{d-1} \frac{a_i}{a_d} n^i + \frac{n}{2} \cdot n^{d-1} + \frac{n}{2} \cdot n^{d-1} \right) \\
&\geq a_d \left(\sum_{i=0}^{d-1} \frac{a_i}{a_d} n^i + d\frac{a_M}{a_d} \cdot n^{d-1} + \frac{n^d}{2} \right) && \left(\because n \geq n_0 = d \left\lceil \frac{a_M}{a_d} \right\rceil \geq d\frac{a_m}{a_d} + \frac{n^d}{2} \right) \\
&= a_d \left(\sum_{i=0}^{d-1} \frac{a_i}{a_d} n^i + \sum_{i=0}^{d-1} \frac{a_M}{a_d} \cdot n^{d-1} + \frac{n^d}{2} \right) \\
&= a_d \left(\sum_{i=0}^{d-1} \frac{a_M \cdot n^{d-i-1} + a_i}{a_d} \cdot n^i + \frac{n^d}{2} \right) \\
&\geq a_d \left(0 + \frac{n^d}{2} \right) = a_d \frac{n^d}{2} && (\because \text{by definition, } a_M \geq |a_i| \; \forall i, \text{ and } n \geq 2) \\
&\geq a_d \frac{n^k}{2} = cn^k && (\text{by assumption k} \leq d \text{ and } n \geq 2)
\end{aligned}$$

$$\therefore p(n) \geq cn^k \geq 0 \quad \forall\; n \geq n_0 := 2d \left\lceil \frac{a_M}{a_d} \right\rceil, \; c := \frac{a_d}{2}$$

Therefore, for c and n_0 as defined above, we have $0 \leq cn^k \leq p(n)$ for all $n \geq n_0$, and, consequently, $p(n) = \Omega(n)$. ■

(3) By definition of Θ-notation, $p(n) = \Theta(n^k)$ if there exist positive constants c_1, c_2, n_0 such that $0 \leq c_1 n^k \leq p(n) \leq c_2 n^k$ for all $n \geq n_0$. From (1), we know for $k \geq d$, $p(n) = O(n^k)$, and for $k \leq d$, $p(n) = \Omega(n^k)$. Therefore for $k = d$, we know $p(n) = O(n^k)$ and $p(n) = \Omega(n^k)$, and consequently, $p(n) = \Theta(n^k)$. We show this more rigorously below.

Let $c_1 := \frac{a_d}{2}$ (i.e., c as defined in (3)), $c_2 := (d+1)a_M$ (i.e., c as defined in (2)), and $n_0 := \max\left\{1, 2d\left\lceil \frac{a_M}{a_d} \right\rceil\right\}$ (i.e., the maximum of n_0 as defined in (2) and (3) each). We therefore have, from (2) and (3),

$$0 \leq c_1 n^k \leq p(n) \leq c_2 n^k \quad \forall\; n \geq n_0$$

Therefore, by definition of Θ-notation, $p(n) = \Theta(n^k)$. ■

(4) By definition of o-notation, $p(n) = o(n^k)$ if $\lim_{n\to\infty} \frac{p(n)}{n^k} = 0$. We have

$$\lim_{n\to\infty} \frac{p(n)}{n^k} = \lim_{n\to\infty} \sum_{i=0}^{d} a_i \frac{n^i}{n^k} = \lim_{n\to\infty} \sum_{i=0}^{d} a_i \frac{1}{n^{k-i}}$$

For $k > d$, we have $k - i > 0$ for $i = 0, 1, \ldots, d$. We therefore have

$$\lim_{n\to\infty} \frac{p(n)}{n^k} = \lim_{n\to\infty} \sum_{i=0}^{d} a_i \frac{1}{n^{k-i}} = \sum_{i=0}^{d} a_i \lim_{n\to\infty} \frac{1}{n^{k-i}} = \sum_{i=0}^{d} a_i \cdot 0 = 0$$

Therefore, by definition of o-notation, we have $p(n) = o(n^k)$ for $k > d$. ■

(5) By definition of ω-notation, $p(n) = \omega(n^k)$ if $\lim_{n\to\infty} \frac{p(n)}{n^k} = \infty$. We have

$$\lim_{n\to\infty}\frac{p(n)}{n^k} = \lim_{n\to\infty}\sum_{i=0}^{d} a_i\frac{n^i}{n^k} = \lim_{n\to\infty} a_i\sum_{i=0}^{d} n^{i-k}$$

For $k < d$, we have $i-k<0$ for $i=0,1,\ldots,k-1$, and $i-k>0$ for $i=k,k+1,\ldots,d$. We therefore have

$$\lim_{n\to\infty}\frac{p(n)}{n^k} = \lim_{n\to\infty}\sum_{i=0}^{d} a_i n^{i-k} = \left(\sum_{i=0}^{k-1} a_i \lim_{n\to\infty} n^{i-k} + a_k \lim_{n\to\infty} n^{k-k} + \sum_{i=k+1}^{d} a_i \lim_{n\to\infty} n^{i-k}\right)$$

Since the powers of n are negative in the first sum, its limit as n tends to ∞ is 0. The second term has power 0 and naturally evalutes to 1. The third term, however, contains only positive powers of n (and at-least one term, since $k < d$ implies the sum from $k+1$ to d is not an empty sum) and therefore evaluates to ∞.

We therefore have, by definition of o-notation, $p(n) = \omega(n^k)$ for $k < d$. ■

Exercise 6. *Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.*

(1) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.
(2) $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.
(3) $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for sufficiently large n.
(4) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.
(5) $f(n) = O((f(n))^2)$.
(6) $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.
(7) $f(n) = \Theta(f(n/2))$.
(8) $f(n) + o(f(n)) = \Theta(f(n))$.

Solution.
(1) We show $f(n) = O(g(n))$ does not imply $g(n) = O(f(n))$ by counter-example. Let $f(n) := 1$, $g(n) := n$. Since for $c := 1$, $n_0 := 1$, we have $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$, by definition of O-notation, $f(n) = O(g(n))$.

We show $g(n) \neq O(f(n))$. Let $c > 0$, $n_0 > 0$ be arbitrary. Let $N := \max\{n_0, c\} + 1$. By definition, we have $N > n_0$ and $N > c$. In other words, $g(N) \geq cf(N)$ for $N > n_0$. Since n_0 and c were chosen arbitrarily, this is sufficient to show $g(n) \neq O(f(n))$.

The conjecture is therefore false. ■

(2) We show $f(n) + g(n) \neq \Theta(\min(f(n), g(n)))$ by counter-example.

Let $f(n) := 1$, $g(n) := n$. For all $n \geq 1$, we know $\min(f(n), g(n)) = \min(1, n) = 1$. It is therefore sufficient to show $f(n) + g(n) = n + 1 \neq Theta(1)$.

Let $c > 0$, $n_0 > 0$ be arbitrary. Let $N := \max\{n_0, c\} + 1$. By definition, we have $N > n_0$ and $N + 1 > N > c$. In other words, $g(N) + f(N) = N + 1 \geq c = cf(N)$ for $N > n_0$. Since n_0 and c were chosen arbitrarily, this is sufficient to show $g(n) + f(n) \neq O(\min(f(n), g(n)))$, which in turn implies $g(n) + f(n) \neq \Theta(\min(f(n), g(n)))$.

The conjecture is therefore false. ■

(3) We show this conjecture is true. By assumption that $f(n) = O(g(n))$, and by definition of O-notation, we know there exist positive constants c_1, n_1 such that $0 \leq f(n) \leq c_1 g(n)$ for all $n \geq n_1$. Let c_1, n_1 be as defined above.

By assumption that $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for sufficiently large n, we know there exists n_2 such that $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all $n \geq n_2$. Let n_2 be as defined above.

Let $n_0 := \max n_1, n_2$. By definition of n_1 and n_2, we have for all $n \geq n_0$

$$0 \leq f(n) \leq c_1 g(n), \ \lg(g(n)) \geq 1, \ f(n) \geq 1$$

Since $f(n) \geq 0$ for all $n \geq n_0$, we also know $\lg(f(n)) \geq 0$ (because only for non-negative k is $2^k \geq 1$). Furthermore, since we already know $\lg(g(n) \geq 1 > 0$ and lg is a continuous, increasing function, the inequality is preserved when we apply lg to it to get

$$0 \leq \lg(f(n)) \leq \lg(c_1 g(n)) = \lg c_1 + \lg(g(n)) \tag{1}$$

Let $c := \lg c_1 + 1$. Applying the assumption $\lg(g(n)) \geq 1$ for all $n \geq n_0$, we have

$$c \lg(g(n)) = (\lg c_1 + 1) \lg(g(n)) = \lg c_1 \cdot \lg(g(n)) + \lg(g(n)) \geq \lg c_1 + \lg(g(n)) \tag{2}$$

Therefore, from inequalities (1) and (2), we have for all $n \geq n_0$

$$0 \leq \lg(f(n)) \leq \lg c_1 + \lg(g(n)) \leq c \lg(g(n)) \implies 0 \leq \lg(f(n)) \leq c \lg(g(n))$$

Therefore, there exist positive constants c, n_0 such that $0 \leq \lg(f(n)) \leq c \lg(g(n))$ for all $n \geq n_0$. By definition of O-notation, we therefore have $\lg(f(n)) = O(\lg(g(n)))$. ■

(4) We show $f(n) = O(g(n))$ does not imply $2^{f(n)} = O(2^{g(n)})$ by counter-example. Let $f(n) := 2 \lg n$, $g(n) := \lg n$. Since for $c := 3$, $n_0 := 1$, we have $0 \leq f(n) = 2 \lg n \leq 3 \lg n = cg(n)$ for all $n \geq n_0$, by definition of O-notation, $f(n) = O(g(n))$.

We show $2^{f(n)} \neq O(2^{g(n)})$. Let $c > 0$, $n_0 > 0$ be arbitrary. We have $2^{f(n)} = 2^{2 \lg n} = 2^{\lg n^2} = n^2$, and $2^{(g(n)} = 2^{\lg n} = n$. Let $N := \max\{n_0, c\}$. By definition, we have $N > n_0$ and $N > c$. Multiplying both sides of the latter inequality by N, we have $N^2 > cN$. In other words, $2^{f(N)} > 2^{g(N)}$ for $N > n_0$.

Since n_0 and c were chosen arbitrarily, this is sufficient to show $2^{f(n)} = n^2 \neq O(n) = O(2^{g(n)})$. This conjecture is therefore false. ■

(5) We show $f(n) \neq O(f(n)^2)$ by counter-example. Let $f(n) := 1/n$. Since $1/n > 0$ for all $n > 0$, $f(n)$ is asymptotically positive. Let $c > 0$, $n_0 > 0$ be arbitrary. Let $N := \max\{c, n_0\} + 1$. By definition, we have $N > n_0$ and $N > c$. Dividing both sides of the latter inequality by N^2, we have $1/N > c/N^2$. In other words, $f(N) > cf(N)^2$ for $N > n_0$.

Since n_0 and c were chosen arbitrarily, this is sufficient to show $f(n) \neq O(f(n)^2)$. The conjecture is therefore false. ■

(6) We show this conjecture is true. By assumption that $f(n) = O(g(n))$, and by definition of O-notation, we know there exist positive constants c_1, n_1 such that $0 \leq f(n) \leq c_1 g(n)$ for all $n \geq n_1$. Let c_1, n_1 be as defined above. Dividing the inequality by c_1, we have $0 \leq \frac{1}{c} f(n) \leq g(n)$ for all $n \geq n_1$. Therefore, for $n_0 := n_1$ and $c := 1/c$, and by definition of Ω-notation, we know $g(n) = \Omega(f(n))$. ■

(7) We show $f(n) \neq \Theta(f(n/2))$ by counter-example. We define $f(n)$ as:

$$f(n) := \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

Assume towards contradiction that $f(n) = \Theta(f(n/2))$. By definition of Θ-notation, there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 f(n/2) \leq f(n) \leq c_2 f(n/2)$ for all $n \geq n_0$. Let c_1, c_2, and n_0 be as defined above.

Let $N := 2\max\{n_0, c_2\}$. By definition, we have $N > n_0$, $N > c_2$, and N is even. We define N' as:

$$N' := \begin{cases} N & \text{if } N/2 \text{ is odd} \\ N+2 & \text{if } N/2 \text{ is even} \end{cases}$$

By definition of N', we know $N' > n_0$ and $N'/2$ is odd. Furthermore, we know $N' > c_2$. From $f(N') = N'$ (N' is even) and $f(N'/2) = 1$ ($N'/2$ is odd), we have

$$N' > c_2 \implies f(N') > c_2 f(N'/2)$$

which contradicts our assumption $f(n) \leq c_2 f(n/2)\ \forall n \geq n_0$. The conjecture is therefore false. ■

(8) We show $f(n) + o(f(n)) = \Theta(f(n))$ directly. Let $g(n)$ be an arbitrary function such that $g(n) \in o(f(n))$. By definition of o-notation, we know for all $c > 0$, there exists $n_1 > 0$ such that $g(n) < cf(n)$ for all $n \geq n_0$. Let n_1 be as defined above for $c := 1$.

Since $g(n) \in o(f(n))$ and $f(n)$ is asymptotically positive, $g(n)$ is also asymptotically positive. Let $n_2 > 0$ be defined such that $g(n) \geq 0$ for all $n \geq n_2$.

Let $n_0 := \max\{n_1, n_2\}$. We therefore have, by definition of n_1 and n_2, $f(n) \leq f(n) + g(n) \leq f(n) + f(n) = 2f(n)$ for all $n \geq n_0$. By definition of Θ-notation, we therefore know $f(n) + o(f(n)) = \Theta(f(n))$. The conjecture is therefore true. ■

Exercise 7. *Suppose you have algorithms with the six running times listed below. (Assume these are the exact number of operations performed as a function of the input size n.) Suppose you have a computer that can perform 10^{10} operations per second, and you need to compute a result in at most an hour of computation. For each of the algorithms, what is the largest input size n for which you would be able to get the result within an hour?*

(1) n^2
(2) n^3
(3) $100n^2$
(4) $n \lg n$
(5) 2^n
(6) 2^{2^n}.

Solution. A computer that can perform 10^{10} operations per second can perform $60 \cdot 60 \cdot 10^{10} = 3.6 \cdot 10^{13}$ operations in an hour. In any function $f(n)$, we therefore find the maximum n such that $f(n) \leq 3.6 \cdot 10^{13}$.

(1) We have $f(n) := n^2$. From the above inequality, we get

$$n^2 \leq 3.6 \cdot 10^{13} \implies n \leq \sqrt{3.6 \cdot 10^{13}} = 6 \cdot 10^6$$

The largest input size n for which we could get the result within an hour for is therefore $6 \cdot 10^6$.

(2) We have $f(n) := n^3$. From the above inequality, we get

$$n^3 \leq 3.6 \cdot 10^{13} \implies n \leq \sqrt[3]{3.6 \cdot 10^{13}} \approx 33019.27$$

The largest input size n for which we could get the result within an hour for is therefore 33019.

(3) We have $f(n) := 100n^2$. From the above inequality, we get

$$100n^2 \leq 3.6 \cdot 10^{13} \implies n \leq \sqrt{3.6 \cdot 10^{11}} \approx 33019.27$$

The largest input size n for which we could get the result within an hour for is therefore $6 \cdot 10^5$.

(4) We have $f(n) := n \lg n$. From the above inequality, we get

$$n \lg n \leq 3.6 \cdot 10^{13}$$

This inequality cannot easily be solved analytically. Solving it numerically, we get The largest input size n for which we could get the result within an hour for is 906316482853.

(5) We have $f(n) := 2^n$. From the above inequality, we get

$$2^n \leq 3.6 \cdot 10^{13} \implies n \leq \lg(3.6 \cdot 10^{13}) \approx 45.033$$

The largest input size n for which we could get the result within an hour for is therefore 45.

(6) We have $f(n) := 2^{2^n}$. From the above inequality, we get

$$2^{2^n} \leq 3.6 \cdot 10^{13} \implies n \leq \lg\lg(3.6 \cdot 10^{13}) \approx 5.493$$

The largest input size n for which we could get the result within an hour for is therefore 5.

Exercise 8. *Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$. Justify all consecutive comparisons.*

(1) $g_1(n) = 2^{\sqrt{\lg n}}$
(2) $g_2(n) = 2^n$
(3) $g_3(n) = n^{4/3}$
(4) $g_4(n) = n(\lg n)^3$
(5) $g_5(n) = n^{\lg n}$
(6) $g_6(n) = 2^{2^n}$
(7) $g_7(n) = 2^{n^2}$

Solution. The functions are sorted in ascending order of growth rate as follows:

(1) $g_1(n) = 2^{\sqrt{\lg n}}$
(2) $g_4(n) = n(\lg n)^3$
(3) $g_3(n) = n^{4/3}$
(4) $g_5(n) = n^{\lg n}$
(5) $g_2(n) = 2^n$
(6) $g_7(n) = 2^{n^2}$
(7) $g_6(n) = 2^{2^n}$

In fact, each function is a non-tight asymptotic upper bound of the function before it. We prove this below.

(1) We show $g_1(n) = o(g_4(n))$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{g_1(n)}{g_4(n)} &= \lim_{n\to\infty} \frac{2^{\sqrt{\lg n}}}{n \lg n} = \lim_{n\to\infty} \text{two}\left(\lg\left(\frac{2^{\sqrt{\lg n}}}{n\lg n}\right)\right) \qquad (\text{for two}(n) := 2^n) \\
&= \text{two}\left(\lim_{n\to\infty}\left(\sqrt{\lg n} - \lg(n \lg n)\right)\right) \\
&= \text{two}\left(\lim_{n\to\infty}\left(\sqrt{\lg n} - (\lg n + \lg\lg n)\right)\right) \\
&= \text{two}(-\infty) = 0 \\
\implies g_1(n) &= o(g_4(n)) \quad \blacksquare
\end{aligned}$$

(2) We show $g_4(n) = o(g_3(n))$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{g_4(n)}{g_3(n)} &= \lim_{n\to\infty} \frac{n(\lg n)^3}{n^{4/3}} = \frac{3}{4} \lim_{n\to\infty} \frac{(\lg n)^3 + 3(\lg n)^2}{n^{1/3}} && (\text{L}'\text{Hôpital}'\text{s Rule}) \\
&= \frac{9}{4} \lim_{n\to\infty} \frac{3(\lg n)^2 + 6 \lg n}{n^{1/3}} && (\text{L}'\text{Hôpital}'\text{s Rule}) \\
&= \frac{27}{4} \lim_{n\to\infty} \frac{6(\lg n) + 6}{n^{1/3}} && (\text{L}'\text{Hôpital}'\text{s Rule}) \\
&= \frac{81}{4} \lim_{n\to\infty} \frac{6}{n^{1/3}} && (\text{L}'\text{Hôpital}'\text{s Rule}) \\
&= 0
\end{aligned}$$

$$\implies g_4(n) = o(g_3(n)) \quad \blacksquare$$

(3) We show $g_3(n) = o(g_5(n))$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{g_3(n)}{g_5(n)} &= \lim_{n\to\infty} \frac{n^{4/3}}{n^{\lg n}} = \lim_{n\to\infty} \text{two}\left(\lg\left(\frac{n^{4/3}}{n^{\lg n}}\right)\right) && (\text{for } \text{two}(n) := 2^n) \\
&= \text{two}\left(\lim_{n\to\infty}\left(\frac{4}{3}\lg n - (\lg n)^2\right)\right) \\
&= \text{two}(-\infty) = 0
\end{aligned}$$

$$\implies g_3(n) = o(g_5(n)) \quad \blacksquare$$

(4) We show $g_5(n) = o(g_2(n))$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{g_5(n)}{g_2(n)} &= \lim_{n\to\infty} \frac{n^{\lg n}}{2^n} = \lim_{n\to\infty} \text{two}\left(\lg\left(\frac{n^{\lg n}}{2^n}\right)\right) && (\text{for } \text{two}(n) := 2^n) \\
&= \text{two}\left(\lim_{n\to\infty}\left((\lg n)^2 - n\right)\right) \\
&= \text{two}\left(\lim_{n\to\infty} n\left(\frac{(\lg n)^2}{n} - 1\right)\right)
\end{aligned}$$

We show $\lim_{n\to\infty} (\lg n)^2/n$ exists and equals 0 by successively applying L'Hôpital's Rule:

$$\lim_{n\to\infty} \frac{(\lg n)^2}{n} = \lim_{n\to\infty} \frac{2\lg n}{n} = \lim_{n\to\infty} \frac{2}{n} = 0 \implies \lim_{n\to\infty}\left(\frac{(\lg n)^2}{n} - 1\right) = -1$$

Applying this limit, we get

$$\begin{aligned}
\lim_{n\to\infty} \frac{g_5(n)}{g_2(n)} &= \text{two}\left(\lim_{n\to\infty} n\left(\frac{(\lg n)^2}{n} - 1\right)\right) \\
&= \text{two}\left(\lim_{n\to\infty} n \cdot \lim_{n\to\infty}\left(\frac{(\lg n)^2}{n} - 1\right)\right) && (\text{the limit exists}) \\
&= \text{two}\left(\lim_{n\to\infty} n \cdot (-1)\right) = \text{two}\left(\lim_{n\to\infty} -n\right) \\
&= \text{two}(-\infty) = 0
\end{aligned}$$

$$\implies g_5(n) = o(g_2(n)) \quad \blacksquare$$

(5) We show $g_2(n) = o(g_7(n))$.

$$\begin{aligned}\lim_{n\to\infty}\frac{g_2(n)}{g_7(n)} &= \lim_{n\to\infty}\frac{2^n}{2^{n^2}} = \lim_{n\to\infty}\text{two}\left(\lg\left(\frac{2^n}{2^{n^2}}\right)\right) && (\text{for } \text{two}(n) := 2^n)\\ &= \text{two}\left(\lim_{n\to\infty}(n-n^2)\right)\\ &= \text{two}(-\infty) = 0\\ \implies g_2(n) &= o(g_7(n)) \quad \blacksquare\end{aligned}$$

(6) We show $g_7(n) = o(g_6(n))$.

$$\begin{aligned}\lim_{n\to\infty}\frac{g_7(n)}{g_7(n)} &= \lim_{n\to\infty}\frac{2^{n^2}}{2^{2^n}} = \lim_{n\to\infty}\text{two}\left(\lg\left(\frac{2^{n^2}}{2^{2^n}}\right)\right) && (\text{for } \text{two}(n) := 2^n)\\ &= \text{two}\left(\lim_{n\to\infty}(n^2-2^n)\right) = \text{two}\left(\lim_{n\to\infty}2^n\left(\frac{n^2}{2^n}-1\right)\right)\end{aligned}$$

We show $\lim_{n\to\infty} n^2/2^n$ exists and equals 0 by successively applying L'Hôpital's Rule:

$$\lim_{n\to\infty}\frac{n^2}{2^n} = \frac{2}{\ln 2}\lim_{n\to\infty}\frac{n}{2^n} = \frac{2}{(\ln 2)^2}\lim_{n\to\infty}\frac{1}{2^n} = 0 \implies \lim_{n\to\infty}\left(\frac{(n^2}{2^n}-1\right) = -1$$

Applying this limit, we get

$$\begin{aligned}\lim_{n\to\infty}\frac{g_7(n)}{g_6(n)} &= \text{two}\left(\lim_{n\to\infty}2^n\left(\frac{n^2}{2^n}-1\right)\right)\\ &= \text{two}\left(\lim_{n\to\infty}2^n\cdot\lim_{n\to\infty}\left(\frac{n^2}{2^n}-1\right)\right) && (\text{the limit exists})\\ &= \text{two}\left(\lim_{n\to\infty}n^2\cdot(-1)\right) = \text{two}\left(\lim_{n\to\infty}-n^2\right)\\ &= \text{two}(-\infty) = 0\\ \implies g_7(n) &= o(g_6(n)) \quad \blacksquare\end{aligned}$$

Exercise 9. *Dr. I. J. Matrix has observed a remarkable sequence of formulas:*

$$9\times 1+2 \;=\; 11, \quad 9\times 12+3 \;=\; 111, \quad 9\times 123+4 \;=\; 111, \quad 9\times 1234+5 \;=\; 11111.$$

(1) Write the good doctor's great discovery in terms of the Σ-notation.
(2) Your answer to part (1) undoubtedly involves the number 10 as base of the decimal system; generalize this formula so that you get a formula that will perhaps work in any base b.
(3) Prove your formula from part (2). The summation formulas from HW1 might be helpful.

Solution. (1) In Σ-notation, we have for some integer $k \geq 1$,

$$9\times\sum_{i=0}^{k-1}(k-i)\cdot 10^i + (k+1) = \sum_{i=0}^{k} 10^k$$

(2) Generalising to any base b, we have for any integer $k \geq 1$,

$$(b-1)\times\sum_{i=0}^{k-1}(k-i)\cdot b^i + (k+1) = \sum_{i=0}^{k} b^k$$

(3) Expanding for any base b, we have

$$L := (b-1) \cdot \sum_{i=0}^{k-1} (k-i) \cdot b^i + (k+1) = (b-1) \cdot \left(k \sum_{i=0}^{k-1} b^i - \sum_{i=0}^{k-1} i b^i \right) + k + 1$$

From Homework 1 Exercise 3, we have

$$\sum_{j=0}^{n} j x^j = \frac{n x^{n+2} - (n+1) x^{n+1} + x}{(x-1)^2} \implies \sum_{i=0}^{k-1} i b^i = \frac{(k-1) b^{k+1} - k b^k + b}{(b-1)^2}$$

Applying the formula for a geometric sum and the above expression to the respective summations, we have

$$\begin{aligned} L &= (b-1) \cdot \left(k \sum_{i=0}^{k-1} b^i - \sum_{i=0}^{k-1} i b^i \right) + k + 1 \\ &= (b-1) \cdot \left(k \frac{b^k - 1}{b-1} - \frac{(k-1) b^{k+1} - k b^k + b}{(b-1)^2} \right) + k + 1 \\ &= k(b^k - 1) - \frac{(k-1) b^{k+1} - k b^k + b}{(b-1)} + k + 1 \\ &= \frac{k(b^k - 1)(b-1) - ((k-1) b^{k+1} - k b^k + b) + (k+1)(b-1)}{b-1} \\ &= \frac{k b^{k+1} - k b^k - kb + k - k b^{k+1} + b^{k+1} + k b^k - b + kb - k + b - 1}{b-1} \qquad = \frac{b^{k+1} - 1}{b-1} = \sum_{i=0}^{k} b^k \end{aligned}$$

Hence proven that the general formula works for any base b. ■

Exercise 10. *Let $(F_n)_{n \geq 0}$ be the sequence of Fibonacci numbers, i.e., $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.*

(1) Prove that for all $n \geq 0$, $F_n = (\phi^n - \hat{\phi}^n)/\sqrt{5}$ where $\phi = (1+\sqrt{5})/2$ and $\hat{\phi} = (1-\sqrt{5})/2$. Hint: Use that $\phi, \hat{\phi}$ are both roots of the quadratic polynomial $x^2 - x - 1$.

(2) Let $T(n)$ be the running time of Fibonacci. *Prove that $T(n) = \Theta(F_n)$.*

(3) Prove that $F_n = \Theta(\phi^n)$. Conclude that $T(n) = \Theta(\phi^n)$ runs in exponential time.

Exercise 11 (Programming Exercise)**.** *Recall the nth triangular number is given by $T_n = n(n+1)/2$, so the first few triangular numbers are*

$$1,\ 3,\ 6,\ 10,\ 15,\ 21,\ 28,\ 36,\ 45,\ 55,\ \ldots$$

We can list the divisors of the first seven triangular numbers:

$$\begin{aligned} 1 &: 1 \\ 3 &: 1, 3 \\ 6 &: 1, 2, 3, 6 \\ 10 &: 1, 2, 5, 10 \\ 15 &: 1, 3, 5, 15 \\ 21 &: 1, 3, 7, 21 \\ 28 &: 1, 2, 4, 7, 14, 28 \end{aligned}$$

We see that 28 *is the first triangular number to have more than five divisors. What is the value of the first triangular number to have over a thousand divisors?*

Solution. The following function, `HighlyCompositeTriangular`, takes input `minDivisors` and returns the smallest triangular number that has more than `minDivisors` divisors.

```
#include<cmath> // for sqrt(n)
using namespace std;

int nDivisors(int num);

int HighlyCompositeTriangular(int minDivisors) {
  // returns the smallest triangular number with more than minDivisors divisors

  // iterating till desired number is found
  for (int n = 1; true; n++) {
    int T_n = (n * (n + 1)) / 2;
    if (nDivisors(T_n) > minDivisors)
      return T_n;
      }
}

int nDivisors(int num) {
  // returns the number of divisors of integer num

  int divisorCount = 0; // number of divisors found

  double numRoot = sqrt(num); // floor of sqrt(n)
    // (minor optimisation: don't want to compute sqrt(n) each iteration)

  // iterating up to numRoot (the smaller element of each divisor pair of n is <=
   nRoot)
  for (int d = 1; d < numRoot; d++)
    // checking if divisor pair found
    if (num % d == 0)
      divisorCount += 2;

  // if n is perfect square -> add one divisor
  if (numRoot - floor(numRoot) == 0)
    divisorCount++;

  return divisorCount;
}
```

Calling `HighlyCompositeTriangular(1000)` returns 842161320.

Exercise 12 (Programming exercise)**.** *The following iterative sequence is defined for the set of positive integers:*

$$\begin{aligned} n &\rightarrow n/2 \quad (\textit{if } n \textit{ is even}) \\ n &\rightarrow 3n+1 \quad (\textit{if } n \textit{ is odd}) \end{aligned}$$

Using the rule above and starting with 13*, we generate the following sequence:*

$$13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

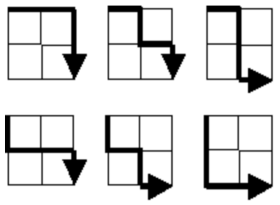
It can be seen that this sequence contains 10 terms. It is conjectured that all starting numbers will finish at 1. *Which starting number, under two million, produces the longest chain? [Note: once the chain starts the terms are allowed to go above two million.]*

Solution. The following function, `longestCollatzChain`, takes input `bound` and returns the starting number under `bound` which has the longest Collatz sequence.

```
int chainLength(long long n);

int longestCollatzChain(int bound) {
  // returns the starting number under bound which generates the longest Collatz
    chain

  int maxChainLength = 0;   // maximum chain length currently found
  int nBest = 0;    // best starting number currently found

  // iterating up to bound
  for (int n = 1; n < bound; n++) {
    int nChainLength = chainLength(n);

    // checking if current chain is longest found so far
    if (nChainLength > maxChainLength) {
      // updating maxChainLength and nBest
      maxChainLength = nChainLength;
      nBest = n;
    }
  }

  // returning best starting number
  return nBest;
}

int chainLength(long long n) {
  // returns the length of the Collatz chain of n
  int length = 1;   // we include n in the chain

  // iterating until n is reduced to 1
  while (n != 1) {
    // divide by 2 if even
    if (n % 2 == 0)
      n = n / 2;
    // multiply by 3, add 1 if odd
    else n = 3 * n + 1;
    // increment chain length
    length++;
  }
  return length;
}
```

Calling `longestCollatzChain(2000000)` returns 1723519.

Exercise 13 (Programming exercise)**.** *Starting in the top left corner of a* 2×2 *grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner. How many such routes are there through a* 30×30 *grid?*



(Analytical) Solution. In any such route through an $n \times n$ grid, we must move right n times and down n times for a total of exactly $2n$ moves. The set of unique routes through an $n \times n$ grid is therefore given by the number of unique sequences of right movements and down movements. We can visual the number of such sets by imagining $2n$ empty spaces, each representing a movement. The number of unique ways to fill the spaces (and, consequently, get a unique sequence representing a route) is equal to the number of ways to choose n empty spaces to represent down movements, where the remaining n spaces automatically represent right movements. We therefore have

$$\text{Number of routes} = \binom{2n}{n} = \frac{(2n)!}{n!n!} = \frac{(2n)!}{(n!)^2}$$

For $n = 30$, we therefore have

$$\text{Number of routes} = \frac{(60)!}{(30!)^2} = 118264581564861424$$

Exercise 14 (Programming exercise)**.** *In the United Kingdom the currency is made up of pound (£) and pence (p). There are eight coins in general circulation:*

$$1p,\ 2p,\ 5p,\ 10p,\ 20p,\ 50p,\ £1\ (100p),\ \textit{and}\ £2\ (200p)$$

It is possible to make £2 in the following way:

$$1 \times £1 + 1 \times 50p + 2 \times 20p + 1 \times 5p + 1 \times 2p + 3 \times 1p$$

How many different ways can £2 be made using any number of coins? Here we don't care about the order of the coins, just how many of each of them there are.

Solution. The following function, `nPartitions`, takes as input integer n and a list of parts, `partSizes`, and returns number of partitions of n parts from `partSizes` can make.

```
int nPartitions(int n, vector<int> partSizes) {
  // returns the number of partitions of n composed of sizes from partSizes
  // dynamic programming solution

  vector<int> partitionSizes(n + 1, 0);
    // a list containing, for each number from 1 through n, the number of
   partitions it has
    // initialised to 0

  // base case: partitionSize[0] represents the termination of a partition
  partitionSizes[0] = 1;

  // iterating over part sizes
  for (int part : partSizes)
    // iterating from 1 through n
    for (int i = 1; i <= n; i++)
      // partitions(i) = partitions[i] + partitions[i - part]
      // (for only parts that have already been processed for smaller i)
      if (i - part >= 0)
        partitionSizes[i] += partitionSizes[i - part];

  // return partitions of n
  return partitionSizes[n];
}
```

Calling `nPartitions(200, vector<int> { 1, 2, 5, 10, 20, 50, 100, 200 })` returns 73682.

Exercise 15 (Programming exercise)**.** *An irrational decimal is created by concatenating the positive integers:*

$$0.123456789101112131415161718192 0\ldots$$

It can be seen that the 12th *digit of the decimal expansion is* 1.

If d_n represents the nth digit of the fractional part, find the value of the following expression:

$$d_1 \times d_{10} \times d_{100} \times d_{1000} \times d_{10000} \times d_{100000} \times d_{1000000}$$

Solution. The following function, `digitProduct`, takes as input a list of indices, `digitIndices`, and returns the product of the digits at those indices in the given irrational decimal.

```
#include<vector>
#include<string>
using namespace std;

int digitProduct(vector<int> digitIndices) {
  // returns the product of digits at each of digitIndices

  // finding max digit index
  unsigned int maxIndex = 0;
  for (int index : digitIndices)
    if (index > maxIndex)
      maxIndex = index;

  // creating string and reserving storage
  string fractionalPart;
  fractionalPart.reserve(maxIndex);

  // iterating until fractionalPart is complete
  for (int i = 1; true; i++) {

    // checking whether fractionalPart is complete
    if (fractionalPart.size() >= maxIndex)
      break;

    // inserting next integer
    fractionalPart += to_string(i);
  }

  // computing product by iterating over indices
  int prod = 1;
  for (int index : digitIndices)
    prod *= fractionalPart[index - 1] - '0';
```

```
        // fractionalPart[index - 1] is implictly cast to the int representing its
    ASCII code
        // subtracting '0' (i.e. the ASCII code for 0) converts it to the desired
    digit

  return prod;
}
```

Calling `digitProduct(vector<int> { 1, 10, 10, 100, 1000, 10000, 100000, 1000000 })` returns 210.

APPENDIX

I. C++ code for Exercise 1:

```cpp
#include<vector>
using namespace std;

vector<vector<int>> exercise1fast(vector<int> A) {
  vector<vector<int>> B(A.size(), vector<int>(A.size(), 0)); // create matrix
  // iterating over rows
  for (unsigned int i = 0; i < A.size(); i++) {
  B[i][i] = A[i]; // setting diagonal element (for correctness)
    // iterating over columns
    for (unsigned int j = i + 1; j < A.size(); j++)
      B[i][j] += B[i][j - 1] + A[j];  // setting value
  }
  return B;
}
```

II. C++ code for Exercise 4:

```cpp
int BinaryGCD(int a, int b) {
    // given two integers a and b, a >= b >= 0, return gcd of a and b
    if (b == 0)
        return a;

    // 2|a and 2|b
    else if (a % 2 == 0 && b % 2 == 0)
        return 2*BinaryGCD(a / 2, b / 2);
    // a is even, b is odd
    else if (a % 2 == 0) {
        if (a / 2 >= b)
            return BinaryGCD(a / 2, b);
        else return BinaryGCD(b, a / 2);
    }
    // b is even, a is odd
    else if (b % 2 == 0)
        return BinaryGCD(a, b / 2);
    // both a and b are odd
    else {
        if ((a - b) / 2 >= b)
            return BinaryGCD((a - b) / 2, b);
        else return BinaryGCD(b, (a - b) / 2);
    }
}
```