MATH182 HOMEWORK #2
DUE July 5, 2020

Note: while you are encouraged to work together on these problems with your classmates, your final work should be written in your own words and not copied verbatim from somewhere else. You need to do at least six (6) of these problems. All problems will be graded, although the score for the homework will be capped at $N :=$ (point value of one problem) $\times 6$ and the homework will be counted out of N total points. Thus doing more problems can only help your homework score. For the programming exercise you should submit the final answer (a number) *and* your program source code.

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) Let us analyze the running time of this algorithm:

1	**for** $i = 1$ **to** n	*cost* : c_1 *times* : $n + 1$
2	**for** $j = i + 1$ **to** n	*cost* : c_2 *times* : n
3	Add up array entries $A[i]$ through $A[j]$	*cost* : $*$ *times* : $*$
4	Store the result in $B[i, j]$	*cost* : c_4 *times* : $\sum_{i=1}^{n} n - i$

All aspects of the running time are clear, except line 3. Line 3 is not done in a constant amount of time, since we have to add the elements between $A[i]$ and $A[j]$, $j - i + 1$ elements with $j - i$ additions. So (ignoring constants), the total amount of work done by line 3 is

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} (j - i) \;=\; \frac{1}{6} n(n^2 - 1)$$

Now we'll compute all the work done:

$$\begin{aligned} T(n) &= c_1(n+1) + c_2 n + \frac{1}{6}n(n^2-1) + c_4 \sum_{i=1}^{n}(n-i) \\ &= c_1 n + c_1 + c_2 n + \frac{1}{6}n^3 - \frac{1}{6}n + \frac{c_4}{2}n^2 - \frac{c_4}{2}n \\ &= \frac{1}{6}n^3 + \frac{c_4}{2}n^2 + (c_1 + c_2 - \frac{1}{6} - \frac{c_4}{2})n + c_1 \end{aligned}$$

Since this is cubic, we will let $f(n) := n^3$. It follows from Exercise 5(1) below that $T(n) = O(f(n))$.

(2) It follows from Exercise 5(2) below that $T(n) = \Omega(f(n))$.

(3) For a more efficient algorithm, we could first compute all of "singletons" $A[1], A[2], \ldots, A[n]$ separately. Then we can compute all the "doubles" $A[1] + A[2].A[2] + A[3], \ldots$ by adding one appropriate entry to each singleton. Then we can compute the "triples" $A[1] + A[2] + A[3], \ldots$ by adding an appropriate entry to each double, etc. Proceeding in this manner, each subarray will take only one additional addition. Here is the pseudocode:

```
1  for length = 2 to n
2       // Initialize all of the "doubles"
3       if length == 2
4            for i = 1 to n − 1
5                 B[i, i + 1] = A[i] + A[i + 1]
6       // Compute the sums of length length from the sums of length length − 1
7       if length > 2
8            for i = 1 to n − length + 1
9                 B[i, i + length − 1] = B[i, i + length − 2] + A[i + length − 1]
```

This algorithm runs in $O(n^2)$ time, so with $g(n) := n^2$ we get an asymptotically better running time. □

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*

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

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?

Proof. (1) We also need to show that the elements of $A'[1\,..\,n]$ consist of the original elements of $A[1\,..\,n]$ (except now in sorted order, which is what (†) would assure us).

(2) We will prove the following about the **for** loop in lines 2-4. Here we are assuming that line 1 was just run and the current value of i is fixed and satisfies $1 \leq i \leq A.\mathit{length} - 1$.

Claim. *Immediately after each time line 2 is run, the following are true:*

(a) The entries $A[1\,..\,i-1]$ are exactly the same as they were immediately after line 2 was run the first time with $j = A.\mathit{length}$.

(b) The entries of A are the same as they were immediately after line 2 was run the first time with $j = A.\mathit{length}$, although perhaps in a different order.

(c) The smallest entry of the subarray $A[i\,..\,A.\mathit{length}]$ is contained in the subarray $A[i\,..\,j]$.

Proof. (Initialization) We have to show both (a), (b), (c) are true. Assume we have just run line 2 for the first time, so $j = A.\mathit{length}$. Then (a), (b) and (c) are automatically true.

(Maintenance) Suppose we have just run line 2 and the current value of j is $j = j_0$ where $i + 1 \leq j_0 \leq A.\mathit{length}$. Furthermore, suppose we know that the loop invariant is true for this value of j. Next we run line 3, possibly line 4, and then return to line 2 to decrease j. Since line 4 only exchanges $A[j]$ with $A[j-1]$ and $j - 1 \geq i - 1$, none of the entries of $A[1\,..\,i-1]$ were modified, so (a) remains true. Furthermore, either line 4 is not run, in which case this iteration does not change A, or line 4 is run, in which case $A[j]$ and $A[j-1]$ are interchanged. In either case, since (b) was true at the beginning of this iteration, it remains true at the end of this iteration after the next time line 2 is run. Finally, since we know (c) is true, this means that the smallest entry of $A[i\,..\,A.\mathit{length}]$ is in the subarray $A[i\,..\,j]$. We have two cases:

(Case 1) Suppose the smallest entry of $A[i\,..\,A.\mathit{length}]$ is $A[j_0 - 1]$. Then in line 3 the condition "$A[j] < A[j-1]$" is false, so we go to line 2, j becomes $j = j_0 - 1$, and the smallest entry remains $A[i\,..\,j]$, so (c) remains true.

(Case 2) Suppose the smallest entry is $A[j]$. Then the condition "$A[j] < A[j-1]$" is true, so we run line 4 and interchange $A[j]$ with $A[j-1]$. Now the smallest entry is in the subarray $A[i\,..\,j_0 - 1]$. Then we go to line 2, decrease j to $j = j_0 - 1$, and then (c) is still true.

(Case 3) Suppose the smallest entry is in the subarray $A[i\,..\,j_0 - 2]$. Then it is not modified by line 3 or line 4. We then go to line 2, decrease j to $j = j_0 - 1$, and (c) remains true.

(Termination) After the last time line 2 is run, we have $j = i$. Thus we conclude that:

(a) The entries and positions of $A[1\,..\,i-1]$ are exactly the same as they were before line 2 was run the first time (for this specific value of i).

(b) The entries of A are the same as they were before line 2 was run the first time, except perhaps in a different order.

(c) The smallest entry of $A[i\,..\,A.\mathit{length}]$ is contained in the subarray $A[i\,..\,i]$, i.e., the smallest entry is at position $A[i]$.

□

(3) Using the loop invariant we just proved for lines 2-4, we will prove the following for the **for** loop of lines 1-4:

Claim. *Each time line 1 has just run, the following are true:*

(a) The current entries of A consist of the original entries of A, perhaps in a different order.

(b) The entries of the subarray $A[1\,..\,i-1]$ are the $i-1$ smallest entries of A in sorted order.

Proof. (Initialization) Suppose we have just run line 1 for the first time, so $i = 1$. Then A is still exactly the same as the original array A, so (a) is true. Furthermore, the subarray $A[1\,..\,i-1] =$

$A[1\,..\,0]$ is the empty array, and it consists of the $i-1=0$ smallest entries of the original array A in sorted order. Thus (b) is true as well.

(Maintenance) Suppose we have just run line 1 and the current value of i is $i=i_0$ where $1 \le i_0 \le A.\mathit{length}-1$. Furthermore, suppose we know the loop invariant is currently true. Next we run the **for** loop of lines 2-4. Since A consists of the original entries of A (perhaps in different order), and part (b) of the above loop invariant tells us that the **for** loop preserves the set of elements in the array, then when we return to line 1, we know that the entries of A still consist of the original entries of A (perhaps in a different sorted order). Thus (a) is true.

Next, suppose we know (b) is true. Thus the first $i-1$ entries $A[1\,..\,i-1]$ are the smallest $i-1$ entries of the original A in sorted order. Condition (a) of the previous loop invariant tells us that this property is preserved. Condition (c) tells us that the smallest entry of the remaining subarray $A[i\,..\,A.\mathit{length}]$ ends up in position $A[i]$. Thus the subarray $A[1\,..\,i_0]$ consists of the smallest i_0 entries of the original array A, in sorted order. Next we go back to line 1 and increase i to $i=i_0+1$. Thus it is still true that $A[1\,..\,i-1]$ consists of the smallest $i-1$ entries of the original array A in sorted order. This proves (b).

(Termination) Suppose we have just run line 1 for the last time. Then $i=A.\mathit{length}$. In particular, since the loop invariant is true we know the following things:

(a) The entries of A consist of the original entries of A, except perhaps in different order.
(b) The entries of $A[1\,..\,A.\mathit{length}-1]$ consist of the $A.\mathit{length}-1$ smallest entries of A in sorted order. In particular, the largest entry of A must be in position $A[A.\mathit{length}]$. Thus all of A is in sorted order.

□

(4) Suppose $n=A.\mathit{length}$. The worst-case running time of bubble-sort is determined by the following analysis (since it is the worst-case, we assume line 4 is run the maximum number of times, which would correspond to the input being in reverse-sorted order):

BUBBLESORT(A)

1	**for** $i=1$ **to** $A.\mathit{length}-1$	*cost* : c_1 *times* : n
2	**for** $j=A.\mathit{length}$ **downto** $i+1$	*cost* : c_2 *times* : $n-1$
3	**if** $A[j]<A[j-1]$	*cost* : c_3 *times* : $\sum_{i=1}^{n-1} n-i$
4	exchange $A[j]$ with $A[j-1]$	*cost* : c_4 *times* : $\sum_{i=1}^{n-1} n-i$

The worst-case running time is

$$T(n) \;=\; c_1 n + c_2(n-1) + c_3\sum_{i=1}^{n-1}(n-i) + c_4\sum_{i=1}^{n-1}(n-i) \;=\; an^2+bn+c \;=\; \Theta(n^2)$$

for appropriate constants $a>0, b, c$. This is the same as the worst-case running time of insertion sort. □

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 $\{1,2,\ldots,n\}$ 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 inversions of $\langle 2, 3, 8, 6, 1\rangle$ correspond to the index pairs

$$(1,5), (2,5), (3,4), (3,5), (4,5)$$

so there are five total inversions.

(2) The array $\langle n, n-1, \ldots, 1\rangle$ has the most inversions, since every pair of indices $i < j$ is an inversion. The total number of inversions is the total number of such pairs, which is

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

(3) Suppose $A[1\,..\,n]$ is an input array. Define Inv to be the total number of inversions in A, and for each $j \in \{2, \ldots, n\}$ define

$$\mathrm{Inv}(j) \;:=\; \#\{i \le j-1 : A[i] > A[j]\}$$

i.e., $\mathrm{Inv}(j)$ is the number of inversions which has j as the second index. Thus $\mathrm{Inv} = \sum_{j=2}^{n} \mathrm{Inv}(j)$. In Insertion-Sort, the role of the inner **while** loop is to fix the inversions of j. Thus for a given j, lines 6 and 7 run $\mathrm{Inv}(j)$ many times. Thus the running time analysis now becomes:

Insertion-Sort(A)

1	**for** $j = 2$ **to** $A.\mathit{length}$	*cost* : c_1 *times* : n
2	$\mathit{key} = A[j]$	*cost* : c_2 *times* : $n-1$
3	// Insert $A[j]$...	
4	$i = j - 1$	*cost* : c_3 *times* : $n-1$
5	**while** $i > 0$ and $A[i] > \mathit{key}$	*cost* : c_4 *times* : $\sum_{j=2}^{n} \mathrm{Inv}(j) + 1$
6	$A[i+1] = A[i]$	*cost* : c_5 *times* : $\sum_{j=2}^{n} \mathrm{Inv}(j)$
7	$i = i - 1$	*cost* : c_6 *times* : $\sum_{j=2}^{n} \mathrm{Inv}(j)$
8	$A[i+1] = \mathit{key}$	*cost* : c_7 *times* : $n-1$

We compute the running time as:

$$\begin{aligned} T(n) \;&=\; c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} (\mathrm{Inv}(j)+1) + c_5 \sum_{j=2}^{n} \mathrm{Inv}(j) + c_6 \sum_{j=2}^{n} \mathrm{Inv}(j) + c_7(n-1) \\ &=\; (c_1 + c_2 + c_3 + c_7)n - (c_2 + c_3 + c_7) + c_4\,\mathrm{Inv} + c_4(n-1) + c_5\,\mathrm{Inv} + c_6\,\mathrm{Inv} \\ &=\; (c_4 + c_5 + c_6)\,\mathrm{Inv} + (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_1 + c_2 + c_3 + c_4 + c_7) \\ &=\; a\,\mathrm{Inv} + bn + c \end{aligned}$$

For appropriate constants a, b, c. Thus the running time is $\Theta(\max(n, \mathrm{Inv}))$.

(4) First we write a subroutine similar to the Merge subroutine which takes an array $A[1\,..\,n]$, three index parameters $p \le q < r$, and then returns the number of inversions in $A[p\,..\,r]$, under the assumption that $A[1\,..\,q]$ and $A[q+1\,..\,r]$ are both sorted. In this scenario, the only inversions will be of the form (i, j) where $p \le i \le q$ and $q+1 \le j \le r$, where the first index is in the first half of the subarray, and the second index is in the second half of the subarray.

```
INVERSIONS-IN-HALVES(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  i = p
 4  j = q + 1
 5  counter = 0
 6  for k = p to r
 7       if A[i] ≤ A[j]
 8            i = i + 1
 9            if i > q
10                 Break // leave loop
11       else // now it must be the case that A[i] > A[j]
12            counter = counter + q − i + 1
13            // The entire subarray A[i..q] must form an inversion with A[j]
14            j = j + 1
15            if j > r
16                 Break // leave loop
17  return counter
```

Just like MERGE, the algorithm INVERSIONS-IN-HALVES runs in $\Theta(n)$ time, where $n := r-p+1$. Next we have our divide-and-conquer inversion counting algorithm. Since needed the subarrays to be sorted for INVERSIONS-IN-HALVES, we will need to perform a merge sort at the same time as counting inversions.

```
INVERSIONS-AND-SORT(A, p, r)
 1  if p == r
 2       return 0
 3  if p < r
 4       inversions = 0
 5       q = ⌊(p + r)/2⌋
 6       inversions = inversions + INVERSIONS-AND-SORT(A, p, q)
 7       inversions = inversions + INVERSION-AND-SORT(A, q + 1, r)
 8       inversions = inversions + INVERSIONS-IN-HALVES(A, p, q, r)
 9       MERGE(A, p, q, r)
10       return inversions
```

Just like MERGE-SORT, the algorithm INVERSIONS-AND-SORT satisfies a recurrence $T(n) = 2T(n/2)+\Theta(n)$, so the running time is $T(n) = \Theta(n \lg n)$. □

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.

Proof. (1) Suppose d is a common divisor of $a/2$ and $b/2$, i.e., $d|(a/2)$ and $d|(b/2)$. Thus there are q_0, q_1 such that $dq_0 = a/2$ and $dq_1 = b/2$. Multiplying each of these by 2 yields $2dq_0 = a$ and

$2dq_0 = b$. Thus $2d|a$ and $2d|b$. This shows that $2 \cdot \gcd(a/2, b/2) \leq \gcd(a, b)$. Now suppose $d|a$ and $d|b$. Since both a and b are even, there are a' and b' such that $a = 2a'$ and $b = 2b'$. Thus $d|2a'$ and $d|2b'$. Since 2 is prime, either $d|2$, or $d|a'$ and $d|b'$. In this first case, if $d|2$, the automatically $d \leq 2 \cdot \gcd(a/2, b/2)$. Now suppose $d \nmid 2$. Then $d|a'$ and $d|b'$. Thus $d|a/2$ and $d|b/2$, and so d is a common divisor of both $a/2$ and $b/2$. Thus $d \leq \gcd(a/2, b/2) \leq 2 \cdot \gcd(a/2, b/2)$. Thus $\gcd(a, b) \leq 2\gcd(a/2, b/2)$.

(2) Since every divisor of $b/2$ is also a divisor of b, we have $\gcd(a, b) \geq \gcd(a, b/2)$. Now assume that $d|a$ and $d|b$. Since a is odd, we have that d is odd. Since $d|b = 2(b/2)$, and 2 is prime, we must have $d|b/2$. Thus d is a common divisor of a and $b/2$. Thus $\gcd(a, b) \leq \gcd(a, b/2)$.

(3) First suppose $d|(a-b)/2$ and $d|b$. Then $d|a-b$, so $d|(a-b)+b = a$, and so d is a common divisor of a and b. Thus $\gcd((a-b)/2, b) \leq \gcd(a, b)$. Now suppose $d|a$ and $d|b$. Since a and b are odd, d is odd. Thus $d|((a-b)/2)2$. Since 2 is prime, we must have $d|(a-b)/2$, so d is a common divisor of $(a-b)/2$ and b. Thus $\gcd(a, b) \leq \gcd((a-b)/2, b)$.

(4) One such implementation is as follows:

```
GCD(a, b)
 1  if b == 1
 2        return 1
 3  if b == 0 and a ≠ 0
 4        return |a|
 5  if b is even and a is even
 6        return 2 · GCD(a/2, b/2)
 7  if b is odd and a is even
 8        return GCD(a/2, b)
 9  if b is even and a is odd
10        return GCD(a, b/2)
11  if b is odd and a is odd
12        return GCD((a−b)/2, b)
```

□

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 $p(n) = \omega(n^k)$.

Proof. (1) Note that

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

since for $i < k$ we have $\lim_{n\to\infty} n^{i-k} = 0$. Now, since $d \le k$, either $d = k$ in which case the limit is $\lim_{n\to\infty} a_d = a_d$ or $d < k$, which case $\lim_{n\to\infty} a_d n^{d-k} = 0$. In either case, the limit is a real number, so $p(n) = O(n^k)$.

(2) Consider the limit

$$\begin{aligned}\lim_{n\to\infty}\frac{n^k}{p(n)} &= \lim_{n\to\infty}\frac{n^k}{\sum_{i=0}^{d}a_i n^i} \\ &= \lim_{n\to\infty}\frac{n^k}{n^d\sum_{i=0}^{d}a_i n^{i-d}} \\ &= \lim_{n\to\infty}\frac{n^k}{n^d}\cdot\frac{1}{a_d+\sum_{i=1}^{d}a_i n^{i-d}} \\ &= \lim_{n\to\infty}n^{k-d}\cdot\frac{1}{a_d+\sum_{i=1}^{d}a_i n^{i-d}} \\ &= \frac{1}{a_d}\lim_{n\to\infty}n^{k-d}\end{aligned}$$

Since $k - d \le 0$, the limit of n^{k-d} as $n \to \infty$ is either 1 if $k = d$, or 0 if $k < d$. In either case, the limit is a real number, so $p(n) = \Omega(n^k)$.

(3) Follows from (1) and (2).

(4) Similar to (1), except we must have $d < k$, so the only possibility for the limit is 0, i.e., $p(n) = o(n^k)$.

(5) Similar to (2), except we must have $d > k$, so the only possibility for the limit is 0, i.e., $p(n) = \omega(n^k)$. □

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* $\lg(g(n)) \ge 1$ *and* $f(n) \ge 1$ *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) $f(n) + o(f(n)) = \Theta(f(n))$.

Proof. (1) False. We can find and use a counterexample: Let $f(n) = n$ and $g(n) = n^2$. Then is it true that $f(n) = O(g(n))$ since $n = O(n^2)$ (since we can find a constant $c = 1$, for example, to show $n \le cn^2 \quad \forall n \ge n_0$ with $n_0 \ge 1$). However, $g(n) = O(f(n))$ is false since

there exists no constants c, n_0 such that $n^2 \leq c\dot{n} \quad \forall n \geq n_0$. (Note that the statement would be true if replacing the second notation with a Ω or replacing both notations with Θ) $\square$

(2) False. Let us use a counterexample: $f(n) = n$ and $g(n) = n^3$. The left hand side of the equation would give us: $f(n)+g(n) = n+n^3$ while the right hand side is $\Theta(\min(n, n^3)) = \Theta(n)$. $f(n)+g(n) = \Theta(\min(f(n), g(n)))$ means, by definition, that $f(n)+g(n) = O(\min(f(n), g(n)))\cap$ $f(n) + g(n) = \Omega(\min(f(n), g(n)))$. Evaluating the first condition for theta, we see that $n^3 + n = O(n)$ is false since there exists no constants c, n_0 such that it satisfies the condition that $n^3 \leq c\dot{n} \quad \forall n \geq n_0$.

(3) True. Since $f(n) = O(g(n))$, let us call c the constant for which

$$f(n) \leq c\dot{g}(n) \quad \forall n \geq n_0$$

Taking the lg of both sides preserves the inequality since lg is a nonnegative, strictly increasing function for inputs greater or equal to 1. Hence. we get:

$$\begin{aligned}\lg(f(n)) &\leq \lg(c\dot{g}(n)) \\ &\leq \lg(c) + \lg(g(n)) \leq m * \lg(g(n)) \quad (m := 1 + \max(0, \lg c) \in \mathbb{R}^+) \\ \lg(f(n)) &\leq m * \lg g(n) \implies \lg f(n) = O(\lg(g(n))).\end{aligned}$$

(4) False. Using the counterexample: let $f(n) = n^2 + 3n$ and $g(n) = 0.5n^2$. ($f(n) = O(g(n))$ since for $c = 2$ and $n_0 = 1$, $f(n) \leq c * g(n)n \quad \forall n \geq n_0$.) We know that $2^{f(n)} = 2^{n^2} * 2^{3n}$ while $2^{g(n))} = 2^{0.5n^2}$. Taking the limit of the ratio gives us:

$$\lim_{n\to+\infty} \frac{2^{g(n)}}{2^{f(n)}} = \lim_{n\to+\infty} \frac{2^{0.5n^2}}{2^{n^2} * 2^{3n}} = \lim_{n\to+\infty} \frac{1}{2^{0.5n^2+3n}} = 0$$

Since the limit of the ration is zero, $2^{g(n)} = o(2^{f(n)})$. If we had $2^{f(n)} = O(2^{g(n)})$, this would imply that $2^{g(n)} = o(2^{g(n)})$, a contradiction. Hence $2^{f(n)} \neq O(2^{g(n)})$.

(5) False. We give the counterexample $f(n) = \frac{1}{n}$. Then there are no constants $c > 0$ and n_0 such that $\frac{1}{n} = f(n) \leq c(f(n))^2 = \frac{c}{n^2} \quad \forall n \geq n_0$, so $f(n) \neq O\big((f(n))^2\big)$.

(6) True. Since $f(n) = O(g(n))$, we can find constants $c > 0$ and n_0 for which

$$0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0.$$

Then

$$0 \leq \frac{1}{c} f(n) \leq g(n) \quad \forall n \geq n_0,$$

so $f(n) = \Omega(g(n))$.

(7) False. We use the counterexample $f(n) = 2^n$. For any $c > 0$ and n_0, if $n > \max(n_0, 2\lg c)$, then $n \geq n_0$ and $f(n) = 2^n > 2^{n/2+\lg c} = cf(n/2)$. There are therefore no constants $c > 0$, n_0 such that $f(n) \leq cf(n/2) \quad \forall n \geq n_0$, and hence $f(n) \neq O(f(n/2))$. Since $f(n) = \Theta(f(n/2))$ would imply $f(n) = O(f(n/2))$, $f(n) \neq \Theta(f(n/2))$.

(8) True. Let $g(n)$ be any function such that $g(n) = o(f(n))$. Then there is an n_0 such that $0 \leq g(n) \leq 1 \cdot f(n) \quad \forall n \geq n_0$. Then $1 \cdot f(n) \leq f(n) + g(n) \leq 2f(n) \quad \forall n \geq n_0$, so $f(n) + g(n) = \Theta(f(n))$. Since $g(n) = o(f(n))$ was arbitrary, $f(n) + o(f(n)) = \Theta(f(n))$.

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. First, let us calculate the maximum number of operations we can perform in an hour: 10^{10}(operations per second) $*3600$ (seconds per hour) $= 3.6 * 10^{13}$ operations per hour. Setting each running time to be equal to the total number of operations per hour lets us find the largest input size for the algorithms (while keeping in mind that input sizes are discrete):

(1) $n = 6 * 10^6$
(2) $n = 33019$
(3) $n = 6 * 10^5$
(4) $n = 9.063 * 10^{11}$
(5) $n = 45$
(6) $n = 5$

□

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 following list is in ascending order of growth rate:

(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) $g_6(n) = 2^{2^n}$

Suppose $n \geq 4$. Then $\lg n \geq 2$, so $\sqrt{\lg n} \leq \lg n \leq \lg n + 3 \lg \lg n$. We also have

$$\frac{4}{3} \lg n \;\leq\; (\lg n)^2 \;\leq\; n \;\leq\; n^2 \;\leq\; 2^n.$$

In other words,

$$\lg g_1(n) \;\leq\; \lg g_4(n) \quad \text{and} \quad \lg g_3(n) \;\leq\; \lg g_5(n) \;\leq\; \lg g_2(n) \;\leq\; \lg g_7(n) \;\leq\; \lg g_6(n).$$

Since exponentiation is an increasing function, this proves all of the consecutive comparisons except $g_4(n) = O(g_3(n))$.

To prove $g_4(n) = O(g_3(n))$, note that by HW1 Exercise 18, $(\ln n)^9 = o(n)$. Taking cube roots (which we may do by Proposition 4 in the discussion 2 worksheet solutions), and multiplying by the constant $(\ln 2)^{-3}$, we obtain $(\lg n)^3 = o(n^{1/3})$. Multiplying by n (Proposition 3 of the same) yields $g_4(n) = o(g_3(n))$. □

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) The number $12\ldots d$ is $d \cdot 1 + (d-1) \cdot 10 + \cdots + 1 \cdot 10^{d-1} = \sum_{j=0}^{d-1}(d-j)10^j$, while $11\ldots1$ is $\sum_{j=0}^{d} 10^j$, so Dr. Matrix's formula is

$$9 \times \sum_{j=0}^{d-1}(d-j)10^j + d + 1 \;=\; \sum_{j=0}^{d} 10^j.$$

(2) The number 10 in the formula should be replaced by b to represent base b numerals. The number 9 occurs because $9 = 10 - 1$, so it should be replaced by $b - 1$. Making these substitutions yields the formula

$$(b-1)\sum_{j=0}^{d-1}(d-j)b^j + d + 1 \;=\; \sum_{j=0}^{d} b^j.$$

(3) Using the geometric series formula and the formula from Exercise 3 of HW1, we have

$$\begin{aligned}
(b-1)\sum_{j=0}^{d-1}(d-j)b^j + d + 1 \;&=\; (b-1)d\sum_{j=0}^{d-1} b^j - (b-1)\sum_{j=0}^{d-1} jb^j + d + 1 \\
&=\; (b-1)d\frac{b^d - 1}{b-1} - (b-1)\frac{(d-1)b^{d+1} - db^d + b}{(b-1)^2} + d + 1 \\
&=\; \frac{1}{b-1}\Big((b-1)d(b^d-1) - \big((d-1)b^{d+1} - db^d + b\big) + (d+1)(b-1)\Big) \\
&=\; \frac{1}{b-1}\big((d-(d-1))b^{d+1} - (d-d)b^d - (b-1)d - b + (d+1)(b-1)\big) \\
&=\; \frac{b^{d+1}-1}{b-1} \\
&=\; \sum_{j=0}^{d} b^j.
\end{aligned}$$

□

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.

Proof. (1) We will prove this by induction on $n \geq 0$. It is easily verified that it is true for the base cases $n = 0, 1$. Now assume we know this is true up to some $n \geq 1$. Note that

$$\begin{aligned} F_{n+1} \;=\; F_n + F_{n-1} \;&=\; \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}} + \frac{\phi^{n-1} - \hat{\phi}^{n-1}}{\sqrt{5}} \\ &= \frac{\phi^n + \phi^{n-1} - \hat{\phi}^n - \hat{\phi}^{n-1}}{\sqrt{5}} \\ &= \frac{\phi^{n-1}(\phi+1) - \hat{\phi}^{n-1}(\hat{\phi}+1)}{\sqrt{5}} \\ &= \frac{\phi^{n-1}\phi^2 - \hat{\phi}^{n-1}\hat{\phi}^2}{\sqrt{2}} \\ &= \frac{\phi^{n+1} - \hat{\phi}^{n+1}}{\sqrt{5}} \end{aligned}$$

using that both ϕ and $\hat{\phi}$ satisfy $x + 1 = x^2$ in the fourth line.

(2) We construct the following cost table:

FIBONACCI(n)

1	**if** $n == 0$ or $n == 1$	cost: c_1	times: 1
2	**return** n	cost: c_2	times: 0 or 1
3	**else**	cost: c_3	times: 1 or 0
4	**return** FIBONACCI($n-1$) + FIBONACCI($n-2$)	cost: $T(n-1) + T(n-2) + c_4$	times: 1 or 0

This shows that $T(n) = c_1 + c_2$ if $n \leq 1$, and $T(n) = T(n-1) + T(n-2) + c_1 + c_3 + c_4$ if $n \geq 2$. Let $a := c_1 + c_2$, and $b := c_1 + c_3 + c_4$. We will show by induction that $T(n) = (a+b)F_{n+1} - b$ for all n This is true for $n = 0$ and $n = 1$, so suppose that it is true up to some $n \geq 1$. Then

$$\begin{aligned} T(n+1) &= T(n) + T(n-1) + b \\ &= (a+b)F_{n+1} - b + (a+b)F_n - b + b \\ &= (a+b)F_{n+2} - b. \end{aligned}$$

Since $F_m \geq 1$ for all $m \geq 1$, we have that $F_m + 1 \leq F_{m+1} \leq 2F_m$ for all $m \geq 2$. In particular, $(a+b)F_n + a \leq T(n) \leq 2(a+b)F_n - b$ for $n \geq 2$. Since $a, b > 0$, we have $T(n) = \Theta(F_n)$.

(3) Since $|\hat{\phi}| = 0.618\ldots < 1$ and $\phi > 1$, we have that $\phi^n = \omega(1)$ and $\hat{\phi}^n = o(1)$. Thus

$$F_n = \frac{1}{\sqrt{5}}(\phi^n + o(1)) = \Theta(\phi^n) + o(\phi^n) = \Theta(\phi^n).$$

Since $T(n) = \Theta(F_n)$, we have $T(n) = \Theta(\phi^n)$, so FIBONACCI 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. (Some of this might be overkill for this particular problem, but otherwise might be useful for other problems involving number of divisors.) First we have some subroutines which grow and maintain a list of prime numbers

```
#this should always contain consecutive primes
primes = [2,3,5]

def next_prime():
    #This will grow primes by one more prime

    value = primes[-1]
    orig_length = len(primes)

    while len(primes) == orig_length:
        value += 1
        if is_prime(value):
            primes.append(value)

def is_prime(N):
    #assumes that primes is big enough to detect primality of N
    for p in primes:
        if N%p == 0 :
            return False
    return True
```

Next we have a function which determines the number of divisors of a number N directly:

```
#this function will compute the number of divisors of N directly
def divisor(N):
    #first make sure we have enough primes
    while primes[-1]<N:
        next_prime()

    prime_factorization = {}
    #Next we grow a prime factorization dictionary
    while N>1:
        for p in primes:
            if p<=N:
                exp=0
                while N%p==0:
                    N/=p
```

```
                    exp+=1
                if exp>=1:
                    prime_factorization[p]=exp

    divisors=1
    #Then we determine the number of divisors from the prime factorization
    for exp in prime_factorization.values():
        divisors*=(exp+1)

    return divisors
```

Next we have a function which grows and maintains a list of the number of divisors of n:

```
divisors = [1]
#divisors[n] will be the number of divisors of n
def dynamic_divisor(N):
    #first check if our divisor list is long enough
    if len(divisors)>N:
        return divisors[N]
    #otherwise grow the divisor list using the divisor function
    while len(divisors)<=N:
        divisors.append(divisor(len(divisors)))

    return divisors[-1]
```

Next we have a function which returns the number of divisors of the Nth triangular number. This algorithm is based on the fact that if $\gcd(m,n)=1$ (i.e., if m and n are relatively prime), then the number of divisors of mn is equal to the number of divisors of m times the number of divisors of n.

```
def triangle_divisor(N):
    if N%2==0:
        return dynamic_divisor(int(N/2))*dynamic_divisor(N+1)
    if N%2==1:
        return dynamic_divisor(N)*dynamic_divisor(int((N+1)/2))
```

Finally, we run the following script:

```
N=1
while triangle_divisor(N)<=1000:
    N+=1

print(int(N*(N+1)/2))
```

which returns

842161320

□

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

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

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. This program stores previously computed path lengths in a python dictionary, so we don't have to keep recomputing path lengths.

```
collatz_length={1:1}
#This dictionary will keep track of all the path lengths we know so far

def succ(N):
    #This function returns the next element in a path after N
    if N%2==0:
        return int(N/2)
    if N%2==1:
        return 3*N+1

def collatz(N):
    #First check if we already know the path length of N
    if N in collatz_length:
        return collatz_length[N]
    else:
        #Otherwise assign the path length to be 1 + the path length of its
    successor (this is a recursive call)
        collatz_length[N] = 1+collatz(succ(N))
        return collatz_length[N]
```

Now we run the following script:

```
longest=1
for n in range(1,2000000):
    if collatz(n)>collatz(longest):
        longest=n

print(longest)
```

which 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?*

Solution. This problem actually has a solution which can be computed by hand. In order to go from the top left to bottom right, we need to go down 30 times and to the right 30 times, in some order. So there are $60 = 30 + 30$ total steps in this path, 30 of these will be down and 30 of these will be to the right. To uniquely determine a path, we need to specify which 30 of the 60 steps are down. This number is

$$\binom{60}{30} = \frac{60!}{30!30!} = 118264581564861424$$ □

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:*

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

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.

Proof. This program involves making an array which keeps track of all possible ways of making a certain value using a given initial segment of the coin values.

```
coin_vals = [1,2,5,10,20,50,100,200]
#the denominations of the coins we are allowed to use
coin_sums = [[1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1]]
#coin_sum[n][j] will be #of ways to make the sum n using coins from coin_vals[0..j]. There is only one way to make n=0 and n=1 using any number of coins provided you are always allowed the pence coin
#ultimately we are interested in coin_sum[n][-1], where we allow all of the coin denominations to make up n

def next_coin_sum():
    #this will grow the array coin_sum to include the next value
    N=len(coin_sums)
    #N-1 is largest number we've analyzed so far, now we'll analyze N
    coin_sums.append([1])
    #Using only the pence coin, there is only 1 way to make N (with N pence's)
    for i in range(1,len(coin_vals)):
        #This will determine number of ways to make N allowing coins from coin_vals[0..i]
        ways=0
        j=0
        #This while loop counts all the valid ways we can use j coins of coin_vals[i] to make N
        while N-j*coin_vals[i]>=0:
            ways+=coin_sums[N-j*coin_vals[i]][i-1]
            j+=1
        coin_sums[-1].append(ways)

def coin_sum(N):
    #First make sure coin_sums is long enough
    while len(coin_sums)<=N:
        next_coin_sum()

    return coin_sums[N][-1]
```

When we run `coin_sum(200)` this returns

73682

□

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

$$0.12345678910111213141516171819 20\ldots$$

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

If d_n *represents the* n*th 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 challenge with this problem is that it requires keeping track of and doing things with indices. For this we have written first several subroutines:

```
def start(digit_number):
    #returns the index where the block of numbers with digit_number of digits begins
    #for instance, start(1) is 1, start(2) is 10 and start(3) is 190
    index = 1
    for n in range(1,digit_number):
        index += (10**n-10**(n-1))*n

    return index

def find_block(n):
    #given the index n, this determines if d_n is a digit in the 1-digit block, the 2-digit block, the 3-digit block, etc.
    digits = 1
    while start(digits) <= n:
        digits+=1
    return digits-1

def rel_index(n):
    # given the index n, this determines the relative index of n inside of its block. For instance, d_{10} has index 0 inside of the 2-digit block, d_{12} has index 3 inside of the 2-digit block, etc.
    block = find_block(n)
    return n-start(block)

def digit_value(n):
    #Given index n, this will return d_n
    digits = find_block(n)
    n = rel_index(n)
    q = n//digits
    #q is the how many numbers from the start of the block is n a part of. For instance, 100 is the first number in the 3-digit block, so if n is part of 100, then q=0. if n is part of 101, then q=1
    r = n%digits
    #r determines if n is referring to the ones digit, the tens digit, etc. of its number.
    start_of_block = 10**(digits-1)
    number = start_of_block+q
    #number is the actual number that n is a part of
    return int(str(number)[r])
```

Here is the main program:

```
prod = 1
for n in range(0,7):
    prod*=digit_value(10**n)
print(prod)
```

When run this outputs:

```
210
```

□