MATH182 HOMEWORK #3
DUE July 12, 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 seven (7) of these problems. All problems will be graded, although the score for the homework will be capped at $N :=$ (point value of one problem) $\times 7$ 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. *Obtain asymptotically tight bounds on* $\lg(n!)$ *without using Stirling's approximation. Instead, find a way to approximate the summation* $\sum_{k=1}^{n} \lg k$ *from above and below.*

Solution. First we will convert $\lg(n!)$ into a definite integral involving floor functions:

$$\begin{aligned}
\lg(n!) &= \sum_{k=1}^{n} \lg k \\
&= \sum_{k=1}^{n} \int_{k}^{k+1} \lg \lfloor x \rfloor \; dx \\
&= \int_{1}^{n+1} \lg \lfloor x \rfloor \; dx.
\end{aligned}$$

First note that since $\lfloor x \rfloor \leq x$, it follows for $x \geq 1$ that $\lg \lfloor x \rfloor \leq \lg x$. Thus

$$\begin{aligned}
\lg(n!) &\leq \int_{1}^{n+1} \lg x \; dx \\
&= \left[\frac{x(\log x - 1)}{\log 2} \right]_{1}^{n+1} \\
&= \frac{(n+1)(\log(n+1) - 1) + 1}{\log 2} \\
&= \frac{1}{\log 2}\big(n \log(n+1) + n + \log(n+1)\big) \\
&= n \log(n+1)/\log 2 + n/\log 2 + \log(n+1)/\log 2 \\
&= n \lg(n+1) + n/\log 2 + \lg(n+1) \\
&\leq n \lg(2n) + n/\log 2 + \lg(2n) \\
&= n \lg n + n/\log 2 + \lg n + 2 \\
&= O(n \lg n).
\end{aligned}$$

For a lower bound, note that

$$\begin{aligned}
\lg(n!) &= \lg 1 + \sum_{k=2}^{n} \lg k \\
&= \sum_{k=2}^{n} \int_{k}^{k+1} \lg \lfloor x+1 \rfloor \, dx \\
&= \int_{k=2}^{n+1} \lg \lfloor x \rfloor \, dx \\
&\geq \int_{k=2}^{n+1} \lg(x-1) \, dx \quad \text{because } \lfloor x \rfloor \geq x-1 \\
&= \int_{1}^{n} \lg x \, dx \\
&= \left[\frac{x(\log x - 1)}{\log 2} \right]_{1}^{n} \\
&= \frac{1}{\log 2}\bigl(n \log n - n - 1\bigr) \\
&= n \lg n - n/\log 2 - 1/\log 2 \\
&= \Omega(n \lg n).
\end{aligned}$$

We conclude that $\lg(n!) = \Theta(n \lg n)$. □

Exercise 2. *Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of $A[1\,..\,j]$, extend the answer to find a maximum subarray ending at an index $j+1$ by using the following observation: a maximum subarray of $A[1\,..\,j+1]$ is either a maximum subarray of $A[1\,..\,j]$ or a subarray $A[i\,..\,j+1]$, for some $1 \leq i \leq j+1$. Determine a maximum subarray of the form $A[i\,..\,j+1]$ in constant time based on knowing a maximum subarray ending at index j.*

Your answer should consist of:

(1) Pseudocode for your algorithm.
(2) Proof of correctness.
(3) An analysis of the running time.

Solution. (1) The following is the pseudocode for our algorithm.

```
MAX-SUBARRAY($A$)
1   $BestL = 1$
2   $BestR = 1$
3   $BestValue = A[1]$
    // $(BestL, BestR, BestValue)$ will maintain best value seen so far in $A[1\,..\,j]$
4   $FlushLeft = 1$
5   $FlushBest = A[1]$
    // $(FlushLeft, j, FlushBest)$ will maintain best value of the form $A[i\,..\,j]$
6   for $j = 2$ to $A.length$
7       if $FlushBest + A[j] > A[j]$
8           $FlushBest = FlushBest + A[j]$
9       else $FlushBest = A[j]$
10          $FlushLeft = j$
11      if $BestValue < FlushBest$
12          $BestValue = FlushBest$
13          $BestL = FlushLeft$
14          $BestR = j$
15  return $(BestL, BestR, BestValue)$
```

(2) Here is the proof of correctness:

Claim. *When* MAX-SUBARRAY(A) *terminates, we have*

$$BestValue \;=\; \sum_{k=BestL}^{BestR} A[k] \;=\; \max\left\{\sum_{k=i}^{j} A[k] : 1 \le i \le j < A.length + 1\right\}$$

Proof of claim. After lines 1-5, we have

- $BestL = BestR = 1$
- $BestValue = A[1] = \sum_{k=BestL}^{BestR} A[k]$
- $FlushBest = A[1] = \sum_{k=FlushLeft}^{1} A[k]$.

We will prove the following loop invariant for the **for** loop of lines 6-14:

(Loop invariant) Each time line 6 is finished running, we have
(a) $BestValue = \sum_{k=BestL}^{BestR} A[k] = \max\{\sum_{k=l}^{r} A[k] : 1 \le l \le r < j\}$
(b) $FlushValue = \sum_{k=FlushLeft}^{j-1} A[k] = \max\{\sum_{k=l}^{j-1} A[k] : 1 \le l < j\}$

(Initialization) The first time line 6 is finished, the current value of j is $j = 2$. Both (a) and (b) are true in this situation.

(Maintenance) Suppose line 6 has just finished running, the loop invariant is assumed to be true, and the current value of j is $j = j_0$ for some $2 \le j_0 \le A.length$. Next we enter the **if** statement of line 7 and we have two cases:

(Case 1) Suppose $FlushBest + A[j_0] > A[j_0]$. Then we redefine $FlushBest$ to be $FlushBest + A[j_0]$, and so

$$FlushBest \;=\; \sum_{k=FlushLeft}^{j_0-1} A[k] + A[j_0] \;=\; \sum_{k=FlushLeft}^{j_0} A[k].$$

Since $\sum_{k=FlushLeft}^{j_0-1} A[k] = \max\{\sum_{k=l}^{j_0-1} A[k] : 1 \le l < j_0\}$, it follows that

$$\sum_{k=FlushLeft}^{j_0} A[k] \;=\; \max\{\sum_{k=l}^{j_0} A[k] : 1 \le l < j_0\}.$$

Furthermore, since *FlushBest* $> A[j_0]$ at this point, we also have *FlushBest* $> \sum_{k=j_0}^{j_0} A[k]$. Thus

$$\mathit{FlushBest} \;=\; \max\{\sum_{k=l}^{j_0} A[k] : 1 \leq l < j_0\}$$

(Case 2) Suppose *FlushBest* $+ A[j_0] \leq A[j_0]$. Then

$$\sum_{k=\mathit{FlushLeft}}^{j_0-1} A[k] \;=\; \max\{\sum_{k=l}^{j_0-1} A[k] : 1 \leq l < j_0\} + A[j_0] \;\leq\; A[j_0]$$

i.e.,

$$\max\{\sum_{k=l}^{j_0} A[k] : 1 \leq l < j_0\} \;\leq\; \sum_{k=j_0}^{j_0} A[k]$$

and so

$$\sum_{k=j_0}^{j_0} A[k] \;=\; \max\{\sum_{k=l}^{j_0} A[k] : 1 \leq l < j_0 + 1\}.$$

Then we redefine *FlushLeft* to *FlushLeft* $= j_0$ and *FlushBest* $= A[j_0] = \sum_{k=\mathit{FlushLeft}}^{j_0} A[k]$. Thus

$$\mathit{FlushBest} \;=\; \sum_{k=\mathit{FlushLeft}}^{j_0} A[k] \;=\; \max\{\sum_{k=l}^{j_0} A[k] : 1 \leq l < j_0 + 1\}.$$

At this point, since we no longer modify *FlushLeft* or *FlushBest*, we know that (b) is satisfied next time we run line 6 and increment j to $j_0 + 1$.

Now we proceed to line 11. This **if** clause also yields two cases:

(Case 1') Suppose *BestValue* $<$ *FlushBest*. Then

$$\max\{\sum_{k=l}^{r} A[k] : 1 \leq l \leq r < j_0\} \;<\; \max\{\sum_{k=l}^{j_0} A[k] : 1 \leq l < j_0 + 1\}.$$

Thus

$$\mathit{FlushBest} \;=\; \max\{\sum_{k=l}^{r} A[k] : 1 \leq l \leq r < j_0 + 1\}.$$

Then we redefine *BestValue* $=$ *FlushBest*, *BestL* $=$ *FlushLeft* and *BestR* $= j_0$. Now we have

$$\mathit{BestValue} \;=\; \mathit{FlushBest} \;=\; \sum_{k=\mathit{FlushLeft}}^{j_0} A[k] \;=\; \sum_{k=\mathit{BestL}}^{\mathit{BestR}} A[k] \;=\; \max\{\sum_{k=l}^{r} A[k] : 1 \leq l \leq r < j_0 + 1\}.$$

(Case 2') Next suppose *BestValue* $\geq$ *FlushBest*. Then

$$\mathit{BestValue} \;=\; \max\{\sum_{k=l}^{r} A[k] : 1 \leq l \leq r < j_0\} \;\geq\; \max\{\sum_{k=l}^{j_0} A[k] : 1 \leq l < j_0 + 1\}$$

and so

$$\mathit{BestValue} \;=\; \sum_{k=\mathit{BestL}}^{\mathit{BestR}} A[k] \;=\; \max\{\sum_{k=l}^{r} A[k] : 1 \leq l \leq r < j_0 + 1\}.$$

In either case, when we proceed to run line 6 again, part (a) is satisfied when we increment j to $j_0 + 1$.

(Termination) After the last time we run line 6, the loop invariant is true and the value of j is $j = A.\mathit{length} + 1$. Thus by (a) we have

$$\mathit{BestValue} \;=\; \sum_{k=\mathit{BestL}}^{\mathit{BestR}} A[k] \;=\; \max\{\sum_{k=l}^{r} A[k] : 1 \le l \le r < A.\mathit{length} + 1\}.$$

This is what needs to be true in line 15 in order for the claim to be proved. □

(3) We will now analyze the running time, with $n = A.\mathit{length}$:

```
MAX-SUBARRAY(A)
1   BestL = 1                                  cost : $c_1$ times : 1
2   BestR = 1                                  cost : $c_2$ times : 1
3   BestValue = A[1]                           cost : $c_3$ times : 1
4   FlushLeft = 1                              cost : $c_4$ times : 1
5   FlushBest = A[1]                           cost : $c_5$ times : 1
6   for j = 2 to A.length                      cost : $c_6$ times : $n$
7       if FlushBest + A[j] > A[j]             cost : $c_7$ times : $n-1$
8           FlushBest = FlushBest + A[j]       cost : $c_8$ times : $\sum_{j=2}^{n} \delta_j$
9       else FlushBest = A[j]                  cost : $c_9$ times : $\sum_{j=2}^{n} \delta'_j$
10          FlushLeft = j                      cost : $c_{10}$ times : $\sum_{j=2}^{n} \delta'_j$
11      if BestValue < FlushBest               cost : $c_{11}$ times : $n-1$
12          BestValue = FlushBest              cost : $c_{12}$ times : $\sum_{j=2}^{n} \delta''_j$
13          BestL = FlushLeft                  cost : $c_{13}$ times : $\sum_{j=2}^{n} \delta''_j$
14          BestR = j                          cost : $c_{14}$ times : $\sum_{j=2}^{n} \delta'_j$
15  return (BestL, BestR, BestValue)           cost : $c_{15}$ times : 1
```

where each $\delta_j, \delta'_j, \delta''_j \in \{0, 1\}$. The expression for the running time is

$$T(n) \;=\; (c_1 + c_2 + c_3 + c_4 + c_5 - c_7 - c_{11} + c_{15}) + (c_6 + c_7 + c_{11})n$$

$$+ \sum_{j=2}^{n} \left(c_8\delta_j + (c_9 + c_{10})\delta'_j + (c_{12} + c_{13} + c_{14})\delta''_j\right)$$

For an upper bound, we can take every $\delta_j, \delta'_j, \delta''_j = 1$, yielding

$$T(n) \;=\; n\sum_{i=6}^{14} c_i + \sum_{i=1}^{5} c_i - \sum_{i=7}^{14} c_i + c_{15} \;=\; an + b \;=\; O(n)$$

for appropriate constants a and b. For a lower bound, we can take every $\delta_j, \delta'_j, \delta''_j = 0$, yielding

$$T(n) \;=\; (c_6 + c_7 + c_{11})n + \sum_{i=1}^{5} c_i - c_7 - c_{11} + c_15 \;=\; an + b \;=\; \Omega(n)$$

for appropriate constants a and b. We conclude that $T(n) = \Theta(n)$, as desired. □

Exercise 3. *How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithm as a subroutine? Answer the same question with the order of the input matrices reversed.*

Solution. We are assuming we have a subroutine STRASSEN-MULTIPLY(A, B) which runs in $\Theta(n^{\lg 7})$ times, where A and B are both $n \times n$ matrices. Now suppose that A is $kn \times n$ and B is $n \times kn$.

We can view these matrices as block matrices:

$$A \;=\; \begin{bmatrix} A_1 \\ \vdots \\ A_k \end{bmatrix} \quad \text{and} \quad B \;=\; \begin{bmatrix} B_1 & \cdots & B_k \end{bmatrix}$$

where each A_i and B_j is an $n \times n$ matrix. In terms of block-matrix multiplication, the product AB is

$$AB \;=\; \begin{bmatrix} A_1B_1 & \cdots & A_1B_k \\ \vdots & \ddots & \vdots \\ A_kB_1 & \cdots & A_kB_k \end{bmatrix}$$

This requires k^2 multiplications of two $n \times n$ matrices. This can be done in $\Theta(k^2 n^{\lg 7})$ time. For the other way around, the product BA is

$$BA \;=\; \begin{bmatrix} B_1A_1 + \cdots + B_kA_k \end{bmatrix}$$

which requires k multiplications of two $n \times n$ matrices. This can be done in $\Theta(kn^{\lg 7})$ time. □

Exercise 4. *Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three multiplications of real numbers. The algorithm should take a, b, c, d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.*

Proof. Given the input a, b, c, d, compute the following three products (3 multiplications required):

$$\begin{aligned} k_1 &:= a \cdot (c + d) \\ k_2 &:= d \cdot (d + a) \\ k_2 &:= b \cdot (d - c) \end{aligned}$$

Note that

$$k_1 - k_2 \;=\; a(c + d) - d(b + a) \;=\; ac + ad - bd - ad \;=\; ac - bd$$

and

$$k_2 - k_3 \;=\; d(b + a) - b(d - c) \;=\; bd + ad - bd + bc \;=\; ad + bc.$$

Thus we can compute the real and imaginary components $ac - bd$ and $ad + bc$ using only 3 multiplications (and 3 additions and 2 subtractions). □

Exercise 5. *Show that the solution of $T(n) = T(n - 1) + n$ is $O(n^2)$.*

Proof. We will use the substitution method with subtracting two lower order terms. We want to show there are $c > 0$ and $d, e \in \mathbb{R}$ such that for sufficiently large n, $T(n) \leq cn^2 - dn - e$. Assume inductively that $T(m) \leq cm^2 - dm$ for all $m < n$. Then

$$\begin{aligned} T(n) &= T(n - 1) + n \\ &\leq c(n - 1)^2 - d(n - 1) - e + n \\ &= c(n^2 - 2n + 1) - dn + d - e + n \\ &= cn^2 - 2cn + c - dn + d - e + n \\ &= cn^2 - (2c + d - 1)n - (e - c - d) \\ &\leq cn^2 - dn - e \end{aligned}$$

where the final inequality holds if:

(1) $2c + d - 1 \geq d$
(2) $e - c - d \geq e$

First we note that $2c + d - 1 \geq d$ is equivalent to $2c \geq 1$, i.e., $c \geq 1/2$. The second inequality $e - c - d \geq e$ is equivalent to $-c - d \geq 0$, i.e., $-d \geq c$, i.e., $d \leq -c$. Thus, as long as:

(1) $c \geq 1/2$, and
(2) $d \leq -c$

the inductive step will work. Since these inequalities allow us the freedom to make c larger if necessary to make the base cases work, we conclude that $T(n) = O(n^2)$. □

Exercise 6. *Solve the recurrence $T(n) = 3T(\sqrt{n}) + \log n$ by making a change of variables. Your solution should be asymptotically tight. Do not worry about whether values are integers.*

Solution. Let's rename $m = \lg n$. Then the recurrence becomes

$$T(2^m) \;=\; 3T(2^{m/2}) + m \log 2$$

Next we define $S(m) := T(2^m)$. The recurrence in terms of S becomes

$$S(m) \;=\; 3S(m/2) + m \log 2$$

We will solve this using the Master Theorem. $a = 3$, $b = 2$, $f(n) = \Theta(n) = \Omega(n^{\log_3 2+\epsilon})$ for some $\epsilon > 0$ since $\log_3 2 < 1$. Thus we are in case (3). Furthermore, we see that the regularity condition holds for $c = 2/3$. Thus $S(m) = \Theta(m)$. It follows that $T(n) = T(2^m) = S(m) = \Theta(m) = \Theta(\lg n)$. □

Exercise 7. *Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$. Use the substitution method to verify your answer.*

Solution. We draw a recursion tree in Figure 1. The recursion tree suggests that the total work done is

$$\sum_{h=0}^{\lg n-1} \frac{n^2}{4^h} + T(1)$$

We can get an upper bound by completing the summation to an infinite geometric sum:

$$\begin{aligned}\sum_{h=0}^{\lg n-1} \frac{n^2}{4^h} + T(1) \;&\leq\; n^2 \sum_{h=0}^{\infty} (1/4)^h + T(1) \\ &=\; n^2 \frac{1}{1-1/4} + T(1) \\ &=\; \frac{4}{3} n^2 + T(1).\end{aligned}$$

This suggests that $T(n) = O(n^2)$ (and thus also $\Theta(n^2)$). We will verify this with the substitution method. Suppose we know there is some $c > 0$ such that $T(m) \leq cm^2$ for all $m < n$. Note that

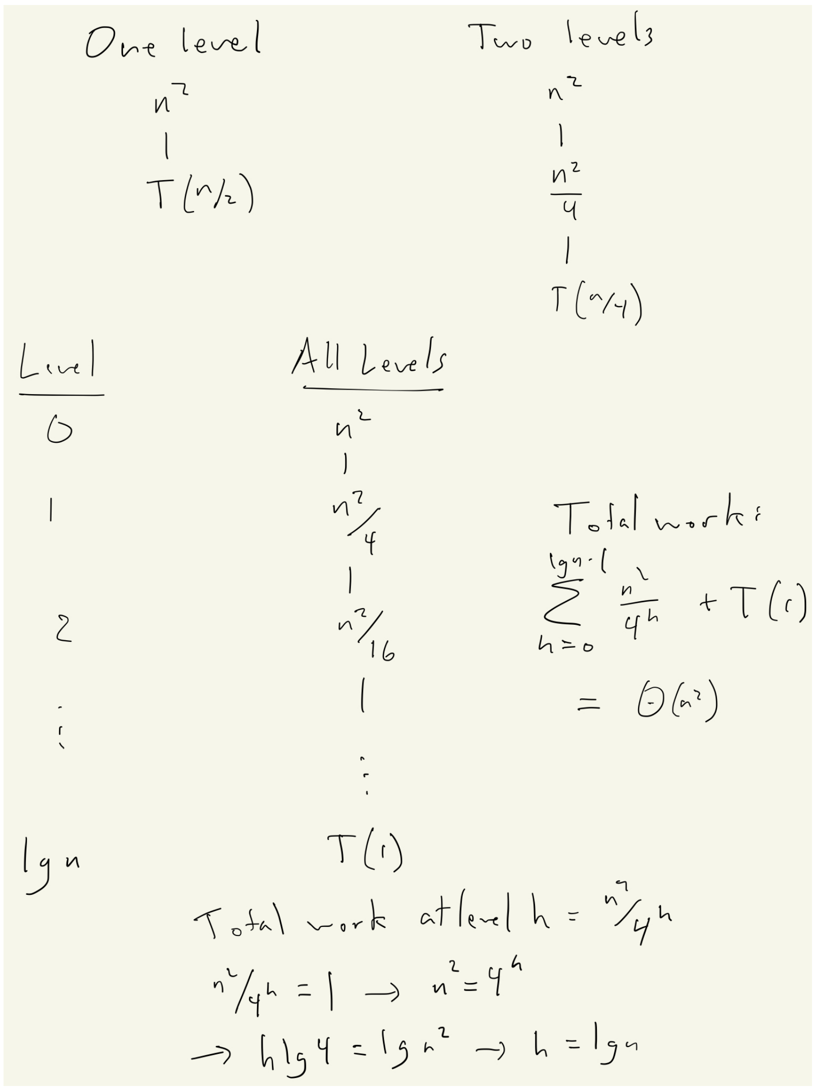
$$\begin{aligned}T(n) \;&=\; T(n/2) + n^2 \\ &\leq\; c(n/2)^2 + n^2 \\ &=\; (c/4 + 1)n^2 \\ &\leq\; cn^2\end{aligned}$$

where the final inequality only holds if $c/4 + 1 \leq c$, i.e., if $1 \leq 3c/4$, i.e., if $4/3 \leq c$. Since this condition can be met by making c large enough, we conclude that $T(n) = O(n^2)$. □

Exercise 8. *Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n-1) + T(n/2) + n$. Use the substitution method to verify your answer.*

Exercise 9. *Use the master method to give tight asymptotic bounds for the following recurrences:*

(1) $T(n) = 2T(n/4) + 1$



One level
n^2
$T(n/2)$
Two levels
n^2
$\frac{n^2}{4}$
$T(n/4)$
Level
0
1
2
$\lg n$
All Levels
n^2
$n^2/4$
$n^2/16$
$T(1)$
Total work:
$\sum_{h=0}^{\lg n - 1} \frac{n^2}{4^h} + T(1)$
$= \Theta(n^2)$
Total work at level $h = n^2/4^h$
$n^2/4^h = 1 \rightarrow n^2 = 4^h$
$\rightarrow h \lg 4 = \lg n^2 \rightarrow h = \lg n$

Figure 1: Recursion tree for $T(n) = T(n/2) + n^2$

(2) $T(n) = 2T(n/4) + \sqrt{n}$
(3) $T(n) = 2T(n/4) + n$
(4) $T(n) = 2T(n/4) + n^2$

Proof. (1) In this case $a = 2$, $b = 4$, $f(n) = 1 = \Theta(1)$ and $n^{\log_4 2} = n^{1/2}$. Since $f(n) = 1 = O(n^{1/2-1/2})$, by case 1 of the Master Theorem we have that $T(n) = \Theta(n^{1/2})$.

(2) In this case $f(n) = n^{1/2} = \Theta(n^{\log_4 2})$, so by case 2 of the Master Theorem $T(n) = \Theta(n^{1/2} \lg n)$.

(3) In this case $f(n) = n = \Theta(n)$, and so $f(n) = \Omega(n^{1/2+1/2})$. Thus we are in case 3 of the Master Theorem and we need to check the Regularity condition of $f(n)$. Note that

$$2f(n/4) \;=\; \frac{1}{2}n \;\leq\; \frac{1}{2}f(n),$$

and so the regularity condition holds for $c := 1/2$. Thus by the Master Theorem we have $T(n) = \Theta(n)$.

(4) In this case $f(n) = \Theta(n^2)$ and $f(n) = \Omega(n^{1/2+1})$. Thus we are in case 3 of the Master Theorem and need to check the regularity condition. Note that

$$2f(n/4) \;=\; \frac{2n^2}{16} \;=\; \frac{1}{8}n^2 \;\leq\; \frac{1}{8}f(n).$$

Thus the regularity condition holds for $c := 1/8$. By the Master Theorem we have $T(n) = \Theta(n^2)$. □

Exercise 10. *Professor Diogenes has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:*

Chip A says	*Chip B says*	*Conclusion*
B is good	*A is good*	*both are good, or both are bad*
B is good	*A is bad*	*at least one is bad*
B is bad	*A is good*	*at least one is bad*
B is bad	*A is bad*	*at least one is bad*

(1) Show that if at least $n/2$ chips are bad, the professor cannot necessarily determine whip chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

(2) Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

(3) Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Exercise 11. *Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are* equivalent *if they correspond to the same account.*

It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Exercise 12. *Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a* local minimum *if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.*

You are given such a complete binary tree T, but the labeling is only specified in the following implicit *way: for each node v, you can determine the value x_v by* probing *the node v. Show how to find a local minimum of T using only $O(\log n)$* probes *to the nodes of T.*

Solution. We will interpret "less than" as "less than or equal to", since otherwise a tree could fail to have a local min if all values are equal.

```
FindMin($v$)
1  Given current node $v$ with label $x_v$:
2      if $v$ has no children
3          return $v$
4      if $v$ has one child $c_1$
5          if $x_{c_1} \geq x_v$
6              return $v$
7          else FindMin($c_1$)
8      if $v$ has two children $c_1$ and $c_2$
9          if $x_v \leq x_{c_1}$ and $x_v \leq x_{c_2}$
10             return $v$
11         else FindMin($c_i$) where $x_{c_i}$ is the smaller child
```

For the running time, since we are traversing along a bath down a complete binary tree of n nodes, the running times is $O(\lg n)$. More specifically, we can bound the running time by an inequality:

$$T(n) \;\leq\; T(n/2) + \Theta(1)$$

and then invoke the Master Theorem.

To prove the correctness of this algorithm, we prove the following claim:

Claim. *Suppose v does not have a parent or $x_{\text{parent}(v)} \geq x_v$. Then calling* FindMin($v$) *will return a local minimum.*

Proof. This is proved by induction on the size of the subtree which has v as its root. If v is the root of a subtree of size 1 (i.e., v is a leaf), then the hypothesis ensures that v is a local minimum. For the inductive step, assume that v does not have a parent or $x_{\text{parent}(v)} \geq x_v$ and assume that FindMin(w) returns a local minimum whenever w is a root of a subtree with few nodes and w satisfies the hypotheses of this claim. Running FindMin(v) has three cases:

(Case 1) Suppose v has no children. Then the assumptions on v make v a local minimum, so we are done.

(Case 2) Suppose v has one child. Then either $x_{c_1} \geq x_v$, in which case v is a local minimum and we are done, or else $x_{c_1} < x_v$, and we call FindMin(c_1). Then the hypotheses are true for x_{c_1} and so by the inductive assumption we return a local minimum.

(Case 3) Suppose v has two children. In the first case, if $x_v \leq x_{c_1}$ and $x_v \leq x_{c_2}$, x_v is itself a local minimum and we are done. Otherwise, if we take the smallest child x_{c_i}, then $x_{c_i} < x_v$, so the hypotheses are true for x_{c_i}, so by the inductive assumption calling FindMin(c_i) will return a local minimum. □

□

Exercise 13 (Programming exercise). *There are exactly ten ways of selecting three from five,* 12345*:*

$$123, 124, 125, 134, 135, 145, 234, 235, 245, \;\; and 345$$

In combinatorics, we use the notation $\binom{5}{3} = 10$.

In general, $\binom{n}{r} = \frac{n!}{r!(n-r)!}$*, where* $r \leq n$*,* $n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$*, and* $0! = 1$.

It is not until $n = 23$*, that a value exceeds one-million:* $\binom{23}{10} = 1144066$.

How many, not necessarily distinct, values of $\binom{n}{r}$*, for* $1 \leq n \leq 100$*, are greater than one-million?*

Solution. First we create Pascal's Triangle with rows up to $n = 100$:

```
binomial = [[1]]

```

```
def next_binomial_row():
    n = len(binomial)
    binomial.append([1])
    for k in range(1,n):
        binomial[n].append(binomial[n-1][k]+binomial[n-1][k-1])
    binomial[n].append(1)
```

Next we count how many entries exceed 1000000:

```
while len(binomial)<=101:
    next_binomial_row()

counter = 0
for n in range(0,101):
    for k in range(0,len(binomial[n])):
        if binomial[n][k]>1000000:
            counter+=1

print(counter)
```

Running this script outputs:

```
4075
```

□

Exercise 14 (Programming exercise)**.** *Euler's Totient function, $\varphi(n)$ (see* ***https://en.wikipedia.org/wiki/Euler%27s_totient_function****) is used to determine the number of numbers less than n which are relatively prime to n (d is* ***relatively prime*** *to n if $\gcd(d, n) = 1$). For example, as $1, 2, 4, 5, 7, 8$ are all less than nine and relatively prime to nine, $\varphi(9) = 6$.*

n	*Relatively Prime*	$\varphi(n)$	$n/\varphi(n)$
2	1	1	2
3	$1, 2$	2	1.5
4	$1, 3$	2	2
5	$1, 2, 3, 4$	4	1.25
6	$1, 5$	2	3
7	$1, 2, 3, 4, 5, 6$	6	1.1666...
8	$1, 3, 5, 7$	4	2
9	$1, 2, 4, 5, 7, 8$	6	1.5
10	$1, 3, 7, 9$	4	2.5

It can be seen that $n = 6$ produces a maximum $n/\varphi(n)$ for $n \leq 10$.

Find the value of $n \leq 1000000$ for which $n/\varphi(n)$ is a maximum.

Solution. First we have some subroutines which grow and maintain a list of primes:

```
import math
primes = [2,3,5]

def next_prime():
    N = primes[-1]+1

    while is_prime(N)==False:
        N+=1

    primes.append(N)
```

```

def is_prime(N):
    #assume primes is long enough to detect primality of N
    sqrtN = int(math.sqrt(N))+1
    for p in primes:
        if N%p==0:
            return False
        if p>sqrtN:
            break

    return True
```

Next we have a subroutine which computes the φ-values. This relies on the formula $\varphi(n) = \varphi(p^k)\varphi(m) = p^{k-1}(p-1)\varphi(m)$ if $n = p^k m$ where $\gcd(p, m) = 1$ and p is prime. We also assume that the array **phi_vals** is long enough and correct enough for this to work.

```
phi_vals = [1,1,1]

def phi(N):
    #assumes phi_vals[k] exists and is correct for k<N
    while primes[-1]<N:
        next_prime()

    exponent=0
    for p in primes:
        if N%p==0:
            while N%p==0:
                exponent+=1
                N=int(N/p)
            return p**(exponent-1)*(p-1)*phi_vals[N]
```

Next we consecutively assign correct values to **phi_vals**:

```
for n in range(2,1000001):
    phi_vals.append(1)

for n in range(2,1000001):
    phi_vals[n] = phi(n)
```

Now that we have all our φ-values computed, we run the following script:

```
max_ratio = 1
max_n = 1
for n in range(2,1000001):
    if n/phi_vals[n]>max_ratio:
        max_ratio = n/phi_vals[n]
        max_n = n

print(max_n)
```

which prints:

```
510510
```

□

Exercise 15 (Programming exercise)**.** *It is possible to write five as a sum in exactly six different ways:*

$$4+1$$

$$3+2$$

$$3+1+1$$

$$2+2+1$$

$$2+1+1+1$$

$$1+1+1+1+1$$

How many different ways can one hundred be written as a sum of at least two positive integers?

Solution. In this problem we are essentially asked to compute the *partition number* of n (really it is one less than the partition number). See `https://en.wikipedia.org/wiki/Partition_function_(number_theory)`. In particular, we use the recurrence:

$$p(n) \;=\; \sum_{k\neq 0}(-1)^{k+1}p(n-k(2k-1)/2)$$

found on the wikipedia page. The following code implements this:

```
partitions=[1]
import math

for n in range(1,101):
    parts=0
    k=1
    while k<=int((math.sqrt(24*n+1)+1)/6):
        parts+=int((-1)**(k+1)*partitions[int(n-k*(3*k-1)/2)])
        k+=1
    k=-1
    while k>=int(-(math.sqrt(24*n+1)-1)/6):
        parts+=int((-1)**(k+1)*partitions[int(n-k*(3*k-1)/2)])
        k-=1
    partitions.append(parts)

print(partitions[100]-1)
```

and it returns

```
190569291
```

□

Exercise 16 (Programming exercise)**.** *The most naive way of computing n^{15} requires fourteen multiplications:*

$$n\times n\times\cdots\times n \;=\; n^{15}$$

But using a "binary" method you can compute it in six multiplications:

$$\begin{aligned} n^2 &= n \times n \\ n^4 &= n^2 \times n^2 \\ n^8 &= n^4 \times n^4 \\ n^{12} &= n^8 \times n^4 \\ n^{14} &= n^{12} \times n^2 \\ n^{15} &= n^{14} \times n \end{aligned}$$

However it is yet possible to compute it in only five multiplications:

$$\begin{aligned} n^2 &= n \times n \\ n^3 &= n^2 \times n \\ n^6 &= n^3 \times n^3 \\ n^{12} &= n^6 \times n^6 \\ n^{15} &= n^{12} \times n^3 \end{aligned}$$

We shall define $m(k)$ to be the minimum number of multiplications to compute n^k, for example $m(15) = 5$.

For $1 \leq k \leq 200$, find $\sum m(k)$.

Solution. A sequence of numbers $a_0, \ldots, a_n$ such that $a_0 = 1$ and for each $i > 1$, $a_i = a_j + a_k$ for some $j, k < i$, is called an addition chain for a_n. Its length is n. This problem asks for the sum over k from 1 to 200 of the lengths of the shortest addition chain for k.

A straightforward search of possible chains is infeasibly slow. In order to cut the running time down, we make two observations:

First, if $a_1, \ldots, a_n$ is an addition chain, then for any $1 \leq k < n$, $a_{k+1} \leq 2a_k$, and therefore $a_n \leq 2^{n-j} a_k$. If we know that there is a chain of length at most n whose last element is m, and we have a candidate addition chain of length r which we think might extend to an optimal addition chain for m, then its last element must be at least $2^{r-n} m$. We will use this to discard candidate chains which do not grow quickly enough.

Second, while it is difficult to find optimal addition chains (a closely-related problem is known to be NP-complete), it is much less difficult to find nearly-optimal addition chains. If we assume that an optimal addition chain must be of the form $a_0, \ldots, a_n$ with $a_n = a_{n-1} + a_i$ for some $i \leq n-1$ and $a_0, \ldots, a_{n-1}$ an optimal addition chain for a_{n-1}, then we can find optimal addition chains by dynamic programming. This assumption is false in general (although the smallest counterexample is greater than 200), but we can still use it to find good upper bounds.

The following Python code implements the dynamic programming approximation:

```python
def guess_lengths(n):
    #This algorithm uses a dynamic programming approach
    #to attempt to find optimal chains for numbers below n.
    #It relies on assumptions about optimal chains which
    #are not true in general, so it may produce chains
    #which are too long.  We will use it as an upper bound.
    chains=[[1]]
    cur_chains=[[1]]
    min_not_found=2
    not_found=set(range(2,n+1))
    while len(not_found)>0:
        to_extend=cur_chains.copy()
        cur_chains=[]
        just_found=[]
        for chain in to_extend:
            for j in chain:
                new=chain[-1]+j
                if new in not_found:
                    cur_chains.append(chain+[new])
                    just_found.append(new)
        for new in set(just_found):
            not_found.remove(new)
        chains+=cur_chains
    lengths=[0]*n
    for chain in chains:
        lengths[chain[-1]-1]=len(chain)-1
    return lengths
```

We then use this to bound the lengths of chains when searching for optimal chains:

```
def min_addition_chains(n):
    lengths_guess=guess_lengths(n)
    bound=max(lengths_guess)-1
    #We'll only search for chains strictly shorter than
    #those found by guess_lengths.
    chains={tuple([1])}
    min_not_found=2
    not_found=set(range(2,n+2))
    min_lengths={1:0}
    done=False
    for i in range(1,bound+1):
        tempchains=set([])
        comp=min_not_found>>(bound-i)
        #A chain of length i which ends in a number below
        #comp cannot extend to an optimal chain for a
        #number above min_not_found.
        for chn in chains:
            r=chn[-1]
            for a in reversed(chn):
                for b in reversed(chn):
                    new=a+b
                    if new<=r or b<a or new<comp:
                        break
                    if new in not_found and new<=n:
                        not_found.remove(new)
                        min_lengths[new]=i
                        min_not_found=min(not_found)
                    tempchains.add(chn+tuple([new]))
            if min_not_found>n:
                break
        if min_not_found>n:
            break
        chains=tempchains.copy()
    for new in not_found:
        min_lengths[new]=bound+1
        #If we haven't found a chain of length at most
        #bound, then the shortest one has length bound+1.
    ret=[min_lengths[j+1] for j in range(n)]
    return ret
```

The desired result is then computed by

```
sum(min_addition_chains(200))
```

which returns

```
1582
```

□

Exercise 17 (Programming exercise)**.** *Looking at the table below, it is easy to verify that the maximum possible sum of adjacent numbers in any direction (horizontal, vertical, diagonal, or anti-diagonal) is* $16(= 8 + 7 + 1)$.

-2	5	3	2
9	-6	5	1
3	2	7	3
-1	8	-4	8

Now, let us repeat the search, but on a much larger scale:

First, generate four million pseudo-random numbers using a specific form of what is known as a "Lagged Fibonacci Generator":

- *For* $1 \le k \le 55$, $s_k = [100003 - 200003k + 300007k^3] (\bmod 1000000) - 500000$.
- *For* $56 \le k \le 4000000$, $s_k = [s_{k-24} + s_{k-55} + 1000000] (\bmod 1000000) - 500000$

Thus, $s_{10} = -393027$ *and* $s_{100} = 86613$.

The terms of s *are then arranged in a* 2000×2000 *table, using the first* 2000 *numbers to fill the first row (sequentially), the next* 2000 *numbers to fill the second row, and so on.*

Finally, find the greatest sum of (any number of) adjacent entries in any direction (horizontal, vertical, diagonal, or anti-diagonal).

Solution. The following Python code generates an $n \times n$ matrix using the specified lagged Fibonacci generator:

```
def generate_matrix(n):
    l=[]
    for j in range(55):
        l.append(((100003-200003*(j+1)+300007*((j+1)**3))%1000000)-500000)
    for j in range(55,n*n):
        l.append(((l[j-24]+l[j-55])%1000000)-500000)
    outarray=[l[n*i:n*(i+1)] for i in range(n)]
    return outarray
```

The following function executes the maximum subarray sum algorithm from Exercise 2 on the rows, columns, and diagonals of a square matrix:

```
def max_adjacent_sum(mat):
    n=len(mat)
    curmax=mat[0][0]
    for i in range(n):
        maxhere=0
        for j in range(n):
            maxhere=mat[i][j]+max(0,maxhere)
            curmax=max(curmax,maxhere)
        maxhere=0
        for j in range(n):
            maxhere=mat[j][i]+max(0,maxhere)
            curmax=max(curmax,maxhere)
        maxhere=0
        for j in range(n-i):
            maxhere=mat[i+j][j]+max(0,maxhere)
            curmax=max(curmax,maxhere)
        maxhere=0
        for j in range(n-i):
            maxhere=mat[j][i+j]+max(0,maxhere)
            curmax=max(curmax,maxhere)
        maxhere=0
        for j in range(i+1):
            maxhere=mat[j][i-j]+max(0,maxhere)
            curmax=max(curmax,maxhere)
        maxhere=0
        for j in range(i+1,n):
            maxhere=mat[j][n+i-j]+max(0,maxhere)
            curmax=max(curmax,maxhere)
    return curmax
```

Calling

```
max_adjacent_sum(generate_matrix(2000))
```

then returns

```
52852124
```

□