MATH182 HOMEWORK #4
DUE July 19, 2020

Exercise 1. *This problem is about heaps:*

1. *(1) What are the minimum and maximum numbers of elements in a heap of height h?*
2. *(2) Show that an n-element heap has height $\lfloor \lg n \rfloor$.*
3. *(3) Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.*

Solution. (1) Since the k^{th} level of a heap has exactly 2^k elements for $0 \leq k < h$, and at least 1 element in the last level (else heap would have height $h-1$) and at most 2^h elements in the last level (else heap would need an additional level), the minimum and maximum number of elements in a heap, $n_{\min}$ and $n_{\max}$ are given by

$$n_{\min} = \sum_{k=0}^{h-1} 2^k + 1 = \frac{2^h - 1}{2-1} + 1 = \underline{2^h}$$

$$n_{\max} = \sum_{k=0}^{h} 2^k = \frac{2^{h+1} - 1}{2-1} = \underline{2^{h+1} - 1}$$

(2) Let h be the height of an n-element heap. From (1), we have

$$\begin{aligned}
2^h \leq n \leq 2^{h+1} - 1 &\implies \lg 2^h \leq \lg n \leq \lg(2^{h+1} - 1) && \text{(lg is strictly increasing)} \\
&\implies h \leq \lg n \leq \lg(2^{h+1} - 1) < \lg 2^{h+1} \\
&\implies h \leq \lg n < h + 1 \\
&\implies h = \lfloor \lg n \rfloor \ \blacksquare && \text{(by definition of floor function)}
\end{aligned}$$

(3) Let h be the height of the tree and l_h be the number of elements in the last level. Since n is the total number of elements in the tree and each level $k \in [0, h-1]$ of the tree has 2^k elements, we have

$$l_h = n - \sum_{k=0}^{h-1} 2^k = n - \frac{2^h - 1}{2-1} = n - 2^h + 1$$

Since each of these elements are in the last level of the heap, we have exactly l_h leaves in the last level. Furthermore, since these elements are filled left-to-right, the number of parents they share (all in the second-last level) is given by $\lceil l_h/2 \rceil$. We therefore have the number of leaves (i.e., child-less) nodes in the second-last level given by

$$l_{h-1} = 2^{h-1} - \left\lceil \frac{l_h}{2} \right\rceil = 2^{h-1} - \left\lceil \frac{n - 2^h + 1}{2} \right\rceil = 2^{h-1} - \left\lceil \frac{n+1}{2} \right\rceil + 2^{h-1} = 2^h - \left\lceil \frac{n+1}{2} \right\rceil$$

Since all leaves in levels above the second-last have children and are therefore not leaves, the total number of leaves in the heap is given by

$$\begin{aligned} l = l_h + l_{h-1} &= n - 2^h + 1 + 2^h - \left\lceil \frac{n+1}{2} \right\rceil \\ &= n + 1 - \left\lceil \frac{n+1}{2} \right\rceil = n - \left(\left\lceil \frac{n+1}{2} \right\rceil - 1 \right) \\ &= n - \left(\left\lceil \frac{n+1}{2} - 1 \right\rceil \right) = n - \left\lceil \frac{n-1}{2} \right\rceil \end{aligned}$$

In the case that n is odd, we have

$$2|(n-1) \implies \left\lceil \frac{n-1}{2} \right\rceil = \frac{n-1}{2},\ \left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2} \implies \left\lceil \frac{n-1}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$$

In the case that n is even, we have

$$2|n \implies \left\lceil \frac{n-1}{2} \right\rceil = \frac{n}{2},\ \left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2} \implies \left\lceil \frac{n-1}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$$

Therefore, for all integer $n > 0$, we have $\lceil (n-1)/2 \rceil = \lfloor n/2 \rfloor$. Plugging this into our expression for l, we have

$$l = n - \left\lceil \frac{n-1}{2} \right\rceil = n - \left\lfloor \frac{n}{2} \right\rfloor$$

Therefore, in an array-representation for storing an n-element heap, the last $n - \lfloor n/2 \rfloor$ elements are leaves. In other words, all elements after the first $\lfloor n/2 \rfloor$ elements are leaves. The leaves are therefore the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$. ■

Exercise 2. *The code for* MAX-HEAPIFY *is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient* MAX-HEAPIFY *that uses an iterative control construct (a loop) instead of recursion.*

Solution. Following is pseudocode for a MAX-HEAPIFY that uses a while loop instead of recursion:

```
MAX-HEAPIFY(A, i):
k = i // current node initialised to starting node
while true
    // finding left and right child of current node
    l = Left(k)
    r = Right(k)
    // finding largest of current node and children
    largest = k   // current node is assumed largest by default
    if l ≤ A.heap-size and A[l] > A[largest]
        largest = l
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    // there is a violation of the heap property, swap required
    if largest ≠ k
        exchange A[k] with A[largest]
        k = largest
    // no violation of the heap property, so the subtree starting at i has been max-heapified
    else break
```

Exercise 3. *Show that there are at most* $\lceil n/2^{h+1} \rceil$ *nodes of height* h *in any* n*-element heap.*

Exercise 4. *Argue the correctness of* HEAPSORT *using the following loop invariant:*

> *(Loop Invariant) At the start of each iteration of the **for** loop of lines 2-5, subarray* $A[1 \mathinner{..} i]$ *is a max-heap containing the* i *smallest elements of* $A[1 \mathinner{..} n]$*, and the subarray* $A[i+1 \mathinner{..} n]$ *contains the* $n-i$ *largest elements of* $A[1 \mathinner{..} n]$*, sorted.*

Solution. The HEAPSORT algorithm is as follows:

HEAPSORT(A):

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       exchange A[1] with A[i]
4       A.heap-size = A.heap-size −1
5       MAX-HEAPIFY(A, 1)
```

We prove the loop invariant.

Initialisation: When line 2 is first run, we have $i = n$ for $n := A.length$.

Since $A[1 \mathinner{..} i] = A[1 \mathinner{..} n] \equiv A$, $A[1 \mathinner{..} i]$ is a max-heap since BUILD-MAX-HEAP was called on A in line 1. Furthermore, $A[1 \mathinner{..} i]$ (trivially) contains the $i = n$ smallest elements of $A[1 \mathinner{..} n]$, and the subarray $A[i+1 \mathinner{..} n] = A[n+1 \mathinner{..} n]$ is empty and therefore trivially contains the $n - i = n - n = 0$ largest elements of $A[1 \mathinner{..} n]$, sorted. The loop invariant is therefore true at initialisation.

Maintenance: Assume the loop invariant is true before some iteration with $i = i_0 \in [2, A.length]$. From the loop invariant, we know $A[1 \mathinner{..} i_0]$ is a max-heap containing the i_0 smallest elements of $A[1 \mathinner{..} n]$, and the subarray $A[i_0 + 1 \mathinner{..} n]$ contains the $n - i_0$ largest elements of $A[1 \mathinner{..} n]$, sorted.

Since $a_m := A[1] \in A[1 \mathinner{..} i_0]$, by assumption on the subarrays $A[1 \mathinner{..} i_0]$ and $A[i_0 + 1 \mathinner{..} n]$, we have $a_m \leq A[j]$ for all $j \in [i_0 + 1, n]$. Furthermore, by assumption $A[1 \mathinner{..} i_0]$ is a max-heap, we also know $A[1] \geq A[j]$ for all $j \in [1, i_0]$. In line 3, we exchange $A[1] = a_m$ with $A[i] = a_r$. The modified subarray $A[1 \mathinner{..} i_0 - 1]$ now has the $i_0 - 1$ smallest elements of $A[1 \mathinner{..} n]$, since only its largest element was removed (and by assumption on the subarray that was previously $A[1 \mathinner{..} i_0]$). Furthermore, the subarray $A[i_0 \mathinner{..} n]$ conversely contains the $n - i_0 + 1$ largest elements of $A[1 \mathinner{..} n]$ and is sorted, since $A[i_0 + 1 \mathinner{..} n]$ was already sorted and we have already shown $A[i_0] = a_m \leq A[j]$ for all $j \in [i_0 + 1, n]$.

In line 4, we decrement A.heap-size so as to no longer include $A[i_0]$ in the heap, i.e., the heap is now represented by $A[1 \mathinner{..} i_0 - 1]$. In line 5, we call MAX-HEAPIFY on the root node of A. Since both sub-trees of $A[1]$ are still exactly the same as the sub-trees of $A[1]$ at the beginning of the iteration with only the leaf element $A[i_0]$ removed (which doesn't invalidate the max-heap property), both sub-trees are max-heaps. Therefore, when MAX-HEAPIFY on $A[1]$, it makes the subarray $A[1 \mathinner{..} i_0 - 1]$ into a max-heap.

When the iteration finishes and line 2 is run again, we have $i = i_0 - 1$. We have shown $A[1 \mathinner{..} i_0 - 1]$ is a max-heap containing the $i_0 - 1$ smallest elements of $A[1 \mathinner{..} n]$, and the subarray $A[i_0 \mathinner{..} n]$ contains the $n - i_0 + 1$ largest elements of $A[1 \mathinner{..} n]$, sorted. The loop invariant is therefore true for $i = i_0 - 1$.

Termination: When line 2 is last run, we have $i = 1$. From the loop-invariant, we know the subarray $A[i+1 \mathinner{..} n] = A[2 \mathinner{..} n]$ contains the $n - i = n - 1$ largest elements of $A[1 \mathinner{..} n]$, sorted, and $A[1 \mathinner{..} i] = A[1]$ contains the smallest element of $A[1 \mathinner{..} n]$.

The array $A[1 \mathinner{..} n]$ is therefore sorted, and HEAPSORT is correct. ■

Exercise 5. *What is the running time of* HEAPSORT *on an array* A *of length* n *that is already sorted in increasing order? What about decreasing order?*

Solution. Since an array that is sorted in increasing order is a min-heap, BUILD-MAX-HEAP must do at least as much work to convert it into a max-heap as it would with with any other heap. At the end of that process, the array will no longer be in increasing order, and will look more or less like an average n-element max-heap. HEAPSORT will therefore still take $O(n \lg n)$ time, since the **for** loop will still run $n-1$ times and MAX-HEAPIFY will still cost $O(\lg n)$ at each call.

The same holds true for an array that is already sorted in descending order. While such an array would be a max-heap before BUILD-MAX-HEAP is called (and BUILD-MAX-HEAP may therefore run quicker than it would in the case of an array sorted in increasing order), the time complexity of the loop is still $O(n \lg n)$.

The running time of HEAPSORT is therefore $O(n \lg n)$ in both cases.

Exercise 6. *Argue the correctness of* HEAP-INCREASE-KEY *using the following loop invariant:*

> *(Loop Invariant) At the start of each iteration of the* ***while*** *loop of lines 4-6,* $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ *and* $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$*, if these nodes exist, and the subarray* $A[1 \,..\, A.\textit{heap-size}]$ *satisfies the max-heap property, except that there may be one violation:* $A[i]$ *may be larger than* $A[\text{PARENT}(i)]$*.*

You may assume that the the subarray $A[1 \,..\, A.\textit{heap-size}]$ *satisfies the max-heap property at the time* HEAP-INCREASE-KEY *is called.*

Exercise 7. *A* d***-ary heap*** *is just like a binary heap, but (with one possible exception) non-leaf nodes have* d *children instead of* 2 *children.*

(1) How would you represent a d-ary heap in an array?
(2) What is the height of a d-ary heap of n elements in terms of n and d?
(3) Give an efficient implementation of EXTRACT-MAX *in a d-ary max-heap. Analyze its running time in terms of d and n.*
(4) Give an efficient implementation of INSERT *in a d-ary max-heap. Analyze its running time in terms of d and n.*
(5) Give an efficient implementation of INCREASE-KEY(A, i, k)*, which flags an error if* $k < A[i]$*, but otherwise sets* $A[i] = k$ *and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.*

Solution. (1) A d-ary heap would perhaps best be represented the same way a binary heap is, by filling the array level-by-level. The first element of the array would therefore be the root node of the heap, the next d elements would be the children of the root node from left to right, and so on. This gives us the following generalised function for finding the parent and i^{th} child indices (with zero-based indexing) of a node at index k in a d-ary heap:

PARENT(d, k):
if $k == 0$
    error "root node has no parent"
return $\lfloor (k-1)/d \rfloor$

CHILD(d, k, i):
if $i < 1$ **or** $i > d$
    error "i^{th} child cannot exist"
return $dk + i$ // return index of i^{th} child of node at index k

(2) We first address the trivial case where $d = 1$; in this case, the height h of the heap is simply the number of elements n. Now assume $d \geq 2$. Since the k^{th} level of a d-ary heap of height h has exactly d^k elements for $0 \leq k < h$, and at least 1 element in the last level (else heap would have

height $h-1$) and at most d^h elements in the last level (else heap would need an additional level), the minimum and maximum number of elements in a heap, n_{min} and n_{max} are given by

$$n_{\text{min}} = \sum_{k=0}^{h-1} d^k + 1 = \frac{d^h - 1}{d-1} + 1$$

$$n_{\text{max}} = \sum_{k=0}^{h} d^k = \frac{d^{h+1} - 1}{d-1}$$

Let h be the height of an n-element d-ary heap. From the equations for n_{min} and n_{max}, we have

$$\begin{aligned} n_{\text{min}} \leq n \leq n_{\text{max}} &\implies \frac{d^h - 1}{d-1} + 1 \leq n \leq \frac{d^{h+1} - 1}{d-1} \\ &\implies \frac{d^h + d - 2}{d-1} \leq n \leq \frac{d^{h+1} - 1}{d-1} \\ &\implies d^h + d - 2 \leq n(d-1) \leq d^{h+1} - 1 \end{aligned}$$

Notice that $d^{h+1} - 1 < d^{h+1}$ and, since $d \geq 2$ (by assumption), $d^h \leq d^h + d - 2$.

$$\begin{aligned} &\implies d^h \leq n(d-1) < d^{h+1} \\ &\implies h \leq \log_d(n(d-1)) < h + 1 && \text{(taking } \log_d \text{ on all sides)} \\ &\implies h = \lfloor \log_d(n(d-1)) \rfloor && \text{(by definition of floor function)} \end{aligned}$$

The height of a d-ary heap of n elements is therefore given by

$$h(d,n) = \begin{cases} n & \text{if } d = 1 \\ \lfloor \log_d(n(d-1)) \rfloor & \text{if } d \geq 2 \end{cases}$$

(3) We first need to define a MAX-HEAPIFY function that max-heapifies a node at index i (in zero-based indexing, since we've defined our PARENT and CHILD functions with zero-based indexing) in a d-ary heap.

MAX-HEAPIFY(A, d, k):

```
1   k = i // current node initialised to starting node
2   while true
3       // finding largest of current node and all its children
4       largest = k              // current node is assumed largest by default
5       c_1 = CHILD(d, k, 1)     // first child index of current node
6       // iterating over all children and comparing largest
7       for c_i = c_1 to c_1 + d
8           if c_i < A.heap-size and A[c_i] > A[largest]
9               largest = c_i
10      // there is a violation of the heap property, swap required
11      if largest ≠ k
12          exchange A[k] with A[largest]
13          k = largest
14      // no violation of the heap property, so the subtree starting at i has been max-heapified
15      else break
```

To simplify our analysis of running time of MAX-HEAPIFY, we assume $d \geq 2$. We note that line 1 outside the **while** loop and lines 3-6 and 10-15 all take constant time per iteration of the loop, and focus our analysis on the number of times the outer **while** and inner **for** loops run.

Since in each iteration we descend one level, the number of iterations of the **while** loop is bounded by the height of the tree, $O(\log_d(n(d-1)))$. Furthermore, since the inner **for** loop of lines 7-9 runs exactly d times, the running time of each iteration of the while loop is given by $\Theta(d)$. The running time of MAX-HEAPIFY is therefore $O(d \log_d(n(d-1)))$.

Since the EXTRACT-MAX algorithm we've discussed for a binary heap nowhere relies on the binary nature of the heap, we can easily generalise it to a d-ary heap as follows:

EXTRACT-MAX(A, d):

1 **if** A.heap-size < 1
2   **error** "heap underflow"
3 $max = A[0]$
4 $A[0] = A[\text{heap-size} - 1]$
5 A.heap-size $= A$.heap-size $- 1$
6 MAX-HEAPIFY$(A, d, 0)$
7 **return** max

Since lines 1-5 and 7 all take constant time, the running time of EXTRACT-MAX is simply the running time of MAX-HEAPIFY, i.e., $\underline{O(d \log_d(n(d-1)))}$.

(4) Since the INSERT algorithm we've discussed for a binary heap nowhere relies on the binary nature of the heap, we can easily generalise it to a d-ary heap as follows:

INSERT(A, d, key):

A.heap-size $= A$.heap-size $+ 1$
$A[A.\text{heap-size} - 1] = -\infty$
INCREASE-KEY$(A, d, A.\text{heap-size} - 1, key)$

This implementation relies on the INCREASE-KEY function defined in part (4), which has running time $\underline{O(\log_d(n(d-1)))}$.

(5) Since the INCREASE-KEY algorithm we've discussed for a binary heap nowhere relies on the binary nature of the heap, we can easily generalise it to a d-ary heap as follows:

INCREASE-KEY(A, d, i, k):

1 **if** $k < A[i]$
2   **error** "new key is smaller than current key"
3 $A[i] = k$
4 **while** $i > 0$ **and** $A[\text{PARENT}(d, i)] < A[i]$
5   exchange $A[i]$ with $A[\text{PARENT}(d, i)]$
6     $i = \text{PARENT}(d, i)$

Since lines 1-3 take constant time, and the while loop in lines 4-6 runs a number of times bounded by the height of the tree, the running time of Increase-Key is $\underline{O(\log_d(n(d-1)))}$.

Exercise 8. *An $m \times n$ **Young tableau** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus a Young tableau can be used to hold $r \leq mn$ finite numbers.*

(1) Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

(2) Argue that an $m \times n$ Young tableau Y is empty if $Y[1,1] = \infty$. Arge that Y is full (contains mn elements) if $Y[m,n] < \infty$.

(3) Give an algorithm to implement Extract-Min *on a nonempty $m \times n$ Young tableau that runs in $O(m+n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m-1) \times n$ or an $m \times (n-1)$ subproblem. (Hint: Think about* Max-Heapify*.) Define $T(p)$, where $p = m+n$, to be the maximum running time of* Extract-Min *on any $m \times n$ Young tableau. Give and solve a recurrence for $T(p)$ that yields the $O(m+n)$ time bound.*

(4) Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m+n)$ time.

(5) Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau in to sort n^2 numbers in $O(n^3)$ time.

(6) Given an $O(m+n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Solution. (1) The following matrix is a Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$:

$$\begin{pmatrix} 2 & 3 & 4 & 5 \\ 8 & 9 & 12 & 14 \\ 16 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix}$$

(2) Let Y be an $m \times n$ Young tableau. We first assume $Y[1,1] = \infty$. By definition of a Young tableau, every element in in the first row $A[1, 1 \,..\, n] \geq A[1,1] = \infty$. The first row is therefore fully populated by ∞. Similarly, the first element in each column is the smallest in that column (by definition of a Young tableau) and since, in the case $Y[1,1] = \infty$, we have shown the first row is fully populated by ∞, every element $A[i,j]$ in the Young tableau must be such that $A[i,j] \geq \infty$. In other words, the Young tableau is completely populated by ∞. Since entries represented by ∞ are treated as non-existent elements, the Young tableau must be empty.

Now assume $Y[m,n]\infty$. By definition of a Young tableau, every element in in the last row $A[m, 1 \,..\, n] \leq A[m,n] = \infty$. The first row is therefore fully populated by finite elements. Assume towards contradiction that Y has an empty entry at some $A[i,j]$ for $i \in [1,m)$ (we have already shown last row is full) and $j \in [1,n]$, i.e., $A[i,j] = \infty$. Since by assumption of a Young tableau, $A[i,j] = \infty \leq A[m,j] < \infty$, we have the contradiction $\infty < \infty$.

The Young tableau must therefore be full. ■

(3) Following is pseudocode that extracts the minimum from a nonempty $m \times n$ Young tableau:

Extract-Min() :

// reached empty part of Young tableau
if $Y[1,1] == \infty$
    return ∞
// extracting minimum
$min = Y[1,1]$
// extracting numbers of rows and columns
$m = Y$.rows
$n = Y$.cols
// one-element base case
if $m == n == 1$
    $Y[1,1] = \infty$
// there is an element below and to the right
else if $m > 1$ **and** $n > 1$
    if $Y[1,2] \leq Y[2,1]$
        // element to the right is larger; update $Y[1,1]$ to minimum of right sub-tableau
        $Y[1,1] =$ Extract-Min($Y[1\,..\,m, 2\,..\,n]$)
    else
        // element below is larger; update $Y[1,1]$ to minimum of below sub-tableau
        $Y[1,1] =$ Extract-Min($Y[1\,..\,m, 2\,..\,n]$)
// only one column, multiple rows
else if $m > 1$
    $Y[1,1] =$ Extract-Min($Y[2\,..\,m, 1]$)
// only one row, multiple columns
else
    $Y[1,1] =$ Extract-Min($Y[1, 2\,..\,n]$)
return min // returning min

Exercise 9. *Explain how to implement two stacks in one array $A[1\,..\,n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The* Push *and* Pop *operations should run in $O(1)$ time.*

Exercise 10. *Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.*

Exercise 11. *Modify* Memoized-Cut-Rod *to return not only the value but the actual solution, too.*

Exercise 12 (Programming Exercise)**.** *Let $d(n)$ be defined as the sum of proper divisors of n (numbers less than n which divide evenly into n). If $d(a) = b$ and $d(b) = a$, where $a \neq b$, then a and b are an **amicable pair** and each of a and b are called **amicable numbers**.*

For example, the proper divisors of 220 *are* $1, 2, 4, 5, 10, 11, 20, 22, 44, 55$ *and* 110*; therefore* $d(220) = 284$. *The proper divisors of* 284 *are* $1, 2, 4, 71$ *and* 142*; so* $d(284) = 220$.

Evaluate the sum of all the amicable numbers under 10000.

(This page might be helpful: https://en.wikipedia.org/wiki/Divisor$_f$unction)

Solution. The following function, `findAmicableSum`, takes input `bound` and computes the sum of all amicable numbers under `bound`.

Calling `findAmicableSum(10000)` returns 31626.

```
#include <unordered_map>
using namespace std;

// auxiliary functions
int checkAmicability(int a, unordered_map<int, int>& divisorSums);
int findDivisorSum(int n, unordered_map<int, int>& divisorSums);
  // defined in appendix

int findAmicableSum(int bound) {
  // returns the sum of all amicable numbers under bound

  // vector to store sum of all divisors
  unordered_map<int, int> divisorSums;

  int amicableSum = 0;
  // finding sum of amicable numbers less than bound
  for (int n = 1; n < bound; n++)
    if (checkAmicability(n, divisorSums))
      amicableSum += n;

  return amicableSum;
}

int checkAmicability(int a, unordered_map<int, int>& divisorSums) {
  // checks whether input number a is amicable

  // finding divisor sum for a
  int b = findDivisorSum(a, divisorSums); // divisorSums[a] = b
  if (b != a) {
    // finding d(b)
    int d_b = findDivisorSum(b, divisorSums);
    // we check if d(b) == a
    if (d_b == a)
      return true;  // a is amicable
  }
  // a is not amicable
  return false;
}
```

Exercise 13 (Programming Exercise)**.** *A **perfect number** is a number for which the sum of its proper divisors is exactly equal to the number. For example, the sum of the proper divisors of* 28 *would be* $1 + 2 + 4 + 7 + 14 = 28$*, which means that* 28 *is a perfect number.*

A number n *is called **deficient** if the sum of its proper divisors is less than* n *and it is called **abundant** if this sum exceeds* n*.*

As 12 *is the smallest abundant number,* $1 + 2 + 3 + 4 + 6 = 16$*, the smallest number that can be written as the sum of two abundant numbers is* 24*. By mathematical analysis, it can be shown that all integers greater than* 28123 *can be written as the sum of two abundant numbers. However, this upper limit cannot be reduced any further by analysis even though it is known that the greatest number that cannot be expressed as the sum of two abundant numbers is less than this limit.*

Find the sum of all the positive integers which cannot be written as the sum of two abundant numbers.

Solution. The following function, `findNonAbundantSum` returns the sum of all positive integers (known to be less than 28123) which cannot be written as the sum of two abundant numbers.

Calling `findNonAbundantSum()` returns 4179871.

```
#include <unordered_map>
using namespace std;

int findDivisorSum(int n, unordered_map<int, int>& divisorSums);
    // defined in appendix

int checkAbundance(int n, unordered_map<int, int>& divisorSums) {
  // check if n is an abundant number
  if (findDivisorSum(n, divisorSums) > n)
    return true;
  else return false;
}

int findNonAbundantSum() {
  // returns the sum of all positive integers that can't be expressed as a sum of
    two abundant numbers

  const int upperBound = 28123;
  // known upper bound on positive integers that can't be expressed as sum of two
    abundant numbers

  unordered_map<int, int> divisorSums;
      // map of numbers to the sum of their divisors
  vector<int> abundantNumbers;
      // vector of abundant numbers (for ordered iteration)

  // generating abundant numbers up to upperBound
  for (int k = 1; k <= upperBound; k++) {
    if (checkAbundance(k, divisorSums))
      abundantNumbers.push_back(k);
  }

  vector<int> abundantSumExpressible(upperBound + 1, false);
    // creating a vector of upperBound + 1 candidates for the desired numbers
    // abundantSumExpressible[k] = true if k can be expressed as a sum of two
    abundant numbers, false otherwise

  // number of abundant numbers found
  const int abundantCount = abundantNumbers.size();

  // finding all sums of abundant numbers <= upperBound
  // iterating over abundant numbers
  for (int i = 0; i < abundantCount; i++) {
    int abundantOne = abundantNumbers[i];
    // iterating over abundant numbers >= abundantone (to avoid overlap)
    for (int j = i; j < abundantCount; j++) {
```

```

      int sum = abundantOne + abundantNumbers[j]; // sum of abundant numbers
      // updating abundantSumExpressible[sum]
      if (sum <= upperBound)
        abundantSumExpressible[sum] = true;
      // if sum > upperBound, future sums will also be greater -> break
      else break;
    }
  }

  // finding sum of numbers not expressible as sums of abundant numbers
  int nonAbundantSum = 0;
  // iterating over abundantSumExpressible
  for (int k = 1; k < abundantSumExpressible.size(); k++)
    // if k is not expressible as sum of two abundant numbers, add k to nonAbundantSum
    if (!abundantSumExpressible[k])
      nonAbundantSum += k;

  return nonAbundantSum;
}
```

Exercise 14 (Programming Exercise)**.** *The number* 3797 *has an interesting property. Being prime itself, it is possible to continuously remove digits from left to right, and remain prime at each stage:* $3797, 797, 97,$ *and* 7*. Similarly we can work from right to left:* $3797, 379, 37,$ *and* 3*.*

Find the sum of the only eleven primes that are both truncatable from left to right and right to left.

Note: $2, 3, 5,$ *and* 7 *are not considered to be truncatable primes.*

Solution. The following function, `findTruncatablePrimeSum` returns the sum of all eleven primes that are both truncatable from left to right.

Calling `findTrunctablePrimeSum()` returns <u>748317</u>.

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

bool checkTruncatability(int prime, unordered_set<int>& primesSet);

int findTruncatablePrimeSum() {
  // returns sum of all primes truncatable from left to right with preserved primality

  unordered_set<int> primesSet;
      // set of primes up to primeBound (for constant search)
  vector<int> primesVec;
      // vector of primes up to primeBound (for ordered iteration)

  // generating primes and populating primesSet and primesVec
  // generatePrimes(1000000, primesSet, primesVec);

  const int maxTruncatablePrimes = 11;  // known number of truncatable primes
```

```cpp
  int truncatablePrimeCount = 0;  // number of trunctable primes found

  int truncatablePrimeSum = 0;  // sum of truncatable primes

  // continually generating and testing primes
  for (int n = 2; truncatablePrimeCount < maxTruncatablePrimes; n++) {

    // finding prime
    bool primeFound = true;
        // we assume we have a prime until and unless we find a prime factor
    int maxPrimeFactor = ceil(sqrt(n));

    // searching for prime factor
    for (int prime : primesVec) {
      if (n % prime == 0)
        primeFound = false;
      if (prime > maxPrimeFactor || !primeFound)
        break;
    }

    // factor of i found -> i is a prime
    if (!primeFound)
      continue;

    // no factor of i found -> i is a prime
    else {
      int prime = n;

      // updating primesVec and primesSet
      primesVec.push_back(prime);
      primesSet.insert(prime);

      if (checkTruncatability(prime, primesSet)) {
        // updating truncatable prime sum and count
        truncatablePrimeSum += prime;
        truncatablePrimeCount++;
      }
    }
  }
  return truncatablePrimeSum;
}

bool checkTruncatability(int prime, unordered_set<int>& primesSet) {
  // returns whether a prime is truncatable

  // 2, 3, 5, and 7 don't count as truncatable primes
  if (prime < 10)
    return false;

  bool truncatablePrime = true; // we assume prime is truncatable by default
  int primeLength = ceil(log10(prime)); // number of digits in prime

  // truncating prime from left to right and checking primality
```

```
  for (int p = primeLength - 1; p >= 1 && truncatablePrime; p--) {
    int truncatedPrime = prime % int(pow(10, p)); // truncated prime
    // checking primality
    if (primesSet.find(truncatedPrime) == primesSet.end())
      truncatablePrime = false;
  }

  // truncating primes from right to left and checking primality
  for (int p = 1; p < primeLength && truncatablePrime; p++) {
    int truncatedPrime = prime / int(pow(10, p)); // truncated prime
    // checking primality
    if (primesSet.find(truncatedPrime) == primesSet.end())
      truncatablePrime = false;
  }

  return truncatablePrime;
}
```

Exercise 15 (Programming Exercise)**.** *If p is the perimeter of a right angle triangle with integer length sides, $\{a, b, c\}$, there are exactly three solutions for $p = 120$. $\{20, 48, 52\}$, $\{24, 45, 51\}$, $\{30, 40, 50\}$.*

For which value of $p \leq 1000$, is the number of solutions maximised? (This page might be helpful: https://en.wikipedia.org/wiki/Pythagorean_triple)

Solution. We use Euclid's formula to generate all Pythagorean triples up to the given bound of 1000. Given any arbitrary integers m, n such that $0 < n < m$, Euclid's formula gives all primitive triples (a, b, c) as follows:

$$a = m^2 - n^2, \ b = 2mn, \ c = m^2 + n^2$$

If we define B to be the bound on the perimeter of each of these triplets, we have

$$a + b + c \leq B \implies 2m(m + n) \leq B \implies 2m^2 + 2mn \leq B$$

We wish to iterate over all m such that the inequality holds true for some n. Since the left-hand side of the inequality is strictly increasing for increasing m and n, we can find the maximal value of m such that the inequality is true for some n by simply setting $n = 1$. In other words, we iterate m from 2 upwards as long as $2m^2 + 2m = 2m(m + 1) \leq B$.

Similarly, for each m up to this limit, we will iterate n upwards from 1 until the perimeter exceeds B, i.e., $2m^2 + 2mn > B$, only considering those n that are co-prime to m so as to guarantee uniqueness of primitive solutions. Furthermore, with each primitive triplet generated, we will find all non-primitive triplets with a perimeter less than B by simply multiplying the terms in each triplet with reasonable constants. The rest of the algorithm is obvious from the code below.

The following function, `findOptimalPerimeter` takes input `bound` and returns $p \leq$ `bound` such that the number of Pythagorean solutions that give perimeter p is maximised.

Calling `findOptimalPerimeter(1000)` returns 840.

```
#include <vector>
#include <set>
#include <algorithm>  // sort
using namespace std;

void generateAndSortTriplet(int m, int n, int& a, int& b, int& c);
```

```
bool checkCoprimality(int m, int n);
int euclid(int m, int n);

int findOptimalPerimeter(int bound) {
  // finds perimeter p <= bound with such that maximum number of triplets give perimeter p

  vector<vector<vector<int>>> pSolutions(bound + 1, vector<vector<int>> { });
    // pSolutions[p] is a vector of all triplets that sum to p

  // iterating m up to maximal value m can take given bound
  for (int m = 2; 2 * m * (m + 1) <= bound; m++) {
    // iterating n upto maximal value n can take given bound
    for (int n = 1; n < m && 2 * m * (m + n) <= bound; n++) {

      // checking whether m and p are coprime
      if (!checkCoprimality(m, n))
        continue;

      // generating and sorting triplet
      int a, b, c;
      generateAndSortTriplet(m, n, a, b, c);

      // generating perimeter
      int primitivePerimeter = a + b + c;

      // iterating over all triplets associated with primitive triplet
      for (int multiplier = 1; multiplier <= bound / primitivePerimeter; multiplier++) {

        // perimeter of regular triplet
        int perimeter = multiplier * primitivePerimeter;
        // finding current ordered triplet
        vector<int> currentTriplet = { a * multiplier, b * multiplier, c * multiplier };

        bool newSolution = true;  // we assume solution is novel by default

        // checking triplets already found
        for (vector<int> triplet : pSolutions[perimeter]) {
          // comparing triplets
          if (currentTriplet == triplet) {
            newSolution = false;
            break;
          }
        }
        // if current triplet is novel, insert it into pSolutions[perimeter]
        if (newSolution)
          pSolutions[perimeter].push_back(currentTriplet);
      }
    }
  }

```

```
  // finding optimal p
  int pOptimal = 0; // optimal p
  int pOptimalSolution = 0; // number of solutions for optimal p

  // iterating p over 1 through bound to find optimal p
  for (int p = 1; p <= bound; p++)
    if (pSolutions[p].size() > pOptimalSolution) {
      // better p found -> update pOptimal, pOptimalSolution
      pOptimal = p;
      pOptimalSolution = pSolutions[p].size();
    }

  return pOptimal;
}

bool checkCoprimality(int m, int n) {
  // checks whether m and n are co-prime (assumes m > n)
  if (euclid(m, n) == 1)
    return true;
  else return false;
}

void generateAndSortTriplet(int m, int n, int& a, int& b, int& c) {
  // generates triplets from m and n using Euclid's formula, sorts and stores into a, b, and c

  // generating and sorting current triplet
  vector<int> currentPrimitiveTriplet = {
    int(pow(m, 2) - pow(n, 2)),
    2 * m * n,
    int(pow(m, 2) + pow(n, 2))
  };
  sort(currentPrimitiveTriplet.begin(), currentPrimitiveTriplet.end());

  // extracting a, b, and c (ordered)
  a = currentPrimitiveTriplet[0];
  b = currentPrimitiveTriplet[1];
  c = currentPrimitiveTriplet[2];
}

int euclid(int a, int b) {
  // returns 1 iff a and b are coprime
  if (b == 0)
    return a;
  else return euclid(b, a % b);
}
```

Exercise 16 (Programming Exercise)**.** *It is possible to show that the square root of two can be expressed as an infinite continued fraction.*

$$\sqrt{2} \;=\; 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cdots}}}$$

By expanding this for the first four iterations, we get:

$$\begin{aligned}
1 + \frac{1}{2} &= \frac{3}{2} \\
1 + \frac{1}{2 + \frac{1}{2}} &= \frac{7}{5} \\
1 + \frac{1}{2 + \frac{1}{2+\frac{1}{2}}} &= \frac{17}{12} \\
1 + \frac{1}{2 + \frac{1}{2+\frac{1}{2+\frac{1}{2}}}} &= \frac{41}{29}
\end{aligned}$$

The next tree expansions are 99/70*,* 239/169*, and* 577/408*, but the eighth expansion,* 1393/985*, is the first example where the number of digits in the numerator exceeds the number of digits in the denominator.*

In the first one-thousand expansions, how many fractions (when put in lowest terms) contain a numerator with more digits than the denominator?

(This page might be helpful: https://en.wikipedia.org/wiki/Continued_fraction)

Solution. Let t_k be the k^{th} tree expansion for the $\sqrt{2}$. Furthermore, let n_k and d_k are the numerator and denominator of t_k (in lowest form) respectively. From the given iteration it is clear that, for any $t_k = n_k/d_k$, we have

$$t_{k+1} = 1 + \frac{1}{1 + t_k}$$

Plugging in $t_k = n_k/d_k$, we get

$$t_{k+1} = 1 + \frac{1}{1 + t_k} = 1 + \frac{d_k}{n_k + d_k} = \frac{n_k + 2d_k}{n_k + d_k}$$

We can therefore iteratively generate the first one-thousand tree expansions in linear time. We also show that no reduction is needed as we iterate over the continued fractions. In other words, if t_k is in lowest form (i.e., n_k and d_k are co-prime), t_{k+1} is also in lowest form without reducing n_{k+1} and d_{k+1}.

Let $t_k = n_k/d_k$ such that n_k and d_k are co-prime. In other words, there exists no integer $f > 1$ such that $n_k = fm$ and $d_k = fn$ for some m, $n > 0$. Assume towards contradiction that n_{k+1} and d_{k+1} are not co-prime, i.e., there exists an integer $f > 1$ such that $n_{k+1} = fm$ and $d_{k+1} = fn$ for some $m > n > 0$. (Note that $2n > m > n$, since $n_k/d_k = m/n$ is an approximation for $\sqrt{2}$ that is at least as precise as $t_1 = 3/2$). We therefore have

$$n_{k+1} = n_k + 2d_k = fm, \ d_{k+1} = n_k + d_k = fn$$

Plugging $n_k + d_k = fn$ into the first equation, we have

$$fn + d_k = fm \implies d_k = f(m - n) \implies f \text{ divides } d_k$$

Plugging $d_k = f(m - n)$ into the second equation, we have

$$n_k + f(m - n) = fn \implies n_k = f(2n - m) \implies f \text{ divides } n_k$$

We therefore know f is a common factor of n_k and d_k, which contradicts our assumption they are co-prime.

Therefore, since the first expansion 3/2 has co-prime numerator and denominator, by induction, every subsequent expansion will have a co-prime numerator and denominator. We may therefore compute the expansions without reducing them to lowest terms, since they are already in lowest terms when generated.

The following function, `expansionsWithLargeNumerators` takes input `nIterations` and returns the number of continued expansions in the first `nIterations` fractions that have numerators with more digits than their respective denominators (in reduced form).

Calling `expansionsWithLargeNumerators(1000)` returns 153.

```cpp
#include <vector>
using namespace std;
#include <boost/multiprecision/cpp_int.hpp> // for large ints
using namespace boost::multiprecision;

int expansionsWithLargeNumerators(int nIterations) {
  // returns the number of continued fraction expansions (in lowest form) of sqrt
    (2) that have more digits in the numerator than in the denominator

  cpp_int nk = 3; cpp_int dk = 2;

  int largeNumeratorCount = 0;  // we're starting at

  for (int itr = 2; itr <= nIterations; itr++) {
    // generating t_{k+1}
    cpp_int nkp1 = nk + 2 * dk;
    cpp_int dkp1 = nk + dk;

    nk = nkp1; dk = dkp1;

    if (nk.str().size() > dk.str().size())
      largeNumeratorCount++;
  }
  return largeNumeratorCount;
}
```

APPENDIX

Following is the function definition for the `findDivisorSum` function, which takes input `n` and updates a map of divisor sums with `n`'s own divisor sum.

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

// auxiliary function for exercises 12, 13
int findDivisorSum(int n, unordered_map<int, int>& divisorSums) {
  // updates divisorSums with the sum of divisors of n (less than n)

  // checking if divisor sum already found (unordered_map search is constant time)
  if (divisorSums.find(n) != divisorSums.end())
    return divisorSums[n];

  // divisor sum not already found -> find sum of all divisors

  int nDivisorSum = 0;
  int rootN = ceil(sqrt(n));

  // we iterate up to root n to find all divisors pairs
  for (int d = 1; d < rootN; d++)
    // if d divides n, add both divisors to divisorSums[n]
    if (n % d == 0)
      nDivisorSum += d + n / d;

  // checking for case where n is a perfect square
  if (pow(rootN, 2) == n)
    nDivisorSum += rootN;

  // subtracting n, since we only want sum of divisors less than n (and n was added as part of (1, n) pair)
  nDivisorSum -= n;

  // inserting divisor sum in map (unordered_map insertion is constant time)
  divisorSums[n] = nDivisorSum;

  return nDivisorSum;
}
```