

201A-MATH182-1 HW3

ESHAN UNIYAL

TOTAL POINTS

55 / 85

QUESTION 1

1 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 2 pts Bounds not asymptotically tight

QUESTION 2

2 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 2 pts Incorrect
- 2 pts Incomplete

QUESTION 3

3 0 / 5

- 0 pts Correct
- ✓ - 5 pts No submission
- 1 pts Error

QUESTION 4

4 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 1 pts Error

QUESTION 5

5 5 / 5

- ✓ - 0 pts Correct
 - 5 pts No submission
 - 1 pts Error
 - 2 pts Invalid induction
- 💬 If you fix \$\$c\$\$, then you lose the ability to handle base cases by increasing \$\$c\$\$. So, e.g., if you had the base case $T(1)=1000$, then this proof would not be valid.

QUESTION 6

6 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 1 pts Fail to re-substitute at the end
- 2 pts Wrong asymptotics for transformed recurrence
- 1 pts Power error

QUESTION 7

7 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 2 pts Missing substitution method verification
- 3 pts Incomplete

QUESTION 8

8 0 / 5

- 0 pts Correct
- ✓ - 5 pts No submission
- 3 pts Recursion tree, no bound or proof
- 1.5 pts No substitution method proof
- 0.5 pts Substitution proof flawed
- 2.5 pts Wrong bound
- 2 pts Incorrect calculation, no substitution method proof

QUESTION 9

9 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 1 pts Incorrect Part 4
- 1 pts Incorrect Part 3
- 4 pts All incorrect

QUESTION 10

10 0 / 5

- 0 pts Correct
- ✓ - 5 pts No submission
 - 1 pts Insufficient reason in (1)
 - 1 pts Fail to identify all good chips in (3)
 - 1.5 pts Incomplete reduction in (2); don't identify all good chips in (3)
 - 4 pts No submission for (2) or (3); insufficient reason for (1)

QUESTION 11

11 0 / 5

- 0 pts Correct
- ✓ - 5 pts No submission
 - 2 pts Incorrect combine step
 - 2.5 pts Slow brute-force algorithm
 - 2 pts Unclear algorithm
 - 1 pts Incomplete solution

QUESTION 12

12 0 / 5

- 0 pts Correct
- ✓ - 5 pts No submission

QUESTION 13

13 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 1 pts Incorrect answer, code almost correct
- 1 pts Code probably correct, but no answer provided

QUESTION 14

14 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 1 pts No answer
- 0 pts Code would take an infeasible amount of time to run
 - 0 pts Code doesn't return answer
 - 💡 This algorithm has running time $\Omega(\log^2 n)$

$n^2 \log n$. Using the Sieve of Eratosthenes for faster prime-finding, you can bring that down to $O(\log n \log \log n)$. The brute-force algorithm is $\tilde{\Theta}(n^2)$ (i.e. n^2 up to logarithmic factors).

QUESTION 15

15 5 / 5

- 5 pts No submission
- ✓ - 0 pts Correct
 - 1 pts Code appears correct, no answer provided
 - 2 pts Incorrect

QUESTION 16

16 0 / 5

- 0 pts Correct
- ✓ - 5 pts No submission
 - 0 pts Generally invalid assumption, correct answer
 - 1 pts Generally invalid assumption, no answer
 - 2 pts Incorrect
 - 2.5 pts Binary method only

QUESTION 17

17 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 0.5 pts Omit diagonals
- 2 pts Solved a different problem

MATH182 HOMEWORK #3
DUE July 12, 2020

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. We first find an upper bound on $\lg(n!)$:

$$\lg(n!) = \sum_{k=1}^n \lg k \leq \sum_{k=1}^n \lg n = n \lg n \implies \lg(n!) = \underline{O(n \lg n)}$$

We now prove the same $\Omega(n \lg n)$ is also a lower bound on $\lg(n!)$:

$$(1) \quad \lg(n!) = \sum_{k=1}^n \lg k \geq \sum_{k=n/2}^n \lg k \geq \sum_{k=n/2}^n \lg \frac{n}{2} \geq \frac{n}{2} \cdot \lg \frac{n}{2} = \frac{n}{2} \cdot (\lg n - 1)$$

For $n \geq 4$, we know $\lg n \geq 2$. We therefore have, for $n \geq 4$,

$$(2) \quad \lg n - 2 \geq 0 \implies 2 \lg n - 2 \geq \lg n \implies \frac{n}{4} \cdot (2 \lg n - 2) \geq \frac{n}{4} \lg n \implies \frac{n}{2} \cdot (\lg n - 1) \geq \frac{n}{4} \lg n$$

Therefore, from (1) and (2), we also have

$$\lg(n!) \geq \frac{n}{2} \cdot (\lg n - 1) \geq \frac{n}{4} \lg n = \underline{\Omega(n \lg n)}$$

$\Theta(n \lg n)$ is therefore an asymptotically tight bound on $\lg(n!)$.

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) We note that a maximum subarray ending at $j + 1$ is either an extension of the maximum subarray ending at j (if the sum of that subarray is positive) or simply the subarray $A[j + 1]$. We can then extend the given observation as follows: a maximum subarray of $A[1..j + 1]$ is either a maximum subarray of $A[1..j]$, a maximum subarray ending at j with the addition of $A[j + 1]$, or simply the subarray $A[j + 1]$. Following is pseudocode¹ for an algorithm that applies this observation to find the maximal subarray of $A[1..n]$ (for $n := A.length$) in linear time:

MAXSUBARRAY(A) :

¹See Appendix I for C++ code

1 5 / 5

✓ - 0 pts Correct

- 5 pts No submission

- 2 pts Bounds not asymptotically tight

MATH182 HOMEWORK #3
DUE July 12, 2020

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. We first find an upper bound on $\lg(n!)$:

$$\lg(n!) = \sum_{k=1}^n \lg k \leq \sum_{k=1}^n \lg n = n \lg n \implies \lg(n!) = \underline{O(n \lg n)}$$

We now prove the same $\Omega(n \lg n)$ is also a lower bound on $\lg(n!)$:

$$(1) \quad \lg(n!) = \sum_{k=1}^n \lg k \geq \sum_{k=n/2}^n \lg k \geq \sum_{k=n/2}^n \lg \frac{n}{2} \geq \frac{n}{2} \cdot \lg \frac{n}{2} = \frac{n}{2} \cdot (\lg n - 1)$$

For $n \geq 4$, we know $\lg n \geq 2$. We therefore have, for $n \geq 4$,

$$(2) \quad \lg n - 2 \geq 0 \implies 2 \lg n - 2 \geq \lg n \implies \frac{n}{4} \cdot (2 \lg n - 2) \geq \frac{n}{4} \lg n \implies \frac{n}{2} \cdot (\lg n - 1) \geq \frac{n}{4} \lg n$$

Therefore, from (1) and (2), we also have

$$\lg(n!) \geq \frac{n}{2} \cdot (\lg n - 1) \geq \frac{n}{4} \lg n = \underline{\Omega(n \lg n)}$$

$\Theta(n \lg n)$ is therefore an asymptotically tight bound on $\lg(n!)$.

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) We note that a maximum subarray ending at $j + 1$ is either an extension of the maximum subarray ending at j (if the sum of that subarray is positive) or simply the subarray $A[j + 1]$. We can then extend the given observation as follows: a maximum subarray of $A[1..j + 1]$ is either a maximum subarray of $A[1..j]$, a maximum subarray ending at j with the addition of $A[j + 1]$, or simply the subarray $A[j + 1]$. Following is pseudocode¹ for an algorithm that applies this observation to find the maximal subarray of $A[1..n]$ (for $n := A.length$) in linear time:

MAXSUBARRAY(A) :

¹See Appendix I for C++ code

```

1  i2j_startIndex = 1          // starting index of maximal subarray of form  $A[i..j]$  for some  $i$ 
2  i2j_sum = A[1]             // sum of maximal subarray of form  $A[i..j]$ 
3  one2j_startIndex = 1       // starting index of maximal subarray of  $A[1..j]$ 
4  one2j_endIndex = 1         // ending index of maximal subarray of  $A[1..j]$ 
5  one2j_sum = A[1]           // sum of maximal subarray of  $A[1..j]$ 
6  for  $j = 1$  to  $A.length - 1$ 
7      // updating  $i2j\_sum$ ,  $startIndex$  for  $j + 1$  (finding max subarray ending at  $j + 1$ )
8      if  $i2j\_sum > 0$ 
9          // max subarray ending at  $j + 1$  is max subarray ending at  $j$ , plus  $A[j + 1]$ 
10          $i2j\_sum = i2j\_sum + A[j + 1]$ 
11     else
12         // max subarray ending at  $j + 1$  is simply  $A[j + 1]$ 
13          $i2j\_startIndex = j + 1$ 
14          $i2j\_sum = A[j + 1]$ 
15     // updating  $one2j\_startIndex$ ,  $endIndex$  and  $sum$  (finding max subarray of  $A[1..j + 1]$ )
16     if  $i2j\_sum > one2j\_sum$ 
17         // max subarray of  $A[1..j + 1]$  is max subarray ending at  $j + 1$ 
18          $one2j\_startIndex = i2j\_startIndex$ 
19          $one2j\_endIndex = j + 1$ 
20          $one2j\_sum = i2j\_sum$ 
21     // else, max subarray of  $A[1..j + 1]$  is max subarray of  $A[1..j]$ 
22     // no need to update variables
23 return  $one2j\_startIndex$ ,  $one2j\_endIndex$ ,  $one2j\_sum$ 

```

(2) We prove correctness of this algorithm by using the following loop invariant:

Loop Invariant: After line 6 is run, $one2j_startIndex$ and $one2j_endIndex$ are the starting and ending indices of the maximal subarray of $A[1..j]$, and $one2j_sum$ is the sum of the same. Additionally, $i2j_startIndex$ is the starting index of the maximal subarray of the form $A[i..j]$ for some $1 \leq i \leq j$, and $i2j_sum$ is the sum of the same.

We show this loop invariant holds.

Initialisation: When line 6 is first run, we have $j = 1$. Both the maximal subarray of the form $A[i..j]$ and the maximal subarray of $A[1..j]$ are simply $A[1]$, with starting and ending indices 1 and sum $A[1]$. $i2j_startIndex$, $i2j_sum$, $one2j_startIndex$, $one2j_endIndex$, and $one2j_sum$ are therefore all appropriately initialised from lines 1-5 and the loop invariant holds for $j = 1$.

Maintenance: Assume the loop invariant is true before some iteration with $j = j_0 \in [1..A.length - 1]$. By assumption and definition of the loop invariant, we have

$$\begin{aligned}
&i2j_startIndex = \text{starting index of maximal subarray ending at } j_0 \\
&i2j_sum = \text{sum of maximal subarray ending at } j_0 \\
&one2j_startIndex = \text{starting index of maximal subarray of } A[1..j_0] \\
&one2j_endIndex = \text{ending index of maximal subarray of } A[1..j_0] \\
&one2j_sum = \text{sum of maximal subarray of } A[1..j_0]
\end{aligned}$$

We first consider how we find a maximal subarray ending at $j_0 + 1$. We have two cases:

Case 1: In this case, the sum of a maximal subarray ending at j_0 , given by $i2j_sum$, is positive, and the **if** condition in line 8 holds true. A maximal subarray ending at $j_0 + 1$ is therefore given by the same maximal subarray with $A[j_0 + 1]$ included. We therefore leave $i2j_startIndex$ unchanged, and correctly update $i2j_sum$ by adding $A[j_0 + 1]$ to it in line 10.

Case 2: In this case, the sum of a maximal subarray ending at j_0 , given by `i2j_sum`, is non-positive, and we run the `else` block comprising lines 13 and 14. Since $i2j_sum + A[j_0+1] \leq A[j_0+1]$, $A[j_0+1]$ is a maximal subarray ending at j_0+1 . In line 13, we therefore correctly set the starting index of the maximal subarray ending at j_0+1 , `i2j_startIndex`, to j_0+1 ; similarly in line 14, we correctly set the sum of the same subarray to $A[j_0+1]$.

In either case, then, `i2j_startIndex` and `i2j_sum` reflect the starting index and sum of a maximal subarray ending at j_0+1 respectively.

We now consider how we find a maximal subarray of $A[1..j_0+1]$. We have two cases:

Case 1: A maximal subarray of $A[1..j_0+1]$ is also a maximal subarray of $A[1..j_0]$. In this case, we may leave `one2j_startIndex`, `one2j_endIndex`, and `one2j_sum` may be left unchanged.

Case 2: A maximal subarray of $A[1..j_0+1]$ is not a maximal subarray of $A[1..j_0]$. This implies a maximal subarray of $A[1..j_0+1]$ includes j_0+1 ; in other words, it ends at j_0+1 . We have already found such a maximal array in lines 7 through 14, and test for this case by comparing `i2j_sum` (the sum of a maximal subarray ending at j_0+1) with `one2j_sum` (the sum of a maximal subarray of $A[1..j]$). This case applies when the former is larger.

We therefore correctly update, in lines 18 and 19, `one2j_startIndex` and `one2j_endIndex` to the starting index of the maximal subarray ending at j_0+1 , `i2j_startIndex`, and the ending index of the same array, j_0+1 , respectively. We also correctly update, in line 20, the variable `one2j_sum` to the sum of the maximal subarray ending at j_0+1 , `i2j_sum`.

After completing the iteration and running line 6 again, we now have $j = j_0+1$. We have shown the following now holds:

$$i2j_startIndex = \text{starting index of maximal subarray ending at } j_0+1 \quad (\text{from lines 7-14})$$

$$i2j_sum = \text{sum of maximal subarray ending at } j_0+1 \quad (\text{from lines 7-14})$$

$$one2j_startIndex = \text{starting index of maximal subarray of } A[1..j_0+1] \quad (\text{from lines 15-22})$$

$$one2j_endIndex = \text{ending index of maximal subarray of } A[1..j_0+1] \quad (\text{from lines 15-22})$$

$$one2j_sum = \text{sum of maximal subarray of } A[1..j_0+1] \quad (\text{from lines 15-22})$$

Therefore, we have, by its definition, the loop invariant holds for $j = j_0+1$.

Termination: After the last iteration of the `for` loop, we have $j = A.length - 1 + 1 = A.length$, and therefore $A[1..j] = A[1..A.length] = A$. By the loop invariant, we have

$$one2j_startIndex = \text{starting index of maximal subarray of } A$$

$$one2j_endIndex = \text{ending index of maximal subarray of } A$$

$$one2j_sum = \text{sum of maximal subarray of } A$$

which are the desired outputs of `MAXSUBARRAY(A)` and are returned in line 23.

The algorithm is therefore correct. ■

(3) Analysing the running time of the algorithm is rather simple — since lines 1-5 and 15 run in constant time and only once, and lines 6 through 15 all run in constant time and a maximum of $n := A.length$ times (by scope of the `for` loop), the algorithm is $\Theta(n)$.

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.

2 5 / 5

- ✓ - 0 pts Correct
- 5 pts No submission
- 2 pts Incorrect
- 2 pts Incomplete

3 0 / 5

- 0 pts Correct
- ✓ - 5 pts No submission
- 1 pts Error

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.

Solution. Given imaginary numbers $a + bi$, $c + di$, we first compute the following three products:

$$k_1 := a \cdot (c + d), \quad k_2 := d \cdot (b + a), \quad k_3 := b \cdot (d - c)$$

We now show these products are sufficient to find the real and imaginary components of the product of given imaginary numbers. We first show $k_1 - k_2$ gives the real component:

$$k_1 - k_2 = a \cdot (c + d) - d \cdot (b + a) = ac + ad - bd - ad = \underline{ac - bd}$$

We now show $k_2 - k_3$ gives the imaginary component:

$$k_2 - k_3 = d \cdot (b + a) - b \cdot (d - c) = bd + ad - bd + bc = \underline{ad + bc}$$

Hence shown, using the above algorithm, that three multiplications of real numbers are sufficient to generate the real and imaginary components of a product of any two imaginary numbers.

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

Solution. We show, using the substitution method, that $T(n) = O(n^2)$. Assume inductively that there is some $d > 0, c > 0$ such that $T(m) \leq cm^2 - dm$ for all $m < n$. We then have:

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

where the last inequality applies if

$$dn \leq (2c + d - 1)n - (c + d) \implies 0 \leq (2c - 1)n - (c + d)$$

We see that for all $n \geq 2$, this is indeed true for $c = 1$ and $d = 1$, since then

$$(2c - 1)n - (c + d) = n - 2 \geq 0$$

Since this inductive proof will work for $c = d = 1$, we therefore have $\underline{T(n) = O(n^2)}$. ■

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. We define $m := \log_k n$ where k is the base of the log in the given recurrence relation. We can therefore rewrite the recurrence as:

$$\begin{aligned} T(n) = 3T(\sqrt{n}) + \log n &\implies T(k^m) = 3T(\sqrt{k^m}) + m && (m = \log_k n \implies n = k^m) \\ &\implies T(k^m) = 3T(k^{m/2}) + m \\ &\implies S(m) = 3S(m/2) + m && \text{(for } S(m) := T(k^m)\text{)} \end{aligned}$$

We now apply Master Theorem to find a tight asymptotic bound for $S(m)$. From the above recurrence relation, we have $a = 3$, $b = 2$, and $f(m) = m$. Since $\log_b a = \lg 3 > 1$, we have $f(m) = m^1 = O(m^{\log_b a - \epsilon})$ for any $\epsilon \in (0, \lg 3 - 1]$. We are therefore in the leaf-heavy case of the Master Theorem and have $S(m) = \Theta(m^{\log_b a}) = \Theta(m^{\lg 3})$. By definition of $S(m)$ and m , we have

$$T(n) = T(k^m) = S(m) = \Theta(m^{\lg 3}) = \underline{\Theta(\log_k^{\lg 3} n)}$$

4 5 / 5

✓ - 0 pts Correct

- 5 pts No submission

- 1 pts Error

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.

Solution. Given imaginary numbers $a + bi$, $c + di$, we first compute the following three products:

$$k_1 := a \cdot (c + d), \quad k_2 := d \cdot (b + a), \quad k_3 := b \cdot (d - c)$$

We now show these products are sufficient to find the real and imaginary components of the product of given imaginary numbers. We first show $k_1 - k_2$ gives the real component:

$$k_1 - k_2 = a \cdot (c + d) - d \cdot (b + a) = ac + ad - bd - ad = \underline{ac - bd}$$

We now show $k_2 - k_3$ gives the imaginary component:

$$k_2 - k_3 = d \cdot (b + a) - b \cdot (d - c) = bd + ad - bd + bc = \underline{ad + bc}$$

Hence shown, using the above algorithm, that three multiplications of real numbers are sufficient to generate the real and imaginary components of a product of any two imaginary numbers.

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

Solution. We show, using the substitution method, that $T(n) = O(n^2)$. Assume inductively that there is some $d > 0, c > 0$ such that $T(m) \leq cm^2 - dm$ for all $m < n$. We then have:

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

where the last inequality applies if

$$dn \leq (2c + d - 1)n - (c + d) \implies 0 \leq (2c - 1)n - (c + d)$$

We see that for all $n \geq 2$, this is indeed true for $c = 1$ and $d = 1$, since then

$$(2c - 1)n - (c + d) = n - 2 \geq 0$$

Since this inductive proof will work for $c = d = 1$, we therefore have $\underline{T(n) = O(n^2)}$. ■

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. We define $m := \log_k n$ where k is the base of the log in the given recurrence relation. We can therefore rewrite the recurrence as:

$$\begin{aligned} T(n) = 3T(\sqrt{n}) + \log n &\implies T(k^m) = 3T(\sqrt{k^m}) + m && (m = \log_k n \implies n = k^m) \\ &\implies T(k^m) = 3T(k^{m/2}) + m \\ &\implies S(m) = 3S(m/2) + m && \text{(for } S(m) := T(k^m)\text{)} \end{aligned}$$

We now apply Master Theorem to find a tight asymptotic bound for $S(m)$. From the above recurrence relation, we have $a = 3$, $b = 2$, and $f(m) = m$. Since $\log_b a = \lg 3 > 1$, we have $f(m) = m^1 = O(m^{\log_b a - \epsilon})$ for any $\epsilon \in (0, \lg 3 - 1]$. We are therefore in the leaf-heavy case of the Master Theorem and have $S(m) = \Theta(m^{\log_b a}) = \Theta(m^{\lg 3})$. By definition of $S(m)$ and m , we have

$$T(n) = T(k^m) = S(m) = \Theta(m^{\lg 3}) = \underline{\Theta(\log_k^{\lg 3} n)}$$

5 5 / 5

✓ - 0 pts Correct

- 5 pts No submission

- 1 pts Error

- 2 pts Invalid induction

💬 If you fix $\$c$, then you lose the ability to handle base cases by increasing $\$c$. So, e.g., if you had the base case $T(1)=1000$, then this proof would not be valid.

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.

Solution. Given imaginary numbers $a + bi$, $c + di$, we first compute the following three products:

$$k_1 := a \cdot (c + d), \quad k_2 := d \cdot (b + a), \quad k_3 := b \cdot (d - c)$$

We now show these products are sufficient to find the real and imaginary components of the product of given imaginary numbers. We first show $k_1 - k_2$ gives the real component:

$$k_1 - k_2 = a \cdot (c + d) - d \cdot (b + a) = ac + ad - bd - ad = \underline{ac - bd}$$

We now show $k_2 - k_3$ gives the imaginary component:

$$k_2 - k_3 = d \cdot (b + a) - b \cdot (d - c) = bd + ad - bd + bc = \underline{ad + bc}$$

Hence shown, using the above algorithm, that three multiplications of real numbers are sufficient to generate the real and imaginary components of a product of any two imaginary numbers.

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

Solution. We show, using the substitution method, that $T(n) = O(n^2)$. Assume inductively that there is some $d > 0, c > 0$ such that $T(m) \leq cm^2 - dm$ for all $m < n$. We then have:

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

where the last inequality applies if

$$dn \leq (2c + d - 1)n - (c + d) \implies 0 \leq (2c - 1)n - (c + d)$$

We see that for all $n \geq 2$, this is indeed true for $c = 1$ and $d = 1$, since then

$$(2c - 1)n - (c + d) = n - 2 \geq 0$$

Since this inductive proof will work for $c = d = 1$, we therefore have $\underline{T(n) = O(n^2)}$. ■

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. We define $m := \log_k n$ where k is the base of the log in the given recurrence relation. We can therefore rewrite the recurrence as:

$$\begin{aligned} T(n) = 3T(\sqrt{n}) + \log n &\implies T(k^m) = 3T(\sqrt{k^m}) + m && (m = \log_k n \implies n = k^m) \\ &\implies T(k^m) = 3T(k^{m/2}) + m \\ &\implies S(m) = 3S(m/2) + m && \text{(for } S(m) := T(k^m)\text{)} \end{aligned}$$

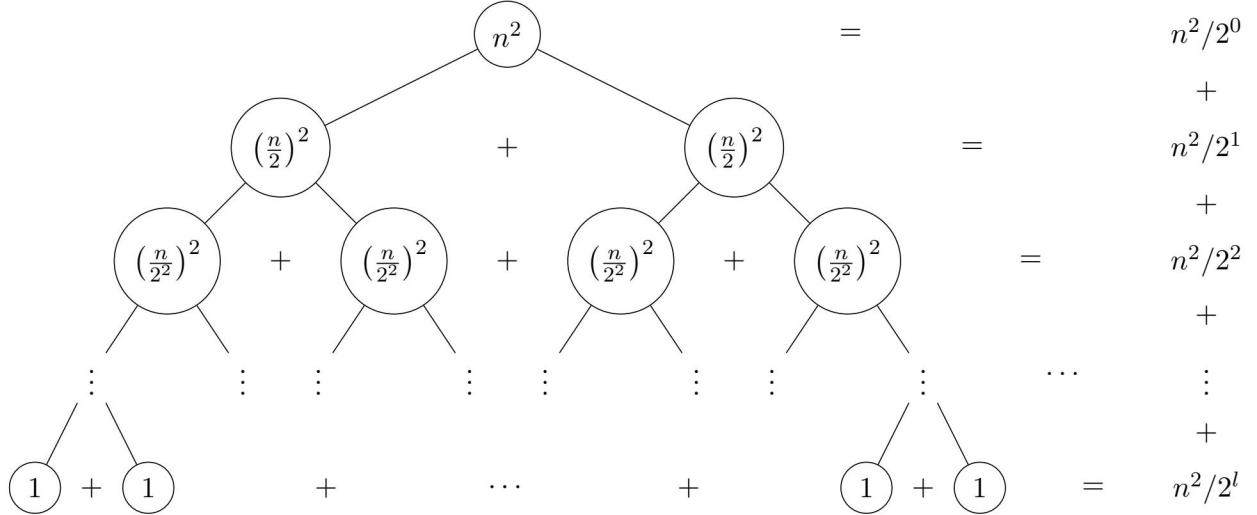
We now apply Master Theorem to find a tight asymptotic bound for $S(m)$. From the above recurrence relation, we have $a = 3$, $b = 2$, and $f(m) = m$. Since $\log_b a = \lg 3 > 1$, we have $f(m) = m^1 = O(m^{\log_b a - \epsilon})$ for any $\epsilon \in (0, \lg 3 - 1]$. We are therefore in the leaf-heavy case of the Master Theorem and have $S(m) = \Theta(m^{\log_b a}) = \Theta(m^{\lg 3})$. By definition of $S(m)$ and m , we have

$$T(n) = T(k^m) = S(m) = \Theta(m^{\lg 3}) = \underline{\Theta(\log_k^{\lg 3} n)}$$

which is a tight asymptotic bound on $T(n)$.

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. Below is a recursion tree for the given recurrence relation:



On each level k , we call $T(n/2^k)$ (with cost $(n/2^k)^2$ at that level) 2^k times, giving a total cost per level of

$$\left(\frac{n}{2^k}\right)^2 \cdot 2^k = \frac{n^2}{2^k}$$

Assuming running $T(1)$ costs 1, the leaves of the recursion tree each represent a $T(1)$ call. In the last layer l , we therefore have $T(n/2^l) = T(1) \implies n/2^l = 1 \implies l = \lg n$. Our guess of the total cost of $T(n)$ is therefore given by

$$T(n) = \sum_{k=0}^l \frac{n^2}{2^k} = n^2 \sum_{k=0}^{\lg n} \frac{1}{2^k} \leq n^2 \sum_{k=0}^{\infty} \frac{1}{2^k} = n^2 \cdot \frac{1}{1 - 1/2} = 2n^2$$

We now use substitution method to verify that $T(n) = O(n^3)$. Assume inductively that there is some $c > 0$ such that $T(m) \leq cm^2$ for all $m < n$. We then have:

$$\begin{aligned}
 T(n) = T(n/2) + n^2 &\leq c \left(\frac{n}{2}\right)^2 + n^2 && \text{(by inductive assumption)} \\
 &= \left(\frac{c}{4} + 1\right)n^2 \\
 &\leq cn^2 && \text{(for any } c \geq 4/3\text{)}
 \end{aligned}$$

Since this inductive proof will work for any value of $c > 4/3$ and any $n \geq 0$ (assuming $T(0) = 0$), we therefore have $T(n) = O(n^2)$.

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:

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

6 5 / 5

✓ - 0 pts Correct

- 5 pts No submission

- 1 pts Fail to re-substitute at the end

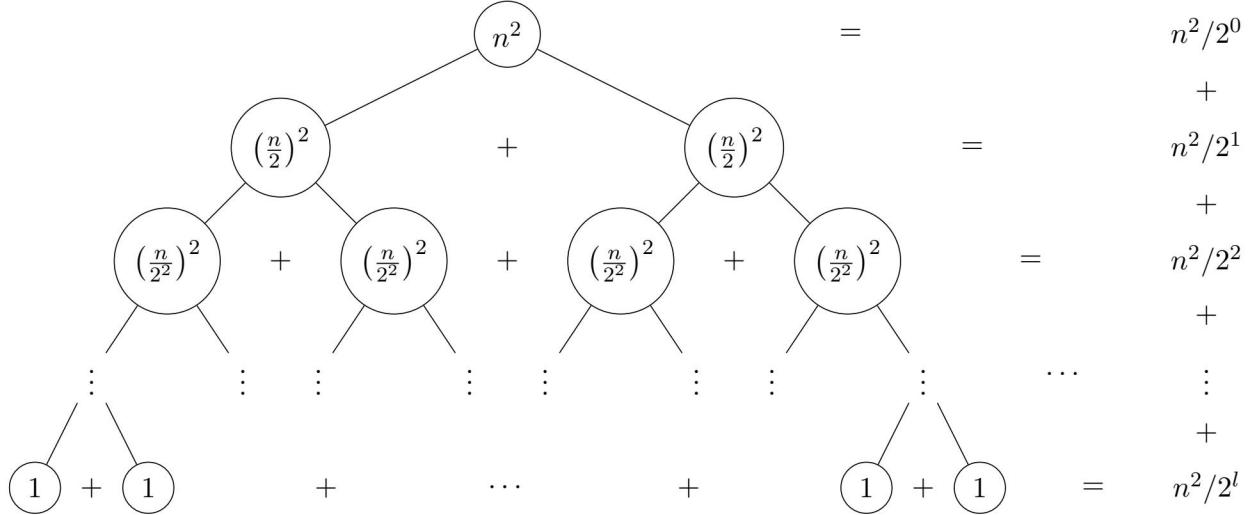
- 2 pts Wrong asymptotics for transformed recurrence

- 1 pts Power error

which is a tight asymptotic bound on $T(n)$.

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. Below is a recursion tree for the given recurrence relation:



On each level k , we call $T(n/2^k)$ (with cost $(n/2^k)^2$ at that level) 2^k times, giving a total cost per level of

$$\left(\frac{n}{2^k}\right)^2 \cdot 2^k = \frac{n^2}{2^k}$$

Assuming running $T(1)$ costs 1, the leaves of the recursion tree each represent a $T(1)$ call. In the last layer l , we therefore have $T(n/2^l) = T(1) \implies n/2^l = 1 \implies l = \lg n$. Our guess of the total cost of $T(n)$ is therefore given by

$$T(n) = \sum_{k=0}^l \frac{n^2}{2^k} = n^2 \sum_{k=0}^{\lg n} \frac{1}{2^k} \leq n^2 \sum_{k=0}^{\infty} \frac{1}{2^k} = n^2 \cdot \frac{1}{1 - 1/2} = 2n^2$$

We now use substitution method to verify that $T(n) = O(n^3)$. Assume inductively that there is some $c > 0$ such that $T(m) \leq cm^2$ for all $m < n$. We then have:

$$\begin{aligned}
 T(n) = T(n/2) + n^2 &\leq c \left(\frac{n}{2}\right)^2 + n^2 && \text{(by inductive assumption)} \\
 &= \left(\frac{c}{4} + 1\right)n^2 \\
 &\leq cn^2 && \text{(for any } c \geq 4/3\text{)}
 \end{aligned}$$

Since this inductive proof will work for any value of $c > 4/3$ and any $n \geq 0$ (assuming $T(0) = 0$), we therefore have $T(n) = O(n^2)$.

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:

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

7 5 / 5

✓ - 0 pts Correct

- 5 pts No submission

- 2 pts Missing substitution method verification

- 3 pts Incomplete

8 0 / 5

- 0 pts Correct

✓ - 5 pts No submission

- 3 pts Recursion tree, no bound or proof

- 1.5 pts No substitution method proof

- 0.5 pts Substitution proof flawed

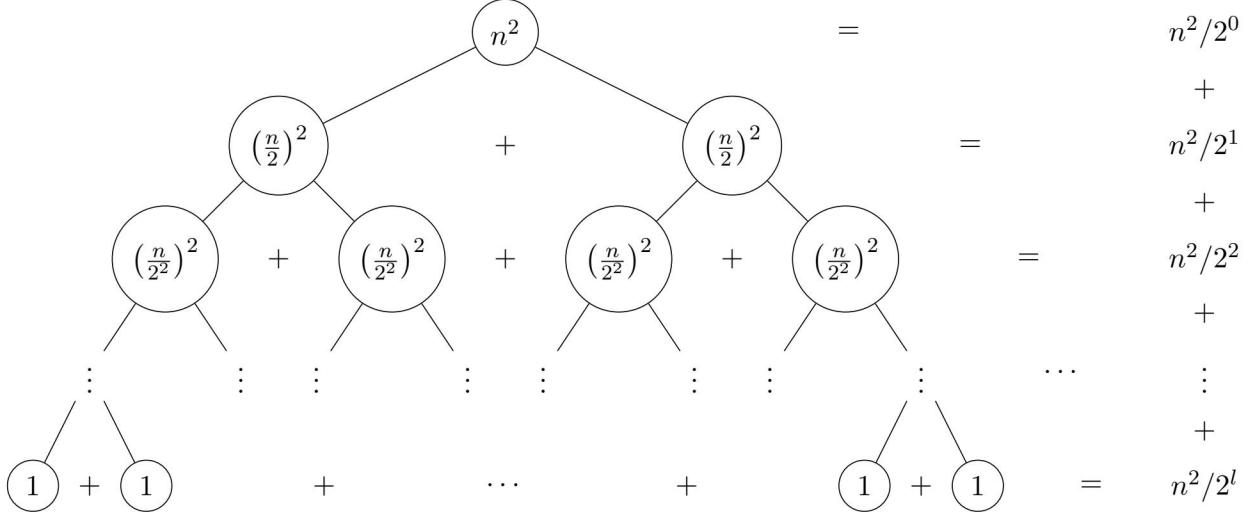
- 2.5 pts Wrong bound

- 2 pts Incorrect calculation, no substitution method proof

which is a tight asymptotic bound on $T(n)$.

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. Below is a recursion tree for the given recurrence relation:



On each level k , we call $T(n/2^k)$ (with cost $(n/2^k)^2$ at that level) 2^k times, giving a total cost per level of

$$\left(\frac{n}{2^k}\right)^2 \cdot 2^k = \frac{n^2}{2^k}$$

Assuming running $T(1)$ costs 1, the leaves of the recursion tree each represent a $T(1)$ call. In the last layer l , we therefore have $T(n/2^l) = T(1) \implies n/2^l = 1 \implies l = \lg n$. Our guess of the total cost of $T(n)$ is therefore given by

$$T(n) = \sum_{k=0}^l \frac{n^2}{2^k} = n^2 \sum_{k=0}^{\lg n} \frac{1}{2^k} \leq n^2 \sum_{k=0}^{\infty} \frac{1}{2^k} = n^2 \cdot \frac{1}{1 - 1/2} = 2n^2$$

We now use substitution method to verify that $T(n) = O(n^3)$. Assume inductively that there is some $c > 0$ such that $T(m) \leq cm^2$ for all $m < n$. We then have:

$$\begin{aligned}
 T(n) &= T(n/2) + n^2 \leq c \left(\frac{n}{2}\right)^2 + n^2 && \text{(by inductive assumption)} \\
 &= \left(\frac{c}{4} + 1\right) n^2 \\
 &\leq cn^2 && \text{(for any } c \geq 4/3\text{)}
 \end{aligned}$$

Since this inductive proof will work for any value of $c > 4/3$ and any $n \geq 0$ (assuming $T(0) = 0$), we therefore have $T(n) = O(n^2)$.

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:

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

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

Solution. For each of the given recurrence relations, we have $a = 2$ and $b = 4$. We therefore have $\log_b a = 1/2$.

- (1) For the recurrence $T(n) = 2T(n/4) + 1$, we have $f(n) = 1$. Since $f(n) = 1 = O(n^0) = O(n^{1/2-\epsilon})$ for $\epsilon \in (0, 1/2]$, we have $f(n) = O(n^{\log_b a-\epsilon})$ and are therefore in the leaf-heavy case of Master Theorem. $T(n)$ is therefore $\Theta(n^{\log_b a}) = \Theta(\sqrt{n})$.
- (2) For the recurrence $T(n) = 2T(n/4) + \sqrt{n}$, we have $f(n) = \sqrt{n}$. Since $f(n) = \sqrt{n} = \Theta(\sqrt{n}) = \Theta(n^{1/2}) = \Theta(n^{1/2} \lg^0 n)$, we have $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for $k = 0$ and are therefore in the middle case of Master Theorem. Since $k = 0 > -1$ $T(n)$ is therefore $\Theta(n^{\log_b a} \lg^{k+1} n) = \Theta(\sqrt{n} \lg n)$.
- (3) For the recurrence $T(n) = 2T(n/4) + n$, we have $f(n) = n$. Since $f(n) = n = \Omega(n^1) = \Omega(n^{1/2+\epsilon})$ for $\epsilon \in (0, 1/2]$, we have $f(n) = \Omega(n^{\log_b a+\epsilon})$ and are therefore in the root-heavy case of Master Theorem. We now test the Regularity Condition by searching for $c < 1$ such that $2f(n/4) \leq cf(n)$ for all $n \geq n_0$ for some $n_0 \in \mathbb{Z}$. In other words, we have

$$2f(n/4) \leq cf(n) \implies 2 \cdot \frac{n}{4} \leq cn \implies \frac{1}{2} \leq c$$

for all $n \geq 0$. We can therefore choose any $c \in [1/2, 1)$ to satisfy the Regularity Condition. $T(n)$ is therefore $\Theta(f(n)) = \Theta(n)$.

- (4) For the recurrence $T(n) = 2T(n/4) + n^2$, we have $f(n) = n^2$. Since $f(n) = n^2 = \Omega(n^2) = \Omega(n^{1/2+\epsilon})$ for $\epsilon \in (0, 3/2]$, we have $f(n) = \Omega(n^{\log_b a+\epsilon})$ and are therefore in the root-heavy case of Master Theorem. We now test the Regularity Condition by searching for $c < 1$ such that $2f(n/4) \leq cf(n)$ for all $n \geq n_0$ for some $n_0 \in \mathbb{Z}$. In other words, we have

$$2f(n/4) \leq cf(n) \implies 2 \cdot \left(\frac{n}{4}\right)^2 \leq cn^2 \implies \frac{1}{8} \leq c$$

for all $n \geq 0$. We can therefore choose any $c \in [1/8, 1)$ to satisfy the Regularity Condition. $T(n)$ is therefore $\Theta(f(n)) = \Theta(n^2)$.

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 which 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 .

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. The following function, `largeCombinations`, takes as input integers `minCombinations`, the lower-bound for the values of combinations we're searching for, and `nBound`, the upper bound for n in our search.

```

1 #include <vector>
2 using namespace std;
3
4 #include <boost/multiprecision/cpp_int.hpp>
5     // for large ints, from the non-standard boost library
6 using namespace boost::multiprecision;
7
8 int largeCombinations(int minCombinations, int nBound) {
9     // returns the number of values of (n r), with 0 <= r <= n <= nBound, that are
10    greater than minCombinations

```

9 5 / 5

✓ - 0 pts Correct

- 5 pts No submission

- 1 pts Incorrect Part 4

- 1 pts Incorrect Part 3

- 4 pts All incorrect

10 0 / 5

- 0 pts Correct

✓ - 5 pts No submission

- 1 pts Insufficient reason in (1)

- 1 pts Fail to identify all good chips in (3)

- 1.5 pts Incomplete reduction in (2); don't identify all good chips in (3)

- 4 pts No submission for (2) or (3); insufficient reason for (1)

11 0 / 5

- 0 pts Correct
- ✓ - 5 pts No submission
- 2 pts Incorrect combine step
- 2.5 pts Slow brute-force algorithm
- 2 pts Unclear algorithm
- 1 pts Incomplete solution

12 0 / 5

- 0 pts Correct
- ✓ - 5 pts No submission

(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 .

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. The following function, `largeCombinations`, takes as input integers `minCombinations`, the lower-bound for the values of combinations we're searching for, and `nBound`, the upper bound for n in our search.

```

1 #include <vector>
2 using namespace std;
3
4 #include <boost/multiprecision/cpp_int.hpp>
5     // for large ints, from the non-standard boost library
6 using namespace boost::multiprecision;
7
8 int largeCombinations(int minCombinations, int nBound) {
9     // returns the number of values of (n r), with 0 <= r <= n <= nBound, that are
10    greater than minCombinations

```

```

11 // computing a vector of factorials
12 vector<cpp_int> factorials = { 1 }; // initialised to 0!
13 // iterating to build factorials
14 for (int n = 1; n <= nBound; n++)
15     factorials.push_back(factorials.back() * n);
16
17 int count = 0; // number of combinations found that are greater than min
18
19 // iterating over all (n, r) for 0 <= n <= nBound
20 for (int n = 1; n <= nBound; n++) {
21     for (int r = 0; r <= n; r++) {
22         // computing nCr
23         cpp_int nCr = factorials[n] / (factorials[r] * factorials[n - r]);
24         // checking if nCr > min
25         if (nCr > minCombinations)
26             count++;
27     }
28
29 return count;
30 }
```

Calling `largeCombinations(1000000, 100)` returns 4075.

Exercise 14 (Programming exercise). Euler's Totient function, $\varphi(n)$ 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. Even with some optimisations, the brute force solution², using Euler's Product Formula, is quite inefficient and takes 10 seconds to run (in C++). We find a more efficient algorithm.

From Euler's Product Formula, we have

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right) = n \prod_{p|n} \left(\frac{p-1}{p}\right) \implies \frac{n}{\varphi(n)} = \prod_{p|n} \left(\frac{p}{p-1}\right)$$

where p is prime. In other words, for $f(p) := p/(p-1)$, the totient ratio (i.e., $n/\varphi(n)$) is given by the product of $f(p)$ over the set of primes that divide n . We approach the problem by searching

²C++ code in Appendix II

13 5 / 5

✓ - 0 pts Correct

- 5 pts No submission

- 1 pts Incorrect answer, code almost correct

- 1 pts Code probably correct, but no answer provided

```

11 // computing a vector of factorials
12 vector<cpp_int> factorials = { 1 }; // initialised to 0!
13 // iterating to build factorials
14 for (int n = 1; n <= nBound; n++)
15     factorials.push_back(factorials.back() * n);
16
17 int count = 0; // number of combinations found that are greater than min
18
19 // iterating over all (n, r) for 0 <= n <= nBound
20 for (int n = 1; n <= nBound; n++) {
21     for (int r = 0; r <= n; r++) {
22         // computing nCr
23         cpp_int nCr = factorials[n] / (factorials[r] * factorials[n - r]);
24         // checking if nCr > min
25         if (nCr > minCombinations)
26             count++;
27     }
28
29 return count;
30 }
```

Calling `largeCombinations(1000000, 100)` returns 4075.

Exercise 14 (Programming exercise). Euler's Totient function, $\varphi(n)$ 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. Even with some optimisations, the brute force solution², using Euler's Product Formula, is quite inefficient and takes 10 seconds to run (in C++). We find a more efficient algorithm.

From Euler's Product Formula, we have

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right) = n \prod_{p|n} \left(\frac{p-1}{p}\right) \implies \frac{n}{\varphi(n)} = \prod_{p|n} \left(\frac{p}{p-1}\right)$$

where p is prime. In other words, for $f(p) := p/(p-1)$, the totient ratio (i.e., $n/\varphi(n)$) is given by the product of $f(p)$ over the set of primes that divide n . We approach the problem by searching

²C++ code in Appendix II

for the set of primes that comprise a prime factorisation for some $n \leq 1000000$ such that the above product over that set is maximal.

Since $f(p)$ is strictly decreasing (easily verified by differentiating with respect to n), smaller prime factors yield a larger product. Furthermore, since $f(p) > 1$ for all primes p , each unique prime factor increases the totient ratio. We therefore want to maximise the number of unique factors (so more terms contribute to the product) and minimise the value of each unique prime factor (so each term contributes more to the product).

In fact, these goals go hand-in-hand — in minimising the value of each prime factor, we can only increase (non-strictly) the number of unique prime factors. Let $P_k := \{p_1, p_2, \dots, p_k\}$ where $p_1 = 2$ and each p_{i+1} is the prime immediately after p_i . We propose the value of n that gives a maximal totient ratio under a given bound B is given by the product over P_k where k is chosen such that

$$\prod_{p \in P_k} p \leq B < \prod_{p \in P_{k+1}} p$$

We show this proposition is true. Assume towards contradiction that $n := \prod_{p \in P_k} p$ does not give a maximal totient ratio. This means there exists $N \in [2, B]$ such that for the (sorted) set of prime factors of N , $PF(N)$, we have

$$\frac{N}{\varphi(N)} > \frac{n}{\varphi(n)} \implies \prod_{p \in PF(N)} f(p) > \prod_{p \in P_k} f(p)$$

From our discussion on the product of $f(p)$ over a set of primes, we know this implies one of the following two cases must be true:³

Case 1: $PF(N)$ has more terms than P_k , i.e., $|PF(N)| \geq k+1$. Since $PF(N)$ is (by assumption) sorted in increasing order, it is clear that, by definition of p_i , $PF(N)[i] \geq p_i$ for all $i \in [1, k+1]$. We therefore have

$$N \geq \prod_{p \in PF(N)} p \geq \prod_{1 \leq i \leq k+1} PF(N)[i] \geq \prod_{1 \leq i \leq k+1} p_i = \prod_{p \in P_k} p > B \quad (\text{by definition of } P_k)$$

which contradicts our assumption $N \leq B$. Case 1 is therefore not possible.

Case 2: $PF(N)$ has at most as many terms as P_k (i.e., $|PF(N)| \leq k$), and $f(PF[i]) > f(p_i)$ for some $i \in [1, |PF(N)|]$. From our discussion in Case 1, we already know $PF(N)[i] \geq p_i$ for all $i \in [1, |PF(N)|]$. Since $f(p)$ is strictly decreasing, this implies $f(PF(N)[i]) \leq f(p_i)$ for all $i \in [1, |PF(N)|]$, which contradicts our assumption there exists $i \in [1, |PF(N)|]$ such that $f(PF[i]) > f(p_i)$. Case 2 is therefore not possible.

Hence shown that value of $n \leq B$ such that the totient ratio is maximal is given by the maximal product over consecutive primes starting from 2, i.e., the product of P_k . ■

This results in the following algorithm, which runs in effectively constant time for even large n :

```

1 #include <vector>
2 using namespace std;
3
4 int maxEulerTotientRatio(int bound) {
5     // returns that maximum value of n / totient(n) for n <= bound
6
7     double totientRatio = 1; // to keep track of the optimal totient ratio
8     int n = 1; // to keep track of n that gives the optimal totient ratio
9

```

³Note that for any $N \in [2, B]$, if either case is true, that itself is not sufficient to say N has a higher totient ratio than n . However, if we assume N has a higher totient ratio, then one of the cases be true.

```

10 vector<int> primes;
11
12 // iterating over natural numbers to identify primes until n exceeds bound
13 for (int p = 2; n*p <= bound; p++) {
14     // checking if p is prime by iterating over primes
15     bool primeFound = true;
16     for (int prime : primes)
17         if (p % prime == 0) {
18             primeFound = false;
19             break;
20         }
21     // prime found
22     if (primeFound) {
23         primes.push_back(p); // update primes
24         totientRatio *= p / double(p - 1); // update totientRatio
25         n *= p; // update n
26     }
27 }
28 return n;
29 }
```

Calling `maxEulerTotientRatio(1000000)` returns 510510 (i.e., the product over $\{2, 3, 5, 7, 11, 13, 17\}$), which has totient ratio ≈ 5.53939 .

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

$$\begin{aligned}
 & 4 + 1 \\
 & 3 + 2 \\
 & 3 + 1 + 1 \\
 & 2 + 2 + 1 \\
 & 2 + 1 + 1 + 1 \\
 & 1 + 1 + 1 + 1 + 1
 \end{aligned}$$

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

Solution. The solution to this programming exercise can be obtained with a few simple modifications to the solution of Exercise 14 from Homework 2 — instead of searching for the number of partitions of $200p$ using $\{1p, 2p, 5p, 10p, 20p, 50p, 100p, 200p\}$, we are now searching for the number of partitions of 100 using $\{1, 2, \dots, 99\}$ (we exclude 100 since we don't want to include any partitions with only one integer). The following function, `nPartitions`, takes input integer n and returns the number of partitions of n with at least two positive integers:

```

1 #include <vector>
2 using namespace std;
3
4 int nPartitions(int n) {
5     // returns the number of partitions of n with at least two integers
6
7     vector<int> partitionSizes(n + 1, 0);
8     // a list containing, for each number from 1 through n, the number of partitions
9     // it has
10    // initialised to 0
```

14 5 / 5

✓ - 0 pts Correct

- 5 pts No submission

- 1 pts No answer

- 0 pts Code would take an infeasible amount of time to run

- 0 pts Code doesn't return answer

💬 This algorithm has running time $\Omega(\frac{\log^2 n}{\log \log n})$. Using the Sieve of Eratosthenes for faster prime-finding, you can bring that down to $O(\log n \log \log n)$. The brute-force algorithm is $\tilde{\Theta}(n^2)$ (i.e. n^2 up to logarithmic factors).

```

10 vector<int> primes;
11
12 // iterating over natural numbers to identify primes until n exceeds bound
13 for (int p = 2; n*p <= bound; p++) {
14     // checking if p is prime by iterating over primes
15     bool primeFound = true;
16     for (int prime : primes)
17         if (p % prime == 0) {
18             primeFound = false;
19             break;
20         }
21     // prime found
22     if (primeFound) {
23         primes.push_back(p); // update primes
24         totientRatio *= p / double(p - 1); // update totientRatio
25         n *= p; // update n
26     }
27 }
28 return n;
29 }
```

Calling `maxEulerTotientRatio(1000000)` returns 510510 (i.e., the product over $\{2, 3, 5, 7, 11, 13, 17\}$), which has totient ratio ≈ 5.53939 .

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

$$\begin{aligned}
 & 4 + 1 \\
 & 3 + 2 \\
 & 3 + 1 + 1 \\
 & 2 + 2 + 1 \\
 & 2 + 1 + 1 + 1 \\
 & 1 + 1 + 1 + 1 + 1
 \end{aligned}$$

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

Solution. The solution to this programming exercise can be obtained with a few simple modifications to the solution of Exercise 14 from Homework 2 — instead of searching for the number of partitions of $200p$ using $\{1p, 2p, 5p, 10p, 20p, 50p, 100p, 200p\}$, we are now searching for the number of partitions of 100 using $\{1, 2, \dots, 99\}$ (we exclude 100 since we don't want to include any partitions with only one integer). The following function, `nPartitions`, takes input integer n and returns the number of partitions of n with at least two positive integers:

```

1 #include <vector>
2 using namespace std;
3
4 int nPartitions(int n) {
5     // returns the number of partitions of n with at least two integers
6
7     vector<int> partitionSizes(n + 1, 0);
8     // a list containing, for each number from 1 through n, the number of partitions
9     // it has
10    // initialised to 0
```

```

11 // base case: partitionSize[0] represents the termination of a partition
12 partitionSizes[0] = 1;
13
14 // iterating over 1 through n - 1
15 for (int part = 1; part < n; part++)
16     // iterating from 1 through n
17     for (int i = 1; i <= n; i++)
18         // partitions(i) = partitions[i] + partitions[i - part]
19         // (for only parts that have already been processed for smaller i)
20         if (i - part >= 0)
21             partitionSizes[i] += partitionSizes[i - part];
22
23 // return partitions of n
24 return partitionSizes[n];
25 }
```

Calling `nPartitions(100)` 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 $\sum m(k)$.

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

15 5 / 5

- 5 pts No submission
- ✓ - 0 pts Correct
- 1 pts Code appears correct, no answer provided
- 2 pts Incorrect

16 0 / 5

- 0 pts Correct

✓ - 5 pts No submission

- 0 pts Generally invalid assumption, correct answer

- 1 pts Generally invalid assumption, no answer

- 2 pts Incorrect

- 2.5 pts Binary method only

```

11 // base case: partitionSize[0] represents the termination of a partition
12 partitionSizes[0] = 1;
13
14 // iterating over 1 through n - 1
15 for (int part = 1; part < n; part++)
16     // iterating from 1 through n
17     for (int i = 1; i <= n; i++)
18         // partitions(i) = partitions[i] + partitions[i - part]
19         // (for only parts that have already been processed for smaller i)
20         if (i - part >= 0)
21             partitionSizes[i] += partitionSizes[i - part];
22
23 // return partitions of n
24 return partitionSizes[n];
25 }
```

Calling `nPartitions(100)` 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 $\sum m(k)$.

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 $1 \leq k \leq 55$, $s_k = [100003 - 200003k + 300007k^3] \pmod{1000000} - 500000$.
- For $56 \leq k \leq 4000000$, $s_k = [s_{k-24} + s_{k-55} + 1000000] \pmod{1000000} - 500000$

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. We first generate the matrix of specified size $\text{sz} \times \text{sz}$ using the given Lagged Fibonacci Generator in the `generateMatrix` function. Then iterating over the matrix, we generate all the matrix's rows, columns, diagonals and anti-diagonals and call on each `checkArrayAndUpdateMax`, which uses the algorithm designed in Exercise 2 (see Appendix I for code) to find the sum of a maximal subarray of given vector.

```
1 #include <vector>
2 #include<iostream>
3 using namespace std;
4
5 #include <boost/multiprecision/cpp_int.hpp> // for large ints, from the non-
6   standard boost library
6 using namespace boost::multiprecision;
7
8 vector<vector<int>> generateMatrix(int sz);
9 vector<int> maxSubArray(const vector<int>& A);
10 // from exercise 2, for finding maximal subarray
11
12 void checkArrayAndUpdateMax(const vector<int>& arr, int& max) {
13   // checks input array for sum of maximal subarray (arrMax) and updates max if
14   // arrMax > max
15
16   // computing sum of maximal subarray
17   int arrMax = maxSubArray(arr)[2];
18   // updating max if necessary
19   if (arrMax > max)
20     max = arrMax;
21 }
22
23 int greatestAdjacentEntrySum(int sz) {
24   // returns the greatest sum of any number of adjacent entries (horizontal,
25   // vertical, diagonal, or anti-diagonal)
26   // in a matrix of size sz*sz (assumed sz <= 2000)
27
28   vector<vector<int>> matrix = generateMatrix(sz);
29
30   int max = matrix[0][0]; // tracks the greatest adjacent entry sum
31
32   // checking rows
33   for (int row = 0; row < sz; row++)
34     checkArrayAndUpdateMax(matrix[row], max);
```

```

33
34 // checking columns
35 for (int col = 0; col < sz; col++) {
36     // creating column vector
37     vector<int> column(sz, 0);
38
39     // iterating over rows to insert elements
40     for (int row = 0; row < sz; row++)
41         column[row] = matrix[row][col];
42
43     checkArrayAndUpdateMax(column, max);
44 }
45
46 // checking diagonals starting in top row
47 for (int startCol = 0; startCol < sz; startCol++) {
48
49     // creating diagonal vector
50     vector<int> diagonal; diagonal.reserve(sz);
51
52     // iterating over rows to insert elements
53     for (int rowNum = 0; startCol + rowNum < sz; rowNum++)
54         diagonal.push_back(matrix[rowNum][startCol + rowNum]);
55
56     checkArrayAndUpdateMax(diagonal, max);
57 }
58
59 // checking diagonals starting in leftmost column
60 for (int startRow = 1; startRow < sz; startRow++) {
61     // starting row at 1 since diagonal starting at [0][0] already checked
62     // creating diagonal vector
63     vector<int> diagonal; diagonal.reserve(sz);
64
65     // iterating over columns to insert elements
66     for (int colNum = 0; startRow + colNum < sz; colNum++)
67         diagonal.push_back(matrix[startRow + colNum][colNum]);
68
69     checkArrayAndUpdateMax(diagonal, max);
70 }
71
72
73 // checking anti-diagonals starting in top row
74 for (int startCol = sz - 1; startCol >= 0; startCol--) {
75
76     // creating anti-diagonal vector
77     vector<int> antiDiagonal; antiDiagonal.reserve(sz);
78
79     // iterating over rows to insert elements
80     for (int rowNum = 0; startCol - rowNum >= 0; rowNum++)
81         antiDiagonal.push_back(matrix[rowNum][startCol - rowNum]);
82
83     checkArrayAndUpdateMax(antiDiagonal, max);
84 }
85

```

```

86 // checking anti-diagonals starting in rightmost column
87 for (int startRow = 1; startRow < sz; startRow++) {
88
89     // creating anti-diagonal vector
90     vector<int> antiDiagonal; antiDiagonal.reserve(sz);
91
92     // iterating over columns to insert elements
93     for (int colNum = 0; startRow + colNum < sz; colNum++)
94         antiDiagonal.push_back(matrix[startRow + colNum][sz - 1 - colNum]);
95
96     checkArrayAndUpdateMax(antiDiagonal, max);
97 }
98
99 return max;
100}
101
102vector<vector<int>> generateMatrix(int sz) {
103    // generates a matrix of size sz*sz (assumed sz <= 2000) using lagged fibonacci
104    // numbers
105
106    // creating a vector of generated numbers
107    vector<int> generatedNums(sz*sz, 0);
108
109    // itearting to sz^2 times to generate all numbers
110    for (int k = 1; k <= sz * sz; k++) {
111        cpp_int Sk;    // cpp_int is variable size
112        // generating Sk
113        if (k <= 55)
114            Sk = (100003 - 200003 * k + 300007 * cpp_int(pow(k, 3))) % 1000000 - 500000;
115        else
116            Sk = (generatedNums[k - 25] + generatedNums[k - 56] + 1000000) % 1000000 -
117            500000;
118        // inserting Sk into generatedNums
119        generatedNums[k - 1] = int(Sk);
120    }
121
122    // creating a matrix from generatedNums
123    vector<vector<int>> matrix(sz, vector<int>(sz, 0));
124    // initialising to a matrix of zeros
125
126    int k = 0;
127    for (int rowNum = 0; rowNum < sz; rowNum++) {
128        // iterating over columns
129        for (int colNum = 0; colNum < sz; colNum++) {
130            // inserting generatedNums[k]
131            matrix[rowNum][colNum] = generatedNums[k];
132            k++; // incrementing k
133        }
134    }

```

Calling greatestAdjacentEntrySum(2000) returns 52852124.

17 5 / 5

✓ - 0 pts Correct

- 5 pts No submission

- 0.5 pts Omit diagonals

- 2 pts Solved a different problem