CSCE 221 Cover Page

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more Aggie Honor System Office https://aggiehonor.tamu.edu/

Name	Clayton Kristiansen
UIN	328003173
Email address	kristiansenc@tamu.edu

Cite your sources using the table below. Interactions with TAs and resources presented in lecture do not have to be cited.

People	1. None
Webpages	1. None
Printed Materials	1. None
Other Sources	1. None

Homework 2

Due March 25 at 11:59 PM

Typeset your solutions to the homework problems preferably in LaTeXor LyX. See the class webpage for information about their installation and tutorials.

1. (15 points) Provided two sorted lists, `l1` and `l2`, write a function in C++ to *efficiently* compute `l1` $\cap$ `l2` using only the basic STL list operations. The lists may be empty or contain a different number of elements e.g $|$`l1`$| \neq |$`l2`$|$. You may assume `l1` and `l2` will not contain duplicate elements.

Examples (all set members are list node):

- $\{1, 2, 3, 4\} \cap \{2, 3\} = \{2, 3\}$
- $\emptyset \cap \{2, 3\} = \emptyset$
- $\{2, 9, 14\} \cap \{1, 7, 15\} = \emptyset$

(a) Complete the function below. Do not use any routines from the algorithm header file.

```
#include <list>

std::list<int> intersection(const std::list<int> & l1, const std::list<int> & l2
    )
{
    std::list<int> l3;

    std::list<int>::const_iterator el1 = l1.begin();
    std::list<int>::const_iterator el2 = l2.begin();

    while(el1 != l1.end())
    {
        el2 = l2.begin();
        while(el2 != l2.end())
        {
            if(*el1 == *el2)
            {
                l3.push_back(*el1);
            }
            el2++;
        }
        el1++;
    }

    return l3;
}
```

(b) Verify that your implementation works properly by writing two test cases. Provide screenshot(s) with the results of your testing.

```
int main()
{
    std::list<int> l1;
    std::list<int> l2;
    l1.push_back(2);
```

```
    l1.push_back(3);
    l1.push_back(7);
    l1.push_back(8);
    l1.push_back(6);
    l2.push_back(1);
    l2.push_back(2);
    l2.push_back(3);
    l2.push_back(7);
    std::cout << "TEST CASE 1:\n";
    intersection(l1, l2);
    l1.clear();
    l2.clear();
    l1.push_back(1);
    l1.push_back(2);
    l1.push_back(3);
    l1.push_back(4);
    l1.push_back(5);
    l1.push_back(6);
    l1.push_back(7);
    l1.push_back(7);
    l1.push_back(8);
    l1.push_back(10);
    l2.push_back(5);
    l2.push_back(4);
    l2.push_back(3);
    l2.push_back(8);
    std::cout << "\n\n\nTEST CASE 2:\n";
    intersection(l1, l2);
}
```

Figure 1: Results of testing the `intersection` function

(c) What is the running time of your algorithm? Provide a big-O bound. Justify.

$$f(n) \in O(n^2) \tag{1}$$

Assuming n is the max of the two set sizes:

$$f(n) = 3n^2 + 3n + 3 \tag{2}$$
$$= n^2 \tag{3}$$

2. (15 points) Write a C++ recursive function that counts the number of nodes in a singly linked list. Do not modify the list.

Examples:

- `count_nodes`$((2) \rightarrow (4) \rightarrow (3) \rightarrow$ `nullptr`$) = 3$
- `count_nodes(nullptr)` $= 0$

(a) Complete the function below:

```
template <typename T>
struct Node
{
    Node *next;
    T obj;

    Node(T obj, Node *next = nullptr)
        : obj(obj), next(next)
    { }
};

template <typename T>
int count_nodes(Node<T>* n)
{
    while(n->next != nullptr)
    {
        return 1 + count_nodes(n->next);
    }
    return 1;
}
```

(b) Verify that your implementation works properly by writing two test cases for the function you completed in part (a). Provide screenshot(s) with the results of your testing.

```
int main()
{
    Node<int>* n1 = new Node<int>(0);
    Node<int>* first = n1;
    n1->next = new Node<int>(4); n1 = n1->next;
    n1->next = new Node<int>(3); n1 = n1->next;
    n1->next = new Node<int>(6); n1 = n1->next;
    n1->next = new Node<int>(2); n1 = n1->next;
    n1->next = new Node<int>(7); n1 = n1->next;
    n1->next = new Node<int>(8); n1 = n1->next;
    n1->next = new Node<int>(1);

    std::cout << "First Test:\n\nSize: ";
    std::cout << count_nodes<int>(first);

```

```
16
17      Node<int>* n2 = new Node<int>(0);
18      first = n2;
19      n2->next = new Node<int>(2); n2 = n2->next;
20      n2->next = new Node<int>(5); n2 = n2->next;
21      n2->next = new Node<int>(68); n2 = n2->next;
22      n2->next = new Node<int>(232); n2 = n2->next;
23      n2->next = new Node<int>(713241534); n2 = n2->next;
24      n2->next = new Node<int>(86); n2 = n2->next;
25      n2->next = new Node<int>(0); n2 = n2->next;
26      n2->next = new Node<int>(1); n2 = n2->next;
27      n2->next = new Node<int>(2); n2 = n2->next;
28      n2->next = new Node<int>(3); n2 = n2->next;
29      n2->next = new Node<int>(12);
30
31      std::cout << "\n\n\nSecond Test:\n\nSize: ";
32      std::cout << count_nodes<int>(first) << "\n";
33  }
```

Figure 2: Results of testing the `count nodes` function

(c) Write a recurrence relation that represents your algorithm.

$$T(n) = \begin{cases} T(n-1) + O(1), & \text{if } n > 1 \\ O(1), & \text{if } n = 1 \end{cases} \tag{4}$$

$$T(0) = 1 \tag{5}$$

(d) Solve the recurrence relation using the iterating or recursive tree method to obtain the running time of the algorithm in Big-O notation.

You may want to embed an image. If so, use a figure code block.

$$
\begin{aligned}
T(n) &= T(n-1) + O(1) && (6)\\
&= T(n-2) + 2 * O(1) && (7)\\
&= T(n-3) + 3 * O(1) && (8)\\
&= T(n-4) + 4 * O(1) && (9)\\
&\dots && (10)\\
&= T(n-n) + n * O(1) && (11)\\
&= T(0) + n * O(1) && (12)\\
&= 1 + n * O(1) && (13)\\
&= O(n) && (14)
\end{aligned}
$$

3. (15 points) Write a C++ recursive function that finds the maximum value in an array (or vector) of integers *without* using any loops. You may assume the array will always contain at least one integer. Do not modify the array.

(a) Complete the function below:

```
#include <iostream>
#include <vector>

using std::vector;

int find_max_value(vector<int> v, int index = 0)
{
    if(index + 1 >=  v.size())
    {
        return v[index];
    }
    int val = find_max_value(v, ++index);
    if(val > v[index])
    {
        return val;
    }
    return v[index];
}
```

(b) Verify that your implementation works properly by writing two test cases. Provide screenshot(s) with the results of the tests.

```
int main()
{
    std::vector<int> vec;
    vec.push_back(12);
    vec.push_back(21);
    vec.push_back(3);
    vec.push_back(234);
    vec.push_back(3);
    vec.push_back(12);
    vec.push_back(7);
    vec.push_back(1);
    vec.push_back(2);

```

```
14      std::cout << "First Test:\n\nSize: ";
15      std::cout << find_max_value(vec);
16
17      vec.clear();
18      vec.push_back(1);
19      vec.push_back(2);
20      vec.push_back(3);
21      vec.push_back(4);
22      vec.push_back(5);
23      vec.push_back(6);
24      vec.push_back(7);
25      vec.push_back(8);
26      vec.push_back(9);
27
28      std::cout << "\n\n\nSecond Test:\n\nSize: ";
29      std::cout << find_max_value(vec) << "\n";
30  }
```

Figure 3: Results of testing the `find_max_value` function

(c) Write a recurrence relation that represents your algorithm.

$$T(n) = \begin{cases} T(n-1) + O(1), & \text{if } n > 1 \\ O(1), & \text{if } n = 1 \end{cases} \tag{15}$$

(d) Solve the recurrence relation and obtain the running time of the algorithm in Big-O notation. Show your process.

You may want to embed an image. If so, use a figure code block.

$$
\begin{aligned}
T(n) &= T(n-1) + O(1) && (16)\\
&= T(n-2) + 2 * O(1) && (17)\\
&= T(n-3) + 3 * O(1) && (18)\\
&= T(n-4) + 4 * O(1) && (19)\\
&\cdots && (20)\\
&= T(n-n) + n * O(1) && (21)\\
&= T(0) + n * O(1) && (22)\\
&= 1 + n * O(1) && (23)\\
&= O(n) && (24)
\end{aligned}
$$

4. (15 points) What is the best, worst and average running time of quick sort algorithm?

Best case:

$$O(nlog(n)) \tag{25}$$

Average case:

$$O(nlog(n)) \tag{26}$$

Worst case:

$$O(n^2) \tag{27}$$

(a) Provide recurrence relations. For the average case, you may assume that quick sort partitions the input into two halves proportional to c and $1-c$ on each iteration.

Best:

$$T(n) = \begin{cases} 2T(n/2) + O(n), & \text{if } n > 1 \\ 0, & \text{if } n = 1 \end{cases} \tag{28}$$

Average:

$$T(n) = \begin{cases} T(cn) + T((1-c)n) + n, & \text{if } n > 1 \\ 0, & \text{if } n = 1 \end{cases} \tag{29}$$

Worst:

$$T(n) = \begin{cases} T(n-1) + n, & \text{if } n > 1 \\ 0, & \text{if } n = 1 \end{cases} \tag{30}$$

(b) Solve each recurrence relation you provided in part (a)

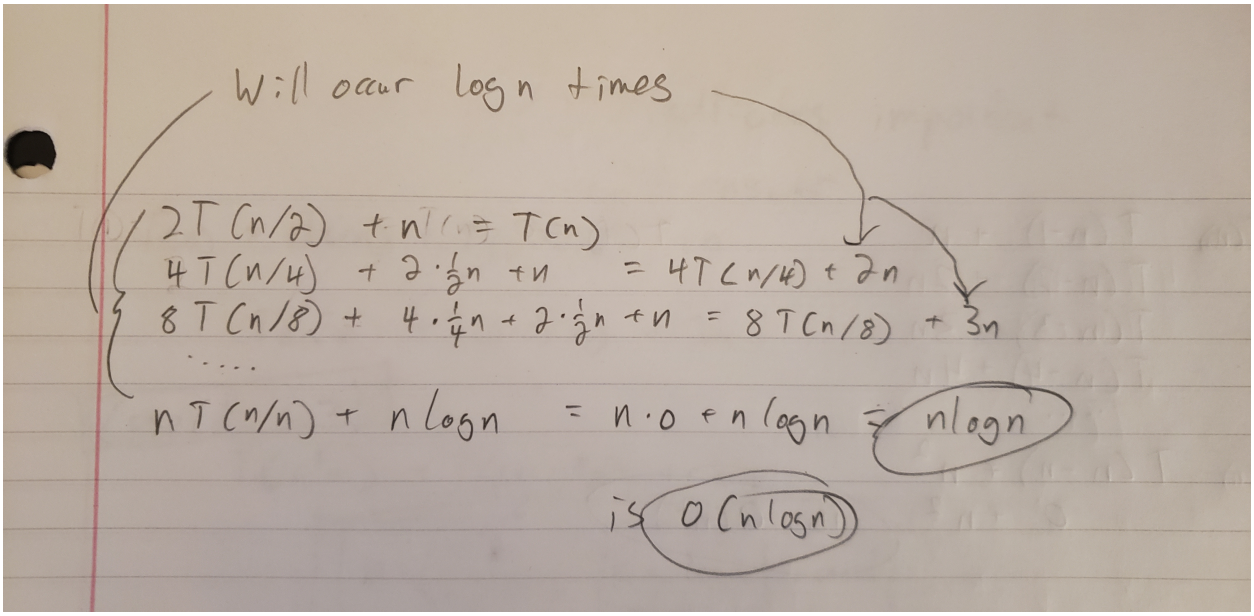

Figure 4: Solution to best case recurrence relation

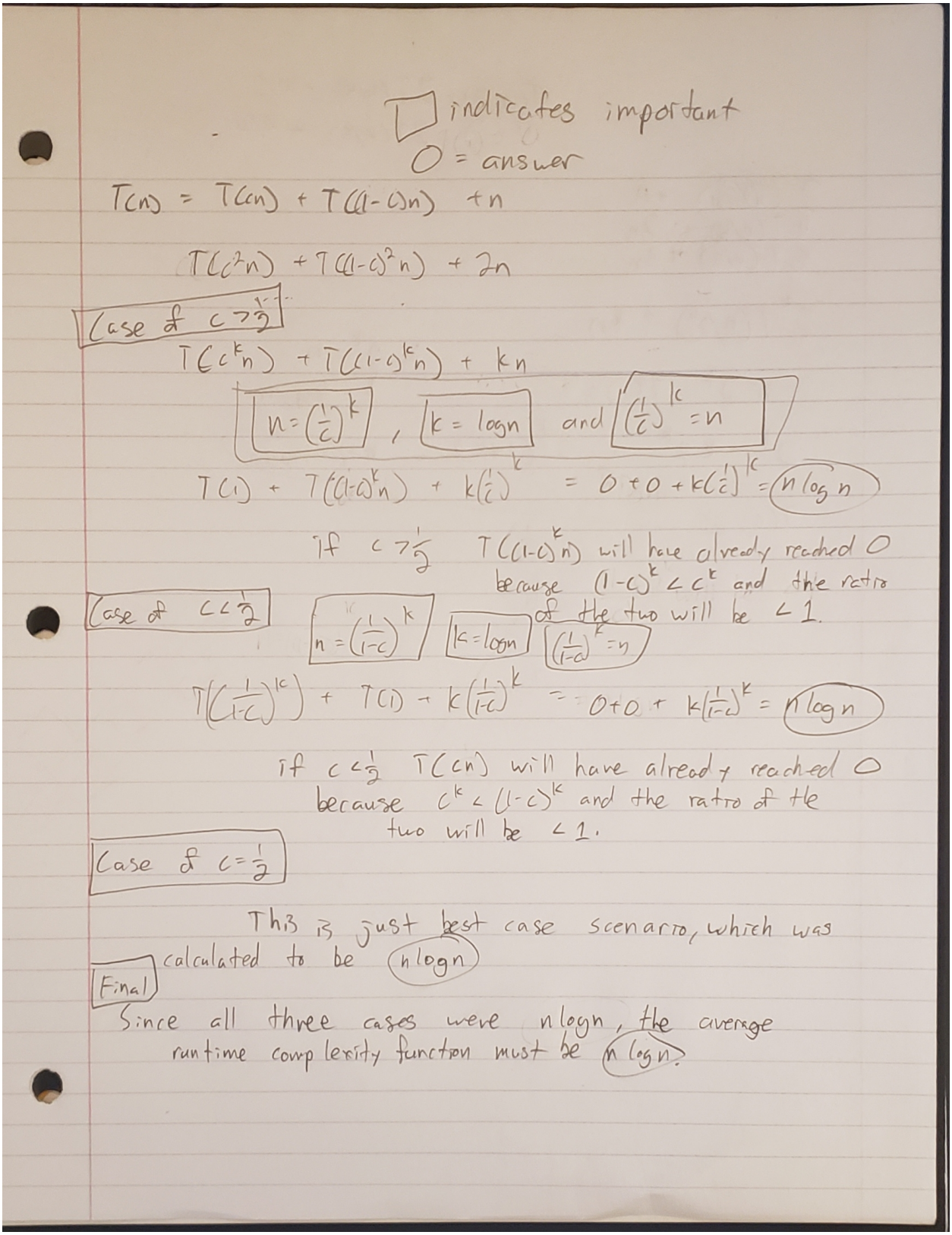

Figure 5: Solution to average case recurrence relation

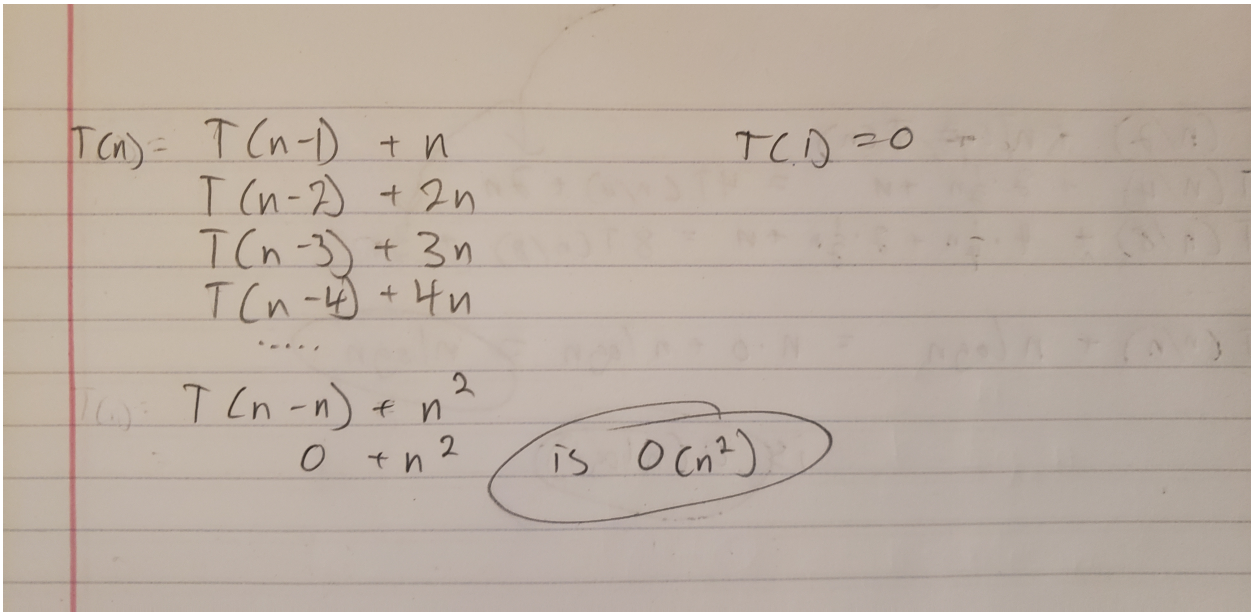

Figure 6: Solution to worst case recurrence relation

(c) Provide an arrangement of the input array which results in each case. Assume the first item is always chosen as the pivot for each iteration.

Best	$\{5, 2, 3, 4, 1, 7, 8, 9, 6\}$
Average	$\{9, 6, 7, 4, 5, 8, 3, 1, 2\}$
Worst	$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

5. (15 points) Write a C++ function that counts the total number of nodes with two children in a binary tree (do not count nodes with one or none child). You can use a STL container if you need to use an additional data structure to solve this problem.

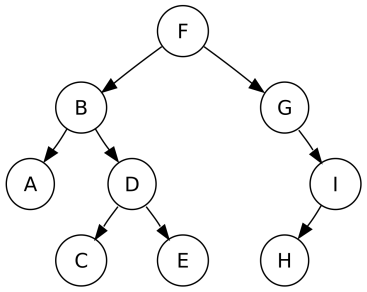

Figure 7: Calling `count_filled_nodes` on the root node F returns `3`

(a) Complete the function below. The function will be called with the root node (e.g. `count_filled_nodes(root)`). The tree may be empty. Do not modify the tree.

```
1 #include <vector>
2
```

```
3  template<typename T>
4  struct Node {
5    Node<T> *left, *right;
6    T obj;
7
8    Node(T obj, Node<T> * left = nullptr, Node<T> * right = nullptr)
9      : obj(obj), left(left), right(right)
10   { }
11 };
12
13 template <typename T>
14 int count_filled_nodes(const Node<T> *node)
15 {
16     if(node->left != nullptr && node->right != nullptr)
17     {
18         int numNodes = 0;
19         numNodes += count_filled_nodes(node->left);
20         numNodes += count_filled_nodes(node->right);
21         return 1 + numNodes;
22     }
23     else
24     {
25         if(node->left != nullptr)
26         {
27             return count_filled_nodes(node->left);
28         }
29         else if(node->right != nullptr)
30         {
31             return count_filled_nodes(node->right);
32         }
33     }
34     return 0;
35 }
```

(b) Use big-O notation to classify your algorithm. Show how you arrived at your answer.

$$f(n) \in O(n) \tag{31}$$

The recursive function visits each node in the tree once to determine whether any node has two existing children. This is a simple Preorder Traversal algorithm where the status of the current node's children are determined, then the left node is visited, then the right node is visited (if they exists respectively). The key is that each node is visited exactly once, and no more. This directly indicates a big-O classification of O(n). Even though there are multiple operations being done at each node, these constants are removed from the final big-O classification.

6. (15 points) For the following statements about red-black trees, provide a justification for each true statement and a counterexample for each false one.

(a) A subtree of a red-black tree is itself a red-black tree.

A subtree of a red-black tree may have a root with a red node. One of the 4 rules for a red-black tree is that the root node is black. This would mean that any subtree with a red root would not be themselves a red-black tree.

(b) The sibling of an external node is either external or red.

No. If you have a subtree of the red-black tree with a red node root, and one black child, the sibling of the external node off the red root would have a black sibling.

(c) There is a unique 2-4 tree associated with a given red-black tree.

No, you can represent a given red-black tree with many different 2-4 trees because 2-4 trees can preseve the order and relative positioning of values, but have them in different groups of 3.

(d) There is a unique red-black tree associated with a given 2-4 tree.

Yes, red-black trees are forced to be unique for a given relative positioning of nodes. This means that every 2-4 tree (which has a unique relative positioning) must also have a unique red-black tree.

7. (10 points) Modify this skip list after performing the following series of operations: `erase(38)`, `insert(48,x)`, `insert(24, y)`, `erase(42)`. Provided the recorded coin flips for `x` and `y`. Provide a record of your work for partial credit.

$-\infty$	—	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	—	17	—	—	42	$+\infty$
$-\infty$	—	17	—	—	42	$+\infty$
$-\infty$	12	17	—	38	42	$+\infty$
$-\infty$	12	17	20	38	42	$+\infty$
$-\infty$	—	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	—	17	—	42	—	$+\infty$
$-\infty$	—	17	—	42	—	$+\infty$
$-\infty$	12	17	—	42	—	$+\infty$
$-\infty$	12	17	20	42	—	$+\infty$

Next:

$-\infty$	—	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	—	17	—	42	—	$+\infty$
$-\infty$	—	17	—	42	—	$+\infty$
$-\infty$	12	17	—	42	48	$+\infty$
$-\infty$	12	17	20	42	48	$+\infty$

Next:

$-\infty$	—	—	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	42	—	$+\infty$
$-\infty$	—	17	—	24	42	—	$+\infty$
$-\infty$	12	17	—	24	42	48	$+\infty$
$-\infty$	12	17	20	24	42	48	$+\infty$

Next:

$-\infty$	—	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	—	17	—	24	—	$+\infty$
$-\infty$	12	17	—	24	48	$+\infty$
$-\infty$	12	17	20	24	48	$+\infty$