Chapter 10
Sorting and Searching Algorithms
• Sorting rearranges the elements into either ascending or descending order within the array. (We’ll use ascending order.)

• The values stored in an array have keys of a type for which the relational operators are defined. (We also assume unique keys.)
Divides the array into two parts: already sorted, and not yet sorted.

On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.
Selection Sort: Pass One
Selection Sort: End Pass One

values
[ 0 ] 6
[ 1 ] 24
[ 2 ] 10
[ 3 ] 36
[ 4 ] 12

UNSORTED
SORTED

Copyright ©2013 by Jones & Bartlett Learning, LLC an Ascend Learning Company
www.jblearning.com
Selection Sort: Pass Two

values
[0] 6
[1] 24
[2] 10
[3] 36
[4] 12

SORTED

UNSORTED
Selection Sort: End Pass Two

values

[ 0 ] 6
[ 1 ] 10
[ 2 ] 24
[ 3 ] 36
[ 4 ] 12

SORTED

UNSORTED
Selection Sort: Pass Three

values

[ 0 ] 6
[ 1 ] 10
[ 2 ] 24
[ 3 ] 36
[ 4 ] 12

SORTED

UNSORTED

Copyright ©2013 by Jones & Bartlett Learning, LLC an Ascend Learning Company
www.jblearning.com
Selection Sort: End Pass Three

values

[0] 6
[1] 10
[2] 12
[3] 36
[4] 24
Selection Sort: Pass Four

values
[0] 6
[1] 10
[2] 12
[3] 36
[4] 24
Selection Sort: End Pass Four

values
[ 0 ]
[ 1 ]
[ 2 ]
[ 3 ]
[ 4 ]

6
10
12
24
36

SORTED
Selection Sort:
How many comparisons?

<table>
<thead>
<tr>
<th>values</th>
<th>0</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

4 compares for values[0]
3 compares for values[1]
2 compares for values[2]
1 compare for values[3]

= 4 + 3 + 2 + 1
• The number of comparisons when the array contains N elements is

$$\text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1$$
\[
\text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1
\]

\[
\text{+ \ Sum} = 1 + 2 + \ldots + (N-2) + (N-1)
\]

\[
2 \times \text{Sum} = N + N + \ldots + N + N
\]

\[
2 \times \text{Sum} = \frac{N \times (N-1)}{2}
\]

\[
\text{Sum} = \frac{N \times (N-1)}{2}
\]
• The number of comparisons when the array contains $N$ elements is

$$\text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1$$

$$\text{Sum} = N \times (N-1) / 2$$

$$\text{Sum} = 0.5 N^2 - 0.5 N$$

$$\text{Sum} = O(N^2)$$
template <class ItemType>
int MinIndex(ItemType values [], int start, int end)
// Post: Function value = index of the smallest value
// in values [start] . . . values [end].
{
    int indexOfMin = start;

    for(int index = start + 1 ; index <= end ; index++)
        if (values[ index ] < values [indexOfMin])
            indexOfMin = index ;

    return indexOfMin;
}
template <class ItemType>
void SelectionSort (ItemType values[ ],
                   int numValues )

    // Post: Sorts array values[0 . . numValues-1 ]
    // into ascending order by key
    {
        int endIndex = numValues - 1;

        for (int current = 0; current < endIndex;
            current++)
        {
            Swap (values[current],
                values[MinIndex(values,current, endIndex)]);
        }
Compares neighboring pairs of array elements, starting with the last array element, and swaps neighbors whenever they are not in correct order.

On each pass, this causes the smallest element to “bubble up” to its correct place in the array.
Snapshot of BubbleSort

- **sorted part**: `values[0]..values[current-1]`
- **In BubbleUp**: Not yet examined: `values[current]..values[index-1]`
- **Examined**: `values[index+1]..values[numValues-1]` are all greater than `values[index]`
Code for BubbleSort

template<class ItemType>
void BubbleSort(ItemType values[],
    int numValues)
{
    int current = 0;
    while (current < numValues - 1)
    {
        BubbleUp(values, current, numValues-1);
        current++;
    }
}
Code for BubbleUp

template<class ItemType>
void BubbleUp(ItemType values[],
    int startIndex, int endIndex)
// Post: Adjacent pairs that are out of
// order have been switched between
// values[startIndex]..values[endIndex]
// beginning at values[endIndex].
{
    for (int index = endIndex;
         index > startIndex; index--)
    if (values[index] < values[index-1])
        Swap(values[index], values[index-1]);
}
Observations on BubbleSort

This algorithm is *always* $O(N^2)$.

There can be a large number of intermediate swaps.

Can this algorithm be improved?
One by one, each as yet unsorted array element is inserted into its proper place with respect to the already sorted elements.

On each pass, this causes the number of already sorted elements to increase by one.
Insertion Sort

Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.
Insertion Sort

Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.
Insertion Sort

Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.
Insertion Sort

Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.
A Snapshot of the Insertion Sort Algorithm

Sort part
values[0]..values[current-1]

values[current] is inserted into sorted portion

Nothing is known about
values[current+1]..values[numValues-1]
template <class ItemType >
void InsertItem  ( ItemType values [ ] , int start ,
                 int end )
  //  Post: Elements between values[start] and values[end] have been sorted into ascending order by key.
{
  bool finished = false;
  int current = end;
  bool moreToSearch = (current != start);

  while (moreToSearch && !finished )
  {
    if  (values[current] < values[current - 1])
    {
      Swap(values[current], values[current - 1]);
      current--;
      moreToSearch = (current != start);
    }
    else
      finished = true;
  }
}
template <class ItemType>
void InsertionSort ( ItemType values [],
                   int numValues )

   // Post: Sorts array values[0 .. numValues-1] into
   //      ascending order by key
   {
      for (int count = 0 ; count < numValues; count++)
         InsertItem ( values , 0 , count );
   }
Sorting Algorithms and Average Case Number of Comparisons

Simple Sorts
- Straight Selection Sort
- Bubble Sort
- Insertion Sort

More Complex Sorts
- Quick Sort
- Merge Sort
- Heap Sort

\[ O(N^2) \]

\[ O(N \times \log N) \]
Divide and Conquer Sorts

\[ N = 100 \]
\[ N^2 = (100)^2 = 10,000 \]

\[ (\frac{1}{2}N)^2 + (\frac{1}{2}N)^2 + N \]
\[ = (50)^2 + (50)^2 + 100 \]
\[ = 5100 \]
A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- Its shape must be a complete binary tree.

- For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.
The largest element in a heap is always found in the root node.
The heap can be stored in an array.

```
values

[0] 70
[1] 60
[2] 12
[3] 40
[4] 30
[5] 8
[6] 10
```

```
root

0

70

60

1

40

4

30

12

2

8

5

10

6
```
Heap Sort Approach

First, make the unsorted array into a heap by satisfying the order property. Then repeat the steps below until there are no more unsorted elements.

- **Take the root (maximum) element off the heap** by swapping it into its correct place in the array at the end of the unsorted elements.
- **Reheap the remaining unsorted elements.** (This puts the next-largest element into the root position).
After creating the original heap

values

[0] 70
[1] 60
[2] 12
[3] 40
[4] 30
[5] 8
[6] 10

root

70
60
40
30
8
10

Copyright ©2013 by Jones & Bartlett Learning, LLC an Ascend Learning Company
www.jblearning.com
Swap root element into last place in unsorted array

values

[ 0 ] 70
[ 1 ] 60
[ 2 ] 12
[ 3 ] 40
[ 4 ] 30
[ 5 ] 8
[ 6 ] 10

root

70

0

60

1

40

3

30

4

12

2

8

5

10

6
After swapping root element into its place

values

[ 0 ]  10
[ 1 ]  60
[ 2 ]  12
[ 3 ]  40
[ 4 ]  30
[ 5 ]   8
[ 6 ]  70

NO NEED TO CONSIDER AGAIN
After reheaping remaining unsorted elements

values

[0] 60
[1] 40
[2] 12
[3] 10
[4] 30
[5] 8
[6] 70

root

60

10

40

30

12

8

70

10

30

8

70

Copyright ©2013 by Jones & Bartlett Learning, LLC an Ascend Learning Company
www.jblearning.com
Swap root element into last place in unsorted array
After swapping root element into its place

[0] 8
[1] 40
[2] 12
[3] 10
[4] 30
[5] 60
[6] 70

values

root

NO NEED TO CONSIDER AGAIN
After reheaping remaining unsorted elements

```
values

[ 0 ] 40
[ 1 ] 30
[ 2 ] 12
[ 3 ] 10
[ 4 ] 6
[ 5 ] 60
[ 6 ] 70
```

Diagram of a tree with values:

- Root: 40
- Children:
  - Left: 30 (children: 10, 6)
  - Right: 12 (children: 60, 70)

```
Swap root element into last place in unsorted array
After swapping root element into its place

values

[ 0 ] 6
[ 1 ] 30
[ 2 ] 12
[ 3 ] 10
[ 4 ] 40
[ 5 ] 60
[ 6 ] 70

root

NO NEED TO CONSIDER AGAIN
After reheaping remaining unsorted elements

values

[0] 30
[1] 10
[2] 12
[3] 6
[4] 40
[5] 60
[6] 70

root

[0] 30
[1] 10
[2] 12
[3] 6
[4] 40
[5] 60
[6] 70

Copyright ©2013 by Jones & Bartlett Learning, LLC an Ascend Learning Company
www.jblearning.com
Swap root element into last place in unsorted array
After swapping root element into its place

values

[ 0 ]  6
[ 1 ]  10
[ 2 ]  12
[ 3 ]  30
[ 4 ]  40
[ 5 ]  60
[ 6 ]  70

NO NEED TO CONSIDER AGAIN
After reheaping remaining unsorted elements

values

0: 12
1: 10
2: 6
3: 30
4: 40
5: 60
6: 70

root

[12]

[10]

[30, 40]

[60]

[70]
Swap root element into last place in unsorted array.
After swapping root element into its place

values

0 6
1 10
2 12
3 30
4 40
5 60
6 70

root

NO NEED TO CONSIDER AGAIN
After reheaping remaining unsorted elements
Swap root element into last place in unsorted array
After swapping root element into its place

ALL ELEMENTS ARE SORTED
template <class ItemType>
void HeapSort (ItemType values[], int numValues)
// Post: Sorts array values[0..numValues-1] into ascending order by key
{
    int index;

    // Convert array values[0..numValues-1] into a heap
    for (index = numValues/2 - 1; index >= 0; index--)
        ReheapDown (values, index, numValues - 1);

    // Sort the array.
    for (index = numValues - 1; index >= 1; index--)
    {
        Swap (values [0], values[index]);
        ReheapDown (values, 0, index - 1);
    }
}
template< class ItemType >
void ReheapDown ( ItemType values [ ], int root, int bottom )
{
    int maxChild;
    int rightChild;
    int leftChild;

    leftChild = root * 2 + 1;
    rightChild = root * 2 + 2;

// Pre: root is the index of a node that may violate the
//      heap order property
// Post: Heap order property is restored between root and
//      bottom
if (leftChild <= bottom) // ReheapDown continued
{
  if (leftChild == bottom)
    maxChild = leftChild;
  else
  {
    if (values[leftChild] <= values[rightChild])
      maxChild = rightChild;
    else
      maxChild = leftChild;
  }
  if (values[root] < values[maxChild])
  {
    Swap (values[root], values[maxChild]);
    ReheapDown (maxChild, bottom);
  }
}
Heap Sort:  
How many comparisons?

In reheap down, an element is compared with its 2 children (and swapped with the larger). But only one element at each level makes this comparison, and a complete binary tree with N nodes has only $O(\log_2 N)$ levels.
Heap Sort of N elements: How many comparisons?

\[ \frac{N}{2} \times O(\log N) \] compares to create original heap

\[ (N-1) \times O(\log N) \] compares for the sorting loop

\[ = O(N \times \log N) \] compares total
Using quick sort algorithm
// Recursive quick sort algorithm

template <class ItemType>
void QuickSort ( ItemType values[ ], int first, int last )

// Pre: first <= last
// Post: Sorts array values[first..last] into ascending order
{
    if ( first < last )
    {
        int splitPoint;
        Split ( values, first, last, splitPoint ) ;
        // values[first]..values[splitPoint - 1] <= splitVal
        // values[splitPoint] = splitVal
        // values[splitPoint + 1]..values[last] > splitVal
        QuickSort(values, first, splitPoint - 1);
        QuickSort(values, splitPoint + 1, last);
    }
} ;
Before call to function Split

\[
\text{splitVal} = 9
\]

**GOAL:** place \( \text{splitVal} \) in its proper position with all values less than or equal to \( \text{splitVal} \) on its left and all larger values on its right

\[
\begin{array}{cccccccc}
9 & 20 & 6 & 18 & 14 & 3 & 60 & 11 \\
\end{array}
\]
After call to function Split

\[
\text{splitVal} = 9
\]

smaller values in left part

larger values in right part

values[first]

[\text{last}]

\text{splitVal in correct position}
Quick Sort of N elements:

How many comparisons?

N  For first call, when each of N elements
    is compared to the split value

2 * N/2  For the next pair of calls, when N/2
         elements in each “half” of the original
         array are compared to their own split values.

4 * N/4  For the four calls when N/4 elements in each
         “quarter” of original array are compared to
         their own split values.

HOW MANY SPLITS CAN OCCUR?
Quick Sort of N elements: How many splits can occur?

It depends on the order of the original array elements!

If each split divides the subarray approximately in half, there will be only \( \log_2 N \) splits, and QuickSort is \( O(N \log_2 N) \).

But, if the original array was sorted to begin with, the recursive calls will split up the array into parts of unequal length, with one part empty, and the other part containing all the rest of the array except for split value itself. In this case, there can be as many as \( N-1 \) splits, and QuickSort is \( O(N^2) \).
Before call to function Split

splitVal = 9

GOAL: place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

values[first] | [last]
After call to function Split

`splitVal = 9`

- no smaller values
- empty left part
- larger values
- in right part with N-1 elements

values[first] | 9 | 20 | 26 | 18 | 14 | 53 | 60 | 11 | [last]

splitVal in correct position
Merge Sort Algorithm

Cut the array in half.
Sort the left half.
Sort the right half.
Merge the two sorted halves into one sorted array.
// Recursive merge sort algorithm

template <class ItemType>
void MergeSort ( ItemType values[], int first, int last )
// Pre:   first <= last
// Post: Array values[first..last] sorted into
//       ascending order.
{
  if ( first < last ) // general case
  {
    int middle = ( first + last ) / 2;
    MergeSort ( values, first, middle );
    MergeSort( values, middle + 1, last );
    
    // now merge two subarrays
    // values [ first . . . middle ] with
    // values [ middle + 1, . . . last ].

    Merge(values, first, middle, middle + 1, last);
  }
}
Using Merge Sort Algorithm
with N = 16
Merge Sort of N elements:
How many comparisons?

The entire array can be subdivided into halves only \( \log_2 N \) times.

Each time it is subdivided, function Merge is called to re-combine the halves. Function Merge uses a temporary array to store the merged elements. Merging is \( O(N) \) because it compares each element in the subarrays.

Copying elements back from the temporary array to the values array is also \( O(N) \).

**MERGE SORT IS** \( O(N \cdot \log_2 N) \).
Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Sort</th>
<th>Best Case</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>selectionSort</td>
<td>O($N^2$)</td>
<td>O($N^2$)</td>
<td>O($N^2$)</td>
</tr>
<tr>
<td>bubbleSort</td>
<td>O($N^2$)</td>
<td>O($N^2$)</td>
<td>O($N^2$)</td>
</tr>
<tr>
<td>shortBubble</td>
<td>O($N$) (*)</td>
<td>O($N^2$)</td>
<td>O($N^2$)</td>
</tr>
<tr>
<td>insertionSort</td>
<td>O($N$) (*)</td>
<td>O($N^2$)</td>
<td>O($N^2$)</td>
</tr>
<tr>
<td>mergeSort</td>
<td>O($N\log_2 N$)</td>
<td>O($N\log_2 N$)</td>
<td>O($N\log_2 N$)</td>
</tr>
<tr>
<td>quickSort</td>
<td>O($N\log_2 N$)</td>
<td>O($N\log_2 N$)</td>
<td>O($N^2$) (depends on split)</td>
</tr>
<tr>
<td>heapSort</td>
<td>O($N\log_2 N$)</td>
<td>O($N\log_2 N$)</td>
<td>O($N\log_2 N$)</td>
</tr>
</tbody>
</table>

*Data almost sorted.*
Testing

• To thoroughly test our sorting methods we should vary the size of the array they are sorting
• Vary the original order of the array-test
  • Reverse order
  • Almost sorted
  • All identical elements
Sorting Objects

• When sorting an array of objects we are manipulating references to the object, and not the objects themselves.
Stability

• Stable Sort: A sorting algorithm that preserves the order of duplicates

• Of the sorts that we have discussed in this book, only heapSort and quickSort are inherently unstable
Function BinarySearch( )

• BinarySearch takes sorted array info, and two subscripts, fromLoc and toLoc, and item as arguments. It returns false if item is not found in the elements info[fromLoc...toLoc]. Otherwise, it returns true.

• BinarySearch is $O(\log_2 N)$. 
found = BinarySearch(info, 25, 0, 14);

NOTE: denotes element examined
template<class ItemType>
bool BinarySearch(ItemType info[], ItemType item, 
    int fromLoc, int toLoc )
    // Pre: info [ fromLoc .. toLoc ] sorted in ascending order
    // Post: Function value = ( item in info[fromLoc .. toLoc] )
{
    int mid ;
    if ( fromLoc > toLoc ) /* base case -- not found */
        return false ;
    else
    {
        mid = ( fromLoc + toLoc ) / 2 ;
        if ( info[mid] == item ) /* base case-- found at mid */
            return true ;
        else
        {
            if ( item < info[mid] ) /* search lower half */
                return BinarySearch( info, item, fromLoc, mid-1 ) ;
            else /* search upper half */
                return BinarySearch( info, item, mid + 1, toLoc ) ;
        }
    }
• is a means used to order and access elements in a list quickly -- the goal is O(1) time -- by using a function of the key value to identify its location in the list.

• The function of the key value is called a hash function.

FOR EXAMPLE . . .
Using a hash function

HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

Hash(key) = partNum % 100
Use the hash function

\[
\text{Hash(key)} = \text{partNum} \mod 100
\]

to place the element with part number 5502 in the array.
Next place part number 6702 in the array.

Hash(key) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS
How to Resolve the Collision?

One way is by linear probing. This uses the rehash function

\[(\text{HashValue} + 1) \mod 100\]

repeatedly until an empty location is found for part number 6702.
Still looking for a place for 6702 using the function

\[(\text{HashValue} + 1) \mod 100\]
Collision Resolved

Part 6702 can be placed at the location with index 4.

<table>
<thead>
<tr>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 0 ]</td>
</tr>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>[ 1 ]</td>
</tr>
<tr>
<td>4501</td>
</tr>
<tr>
<td>[ 2 ]</td>
</tr>
<tr>
<td>5502</td>
</tr>
<tr>
<td>[ 3 ]</td>
</tr>
<tr>
<td>7803</td>
</tr>
<tr>
<td>[ 4 ]</td>
</tr>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>[ 97 ]</td>
</tr>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>[ 98 ]</td>
</tr>
<tr>
<td>2298</td>
</tr>
<tr>
<td>[ 99 ]</td>
</tr>
<tr>
<td>3699</td>
</tr>
</tbody>
</table>
Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?
Radix Sort

Radix sort
Is not a comparison sort

Uses a radix-length array of queues of records

Makes use of the values in digit positions in the keys to select the queue into which a record must be enqueued
Original Array

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>762</td>
<td>124</td>
<td>432</td>
<td>761</td>
</tr>
<tr>
<td>800</td>
<td>402</td>
<td>976</td>
<td>100</td>
</tr>
<tr>
<td>001</td>
<td>999</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Queues After First Pass

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>800</td>
<td>761</td>
<td>762</td>
<td>124</td>
<td>976</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>999</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>001</td>
<td>432</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>402</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Array After First Pass

800
100
761
001
762
432
402
124
976
999
### Queues After Second Pass

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>124</td>
<td>432</td>
<td>761</td>
<td>976</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td>762</td>
</tr>
<tr>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>402</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Array After Second Pass

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>100</td>
<td>001</td>
<td>402</td>
<td>124</td>
<td>432</td>
<td>761</td>
<td>762</td>
<td>976</td>
<td>999</td>
</tr>
</tbody>
</table>
### Queues After Third Pass

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>100</td>
<td></td>
<td>402</td>
<td></td>
<td></td>
<td>761</td>
<td>800</td>
<td>976</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td></td>
<td>432</td>
<td></td>
<td></td>
<td>762</td>
<td></td>
<td></td>
<td>999</td>
<td></td>
</tr>
</tbody>
</table>
Array After Third Pass

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>402</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>432</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>761</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>762</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>976</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>999</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>