CS 187: Programming with Data Structures (Spring 2010)

Lecture 16: Merge Sort

Rui Wang
Advanced Sorting

• We’ve learned some simple sorting methods, which all have quadratic costs.
  – Easy to implement but slow.

• Some advanced sorting methods that are much faster:
  – Merge Sort
  – Quick Sort
  – Radix Sort
Merge Sort

• Divide and conquer
• Uses recursion
• Much faster than simple sorting algorithms
• Requires additional memory space
  – In fact, it requires a temporary array that’s as large as the original array.
Merging Two Sorted Arrays

- A **key step** in mergesort
- Assume arrays A and B are **already sorted**.
- Merge them to array C, such as C contains all elements from A and B, and remains sorted.
Merging Two Sorted Arrays

• A key step in mergesort
• Assume arrays A and B are already sorted.
• Merge them to array C, such as C contains all elements from A and B, and remains sorted.

• Example:
  A:  23  47  81  95
  B:   7  14  39  55  62  74
  (Note that A and B can be of different sizes)
Merging Two Sorted Arrays

a) Before Merge

b) After Merge
## Merging Two Sorted Arrays

<table>
<thead>
<tr>
<th>Step</th>
<th>Comparison (If Any)</th>
<th>Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Compare 23 and 7</td>
<td>Copy 7 from B to C</td>
</tr>
<tr>
<td>2</td>
<td>Compare 23 and 14</td>
<td>Copy 14 from B to C</td>
</tr>
<tr>
<td>3</td>
<td>Compare 23 and 39</td>
<td>Copy 23 from A to C</td>
</tr>
<tr>
<td>4</td>
<td>Compare 39 and 47</td>
<td>Copy 39 from B to C</td>
</tr>
<tr>
<td>5</td>
<td>Compare 55 and 47</td>
<td>Copy 47 from A to C</td>
</tr>
<tr>
<td>6</td>
<td>Compare 55 and 81</td>
<td>Copy 55 from B to C</td>
</tr>
<tr>
<td>7</td>
<td>Compare 62 and 81</td>
<td>Copy 62 from B to C</td>
</tr>
<tr>
<td>8</td>
<td>Compare 74 and 81</td>
<td>Copy 74 from B to C</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Copy 81 from A to C</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Copy 95 from A to C</td>
</tr>
</tbody>
</table>
Merging Two Sorted Arrays

- **Summary of steps:**
  1. Start from the first elements of A and B;
  2. Compare and copy the smaller element to C;
  3. Increment indices, and continue;
  4. If reaching the end of either A or B, quit loop;
  5. If either A (or B) contains remaining elements, append them to C.
public static void merge( int[] arrayA, int sizeA,
    int[] arrayB, int sizeB,
    int[] arrayC )
{
    int aDex=0, bDex=0, cDex=0;

    while(aDex < sizeA && bDex < sizeB)  // neither array empty
        if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
        else
            arrayC[cDex++] = arrayB[bDex++];

    while(aDex < sizeA)  // arrayB is empty,
        arrayC[cDex++] = arrayA[aDex++];  // but arrayA isn't

    while(bDex < sizeB)  // arrayA is empty,
        arrayC[cDex++] = arrayB[bDex++];  // but arrayB isn't
}  // end merge()
public static void merge( int[] arrayA, int sizeA,
    int[] arrayB, int sizeB,
    int[] arrayC )
{
    int aDex=0, bDex=0, cDex=0;

    while(aDex < sizeA && bDex < sizeB) // neither array empty
        if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
        else
            arrayC[cDex++] = arrayB[bDex++];

    while(aDex < sizeA) // arrayB is empty,
        arrayC[cDex++] = arrayA[aDex++]; // but arrayA isn't

    while(bDex < sizeB) // arrayA is empty,
        arrayC[cDex++] = arrayB[bDex++]; // but arrayB isn't
} // end merge()
public static void merge( int[] arrayA, int sizeA, 
    int[] arrayB, int sizeB, 
    int[] arrayC )
{
    int aDex=0, bDex=0, cDex=0;

    while(aDex < sizeA && bDex < sizeB) // neither array empty
        if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
        else
            arrayC[cDex++] = arrayB[bDex++];

    while(aDex < sizeA) // arrayB is empty, 
        arrayC[cDex++] = arrayA[aDex++]; // but arrayA isn't

    while(bDex < sizeB) // arrayA is empty, 
        arrayC[cDex++] = arrayB[bDex++]; // but arrayB isn't
} // end merge()
Merging Two Sorted Arrays

Questions:

• Is it possible that both A and B contain remaining elements after the first loop?
• What happens if A is an empty array to start with?
• How many comparisons is required?

• How many copies?
Merging Two Sorted Arrays

Questions:

• Is it possible that both A and B contain remaining elements after the first loop? No.
• What happens if A is an empty array to start with?
• How many comparisons is required? at most (A.length + B.length)
• How many copies? A.length + B.length
Sorting by Merging

• Merging two sorted arrays is the key step in mergesort.

• Once we have it, mergesort is pretty simple:
  – Divide array in half
  – Sort each half (how?)
  – Merge the two sorted halves
Sorting by Merging

• Merging two sorted arrays is the key step in mergesort.

• Once we have it, mergesort is pretty simple:
  – Divide array in half
  – Sort each half (this step uses recursion!)
  – Merge the two sorted halves

How does this work internally?
Sorting by Merging

• Merging two sorted arrays is the key step in mergesort.

• Once we have it, mergesort is pretty simple:
  – Divide array in half
  – Sort each half (this step uses recursion!)
  – Merge the two sorted halves

• What’s the base case?
  – When you have only one element left.
Sorting by Merging

• This is a divide and conquer approach:
  – Partition the original problem into two sub-problems;
  – Use recursion to solve each sub-problem;
  – Sub-problem eventually reduces to base case;
  – The results are then combined to solve the original problem.
Divide
Divide
Conquer
private void recMergeSort(long[] workSpace, int lowerBound, int upperBound)
{
    if(lowerBound == upperBound) // if range is 1, no use sorting
        return;
    else
    {
        // find midpoint
        int mid = (lowerBound + upperBound) / 2;
        // sort low half
        recMergeSort(workSpace, lowerBound, mid);
        // sort high half
        recMergeSort(workSpace, mid + 1, upperBound);
        // merge them
        merge(workSpace, lowerBound, mid + 1, upperBound);
    } // end else
} // end recMergeSort
private void recMergeSort(long[] workSpace, int lowerBound, int upperBound)
{
    if(lowerBound == upperBound) // if range is 1, no use sorting
        return;
    else
    {
        // find midpoint
        int mid = (lowerBound + upperBound) / 2;
        // sort low half
        recMergeSort(workSpace, lowerBound, mid);
        // sort high half
        recMergeSort(workSpace, mid + 1, upperBound);
        // merge them
        merge(workSpace, lowerBound, mid + 1, upperBound);
    } // end else
} // end recMergeSort
private void recMergeSort(long[] workSpace, int lowerBound,
    int upperBound)
{
    if(lowerBound == upperBound) // if range is 1, // no use sorting
        return;
    else { // find midpoint
        int mid = (lowerBound+upperBound) / 2;
            // sort low half
        recMergeSort(workSpace, lowerBound, mid);
            // sort high half
        recMergeSort(workSpace, mid+1, upperBound);
            // merge them
        merge(workSpace, lowerBound, mid+1, upperBound);
    } // end else
} // end recMergeSort
private void recMergeSort(long[] workSpace, int lowerBound, int upperBound)
{
    if(lowerBound == upperBound) // if range is 1, no use sorting
        return;
    else
    {
        // find midpoint
        int mid = (lowerBound + upperBound) / 2;
        // sort low half
        recMergeSort(workSpace, lowerBound, mid);
        // sort high half
        recMergeSort(workSpace, mid + 1, upperBound);
        // merge them
        merge(workSpace, lowerBound, mid + 1, upperBound);
    } // end else
} // end recMergeSort
First base case encountered
Return, and continue.
Merge
Merge
private void recMergeSort(long[] workSpace, int lowerBound,
                        int upperBound)
{
    if(lowerBound == upperBound) // if range is 1, no use sorting
        return;
    else
    {
        // find midpoint
        int mid = (lowerBound+upperBound) / 2;
        // sort low half
        recMergeSort(workSpace, lowerBound, mid);
        // sort high half
        recMergeSort(workSpace, mid+1, upperBound);
        // merge them
        merge(workSpace, lowerBound, mid+1, upperBound);
    } // end else
} // end recMergeSort

Does the program look familiar? Think about Binary Search.
Sorting by Merging

- The `merge()` method
  - A `workspace` array is created to serve as array C (the merged array).
  - It’s the same size as the input array.
  - During each merge step, two subarrays in the input array are merged into a section of the workspace array.
  - After merge, the corresponding section of the workspace array is copied back to the input array.
private void merge(long[] workSpace, int lowPtr,
    int highPtr, int upperBound)
{
    int j = 0;  // workspace index
    int lowerBound = lowPtr;
    int mid = highPtr-1;
    int n = upperBound-lowerBound+1;  // # of items

    while(lowPtr <= mid && highPtr <= upperBound)
    {
        if( theArray[lowPtr] < theArray[highPtr] )
            workSpace[j++] = theArray[lowPtr++];
        else
            workSpace[j++] = theArray[highPtr++];
    }

    while(lowPtr <= mid)
        workSpace[j++] = theArray[lowPtr++];

    while(highPtr <= upperBound)
        workSpace[j++] = theArray[highPtr++];

    for(j=0; j<n; j++)
        theArray[lowerBound+j] = workSpace[j];
}  // end merge()
private void merge(long[] workSpace, int lowPtr, 
    int highPtr, int upperBound)
{
    int j = 0; // workspace index
    int lowerBound = lowPtr;
    int mid = highPtr-1;
    int n = upperBound-lowerBound+1; // # of items

    while(lowPtr <= mid && highPtr <= upperBound)
        if( theArray[lowPtr] < theArray[highPtr] )
            workSpace[j++] = theArray[lowPtr++];
        else
            workSpace[j++] = theArray[highPtr++];

    while(lowPtr <= mid)
        workSpace[j++] = theArray[lowPtr++];

    while(highPtr <= upperBound)
        workSpace[j++] = theArray[highPtr++];

    for(j=0; j<n; j++)
        theArray[lowerBound+j] = workSpace[j];
} // end merge()
private void merge(long[] workSpace, int lowPtr,
                   int highPtr, int upperBound)
{
    int j = 0;                  // workspace index
    int lowerBound = lowPtr;
    int mid = highPtr - 1;
    int n = upperBound - lowerBound + 1;  // # of items

    while (lowPtr <= mid && highPtr <= upperBound)
    {
        if (theArray[lowPtr] < theArray[highPtr])
            workSpace[j++] = theArray[lowPtr++];
        else
            workSpace[j++] = theArray[highPtr++];
    }

    while (lowPtr <= mid)
        workSpace[j++] = theArray[lowPtr++];

    while (highPtr <= upperBound)
        workSpace[j++] = theArray[highPtr++];

    for (j = 0; j < n; j++)
        theArray[lowerBound + j] = workSpace[j];
}  // end merge()


private void merge(long[] workSpace, int lowPtr, int highPtr, int upperBound)
{
    int j = 0;  // workspace index
    int lowerBound = lowPtr;
    int mid = highPtr-1;
    int n = upperBound-lowerBound+1;  // # of items

    while(lowPtr <= mid && highPtr <= upperBound)
    {
        if( theArray[lowPtr] < theArray[highPtr] )
            workSpace[j++] = theArray[lowPtr++];
        else
            workSpace[j++] = theArray[highPtr++];
    }
    while(lowPtr <= mid)
        workSpace[j++] = theArray[lowPtr++];
    while(highPtr <= upperBound)
        workSpace[j++] = theArray[highPtr++];

    for(j=0; j<n; j++)
        theArray[lowerBound+j] = workSpace[j];
}  // end merge()
private void merge(long[] workSpace, int lowPtr, int highPtr, int upperBound)
{
    int j = 0; // workspace index
    int lowerBound = lowPtr;
    int mid = highPtr-1;
    int n = upperBound-lowerBound+1; // # of items

    while(lowPtr <= mid && highPtr <= upperBound)
    {
        if( theArray[lowPtr] < theArray[highPtr] )
            workSpace[j++] = theArray[lowPtr++];
        else
            workSpace[j++] = theArray[highPtr++];
    }

    while(lowPtr <= mid)
        workSpace[j++] = theArray[lowPtr++];

    while(highPtr <= upperBound)
        workSpace[j++] = theArray[highPtr++];

    for(j=0; j<n; j++)
        theArray[lowerBound+j] = workSpace[j];
} // end merge()
An important insight about merge sort.
An important insight about merge sort.
An important insight about merge sort.
• Given this insight, you can design a more **direct and efficient** method that avoids recursion:

1. Every element itself is trivially sorted;
2. Start by merging every two adjacent elements;
3. Then merge every four;
4. Then merge every eight;
5. …
6. Done.
Merge Sort

Cost Analysis

• What’s the cost of mergesort?
• Using our insight (the direct method), it’s quite easy to analyze the cost.
Merge Sort

Cost Analysis

• What’s the cost of mergesort?
• Using our insight (the direct method), it’s quite easy to analyze the cost.

\[ O(N \times \log N) \]

This is called log-linear cost.

• How does this compare with Binary Search?
## Merge Sort

Is this a lot better than simple sorting?

<table>
<thead>
<tr>
<th># of elements</th>
<th>N^2</th>
<th>N logN</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>200</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000,000</td>
<td>3,000</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000,000</td>
<td>40,000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Summary

• Merging two sorted array is a key step in merge sort.
• Merge sort uses a divide and conquer approach.
• It repeatedly splits an input array to two sub-arrays, sort each sub-array, and merge the two.
• It requires $O(N\times\log N)$ time.

• On the downsize, it requires additional memory space (the workspace array).
Announcements

• Quiz 6 is available in SPARK.
• It’s shorter than before (8 questions, 15 minutes)
• Pick up midterm exams.