MergeSort - Continued

template <class Comparable>
void mergeSort( vector<Comparable> & a )
{
    vector<Comparable> tmpArray( a.size( ) );
    mergeSort( a, tmpArray, 0, a.size( ) - 1 );
}

template <class Comparable>
void mergeSort( vector<Comparable> & a,
                vector<Comparable> & tmpArray, int left, int right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
template <class Comparable>
void merge( vector<Comparable> & a, vector<Comparable> & tmpArray,
           int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    // Main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd )
    {
        if (a[leftPos] <= a[rightPos])
            tmpArray[tmpPos++] = a[leftPos++];
        else
            tmpArray[tmpPos++] = a[rightPos++];
    }
    while( leftPos <= leftEnd )    // Copy rest of first half
        tmpArray[tmpPos++] = a[leftPos++];
    while( rightPos <= rightEnd )  // Copy rest of right half
        tmpArray[tmpPos++] = a[rightPos++];

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[rightEnd] = tmpArray[rightEnd];
}

---

**Analysis of MergeSort**

- **Mergesor()** is a recursive routine
- There ia general technique to analyze recursive routines
- First we need to write down a *recurrence relation* that expresses the cost of procedure.
  - \( T(N) = \ldots \)
- Assume the *input size to the MergeSort*, \( N \), is a power of 2.
Analysis of MergeSort

- Lets compute the running time
- If \( N = 1 \)
  - The cost of mergesort is \( O(1) \). We will denote this as 1 in \( T(N) \) formula.
- If \( (N > 1) \)
  - The mergesort algorithm consists of:
    - Two mergesorts on input of \( N/2 \). Running time = \( T(N/2) \)
    - A merge routing that is linear with respect to input size. \( O(N) \).
- Then: \( T(N) = 2T(N/2) + 1 \)

Analysis of MergeSort

- We need to solve this recurrence relation!
- One way is like the following:
  - The idea is to expand each recursive part by substitution.
  - \( T(N) = 2 \ T(N/2) + 1 \) (1)
  - \( T(N/2) = 2T(N/4) + 1 \) (2)
  - Substitute \( T(N/2) \) in formula (1)
  - \( T(N) = 2 \ (2 \ T(N/4) + 1) + N \)
Analysis of MergeSort

- Continue doing this
  - \( T(N) = 2 \left( 2 \left( 2T(N/8) + N \right) + N \right) + N \)
    = \( 2^3T(N/2^3) + 3 \)
  - In termination case we have \( T(1) = 1 \)
  - For having \( T(1) = T(N/2^k) \), we should have \( k = \log N \)
- \( k = \log N \)
  - \( T(N) = 2^kT(N/2^k) + kN \)
  - \( T(N) = 2^{\log N}T(N/2^{\log N}) + N\log N \)
  - \( T(N) = NT(1) + N\log N \)
  - \( T(N) = N + N\log N \)

QuickSort

- Fastest known sorting algorithm in practice.
  - For in-memory sorting.
- \( O(N\log N) \) average running time
- \( O(N^2) \) worst-case performance, which can be very rare.
- The inner loop in algorithms is very optimized.
QuickSort - Algorithm

- Input: S – an array of elements of size N.
- Output: S – in sorted order.
  1. If the number of elements in S is 0 or 1, then return.
  2. Pick any element v in S. This is called the pivot
  3. Partition S-{v} into two disjoint groups:
     \[ S_1 = \{ x \in S-v \mid x <= v \} \] and
     \[ S_2 = \{ x \in S-v \mid x >= v \} \]
  4. Return \{quicksort(S_1)\} followed by v followed by \{quicksort(S_2)\}

Example
Partitioning can be performed over the same array.
After partition, the two parts may be equal sized.
Choosing the pivot value is important to have
- Both parts $S_1$ and $S_2$ to have close to equal sizes.

**Picking the Pivot**

- Wrong way:
  - Choose the first element of array
  - What is the array was sorted!
- A safe method
  - Pick it up randomly among the elements of array
  - Depends on the quality of random number generator
- A good method:
  - Pick the median of three elements:
    - First elements
    - Last element
    - Middle element (lowerbound((first+last)/2))
  - Definition: Median of $N$ elements is the lowerbound($N/2$)th largest element.
  - Example: Median of $\{7, 3, 4\}$ is 4.
Partitioning Strategy

- Requires O(N) running time.
  1. First find the pivot.
  2. Then swap the pivot with the last element.
  3. Then do the following operations on elements from first to last-1 (last contains the pivot):
     - Move all elements smaller than pivot to the left of array.
     - Move all elements greater than pivot to the right of array.

Partitioning Strategy

- For Step 3:
  - Keep two index counters: i and j.
    1. Initialize i to first and j to last-1.
    2. While i is smaller or equal to j do
       1. Move i towards right until array[i] > pivot.
       3. Swap array[i] and array[j].
Example

\[
\begin{array}{cccccccc}
8 & 1 & 4 & 9 & 0 & 3 & 5 & 2 & 7 & 6 \\
\text{i} & & & & & & & \text{j} & \text{pivot} \\
\end{array}
\]

Moved j

\[
\begin{array}{cccccccc}
8 & 1 & 4 & 9 & 0 & 3 & 5 & 2 & 7 & 6 \\
\text{i} & & & & & & & \text{j} & \text{pivot} \\
\end{array}
\]

Swapped

\[
\begin{array}{cccccccc}
2 & 1 & 4 & 9 & 0 & 3 & 5 & 8 & 7 & 6 \\
\text{i} & & & & & & & \text{j} & \text{pivot} \\
\end{array}
\]

Swapped

\[
\begin{array}{cccccccc}
2 & 1 & 4 & 9 & 0 & 3 & 5 & 8 & 7 & 6 \\
\text{i} & & & & & & & \text{j} & \text{pivot} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
2 & 1 & 4 & 9 & 0 & 3 & 5 & 8 & 7 & 6 \\
\text{i} & & & & & & & \text{j} & \text{pivot} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
2 & 1 & 4 & 5 & 0 & 3 & 9 & 8 & 7 & 6 \\
\text{i} & & & & & & & \text{j} & \text{pivot} \\
\end{array}
\]

Moved i and j

\[
\begin{array}{cccccccc}
2 & 1 & 4 & 5 & 0 & 3 & 9 & 8 & 7 & 6 \\
\text{i} & & & & & & & \text{j} & \text{pivot} \\
\end{array}
\]

Swapped

\[
\begin{array}{cccccccc}
2 & 1 & 4 & 5 & 0 & 3 & 9 & 8 & 7 & 6 \\
\text{i} & & & & & & & \text{j} & \text{pivot} \\
\end{array}
\]

i crossed j

\[
\begin{array}{cccccccc}
2 & 1 & 4 & 5 & 0 & 3 & 9 & 8 & 7 & 6 \\
\text{i} & & & & & & & \text{j} & \text{pivot} \\
\end{array}
\]

STOP
After swapping pivot (last element) with array[i])

Part 1

2 1 4 5 0 3

Part 2

8 7 9

call quicksort recursively on these parts

QuickSort Code

template <class Comparable>
void quicksort( vector<Comparable> & a )
{
    quicksort( a, 0, a.size( ) - 1 );
}

template <class Comparable>
const Comparable &median3( vector<Comparable> & a, int left, int right )
{
    int center = ( left + right ) / 2;
    if( a[ center ] < a[ left ] )
        swap( a[ left ], a[ center ] );
    if( a[ right ] < a[ left ] )
        swap( a[ left ], a[ right ] );
    if( a[ right ] < a[ center ] )
        swap( a[ center ], a[ right ] );
    swap( a[ center ], a[ right - 1 ] ); // Place pivot at position right - 1
    return a[ right - 1 ];
}
template <class Comparable>
void quicksort( vector<Comparable> & a, int left, int right )
{
    /* 1*/   if( left + 10 <= right )
    { /* 2*/          Comparable pivot = median3( a, left, right );
        // Begin partitioning
    /* 3*/          int i = left, j = right - 1;
    /* 4*/          for( ; ; )
    { /* 5*/              while( a[ ++i ] < pivot ) { } ; // move i to right
    /* 6*/              while( pivot < a[ --j ] ) { } ; // move j to left
    /* 7*/              if( i < j )
    /* 8*/                  swap( a[ i ], a[ j ] ); // swap array[i] with array[j]
                else
    /* 9*/                  break;
            }
    /*10*/          swap( a[ i ], a[ right - 1 ] ); // Restore pivot – put pivot at ith position
    /*11*/          quicksort( a, left, i - 1 ); // Sort small elements – recursive call
    /*12*/          quicksort( a, i + 1, right ); // Sort large elements
    }
    else // Do an insertion sort on the subarray if array size is smaller than 10
    /*13*/          insertionSort( a, left, right );
}

Analysis of Quicksort

- It is a recursive algorithm like mergesort.
- We will again use recurrence relations
- We will analyze of 3 cases
  - Worst case
  - Best case
  - Average case
Analysis of Quicksort

- For \( N = 1 \) or \( N = 0 \)
  - \( T(N) = 1 \)
- For \( (N > 1) \)
  - Running time \( T(N) \) is equal to the running time of the two recursive calls plus the linear time spent in partitioning
  - \( T(N) = T(i) + T(N-i-1) + cN \), where \( i \) is the number of elements in the first part \( S_1 \)

Worst Case (\( i = 0 \)) Analysis

The pivot is the smallest element all the time, \( i = 0 \)

\[
\begin{align*}
T(N) &= T(N-1) + cN, \quad N > 1 \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
& \quad \vdots \\
T(2) &= T(1) + c(2)
\end{align*}
\]

Adding them all yields:

\[
T(N) = T(1) + c \sum_{i=2}^{N} i = O(N^2)
\]
**Best Case (i ~= array.size()/2) Analysis**

- The pivot is in the middle
  \[ T(N) = 2T(N/2) + cN, \quad N > 1 \]
- Similar to mergesort analysis
  \[ T(N) = cN \log N + N = O(N \log N) \]

---

**Average-Case Analysis**

- Each of the sizes of S1 is equally likely.
- The sizes are in range \{0, ..., N-1\}
- The probability of an array having one of these sizes is: \(1/N\)
- Assuming partitioning strategy is random
  - Otherwise analysis is not correct!
- The average value of \(T(i)\) is like the following:
  \[ T(i) = \frac{1}{N} \sum_{j=0}^{N-1} T(j) = T(N - i - 1) \]
Average-Case Analysis

\[ T(N) = \frac{1}{N} \left( \sum_{j=0}^{N-1} T(j) \right) + cN \]  
Equation 1

Multiply the above equation by N

\[ NT(N) = 2 \left( \sum_{j=0}^{N-1} T(j) \right) + cN^2 \]  
Equation 2

Substitute N with N - 1

\[ (N - 1)T(N - 1) = 2 \left( \sum_{j=0}^{N-2} T(j) \right) + c(N - 1)^2 \]  
Equation 3

Average-Case Analysis

Subtract equation 3 from equation 2

\[ NT(N) - (N - 1)T(N - 1) = 2T(N - 1) + 2cN - c \]
\[ NT(N) = (N + 1)T(N - 1) + 2cN \]  
(ignoring c)

divide both sides with \( N(N + 1) \)

\[ \frac{T(N)}{N + 1} = \frac{T(N - 1)}{N} + \frac{2c}{N + 1} \]  
E1
Average-Case Analysis

Now telescope (write down equations depending on smaller N)

\[
\begin{align*}
\frac{T(N-1)}{N} &= \frac{T(N-2)}{N-1} + \frac{2c}{N} \quad \text{E2} \\
\frac{T(N-2)}{N-1} &= \frac{T(N-3)}{N-2} + \frac{2c}{N-1} \\
\frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2c}{3} \quad \text{E4}
\end{align*}
\]

Average-Case Analysis

Add all these equations E1 through E4 and obtain:

\[
\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} \frac{1}{i}
\]

The sum is:

\[
\sum_{i=3}^{N+1} \frac{1}{i} = \log_e(N+1) + 0.577 - \frac{3}{2}
\]

Then

\[
\frac{T(N)}{N+1} = O(\log N)
\]

So the result is

\[
T(N) = O(N \log N)
\]
External Sorting

- So far we have assumed that all the input data can fit into main memory (RAM)
  - This means random access to data is possible and is not very costly.
- Algorithms such as shell-sort, and quick-sort make random access to array elements.
- If data is in a hard-disk or in a tape (in a file) random access is very costly.
- External sorting algorithms deal with these cases and can sort very large input sizes.

External Sorting

- External sorting algorithms makes sequential accesses to a storage device.
  - Tape or hard-disk.
  - In this way, the setup cost of retrieval is got rid of.
- Our model for external devices are (tapes)
  - They will be read from and written to sequentially.
    - In forward or reverse direction.
  - We can rewind the head to the beginning of the device (tape)
  - Assume we have at least three tape drives.
The Simple Algorithm

- Uses the merge idea from mergesort.
- Assume data is stored in a tape.
- Assume we have 4 tapes available.
- We will read M items at a time from input tape.
- We will sort them in memory and write to one of the output tapes. (set of M items will be called a Run)
- We will continue doing this until we finish with the input.
- Then we will go to the merge step.

Algorithm Sketch

1. **Constructing the runs**
   1. If tape 1 is not finished
      1. Read M items (if available) from tape 1
      2. Sort them in memory
      3. Write them to tape 3 (these M items is called one run)
   2. If tape 1 is not finished
      1. Read M items (if available) from tape 1
      2. Sort them in memory
      3. Write them to tape 4
   3. Repeat steps 1 and 2 until tape 1 is finished.

2. **Merging runs**
   1. Merge runs in tapes 3 and 4 into tape 1 and 2.
      1. By taking one run from tape 3 and one run from tape 4.
      2. Continue in this way
      At the end of this we have runs of size 2*M in tape 1 and 2
   2. Merge runs in tape 1 and 2 into tapes 3 and 4.
      At the end of this we have runs of size 4*M in tape 3 and 4.
   3. Repeat steps 1 and 3 until we have a single run of size N (input size)
Idea – constructing the runs

input

1  2  3  4  5  6  7  8
Tape 1

Memory

1  
3  
5  
7  
Tape 3

2  
3  
4  
5  
Tape 4

---

Idea – merging the runs

Pass 1

1  
3  
5  
7  
Tape 3

2  
4  
6  
8  
Tape 4

Pass 2

1.2  
5.6  
Tape 1

3.4  
7.8  
Tape 2

Pass 3

1,2,3,4,5,6,7,8  
Tape 3
### Example (M=3)

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>81</td>
<td>94</td>
<td>11</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>35</td>
<td>19</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>15</td>
<td>28</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>99</td>
<td>35</td>
<td>94</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>41</td>
<td>96</td>
<td>17</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>11</td>
<td>91</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>17</td>
<td>28</td>
<td>58</td>
</tr>
</tbody>
</table>

**After constructing the runs**

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>12</td>
<td>17</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>96</td>
<td>41</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>99</td>
<td>75</td>
<td>99</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>28</td>
</tr>
</tbody>
</table>

**After first pass**

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>12</td>
<td>35</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>96</td>
<td>15</td>
<td>94</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>99</td>
<td>41</td>
<td>58</td>
</tr>
</tbody>
</table>

**After second pass**

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>12</td>
<td>17</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>35</td>
<td>51</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>81</td>
<td>94</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
</tbody>
</table>
After third pass

<table>
<thead>
<tr>
<th>T1</th>
<th>11</th>
<th>12</th>
<th>15</th>
<th>17</th>
<th>28</th>
<th>35</th>
<th>51</th>
<th>58</th>
<th>75</th>
<th>81</th>
<th>94</th>
<th>96</th>
<th>99</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Polyphase Merge

- In the previous example, we have used 4 tapes.
  - We did 2-way merge
  - It is possible to use 3 tapes in 2-way merge
- We can perform k-way merge similarly.
  - We need 2k tapes for simple algorithm
  - We need k+1 tapes for polyphase merge
Polyphase merge

- The idea is to not put the runs evenly to output tapes.
  - Some tapes should have more runs than the others.
- For two way merge
  - Have the number of runs in output tapes according to the Fibonacci numbers
    - Input = 8 ➔ output tape 1 = 3, output tape 2 = 5
    - Input = 13 ➔ output tape 1 = 5, output tape 2 = 8
    - Input = 21 ➔ output tape 1 = 13, output tape 2 = 8
    - …
  - Add some dummy items to input if the size is not Fibonacci.

Assume N = 33 (input size)

<table>
<thead>
<tr>
<th></th>
<th>After T3+T2</th>
<th>After T1+T2</th>
<th>After T1+T3</th>
<th>After T2+T3</th>
<th>After T1+T3</th>
<th>After T2+T3</th>
<th>After T2+T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>13</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T2</td>
<td>21</td>
<td>8</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T3</td>
<td>13</td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

All run sizes are Fibonacci numbers.
Replacement Selection

- A method for constructing the runs.
- Will produce variable sized runs.
  - All runs do not have equal sizes

Example

Input tape

Read M elements

BuildHeap

memory

deleteMin

Output tape

<table>
<thead>
<tr>
<th>T1</th>
<th>81</th>
<th>94</th>
<th>11</th>
<th>96</th>
<th>12</th>
<th>35</th>
<th>17</th>
<th>99</th>
<th>28</th>
<th>58</th>
<th>41</th>
<th>75</th>
<th>15</th>
</tr>
</thead>
</table>

| T2  | 11  |     |     |     |     |     |     |     |     |     |     |     |     |
T1  
81 94 11 96 12 35 17 99 28 58 41 75 15

Input tape

Read next element
96

Is 96 > 11

Yes, put it into heap
81 94 96

deleteMin

T2  
11 81

Output tape

T1  
81 94 11 96 12 35 17 99 28 58 41 75 15

Input tape

Read next element
12

Is 12 > 81

No, Don't include in heap
94 96 12

deleteMin

T2  
11 81 94

Output tape
**Input tape**

Read next element

35

Is $35 > 94$

No, Don’t include in heap

96 35 12

**Output tape**

**Input tape**

Read next element

17

Is $17 > 35$

No, Don’t include in heap

17 35 12

**Output tape**

We have empty heap. Mark end of run!

17 35 12

E