Sorting Algorithms
Sorting methods

- Comparison based sorting
  - $O(n^2)$ methods
    - E.g., Insertion, bubble
  - Average time $O(n \log n)$ methods
    - E.g., quick sort
  - $O(n \log n)$ methods
    - E.g., Merge sort, heap sort

- Non-comparison based sorting
  - Integer sorting: linear time
    - E.g., Counting sort, bin sort
  - Radix sort, bucket sort

- Stable vs. non-stable sorting
Insertion sort: snapshot at a given iteration

Worst-case run-time complexity: $\Theta(n^2)$
Best-case run-time complexity: $\Theta(n)$

Cpt S 223. School of EECS, WSU
The Divide and Conquer Technique

- **Input:** A problem of size n

- Recursive

- At each level of recursion:
  - (Divide)
    - Split the problem of size n into a fixed number of sub-problems of smaller sizes, and solve each sub-problem recursively
  - (Conquer)
    - Merge the answers to the sub-problems
Two Divide & Conquer sorts

- **Merge sort**
  - Divide is trivial
  - Merge (i.e, conquer) does all the work

- **Quick sort**
  - Partition (i.e, Divide) does all the work
  - Merge (i.e, conquer) is trivial
Merge Sort

Main idea:
- Dividing is trivial
- Merging is non-trivial

Input

(divide)

How much work at every step?

O(n) sub-problems

(conquer)

How much work at every step?

O(lg n) steps

O(lg n) steps

Image courtesy: McQuain WD, VA Tech, 2004

Cpt S 223. School of EECS, WSU
How to merge two sorted arrays?

1. \( B[k++] = \text{Populate min}\{ A1[i], A2[j] \} \)
2. Advance the minimum contributing pointer

Temporary array to hold the output

\( \Theta(n) \) time

Do you always need the temporary array \( B \) to store the output, or can you do this inplace?
Merge Sort Analysis

**Merge Sort takes $\Theta(n \log n)$ time**

**Proof:**
- Let $T(n)$ be the time taken to merge sort $n$ elements.
- Time for each comparison operation $= O(1)$

**Main observation:** To merge two *sorted* arrays of size $n/2$, it takes $n$ comparisons at most.

Therefore:
- $T(n) = 2T(n/2) + n$
- Solving the above recurrence:
  - $T(n) = 2[2T(n/2^2) + n/2] + n$
    - $= 2^2 T(n/2^2) + 2n$
    - ... $(k$ times$)$
    - $= 2^k T(n/2^k) + kn$
- At $k = \log n$, $T(n/2^k) = T(1) = 1$ (termination case)
- $\implies T(n) = \Theta(n \log n)$
QuickSort

- Divide-and-conquer approach to sorting
- Like MergeSort, except
  - Don’t divide the array in half
  - Partition the array based elements being less than or greater than some element of the array (the pivot)
  - i.e., divide phase does all the work; merge phase is trivial.
- Worst case running time $O(N^2)$
- Average case running time $O(N \log N)$
- Fastest generic sorting algorithm in practice
- Even faster if use simple sort (e.g., InsertionSort) when array becomes small

Main idea:
- Dividing ("partitioning") is non-trivial
- Merging is trivial
QuickSort Algorithm

QuickSort(Array: S)

1. If size of S is 0 or 1, return

2. Pivot = Pick an element v in S

1. Partition S – {v} into two disjoint groups
   - S1 = \{x \in (S – \{v\}) | x < v\}
   - S2 = \{x \in (S – \{v\}) | x > v\}

2. Return QuickSort(S1), followed by v, followed by QuickSort(S2)

Q) What's the best way to pick this element? (arbitrary? Median? etc)
QuickSort vs. MergeSort

- **Main problem with quicksort:**
  - QuickSort may end up dividing the input array into subproblems of size 1 and N-1 in the worst case, at every recursive step (unlike merge sort which always divides into two halves)
    - When can this happen?
    - Leading to $O(N^2)$ performance
  
  => Need to choose pivot wisely (but efficiently)

- **MergeSort** is typically implemented using a temporary array (for merge step)
  - QuickSort can partition the array “in place”
Goal: A “good” pivot is one that creates two even sized partitions

=> Median will be best, but finding median could be as tough as sorting itself

Picking the Pivot

How about choosing the first element?

- What if array already or nearly sorted?
- Good for a randomly populated array

How about choosing a random element?

- Good in practice if “truly random”
- Still possible to get some bad choices
- Requires execution of random number generator
Picking the Pivot

- Best choice of pivot
  - Median of array
  - But median is expensive to calculate

- **Next strategy: Approximate the median**
  - *Estimate* median as the median of any three elements
    
    Median = median \{first, middle, last\}

    Has been shown to reduce running time (comparisons) by 14%
How to write the partitioning code?

- Goal of partitioning:
  - i) Move all elements < pivot to the left of pivot
  - ii) Move all elements > pivot to the right of pivot

- Partitioning is conceptually straightforward, but easy to do inefficiently

- One bad way:
  - Do one pass to figure out how many elements should be on either side of pivot
  - Then create a temp array to copy elements relative to pivot
Partitioning strategy

- A good strategy to do partition: **do it in place**

  ```
  // Swap pivot with last element S[right]
  i = left
  j = (right – 1)
  While (i < j) {
    // advance i until first element > pivot
    // decrement j until first element < pivot
    // swap A[i] & A[j] (only if i<j)
  }
  Swap (pivot, S[i])
  ```

OK to also swap with S[left] but then the rest of the code should change accordingly.

This is called "in place" because all operations are done in place of the input array (i.e., without creating temp array).
Partitioning Strategy

- An in place partitioning algorithm

- **Swap pivot with last element** $S[right]$

- $i = left$

- $j = (right – 1)$

- **while** ($i < j$)
  - { $i++; }$ until $S[i] > pivot$
  - { $j--; }$ until $S[j] < pivot$
  - If ($i < j$), then swap($S[i], S[j]$)

- **Swap (pivot, S[i])**

Needs a few boundary case handling
“Median of three” approach to picking the pivot:
=> compares the first, last and middle elements and pick the median of those three
pivot = min{8, 6, 0} = 6

Partitioning Example

Initial array

Swap pivot; initialize i and j

Move i and j inwards until conditions violated

After first swap

While (i < j) {
    { i++; } until S[i] > pivot
    { j--; } until S[j] < pivot
    If (i < j), then swap( S[i], S[j] )
}

Swap pivot with last element S[right]
i = left
j = (right – 1)
Partitioning Example (cont.)

After a few steps ...

Before second swap

After second swap

i has crossed j

After final swap with pivot

Swap (pivot, S[i])
Handling Duplicates

What happens if all input elements are equal?

- **Current approach:**
  - `{ i++; } until S[i] > pivot`
  - `{ j--; } until S[j] < pivot`

- **Special case:** 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6

- **What will happen?**
  - i will advance all the way to the right end
  - j will advance all the way to the left end

- => pivot will remain in the right position, creating the left partition to contain N-1 elements and empty right partition
  - Worst case O(N^2) performance
Handling Duplicates

- A better code
  - Don’t skip elements equal to pivot
    - \( \{ i++; \} \text{ until } S[i] \geq \text{pivot} \)
    - \( \{ j--; \} \text{ until } S[j] \leq \text{pivot} \)
  
  Special case: 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
  
  What will happen now?

- Adds some unnecessary swaps
- But results in perfect partitioning for array of identical elements
  - Unlikely for input array, but more likely for recursive calls to QuickSort
Small Arrays

- When S is small, recursive calls become expensive (*overheads*)

- General strategy
  - When size < threshold, use a sort more efficient for small arrays (e.g., InsertionSort)
  - Good thresholds range from 5 to 20
  - Also avoids issue with finding median-of-three pivot for array of size 2 or less
  - Has been shown to reduce running time by 15%
QuickSort Implementation

```cpp
/**
 * Quicksort algorithm (driver).
 */

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

```cpp
/**
 * Return median of left, center, and right.
 * Order these and hide the pivot.
 */

template <typename 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 ] );

  // Place pivot at position right - 1
  swap( a[ center ], a[ right - 1 ] );
  return a[ right - 1 ];
}
```

Cpt S 223. School of EECS, WSU
/**
 * Internal quicksort method that makes recursive calls.
 * Uses median-of-three partitioning and a cutoff of 10.
 * a is an array of Comparable items.
 * left is the left-most index of the subarray.
 * right is the right-most index of the subarray.
 */

template<typename Comparable>
void quicksort(vector<Comparable> & a, int left, int right)
{
    if(left + 10 <= right)
    {
        Comparable pivot = median3(a, left, right);

        // Begin partitioning
        int i = left, j = right - 1;
        for( ; ; )
        {
            while(a[++i] < pivot) {} // sorted into a smaller set
            while(pivot < a[--j]) {} // sorted into a larger set
            if(i < j)
                swap(a[i], a[j]);
            else
                break;
        }

        swap(a[i], a[right - 1]); // Restore pivot
        quicksort(a, left, i - 1); // Sort small elements
        quicksort(a, i + 1, right); // Sort large elements
    }
    else // Do an insertion sort on the subarray
        insertionSort(a, left, right);
}
Analysis of QuickSort

- Let $T(N) =$ time to quicksort $N$ elements
- Let $L =$ number of elements in left partition
  => number of elements in right partition $= N - L - 1$

- **Base:** $T(0) = T(1) = O(1)$
- $T(N) = T(L) + T(N - L - 1) + O(N)$
Analysis of QuickSort

- **Worst-case analysis**
  - Pivot is the smallest element \((L = 0)\)
  
  \[
  T(N) = T(0) + T(N - 1) + O(N)
  \]

  
  
  \[
  = O(1) + T(N - 1) + O(N)
  \]

  
  
  \[
  = T(N - 1) + O(N)
  \]

  
  
  \[
  = T(N - 2) + O(N - 1) + O(N)
  \]

  
  
  \[
  = T(N - 3) + O(N - 2) + O(N - 1) + O(N)
  \]

  
  
  \[
  = \sum_{i=1}^{N} O(i) = O(N^2)
  \]
Analysis of QuickSort

- **Best-case analysis**
  - Pivot is the median (sorted rank = N/2)
  
  \[ T(N) = T(N/2) + T(N/2) + O(N) \]
  
  \[ = 2T(N/2) + O(N) \]
  
  \[ = O(N \log N) \]

- **Average-case analysis**
  - Assuming each partition equally likely
  
  \[ T(N) = O(N \log N) \]  HOW?
QuickSort: Avg Case Analysis

- \( T(N) = T(L) + T(N-L-1) + O(N) \)
  
  *All partition sizes are equally likely*

\[ \Rightarrow \text{Avg } T(L) = \text{Avg } T(N-L-1) = \frac{1}{N} \sum_{j=0}^{N-1} T(j) \]

\[ \Rightarrow \text{Avg } T(N) = \frac{2}{N} \left[ \sum_{j=0}^{N-1} T(j) \right] + cN \]

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

*Substituting N by N-1 ...

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

\[(1)-(2) \]

\[ \Rightarrow N \cdot T(N) - (N-1) \cdot T(N-1) = 2 \cdot T(N-1) + c(2N-1) \]
Avg case analysis ...

- \( NT(N) = (N+1)T(N-1) + c (2N-1) \)
- \( T(N)/(N+1) \approx T(N-1)/N + c2/(N+1) \)
- Telescope, by substituting \( N \) with \( N-1, N-2, N-3, \ldots \) 2
- ...
- \( T(N) = O(N \log N) \)
## Comparison Sorting

<table>
<thead>
<tr>
<th>Sort</th>
<th>Worst Case</th>
<th>Average Case</th>
<th>Best Case</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertionSort</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N)$</td>
<td>Fast for small $N$</td>
</tr>
<tr>
<td>MergeSort</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>Requires memory</td>
</tr>
<tr>
<td>HeapSort</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>Large constants</td>
</tr>
<tr>
<td>QuickSort</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
<td>Small constants</td>
</tr>
</tbody>
</table>
## Comparison Sorting

<table>
<thead>
<tr>
<th>$N$</th>
<th>Insertion Sort $O(N^2)$</th>
<th>Shellsort $O(N^{7/6})(?)$</th>
<th>Heapsort $O(N \log N)$</th>
<th>Quicksort $O(N \log N)$</th>
<th>Quicksort (opt.) $O(N \log N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000001</td>
<td>0.000002</td>
<td>0.000003</td>
<td>0.000002</td>
<td>0.000002</td>
</tr>
<tr>
<td>100</td>
<td>0.000106</td>
<td>0.000039</td>
<td>0.000052</td>
<td>0.000025</td>
<td>0.000023</td>
</tr>
<tr>
<td>1000</td>
<td>0.011240</td>
<td>0.000678</td>
<td>0.000750</td>
<td>0.000365</td>
<td>0.000316</td>
</tr>
<tr>
<td>10000</td>
<td>1.047</td>
<td>0.009782</td>
<td>0.010215</td>
<td>0.004612</td>
<td>0.004129</td>
</tr>
<tr>
<td>100000</td>
<td>110.492</td>
<td>0.13438</td>
<td>0.139542</td>
<td>0.058481</td>
<td>0.052790</td>
</tr>
<tr>
<td>1000000</td>
<td>NA</td>
<td>1.6777</td>
<td>1.7967</td>
<td>0.6842</td>
<td>0.6154</td>
</tr>
</tbody>
</table>

### Good sorting applets
- [http://math.hws.edu/TMCM/java/xSortLab/](http://math.hws.edu/TMCM/java/xSortLab/)

### Sorting benchmark:
- [http://sortbenchmark.org/](http://sortbenchmark.org/)
Lower Bound on Sorting

What is the best we can do on comparison based sorting?

- Best worst-case sorting algorithm (so far) is $O(N \log N)$
  - Can we do better?

- Can we prove a lower bound on the sorting problem, independent of the algorithm?
  - For comparison sorting, no, we can’t do better than $O(N \log N)$
  - Can show lower bound of $\Omega(N \log N)$
Proving lower bound on sorting using “Decision Trees”

A decision tree is a binary tree where:

- Each node
  - lists all left-out open possibilities (for deciding)
- Path of each node
  - represents a decided sorted prefix of elements
- Each branch
  - represents an outcome of a particular comparison
- Each leaf
  - represents a particular ordering of the original array elements
A decision tree to sort three elements \{a, b, c\} (assuming no duplicates)

- **Root** = all open possibilities

- **IF** \( a < b \):
  - all remaining open possibilities
  - Height = \( \Omega(\lg n!) \)

- **IF** \( a < c \):
  - possible

- **Worst-case evaluation path for any algorithm**

- \( n! \) leaves in this tree
Decision Tree for Sorting

- The logic of *any sorting algorithm* that uses comparisons can be represented by a decision tree.

- In the worst case, the number of comparisons used by the algorithm equals the HEIGHT OF THE DECISION TREE.

- In the average case, the number of comparisons is the average of the depths of all leaves.

- There are N! different orderings of N elements.
Lower Bound for Comparison Sorting

Lemma: A binary tree with L leaves must have depth at least \( \lceil \lg L \rceil \)

Sorting’s decision tree has \( N! \) leaves

Theorem: Any comparison sort may require at least \( \lceil \log(N!) \rceil \) comparisons in the worst case
Lower Bound for Comparison Sorting

Theorem: Any comparison sort requires $\Omega(N \log N)$ comparisons

Proof (uses Stirling’s approximation)

\[ N! \approx \sqrt{2\pi N} \left( \frac{N}{e} \right)^N (1 + \Theta(1/N)) \]

\[ N! > \left( \frac{N}{e} \right)^N \]

\[ \log(N!) > N \log N - N \log e = \Theta(N \log N) \]

\[ \log(N!) > \Theta(N \log N) \]

\[ \log(N!) = \Omega(N \log N) \]
Implications of the sorting lower bound theorem

- Comparison based sorting cannot be achieved in less than \((n \lg n)\) steps

  => Merge sort, Heap sort are optimal

  => Quick sort is not optimal but pretty good as optimal in practice

  => Insertion sort, bubble sort are clearly sub-optimal, even in practice
Non comparison based sorting

Integer sorting
  e.g., Counting sort
  Bucket sort
  Radix sort
Integer Sorting

- Some input properties allow to eliminate the need for comparison
  - E.g., sorting an employee database by age of employees

- Counting Sort
  - *Given array A[1..N], where 1 ≤ A[i] ≤ M*
  - Create array C of size M, where C[i] is the number of i’s in A
  - Use C to place elements into new sorted array B
  - Running time $\Theta(N+M) = \Theta(N)$ if $M = \Theta(N)$
Counting Sort: Example

Input A: 0 3 2 1 3 2 1 2 2 3

(all elements in input between 0 and 3)

Count array C:

Output sorted array:

Time = \( O(N + M) \)

If \( M < N \), Time = \( O(N) \)

Cpt S 223. School of EECS, WSU
Stable vs. nonstable sorting

- A “stable” sorting method is one which preserves the original input order among duplicates in the output.

Input: 0 3 2 1 3 2 1 2 2 3

Output: 0 1 1 2 2 2 2 3 3 3

Useful when each data is a struct of form { key, value }
How to make counting sort “stable”? (one approach)

Input A:

<table>
<thead>
<tr>
<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

(count all elements in input between 0 and 3)

Count array C:

Output sorted array:

<table>
<thead>
<tr>
<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

But this algorithm is NOT in-place!

Can we make counting sort in place?

(i.e., without using another array or linked list)
How to make counting sort in place?

```cpp
void CountingSort_InPlace(vector a, int n) {
    1. First construct Count array C s.t C[i] stores the last index in the bin corresponding to key i, where the next instance of i should be written to. Then do the following:

    i=0;
    while(i<n) {
        e=A[i];
        if c[e] has gone below range, then continue after i++;
        if(i==c[e]) i++; // Note: This code has to keep track of the valid range for each key
        tmp = A[c[e]];
        A[c[e]] = e;
        A[i] = tmp;
    }
}
```

Cpt S 223. School of EECS, WSU
Cpt S 223. School of EECS, WSU

End points

<table>
<thead>
<tr>
<th>C:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A:

0     | 3 | 2 | 1 | 3 | 2 | 1 | 2 | 2 | 3
0     | 3 | 2 | 1 | 3 | 2 | 1 | 2 | 2 | 3
0     | 2 | 2 | 1 | 3 | 2 | 1 | 2 | 3 | 3
0     | 1 | 2 | 1 | 3 | 2 | 2 | 2 | 3 | 3
0     | 2 | 1 | 1 | 3 | 2 | 2 | 2 | 3 | 3
0     | 3 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3
0     | 3 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3
0     | 2 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3
0     | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3
Bucket sort

- Assume N elements of A uniformly distributed over the range [0,1]
- Create M equal-sized buckets over [0,1], s.t., M ≤ N
- Add each element of A into appropriate bucket
- Sort each bucket internally
  - Can use recursion here, or
  - Can use something like InsertionSort
- Return concatenation of buckets
- Average case running time Θ(N)
  - assuming each bucket will contain Θ(1) elements
Radix Sort

- Sort N numbers, each with k bits
- E.g., input \{4, 1, 0, 10, 5, 6, 1, 8\}

\[
\begin{align*}
4 & \quad 0100 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 4 \\
1 & \quad 0001 \quad 0000 \quad 0000 \quad 1000 \quad 0001 \quad 1 \\
0 & \quad 0000 \quad 1010 \quad 1000 \quad 0001 \quad 0001 \quad 1 \\
10 & \quad 1010 \quad 0110 \quad 0001 \quad 0001 \quad 0010 \quad 4 \\
5 & \quad 0101 \quad 1000 \quad 0101 \quad 1010 \quad 0101 \quad 5 \\
6 & \quad 0110 \quad 0001 \quad 0001 \quad 0100 \quad 0110 \quad 6 \\
1 & \quad 0001 \quad 0101 \quad 1010 \quad 0101 \quad 1000 \quad 8 \\
8 & \quad 1000 \quad 0001 \quad 0110 \quad 0110 \quad 1010 \quad 10
\end{align*}
\]

lsb \quad msb

Radix sort achieves stable sorting
To sort each column, use counting sort (O(n))
=> To sort k columns, O(nk) time

Cpt S 223. School of EECS, WSU
External Sorting

- What if the number of elements N we wish to sort do not fit in memory?
- Obviously, our existing sort algorithms are inefficient
  - Each comparison potentially requires a disk access
- Once again, we want to minimize disk accesses
External MergeSort

- \( N = \) number of elements in array \( A[1..N] \) to be sorted
- \( M = \) number of elements that fit in memory at any given time
- \( K = \left\lceil \frac{N}{M} \right\rceil \)
External MergeSort

**Approach**

1. Read in $M$ amount of $A$, sort it using local sorting (e.g., quicksort), and write it back to disk
2. Repeat above $K$ times until all of $A$ processed
3. Create $K$ input buffers and 1 output buffer, each of size $M/(K+1)$
4. Perform a *K-way merge*:
   1. Update input buffers one disk-page at a time
   2. Write output buffer one disk-page at a time
K-way merge

Q) How to merge $k$ sorted arrays of total size $N$ in $O(N \log k)$ time?

- For external merge sort:
  
  $r = M/(k+1)$ and $\sum_{i=0}^{k} |L_i| = M$
K-way merge – a simple algo

Input:

\[ L_1, L_2, L_3, L_4, \ldots, L_{k-1}, L_k \]

Sum of sorted list lengths

\[ = N \ (= kr) \]

Output

merged arrays
(temp)

Q) What is the problem with this approach?

• k-1 stages

• Total time

\[ = 2r + 3r + 4r + 5r + \ldots + kr \]

\[ = O(k^2r) \]

\[ = O(Nk) \]

• Even worse, if individual input arrays are of variable sizes

We can do better than this!

Cpt S 223. School of EECS, WSU
K-way merge – a better algo

Input:

sorted

Output

merged arrays (temp)

Run-time Analysis

- \( \lg k \) stages
- Total time

\[ = (N) + (N) + \ldots : \lg k \text{ times} \]

\[ = O(N \lg k) \]

Sum of sorted list lengths

\[ = N \quad (= kr) \]
External MergeSort

- Computational time $T(N,M)$:
  
  $= O(K \times M \log M) + O(N \log K)$
  
  $= O((N/M) \times M \log M) + O(N \log K)$
  
  $= O(N \log M + N \log K)$
  
  $= O(N \log M)$

- Disk accesses (all sequential)
  
  - $P = \text{page size}$
  
  - Accesses $= O(N/P)$
Sorting: Summary

- Need for sorting is ubiquitous in software
- Optimizing the sort algorithm to the domain is essential
- Good general-purpose algorithms available
  - QuickSort
- Optimizations continue...
  - Sort benchmarks

http://sortbenchmark.org/
http://research.microsoft.com/barc/sortbenchmark