Sorting

CPE 212 -- Lecture 18

Sorting

- Arranging a collection of data values into an order
  - Examples:
    - Alphabetizing a list of names
    - Sorting exams by grade received

Sorting Algorithm Efficiency

- Speed
  - Number of comparisons made
  - Number of swaps required
- Space
  - Memory space required
- Typical Tradeoff
  - More memory => less time required

$O(N^2)$ Sorting Algorithms

- Selection Sort
- Bubble Sort
- Insertion Sort

Selection Sort Example

Number Of Elements N = 4

Unsorted

Sorted

values (i=0)

[0]	18
[1]	9
[2]	5
[3]	10

i=0

values (i=1)

[0]	5
[1]	9
[2]	18
[3]	10

i=1

values (i=2)

[0]	5
[1]	9
[2]	18
[3]	10

i=2

values (i=3)

[0]	5
[1]	9
[2]	10
[3]	18

i=3

UAHuntsville

Selection Sort Algorithm

Concept: Repeatedly select smallest unsorted element and move it to the front of the unsorted elements

Suppose the unsorted elements are in variables A_1 through A_N

For each pass i for 1 <= i <= N-1,

- Find the smallest unsorted element in the range A_i through A_N
- Swap that element with element A_i
- Now A_1 through A_i contain the i smallest elements
- Repeat for each i listed above

Selection Sort Implementation

```
template<class ItemType>
void Swap(ItemType& item1, ItemType& item2)
// Post: Contents of item1 and item2 have been swapped
{
  ItemType tempItem;

  tempItem = item1;
  item1 = item2;
  item2 = tempItem;
}

template<class ItemType>
int MinIndex(ItemType values[], int startIndex, int endIndex)
// Post: Returns the index of the smallest value in
//       values[startIndex]..values[endIndex].
{
  int indexOfMin = startIndex;
  for (int index = startIndex + 1; index <= endIndex; index++)
    if (values[index] < values[indexOfMin])
      indexOfMin = index;
  return indexOfMin;
}

template<class ItemType>
void SelectionSort(ItemType values[], int numValues)
// Post: The elements in the array values are sorted by key.
{
  int endIndex = numValues-1;
  for (int current = 0; current < endIndex; current++)
    Swap(values[current], values[MinIndex(values, current, endIndex)]);
}
```

UAHuntsville

Selection Sort Analysis

- Comparison Analysis
  - First position requires N-1 comparisons
  - Second position requires N-2 comparisons
  - Third position requires N-3 comparisons

  …
  - Position N-1 requires 1 comparison

$$(N-1) + (N-2) + (N-3) + \ldots + 1 = N(N-1)/2 = O(N^2)$$

- Swap Analysis
  - Number of Swaps Performed = N - 1 = O(N)

Bubble Sort Example

Number Of Unsorted Elements
N = 4

Unsorted
Sorted

values

[0]	18
[1]	10
[2]	9
[3]	5

values

[0]	18
[1]	10
[2]	5
[3]	9

i=0

3 comparisons

values

[0]	18
[1]	5
[2]	10
[3]	9

values

[0]	5
[1]	18
[2]	10
[3]	9

Bubble Sort Example

Number Of Unsorted Elements N = 3

Unsorted

Sorted

values

[0]	5
[1]	18
[2]	10
[3]	9

values

[0]	5
[1]	18
[2]	9
[3]	10

i=1

2 comparisons

values

[0]	5
[1]	9
[2]	18
[3]	10

UAHuntsville

Bubble Sort Example

Number Of Unsorted Elements N = 2

☐ **Unsorted**
■ **Sorted**

values

[0]	5
[1]	9
[2]	18
[3]	10

i=2

1 comparison

values

[0]	5
[1]	9
[2]	10
[3]	18

Bubble Sort Algorithm

Concept: “Lighter” elements bubble to the top

Suppose the unsorted elements are in variables A_1 through A_N

For each pass i for 1 <= i <= N-1,

- Bubble the smallest unsorted element in the range A_i through A_N up to position A_i by swapping as needed
- Now A_1 through A_i contain the i smallest elements
- Shrink the unsorted porition of the array by incrementing i across the range listed above

Bubble Sort Implementation

```
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]);
}


template<class ItemType>
void BubbleSort(ItemType values[], int numValues)
// Post: The elements in the array values are sorted by key.
{
  int current = 0;

  while (current < numValues - 1)
  {
    BubbleUp(values, current, numValues-1);
    current++;
  }
}
```

Bubble Sort Analysis

- Comparison Analysis
  - First position requires N-1 comparisons
  - Second position requires N-2 comparisons
  - Third position requires N-3 comparisons

  …
  - Position N-1 requires 1 comparison

$$(N-1) + (N-2) + (N-3) + \ldots + 1 = N(N-1)/2 = O(N^2)$$

- Swap Analysis
  - Number of Swaps in Worst Case = $N(N-1)/2 \approx N^2/2$
  - Average Number of Swaps $\approx 0.5*(N^2/2) = N^2/4$

ShortBubble Modification

```
template<class ItemType>
void ShortBubble(ItemType values[], int numValues)
// Post: The elements in the array values are sorted by key.
//       The process stops as soon as values is sorted.
{
  int current = 0;
  bool sorted = false;
  while (current < numValues - 1 && !sorted)
  {
    BubbleUp2(values, current, numValues-1, sorted);
    current++;
  }
}


template<class ItemType>
void BubbleUp2(ItemType values[], int startIndex, int endIndex, bool& sorted)
// Post: Adjacent pairs that are out of order have been switched
//       between values[startIndex]..values[endIndex] beginning at
//       values[endIndex].
//       sorted is false if a swap was made; otherwise, true.
{
  sorted = true;
  for (int index = endIndex; index > startIndex; index--)
    if (values[index] < values[index-1])
    {
      Swap(values[index], values[index-1]);
      sorted = false;
    }
}
```

ShortBubble Efficiency

- Also an $O(N^2)$ algorithm
- Only algorithm that recognizes an already sorted list and terminates

- See text for more details

Insertion Sort Example

Number Of Elements N = 4

Unsorted

Sorted

	values
[0]	18
[1]	9
[2]	5
[3]	10

i=0

	values
[0]	9
[1]	18
[2]	5
[3]	10

i=1

	values
[0]	5
[1]	9
[2]	18
[3]	10

i=2

	values
[0]	5
[1]	9
[2]	10
[3]	18

i=3

Insertion Sort Algorithm

Concept:

Create in place a sorted list by taking the values and inserting them in order into the list

Suppose the unsorted elements are in variables A_1 through A_N

For each pass i for 1 <= i <= N-1,

- Take the current element A_i and insert it into it proper position among the elements A_1 through A_{i-1}
- Now A_1 through A_i contain the i smallest elements in sorted order
- Repeat for each i listed above

Insertion Sort Implementation

```
template<class ItemType>
void InsertItem(ItemType values[], int startIndex, int endIndex)
// Post: values[0]..values[endIndex] are now sorted.
{
  bool finished = false;
  int current = endIndex;
  bool moreToSearch = (current != startIndex);

  while (moreToSearch && !finished)
  {
    if (values[current] < values[current-1])
    {
      Swap(values[current], values[current-1]);
      current--;
      moreToSearch = (current != startIndex);
    }
    else
      finished = true;
  }
}


template<class ItemType>
void InsertionSort(ItemType values[], int numValues)
// Post: The elements in the array values are sorted by key.
{
  for (int count = 0; count < numValues; count++)
    InsertItem(values, 0, count);
}
```

UAHuntsville

Insertion Sort Analysis

- Comparison Analysis
  - Similar to previous examples
    $O(N^2)$

- Swap Analysis
  - Average Number of Swaps ≈ $N^2/4$

Coping with High Swap Cost

- If elements are large records, an algorithm which requires $O(N^2)$ swaps will have a significant impact on overall execution time
- Instead of swapping the records, maintain an array of pointers to the records, and reorder the array of pointers using the sorting algorithm
- Once the pointers have been rearranged, the records themselves can be reordered at $O(N)$ cost

$O(N^2)$ Sorting Algorithms

- Selection Sort
- Bubble Sort
  - ShortBubble
- Insertion Sort

Sorting Algorithms

Utilizing Divide and Conquer

More Sorting Algorithms

- General Tradeoff
  - Algorithm complexity versus Performance

- Quicksort
- Mergesort

Quicksort

- ***Quicksort*** Algorithm

if there is more than one item in ***values[first]…values[last]***

Select ***splitVal***

Split the array so that

values[first]…values[splitPoint-1] <= ***splitVal***

values[splitPoint] = ***splitVal***

values[splitPoint+1]…values[last] > ***splitVal***

QuickSort the left half

QuickSort the right half

Quicksort Implementation - 1

```
typedef int ItemType;



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

Quicksort Example

Initial Order

9	20	6	10	14	8	60	11

Split Element
(first element)

9	20	6	10	14	8	60	11

9	8	6	10	14	20	60	11

<= Split

> Split

Swap Split
With Element
At Split Point

6	8	9	10	14	20	60	11

6	8	9

10	14	20	60	11

UAHuntsville

Quicksort Implementation - 2

```
void Split(ItemType values[], int first, int last, int& splitPoint)
{
  ItemType splitVal = values[first];
  int saveFirst = first;
  bool onCorrectSide;

  first++;
  do
  {
    onCorrectSide = true;
    while (onCorrectSide)              // Move first toward last.
      if (values[first] > splitVal)
        onCorrectSide = false;
      else
      {
        first++;
        onCorrectSide = (first <= last);
      }

    onCorrectSide = (first <= last);
    while (onCorrectSide)             // Move last toward first.
      if (values[last] <= splitVal)
        onCorrectSide = false;
      else
      {
        last--;
        onCorrectSide = (first <= last);
      }

    if (first < last)
    {
      Swap(values[first], values[last]);
      first++;
      last--;
    }
  } while (first <= last);

  splitPoint = last;
  Swap(values[saveFirst], values[splitPoint]);
}
```

Split Function - 1

Initialization

9	20	6	10	14	8	60	11
Save First	first						last

Increment first
Until
value[first]>splitVal

9	20	6	10	14	8	60	11
Save First	first						last

Decrement last
Until
value[last]<=splitVal

9	20	6	10	14	8	60	11
Save First	first				last		

Split Function - 2

Swap value[first] & value[last]
Then move first and last
toward each other

9	8	6	10	14	20	60	11
Save First		first		last			

Increment first Until value[first]>splitVal
Decrement last Until value[last]<=splitVal
Or first > last

9	8	6	10	14	20	60	11
Save First		last	first				

first>last so no swap within loop
Swap value[savefirst] & value[last]

6	8	9	10	14	20	60	11
Save First		last Split point					

UAHuntsville

Comments on Quicksort

- In-place sorting algorithm
- Non-recursive case (base case)
  - Segment contains at most one element
- Each recursive case splits problem into smaller, not necessarily even, pieces
- Algorithm works even if splitting value is the smallest or largest element in the subarray
  - Efficiency suffers
- Worst case: array sorted or mostly sorted
  - $O(N^2)$
- Best case: each split produces equal size subarrays
  - $O(N\log_2 N)$

Mergesort

- $O(N \cdot \log_2 N)$
- ***Mergesort*** Algorithm

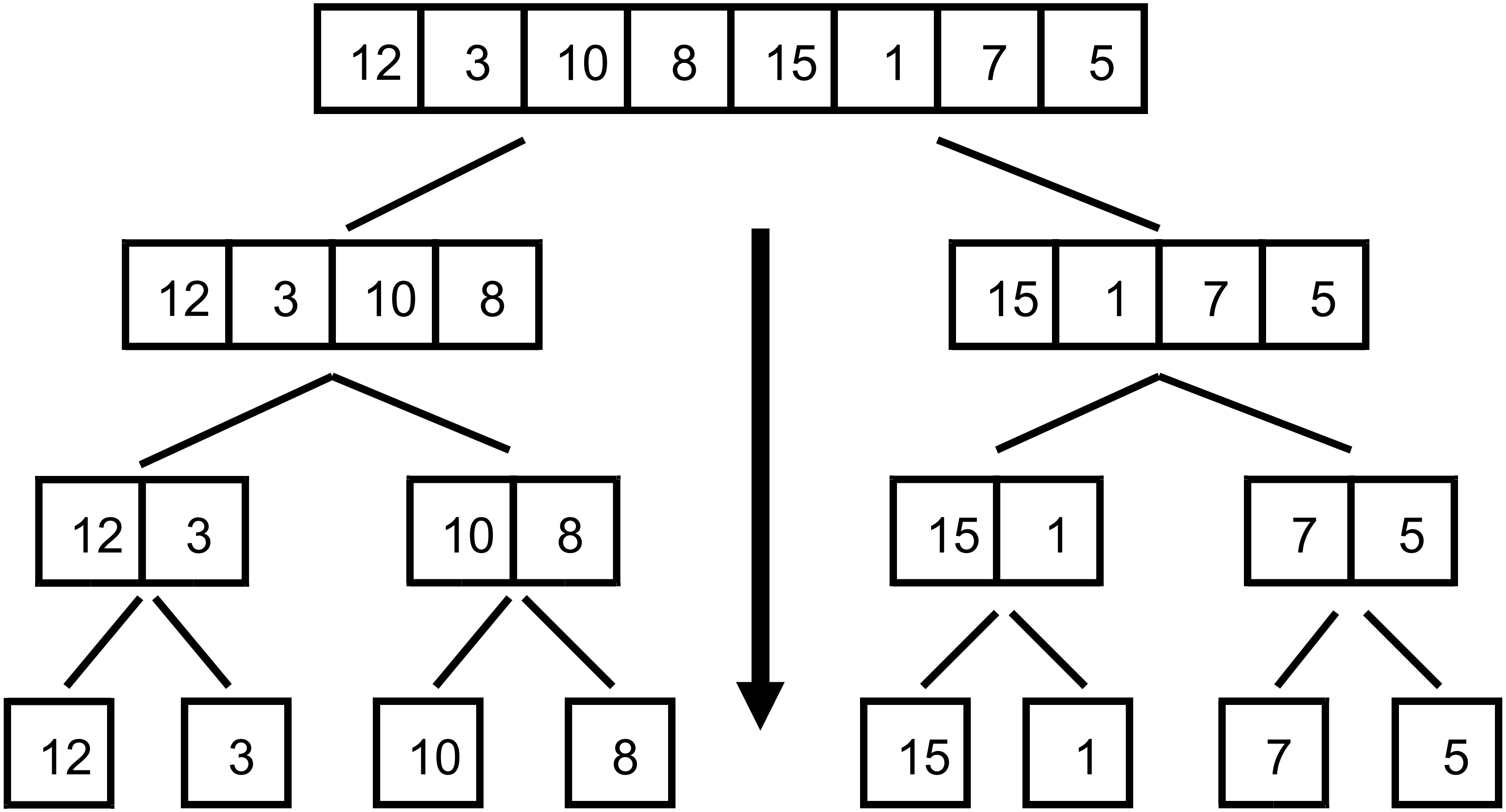
  Cut the array in half

  MergeSort the left half

  MergeSort the right half

  Merge the two sorted halves into one sorted array

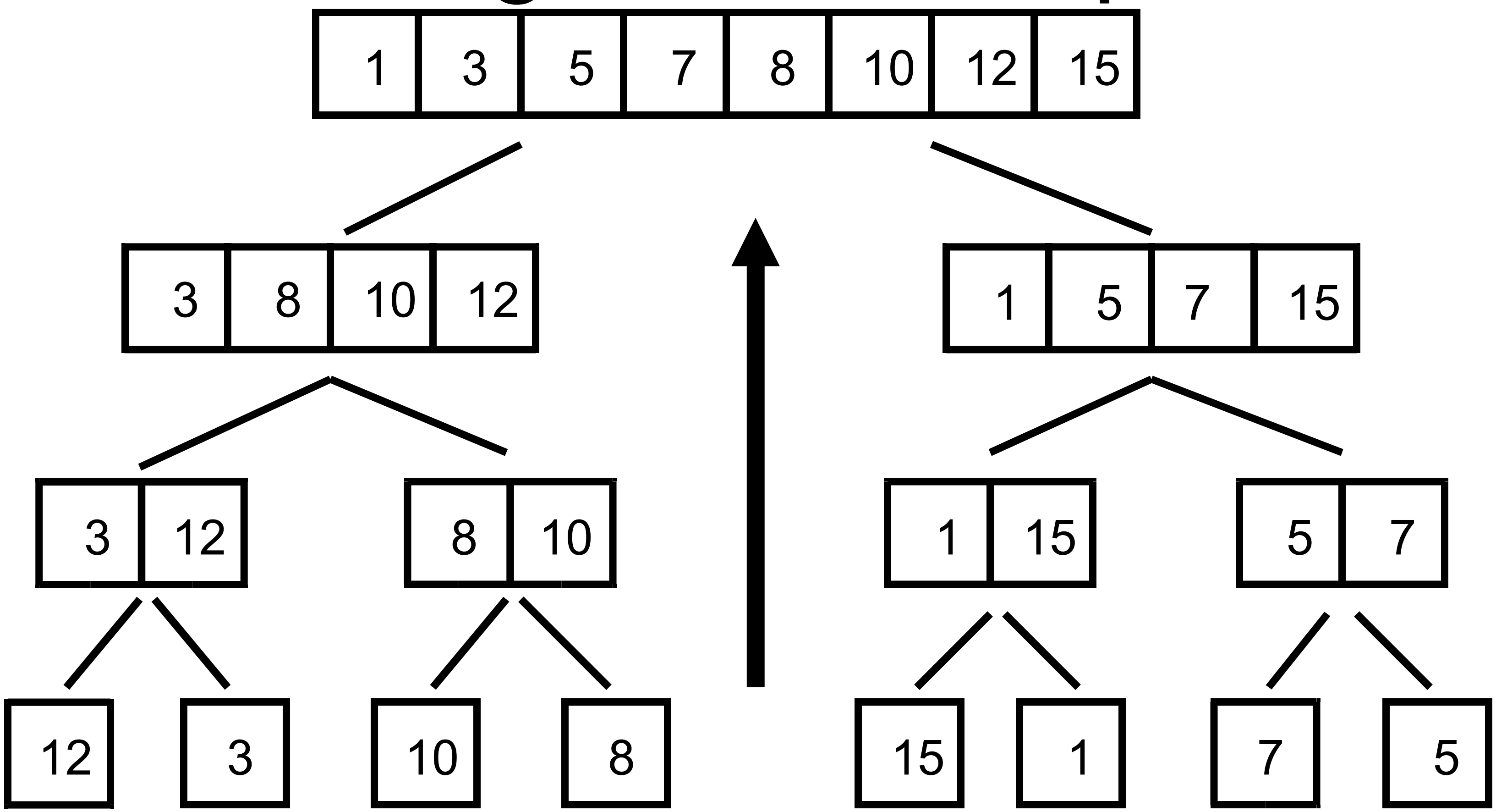
Mergesort Example



Recursive Splitting

UAHuntsville

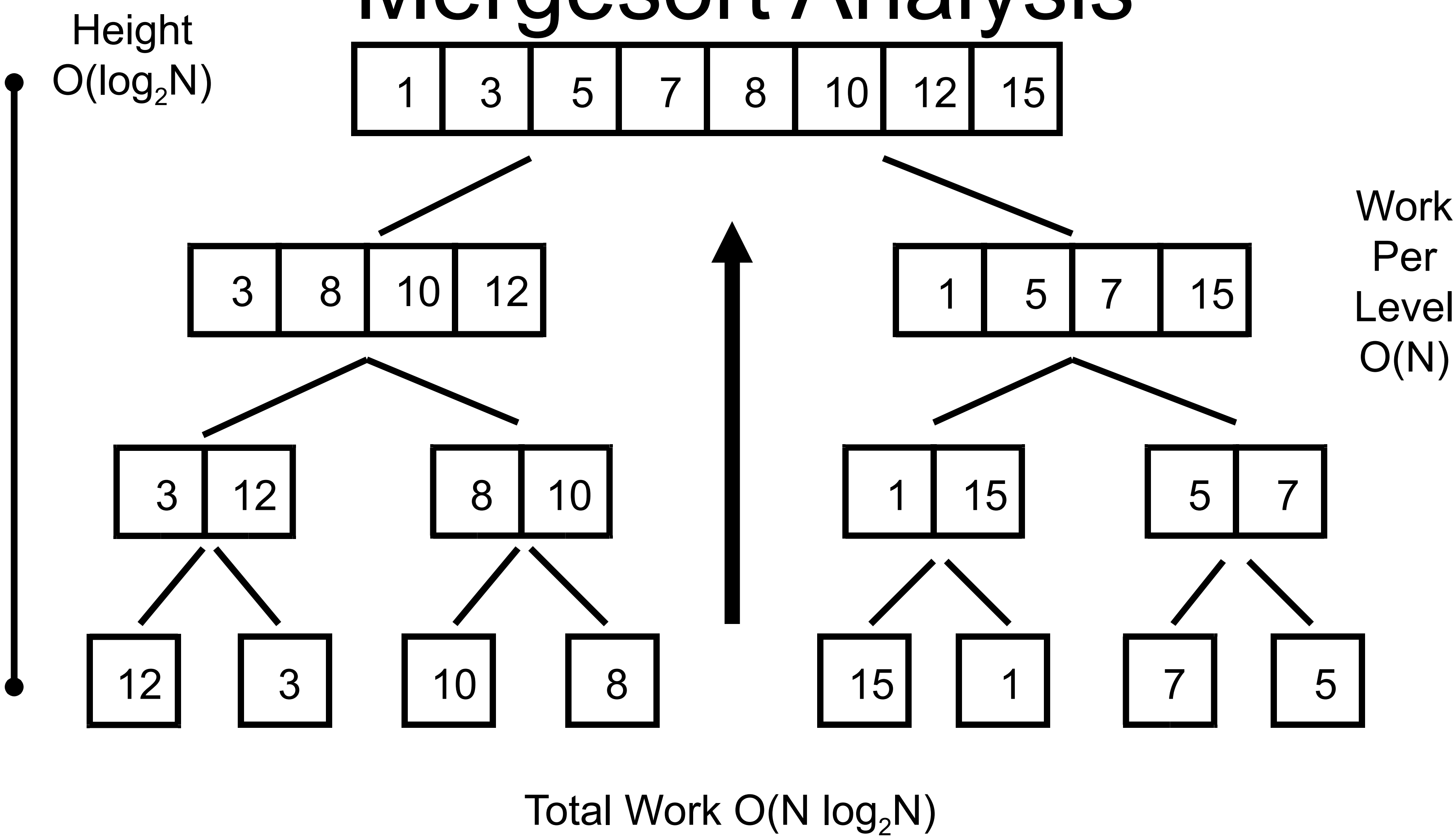
Mergesort Example



Merging Sorted SubArrays

UAHuntsville

Mergesort Analysis



Mergesort Implementation

```
template<class ItemType>
void MergeSort(ItemType values[], int first, int last)
// first = starting index
// last = ending index
// Post: The elements in values are sorted by key.
{
  if (first < last)
  {
    int middle = (first + last) / 2;             // Calculate split index
    MergeSort(values, first, middle);            // Sort left half of array
    MergeSort(values, middle + 1, last);         // Sort right half of array
    Merge(values, first, middle, middle + 1, last); // Combine the sorted halves
  }
}
```

```cpp
template<class ItemType>
void Merge (ItemType values[], int leftFirst, int leftLast, int rightFirst, int rightLast)
// Post: values[leftFirst]..values[leftLast] and values[rightFirst]..values[rightLast] have been merged.
//       values[leftFirst]..values[rightLast] is now sorted.
{
  ItemType tempArray[SIZE];
  int index = leftFirst;
  int saveFirst = leftFirst;

  while ((leftFirst <= leftLast) && (rightFirst <= rightLast))
  {
    if (values[leftFirst] < values[rightFirst])
    {
      tempArray[index] = values[leftFirst];
      leftFirst++;
    }
    else
    {
      tempArray[index] = values[rightFirst];
      rightFirst++;
    }
    index++;
  }

  while (leftFirst <= leftLast)
  // Copy remaining items from left half.

  {
    tempArray[index] = values[leftFirst];
    leftFirst++;
    index++;
  }

  while (rightFirst <= rightLast)
  // Copy remaining items from right half.
  {
    tempArray[index] = values[rightFirst];
    rightFirst++;
    index++;
  }

  for (index = saveFirst; index <= rightLast; index++)
    values[index] = tempArray[index];
}
```

Big-O Comparisons

N	$\log_2 N$	N^2	$N\log_2 N$
32	5	1024	160
64	6	4096	384
128	7	16,384	896
256	8	65,536	2,048
512	9	262,144	4,608
1024	10	1,048,576	10,240
2048	11	4,194,304	22,528
4096	12	16,777,216	49,152

Relative Algorithm Efficiency

Sort	Best Case	Average Case	Worst Case
SelectionSort	$O(N^2)$	$O(N^2)$	$O(N^2)$
BubbleSort	$O(N^2)$	$O(N^2)$	$O(N^2)$
ShortBubble	$O(N)$*	$O(N^2)$	$O(N^2)$
InsertionSort	$O(N)$*	$O(N^2)$	$O(N^2)$
MergeSort	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N\log_2 N)$
QuickSort	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N^2)$
HeapSort	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N\log_2 N)$

* Data Almost Sorted

Radix Sort

- Not a ***comparison*** sort
- Makes use of the values in the keys to order the items
- Radix
  - The number of possibilities for each position

    or
  - The number of digits in a number system

Radix Sort Example - 1

Unsorted Array

762
124
432
761
800
402
976
100
001
999

Radix Sort Procedure
- Divide by radix subgroup starting with least significant position first
- Recombine the values in order
- Repeat through most significant position

Radix Sort Example - 2

[0]	[1]	[2]	[3]	[4]
800	761	762		124
100	001	432		
		402		

[5]	[6]	[7]	[8]	[9]
	976			999

Array of Queues for Pass 1

Array After Pass 1

800
100
761
001
762
432
402
124
976
999

Radix Sort Example - 3

[0]	[1]	[2]	[3]	[4]
800		124	432	
100				
001				
402				

[5]	[6]	[7]	[8]	[9]
	761	976		999
	762			

Array of Queues for Pass 2

Array After Pass 2

800
100
001
402
124
432
761
762
976
999

UAHuntsville

Radix Sort Example - 4

[0]	[1]	[2]	[3]	[4]
001	100			402
	124			432

[5]	[6]	[7]	[8]	[9]
		761	800	976
		762		999

Array of Queues for Pass 3

Array After Pass 3

001
100
124
402
432
761
762
800
976
999

```
// This file contains the functions for Radix Sort.
// In the RadixSort function, the parameters have these meanings:
//
// values        is the array to be sorted
// numValues     is the size of the array to be sorted
// numPositions  is the size of the key measured in digits, characters etc..
//               If keys have 3 digits, this has the value 3,
//               If 10 digit keys, this has the value 10.
//               If word keys, then this is the number of characters in
//                           the longest word.
// radix         is the radix of the key, 10 in the case of decimal digit keys
//               26 for case-insensitive letters, 52 if case-sensitive letters.

#include "QueType.h"
typedef int ItemType;

template<class ItemType>
void RadixSort(ItemType values[], int numValues, int numPositions, int radix)
// Post: Elements in values are in order by key.
{
  QueType<ItemType> queues[radix];
  // With default constructor, each queue size is 500
  int whichQueue;

  for (int position = 1; position <= numPositions; position++)
  {
    for (int counter = 0; counter < length; counter++)
    {
      whichQueue = values[counter].SubKey(position);   // SubKey extracts digit at specified position
      queues[whichQueue].Enque(values[counter]);
    }
    CollectQueues(values, queues, radix);
  }
}
```

```
template<class ItemType>
void CollectQueues(ItemType values[], QueType<ItemType> queues[], int radix)

// Post: queues are concatenated with queue[0]'s on top and
//       queue[9]'s on the bottom and copied into values.
{
  int index = 0;
  ItemType item;

  for (int counter = 0; counter < radix; counter++)
  {
    while (!queues[counter].IsEmpty())
    {
      queues[counter].Deque(item);
      values[index] = item;
      index++;
    }
  }
}
```

SubKey

- Suppose itemKey = 8749

  Position 1: itemKey % 10 = 9

  Position 2: (itemKey / 10) % 10 = 4

  Position 3: (itemKey / 100) % 10 = 7

  Position 4: (itemKey / 1000) % 10 = 8

- values[counter].SubKey(position)

  Digit = (itemKey / $10^{position-1}$) % 10

Radix Sort Efficiency

- Does not compare items
- Efficiency depends upon implementation of supporting data structures
  – List of values
  – Radix queues

Heap Sort

- General approach
  - Take the root (maximum) element off the heap and put it in its place
  - Reheap remaining elements
  - Repeat until no more elements left in heap
- Sounds similar to selection sort
  - Heap order property makes locating next element faster $O(\log_2 N)$

Heap Sort

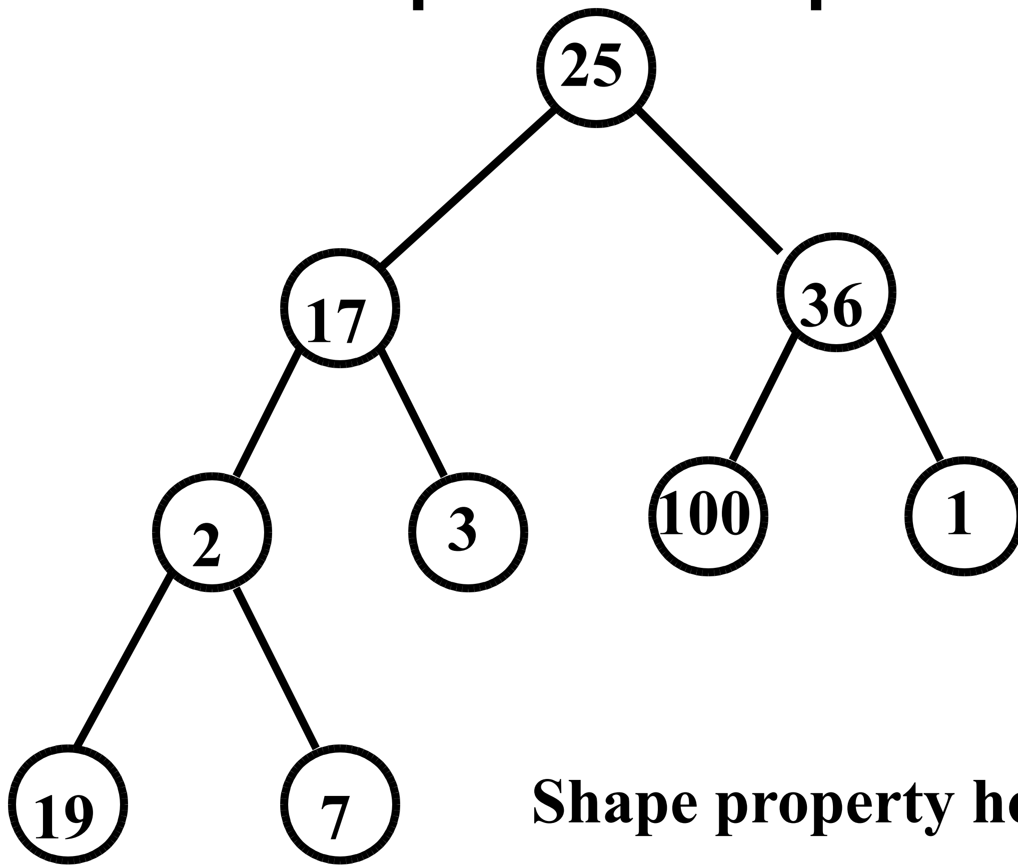
- Problem
  - Starting with an unsorted list
  - Must build a heap from this list of elements

Building a Heap

- Given an unsorted array of elements
- Construct a heap
  - Shape property
  - Order property

Build Heap Example

[0]	25
[1]	17
[2]	36
[3]	2
[4]	3
[5]	100
[6]	1
[7]	19
[8]	7



Shape property holds

Must reorder, but how?

Build Heap Example

- ReheapDown
  - Corrects order if each subtree is already ordered
- Apply ReheapDown to all subtrees on a level, then move up a level and ReheapDown again until root level reached

BuildHeap Algorithm

- FOR ***index*** going from first nonleaf node up to the root node

  ReheapDown(values, index, numValues-1)

Heap Sort Implementation

```
template<class ItemType>
void HeapSort(ItemType  values[], int numValues)
{
    int index;

    // Convert the array of values 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);
    }
}
```

Heap Sort Efficiency

- $O(N\log_2 N)$ comparisons
- Efficiency not affected by initial order of elements

Sorting Algorithm Efficiency

- $O(N\log_2 N)$ is theoretically optimal for key comparison sorting
- Other efficiency considerations
  - Actual size of N
  - Programmer time
  - Memory space
  - Eliminating function calls can improve proportionality constants
    - Swap(X,Y)
    - Temp = X;   X=Y;   Y=Temp;