Sorting II

Divide and Conquer Sorts

Announcement

- Final Exam
  - Wednesday, February 25, 2004
  - 8:00am – 10:00 am
  - 70-3435
- Please report all exam conflicts now!

Announcement

- Course Withdrawal
  - Last day to withdraw is this Friday, Jan 23rd

Announcement

- Project 2
  - To go live later this week
  - More Tomorrow

Sorting

- Any questions from yesterday?

Sorting

- Problem: Given an array of items, sort the elements in the array
  - Given:
    - array of objects to be sorted (x)
  - Calculation
    - Sorts the objects in the collection such that
      - x[i-1] <= x[i] for 0 < i < length of x

Evaluation

- Time Analysis
  - Best case
  - Worst case
  - Average Case

Comparable

- Let's generalize to sort for any object
  - By this time, you are probably painfully aware of the Comparable interface:
    - public int **compareTo**(Object o)
      - Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
      - Assumes that o is of same class of object being compared to.

Swap

- Swaps the item at one index of the array with an object of another

```
public void swap
   (Comparable x[], int i, int j)
   {
         Comparable tmp = x[i];
         x[i] = x[j];
         x[j] = tmp;
   }
```

Sort Algorithms

- Last Time
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - All pretty much all $\Theta(n^2)$ when it comes to comparisons.

Sort Algorithms

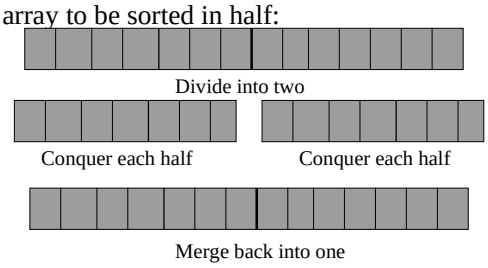
  - Divide and Conquer Sorts
    - Today: Merge Sort
    - Tomorrow: Quick Sort
- But first…
  - Zen sorting
    - See the sort, feel the sort, BE the sort.

Divide and Conquer

- Divide the elements to be sorted into two groups of equal size
- Sort each of the smaller groups
- Combine the smaller groups into 1 larger group
- Hmm: Smells like recursion to me!

Mergesort

- Like binary search, mergesort, divides the array to be sorted in half:



Mergesort

```
public void mergeSort
 (Comparable x[], int low, int high) {
    // Find the midpoint
    int mid = ( low + high ) / 2;

    // Do the sort
    mergeSort (x, low, mid);
    mergeSort (x, mid+1, high);

    // Do the merge
    merge (x, low, mid, high);
}
```

Mergesort

- But won't this recursion go on forever?

Mergesort

```
public void mergeSort
 (Comparable x[], int low, int high) {

    // test to end recursion
    if ((high – low) > 0) {
        // Find the midpoint
         int mid = ( low + high ) / 2;

        // Do the sort
       mergeSort (x, low, mid);
       mergeSort (x, mid+1, high);

       // Do the merge
       merge (x, low, mid, high);
    }
 }
```

Mergesort

```
public void mergeSort
 (Comparable x[], int low, int high) {

   // test to end recursion
   if ((high – low) > 0) {
       // Find the midpoint
        int mid = ( low + high ) / 2;

       // Do the sort
      mergeSort (x, low, mid);
      mergeSort (x, mid+1, high)l

      // Do the merge
      merge (x, low, mid, high);
   }
}
```

Test

Stop = do nothing

Continue

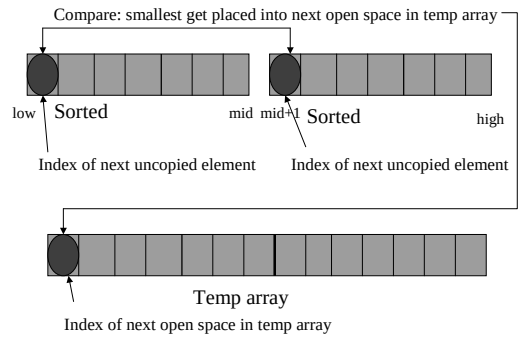
Mergesort

- Let's look at merge
  - By the time we get to merge, we have 2 sorted, smaller array portions
  - Use a temp array to assemble the elements of these smaller arrays back into a larger array.

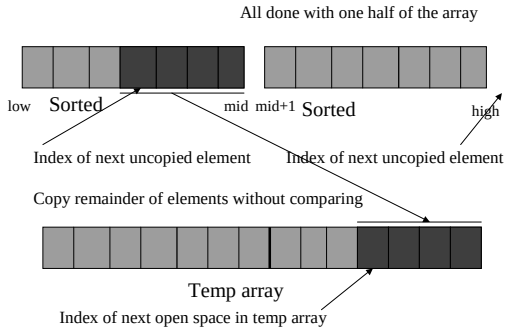
Mergesort

- Merge
  - Basic idea
    - Compare smallest uncopied element of 1st array half with smallest uncopied element of 2nd array half.
    - Place smallest into the temporary array
    - Iterate until all elements are copied into the temp array
    - Copy all elements from temp array back to original array

Mergesort



Mergesort



Mergesort

```
private void merge
 (Comparable x[], int low, int mid, int high) {

     // create temp array
     Comparable temp[] = new Comparable[high – low + 1];
     int uc1 = low;  // index of next uncopied in half 1
     int uc2 = mid+1;  // index of next uncopied in half 2
     int next = 0;  // next free slot in temp array;

     // Do merge
     while ((uc1 <= mid) && (uc2 <= high)) {
       if (x[uc1].compareTo (x[uc2]) < 0) {
          temp[next] = x[uc1]; uc1++;}
       else { temp[next] = x[uc2]; uc2++;}
       next++;
     }
```

Mergesort

```
     // at this point we're done with one of the array halves
     while (uc1 <= mid) {
        temp[next] = x[uc1];
        uc1++;  next++;
     }
     while (uc2 <= high) {
        temp[next] = x[uc2];
        uc2++;  next++;
     }

     // We're done with the merge, copy from temp back to x
     for (int i=0; i <= high-low; i++)
        x[low+i] = temp[i];
}
```

Mergesort

- Analysis
  - Real work is done in merge
    - Merge does no swaps
      - Does Θ (n) comparisons
      - Does Θ (n) copies
    - First recursive level, merge will merge n elements
      - T(n) = Θ (n)
    - Second recursive level, merge will merge n/2 element
      - However, merge will be activated twice
      - T(n) = Θ (n)
    - Third recursive level, merge will merge n/4 elements
      - However, merge will be activated four times
      - T(n) = Θ (n)

Mergesort

- Analysis
  - So on each recursive level
    - $T(n) = \Theta(n)$
  - Since arrays are being split in half, there will be $\log_2 n$ recursive levels
  - Runtime of complete algorithm
    - Best case, worst case, avg case:
      - $\Theta(n \log n)$

Mergesort

- Let's compare with sorts from last class:

n	n log n	n^2
2	2	4
8	24	64
32	160	1024
128	896	16,384
512	4608	262,144
1024	22,528	4,194,304

Mergesort

- Down side of mergesort
  - That temp array
    - Takes extra space
    - Takes extra time to allocate
    - Can speed up by having all calls to merge use the same temp array
  - Questions?
  - Merge Sort Applet

Quicksort

- Like mergesort:
  - Will divide the array into two portions
  - Will sort each portion recursively
  - Will merge sorted portions together once each is sorted.
- Unlike mergesort
  - Will not necessarily divide the array in half.

Quicksort

- In fact:
  - In mergesort
    - Array subdivision was trivial
    - Merge was sophisticated
  - In quicksort
    - Array subdivision is sophisticated
    - Merge is trivial.

Quicksort

- Basic idea:
  - Find a value that belongs near the middle of the array
    - Call this value the pivot
  - Place all values less than the pivot before the pivot location in the array
  - Place all values greater than the pivot after the pivot location in the array
  - Apply this same algorithm to the potion of the array before the pivot and the portion of the array after the pivot.

Quicksort

Choose pivot

Move all elements less than the pivot to lie before the pivot in the array

Move all elements less than the pivot to lie after the pivot in the array

Apply same algorithm to 1st portion of array

Apply same algorithm to 2nd portion of array

Quicksort

```
public void quickSort
 (Comparable x[], int low, int high) {
    // Find the pivot
    int pivotIdx = partition(x, low, high);

    // recursively call on array portions
    quickSort (x, low, pivotIdx -1);
    quickSort (x, pivotIdx+1, high);

}
```

Quicksort

- But won't this recursion go on forever?

Quicksort

```
public void quickSort
 (Comparable x[], int low, int high) {
   if ((high-low) > 0) {
      // Find the pivot
      int pivotIdx = partition(x, low, high);

      // recursively call on array portions
      quickSort (x, low, pivotIdx -1);
      quickSort (x, pivotIdx+1, high);
   }
}
```

Test

Stop = do nothing

Continue

Quicksort

- Let's take a look at partition
  - partition will have to do 3 things:
    1. Choose a pivot value
    2. Place the pivot value in the correct spot in the array
    3. Arrange the other elements of the array so that
       1. all less than the pivot lies before the pivot in the array
       2. All greater than the pivot lies after the pivot in the array.

Quicksort

- partition
  - Basic idea:
    - Arbitrarily choose first element as your pivot
    - Rearranging elements
      - Starting from 2nd elem, going forward, find first element > pivot
      - Starting from last elem, going backwards, find first element < pivot
      - Swap elements
      - Continue as long as the two paths don't cross
    - Finally, move pivot element to where the paths cross.

Quicksort

Choose first to be the pivot

Staring from 2nd, iterate forward until you find an element > pivot

Staring from last, iterate backwards until you find an element < pivot

Swap them

Quicksort

Eventually, the paths will cross

This element is < pivot

This element is > pivot

Swap pivot

All elements before pivot will be < pivot

All elements after pivot will be > pivot

Return the index where the pivot ended up

Quicksort

```
public int partition(Comparable x[], int low, int high)
{
    Comparable pivot = x[low];
    int tooBigIdx = low + 1;
    int tooSmallIdx = high;

    while (tooBigIdx < tooSmallIdx) {
      while ((tooBigIdx <= high) &&
             (x[tooBigIdx].compareTo (pivot) <= 0))
             tooBigIdx++;

      while ((x[tooSmallIdx].compareTo(pivot) > 0))
             tooSmallIdx--;
      if (tooBigIdx < tooSmallIdx)
         swap(x, tooBigIdx, tooSmallIdx);
    }
```

Quicksort

```
    // At this point, the elements of the array
    // have been arranged correctly
    // need to move the pivot to it's rightful place
    swap (x, low, tooSmallIdx);

    return tooSmallIdx;
  }
```

Quicksort

- Quicksort applet

Quicksort

- Analysis
  - The work carried out by partition is Θ (n)
  - In the best case, partition will place the pivot in the exact middle of the array in which case log n recursive calls will be made:
    - Doing Θ (n) work, log n times = Θ (n log n)

Quicksort

- In the worst case:
  - When array is already sorted

Already < than remainder of array elements

  - Recursive calls will call quick sort on array sizes of n, n-1, n-2, …, 1
    - $T(n) = n + (n-1) + (n-2) + \ldots + 1 = \Theta(n^2)$

Quicksort

- Can be shown:
  - Average case: Θ (n log n)
- Summary:
  - Best case: Θ (n log n)
  - Average case: Θ (n log n)
  - Worst case: Θ (n^2)
- However,
  - Quicksort does not require that temp array!
  - Quicksort can do "less work" per unit of computation
    - Copy vs. comparison

Quicksort

- We can avoid the worst case
  - If we become more carefully about choosing our pivot.
    - Choose 3 values from array
    - Make the median of the 3 be the pivot.

Summary

- Mergesort
  - Best, worst, average:Θ (n log n)
  - Needs temp array
- Quicksort
  - Best, average: Θ (n log n)
  - Worst: Θ (n^2)
  - Doesn't need temp array
  - Does "less work" per computation unit

Questions?