Topic 17
Fast Sorting

"The bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems."
- Don Knuth

Previous Sorts

- Insertion Sort and Selection Sort are both average case $O(N^2)$
- Today we will look at two faster sorting algorithms.
  - quicksort
  - mergesort

Stable Sorting

- A property of sorts
- If a sort guarantees the relative order of equal items stays the same then it is a stable sort
- $[7_1, 6, 7_2, 5, 1, 2, 7_3, -5]$  
  - subscripts added for clarity
- $[-5, 1, 2, 5, 6, 7_1, 7_2, 7_3]$  
  - result of stable sort
- Real world example:  
  - sort a table in Wikipedia by one criteria, then another  
  - sort by country, then by major wins

Quicksort

- Invented by C.A.R. (Tony) Hoare
- A divide and conquer approach that uses recursion
  1. If the list has 0 or 1 elements it is sorted
  2. otherwise, pick any element $p$ in the list. This is called the pivot value
  3. **Partition** the list minus the pivot into two sub lists according to values less than or greater than the pivot. (equal values go to either)
  4. return the quicksort of the first list followed by the quicksort of the second list
Quicksort in Action

39 23 17 90 33 72 46 79 11 52 64 5 71
Pick middle element as pivot: 46
Partition list
23 17 5 33 39 11 46 79 72 52 64 90 71
quick sort the less than list
Pick middle element as pivot: 33
23 17 5 11 33 39
quick sort the less than list, pivot now 5
[] 5 23 17 11
quick sort the less than list, base case
quick sort the greater than list
Pick middle element as pivot: 17
and so on....

private static void swapReferences(Object[] a, int index1, int index2) {
    Object tmp = a[index1];
    a[index1] = a[index2];
    a[index2] = tmp;
}

private void quicksort(Comparator[] list, int start, int stop) {
    if (start >= stop)
        return; //base case list of 0 or 1 elements
    int pivotIndex = (start + stop) / 2;
    // Place pivot at start position
    swapReferences(list, pivotIndex, start);
    Comparable pivot = list[pivotIndex];
    // Begin partitioning
    int i = start;
    // from first to i are elements less than or equal to pivot
    // from j to 1 are elements greater than pivot
    // elements beyond i have not been checked yet
    for (int j = start + 1; i < j; i++)
        if (list[i].compareTo(pivot) <= 0)
            swapReferences(list, i, j);
    // so move it to the less than or equal portion
    swapReferences(list, 1, j);
    //restore pivot to correct spot
    swapReferences(list, start, i);
    quicksort(list, start, j - 1); // Sort small elements
    quicksort(list, j + 1, stop); // Sort large elements

Quicksort on Another Data Set

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
44 68 191 119 119 37 83 82 191 45 158 130 76 153 39 25

Big O of Quicksort?

Clicker Question 1

What are the best case and worst case Orders (Big O) for quicksort?

- Best
  - A. O(NlogN)
  - B. O(N^2)
  - C. O(N^2)
  - D. O(NlogN)
  - E. O(N)

- Worst
  - A. O(NlogN)
  - B. O(N^2)
  - C. O(N^2)
  - D. O(NlogN)
  - E. O(NlogN)
Clicker Question 2

Is quicksort always stable?
A. Yes
B. No

Merge Sort Algorithm

Don Knuth cites John von Neumann as the creator of this algorithm

1. If a list has 1 element or 0 elements it is sorted
2. If a list has more than 1 split into into 2 separate lists
3. Perform this algorithm on each of those smaller lists
4. Take the 2 sorted lists and merge them together

Merge Sort

When implementing one temporary array is used instead of multiple temporary arrays.

Why?

Merge Sort code

```java
/**
 * perform a merge sort on the data in c
 * @param c c != null, all elements of c
 * @param are the same data type
 */
public static void mergeSort(Comparable[] c) {
    Comparable[] temp = new Comparable[ c.length ];
    sort(c, temp, 0, c.length - 1);
}

private static void sort(Comparable[] list, Comparable[] temp,
                        int low, int high) {
    if( low < high ) {
        int center = (low + high) / 2;
        sort(list, temp, low, center);
        sort(list, temp, center + 1, high);
        merge(list, temp, low, center + 1, high);
    }
}
```
Merge Sort Code

```java
private static void merge(Comparable[] list, int leftPos, int rightPos, int rightEnd) {
    int leftEnd = rightPos - 1;
    int tempPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    // Main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( list[ leftPos ].compareTo(list[ rightPos ]) <= 0 ) {
            temp[ tempPos ] = list[ leftPos ];
            leftPos++;
        } else {
            temp[ tempPos ] = list[ rightPos ];
            rightPos++;
        }
        tempPos++;
    // copy rest of left half
    while( leftPos <= leftEnd )
        temp[ tempPos ] = list[ leftPos ];
        leftPos++;
        tempPos++;
    // copy rest of right half
    while( rightPos <= rightEnd )
        temp[ tempPos ] = list[ rightPos ];
        tempPos++;
        rightPos++;
    // copy temp back into list
    for( int i = 0; i < numElements; i++, rightEnd-- )
        list[ rightEnd ] = temp[ rightEnd ];
}
```

Clicker Question 3

- What are the best case and worst case Orders (Big O) for mergesort?
  - **Best**
    - A. O(NlogN)
    - B. O(N^2)
    - C. O(N^2)
    - D. O(NlogN)
    - E. O(N)
  - **Worst**
    - A. O(NlogN)
    - B. O(N^2)
    - C. O(N^2)
    - D. O(NlogN)
    - E. O(NlogN)

Clicker Question 4

- Is mergesort always stable?
  - A. Yes
  - B. No

Clicker Question 5

- You have 1,000,000 items that you will be searching. How many searches need to be performed before the data is changed to make it worthwhile to sort the data before searching?
  - A. 5
  - B. 40
  - C. 1,000
  - D. 10,000
  - E. 500,000
### Comparison of Various Sorts (2001)

<table>
<thead>
<tr>
<th>Num Items</th>
<th>Selection</th>
<th>Insertion</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>59</td>
<td>49</td>
<td>6</td>
</tr>
<tr>
<td>4000</td>
<td>271</td>
<td>175</td>
<td>5</td>
</tr>
<tr>
<td>8000</td>
<td>1056</td>
<td>686</td>
<td>0</td>
</tr>
<tr>
<td>16000</td>
<td>4203</td>
<td>2754</td>
<td>11</td>
</tr>
<tr>
<td>32000</td>
<td>16852</td>
<td>11039</td>
<td>45</td>
</tr>
<tr>
<td>64000</td>
<td>expected?</td>
<td>expected?</td>
<td>68</td>
</tr>
<tr>
<td>128000</td>
<td>expected?</td>
<td>expected?</td>
<td>158</td>
</tr>
<tr>
<td>256000</td>
<td>expected?</td>
<td>expected?</td>
<td>335</td>
</tr>
<tr>
<td>512000</td>
<td>expected?</td>
<td>expected?</td>
<td>722</td>
</tr>
<tr>
<td>1024000</td>
<td>expected?</td>
<td>expected?</td>
<td>1550</td>
</tr>
</tbody>
</table>

(times in milliseconds, 1000 milliseconds = 1 second)

### Comparison of Various Sorts (2011)

<table>
<thead>
<tr>
<th>Num Items</th>
<th>Selection</th>
<th>Insertion</th>
<th>Quicksort</th>
<th>Merge</th>
<th>Arrays.sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.002</td>
<td>0.001</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2000</td>
<td>0.002</td>
<td>0.001</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4000</td>
<td>0.006</td>
<td>0.004</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8000</td>
<td>0.022</td>
<td>0.018</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16000</td>
<td>0.086</td>
<td>0.070</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>32000</td>
<td>0.341</td>
<td>0.280</td>
<td>0.004</td>
<td>0.005</td>
<td>0.003</td>
</tr>
<tr>
<td>64000</td>
<td>1.352</td>
<td>1.123</td>
<td>0.008</td>
<td>0.010</td>
<td>0.007</td>
</tr>
<tr>
<td>128000</td>
<td>5.394</td>
<td>4.499</td>
<td>0.017</td>
<td>0.022</td>
<td>0.015</td>
</tr>
<tr>
<td>256000</td>
<td>21.560</td>
<td>18.060</td>
<td>0.035</td>
<td>0.047</td>
<td>0.031</td>
</tr>
<tr>
<td>512000</td>
<td>86.083</td>
<td>72.303</td>
<td>0.072</td>
<td>0.099</td>
<td>0.066</td>
</tr>
<tr>
<td>1024000</td>
<td>???</td>
<td>???</td>
<td>0.152</td>
<td>0.206</td>
<td>0.138</td>
</tr>
<tr>
<td>2048000</td>
<td>0.317</td>
<td>0.434</td>
<td>0.287</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096000</td>
<td>0.663</td>
<td>0.911</td>
<td>0.601</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8192000</td>
<td>1.375</td>
<td>1.885</td>
<td>1.246</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(times in seconds)

### Concluding Thoughts

- Language libraries often have sorting algorithms in them
  - Java Arrays and Collections classes
  - C++ Standard Template Library
  - Python sort and sorted functions
- Hybrid sorts
  - when size of unsorted list or portion of array is small use insertion sort, otherwise use O(N log N) sort like Quicksort or Mergesort
Concluding Thoughts

- Sorts still being created!
- Timsort (2002)
  - created for Python version 2.3
  - now used in Java version 7.0
  - takes advantage of real world data
  - real world data is usually partially sorted, not totally random
- Library Sort (2006)
  - Like insertion sort, but leaves gaps for later elements