COT 5993: Introduction to Algorithms

Giri Narasimhan
ECS 389; Phone: x3748
giri@cs.fiu.edu
www.cs.fiu.edu/~giri/teach/5993S05.html
Solving Recurrence Relations

Page 62, [CLR]

<table>
<thead>
<tr>
<th>Recurrence; Cond</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n)$; $a = b$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n)$; $a &lt; b$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a - \epsilon})$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a})$</td>
<td>$T(n) = \Theta(n^{\log_b a \log n})$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(f(n))$; $af(n/b) \leq cf(n)$</td>
<td>$T(n) = \Omega(n^{\log_b a \log n})$</td>
</tr>
</tbody>
</table>
Celebrity Problem

- A **Celebrity** is one that knows **nobody** and that **everybody** knows.

Celebrity Problem:

**INPUT:** \( n \) persons with a \( n \times n \) information matrix.

**OUTPUT:** Find the “celebrity”, if one exists.

**MODEL:** Only allowable questions are:
- *Does person \( i \) know person \( j \)?*

- Naive Algorithm: \( O(n^2) \) Questions.
- Using Divide-and-Conquer: \( O(n \log_2 n) \) Questions.
- Improved solution?
Celebrity Problem (Cont’d)

– Induction Hypothesis 2: We know how to find \( n-2 \) non-celebrities among a set of \( n-1 \) people, i.e., we know how to find at most one person among a set of \( n-1 \) people that could potentially be a celebrity.

– Resulting algorithm needs \([3(n-1)-1]\) questions.
Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort
- Shaker Sort
- Shell Sort
- Merge Sort
- Heap Sort
- Quick Sort
- Bucket & Radix Sort
- Counting Sort
### Selection Sort

<table>
<thead>
<tr>
<th>Array Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial State</strong></td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td><strong>After Iteration 1</strong></td>
<td>2</td>
<td>5</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td><strong>After Iteration 2</strong></td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td><strong>After Iteration 3</strong></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td><strong>After Iteration 4</strong></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td><strong>After Iteration 5</strong></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Selection Sort

algorithm selectionSort( array a, integer N)
// given array a[0..N-1]
{
    for( int p = 0; p < N; p++ )
    {
        Compute j, the index of the smallest item in a[p..N];
        Swap a[p] and a[j];
    }
}
Selection Sort

algorithm selectionSort( array a, integer N)
// given array a[0..N-1]
{
    for( int p = 0; p < N-1; p++ )
    {
        // Compute j, the index of the smallest item in a[p..N];
        j = p;
        for (int m = p+1; p < N; p++)
            if (a[m] < a[j]) then j = m;
        // Swap a[p] and a[j];
        temp = a[p];
        a[p] = a[j];
        a[j] = temp;
    }
}
**Figure 8.3**
Basic action of insertion sort (the shaded part is sorted)

<table>
<thead>
<tr>
<th>Array Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..1] is sorted</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..2] is sorted</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..3] is sorted</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..4] is sorted</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..5] is sorted</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
**Figure 8.4**
A closer look at the action of insertion sort (the dark shading indicates the sorted area; the light shading is where the new element was placed).

<table>
<thead>
<tr>
<th>Array Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>8</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After a[0..1] is sorted</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After a[0..2] is sorted</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>After a[0..3] is sorted</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>After a[0..4] is sorted</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..5] is sorted</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
algorithm insertionSort( array a, integer N)
// given array a[0..N-1]
{
    for( int p = 1; p < N; p++ )
    {
        // insert a[p] in its right location
        temp = a[p];
        int j = p;

        while (j > 0 && temp < a[j-1])
            a[j] = a[j-1];
        j = j-1;
        a[j] = temp;
    }
}
Figure 8.5
Shell sort after each pass if the increment sequence is \{1, 3, 5\}

<table>
<thead>
<tr>
<th>Original</th>
<th>81</th>
<th>94</th>
<th>11</th>
<th>96</th>
<th>12</th>
<th>35</th>
<th>17</th>
<th>95</th>
<th>28</th>
<th>58</th>
<th>41</th>
<th>75</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 5-sort</td>
<td>35</td>
<td>17</td>
<td>11</td>
<td>28</td>
<td>12</td>
<td>41</td>
<td>75</td>
<td>15</td>
<td>96</td>
<td>58</td>
<td>81</td>
<td>94</td>
<td>95</td>
</tr>
<tr>
<td>After 3-sort</td>
<td>28</td>
<td>12</td>
<td>11</td>
<td>35</td>
<td>15</td>
<td>41</td>
<td>58</td>
<td>17</td>
<td>94</td>
<td>75</td>
<td>81</td>
<td>96</td>
<td>95</td>
</tr>
<tr>
<td>After 1-sort</td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>28</td>
<td>35</td>
<td>41</td>
<td>58</td>
<td>75</td>
<td>81</td>
<td>94</td>
<td>95</td>
<td>96</td>
</tr>
</tbody>
</table>
ShellSort

```java
algorithm shellsort(array a, integer N)
{
    for( int gap = a.length / 2; gap > 0;
        gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) )
        for( int i = gap; i < a.length; i++ )
        {
            tmp = a[ i ];
            int j = i;

            for(j >= gap && tmp < a[ j - gap ] )
                a[ j ] = a[ j - gap ];
            j = j - gap
            a[ j ] = tmp;
        }
    }
}
algorithm mergeSort( array a, integer left, integer right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, left, center );
        mergeSort( a, center + 1, right );
        merge( a, left, center + 1, right );
    }
}
Merge in Merge Sort

algorithm merge( array a, integer leftPos, integer rightPos, integer rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[leftPos].compareTo( a[rightPos] ) < 0 )
            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++];

    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[rightEnd ] = tmpArray[rightEnd ];
}
Figure 8.10  Quicksort
**Figure A**  If 6 is used as pivot, the end result after partitioning is as shown in the Figure B.

```
2 1 4 5 0 3 9 8 7 6
```

**Figure B**  Result after Partitioning

```
2 1 4 5 0 3 6 8 7 9
```
Algorithm Invariants

- Selection Sort
  - iteration k: the k smallest items are in correct location.

- Insertion Sort
  - iteration k: the first k items are in sorted order.

- Bubble Sort
  - In each pass, every item that does not have a smaller item after it, is moved as far up in the list as possible.
  - Iteration k: k smallest items are in the correct location.

- Shaker Sort
  - In each odd (even) numbered pass, every item that does not have a smaller (larger) item after it, is moved as far up (down) in the list as possible.
  - Iteration k: the k/2 smallest and largest items are in the correct location.
Algorithm Invariants (Cont’d)

- **Merge (many lists)**
  - Iteration $k$: the $k$ smallest items from the lists are merged.

- **Heapify**
  - Iteration with $i = k$: Subtrees with roots at indices $k$ or larger satisfy the heap property.

- **HeapSort**
  - Iteration $k$: Largest $k$ items are in the right location.

- **Partition (two sublists)**
  - Iteration $k$ (with pointers at $i$ and $j$): items in locations $[1..I]$ (locations $[i+1..j]$) are at least as small (large) as the pivot.
Sorting Algorithms

- Number of Comparisons
- Number of Data Movements
- Additional Space Requirements
Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort
- Shaker Sort
- Merge Sort
- Heap Sort
- Quick Sort
- Bucket & Radix Sort
- Counting Sort
Animation Demos

http://www-cse.uta.edu/~holder/courses/cse2320/lectures/applets/sort1/heapsort.html

http://cg.scs.carleton.ca/~morin/misc/sortalg/
algorithm QuickSort(array a, integer p, integer r)
{    if (p < r) then
        q = Partition(a, p, r)
        QuickSort(a, p, q-1)
        QuickSort(a, q+1, r)
}

algorithm Partition(array A, integer p, integer r)
{    x = a[r]
    i = p-1
    for j = p to r-1 do
        if a[j] <= x) then
            i++
            exchange(a[i], a[j])
        exchange(a[i+1], a[r])
    return i+1
}