COT 6405: Analysis of Algorithms

Giri Narasimhan
ECS 389; Phone: x3748
giri@cs.fiu.edu
www.cs.fiu.edu/~giri/teach/6405Spring04.html
Evolution of Data Structures

- Complex problems require complex data structures.
- Simple data types → Lists.
- Applications of lists include: students roster, list of voters, grocery list, list of transactions, etc.
- Array implementation of list: random access.
- Need for list “operations” arose – “Static” vs. “dynamic” lists. “Storing” items in list vs. “Maintaining” items in list.
- Lot of research on “Sorting” and “Searching”.
- “Inserting” in a specified location in a list caused the following evolution: Array implementation → Linked list implementation.
- Other linear structures e.g., stacks, queues, etc.
Evolution of Data Structures

- Trees made hierarchical organization of data easy to handle. Applications of trees: administrative hierarchy in a business set up, storing an arithmetic expression, organization of the functions calls of a recursive program, etc.
- Search trees (e.g., BST) were designed to make search and retrieval efficient in trees. A BST may not allow fast search or retrieval, if it is very unbalanced, since the time complexities of the operations depended on the height of the tree.
- Graphs generalize trees; model more general networks.
- Abstract data types. Advantages include: Encapsulation of data and operations, hiding of unnecessary details, localization and debugging of errors, ease of use since interface is clearly specified, ease of program development, etc.
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>
Sorting Algorithms

• Selection Sort
• Insertion Sort
• Bubble Sort
• Shaker Sort
• Shell Sort
• Merge Sort
• Heap Sort
• Quick Sort

• Bucket & Radix Sort
• Counting Sort
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.
Figure 8.5
Shellsort 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

public static void shellsort( Comparable [ ] a )
{
    for( int gap = a.length / 2; gap > 0; 
        gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) ) 
    for( int i = gap; i < a.length; i++ ) 
    {
        Comparable tmp = a[ i ];
        int j = i;

        for( ; j >= gap && tmp.compareTo( a[ j - gap ] ) < 0; j -= gap ) 
            a[ j ] = a[ j - gap ];
        a[ j ] = tmp;
    }
}
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/
QuickSort(A, p, r)
  if (p < r) then
    q = Partition(A, p, r)
    QuickSort(A, p, q-1)
    QuickSort(A, q+1, r)

Partition(A, p, 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
For the HeapSort analysis, we need to compute:

\[
\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}
\]

We know from the formula for geometric series that

\[
\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}
\]

Differentiating both sides, we get

\[
\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1 - x)^2}
\]

Multiplying both sides by \( x \) we get

\[
\sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2}
\]

Now replace \( x = 1/2 \) to show that

\[
\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \frac{1}{2}
\]
Bucket Sort

- N values in the range [a..a+m-1]
- For e.g., sort a list of 50 scores in the range [0..9].
- **Algorithm**
  - Make m buckets [a..a+m-1]
  - As you read elements throw into appropriate bucket
  - Output contents of buckets [0..m] in that order
- **Time O(N+m)**
Stable Sort

• A sort is stable if equal elements appear in the same order in both the input and the output.
• Which sorts are stable? Homework!
### Radix Sort

<table>
<thead>
<tr>
<th>3 2 9</th>
<th>7 2 0</th>
<th>7 2 0</th>
<th>3 2 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 7</td>
<td>3 5 5</td>
<td>3 2 9</td>
<td>3 5 5</td>
</tr>
<tr>
<td>6 5 7</td>
<td>4 3 6</td>
<td>4 3 6</td>
<td>4 3 6</td>
</tr>
<tr>
<td>8 3 9</td>
<td>4 5 7</td>
<td>8 3 9</td>
<td>4 5 7</td>
</tr>
<tr>
<td>4 3 6</td>
<td>6 5 7</td>
<td>3 5 5</td>
<td>6 5 7</td>
</tr>
<tr>
<td>7 2 0</td>
<td>3 2 9</td>
<td>4 5 7</td>
<td>7 2 0</td>
</tr>
<tr>
<td>3 5 5</td>
<td>8 3 9</td>
<td>6 5 7</td>
<td>8 3 9</td>
</tr>
</tbody>
</table>

**Algorithm**

```
for i = 1 to d do
    sort array A on digit i using a stable sort algorithm
```

**Time Complexity:** $O((n+k)d)$
Counting Sort

Initial Array

Counts

Cumulative Counts