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]

Recurrence; Cond	Solution
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-c) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-c) + O(n)$	$T(n) = O(n^2)$
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a = b$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a < b$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(f(n))$ $af(n/b) \leq cf(n)$	$T(n) = \Omega(n^{\log_b a} \log n)$

Celebrity Problem

- A **Celebrity** is one that knows <u>nobody</u> and that <u>everybody</u> 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

Array Position		0	1	2	3	4	5
Initial State		8	5	9	2	6	3
After Iteration 1		2	5	9	8	6	3
After Iteration 2		2	3	9	8	6	5
After Iteration 3		2	3	5	8	6	9
After Iteration 4		2	3	5	6	8	9
After Iteration 5		2	3	5	6	8	9

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)

Array Position	0	1	2	3	4	5
Initial State	8	5	9	2	6	3
After `a[0..1]` is sorted	5	8	9	2	6	3
After `a[0..2]` is sorted	5	8	9	2	6	3
After `a[0..3]` is sorted	2	5	8	9	6	3
After `a[0..4]` is sorted	2	5	6	8	9	3
After `a[0..5]` is sorted	2	3	5	6	8	9

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).

Array Position	0	1	2	3	4	5
Initial State	8	**5**				
After a[0..1] is sorted	5	8	**9**			
After a[0..2] is sorted	5	8	9	**2**		
After a[0..3] is sorted	2	5	8	9	**6**	
After a[0..4] is sorted	2	5	6	8	9	**3**
After a[0..5] is sorted	2	3	5	6	8	9

Insertion Sort

```
 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

Shellsort after each pass if the increment sequence is {1, 3, 5}

ORIGINAL	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

ShellSort

```
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;
        }
  }
```

Merge Sort

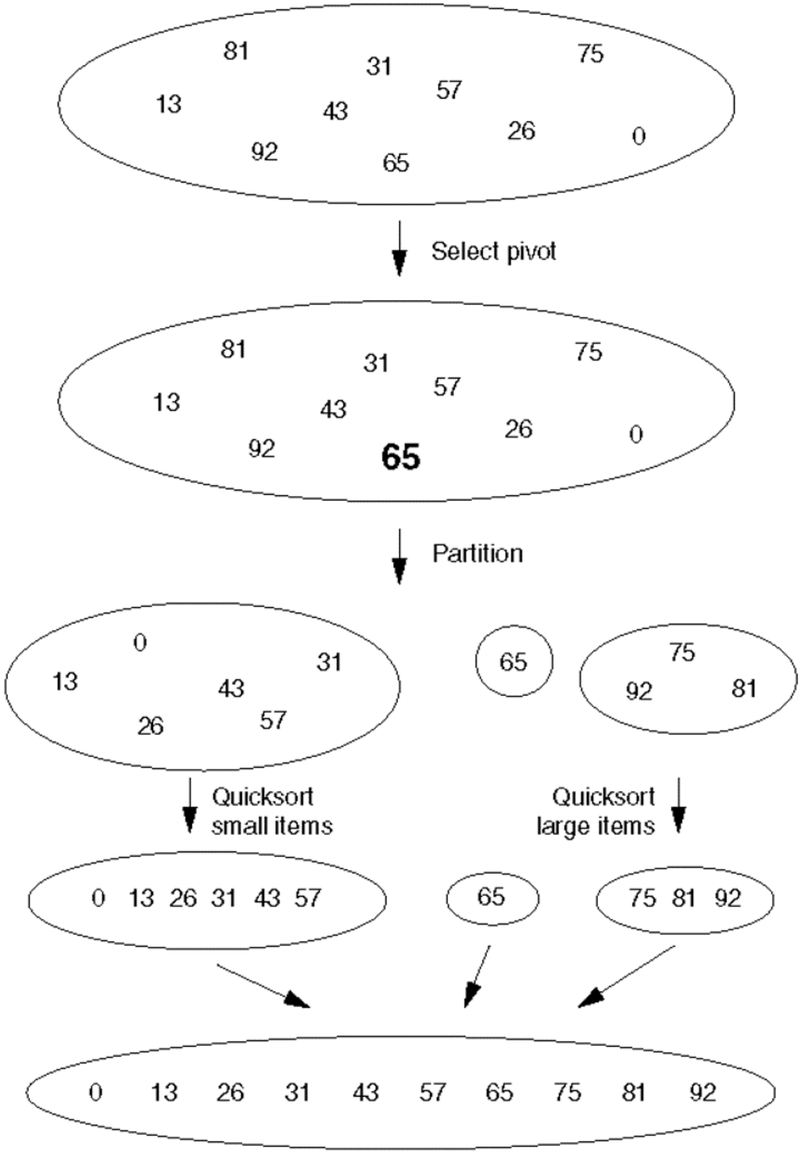
```
algorithm mergeSort( array a, integer left, integer right )
  {
     if( left < right )
     {
        int center = ( left + right ) / 2;
        mergeSort( a, left, center );
        mergeSort( a, center + 1, right );
        merge( a, left, center + 1, right );
     }
  }
```

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



Partition

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
}
```

Page 146, CLRS