Data Structures

(Feodor F. Dragan)
Department of Mathematics and Computer Science
Kent State University

Fall, 2000

CHAPTER 9

Sorting Algorithms

Contents

Sort Methods

- Insertion Sort
- Bubble Sort
- Selection Sort
- Count Sort
- Shaker Sort
- Shell Sort
- Heap Sort
- Merge Sort
- Quick Sort

Introduction

- The most intensive operation studied in computer science
  - Why is Sorting important?
- If the sort cannot be performed in memory ➔ external sorting
- Comparison-based sorting algorithms are based on *operator<*

operators	definition
operator > (a,b)	return b<a
operator >= (a,b)	return !(a<b)
operator <= (a,b)	return !(b<a)
operator != (a,b)	return b<a || a<b
operator == (a,b)	return !(b<a || a<b)

Deriving the relational and equality operators from *operator<*

Simple Insertion Sort

- Sort a set of keys by inserting each into an existing sorted list of keys

```
// insertionSort: sort items in array a

template <class Comparable>
void insertionSort( vector <Comparable> & a )
{
        for( int p = 1; p < a.size( ); p++ )
        {
                Comparable tmp = a[ p ];
                int j;
➔               for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
                        a[ j ] = a[ j - 1 ];
                a[ j ] = tmp;
        }
}
```

- If the initial file is sorted, only one comparison is made on each pass and it takes *O(N)* *(best case)*
- The line marked by ➔ can be executed *p+1* time for each *p;* $\sum_{p=1}^{N-1}(p+1) = O(N^2)$
- If the initial file is sorted in reverse order, it takes $O(N^2)$ *(worst case)*

- An improvement can be made by using a binary search to find a proper position
- An *inversion* is a pair of elements that are out of order
  - if $i < j \Rightarrow A_i > A_j$, where A is an array

Theorem 1. The average number of inversions in an array of *N* distinct members is *N(N-1)/4*

Proof: Consider an array *A* of *N* elements. Let A_r be the reverse of *A*. Consider a pair *(x,y)* in *A* with *y>x*. In exactly one of *A* and A_r this pair represents an inversion. The total number of these pairs in A and A_r together is *N(N-1)/2*. Thus an average array has half this amount, i.e., *N(N-1)/4* inversions.

Theorem 2. An algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average

Proof: On average we have *N(N-1)/4* inversions. Each swap removes only one insertion, so $\Omega(N^2)$ swaps are required.

Shell Sort

- Shell Sort is a significant improvement on insertion sort
- It sorts separate subfiles that contain every *k*th element of the original file
- The value of *k* is called an increment. For k=5, we have

subfile 1	x[0]	x[5]	x[10]	. . .
subfile 1	x[1]	x[6]	x[11]	. . .
subfile 1	x[2]	x[7]	x[12]	. . .
subfile 1	x[3]	x[8]	x[13]	. . .
subfile 1	x[4]	x[9]	x[14]	. . .

- The *i*th element of the *j*th subfile is *x[(i-1)*k+j-1]*
- After the first *k subfiles are sorted, a new smaller k is chosen, and the file is partitioned into a new set of subfiles*

Example: 25 57 48 37 12 92 86 33 with k=(5,3,1)

k=5	subfile 1	25 92	→	25 92
	subfile 2	57 86	→	57 86
	subfile 3	48 33	→	33 48
	subfile 4	37	→	37
	subfile 5	12	→	12

partial sort : 25 57 33 37 12 92 86 48

k=3	subfile 1	25 37 86	→	25 37 86
	subfile 2	57 12 48	→	12 48 57
	subfile 3	33 92	→	33 92

partial sort : 25 12 33 37 48 92 86 57

k=1

sorted file : 12 25 33 37 48 57 86 92

- Each step makes the file partially sorted
- Running time depends on the choice of increment sequence
  - "divide gap by 2" (shell's) ➔ in worst $O(N^2)$, in average $O(N^{3/2})$
  - odd gaps only ➔ in worst $O(N^{3/2})$, in average ?? $O(N^{5/4})$
  - divide by 2.2 ➔ very good in practice, in average ?? $O(N^{7/6})$

  (time complexity computation of shell sort is mathematically involved) in milliseconds

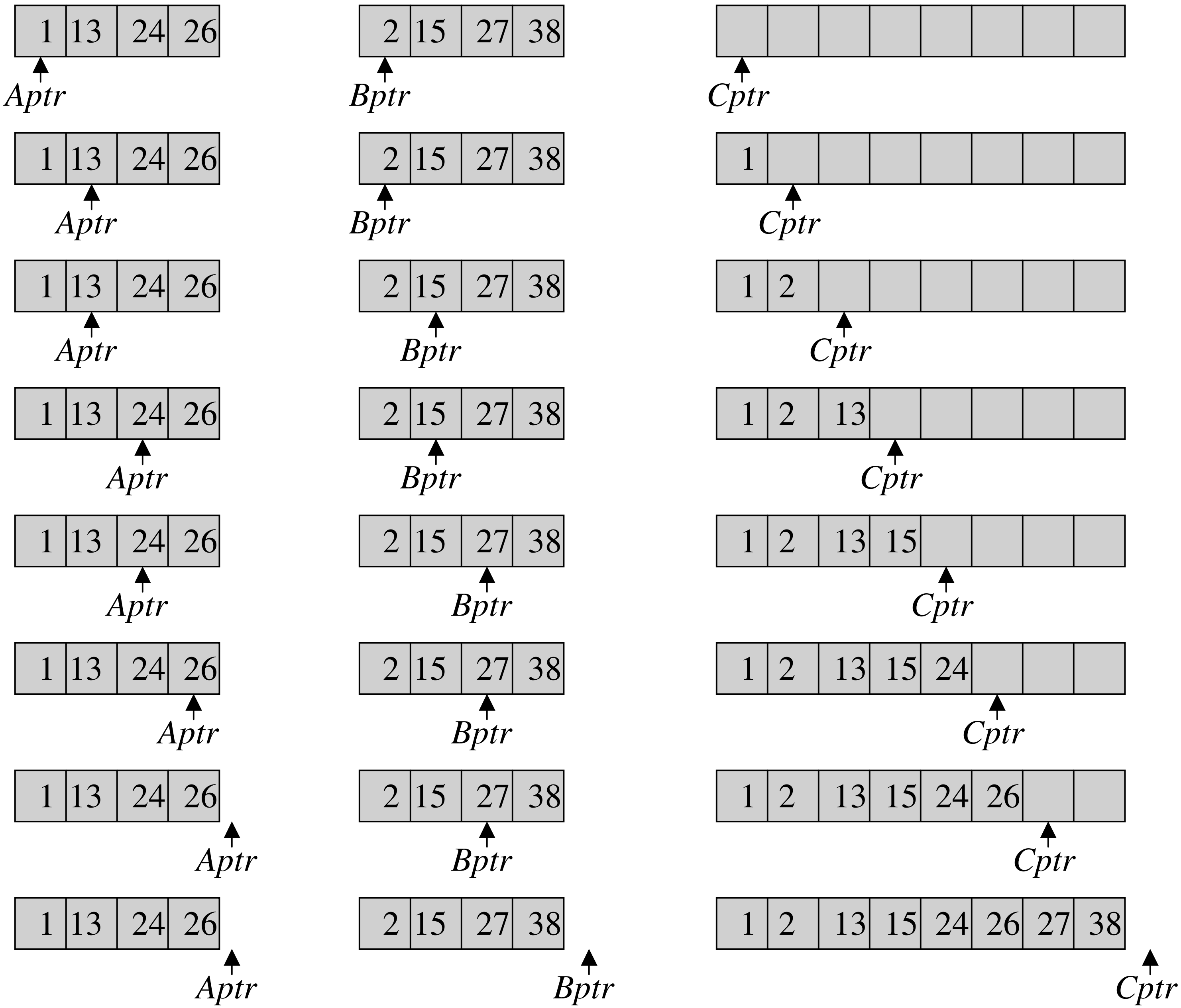
N	Insertion Sort	Shell Sort		
		Shell's	Odd gaps only	Dividing by 2.2
1, 000	122	11	11	9
4, 000	1,936	61	59	54
16, 000	32, 560	358	322	269
64, 000	520, 000	2, 091	1, 705	1, 249

```
// Shellsort, using good increments.

template <class Comparable>
void shellsort( vector <Comparable> & a )
{
        int j;
        for( int gap = a.size( ) / 2; gap > 0;
             gap = gap == 2 ? 1 : static_cast<int> ( gap / 2.2 ) )
        for( int i = gap; i < a.size( ); i++ )
        {
                Comparable tmp = a[ i ];
                for( j = i; j >= gap && tmp < a[ j - gap ]; j -= gap )
                        a[ j ] = a[ j - gap ];
                        a[ j ] = tmp;
        }
}
```

Merge Sort

- Merge Sort uses divide-and-conquer to obtain an *O(N logN)* running time.
- The Merge Sort algorithm involves three steps.
  1. If the number of items to sort is 0 or 1, return.
  2. Recursively sort the first and second halves separately.
  3. Merge the two sorted halves into a sorted group.
- Merging of sorted arrays can be done in linear time (step 3).
- Hence Merge Sort is *O(N logN)* time algorithm. Why?
- It is rarely used because it uses extra memory (*O(N)* additional space).
- The Merge routine is the cornerstone of most external sorting algorithms.

Linear-time merging of sorted arrays. An example.



The *merge* routine

```
// Internal method that merges two sorted halves of a subarray.
// a is an array of Comparable items. tmpArray is an array to place the merged result.
// leftPos is the left-most index of the subarray. rightPos is the index of the start of the second half.
// rightEnd is the right-most index of the subarray.

template <class Comparable>
void merge( vector<Comparable> & a, vector<Comparable> & tmpArray, int leftPos, int rightPos,
                int rightEnd )
{
   int leftEnd = rightPos - 1;
   int tmpPos = leftPos;
   int numElements = rightEnd - leftPos + 1;
// Main loop
   while( leftPos <= leftEnd && rightPos <= rightEnd )
      if( a[ leftPos ] <= a[ rightPos ] )
         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++ ];
// Copy tmpArray back
   for( int i = 0; i < numElements; i++, rightEnd-- )
      a[ rightEnd ] = tmpArray[ rightEnd ];
}
```

```
// Internal method that makes recursive calls.
// a is an array of Comparable items. tmpArray is an array to place the merged result.
// left is the left-most index of the subarray. right is the right-most index of the subarray.

template <class Comparable>
void mergeSort( vector<Comparable> & a, vector<Comparable> & tmpArray, int left, int right )
{
   if( left < right )
   {
      int center = ( left + right ) / 2;
      mergeSort( a, tmpArray, left, center );
      mergeSort( a, tmpArray, center + 1, right );
      merge( a, tmpArray, left, center + 1, right );
   }
}
```

Basic *mergeSort* routines

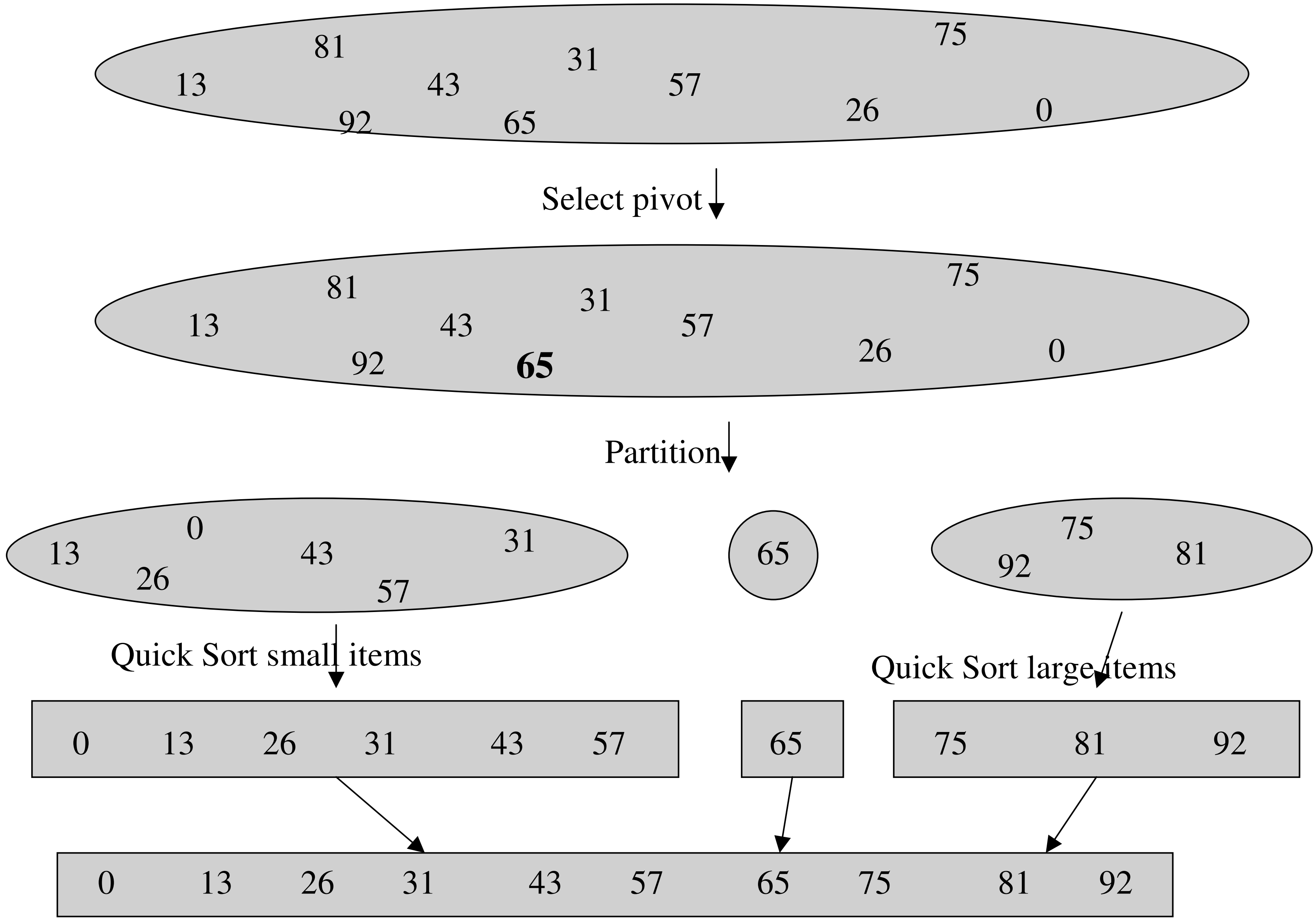
```
// Mergesort algorithm (driver).

template <class Comparable>
void mergeSort( vector<Comparable> & a )
{
   vector<Comparable> tmpArray( a.size( ) );
   mergeSort( a, tmpArray, 0, a.size( ) - 1 );
}
```

Quick Sort

- Quick Sort is a fast divide-and-conquer algorithm for sorting. It is a recursive algorithm.
- In practice it is the fastest comparison-based sorting algorithm.
- The algorithm *Quick_Sort(S)* consists of the following four steps:
    1. If the number of items to sort is 0 or 1, return.
    2. Pick *any* element *v* in *S*. This is called the *pivot*.
    3. *Partition S-{v}* (the remaining elements in *S*) into two disjoint groups: $L = \{x \in S - \{v\} \mid x \leq v\}$ and

       $R = \{x \in S - \{v\} \mid x \geq v\}$.
    4. Return the result of *Quick_Sort(L)* followed by *v* followed by *Quick_Sort(R)*.
- *Questions: How to pick a pivot, how to partition quickly, what to do with duplicates of v ?? (wrong decisions may give quadratic running time)*

The steps of Quick_Sort



Correctness

The correctness of the algorithm is guaranteed as follows.

1. The group of small elements is sorted, by virtue of recursion.
2. The largest element in the group of small elements is not larger than the pivot, by virtue of partition.
3. The pivot is not larger than the smallest element in group of large elements, by virtue of partition.
4. The group of large elements is sorted, by virtue of the recursion.

Quick_Sort is fast because the partitioning step can be performed very quickly and in place.

Analysis of Quick_Sort

• **Best Case**. It occurs when the partition always splits into equal subsets. The running time is $O(N \log N)$.

• **Worst Case**. It occurs when the partition repeatedly generates an empty subset. The running time is $O(N^2)$. Why?

$$\left.\begin{array}{l} T(N)=T(N-1)+N \\ T(0)=T(1)=1 \end{array}\right\} \Rightarrow T(N)=O(N^2)$$

• **Average Case**. The running time is $O(N \log N)$ (can be proved – see next slides).

Average Case Analysis of Quick_Sort

- $T(N)=T(L)+T(R)+N$ (linear cost N needed to perform the partition)

- $$T(L)=T(R)=\frac{T(0)+T(1)+T(2)+\cdots+T(N-1)}{N}$$
  (the average of all possible partitions)

- Then,

$$T(N)=2\left(\frac{T(0)+T(1)+T(2)+\cdots+T(N-1)}{N}\right)+N$$

$$NT(N)=2\big(T(0)+T(1)+T(2)+\cdots+T(N-1)\big)+N^2 \qquad (1)$$

for N-1: $$(N-1)T(N-1)=2\big(T(0)+T(1)+T(2)+\cdots+T(N-2)\big)+(N-1)^2 \qquad (2)$$

(1) - (2) gives: $$NT(N)-(N-1)T(N-1)=2T(N-1)+2N-1 \qquad (3)$$

$$NT(N)=(N+1)T(N-1)+2N \qquad (4)$$

drop

$$\frac{T(N)}{N+1}=\frac{T(N-1)}{N}+\frac{2}{N+1} \qquad (5)$$

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2}{N+1} \qquad \text{for N} \qquad (5)$$

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2}{N} \qquad \text{for N-1}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2}{N-1} \qquad \text{for N-2}$$

$$\cdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2}{3} \qquad \text{for 2}$$

(telescoping sum)

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2\left(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots + \frac{1}{N} + \frac{1}{N+1}\right)$$

$$= 2\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \cdots + \frac{1}{N} + \frac{1}{N+1}\right) - \frac{5}{2} =$$

$$= O(\log N) \qquad \text{(by Theorem 6.5)}$$

Hence, $T(N) = O(N \log N)$

Pivot, Partition, and Duplicates

• Several choices for the pivot have been found to improve the efficiency.

• v=median of first, last, and middle element (vs. first; last; middle; median of all; random element)

• A general partition strategy:

1. Swap the pivot with the element at the end.
2. Run *i* from left to right and *j* from right to left.
    - When *i* encounters a large element, *i* stops.
    - When *j* encounters a small element, *j* stops.
    - If *i* and *j* have not crossed, swap their items and continue.
    - Otherwise, stop this loop.
3. Swap the element in position *i* with pivot.

• Counters i and j must stop when they encounter an item equal to the pivot to guarantee good performance.

Efficient implementation

- Median-of-three partitioning optimization

  - We should not swap the pivot with the element in the last position. Instead, we should swap it with the element in the next to last position.
  - We can start *i* at *low+1* and *j* at *high-2.*
  - We are guaranteed that, whenever *i* searches for a large element, it will stop because in worst case it will encounter the pivot (and we stop on equality).
  - We are guaranteed that, whenever *j* searches for a small element, it will stop because in worst case it will encounter the first element (and we stop on equality).

- Sort 10 or fewer items by Insertion Sort. Place this test in the recursive Quick_Sort routine.

- Quick Sort is a classic example of using an analysis to guide program implementation.

```
// Internal quicksort method that makes recursive calls. Uses median-of-three partitioning and a cutoff
// a is an array of Comparable items. low is the left-most index of the subarray.
// high is the right-most index of the subarray.
template <class Comparable>
void quicksort( vector<Comparable> & a, int low, int high )
{  if( low + 10 > high ) insertionSort( a, low, high );
   else
   {  // Sort low, middle, high
      int middle = ( low + high ) / 2;
      if( a[ middle ] < a[ low ] )  swap( a[ low ], a[ middle ] );
      if( a[ high ] < a[ low ] )   swap( a[ low ], a[ high ] );
      if( a[ high ] < a[ middle ] ) swap( a[ middle ], a[ high ] );
      // Place pivot at position high - 1
      Comparable pivot = a[ middle ];
      swapItems( a[ middle ], a[ high - 1 ] );
      // Begin partitioning
      int i, j;
      for( i = low, j = high - 1; ; )
      {  while( a[ ++i ] < pivot ) { }
         while( pivot < a[ --j ] ) { }
         if( i < j ) swap( a[ i ], a[ j ] );
         else break;
      }
      swap( a[ i ], a[ high - 1 ] );  // Restore pivot
      quicksort( a, low, i - 1 );     // Sort small elements
      quicksort( a, i + 1, high );    // Sort large elements
   }
}
```

Quick Sort

```
// Internal insertion sort routine for subarrays that is used by quicksort.
// a is an array of Comparable items.
// low is the left-most index of the subarray.
// high is the right-most index of the subarray.
template <class Comparable>
void insertionSort( vector<Comparable> & a, int low, int high )
{
   for( int p = low + 1; p <= high; p++ )
   {
      Comparable tmp = a[ p ];
      int j;

      for( j = p; j > low && tmp < a[ j - 1 ]; j-- )
         a[ j ] = a[ j - 1 ];
      a[ j ] = tmp;
   }
}
```

Quick Sort Driver

```
// Quicksort algorithm (driver).
template <class Comparable>
void quicksort( vector<Comparable> & a )
{
   quicksort( a, 0, a.size( ) - 1 );
}
```

Quick Select

- Finding the *k*th smallest element in an array of *N* items.
- The median of *N* items is the *N/2*th smallest element.
- One approach is to sort all items and then take the *k*th smallest element.
- *Quick_Select* Algorithm:
    1. If the number of items in *S* is 1, then presumably *k* is also 1, return the single element.
    2. Pick a *pivot v* in *S*.
    3. *Partition S-{v}* into two disjoint groups (like in *Quick_Sort*):

$$L = \{x \in S - \{v\} \mid x \leq v\}$$
$$R = \{x \in S - \{v\} \mid x \geq v\}.$$

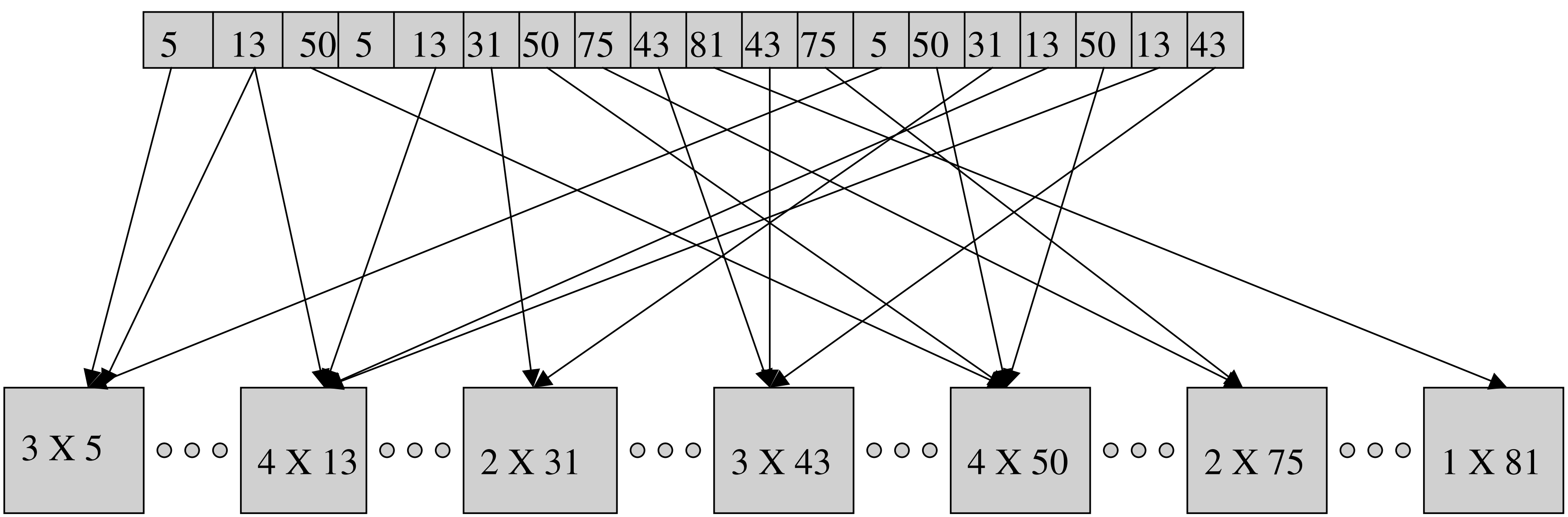
    4. If $k \leq |L|$, then the *k*th smallest is in *L*. Call *Quick_Select(L,k)* recursively.
    5. Else, if *k* is exactly equal to one more than the number of items in *L*, then the pivot is the *k*th smallest element.
    6. Else the *k*th smallest is in *R*, and it is the $(k-|L|-1)$th smallest element in *R*. Again we can make a recursive call *Quick_Select(R,k')* and return the result.
- *Quick_Select* makes only one recursive call (opposite to *Quick_Sort*).
- Time bounds: $O(N^2)$ in worst case; *O(N)* in average. Why?
  (there exists also a worst case linear time algorithm).

A Lower Bound for Sorting

- Any comparison based sorting algorithm must use ~ *N logN* comparisons on average and in the worst case.
- The proof is abstract and based on the following result.

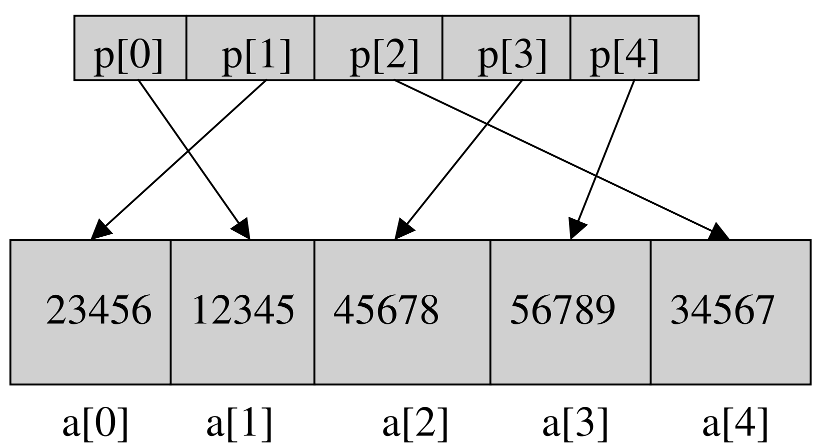
  > Any algorithm that sorts by using element comparisons only must use at least $\lceil \log(N!) \rceil$ comparisons for some input sequence.

- How large is $\lceil \log(N!) \rceil$? It is approximately $N \log N - 1.44N$.
- What is the Bucket Sort? When to use? Complexity.



Indirect Sorting

- Large objects are expensive to move.
- *Indirect Sorting*
  - Constructs an array of pointers and moves the pointers during most of the algorithm.
  - At the end of the algorithm, the array of large objects is then rearranged to match the pointers.
    - The simplest rearrangement strategy uses an extra array of large objects.
    - A more complex strategy performs the rearrangement in place and typically moves each object only once.



Homework # 4

1. Problems 9.5, 9.6 page 358 of the text
2. Problems 9.8, 9.9 page 359 of the text
3. Problem 9.14 page 360 of the text