Sorting Terms & Definitions
- Internal sorts hold all data in RAM
- External sorts use Files
- Ascending Order:
  - Low to High
- Descending Order:
  - High to Low
- Stable Sort
  - Maintains the relative order of equal elements, in situ.
  - Desirable if list is almost sorted or if items with equal values are to also be ordered on a secondary field.

Comparing Sorting Algorithms
- Program efficiency
  - Overall program efficiency may depend entirely upon sorting algorithm => clarity must be sacrificed for speed.
- Sorting Algorithm Analysis
  - Performed upon the “overriding” operation in the algorithm:
    • Comparisons
    • Swaps

Bubble Sort
- **Bubble** elements down (up) to their location in the sorted order.

```c
void swap( int &x, int &y ) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

Bubble Sort: Analysis
- if-statement:
  - 1 compare and in worst case, 1 swap
- inner for-loop:
  - body executed for j-values from n-1 down to i+1, or n-i-1 times
  - each execution of body involves 1 compare and up to 1 swap
- outer for-loop:
  - body executed for i-values from 0 up to n-2 (or 1 to n-1)
  - each execution of body involves n-i-1 compares and up to n-i-1 swaps
Bubble Sort: Analysis

So in the worst case, the number of swaps equals the number of compares, and is:
\[ \sum_{i=1}^{n-1} (n - i - 1) = n(n-1) - \frac{1}{2} (n-1)(n-2) - (n-1) \]

Which is clearly \( O(n^2) \).

Selection Sort

- In the \( i \)th pass, select the element with the lowest value among \( A[i], \ldots, A[n-1] \), & swap it with \( A[i] \).
- Results after \( i \) passes: the \( i \) lowest elements will occupy \( A[0], \ldots, A[i] \) in sorted order.

```plaintext
for (Begin = 0; Begin < Size - 1; Begin++) {
    SmallSoFar = Begin;
    for (Check = Begin + 1; Check < Size; Check++) {
        if (aList[Check] < aList[SmallSoFar])
            SmallSoFar = Check;
    }
    swap(aList[Begin], aList[SmallSoFar]);
}
```

Selection Sort: Graphical Trace

Selection Sort: Analysis

- if-statement: 1 compare
- inner for-loop:
  - body executed \( n-i-1 \) times (\( i \) is Begin and \( n \) is Size)
  - each execution of body involves 1 compare and no swaps
- outer for-loop:
  - body executed \( n-1 \) times
  - each execution of body involves \( n-i-1 \) compares and 1 swap

Selection Sort: Analysis

So in the worst case, the number of swaps is \( n - 1 \), and the number of compares is:
\[ \sum_{i=1}^{n-1} (n - i - 1) = n(n-1) - \frac{1}{2} (n-1)(n-2) - (n-1) \]

which is clearly \( O(n^2) \) and the same as for BubbleSort.

Duplex Selection Sort

- Min / Max Sorting
  - algorithm passes thru the array locating the min and max elements in the array \( A[i], \ldots, A[n-i+1] \). Swapping the min with \( A[i] \) and the max with \( A[n-i+1] \).
  - Results after the \( i \)th pass: the elements \( A[1], \ldots, A[i] \) and \( A[n-i+1], \ldots, A[n] \) are in sorted order.
  - What would be the Big Oh?
Duplex Selection Sort Analysis

- Without going through the figures,
  - Duplex Selection Sort is another O(N^2) sort algorithm.
  - But, the coefficient IS better than for BubbleSort!

Sorting Thoughts

- Comparison-Based Sorting
  - Algorithms which compare element values to each other
  - What is the minimum number of comparisons, required to sort N elements using a comparison-based sort?
- Is a Queue a type of sorting?

Comparison Tree

- Binary tree (hierarchical graph ≤ 2 branches per node) which contains comparisons between 2 elements at each non-leaf node & containing element orderings at its leaf (terminal) nodes.

Comparison Tree for 3 Elements

<table>
<thead>
<tr>
<th>Order</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b &lt; c</td>
<td>0</td>
</tr>
<tr>
<td>a &lt; c &lt; b</td>
<td>0</td>
</tr>
<tr>
<td>b &lt; a &lt; c</td>
<td>0</td>
</tr>
<tr>
<td>b &lt; c &lt; a</td>
<td>0</td>
</tr>
<tr>
<td>c &lt; a &lt; b</td>
<td>0</td>
</tr>
<tr>
<td>c &lt; b &lt; a</td>
<td>0</td>
</tr>
</tbody>
</table>

Comparison Tree Continued

- Any of the 3 elements (a, b, c) could be first in the final order. Thus there are 3 distinct ways the final sorted order could start.
- After choosing the first element, there are two possible selections for the next sorted element.
- After choosing the first two elements there is only 1 remaining selection for the last element.
- Therefore selecting the first element one of 3 ways, the second element one of 2 ways and the last element 1 way, there are 6 possible final sorted orderings = 3 * 2 * 1 = 3!

Order Tree for sorting N

- Any of the N elements (1 ... N) could be first in the final order. Thus there are N distinct ways the final sorted order could start.
- After choosing the first element, there are N-1 possible selections for the next sorted element.
- After choosing the first two elements there N-2 possible selections for the next sorted element, etc.
- Therefore selecting the first element one of N ways, the second element one of N-1 ways, the second element one of N-2 ways, etc., there are N * (N-1) * (N-2) * ... * 2 * 1 possible final sorted orderings which = N!
Comparisons for Sorting N

- The comparison tree for N elements must have $N!$ leaf nodes. Each leaf node contains one of the possible orderings of all of the N elements.
- Consider the previous comparison tree for 3 elements, all of the leaf nodes are at a depth of either 2 or $\lceil \log_2 3! \rceil$.
- The comparison tree for 4 elements must contain $4! = 24$ leaf nodes, all of which would be at a depth of either 4 or $\lceil \log_2 4! \rceil$.
- The “floor” $\lfloor \cdot \rfloor$ symbol means the largest whole number that is less than the number.

General Comparison Trees

- The comparison tree for N elements must contain $N!$ leaf nodes, all of which would be at a depth $> \lceil \log_2 N! \rceil$.
- The minimal number of comparisons required to sort a specific (unsorted) ordering is equal to the depth from the root to a leaf.

Depth vs. N

Minimal # of Comparisons

Quick Sort

- Select an item in the array as the pivot key.
- Divide the array into two partitions: a left partition containing elements < the pivot key and a right partition containing elements > the pivot key.

N log N

- Since the depth of all leaf nodes is $> \lceil \log_2 N! \rceil$ in a comparison tree, the minimal number of comparisons to sort a specific initial ordering of N elements is $> \lceil \log_2 N! \rceil$.
- Stirling’s Approximation for $\log_2 (N!)$ can be used to determine a lower bound for $\log_2 (N!)$ which is $O(N \log N)$.
- No comparison based sorting algorithm can sort faster than $O(N \log N)$. 

$N \log N$
Quicksort Trace

Start with i and j pointing to the first & last elements, respectively.
Select the pivot (3): [3 1 4 1 5 9 2 6 5 8]
Swap the end elements, then move L, R inwards.
[8 1 4 1 5 9 2 6 5 3]
Swap, and repeat: [2 1 4 1 5 9 8 6 5 3]
Swap, and repeat: [2 1 1 4 5 9 8 6 5 3]

Quick Sort Function

const int MISSING = -1;
void QuickSort(Item A[], int start, int end) {
    // Sort the array from start ... end
    int pivotIndex;
    int pivotKey;
    if (pivotIndex != MISSING) {
        pivotKey = A[pivotIndex];
        k = Partition(A, start, end, pivotKey);
        QuickSort(A, start, k-1);
        QuickSort(A, k, end);
    }
}

Pivoting

Partitioning test requires at least 1 key with a value < that of the pivot, and 1 value > to that of the pivot, to execute correctly.
Therefore, pick the greater of the first two distinct values (if any).
– OR Try and pick a pivot such that the list is split into equal size sublists, (a speedup that should cut the number of partition steps to about 2/3 that of picking the first element for the pivot).
  • Choose the middle (median) of the first 3 elements.
  • Pick k elements at random from the list, sort them & use the median.

Find Pivot

const int MISSING = -1;
const int FindPivot(const Item A[], int start, int end) {
    Item firstkey; //value of first key found
    int pivot; //pivot index
    int k; //run right looking for other key
    firstkey = A[start];
    pivot = MISSING;
    k = start + 1;
    //scan for different key
    while ( (k <= end) && (pivot == MISSING) )
        if (firstkey < A[k])      //select key
            pivot = k;
        else if (A[k] < firstkey)
            pivot = start;
        else
            k++;
    return pivot;
}

Average Case

• quicksort is based upon the intuition that swaps, (moves), should be performed over large distances to be most effective.
• quicksort’s average running time is faster than any currently known O(n log_2 n) internal sorting algorithms (by a constant factor).
• For very small n (e.g., n ≤ 16) a simple O(n^2) algorithm is actually faster than Quicksort.
  – Optimization: When the sublist is small, use another sorting algorithm, (selection).
Worst Case

- In the worst case, every partition might split the list of size \( j - i + 1 \) into a list with 1 element, and a list with \( j - i \) elements.
- A partition is split into sublists of size 1 & \( j-i \) when one of the first two items in the sublist is the largest item in the sublist which is chosen by findpivot.
- When will this worst case partitioning always occur?
- \( O(N^2) \)

 iterative Version is Posted

- Iterative implementation requires using a stack to store the partition bounds remaining to be sorted.

```
struct stackItem {
  int low, hi;
};
```

- At the end of any given partition, only one subpartition need be stacked.

Other Quicksort Optimizations

- All function calls should be replaced by inline code to avoid function overhead.
- Current partition bounds should be held in register variables.
- With large data records, swap pointers instead of copying records
  - We're accepting the cost of additional pointer dereferences to avoid the cost of some data copying.
- Carefully investigate the average data arrangement in order to select the optimal sorting algorithm.
  - For example, to identify special cases within Quicksort

Special Case: Mapping

- Better than \( N \log N \)
  - If sort key (member) consists of consecutive (unique) \( N \) integers they can be easily mapped onto the range 0 .. \( N-1 \) & sorted.
  - If the \( N \) elements are initially in array \( A \), then:

```
int A[], B[];
for (int i = 0; i < N; i++)
  B[A[i].GetKey() % N] = A[i];
```

Mapping

- \( O(n) \) time.
- Requires exactly 1 record with each key value!
- Of course, this is a very special circumstance…
- Special case of Bin Sorting. (If integers are not consecutive, but within a reasonable range, bit flags can be used to denote empty array slots.)

Bin Sorting

- Assume we need to sort an array of integers in the range 0-99:
- Assume we have an array of 10 linked lists (bins) for storage.
  - First make a pass through the list of integers, and place each into the bin that matches its 1’s digit.
  - Then, make a second pass, taking each bin in order, and place each integer into the bin that matches its 2’s digit, etc.
Bin Sorting

- Now if you just read the bins, in order, the elements will appear in ascending order.
  - Assuming no Bin with > 1 element
  - Otherwise, use another sort technique to sort bins
- Each pass takes $O(N)$ work, and the number of passes is just the number of digits in the largest integer in the original list.
- That beats QuickSort, but only in a somewhat special case. (When each bin has 1 element)
- Binsort Implementation will be posted online.