Internal Sorting

- Two general types of sorting:
  - Internal sorting, in which all elements are sorted in main memory
  - External sorting, in which elements must be sorted on disk or tape
- The focus here is on internal sorting techniques
- Sorting Classifications:
  - Exchange sorts that all run in $O(n^2)$
    - Insert sort, bubble sort, selection sort
  - Shell sort: an in-the-middle sort that runs in $O(n^{1.5})$ or $o(n^2)$
  - Efficient sorts that run in $O(n \log n)$
    - Heap sort, merge sort, quicksort
  - Special-purpose sorts that run in quicker time:
    - Bin sort, bucket sort, radix sort
- The problem of sorting, in general, is $\Omega(n \log n)$
  - Special cases are allowed to take less time because they are special cases, not general

The Sorting Problem

- Each record contains a field called the key
- Definition of the sorting problem:
  - Given a sequence of records $r_1, r_2, \ldots, r_n$ with key values $k_1, k_2, \ldots, k_n$
  - Arrange the records into any order $s$ such that
    - $r_{s_1}, r_{s_2}, \ldots, r_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq \ldots \leq k_{s_n}$
- Duplicate key values may be (are usually) allowed
  - Implicit ordering of duplicates:
    - After sorting, duplicate keys remain in the order in which they occurred in the input
      - This may be desirable, and the property is called stability
Comparing Performance of Sorting Algorithms

- Most obvious method: run two sorts on identical input and compare times
  - Problem: running time may depend on specifics of input values
  - Factors: number of records, key size, record size, range of key values, amount by which records are out of order

- Analytically, sorts are usually compared using two measures:
  - the number of comparisons
  - the number of swaps

- Common assumptions:
  - Each sort is passed an array containing the elements
  - \( n \) is the number of elements to be sorted
  - While \texttt{int} is the type in all examples, assume that any complex type implementing binary comparators (e.g. \texttt{<} or \texttt{\leq}) can be sorted

Insertion Sort

One of the simplest sorting algorithms to implement.

- Characteristics:
  - Makes \( n-1 \) passes
  - For a given pass numbered \( p \), elements in positions 0 through \( p \) are ensured to be sorted

- Code example:

```
// A hybrid of Shaffer's and other code

void insert_sort(int *array, int n) {
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0 && (array[j] < array[j - 1]); j--)
            swap(array[j], array[j - 1]);
    }
}
```
Insertion Sort

- Example sort

  \[ i = 1 \quad i = 2 \quad i = 3 \quad i = 4 \quad i = 5 \quad i = 6 \quad i = 7 \]
  
  42
  20
  17
  13
  28
  14
  23
  15

- Time complexity

  - Best case:
  - Average case
  - Worst case

Bubble Sort

Another very simple sort.

- Characteristics:
  - Also makes \( n - 1 \) passes, each pass represents a position
  - For a given pass numbered \( i \), “bubble-up” the element in the upper part of the array belonging in position \( i \)

- Code example:

  ```c
  void bubble_sort(int *array, int n) {
    for (int i = 0; i < n - 1; i++)
      for (int j = n - 1; j > i; j--)
        if (array[j] < array[j - 1])
          swap(array[j], array[j - 1]);
  }
  ```

CSC 375-Turner, Page 5

CSC 375-Turner, Page 6
Bubble Sort

- Example sort
  \[ i = 1 \quad i = 2 \quad i = 3 \quad i = 4 \quad i = 5 \quad i = 6 \quad i = 7 \]
  42
  20
  17
  13
  28
  14
  23
  15

- Time complexity
  - Best case:
  - Average case
  - Worst case

Selection Sort

Yet another very simple sort.

- Characteristics:
  - Also makes \( n - 1 \) passes, each pass represents a position
  - For a given pass numbered \( i \), find the element in the upper part of the array belonging in position \( i \). Only swap after that element is found.

- Code example:
  ```c
  void selection_sort(int *array, int n) {
    for (int i = 0; i < n - 1; i++) {
      int lowIndex = i;
      for (int j = n - 1; j > i; j--) {
        if (array[j] < array[lowIndex]) {
          lowIndex = j;
          swap(array[i], array[lowIndex]);
        }
      }
    }
  }
  ```

CSC 375-Turner, Page 7

CSC 375-Turner, Page 8
Selection Sort

- Example sort
  - $i = 1$  $i = 2$  $i = 3$  $i = 4$  $i = 5$  $i = 6$  $i = 7$
  - 42
  - 20
  - 17
  - 13
  - 28
  - 14
  - 23
  - 15

- Time complexity
  - Best case:
  - Average case
  - Worst case

Keeping Swap Costs Low

- Some sorts aren’t practical in an array
  - Desired: a means to swap objects without actually moving them
  - Pointer swapping can accomplish this

- Examples:
  - Array of integers: no pointers necessary
  - Array of student records: pointers necessary

Name = Smith, John
ID = 9922922299
(etc.)

Name = Jones, Bob
ID = 3232323399
(etc.)
Exchange Sorting

- An exchange is a swap of adjacent records.
  - Insert, bubble, and selection sort (basically) perform exchanges to move data
  - Therefore, they are sometimes called the exchange sorts.
- Exchange sort performance is measured based on inversions.
  - An inversion is any pair of array elements out of order with respect to each other
    - That is, consider an ordered pair \((i, j)\) for which \(i < j\) and \(array[i] > array[j]\)
  - Observation: the number of inversions is exactly the number of exchanges used by insert sort
    - If the number of inversions is \(n\), then insert sort is \(O(n)\)
    - The average number of inversions in an array of \(n\) distinct elements is \(n(n - 1)/4\) (a theorem)
    - Exchange sorts are therefore \(\Omega(n^2)\) (another theorem)

CSC 375-Turner, Page 11

Shellsort

The first algorithm to break the quadratic time barrier.

- Characteristics:
  - Consists of \(\log n\) phases
  - Works by comparing (and swapping) distant elements
  - Each phase reduces the distance between compared elements by half
  - Also called diminishing increment sort
- Code example:

```c
void shellsort(int *array, int n) {
    int j;
    for (int gap=n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int tmp = array[i];
            for (j = i; j >= gap && tmp < a[j - gap]; j -= gap) {
                array[j] = array[j - gap];
                array[j] = tmp;
            }
        }
    }
}
```

CSC 375-Turner, Page 12
**Shellsort**

- Shellsort has been shown to be $O(n^{1.5})$ or $o(n^2)$
- Example: sort the list 59, 20, 17, 13, 28, 14, 23, 83, 36, 98, 11, 70, 65, 41, 42, 15

**Quicksort**

Based on the concept of divide and conquer: a list is divided into sublists divided by a pivot.

- Fastest known sorting algorithm
- Basic algorithm: given a list $S$ of numbers:
  - Choose a pivot $v$ from some location in the list.
  - Partition the list into two sublists separated by the pivot:
    - $S_1$ is the sublist having values $> v$
    - $S_2$ is the sublist having values $< v$
  - Quicksort is called recursively on the sublists (when it returns, $S_1$ and $S_2$ will be sorted)
  - The sorted list is $S_1$ followed by $v$ followed by $S_2$.
  - Recursion process stops when a list length of 0 or 1 is reached
Quicksort

- Code example: initial call would be
  \[ qsort(array,0,n-1); \]

  \[
  \text{void qsort(int *array, int left, int right) \{}
  \]
  \[
  \quad \text{int pivot = findpivot(array,left,right);}
  \]
  \[
  \quad \text{swap(array,pivot,right);}
  \]
  \[
  \quad \text{int k = partition(array,left-1,right,array[right]);}
  \]
  \[
  \quad \text{swap(array,k,right);}
  \]
  \[
  \quad \text{if ((k - left) > 1)}
  \]
  \[
  \quad \quad \text{qsort(array,left,k-1);}
  \]
  \[
  \quad \text{if ((right - k) > 1)}
  \]
  \[
  \quad \quad \text{qsort(array,k+1,right);}
  \]
  \[
  \}\]

- int findpivot(int *array, int i, int j) { return (i + j) / 2; }

- int partition(int *a, int l, int r, int &pivot) {
  \[
  \text{do \{}
  \]
  \[
  \quad \text{while (array[++l] < pivot);}
  \]
  \[
  \quad \text{while (r && array[--r] > pivot);}
  \]
  \[
  \quad \text{swap(array,l,r);}
  \]
  \[
  \}\quad \text{while (l < r);}
  \]
  \[
  \quad \text{swap(array,l,r);}
  \]
  \[
  \quad \text{return l;}
  \]

Quicksort

- One pivot and partition run:

<table>
<thead>
<tr>
<th>position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial list</td>
<td>72</td>
<td>6</td>
<td>57</td>
<td>88</td>
<td>60</td>
<td>42</td>
<td>83</td>
<td>73</td>
<td>48</td>
<td>85</td>
</tr>
<tr>
<td>partition swap</td>
<td>72</td>
<td>6</td>
<td>57</td>
<td>88</td>
<td>85</td>
<td>42</td>
<td>83</td>
<td>73</td>
<td>48</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>partition:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pass 1</td>
</tr>
<tr>
<td>swap 1</td>
</tr>
<tr>
<td>pass 2</td>
</tr>
<tr>
<td>swap 2</td>
</tr>
<tr>
<td>pass 3</td>
</tr>
<tr>
<td>swap 3</td>
</tr>
<tr>
<td>reverse swap</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partition relocation</th>
<th>48</th>
<th>6</th>
<th>57</th>
<th>42</th>
<th>85</th>
<th>88</th>
<th>83</th>
<th>73</th>
<th>72</th>
<th>60</th>
</tr>
</thead>
</table>
  | | 48 | 6 | 57 | 42 | 88 | 83 | 73 | 72 | 85 |}

- All values less than 60 are now to its left

- All values greater than 60 are now to its right
Cost of Quicksort

- Best case: always partition in half
  - Cost is $O(n \log n)$

- Worst case: a bad partition
  - Cost is $O(n^2)$

- Average case:
  - Cost is $O(n \log n)$

- Quicksort Optimizations:
  - Choose a better pivot
  - Use a better algorithm for small sublists
  - Eliminate recursion

Mergesort

Based on the concept of merging two sorted lists.

- The general principles:
  - Each list is assumed sorted
  - Merge of two lists is accomplished in one pass
  - Output of merge is placed into a third list

- Pseudocode algorithm:

```plaintext
list mergesort(list inlist) {
    if (length(inlist) == 1)
        return inlist;
    list l1 = first half of inlist;
    list l2 = second half of inlist;
    return merge(mergesort(l1), mergesort(l2));
}
```

- Example:

<table>
<thead>
<tr>
<th>36 20 17 13 28 14 23 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 36 13 17 14 28 15 23</td>
</tr>
<tr>
<td>13 17 20 36 14 15 23 28</td>
</tr>
<tr>
<td>13 14 15 17 20 23 28 36</td>
</tr>
</tbody>
</table>
Heapsort

Heapsort uses a max heap.

- The general procedure:
  - Read in $n$ elements
  - Build the heap
  - Call deleteMax $n$ times in a row
  - The array is sorted at that point.

- Code example:
  ```c
  void heapsort(int *array, int n) {
    heap H(array,n,n);
    for (int i = 0; i < n; i++)
      H.deleteMax();
  }
  ```

Heapsort Example

- Sort the list 73, 6, 57, 88, 60, 42, 83, 72, 48, 85

  - Before buildHeap:
    ```
    Tree (not a heap)
    73
    /  
    6   57
    / 
    88 60
    / 
    42     85
    / 
    72     48
    ```
    Array contents 73 6 57 88 60 42 83 72 48 85

  - After buildHeap:
    ```
    Heap
    88
    /  
    85 73
    / 
    6   42
    / 
    72   57
    / 
    48
    ```
    Array contents 88 85 83 72 73 42 57 6 48 60
Heapsort Example (cont.)

- After first deleteMax:

```
Heap
    85
   /|
  73 83
 /|
72 60 42 57 6 48 88
```

Array contents

85 73 83 72 60 42 57 6 48 88

- After second deleteMax:

```
Heap
    83
   /|
  73 57
 /|
72 60 42 48
```

Array contents

83 73 57 72 60 42 48 6 85 88

CSC 375-Turner, Page 21

Binsort

This is a special-purpose, simple and very efficient sort.

- Given an unsorted array A
  - Let B be the array into which the data is sorted
  - The binsort code is:
    ```
    for (i = 0; i < n; i++)
    B(A[i]) = A[i];
    ```
  - Time complexity: \( \Theta(n) \)

- Why it isn't general:
  - No comparisons are performed
  - It only works on a specific list of numbers:
    - Array A contains exactly \( n \) elements
    - Array A must contain a permutation of the numbers from 0 to \( n \)

- Improvements:
  - Make each bin the head of a list
  - Allow more keys than records

CSC 375-Turner, Page 22
**Improved Binsort**

- **Code:**

  ```c++
  void binsort(int *A, int n) {
    List B[MaxKeyValue];    // an array of lists.
    int item;
    for (i = 0; i < n; i++)
      B[A[i]].append(A[i]);
    for (i = 0; i < MaxKeyValue; i++)
      for (B[i].setFirst(); B[i].isInList(); B[i].next())
        cout << B[i].value() << endl;
  }
  
- **Cost appears to be \( \Theta(n) \)
  - Actual cost also depends on MaxKeyValue

---

**Bucket Sort**

This is a simple generalization of Binsort

- Each bin is associated with a range of values:
  - Assign records to bins
    - Rely on some other sorting technique to sort each bin
    - The other sorting technique is hopefully very efficient.

- Example:
  - Given a sequence of numbers between 0 and 99 inclusive
  - Use 10 bins
  - Assign numbers as follows: \( bin = key \mod 10 \)
Radix Sort

This is one specific type of bin sort.

- General idea:
  - Bin computations are based on the key's radix (its base)
  - Bins are computed in a series of steps using operations mod base
  - Example: for base 10 numbers, there are 10 bins and computations are done mod 10

- Example: sort 26, 93, 3, 97, 15, 77, 23, 48, 82, 87, 55, 36, 54, 46
  - All keys are in the range 0 through $r^2 - 1$ (i.e., $0 \leq \text{key} \leq 99$)
  - First pass: compute bin = key mod $r$ (key mod 10), append to that bin
  - Second pass: compute bin = key/10 mod $r$ (key/10 mod 10), append to that bin

Example:

Initial list: 26, 93, 3, 97, 15, 77, 23, 48, 82, 87, 55, 36, 54, 46
List after second pass: 3, 15, 23, 26, 36, 46, 54, 55, 77, 82, 87, 93, 97
Empirical Comparison

Which algorithm is the fastest?

- Analysis reveals several classes of algorithms but doesn’t distinguish among those in a class

- Some times from Figure 7.13: (data is lists of integers, all times are in seconds)

<table>
<thead>
<tr>
<th>Sort</th>
<th>Size</th>
<th>1K</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
<th>U</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td></td>
<td>2.86</td>
<td>352.1</td>
<td>47241</td>
<td>-</td>
<td>0.0</td>
<td>803.0</td>
</tr>
<tr>
<td>Bubble</td>
<td></td>
<td>9.18</td>
<td>1066.1</td>
<td>123808</td>
<td>-</td>
<td>513.5</td>
<td>812.9</td>
</tr>
<tr>
<td>Selection</td>
<td></td>
<td>5.82</td>
<td>563.5</td>
<td>69437</td>
<td>-</td>
<td>577.8</td>
<td>560.8</td>
</tr>
<tr>
<td>Shell</td>
<td></td>
<td>5.50</td>
<td>9.9</td>
<td>170</td>
<td>3080</td>
<td>2.8</td>
<td>6.1</td>
</tr>
<tr>
<td>Quick</td>
<td></td>
<td>0.33</td>
<td>3.8</td>
<td>49</td>
<td>600</td>
<td>1.7</td>
<td>2.2</td>
</tr>
<tr>
<td>Quick/O</td>
<td></td>
<td>0.27</td>
<td>3.3</td>
<td>44</td>
<td>550</td>
<td>1.7</td>
<td>1.6</td>
</tr>
<tr>
<td>Merge</td>
<td></td>
<td>0.61</td>
<td>60.0</td>
<td>105</td>
<td>1430</td>
<td>6.0</td>
<td>6.1</td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td>0.38</td>
<td>47.2</td>
<td>94</td>
<td>1650</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Radix/8</td>
<td></td>
<td>2.31</td>
<td>23.6</td>
<td>241</td>
<td>2470</td>
<td>23.6</td>
<td>23.6</td>
</tr>
</tbody>
</table>

- U and D columns: data consisted of 10,000 previously sorted in increasing (Up) or decreasing (Down) order
  - Note insert sort performance for U
  - Why is quicksort so good for these two?

General Lower Bound for Sorting

- It is possible to prove a lower bound for all general-purpose sorting algorithms
  - Facts:
    - Sorting is \(O(n \log n)\)
    - Sorting I/O takes \(\Omega(n)\) time
    - This give a “cheap” lower found of \(\Omega(n)\)
  - It can be shown:
    - The problem of sorting is \(\Omega(n \log n)\)
    - Form of the proof:
      - Comparison-based sorting can be modeled by a binary tree
      - It can be shown the tree must have \(\Omega(n!)\) leaves
      - Thus the tree must have \(\Omega(n \log n)\) levels, representing the number of comparisons

CSC 375-Turner, Page 27
CSC 375-Turner, Page 28