Outline

- Introduction (Ch 7.1)
- $\Theta(n^2)$ Sorting (Ch 7.2)
- $\Theta(n \log n)$ Sorting (Ch 7.3 – 7.7)
- Empirical Comparison (Ch 7.8)

Sorting

- Arrange the records in a ascending/descending sequence according to their key values

Definition
Given a sequence of records $R_1, R_2, \ldots, R_n$ with key values $k_1, k_2, \ldots, k_n$, respectively, arrange the records into any orders such that records $R_{s1}, R_{s2}, \ldots, R_{sn}$ have keys obeying the property $k_{s1} \leq k_{s2} \leq \ldots \leq k_{sn}$

Sorting

- Each record contains a field called the key
  - Linear order: comparison

Measure of cost:
- Comparisons
- Swaps
Sorting

- The following sorting methods will be introduced:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Shell Sort
5. Quick Sort
6. Merge Sort
7. Heap Sort
8. Bin Sort
9. Radix Sort

Bubble Sort

- A very well-known method
- Smaller value “bubble” up

For (int i=0; i<n-1; i++)

for (int j=n-1; j>i; j--)

if (Comp::lt(A[j], A[j-1]))

swap(A, j, j-1);

Inner Loop
- Bubble up the small value to the top

Outer Loop
- Repeat the bubble up process for the whole list
**Lec 7: Internal Sorting**

### Bubble Sort

- Number of comparisons made by the inner for loop is always \( i \)

\[
\sum_{i=1}^{n} i = \Theta(n^2)
\]

<table>
<thead>
<tr>
<th></th>
<th>Swap</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>0</td>
<td>( n(n-1)/2 )</td>
</tr>
<tr>
<td>Worst Case</td>
<td>( n(n-1)/2 )</td>
<td>( n(n-1)/2 )</td>
</tr>
<tr>
<td>Average Case</td>
<td>( n(n-1)/4 )</td>
<td>( n(n-1)/2 )</td>
</tr>
</tbody>
</table>

### Selection Sort

- Recall, Bubble Sort:
  - Some swaps are not necessary
  - **Objective**: Find the correct value for a position in each loop

- Selection Sort is a revised version of Bubble Sort

**Selection Sort**

```cpp
template <class Elem, class Comp> void selsort(Elem A[], int n) {
    for (int i=0; i<n-1; i++) {
        int lowindex = i; // Remember its index
        for (int j=n-1; j>i; j--)
            if (Comp::lt(A[j], A[lowindex]))
                lowindex = j; // Put it in place
        if (lowindex != i)
            swap(A, i, lowindex);
    }
}
```

#### Outer Loop

- Repeat the process for the whole list

#### Inner Loop

- Find the correct value for the \( i \)th position

#### Index Checking

- Ensure the index is correct
Selection Sort

<table>
<thead>
<tr>
<th></th>
<th>Swap</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>0</td>
<td>(\frac{n(n-1)}{2})</td>
</tr>
<tr>
<td>Worst Case</td>
<td>(n-1)</td>
<td>(\frac{n(n-1)}{2})</td>
</tr>
<tr>
<td>Average Case</td>
<td>(\frac{n-1}{2})</td>
<td>(\frac{n(n-1)}{2})</td>
</tr>
</tbody>
</table>

- This \(n\) can be removed by deleting the index checking in Outer Loop
  - If the checking is removed, the number of swaps will increase
  - Cost of Swapping VS Cost of Comparison

Insertion Sort

- Add a value to a correction position of a sorted list

\[
\text{template <class Elem, class Comp> void inssort(Elem A[], int n) \{ for (int i=1; i<n; i++) for (int j=i; (j>0)&&(Comp::lt(A[j], A[j-1])); j--) swap(A, j, j-1); \}}
\]

- Inner Loop
  - Find the correct position in the sorted list

- Outer Loop
  - Repeat the process for the whole list by adding a value each time
### Insertion Sort

- Number of comparisons made by the inner for loop is always smaller than or equal to $i$

$$\sum_{i=1}^{n} i = \Theta(n^2)$$

<table>
<thead>
<tr>
<th></th>
<th>Swap</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>0</td>
<td>$n-1$</td>
</tr>
<tr>
<td>Worst Case</td>
<td>$n(n-1)/2$</td>
<td>$n(n-1)/2$</td>
</tr>
<tr>
<td>Average Case</td>
<td>$n(n-1)/4$</td>
<td>$n(n-1)/4$</td>
</tr>
</tbody>
</table>

### Small Exercise!!!!

- **Use**
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

  to sort this list

  - 59
  - 20
  - 17
  - 13
  - 14
  - 15
  - 23
  - 28
  - 42

### Small Exercise!!!!

- **Bubble Sort**

  - 59
  - 13
  - 13
  - 13
  - 13
  - 13
  - 13
  - 20
  - 59
  - 14
  - 14
  - 14
  - 14
  - 14
  - 17
  - 20
  - 59
  - 17
  - 17
  - 17
  - 17
  - 17
  - 13
  - 17
  - 20
  - 59
  - 20
  - 20
  - 20
  - 28
  - 14
  - 17
  - 20
  - 59
  - 23
  - 23
  - 23
  - 23
  - 23
  - 14
  - 28
  - 23
  - 23
  - 59
  - 28
  - 28
  - 23
  - 23
  - 23
  - 23
  - 59
  - 59
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83

### Small Exercise!!!!

- **Selection Sort**

  - 59
  - 13
  - 13
  - 13
  - 13
  - 13
  - 13
  - 13
  - 13
  - 14
  - 14
  - 14
  - 14
  - 14
  - 14
  - 14
  - 14
  - 14
  - 20
  - 20
  - 20
  - 20
  - 20
  - 20
  - 20
  - 20
  - 28
  - 28
  - 28
  - 28
  - 28
  - 28
  - 28
  - 28
  - 28
  - 28
  - 28
  - 28
  - 28
  - 28
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83
  - 83
**Small Exercise!!!! 😊😊**

- **Insertion Sort**

<table>
<thead>
<tr>
<th>59</th>
<th>20</th>
<th>17</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>59</td>
<td>20</td>
<td>17</td>
<td>17</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>59</td>
<td>20</td>
<td>20</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>13</td>
<td>59</td>
<td>28</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>59</td>
<td>28</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>59</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>59</td>
<td>59</td>
</tr>
<tr>
<td>83</td>
<td>83</td>
<td>83</td>
<td>83</td>
<td>83</td>
<td>83</td>
<td>83</td>
<td>83</td>
</tr>
</tbody>
</table>

---

**Comparison**

<table>
<thead>
<tr>
<th>Swaps</th>
<th>Insertion</th>
<th>Bubble</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Average Case</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Worst Case</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comparisons</th>
<th>Insertion</th>
<th>Bubble</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Average Case</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Worst Case</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
</tbody>
</table>


**Comparison**

- Bubble, Selection and Insertion Sorts are called **exchanged sorting**
  - Exchange means swapping adjacent records

- The **running times** of exchanged sorting must be $\Theta(n^2)$
  - Nested for loop is used
Shell Sort

- Try to sort Recall, Insertion Sort:

\[
\begin{array}{cccc}
13 & 13 & 13 & 13 \\
17 & 17 & 17 & 14 \\
20 & 20 & 20 & 17 \\
42 & 28 & 42 & 20 \\
28 & 42 & 14 & 42 \\
\end{array}
\]

Number of swap: 1  
Number of swap: 3

this list is nearly sorted

Insertion Sort is more efficient if the list is nearly sorted
- Number of comparisons can be reduced
- Best case is only $\Theta(n)$

Shell Sort (invented by Donald Shell) try to build a nearly sorted list before sorting the number using insertion sort

Shell Sort

```
// Modified version of Insertion Sort  
template <class Elem, class Comp> void inssort2(Elem A[], int n, int incr) {  
    for (int i=incr; i<n; i+=incr)  
        for (int j=i; (j>=incr) && Comp::lt(A[j], A[j-incr]); j-=incr)  
            swap(A, j, j-incr);
}

template <class Elem, class Comp> void inssort(Elem A[], int n) {  
    for (int i=1; i<n; i++)  
    for (int j=i; (j>0) && Comp::lt(A[j], A[j-1]); j--)  
        swap(A, j, j-1);
}
```

Shell Sort

// Modified version of Insertion Sort

template <class Elem, class Comp>
void inssort2(Elem A[], int n, int incr) {
    for (int i=incr; i<n; i+=incr) {
        for (int j=i; (j>=incr) && Comp::lt(A[j], A[j-incr]); j-=incr)
            swap(A, j, j-incr);
    }
}

A = [0, 1, 2, 3, 4, 5, 6, 7]
incr = 1

A = [2, 2, 2, 2, 2, 2, 2, 2]

Shell Sort

// Modified version of Insertion Sort

template <class Elem, class Comp>
void inssort2(Elem A[], int n, int incr) {
    for (int i=incr; i<n; i+=incr) {
        for (int j=i; (j>=incr) && Comp::lt(A[j], A[j-incr]); j-=incr)
            swap(A, j, j-incr);
    }
}

& A[1] = [0, 1, 2, 3, 4, 5, 6, 7]
incr = 1

incr = 2

Shell Sort

// Modified version of Insertion Sort

template <class Elem, class Comp>
void inssort2(Elem A[], int n, int incr) {
    for (int i=incr; i<n; i+=incr) {
        for (int j=i; (j>=incr) && Comp::lt(A[j], A[j-incr]); j-=incr)
            swap(A, j, j-incr);
    }
}

& A[1] = [0, 1, 2, 3, 4, 5, 6, 7]
incr = 1

 incr = 2

 incr = 2

Shell Sort

// Modified version of Insertion Sort

template <class Elem, class Comp>
void shellsort(Elem A[], int n) {
    // Shellsort
    for (int i=n/2; i>2; i/=2) // For each incr
        for (int j=0; j<i; j++) // Sort sublists
            inssort2(Elem, Comp)(&A[j], n-j, i);
    inssort2(Elem, Comp)(A, n, 1);
}

&A[1] = [0, 1, 2, 3, 4, 5, 6, 7]
in cr = 1

 incr = 2

 incr = 2

Shell Sort

// Modified version of Insertion Sort

template <class Elem, class Comp>
void inssort2(Elem A[], int n, int incr) {
    for (int i=incr; i<n; i+=incr) {
        for (int j=i; (j>=incr) && Comp::lt(A[j], A[j-incr]); j-=incr)
            swap(A, j, j-incr);
    }
}

&A[1] = [0, 1, 2, 3, 4, 5, 6, 7]
in cr = 1

 incr = 2

 incr = 2
Shell Sort

- Select other increments \((\text{incr})\) may improve the performance of shell sort
  - e.g. divide the list by three \((\ldots, 121, 40, 13, 4, 1)\)
- Analysis is difficult
  - Proof will not be included
- Average-case performance is \(O(n^{1.5})\)
- Substantially better than Insertion, Bubble and Selection Sorts

Small Exercise!!!!

- Use Shell Sort to sort this list
  - 59 20 17 13 28 14 23 83

Quick Sort

- Fastest known general-purpose in-memory sorting algorithm in the average case
- A divide and conquer method
- Implement of the concept of Binary Search Tree in an efficient way
Quick Sort

Lec 7: Internal Sorting

Quick Sort

Each Step

template <class Elem, class Comp> void qsort(Elem A[], int i, int j) {
    // Quicksort
    if (j <= i) return; // Don't sort 0 or 1 Elem
    int pivotindex = findpivot(A, i, j);
    swap(A, pivotindex, j); // Put pivot at end
    // k will be the first position in the right subarray
    int k = partition<Elem,Comp>(A, i-1, j, A[j]);
    swap(A, k, j); // Put pivot in place
    qsort<Elem,Comp>(A, i, k-1);
    qsort<Elem,Comp>(A, k+1, j);
}

template <class Elem> int findpivot(Elem A[], int i, int j)
{
    return (i+j)/2;
}

Quick Sort

Lec 7: Internal Sorting
Quick Sort

- The running time is:
  - Best Case: $\Theta(n \log(n))$
  - Worst Case: $\Theta(n^2)$
  - Average Case: $\Theta(n \log(n))$

- The Quick Sort may not be better the Bubble Sort in the worst case

- Optimizations:
  - Better Pivot
  - Better algorithm for small sublists
  - Eliminate recursion

The Quick Sort may not be better than the Bubble Sort in the worst case

Use Quick Sort to sort this list

59 20 17 13 28 14 23 83
Merge Sort

- A divide and conquer method
- Simple to understand but difficult to implement

```
template <class Elem, class Comp> void mergesort(Elem A[], Elem temp[], int left, int right) {
    int mid = (left+right)/2;
    if (left == right) return;
    mergesort<Elem,Comp>(A, temp, left, mid);
    mergesort<Elem,Comp>(A, temp, mid+1, right);
    for (int i=left; i<=right; i++) // Copy
        temp[i] = A[i];
    int i1 = left; int i2 = mid + 1;
    for (int curr=left; curr<=right; curr++) {
        if (i1 == mid+1) // Left exhausted
            A[curr] = temp[i2++];
        else if (i2 > right) // Right exhausted
            A[curr] = temp[i1++];
        else if (Comp::lt(temp[i1], temp[i2]))
            A[curr] = temp[i1++];
        else A[curr] = temp[i2++];
    }
}
```

Some exercise solutions:

1. **Small Exercise!!!**

   - Sort the array: 59, 20, 17, 13, 28, 14, 23, 83
   - Solution: 13, 20, 17, 83, 28, 14, 23, 59

2. **Merge Sort**

   - A divide and conquer method
   - Simple to understand but difficult to implement

```
template <class Elem, class Comp> void mergesort(Elem A[], Elem temp[], int left, int right) {
    int mid = (left+right)/2;
    if (left == right) return;
    mergesort<Elem,Comp>(A, temp, left, mid);
    mergesort<Elem,Comp>(A, temp, mid+1, right);
    for (int i=left; i<=right; i++) // Copy
        temp[i] = A[i];
    int i1 = left; int i2 = mid + 1;
    for (int curr=left; curr<=right; curr++) {
        if (i1 == mid+1) // Left exhausted
            A[curr] = temp[i2++];
        else if (i2 > right) // Right exhausted
            A[curr] = temp[i1++];
        else if (Comp::lt(temp[i1], temp[i2]))
            A[curr] = temp[i1++];
        else A[curr] = temp[i2++];
    }
}
```
Merge Sort

- **Optimizations:**
  - Use insertion sort to sort small sublists

```cpp
template <class Elem, class Comp>
void mergesort(Elem A[], Elem temp[], int left, int right) {
    int mid = (left+right)/2;
    if (left == right) return;
    if ((right-left) > 4) {
        mergesort<Elem,Comp>(A, temp, left, mid);
        mergesort<Elem,Comp>(A, temp, mid+1, right);
    } else
        insertsort(A);
    for (int i=left; i<=right; i++)
        // Copy
        temp[i] = A[i];
    int i1 = left; int i2 = mid + 1;
    for (int curr=left; curr<=right; curr++) {
        if (i1 == mid+1)
            // Left exhausted
            A[curr] = temp[i2++];
        else if (i2 > right)
            // Right exhausted
            A[curr] = temp[i1++];
        else if (Comp::lt(temp[i1], temp[i2]))
            A[curr] = temp[i1++];
        else A[curr] = temp[i2++];
    }
```

- **Optimizations:**
  - Making the two sublists run toward each other
  - Their high ends meet in the middle
  - No need to test for end of list.

```cpp
for (int i=left; i<=mid; i++)
    temp[i] = A[i];
for (j=1; j<=right-mid; j++)
    temp[right-j+1] = A[j+mid];
for (i=left, j=right-mid, k=left;
    k<right; k++)
    if (temp[i] < temp[j])
        A[k] = temp[i++];
    else
        A[k] = temp[j--];
```

- **Running Time:**
  - The best, Average and Worst Cases:
    \( \Theta(n \log n) \)

- **Requires twice the space**
  - Temporarily store the date in merging two sublists
**Small Exercise!!!! 😊😊**
- Use Merge Sort to sort this list

59 20 17 13 28 14 23 83

---

**Heap Sort**
- Quick Sort may face the worst situation
  - Unbalanced BST
- Heap can avoid this problem
- You have learnt Heap in Ch6

---

**Heap Sort**
- Quick Sort may face the worst situation

```
buildheap();
siftdown(4);
siftdown(3);
siftdown(2);
siftdown(1);
siftdown(0);
```

Remove (get) the MAX (root) until the heap is empty
Heap Sort

- The time to build the heap is $\Theta(n)$
- The time to remove the $n$ elements is $\Theta(n \log n)$

So, if we only remove $k$ elements, the total cost is $\Theta(n + k \log n)$

Heap Sort

```cpp
template <class Elem, class Comp> void heapsort(Elem A[], int n) {
    Elem mval;
    maxheap<Elem,Comp> H(A, n, n);
    for (int i=0; i<n; i++)
        H.removemax(mval);
    // Put max at end
}
```

Refer to Chapter 5 (Binary Trees)

Small Exercise!!!! ☺☺ ☺☺

Use Heap Sort to sort this list

59  20  17  13  28  14  23  83
Small Exercise!!!!

Bin Sort

- Maybe the simplest sorting method
template <class Elem>
void binsort(Elem A[], int n) {
  List<Elem> B[MaxKeyValue];
  Elem item;
  for (i=0; i<n; i++) B[A[i]].append(A[i]);
  for (i=0; i<MaxKeyValue; i++)
    for (B[i].setStart(); B[i].getValue(item); B[i].next())
      output(item);
}

Bin Sort

Lec 7: Internal Sorting

Running Time: $\Theta(n + \text{MaxKeyValue})$

Small Exercise!!!! ☺☺ ☺☺

Use Bin Sort to sort this list

59 20 17 13 28 14 23 83

Small Exercise!!!! ☺☺ ☺☺

Use Bin Sort to sort this list

59 20 17 13 28 14 23 83
Radix Sort

- The disadvantage of Bin Sort is the many spaces are waste if the maximum number is very large.
- Radix Sort keep the number of bins and the related processing small.
- Revised Version of Bin Sort

```
template <class Elem, class Comp> void radix(Elem A[], Elem B[], int n, int k, int r, int cnt[]) {
    // cnt[i] stores # of records in bin[i]
    int j;
    for (int i=0, rtok=1; i<k; i++, rtok*=r) {
        for (j=0; j<r; j++) cnt[j] = 0;
        // Count # of records for each bin
        for (j=0; j<n; j++) cnt[(A[j]/rtok)%r]++;
        // cnt[j] will be last slot of bin j.
        for (j=1; j<r; j++) cnt[j] = cnt[j-1] + cnt[j];
        for (j=n-1; j>=0; j--) B[--cnt[(A[j]/rtok)%r]] = A[j];
    }
    for (j=0; j<n; j++) A[j] = B[j];
}
```
Radix Sort

- Can be improved by increasing base \( r \)
  - e.g. \( r = 2^i \)
  - Each time the number of bits is doubled, the number of passes is cut in half
- For Example:
  - \( r_1 = 2^8 = 256 \) can process 1 byte at a time
  - \( r_2 = 2^{16} = 64K \) can process 2 bytes at a time
  - 32bits (4 bytes) data
    - \( r_1 \): 4 passes
    - \( r_2 \): 2 passes

Cost: \( \Theta(k(n + r)) \)
- \( n \): number of distinct values (e.g. 0..10)
- \( k \): maximum number of digits
- \( r \): size of the base (e.g. 2, 10)
- How do \( n \), \( k \), and \( r \) relate?
  - If key range \( (k) \) is small, then this can be \( \Theta(n) \)
    - One pass can complete the sorting
  - If there are \( n \) distinct keys, then the length of a key must be at least \( \log n \)
- Thus, Radix Sort is \( \Theta(n \log n) \) in general case

Small Exercise!!!!

Use Radix Sort to sort this list

59 20 17 13 28 14 23 83
Empirical Comparison

- Which sorting algorithm is fastest?

- We know that \( \Theta(n \log n) \) is faster than \( \Theta(n^2) \)
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Shell Sort
  - Bin Sort
  - Heap Sort
  - Merge Sort
  - Radix Sort

- But how about the algorithms with the same complexity?
- Experiments have been done by sorting 32-bit numbers

Empirical Comparison

- 450MHz Pentium III CPU in Win98
  - Quick/O: Do not partition sub-lists below length 10
  - Shell/O: increments based on division by three
  - Merge/O: Call to Insertion Sort for list of length below 10
  - Radix/4 and /8: 4- and 8-bit-per-pass

- 450MHz Pentium III CPU in Linux
  - Quick/O: Do not partition sub-lists below length 10
  - Shell/O: increments based on division by three
  - Merge/O: Call to Insertion Sort for list of length below 10
  - Radix/4 and /8: 4- and 8-bit-per-pass

Small Exercise!!!! 😊

rtok = 1

A: 
59 20 17 13 28 14 23 83
0 1 2 3 4 5 6 7 8 9
cnt: 1 0 0 3 1 0 0 1 1 1
0 1 2 3 4 5 6 7 8 9
rtok = 10

A: 
20 13 23 83 14 17 28 59
0 1 2 3 4 5 6 7 8 9
cnt: 0 3 3 0 0 1 0 0 1 0
0 1 2 3 4 5 6 7 8 9

Result

A: 
13 14 17 20 23 28 59 83
0 1 2 3 4 5 6 7 8 9

The Best Base
10,000 numbers in correct sorted order

The Worst Case
10,000 numbers in reverse sorted order