What is a sort?

- Sorting is placing a collection of items in order based on a data item (set of data items) called a key.
- Collections are sorted to make it easier to search for a given item.
- Internal sorting is performed with all the records resident in memory.
- External sorting is performed with recorded stored externally.
Efficiency of An Algorithm

• How many times will the innermost loop be performed:
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        cout << x[i][j][k];
  Answer: n^3 times

\[O\] notation

• Given the functions \( f(n) \) and \( g(n) \), we say that \( f(n) = O(g(n)) \) if there exists positive integers \( a \) and \( b \) such that
  \[ f(n) \leq a \times g(n) \quad \text{for all} \quad n \geq b \]
Examples of $O$ notation

- For any constants $c$, $j$, $k$, and $l$:

  \( c = O(1) \)

  \( c = O(\log n) \) but $\log n \neq O(1)$

  \( c \times \log_k n = O(\log n) \)

  \( c \times \log_k n = O(n) \) but $n \neq O(\log n)$

  \( c \times n^k = O(n^k) \) & \( c \times n^k = O(n^{k+1}) \) but $n^{k+1} \neq O(n^k)$

  \( c \times n^j \times (\log_k n)^l = O(n^j \log n)^l \)

  \( c \times n^j \times (\log_k n)^l = O(n^{j+1}) \) but $n^{j+1} \neq O(n^j \log n)^l$
Bubble Sort

/*
 * bubble() - The infamous Bubble Sort
 */
void bubble(int x[], int n)
{
    int hold, i, j, pass;
    int switched = TRUE;

    for (pass = 0; pass < n-1 && switched;
        pass++)
    {
        /* The outer loop controls the number of passes */
        switched = FALSE;
        for (j = 0; j < n-pass-1; j++)
            if (x[j] > x[j+1])
                { /* Elements are out of order 
                   Let's swap */
                    switched = TRUE;
                    hold = x[j];
                    x[j] = x[j+1];
                    x[j+1] = hold;
                }
    }
}
Tracing the Bubble Sort

<table>
<thead>
<tr>
<th>Pass</th>
<th>25</th>
<th>57</th>
<th>48</th>
<th>37</th>
<th>12</th>
<th>92</th>
<th>86</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td>48</td>
<td>37</td>
<td>12</td>
<td>57</td>
<td>86</td>
<td>33</td>
<td>92</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>37</td>
<td>12</td>
<td>48</td>
<td>57</td>
<td>33</td>
<td>86</td>
<td>92</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>12</td>
<td>37</td>
<td>48</td>
<td>33</td>
<td>57</td>
<td>86</td>
<td>92</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>25</td>
<td>37</td>
<td>33</td>
<td>48</td>
<td>57</td>
<td>86</td>
<td>92</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>25</td>
<td>33</td>
<td>37</td>
<td>48</td>
<td>57</td>
<td>86</td>
<td>92</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>25</td>
<td>33</td>
<td>37</td>
<td>48</td>
<td>57</td>
<td>86</td>
<td>92</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>25</td>
<td>33</td>
<td>37</td>
<td>48</td>
<td>57</td>
<td>86</td>
<td>92</td>
</tr>
</tbody>
</table>

Efficiency Of The Bubble Sort

- **Best case** - when completely sorted or almost completely sorted = $O(n)$
- **Worst case** - when random or descending order = $(n-1)^2 = n^2 - 2n + 1 = O(n^2)$
Quick Sort

/*
 * Quick() - The recursive function which partitions the array
 * and sorts the pieces
 */
void quick(int x[], int lb, int ub)
{
    int j = 0;

    if (lb >= ub)
        return;

    partition(x, lb, ub, j);
    quick(x, lb, j-1);
    quick(x, j+1, ub);
}

/*
 * QuickSort() - The main function for the quick sort
 */
void quicksort(int x[], int n)
{
    quick(x, 0, n-1);
}

partition(x, lb, ub, j);
quick(x, lb, j-1);
quick(x, j+1, ub);
Tracing the Quick Sort

25 57 48 37 12 92 86 33
(12) 25 (57 48 37 92 86 33)
12 25 (48 37 33) 57 (92 86)
12 25 (37 33) 48 57 (92 86)
12 25 (33) 37 48 57 (92 86)
12 25 33 37 48 57 (86) 92
12 25 33 37 48 57 86 92

Partition For The Quick Sort

/*
* Partition() - The function used by the Quick Sort to partition array around the pivot element a
*/
void partition(int x[], int lb, int ub, int &j) {
    int a, down, temp, up;
}
/ a is the element whose final
    partition is sought */

    a = x[lb];

    up = ub;
    down = lb;
    while (down < up) {
        while (x[down] <= a && down < ub)
            down++; /* Move up the
                      array */
        while (x[up] > a)
            --up; /* Move down the
                   array */

        if (down < up) {
            /* Swap x[down] and x[up] */
            temp = x[down];
            x[down] = x[up];
            x[up] = temp;
        }
    }

    x[lb] = x[up];
    x[up] = a;
    j = up;
Tracing the Partitioning

25 → 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86
25 57 48 37 12 92 86

Tracing the Partitioning (continued)

25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
25 12 48 37 57 92 86
Efficiency Of The Quick Sort

- **Best case** - when the pivots are all perfectly centered = $O(n \log n)$
- **Worst case** - when in perfect order = $O(n^2)$

---

Selection Sort

```c
/*
 * Selection() - The selection sort which puts
 * the nth largest element in its
 * proper place
 */
void selection(int x[], int n)
{
    int i, indx, j, large;
```
for (i = n-1; i > 0; --i) {
    /* place the largest number of x[0] through x[i] into large and its index in indx */
    large = x[0]; indx = 0;
    for (j = 1; j <= i; j++)
        if (x[j] > large) {
            large = x[j];
            indx = j;
        }
    x[indx] = x[i]; x[i] = large;
}

Tracing the Selection Sort

```
25  57  48  37  12  92  86  33
25  57  48  37  12  33  86  92
25  57  48  37  12  33  86  92
25  33  48  37  12  57  86  92
25  33  12  37  48  57  86  92
25  33  12  37  48  57  86  92
12  25  33  37  48  57  86  92
```
Efficiency Of The Selection Sort

- All cases = $O(n^2)$ - although faster than the Bubble Sort because it makes fewer interchanges.

Heap Sort

```c
/*
 * Heapsort() - The heap sort, which first
 * creates the heap and then uses the
 * heap to place everything in its
 * proper place.
 */
void heapsort(int x[], int n)
{
    int i, elt, son, father, ivalue;
```
/* Preprocessing phase; create the initial heap */
for (i = 1; i < n; i++) {
    elt = x[i];
    /* pqinsert(x, i, elt) */
    son = i;
    father = (son-1)/2;
    while (son > 0 && x[father] < elt){
        x[son] = x[father];
        son = father;
        father = (son-1)/2;
    }
    x[son] = elt;
}

/* Selection phase; Repeatedly remove x[0] and insert it in its proper position and adjust the heap */
for (i = n-1; i > 0; --i) {
    /* pqmaxdelete(x, i+1) */
    ivalue = x[i];
    x[i] = x[0];
    father = 0;
    /* son = largeson(0, i-1) */
    if (i == 1)
        son = -1;
    else
        son = 1;
if (i > 2 && x[2] > x[1])
    son = 2;

while (son >= 0 && ivalue < x[son]){
    x[father] = x[son];
    father = son;

    /* son = largeson(f, i-1) */
    son = 2*father+1;
    if (son+1 <= i-1 && x[son] < x[son+1])
        son++;
}

if (son > i -1)
    son = -1;
}

x[father] = ivalue;
}
Building the Heap

Original Array

25
57
37
12
92
86
33

Building the Heap

25
57
25

25
57
25


25 57 48 37 12 92 86 33
Building the Heap (continued)

```
57
  37
  48
  25

57 37 48 25
```

Building the Heap (continued)

```
57
  37
  48
  25
  12

57 37 48 25 12
```
Building the Heap (continued)

```
92 37 57 25 12 48
```

Building the Heap (continued)

```
92 37 86 25 12 48 57
```
Building the Heap (continued)

Using the Heap
Using the Heap (continued)

Using the Heap (continued)
Using the Heap (continued)

Using the Heap (continued)
Using the Heap (continued)

Using the Heap (continued)
Efficiency Of The Heap Sort

- All cases = $O(n \log n)$ - although slower than the best case of the Quick Sort because of the time it takes to build the Heap and the number of interchanges.
insertion sort

void insertion(int x[], int n)
{
    int i, k, y;

    for (k = 1; k < n; k++) {
        y = x[k];
        for (i = k-1; i >= 0 && y < x[i]; --i)
            x[i+1] = x[i];
    }
/* Insert \( y \) into its proper position */
\[
x[i+1] = y;
\]

Tracing the Insertion Sort

<table>
<thead>
<tr>
<th>12</th>
<th>25</th>
<th>37</th>
<th>48</th>
<th>57</th>
<th>92</th>
<th>86</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>57</td>
<td>48</td>
<td>37</td>
<td>12</td>
<td>92</td>
<td>86</td>
<td>33</td>
</tr>
<tr>
<td>25</td>
<td>57</td>
<td>48</td>
<td>37</td>
<td>12</td>
<td>92</td>
<td>86</td>
<td>33</td>
</tr>
<tr>
<td>25</td>
<td>48</td>
<td>57</td>
<td>37</td>
<td>12</td>
<td>92</td>
<td>86</td>
<td>33</td>
</tr>
<tr>
<td>25</td>
<td>37</td>
<td>48</td>
<td>57</td>
<td>12</td>
<td>92</td>
<td>86</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>37</td>
<td>48</td>
<td>57</td>
<td>92</td>
<td>86</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>37</td>
<td>48</td>
<td>57</td>
<td>86</td>
<td>92</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>37</td>
<td>48</td>
<td>57</td>
<td>86</td>
<td>92</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>33</td>
<td>37</td>
<td>48</td>
<td>57</td>
<td>86</td>
<td>92</td>
</tr>
</tbody>
</table>
Efficiency Of The Insertion Sort

- All cases = $O(n^2)$ - although faster than the Bubble Sort because it makes fewer interchanges.

Shell Sort

```c
/*
 * ShellSort() - The shell, an insertion sort
 * which uses a collection of different
 * increment to merge different data
 * sets
 */
void shellsort(int x[], int n)
{
    int incr, j, k, span, y;
```
/* Span is the size of the increment.
The largest is half the array size
and it is reduced by half until
it's 1 */
for (span = n/2;  span >= 1;  span /= 2)
   for (j = span;  j < n;  j++) {
      /* Insert x[j] into its
         proper place within
         its subfile */
y = x[j];
      for  (k = j-span;
         k >= 0 && y < x[k];
         k -= span)
x[k+span] = x[k];
   }
x[k+span] = y;
Tracing the Shell Sort

25 57 48 37 12 92 86 33

12 57 48 33 25 92 86 37

12 33 25 37 48 57 86 92

Tracing the Shell Sort (continued)

12 33 25 37 48 57 86 92

12 25 33 37 48 57 86 92
Efficiency Of The Shell Sort

• All cases ≈ $O(n(\log n)^2)$ - although the exact values depends on the set of increments used.

Merge Sort

```c
#define NUMELTS 100
void mergesort (int x[], int n)
{
    int aux[NUMELTS], i, j, k,
        l1, l2, size, u1, u2;

    size = 1; /* Initially merge files of size 1 */
```
while (size < n) {
    l1 = 0;   /* Initialize lower bound of 1st file */
    k = 0;   /* Index for auxiliary array */
    /* Check to see if there are two files to merge */
    while (l1+size < n)
        /* Compute remaining indices */
        l2 = l1 + size;
        u1 = l2 - 1;
        u2 = (l2 + size -1 < n)?
            l2 + size - 1: n-1;
    /* Proceed through the two subfiles */
    for (i = l1, j = l2;
        i <= u1 && j <= u2;  k++)
        if (x[i] <=  x[j])
            aux[k] = x[i++];
        else
            aux[k] = x[j++];
    /* At this point, one of the two subfiles is exhausted. Insert any remaining portion of the other file */
for (; i <= u1; k++)
    aux[k] = x[i++];
for (; j <= u2; k++)
    aux[k] = x[j++];

/* Advance l1 to the start of the next pair of subfiles */
    l1 = u2 + 1;
}

/* Copy any remaining single subfile */
for (i = l1; k < n; i++)
    aux[k++] = x[i];

/* Copy aux back into x and adjust the subfile size */
for (i = 0; i < n; i++)
    x[i] = aux[i];
    size *= 2;
Efficiency Of The Merge Sort

- The merge sort never requires more than $n \log n$ comparisons but requires an extra $O(n)$ storage.
/*
* Power(x, n) - Raises x to the nth power -
* not supplied with Turbo C++
*/

int power(int x, int n)
{
    int i, product = 1;

    for (i = 0; i < n; i++)
        product *= x;

    return(product);
}

/*
* RadixSort() - Sorts numbers based by queueing them repeatedly by a particular digit (least to most significant)
*/

void radixsort(int x[], int n)
{
    int front[10], rear[10];
    struct {
        int info;
        int next;
    } node[NUMELTS];
int exp, first, i, j, k, p, q, y;

/* Initialize linked list */
for (i = 0; i < n-1; i++) {
    node[i].info = x[i];
    node[i].next = i+1;
}
node[n-1].info = x[n-1];
node[n-1].next = -1;
first = 0; /* First is the head of the
    linked list */

for (k = 1; k < 5; k++) {
    /* Assume that we have four-digit
        numbers */
    /* Initialize the queues */
    for (i = 0; i < 10; i++) {
        rear[i] = -1;
        front[i] = -1;
    }
}
/* Process each element on the list */
while (first != -1) {
    p = first;
    first = node[first].next;
    y = node[p].info;

    /* Extract the kth digit */
    exp = power(10, k-1);
    j = (y/exp) % 10;

    j = (y/exp) % 10;

    /* Insert y into queue[j] */
    q = rear[j];
    if (q == -1)
        front[j] = p;
    else
        node[q].next = p;
    rear[j] = p;
}

/* Insert y into queue[j] */
q = rear[j];
if (q == -1)
    front[j] = p;
else
    node[q].next = p;
rear[j] = p;
}
/* At this point, each record is
in its proper queue based on digit k
We now form a single list from all
the queues. */
for (j = 0;
    j < 10 && front[j] == -1;
    j++)
    ;
first = front[j];

/* Link up remaining queues
First, check to see if finished */
while (j <= 9)  {
    /* Find the next element */
    for (i = j+1;
        i < 10
            && front[i] == -1;
        i++)
            ;
    if (i <= 9) {
        p = i;
node[rear[j]].next = front[i];

j = i;

node[rear[p]].next = -1;

/* Copy back to the original array */
for (i = 0;  i < n;  i++)    {
    x[i] = node[first].info;
    first = node[first].next;
}
}
## Tracing the Radix Sort

<table>
<thead>
<tr>
<th></th>
<th>25</th>
<th>57</th>
<th>48</th>
<th>37</th>
<th>12</th>
<th>92</th>
<th>86</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
<td>rear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[2]</td>
<td>12</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[3]</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[5]</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[6]</td>
<td>86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[7]</td>
<td>37</td>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[8]</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>92</th>
<th>33</th>
<th>25</th>
<th>86</th>
<th>57</th>
<th>37</th>
<th>48</th>
</tr>
</thead>
</table>

## Tracing the Radix Sort (continued)

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>92</th>
<th>33</th>
<th>25</th>
<th>86</th>
<th>57</th>
<th>37</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
<td>rear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[1]</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[2]</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[3]</td>
<td>33</td>
<td>37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[4]</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[5]</td>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[6]</td>
<td>86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue[7]</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|   | 12 | 25 | 33 | 37 | 48 | 57 | 86 | 92 |
Efficiency Of The Radix Sort

- The efficiency = $O(m \times n)$ storage for $m$ digits and $n$ records, and can approach $O(n \log n)$