Chapter 10

Sorting and Searching Algorithms

Sorting

- **Given** a set (container) of \( n \) elements
  - E.g. array, set of words, etc.
- **Suppose** there is an order relation that can be set across the elements
- **Goal** Arrange the elements in ascending order or descending order
  - Start \( \rightarrow 1 \ 23 \ 2 \ 56 \ 9 \ 8 \ 10 \ 100 \)
  - End \( \rightarrow 1 \ 2 \ 8 \ 9 \ 10 \ 23 \ 56 \ 100 \)
Straight Selection Sort

Divides the array into two parts: already sorted, and not yet sorted.

On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.

Selection Sort: Pass One

On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.
Selection Sort: End Pass One

<table>
<thead>
<tr>
<th>values [0]</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>24</td>
</tr>
<tr>
<td>[2]</td>
<td>10</td>
</tr>
<tr>
<td>[3]</td>
<td>36</td>
</tr>
<tr>
<td>[4]</td>
<td>12</td>
</tr>
</tbody>
</table>

Selection Sort: Pass Two

<table>
<thead>
<tr>
<th>values [0]</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>24</td>
</tr>
<tr>
<td>[2]</td>
<td>10</td>
</tr>
<tr>
<td>[3]</td>
<td>36</td>
</tr>
<tr>
<td>[4]</td>
<td>12</td>
</tr>
</tbody>
</table>
Selection Sort: End Pass Two

values [ 0 ] 6
[ 1 ] 10
[ 2 ] 24
[ 3 ] 36
[ 4 ] 12

Selection Sort: Pass Three

values [ 0 ] 6
[ 1 ] 10
[ 2 ] 24
[ 3 ] 36
[ 4 ] 12
Selection Sort: End Pass Three

Selection Sort: Pass Four
Selection Sort: End Pass Four

Selection Sort: How many comparisons?

4 compares for values[0]
3 compares for values[1]
2 compares for values[2]
1 compare for values[3]

= 4 + 3 + 2 + 1
For selection sort in general

- The number of comparisons when the array contains N elements is
  \[ \text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1 \]

Notice that . . .

\[ \begin{align*}
\text{Sum} &= (N-1) + (N-2) + \ldots + 2 + 1 \\
+ \text{Sum} &= 1 + 2 + \ldots + (N-2) + (N-1) \\
2 \times \text{Sum} &= N + N + \ldots + N + N \\
2 \times \text{Sum} &= N \times (N-1) \\
\text{Sum} &= \frac{N \times (N-1)}{2}
\end{align*} \]
For selection sort in general

• The number of comparisons when the array contains N elements is

\[ \text{Sum} = (N-1) + (N-2) + \ldots + 2 + 1 \]

\[ \text{Sum} = N \times (N-1) / 2 \]

\[ \text{Sum} = .5 \ N^2 - .5 \ N \]

\[ \text{Sum} = O(N^2) \]

---

Number of Comparisons

<table>
<thead>
<tr>
<th>Number of Items</th>
<th>Number of Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>45</td>
</tr>
<tr>
<td>20</td>
<td>190</td>
</tr>
<tr>
<td>100</td>
<td>4950</td>
</tr>
<tr>
<td>1000</td>
<td>499500</td>
</tr>
<tr>
<td>10000</td>
<td>49995000</td>
</tr>
</tbody>
</table>
### Bubble Sort

Comparing neighboring pairs of array elements, starting with the last array element, and swaps neighbors whenever they are not in correct order.

On each pass, this causes the smallest element to “bubble up” to its correct place in the array.

---

<table>
<thead>
<tr>
<th>values</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Observations on BubbleSort

\[(N-1)+(N-2)+(N-3) + \ldots + 1\]

- This algorithm is always \(O(N^2)\).

- There can be a large number of intermediate swaps, more swaps than Selection Sort.
Insertion Sort

Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.
A Snapshot of the Insertion Sort Algorithm

Sorting Algorithms and Average Case Number of Comparisons

Simple Sorts
- Straight Selection Sort
- Bubble Sort
- Insertion Sort

More Complex Sorts
- Heap Sort
- Quick Sort
- Merge Sort

\[ O(N^2) \]

\[ O(N\log N) \]
Recall that . . .

A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- Its shape must be a complete binary tree.
- For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.

The largest element in a heap is always found in the root node.
Heap Sort Approach

First, make the unsorted array into a heap by satisfying the order property. Then repeat the steps below until there are no more unsorted elements.

- Take the root (maximum) element off the heap by swapping it into its correct place in the array at the end of the unsorted elements.
- Reheap the remaining unsorted elements. (This puts the next-largest element into the root position).
An unsorted array and its tree

values

[0] 30
[1] 12
[2] 60
[3] 40
[4] 70
[5] 8
[6] 10

The heap-building Process (1)

All leaf nodes are heaps already!!
Work on subtrees rooted at a non-leaf node – ReheapDown()

Work on subtree rooted at next-level non-leaf node – ReheapDown()
The heap-building Process (4)

A heap!

After creating the original heap
Swap root element into last place in unsorted array

values

[ 0 ] 70
[ 1 ] 40
[ 2 ] 60
[ 3 ] 30
[ 4 ] 12
[ 5 ] 8
[ 6 ] 10

After swapping root element into it place

values

[ 0 ] 10
[ 1 ] 40
[ 2 ] 60
[ 3 ] 30
[ 4 ] 12
[ 5 ] 8
[ 6 ] 70

NO NEED TO CONSIDER AGAIN
After reheaping remaining unsorted elements

After swapping root element into its place

ALL ELEMENTS ARE SORTED
Heap Sort: How many comparisons?

In ReheapDown, an element is compared with its 2 children (and swapped with the larger). But only one element at each level makes this comparison, and a complete binary tree with N nodes has only $O(\log_2 N)$ levels.

Heap Sort of N elements: How many comparisons?

$\frac{N}{2} \times O(\log N)$ compares to create original heap

$(N-1) \times O(\log N)$ compares for the sorting loop

$= O(N \times \log N)$ compares total

• Good for large data set.
Quick Sort

• It is faster and easier to sort two small lists than one larger list.
• Strategy: Divide and Conquer.
  – Divide the unsorted list into two sub-lists for a given split point.
  – Smaller values are in left sub-list.
  – Larger values are in right sub-list.
  – The process is executed recursively.

Using quick sort algorithm
Before call to function Split

\( \text{splitVal} = 9 \)

**GOAL:** place \( \text{splitVal} \) in its proper position with all values less than or equal to \( \text{splitVal} \) on its left and all larger values on its right

<table>
<thead>
<tr>
<th>9</th>
<th>20</th>
<th>6</th>
<th>18</th>
<th>14</th>
<th>3</th>
<th>60</th>
<th>11</th>
</tr>
</thead>
</table>

values[first] [last]

After call to function Split

\( \text{splitVal} = 9 \)

smaller values in left part

larger values in right part

<table>
<thead>
<tr>
<th>6</th>
<th>3</th>
<th>9</th>
<th>18</th>
<th>14</th>
<th>20</th>
<th>60</th>
<th>11</th>
</tr>
</thead>
</table>

values[first] [last]

splitVal in correct position
// Recursive quick sort algorithm
void QuickSort ( ItemType values[ ], int first , int last )
// Pre: first <= last
// Post: Sorts array values[ first . . last ] into ascending order
{
    if ( first < last )  // general case
    {
        int splitPoint ;
        Split ( values, first, last, splitPoint ) ;
        // values [first]..values[splitPoint - 1] <= splitVal
        // values [splitPoint] = splitVal
        // values [splitPoint + 1]..values[last] > splitVal
        QuickSort(values, first, splitPoint - 1);
        QuickSort(values, splitPoint + 1, last);
    }
}

Quick Sort of N elements:
How many comparisons?

N For first call, when each of N elements is compared to the split value
2 * N/2 For the next pair of calls, when N/2 elements in each “half” of the original array are compared to their own split values.
4 * N/4 For the four calls when N/4 elements in each “quarter” of original array are compared to their own split values.

HOW MANY SPLITS CAN OCCUR?
Quick Sort of N elements:  
How many splits can occur?  

It depends on the order of the original array elements!

If each split divides the subarray approximately in half, there will be only $\log_2 N$ splits, and QuickSort is $O(N \cdot \log_2 N)$.

But, if the original array was sorted to begin with, the recursive calls will split up the array into parts of unequal length, with one part empty, and the other part containing all the rest of the array except for split value itself. In this case, there can be as many as $N-1$ splits, and QuickSort is $O(N^2)$.

Before call to function Split

$\text{splitVal} = 9$

**GOAL:** place $\text{splitVal}$ in its proper position with  
all values less than or equal to $\text{splitVal}$ on its left  
and all larger values on its right

<table>
<thead>
<tr>
<th>9</th>
<th>20</th>
<th>26</th>
<th>18</th>
<th>14</th>
<th>53</th>
<th>60</th>
<th>11</th>
</tr>
</thead>
</table>

values[first] [last]
After call to function Split

splitVal = 9

no smaller values
empty left part

larger values
in right part with N-1 elements

values[first]          [last]
9  20  26  18  14  53  60  11

splitVal in correct position

Merge Sort Algorithm

Cut the array in half.
Sort the left half.
Sort the right half.
Merge the two sorted halves into one sorted array.
// Recursive merge sort algorithm
void MergeSort ( ItemType values[], int first, int last )
  // Pre: first <= last
  // Post: Array values[first..last] sorted into ascending order.
{
  if ( first < last ) // general case
  {
    int middle = ( first + last ) / 2 ;
    MergeSort ( values, first, middle ) ;
    MergeSort( values, middle + 1, last ) ;

    // now merge two subarrays
    // values [ first . . . middle ] with
    // values [ middle + 1, . . . last ].
    Merge(values, first, middle, middle + 1, last);
  }
}
Merging Two halves

- O(N) Task
- Go through the sorted halves, comparing successive pairs of values (one in each half) and putting the smaller value into next available spot in the final solution.

<table>
<thead>
<tr>
<th></th>
<th>36</th>
<th>74</th>
<th>...</th>
<th>95</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>29</td>
<td>52</td>
<td>...</td>
<td>75</td>
</tr>
</tbody>
</table>

Merge Sort of N elements:
How many comparisons?

- The entire array can be subdivided into halves only $\log_2 N$ times.
- Each time it is subdivided, function Merge is called to re-combine the halves. Merging is O(N) because it compares each element in the subarrays.

**MERGE SORT IS O(N*\log_2 N).**
## Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Sort</th>
<th>Best Case</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>selectionSort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>bubbleSort</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>shortBubble</td>
<td>$O(N)$ (*)</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>insertionSort</td>
<td>$O(N)$ (*)</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>mergeSort</td>
<td>$O(N \log_2 N)$</td>
<td>$O(N \log_2 N)$</td>
<td>$O(N \log_2 N)$</td>
</tr>
<tr>
<td>quickSort</td>
<td>$O(N \log_2 N)$</td>
<td>$O(N \log_2 N)$</td>
<td>$O(N^2)$ (depends on split)</td>
</tr>
<tr>
<td>heapSort</td>
<td>$O(N \log_2 N)$</td>
<td>$O(N \log_2 N)$</td>
<td>$O(N \log_2 N)$</td>
</tr>
</tbody>
</table>

*Data almost sorted.*

## Testing

- To thoroughly test our sorting methods we should vary the size of the array they are sorting.
- Vary the original order of the array-test
  - Reverse order
  - Almost sorted
  - All identical elements
Sorting Objects

- When sorting an array of objects we are manipulating references to the object, and not the objects themselves.

![Sorting Objects Diagram]

Searching

- **Linear Search** – O(N)
  - Search a unsorted list
- **Binary Search** – O(log₂N)
  - Search a sorted list
- **Searching with Hashing** – O(1)
  - Search a hash table
Hashing

• Hashing is a means used to order and access elements in a list quickly -- the goal is $O(1)$ time -- by using a function of the key value to identify its location in the list.

• The function of the key value is called a hash function.

FOR EXAMPLE . . .

Using a hash function

<table>
<thead>
<tr>
<th>values</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>Empty</td>
</tr>
<tr>
<td>[1]</td>
<td>4501</td>
</tr>
<tr>
<td>[2]</td>
<td>Empty</td>
</tr>
<tr>
<td>[3]</td>
<td>7803</td>
</tr>
<tr>
<td>[4]</td>
<td>Empty</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>[97]</td>
<td>Empty</td>
</tr>
<tr>
<td>[98]</td>
<td>2298</td>
</tr>
<tr>
<td>[99]</td>
<td>3699</td>
</tr>
</tbody>
</table>

*HandyParts* company makes no more than 100 different parts. But the parts all have four digit numbers.

Do we need an array of size 10000 to hold all products?

This hash function can be used to store and retrieve parts in an array.

$\text{Hash(key)} = \text{partNum} \% 100$
Use the hash function

\[ \text{Hash(key)} = \text{partNum} \mod 100 \]

to place the element with part number 5502 in the array.

Next place part number 6702 in the array.

\[ \text{Hash(key)} = \text{partNum} \mod 100 \]

\[ 6702 \mod 100 = 2 \]

But values[2] is already occupied.

**COLLISION OCCURS**
How to Resolve the Collision - Rehash

One way is by linear probing. This uses the \textit{rehash} function
\[(\text{HashValue} + 1) \mod 100\]
repeatedly until an empty location is found for part number 6702.

Resolving the Collision

Still looking for a place for 6702 using the function
\[(\text{HashValue} + 1) \mod 100\]
### Collision Resolved

<table>
<thead>
<tr>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 0 ]</td>
</tr>
<tr>
<td>[ 1 ]</td>
</tr>
<tr>
<td>[ 2 ]</td>
</tr>
<tr>
<td>[ 3 ]</td>
</tr>
<tr>
<td>[ 4 ]</td>
</tr>
<tr>
<td>[ 97]</td>
</tr>
<tr>
<td>[ 98]</td>
</tr>
<tr>
<td>[ 99]</td>
</tr>
</tbody>
</table>

Part 6702 can be placed at the location with index 4.

---

### Collision Resolved

<table>
<thead>
<tr>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 0 ]</td>
</tr>
<tr>
<td>[ 1 ]</td>
</tr>
<tr>
<td>[ 2 ]</td>
</tr>
<tr>
<td>[ 3 ]</td>
</tr>
<tr>
<td>[ 4 ]</td>
</tr>
<tr>
<td>[ 97]</td>
</tr>
<tr>
<td>[ 98]</td>
</tr>
<tr>
<td>[ 99]</td>
</tr>
</tbody>
</table>

Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?
### How to Resolve the Collision – Buckets or Chaining

<table>
<thead>
<tr>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 0 ]</td>
</tr>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>[ 1 ]</td>
</tr>
<tr>
<td>4501</td>
</tr>
<tr>
<td>[ 2 ]</td>
</tr>
<tr>
<td>5502</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>6702</td>
</tr>
<tr>
<td>[ 3 ]</td>
</tr>
<tr>
<td>7803</td>
</tr>
<tr>
<td>[ 4 ]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>[ 97 ]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>[ 98 ]</td>
</tr>
</tbody>
</table>

---

### Use of the Hash Function

- Find a place to store a particular element (Inserting).
- Look for a particular element (Retrieving-Searching)
Other Hash Method

• Elements are string, not integers?
  – Use the internal representations of string characters (ASCII codes)
Reference

• Reproduced by permission of Jones and Bartlett Publishers International.