CMSC 132: Object-Oriented Programming II

Sorting

Department of Computer Science
University of Maryland, College Park
Overview

Comparison sort
- Bubble sort
- Selection sort
- Tree sort
- Heap sort
- Quick sort
- Merge sort

Linear sort
- Counting sort
- Bucket (bin) sort
- Radix sort

\[ O(n^2) \]
\[ O(n \log(n)) \]
\[ O(n) \]
Sorting

Goal

- Arrange elements in **predetermined order**
  - Based on **key** for each element
- Derived from ability to **compare** two keys by size

Properties

- **Stable** ⇒ relative order of **equal** keys unchanged
  - Stable: $3, 1, 4, 3, 3, 2 \rightarrow 1, 2, 3, 3, 3, 4$
  - Unstable: $3, 1, 4, 3, 3, 2 \rightarrow 1, 2, 3, 3, 3, 4$
- **In-place** ⇒ uses only constant additional space
- **External** ⇒ can efficiently sort large # of keys
Sorting

Comparison sort
- Only uses pairwise key comparisons
- Proven lower bound of $O(n \log(n))$

Linear sort
- Uses additional properties of keys
Bubble Sort

Approach
1. Iteratively sweep through shrinking portions of list
2. Swap element $x$ with its right neighbor if $x$ is larger

Performance
- $O(n^2)$ average / worst case
Bubble Sort Example

Sweep 1

```
7 2 8 5 4
2 7 8 5 4
2 7 8 5 4
2 7 5 8 4
2 7 5 4 8
```

Sweep 2

```
2 7 5 4 8
2 7 5 4 8
2 5 7 4 8
2 5 4 7 8
2 4 5 7 8
```

Sweep 3

```
2 5 4 7 8
2 5 4 7 8
2 4 5 7 8
```

Sweep 4

```
2 4 5 7 8
2 4 5 7 8
2 4 5 7 8
```
```c
void bubbleSort(int[ ] a) {
    int outer, inner;
    for (outer = a.length - 1; outer > 0; outer--)
        for (inner = 0; inner < outer; inner++)
            if (a[inner] > a[inner + 1])
                swap(a, inner, inner+1);
}

void swap(int a[ ], int x, int y) {
    int temp = a[x];
    a[x] = a[y];
    a[y] = temp;
}
```
Selection Sort

Approach
1. Iteratively sweep through shrinking portions of list
2. Select smallest element found in each sweep
3. Swap smallest element with front of current list

Performance
- $O(n^2)$ average / worst case

Example
void selectionSort(int[] a) {
    int outer, inner, min;
    for (outer = 0; outer < a.length - 1; outer++) {
        min = outer;
        for (inner = outer + 1; inner < a.length; inner++) {
            if (a[inner] < a[min]) {
                min = inner;
            }
        }
        swap(a, outer, min);
    }
}
Tree Sort

Approach
1. Insert elements in binary search tree
2. List elements using inorder traversal

Performance
- Binary search tree
  - $O(n \log(n))$ average case
  - $O(n^2)$ worst case
- Balanced binary search tree
  - $O(n \log(n))$ average / worst case

Example
Binary search tree

{ 7, 2, 8, 5, 4 }
Heap Sort

Approach
1. Insert elements in heap
2. Remove smallest element in heap, repeat
3. List elements in order of removal from heap

Performance
O(n log(n)) average / worst case

Example
Heap
{ 7, 2, 8, 5, 4 }
Quick Sort

Approach

1. Select pivot value (near median of list)
2. Partition elements (into 2 lists) using pivot value
3. Recursively sort both resulting lists
4. Concatenate resulting lists

For efficiency pivot needs to partition list evenly

Performance

- O( n log(n) ) average case
- O( n^2 ) worst case
Quick Sort Algorithm

1. **If list below size K**
   - Sort w/ other algorithm

2. **Else pick pivot x and partition S into**
   - L elements < x
   - E elements = x
   - G elements > x

3. **Quicksort L & G**

4. **Concatenate L, E & G**
   - If not sorting in place
void quickSort(int[] a, int x, int y) {
    int pivotIndex;
    if ((y - x) > 0) {
        pivotIndex = partionList(a, x, y);
        quickSort(a, x, pivotIndex - 1);
        quickSort(a, pivotIndex+1, y);
    }
}

int partionList(int[] a, int x, int y) {
    ... // partitions list and returns index of pivot
}
Quick Sort Example

Partition & Sort

Result
int partitionList(int[] a, int x, int y) {
    int pivot = a[x];
    int left = x;
    int right = y;
    while (left < right) {
        while ((a[left] < pivot) && (left < right))
            left++;
        while (a[right] > pivot)
            right--;
        if (left < right)
            swap(a, left, right);
    }
    swap(a, x, right);
    return right;
}
Merge Sort

Approach

1. Partition list of elements into 2 lists
2. Recursively sort both lists
3. Given 2 sorted lists, merge into 1 sorted list
   a) Examine head of both lists
   b) Move smaller to end of new list

Performance

- $O(n \log(n))$ average / worst case
Merge Example
Merge Sort Example

Split

Merge
void mergeSort(int[] a, int x, int y) {
    int mid = (x + y) / 2;
    if (y == x) return;
    mergeSort(a, x, mid);
    mergeSort(a, mid+1, y);
    merge(a, x, y, mid);
}
void merge(int[] a, int x, int y, int mid) {
    ... // merges 2 adjacent sorted lists in array
}
void merge (int[] a, int x, int y, int mid) {
    int size = y - x;
    int left = x;
    int right = mid+1;
    int[] tmp; int j;
    for (j = 0; j < size; j++) {
        if (left > mid) tmp[j] = a[right++];
        else if (right > y) || (a[left] < a[right])
            tmp[j] = a[left++];
        else tmp[j] = a[right++];
    }
    for (j = 0; j < size; j++)
        a[x+j] = tmp[j];
}
## Sorting Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Comparison Sort</th>
<th>Avg Case Complexity</th>
<th>Worst Case Complexity</th>
<th>In Place</th>
<th>Can be Stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>√</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Selection</td>
<td>√</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Tree</td>
<td>√</td>
<td>O(n \log(n))</td>
<td>O(n^2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td>√</td>
<td>O(n \log(n))</td>
<td>O(n \log(n))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick</td>
<td>√</td>
<td>O(n \log(n))</td>
<td>O(n^2)</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td>√</td>
<td>O(n \log(n))</td>
<td>O(n \log(n))</td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>
Sorting Summary

- Many different sorting algorithms
- Complexity and behavior varies
- Size and characteristics of data affect algorithm