CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Sorting

Department of Computer Science
University of Maryland, College Park
Sorting

• Goal
  • Arrange elements in **predetermined** order
    • Based on **key** for each element
  • Derived from ability to **compare** two keys

• Properties
  • **Stable** → relative order of **equal** keys unchanged
    • Stable: 3, 1, 4, 3, 3, 2 → 1, 2, 3, 3, 3, 4
    • Unstable: 3, 1, 4, 3, 3, 2 → 1, 2, 3, 3, 3, 4
  • **In-place** → uses only constant additional space
  • **Internal** → all data being sorted fits into main memory
  • **External** → required when the data being sorted do not fit into main memory
  • **Adaptive** → Algorithm takes into account whether the data is sorted or partially sorted
    • Runs faster the more sorted the data is initially
  • Most algorithms discussed in lecture are internal and based on arrays
Type of Sorting Algorithms

- **Comparison-based and Linear Algorithms**
  - Comparison-based Algorithms → Only uses pairwise key comparisons
  - Linear Algorithms → Uses additional properties of keys

- **Comparison-based**
  - Proven lower bound of Omega(n log(n))
  - Examples
    - $O(n^2)$ → Bubblesort, Selection sort, Insertion sort
    - $O(n \log(n))$ → Treesort, Heapsort, Quicksort, Mergesort

- **Linear Algorithms**
  - Counting sort
  - Bucket (bin) sort
  - Radix sort
Bubble Sort

• Approach
  • Iteratively sweep through shrinking portions of list
  • Swap element $x$ with its right neighbor if $x$ is larger
  • After each pass largest element placed at the bottom (lighter elements bubble up)

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

Sweep 1

Sweep 2

Sweep 3

Sweep 4
Bubble Sort Code

```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;
}
```

How can we improve it?

Swap with right neighbor if larger
Selection Sort

• Approach
  • Iteratively sweep through shrinking portions of list
  • Select smallest element found in each sweep
  • Swap smallest element with front of current list

• Performance
  • $O(n^2)$ average / worst case
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);
    }
}
Insertion Sort

• Similar to method used when sorting cards
• Although is $O(n^2)$ it is the preferred algorithm when data is nearly sorted or when the size of the data to sort is small (due to low overhead). For these reasons it is often used in algorithms like quicksort when the size of the data is small
• Has a simple implementation
• More efficient in practice than most simple quadratic algorithms
• It is adaptive
void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int j, temp = a[i];

        for (j = i - 1; j >= 0 && temp < a[j]; j--) {
            a[j + 1] = a[j];
        }
        a[j + 1] = temp;
    }
}
Tree Sort

• **Approach**
  - Insert elements in binary search tree
  - 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
Heap Sort

• Approach
  • Insert elements in heap
  • Remove smallest element in heap, repeat
  • List elements in order of removal from heap

• Performance
  • $O(n \log(n))$ average / worst case
Quick Sort

• Approach
  • Select pivot value (near median of list)
  • Partition elements (into 2 lists) using pivot value
  • Recursively sort both resulting lists
  • Concatenate resulting lists
  • For efficiency pivot needs to partition list evenly

• Performance
  • $O(n \log(n))$ average case
  • $O(n^2)$ worst case

• Used by Arrays.sort

• Runs faster than mergesort in most cases
Quick Sort Algorithm

1. If list below size $K$
   - Sort w/ other algorithm
     - e.g., insertion sort
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
Quick Sort Example

Partition & Sort

Result
Quick Sort Code

```c
void quickSort(int[] a, int x, int y) {
    int pivotIndex;

    if ((y - x) > 0) {
        pivotIndex = partitionList(a, x, y);
        quickSort(a, x, pivotIndex - 1);
        quickSort(a, pivotIndex + 1, y);
    }
}

int partitionList(int[] a, int first, int last) {
    ... // partitions list and returns index of pivot
}
```
Quick Sort Code

int partitionList(int a[], int first, int last) {
    int i, pivot, border;

    pivot = a[first];
    border = first;
    for (i = first + 1; i <= last; i++) {
        if (a[i] <= pivot) {
            border++;
            swap(a, border, i);
        }
    }
    swap(a, first, border);

    return border;
}
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
     - Examine head of both lists
     - Move smaller to end of new list

- **Performance**
  - $O(n \log(n))$ average / worst case

- **Used by Collections.sort**
- External and internal sorting algorithm
- Often preferred for sorting a linked list
- Divide and conquer algorithm
- Uses additional memory to perform merge step (**not in-place**)
- Even though merge sort and quick sort are $O(n \log(n))$ algorithms, quick sort is usually faster in practice and does not require additional memory
Merge Example

2 7

4 5 8

2

7

4 5 8

7

5 8

2 4

© Department of Computer Science UMD
Merge Sort Example

Split

Merge
void mergeSort(int[] a, int x, int y) {
    int mid = (x + y) / 2;

    if (x != y) {
        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 j, size = y - x + 1, left = x, right = mid + 1;
    int[] tmp = new int[a.length];

    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];
}
Radix Sort

- **Does not compare array entries**
- **It is not** suitable as a general-purpose sorting algorithm
- **Approach**
  - Decomposes a key $C$ into its components ($C_1, C_2, \ldots, C_n$) and uses each component to organize to which buckets keys are assigned
    - For an integer we use the digits that make up the integer
    - For a word we use the letters that make up the word
- **Running time $O(n)$, but it cannot sort all types of data**
Radix Sort (Example)

- B(i) - stands for Bucket for digit i
- Original - 122, 397, 220, 017, 512
- Important: Each bucket must retain the order in which it receives values

**Step 1**: Distribute values into buckets according to rightmost digit
- 122, 397, 220, 017, 512
- B(0)\{220\}, B(1)\{}, B(2)\{122, 512\}, B(3)\{}, B(4)\{}, B(5)\{}, B(6)\{}, B(7)\{397, 017\}, B(8)\{}, B(9)\{}
- 220, 122, 512, 397, 017

**Step 2**: Distribute values into buckets according to the next (left) digit
- 220, 122, 512, 397, 017
- B(0)\{}, B(1)\{512, 017\}, B(2)\{220, 122\}, B(3)\{}, B(4)\{}, B(5)\{}, B(6)\{}, B(7)\{}, B(8)\{}, B(9)\{397\}
- 512, 017, 220, 122, 397

**Step 3**: Distribute values into buckets according to the leftmost digit
- 512, 017, 220, 122, 397
- B(0)\{017\}, B(1)\{122\}, B(2)\{220\}, B(3)\{397\}, B(4)\{}, B(5)\{512\}, B(6)\{}, B(7)\{}, B(8)\{}, B(9)\{}
- Sorted \(\rightarrow\) 017, 122, 220, 397, 512
Radix Sort Algorithm for Integer Array

- **radixSort** method
  - Sorts array[first..last] of positive integers in ascending order where maxDigits is the number of digits in the longest integer. Bucket can be a queue.

```java
radixSort(array, first, last, maxDigits) {
    for (i = 0 to maxDigits - 1) {  // d iterations, processing of digits from right to left
        Clear bucket[0], bucket[1], ..., bucket[9]
        for (index = first to last) {  // n iterations, processing all elements of array
            digit = digit i of a[index]
            Place a[index] at end of bucket[digit]
        }
        // Completes a step
        Place contents of bucket[0], bucket[1], ..., bucket[9] into array a
    }
}
```

- For **d** digits for each integer and array with **n** integers: \( O(d \times n) \). As long as **d** is fixed and much smaller than **n** radix sort is \( O(n) \)
- **Radix sort disadvantages**: number of buckets depends on the possible components of a key (e.g., integers 10 buckets, words 26 buckets)
## 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>Insertion</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>
<tr>
<td>Radix</td>
<td></td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>
Links

• Sorting Algorithms Animations
  • https://www.toptal.com/developers/sorting-algorithms/
  • https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

• President and Sorting
  • http://www.youtube.com/watch?v=k4RRi_ntQc8

• Ineffective Sorts
  • http://xkcd.com/1185/

• What different sorting algorithms sound like
  • http://www.youtube.com/watch?v=t8g-iYGHpEA

• Big-O Complexity for Array Sorting Algorithms
  • http://bigocheatsheet.com/