CS61B – Algorithms

Anmol Parande

Spring 2019 - Professor Josh Hug

Contents

Disclaimer: These notes reflect 61B when I took the course (Spring 2019). They may not accurately reflect current course content, so use at your own risk. If you find any typos, errors, etc, please report them on the GitHub repository.

1 Tree Traversals

1.1 Preorder Traversal

In a preorder traversal a node is visited before any of its children are visited.

```
if node == null:
    return
visit(node);
for child in node.children:
    preorder(child)
```

1.2 In Order Traversal

For a Binary Tree, an in order traversal is simply the elements when read from left to right.

```
if node == null
    return
inorder(node.left)
visit(node)
inorder(node.right)
```

1.3 Post Order Traversal

In a postorder traversal, a node is visited after its children are visited.

```
if node == null
    return
for child in node.children:
    postorder(child)
visit(child)
```

1.4 Level Order Traversal

In a level order traversal, the nodes of each level of the tree are visited in order. It is done through a Breadth-First Search of the tree (see below)

2 Graph Traversal

2.1 Breath-First Search

The idea of breadth first search is to visit all children before visiting the children's children. Its runtime is $O(|V| + |E|)$ where V is the set of vertices and E is the set of edges.

```
repeat until queue is empty:
    remove vertex v from queue
    for each unmarked vertex n adjacent to v
        mark n
        keep track of some statistic
        add n to end the queue
```

Breadth-first search is a good algorithm for finding the shortest path between two nodes in an unweighted graph.

2.2 Depth-First Search

The idea of depth-first search is to visit all the children of a child before visiting the nodes other children. The preorder is the order in which DFS calls are made. The postorder is the order of DFS returns. It's runtime is $O(|V| + |E|)$.

2.2.1 Pre-Order

```
add start node to the stack
while stack is not empty:
    v = stack.pop
    mark(v)
    for vertex n adjacent to v:
        mark(n)
        stack.push(n)
```

2.2.2 PostOrder

```
mark v as visited
for each successor v' of v {
    if v' not yet visited:
        DFS(v')
doAction(v)
```

2.2.3 Topological Sort

The problem of a topological sort is given a directed acyclic graph (DAG), find an ordering of vertices that satisfies the directionality of the edges in the graph. I.e, if the edges of a graph indicate that a task must come after a specific task, then find a valid ordering of tasks.

```
while not all visited:
    DFS(v) for arbitrary v
    if not all visited:
        repeat on unmarked vertex
    record DFS post order
reverse DFS post-order
```

Time Complexity: $O(V + E)$
Space Complexity: $\Theta(V)$

3 SPTs

3.1 Dijkstra's Algorithm

Dijkstra's algorithm helps build the shortest paths tree starting from a particular source node.

```
add all vertices to priority queue with infinity priority
set the source's priority to 0
while priority queue is not empty
    remove the smallest vertex v
    mark v
    relax edges from v
```

The relaxation procedure is as follows

```
if w is visited
    return
w = weight(parent) + edge.weight
if w < v.priority
    priority = w
```

Dijkstra's algorithm runs in $O(|E|log|V|)$

3.2 A^* Search

A^* search is like Dijkstra's in that it finds the shortest path between two nodes. However, it is different because the priority of each vertex is the distance to that vertex plus a heuristic value

A^* requires a goal vertex to run. It's runtime is $O(|E|log|V|)$

4 MSTs

A Minimum Spanning Tree (MST) is a tree connecting all vertices in a graph which has the minimum total weight (the sum of its edges is minimized).

4.1 Prim's Algorithm

```
add source vertex to PQ
set all other vertex priorities to infinity
for edge out of V:
    if destination is unmarked:
        relax(V) using the distance to the tree
```

Notice that this is basically Dijkstra's algorithm but we are using distance to tree rather than distance to source. The runtime is $O(|E|log|V|)$.

4.2 Kruskal's Algorithm

```
Make a priority queue of all edges
Create a Disjoint Set of all the vertices
while !pq.empty && mst.size() < V - 1:
    e = pq.pop
    if source not connected to target in DisjointSet
        connect(source, target)
        add e to MST
```

The runtime is also $O(|E|log|V|)$.

5 Sorting

In sorting, an **Inversion** is a pair of elements which is out of order. The goal of sorting is to reduce the number of inversions to 0.

A sort is **stable** if the ordering of equal elements in the sorted array is the same as their ordering in the original array.

All comparison sorts are bounded below by $\Omega(NlogN)$, but sorts which do not rely on comparisons can do better.

5.1 Selection Sort

Selection sort maintains a sorted portion of the array by searching through the array for the minimum element and swapping it to the back of the unsorted part of the array.

```
def selectionSort(arr):
    for (i=0; i < arr.length; i++):
        minIndex = i
        for (j=i+1; j < arr.leng; j++):
            if (arr[j] < arr[i]):
                minIndex = j;
        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
```

Time Complexity: $\Theta(N^2)$
Space Complexity: $\Theta(1)$
Stable?: Yes
Identifying Characteristics: In intermediate steps, the beginning portion of the array will always be in sorted order starting with the smallest values.

5.2 Heap Sort

The essential idea of heap sort is to put the entire array into a max-heap, and then build up the sorted array backwards by popping of the top element.
If we do this naiively (i.e by copying the original array into a new heap and then building a new array, the space complexity is $\Theta(N)$). However, it can be done in place using array heapification.

Best Case Time Complexity: $\Theta(N)$ (All elements are equal)
Worst Case Time Complexity: $\Theta(NlogN)$
Space Complexity: $\Theta(1)$
Stable?: No
Identifying Characteristics: The maximum values are sent to the back of the array during intermediate steps.

5.2.1 Bottom-Up Heapification

Bottom-Up Heapification turns an array into a Max-Heap in place. Simply sink the nodes in reverse level order and you will get a Max Heap. Once this is done, heap sort can proceed mostly as usual.

The runtime of bottom-up heapification is $\Theta(N)$ The worst case will occur when we try to bottom-up heapify a min-heap into a max-heap.

- The bottom level has $\lfloor logN \rfloor$ elements and will sink 0 times.
- The 2nd to last level has $\lfloor logN \rfloor - 1$ elements and will sink 1 time.
- The kth level has $2^{\lfloor logN \rfloor - k}$ elements and will sink k times.

If we count the number of sink operations, we get

$$\sum_{k=0}^{\lfloor logN \rfloor} k2^{log_2 k-1} = 2N - log_2 N - 2 \in \Theta(N)$$

5.3 Merge Sort

Mergesort is a recursive sort which divides the array in two, sorts both halves, and then merges the results.

```
def mergeSort(arr):
    if arr.length = 1:
```

```
        return arr
    mid = arr.length // 2
    leftHalf = mergeSort(arr[:mid])
    rightHalf = mergeSort(arr[mid:])
    i, j, k = 0, 0, 0
    sortedArr = new T[arr.length]
    while (i < leftHalf.length && j < rightHalf.length):
        if (leftHalf[i] <= rightHalf[j]):
            sortedArr[k] = leftHalf[i]
            i++
        else:
            sortedArr[k] = rightHalf[j]
            j++
        k++
    }

    while (i < leftHalf.length):
        sortedArr[k] = leftHalf[i]
        i++
        k++

    while (j < rightHalf.length):
        sortedArr[k] = rightHalf[j]
        j++
        k++
    return sortedArr
```

Time Complexity: $\Theta(NlogN)$
Space Complexity: $\Theta(N)$
Stable?: Yes
Identifying Characteristics: Elements from different halves of the array do not cross during intermediate steps of the algorithm.

5.4 Insertion Sort

Insertion sort maintains a sorted portion in the beginning of the array. Out of place elements are swapped backwards until they arrive in their correct position. Insertion sort performs very well on arrays with few inversions and with few elements.

```
def insertionSort(arr):
    n = arr.length;
    for (int i = 1; i < n; i++) {
        key = arr[i]
        j = i - 1
        while (j >= 0 && arr[j] > key):
            arr[j + 1] = arr[j]
```

```
            j--
        arr[j + 1] = key;
```

Best Case Time Complexity: $\Theta(N)$
Worst Case Time Complexity: $\Theta(N^2)$
Space Complexity: $\Theta(1)$
Stable?: Yes
Identifying Characteristics: A distinct sorted portion exists at the beginning of the array.

5.5 QuickSort

QuickSort works based on the idea of partionining. To partition an array based on a particular pivot element, reorder the array so everything left of the pivot is smaller and everything to the right is larger or equal. Essentially, if we partition the array, and then QuickSort both halves, we will arrive at a sorted array.

```
def quicksort(arr):
    pivotIndex = choosePivot(arr)
    arr = partition(arr, pivotIndex)
    leftHalf = quicksort(arr[:pivotIndex])
    rightHalf = quicksort(arr[pivotIndex+1:])
    return leftHalf + rightHalf
```

5.5.1 3 Scan Partitioning

```
pivot = arr[0]
partitionedArr = new array
copy elements < pivot into partitionedArr
copy elements = pivot into partitionedArr
copy elements > pivot into partitionedArr
return partitionedArr
```

5.5.2 Hoare Partitioning

```
left,right = 0, arr.length - 1
while (left <= right):
    while arr[left] < pivot:
        left++
    while arr[right] > pivot:
        right++
    swap(arr[left], arr[right])
    left++
    right--
swap(pivot, arr[right])
```

Best Case Time Complexity: $\Theta(NlogN)$
Worst Case Time Complexity: $\Theta(N^2)$
Space Complexity: $\Theta(1)$
Stable?: No
Identifying Characteristics: During intermediary steps, everything left of where the first element in the unsorted array ends up is less than the pivot and everything to the right is larger or equal.
Note: The properties of quicksort depends on the partioning scheme, pivot strategy, and whether or not randomization was used.

5.6 Counting Sort

Counting sort is a non-comparison sort which requires a finitely sized alphabet. It merely counts the occurence of each item and then copies them into the proper locations in the sorted array.

```
def countingSort(arr):
    max = maxElement(arr)
    int n= arr.length
    int B[]=new int[n]
    int C[]=new int[max+1]

    for (int i=0; i <=max; i++)
        C[i] = 0

    for (int j=0; j<n; j++):
        C[arr[j]] = C[arr[j]] + 1

    for (int i=1; i<max+1; i++):
        C[i] = C[i] + C[i-1]

    for (int j=n-1; j>=0; j--):
        B[C[arr[j]]-1] = arr[j]
        C[arr[j]] = C[arr[j]] - 1
```

If N is the number of elements in the array and R is the size of the alphabet
Time Complexity: $\Theta(N+R)$
Space Complexity: $\Theta(N+R)$
Stable?: Yes
Note: As long as $N \geq R$ then counting sort will be very fast.

5.7 LSD Radix Sort

Least Significant Digit Radix sort uses counting sort to sort elements comprised of a sequency of a finite alphabet (like Strings or Integers). The idea is to apply counting sort for each digit starting with the least significant digit and working leftwards.

```
for (i=0; i < longestElement.length; i++):
```

```
        counting sort based on ith digit
```

If N is the number of elements, W is the length of the longest word, and R is the size of the alphabet.
Time Complexity: $\Theta(WN + WR)$
Space Complexity: $\Theta(N + R)$
Stable?: Yes
Identifying Characteristics: The last few digits of the array elements are sorted.

5.8 MSD Radix Sort

Most Significant Radix Sort is like LSD Radix Sort except it starts with the most significant digit and works its way rightwards. However, if we proceed the same way as we did LSD radix sort, our sort will not be accurate. For MSD radix sort, it is necessary to only apply counting sort to each subproblem after we sort each subsequent digit.

```
subproblems = [arr]
for (i=longestElement.length-1; i >= 0; i--):
    for each problem in subproblems:
        counting sort problem on ith digit
        add new subproblem to array
combine sorted subproblems to get sorted array
```

Best Case Time Complexity: $\Theta(N + R)$
Worst Case Time Complexity: $\Theta(WN + WR)$
Space Complexity: $\Theta(N + R)$
Stable?: Yes
Identifying Characteristics: Intermediary steps all have the front few digits in sorted order.
Notes: The best case time complexity comes about when each element is distinguished only by the most significant digit.

5.9 Runtime Table

Sort	Space Complexity	Best Case Time Complexity	Worst Case Time Complexity	Stable
Insertion	$\Theta(1)$	$\Theta(N)$	$\Theta(N^2)$	Yes
Selection	$\Theta(1)$	$\Theta(N^2)$	$\Theta(N^2)$	No
Heap	$\Theta(1)$	$\Theta(N)$	$\Theta(NLogN)$	No
Merge	$\Theta(N)$	$\Theta(NlogN)$	$\Theta(NLogN)$	Yes
Quick	$\Theta(1)$	$\Theta(NlogN)$	$\Theta(N^2)$	No
Counting	$\Theta(N + R)$	$\Theta(N + R)$	$\Theta(N + R)$	Yes
LSD	$\Theta(N + R)$	$\Theta(WN + WR)$	$\Theta(WN + WR)$	Yes
MSD	$\Theta(N + R)$	$\Theta(N + R)$	$\Theta(WN + WR)$	Yes

6 Compression

The central idea behind compression is to take a bunch of data and figure out how to compactly represent it. One way to do this is to assign a specific codeword to a symbol in the document we want to compress (i.e each letter in the alphabet or each byte in an image). In **Prefix-Free Codes**, none of our codewords is a prefix for the others (i.e 01 and 010 can't both be codewords because 01 prefixes 010). This avoid ambiguity in decoding.

6.1 Huffman Encoding

Huffman encoding is a special way of constructing a prefix-free code. The pseudo-code below describes Huffman Encoding.

```
Calculate the relative frequency of each symbol
Assign each symbol to a node with weight = relative frequency
Repeat until all nodes belong to a tree:
    p, q = 2 smallest nodes
    Make supernode of pq with weight p + q
```

The resulting tree which Huffman Encoding builds is the encoding tree. It is a Trie where each link represents an element of the alphabet we are using to create our codewords. Each node appears at the end of a codeword.

To encode a document, simply convert each symbol into its codeword by looking it up in the Trie. To decode a document, find the longest matching prefix of the decoding text, retreive the corresponding symbol, and repeat with the rest of the document.