Outline

• Sorting
• Tree
Outline

• Sorting
• Tree
Sorting Algorithms

• We now switch gears and discuss some well known sorting algorithms
• We only talk about comparison based sorting
Selection Sort

- Find the smallest item in the unsorted portion, and place it in front.
- What is the running time (complexity) of this algorithm?
## Time complexity

<table>
<thead>
<tr>
<th>Performance</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Worst-case performance</strong></td>
<td>$O(n^2)$ comparisons, $O(n)$ swaps</td>
</tr>
<tr>
<td><strong>Best-case performance</strong></td>
<td>$O(n^2)$ comparisons, $O(n)$ swaps</td>
</tr>
<tr>
<td><strong>Average performance</strong></td>
<td>$O(n^2)$ comparisons, $O(n)$ swaps</td>
</tr>
</tbody>
</table>
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the smallest element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
Insertion Sort

- Pick one from the unsorted part, and place it in the “right” position in the sorted part.
- Best case?
- Avg. case?
- Worst case?
Insertion Sort

- Pick one from the unsorted part, and place it in the “right” position in the sorted part.
  - Best case? $O(n)$
  - Avg. case? $O(n^2)$
  - Worst case? $O(n^2)$
Time complexity

<table>
<thead>
<tr>
<th><strong>Worst-case performance</strong></th>
<th>( \mathcal{O}(n^2) ) comparisons, ( \mathcal{O}(n^2) ) swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best-case performance</strong></td>
<td>( \mathcal{O}(n) ) comparisons, ( \mathcal{O}(1) ) swaps</td>
</tr>
<tr>
<td><strong>Average performance</strong></td>
<td>( \mathcal{O}(n^2) ) comparisons, ( \mathcal{O}(n^2) ) swaps</td>
</tr>
</tbody>
</table>
/**
 * Simple insertion sort.
 */

template <typename Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = a[ p ];
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
Bubble Sort

• **Simple** and uncomplicated
• Compare neighboring elements
• **Swap** if out of order
• Two nested loops
• $O(n^2)$
Bubble Sort Example

2, 3, 1, 15
2, 1, 3, 15  // after one loop
1, 2, 3, 15  // after second loop
1, 2, 3, 15  // after third loop
Bubble Sort

vector a contains n elements to be sorted.

for (i=0; i<n-1; i++) {
    for (j=0; j<n-1-i; j++)
        if (a[j+1] < a[j]) {
            /* compare neighbors */
            tmp = a[j];       /* swap a[j] and a[j+1] */
            a[j] = a[j+1];
            a[j+1] = tmp;
        }
}
## Time complexity

<table>
<thead>
<tr>
<th>Performance Type</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst-case performance</td>
<td>(O(n^2)) comparisons, (O(n^2)) swaps</td>
</tr>
<tr>
<td>Best-case performance</td>
<td>(O(n)) comparisons, (O(1)) swaps</td>
</tr>
<tr>
<td>Average performance</td>
<td>(O(n^2)) comparisons, (O(n^2)) swaps</td>
</tr>
</tbody>
</table>
Merge Sort

3 7 6 5
3 7 5 6
8 2 1 4
2 8 1 4
3 5 6 7
1 2 4 8
1 2 3 4 5 6 7 8
Merge Sort: Running Time?

\[ O(n)O(\log n) = O(n \log n) \]
Time complexity

<table>
<thead>
<tr>
<th>Performance Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst-case performance</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Best-case performance</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Average performance</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
General Sorting: Running Time

- $O(n \log n)$ is faster than $O(n^2)$ – merge sort is more efficient than selection sort or insertion sort.
- $O(n \log n)$ is the best average complexity that a general (comparison) sorting algorithm can get (assuming you know nothing about the data set).
- With more information about the data set provided, you can sometimes sort things almost linearly.
Quick Sort

• Fastest known sorting algorithm in practice

• Average case complexity $\rightarrow O(N \log N )$

• Worst-case complexity $\rightarrow O(N^2)$
  • Rarely happens, if implemented well
Quick Sort

- Pick a **pivot**, and move numbers that are less than the pivot to front, and ones that are greater than the pivot to end. (Does this sound familiar?)
- On average, $O(n \log n)$
- Depending on how you pick your pivots, it can be as bad as $O(n^2)$
Picking the Pivot

• How would you pick one?
• Strategy 1: Pick the first element in $S$
  • Works only if input is random
  • What if input $S$ is sorted, or even mostly sorted?
    • All the remaining elements would go into either $S1$ or $S2$!
    • Terrible performance!
Picking the Pivot (contd.)

• Strategy 2: Pick the pivot *randomly*
  
  • Would usually work well, even for mostly sorted input
  
  • Unless the random number generator is not quite random!
  
  • Plus random number generation is an expensive operation
Picking the Pivot (contd.)

• Strategy 3: Median-of-three Partitioning
  • Ideally, the pivot should be the median of input array $S$
    • Median = element in the middle of the sorted sequence
  • Would divide the input into two almost equal partitions
  • Unfortunately, its hard to calculate median quickly, even though it can be done in $O(N)$ time!
  • So, find the approximate median
    • Pivot = median of the left-most, right-most and center element of the array $S$
    • Solves the problem of sorted input
**Time complexity**

<table>
<thead>
<tr>
<th>Performance Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst-case performance</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Best-case performance</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Average performance</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this:
2 5 1 7 9 12 11 10
Which statement is correct?

(A) The pivot could be either the 7 or the 9.
(B) The pivot could be the 7, but it is not the 9
(C) The pivot is not the 7, but it could be the 9
(D) Neither the 7 nor the 9 is the pivot.
void permutation(vector<int>& nums, int start) {
    if (start == nums.size() - 1) {
        for (int i = 0; i < nums.size(); ++i) {
            cout << nums[i] << ',';  // Output the current element
        }
        cout << endl;
    }
    permutation(nums, start + 1);
    for (int i = start + 1; i < nums.size(); ++i) {
        swap(nums[start], nums[i]);
        permutation(nums, start + 1);
        swap(nums[start], nums[i]);
    }
}
permutation(nums, 0);  // Call this function
Bound for Comparison Based Sorting

- $O(n \log n)$
  - Optimal bound for comparison-based sorting algorithms
  - Achieved by Quick Sort, Merge Sort, and Heap Sort
Which of the following sorting algorithms has the best worst-case complexity?

(A) Merge Sort
(B) Bubble Sort
(C) Quick Sort
(D) Selection Sort
Stability in sorting algorithms

- A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

Sorting is stable because the order of balls is maintained when values are same. The ball with green color and value 10 appears before the orange color ball with value 10. Similarly order is maintained for 20.
Which sorting algorithms are stable?

• Some Sorting Algorithms are stable by nature, such as Bubble Sort, Insertion Sort, Merge Sort, etc.
Which sorting algorithms are unstable?

• Quick Sort
• Heap Sort
Can we make any sorting algorithm stable?

• **Yes.** Any given sorting algorithm which is not stable can be modified to be stable.

• **How?**

• By changing the key comparison operation so that the comparison of two keys considers position as a factor for objects with equal keys.
Reference

• https://en.wikipedia.org/wiki/Comparison_sort
Outline

• Sorting
• Tree
Tree: Definition
Tree: Definition
Bound on # of edges

How many edges should there be in a tree of \( n \) nodes?
Binary Tree

No node has more than 2 children (left child + right child).
Binary Tree

How many nodes can a binary tree of height $h$ have? (one with max. # of nodes == full binary tree)
Tree is a Data Structure

- For every data structure we need to know:
  - how to insert a node,
  - how to remove a node,
  - search for a node

- and (for tree only)
  - how to traverse the tree

```c
struct Node
{
    ItemType val;
    Node* left;
    Node* right;
};
```
Three Methods of Traversal

```cpp
void preorder(const Node *node) {
    if (node == NULL) return;
    cout << node->val << " ";
    preorder(node->left);
    preorder(node->right);
}

void inorder(const Node *node) {
    if (node == NULL) return;
    inorder(node->left);
    cout << node->val << " ";
    inorder(node->right);
}

void postorder(const Node *node) {
    if (node == NULL) return;
    postorder(node->left);
    postorder(node->right);
    cout << node->val << " ";
}
```
void level_order(const Node *root) {
    queue<Node*> q;
    q.push(root);
    while(!q.empty()) {
        Node *cur = q.first();
        q.pop();
        cout << cur->val;
        if (cur->left) q.push(left);
        if (cur->right) q.push(right);
    }
}
Traversal Orders

Pre-order DFS

In-order DFS

Post-order DFS

Level-order BFS
See you next week!