Sorting 2

- Any comparison based sorting requires Ω(NlogN) comparisons

A General Lower Bound For Sorting

- The root represents the set of all possible orderings: when the sorting algorithm is given a array to sort, any possible ordering is possible!
- Each node performs one comparison, which patricians its set of orderings into two sets based on the comparison results. The two children are in each of its set, respectively.
- The algorithm needs to perform enough comparison to get a set that contains one ordering (the sorting result).
- **Each comparison based sorting algorithm can be represented by a binary decision tree.**
- The worst case is the largest set of comparisons possible needed.
  - This is the height of the decision tree
- Different algorithms differ in items selected for comparison at each node.
- The most efficient algorithm is the one with the smallest height.
- We know the number of leaves in the tree
  - The number of possible sorted orders of N items, let us denote it as X
- To get the minimum tree height of the decision tree, let us decide the number of leaves on the tree
  - Number of leaves = number of orderings N numbers
  - $N! = N*(N-1)*(N-2)*...2*1$
- *For a binary tree to have N! leaves, the tree is at least...*
  - $log(N!) = Log(N) + log(N-1) + ... + 1$
  - ≥ $log(N) + log(N-1) + ... + Log(\frac{N}{2})$
  - ≥ $log(\frac{N}{2}) + log(\frac{N}{2}) + ... + Log(\frac{N}{2})$
  - ≥ $\frac{N}{2} * Log(\frac{N}{2}) = \Omega(NlogN)$
- *No Comparison Based Sorting can be better than N(Log(N))*
- Heapsort, Mergesort, and Quicksort - *N(Log(N))*

Heap Sort

- Build binary minheap of N elements
  - O(N)
- The perform N deletemin operations
  - Log(N) time per deletemin
- *N(Log(N))*
- Requires another array to store results
- To eliminate this requirement
  - using heap to store sorted elements
  - using maxheap instead

```C++
void heapify(int arr[], int N, int i)
{

    // Initialize largest as root
    int largest = i;

    // left = 2*i + 1
```

```
    int l = 2 * i + 1;

    // right = 2*i + 2
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest
    // so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int N)
{

    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element
    // from heap
    for (int i = N - 1; i > 0; i--) {

        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// A utility function to print array of size n
void printArray(int arr[], int N)
{
    for (int i = 0; i < N; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}
```

Merge Sort

- Divide N values to be sorted into two halves

- Recursively sort each half using merge sort
  - Base case N = 1, no sorting required
- Merge two halves into one list
- Keep a counter for each list starting at the start of each list
- Compare the two values indexed by the counters, output the smaller value and increment the counter
- when one list is processed output all items in the other list
  - 4, 1, 9, 4
  - 2, 5, 6, 8
  - Compare 4 and 2 and output 2

Complexity

- T(N) complexity when size N
- Merge O(N)
- Complexity O(NLogN)

Quick Sort

- Fastest sorting algorithm in practice
  - *Caveat*: not always stable
  - Can do it as a stable sort
- Average Complexity: $O(NLog(N))$
- Worst Complexity: $O(N^2)$
  - Rare
- Can vary in space complexity

Sorting 3 For More....
