I Recursion

- **Tail recursion** uses less memory than non-tail recursion.

I.1 Tail Recursion

I.1.1 Cost of recursion

- Each call to a function adds another frame on the stack
- Each frame contains local variables and parameters and where to return the result

I.1.2 Reducing what needs to be stored

- If we can guarantee we won't need them, we can free the memory for the local variables and parameters.
- We won't need them as long as **we do not use them after the recursive call**.
- Tail recursive functions have the **recursive call as the last thing the function does before it returns**.

II Complexity

II.1 Master Theorem

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$$T(x) = \begin{cases} \Theta(n^d) \text{ if } a<b^d \\ \Theta(n^d \log(n)) \text{ if } a=b^d \\ \Theta\left(n^{\log_b(n)}\right) \text{ if } a>b^d \end{cases}$$

a: number of subproblems in the recursion
n/b: size of each subproblem
d: the exponent in the cost of the work done outside the recursive calls, specifically in the non-recursive part of the algorithm (like splitting or merging the problem)

e.g. **Binary Search** a=1 b=2 d=0 $\log(n)$ **Merge Sort** a=2 b=2 d=1 $n\log(n)$

II.2 Bound

II.2.1 Big O - Upper Bound

$f(n) = O(g(n))$ iff (if and only if)
$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 : f(n) \leq cg(n)$

II.2.2 Big Ω - Lower Bound

$f(n) = \Omega(g(n))$ iff (if and only if)
$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 : f(n) \geq cg(n)$

II.2.3 Big Θ - Tight Bound

$f(n) = \Theta(g(n))$ iff (if and only if)
$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$

II.2.4 Little o upper Bound

$\exists c \in \mathbb{R}^*, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 \text{ such that } f(n) < cg(n)\left(\left|\frac{f(n)}{g(n)}\right| < c\right)$
In other words: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

III Sorting

III.1 Selection

不断从未排序部分中选择最小（或最大）元素，并将其与第一个未排序元素交换，每次迭代后将排序边界向前移动一步 repeatedly selects the smallest (or largest) element from the unsorted portion of an array and swaps it with the first unsorted element, moving the sorted boundary one step forward with each iteration

```
for (int i = 0; i < array.size(); i++) {
  // Find min element from i to n-1
  for (int j = i + 1; i < array.size(); j++) {
  }
  // Swap elements at index i and min elements
}
```

III.2 Insertion

逐一将未排序部分的下一个元素插入到（不断交换）已排序部分的正确位置，逐步构建出一个已排序的数组 builds the sorted array one element at a time by repeatedly taking the next unsorted element and inserting it into its correct position in the sorted part

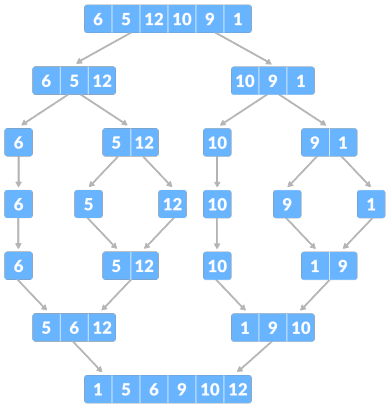
```
for (int i = 1; i < array.size(); i++) {
  for (int j = i; j > 0; j--) {
    if (array.at(j) < array.at(j - 1)) {
      swap(array.at(j - 1), array.at(j));
    } else break;
  }
}
```

III.3 Bubble

repeatedly compares adjacent elements and swaps them if they are in the wrong order, causing larger elements to "bubble" to the end of the array. With each pass through the list, the largest (or smallest) element settles in its correct position, and this process continues until the entire array is sorted.

```
for (int i = array.size() - 1; i > 0; i--) {
    for (int j = 0; j < i; j++) {
      if (array.at(j) > array.at(j + 1)) {
        swap(array.at(j), array.at(j + 1));
      }
    }
  }
```

III.4 Merge




```
if (array.size() == 1) return array;
// Sort left and right subarrays
int mid = array.size() / 2;
// ... Define left and right arrays
vector<int> sortedRightArray = sort(rightArray);
vector<int> sortedLeftArray = sort(leftArray);
// Merge left and right subarrays
vector<int> result;
int l = 0, r = 0;
while (l < sortedLeftArray.size() && r <
sortedRightArray.size()) {
  if (sortedLeftArray.at(l) <
sortedRightArray.at(r)) {
    result.push_back(sortedLeftArray.at(l)); l++;
  } else {
    result.push_back(sortedRightArray.at(r)); r+
+;
  }
}
// ... Add remaining elements form
sortedLeftArray or sortedRightArray
return result;
```

III.5 Quick

```
if (start >= end) return;
// Select the last element as pivot
int pivot = array.at(end);
int pivotIndex = start;
for (int i = start; i < end; i++) {
  if (array.at(i) < pivot) {
    swap(array.at(i), array.at(pivotIndex));
    pivotIndex++;
  }
}
swap(array.at(pivotIndex), pivot);
sort(array, start, pivotIndex - 1);
sort(array, pivotIndex + 1, end);
```

	Best	Average	Worst	Space	Stable
Selection	$O(n^2)$			$O(1)$	
Insertion	$O(n)$	$O(n^2)$		$O(1)$	✓
Bubble	$O(n^2)$			$O(1)$	✓
Merge	$O(n\log(n))$			$O(n)$	✓
Quick	$O(n\log(n))$		$O(n^2)$	$O(\log(n))$	
Bucket	$O(n)$	$O(n+k)$ (k: numbers of buckets)	$O(n^2)$	$O(n+k)$	✓
Counting	$O(n+k)$ (k: range of the input values, max-min)			$O(n+k)$	✓
Heap	$O(n\log(n))$			$O(1)$	

	Worst			Average			Space
	Insert	**Delete**	**Search**	**Insert**	**Delete**	**Search**	
Vector Ordered	$O(n)$		$O(\log(n))$	$O(n)$		$O(\log(n))$	$O(n)$
Vector Unordered	$O(1)$*	$O(n)$		$O(1)$*	$O(n)$		
Linked List	$O(1)$			$O(1)$			
Binary Search Tree	$O(n)$			$O(\log(n))$			
Balanced BST (RBT)	$O(\log(n))$						
Priority Queues	**Insert**	**RMH****	**Peek**	**Insert**	**RMH****	**Peek**	
Linked List	$O(n)$	$O(1)$		$O(n)$	$O(1)$		
Heap	$O(\log(n))$		$O(1)$	$O(1)$	$O(\log(n))$	$O(1)$	

*: Amortised - ie over a sequence of this operation. Resizing vector $O(n)$
**: Remove highest priority