Java 2 Efficiency

- Premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.
- Efficiency - Skillfulness in avoiding wasted time and effort.

VERSION A

```
public boolean search(List<T> a, T target) {
	boolean found - false;
	for (T element : a) {
		if (element.equals(target))
			found = true;
	}
	return found;
}
```

VERSION B (This exits upon finding it)

```
public boolean search(List<T> a, T target) {
	for (T element : a) {
		if (element.equals(target))
			return true;
	}
	return false;
}
```

- Version B is not always more efficient
- In worst case, Version A and B are identical.
- On average, Version B is more efficient.
- We want the, “Chuck Norris” Case, where both worst case and average case are reduced significantly.

- Tune/tweak the algorithm
  - Generally marginal change in efficiency
- Apply a heuristic
  - “A rule of thumb”
  - EX: On each successful search, move the target to the first position in the list.
  - When you have a “working set” of search targets that follows the 90-10 rule. That is you are searching for 10% of the elements 90% of the time.
  - **Average** time of 10,000 runs with N = 100,000: 0.85ms
  - Neither techniques improve worst case.

```
public boolean search(List<T> a, T target) {
	for (T element : a) {
		if (element.equals(target)) {
			a.remove(element)
			a.add(0, element)
			return true;
		}
	}
	return false;
}
```

- Improving worst case efficiency often involves a more fundamental change to the algorithm or to the assumptions/constraints on the problem.
- Current list search constraints: A list of elements that are comparable to each other, and **arranged in non-decreasing order.** (Which, is like cheating, but we can use a BST instead of a linear search because they are always in order.)
- Which, isn’t always more efficient, but is overall more efficient.

Comparing Efficiency

- A linear search scans each element one by one, and eliminates only one element per comparison. So, for an array of size N, there will be N comparisons. If the search space is doubled we need N more comparisons.
- A binary search eliminates half of the elements remaining per comparisons. So, for an array of size N, there will be $\log_2 N$ comparisons. If the search space is doubled we need at worst 1 more comparison.

Categorizing Running Time

- **SCALABILITY SON**
- We are interested in the function T(N) based on the time it takes as the problem size increases. Where T is time and N is problem size.
- T(N) is a time complexity function.
- We basically want an algorithm that doesn't change run-time as problem size grows.