Lecture 2 — Sorting

CITS3001 Advanced Algorithms

Department of Computer Science and Software Engineering
University of Western Australia

Overview

Sorting Recap

- Insertion Sort (CLRS §2.1)
- Merge Sort (CLRS §2.3)
- Heapsort (CLRS §6)

Comparison-Based Sorting (CLRS §8.1)

- Decision Trees
- Lower Bound

Sorting

7	1	3	0	4	6	1	4

$$\Downarrow$$

0	1	1	3	4	4	6	7

- Computer memory is linear
- Elements in a list could be in arbitrary order
  - Knowing any one item tells us nothing about the rest
- Elements could instead be arranged in sorted order
  - Knowing any one item tells us that everything before it must not be greater than it, and everything after must not be less
- Sorting algorithms put a list into sorted order

Insertion Sort

Insertion sort is a simple sorting algorithm based on the following observations:

- Any list of fewer than two elements is trivially already sorted
- Given any sorted list, we can insert a new element while maintaining order
- By induction, we can start with the empty list and introduce each element from the unsorted list one at a time until all have been inserted, giving the sorted list
- We can do this *in place* by growing a sorted prefix of the list such that any element not in the prefix still needs inserting

Insertion Sort

7	1	3	0	4	6	1	4
1	7	3	0	4	6	1	4
1	3	7	0	4	6	1	4
0	1	3	7	4	6	1	4
0	1	3	4	7	6	1	4
0	1	3	4	6	7	1	4
0	1	1	3	4	6	7	4
0	1	1	3	4	4	6	7

Insertion Sort

```python
def insertion_sort(xs: list) -> list:
    # Iterate through prefix lengths
    for l in range(1, len(xs)):
        # Insert xs[l] into sorted prefix xs[0:l]
        for i in range(l, 0, -1):
            # xs[i] is element being inserted
            if xs[i] < xs[i-1]:
                # Wrong way around, swap them
                xs[i-1], xs[i] = xs[i], xs[i-1]
            else:
                # xs[i] is in the right spot
                break
        # xs[0:l+1] is now sorted
    return xs
```

Insertion Sort

Insertion sort is $O(N^2)$:

- Inserting an element into a list of length n is $O(n)$:
  - In the worst case we have to move all n elements over by one, $O(n)$
  - On average we will need to move $n/2$ elements, $n/2 \in O(n)$
- Insertion sort does one insertion for each prefix length $n \in [1, N)$
- $1 + 2 + \cdots + (N-1) = N(N-1)/2 = (N^2 - N)/2 \in O(N^2)$

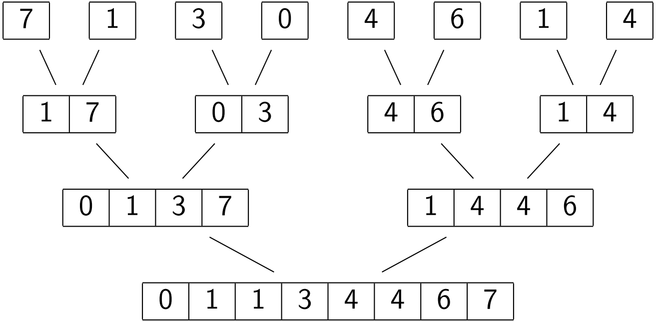
If we get lucky and the list is already sorted or mostly sorted, each insertion might be as quick as $O(1)$, giving just $O(N)$ overall, but this is extremely unlikely for random inputs.

Merge Sort

Merge sort is based on merging sorted lists instead of inserting just a single element at a time:

- Any list of fewer than two elements is trivially already sorted
- Any pair of sorted lists can be merged to give the sorted list of all the elements
- If we have a sorting algorithm that works for lists of length n, then we can sort a list of length $2n$ by cutting it in half, sorting the length n halves, and merging them
- By induction, we can use this strategy to recursively to sort a list of any length

Merge Sort



Merge Sort

```python
def merge(lhs: list, rhs: list) -> list:
    result = []
    # lhs[li] and rhs[ri] are minimum elements not yet in result
    li, ri = 0, 0
    while li < len(lhs) and ri < len(rhs):
        # Append the lesser element
        if lhs[li] <= rhs[ri]:
            result.append(lhs[li])
            li += 1
        else:
            result.append(rhs[ri])
            ri += 1
    # Append any leftovers
    result.extend(lhs[li:] + rhs[ri:])
    return result
```

Merge Sort

```python
def merge_sort(xs: list) -> list:
    # Trivially sorted
    if len(xs) <= 1:
        return xs
    # Cut the list into halves and recursively sort
    mid = len(xs) // 2
    lhs = merge_sort(xs[:mid])
    rhs = merge_sort(xs[mid:])
    # Merge halves back together
    return merge(lhs, rhs)
```

Merge Sort

Merge sort is $O(N \lg N)$:

- The number of recursive layers is $O(\lg N)$
- Therefore each element in the list will take part in $O(\lg N)$ merges
- Each element in a merge takes constant time to be appended to the merged list
- Therefore all N elements take $O(\lg N)$ operations to be merged into the final result, giving $O(N \lg N)$ overall

Did you know?

Python uses an advanced variant of merge sort called Timsort

Heapsort

We will not need to cover heapsort in detail, but recall:

- Using a priority queue, we can sort a list by simply putting everything in the queue and then taking them out in sorted order
- We can implement a priority queue using a *heap* data structure
- A (*leftist*, *binary*) heap can be constructed in $O(N)$ and stored efficiently in a list
- The highest priority element of the heap can be removed in $O(\lg N)$
- Therefore by turning our original list into a heap and pulling values out of it one at a time we can construct the sorted list in $O(N \lg N)$

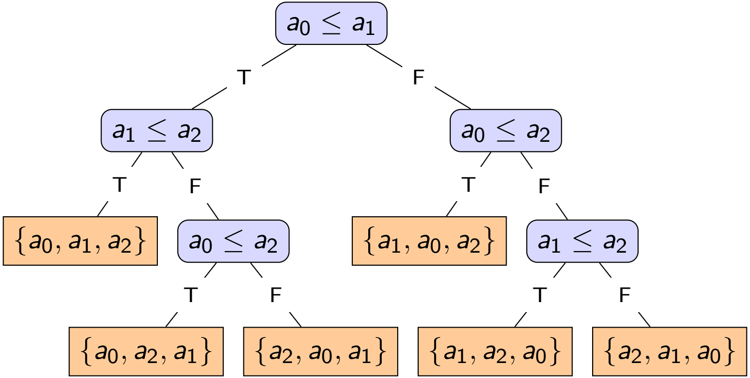
Mid-Lecture Break

Comparison-Based Sorting

- All of the sorting algorithms we have seen so far determine the order elements are meant to be in only by comparing them using relations like $<$, $\leq$, $\geq$, $>$, and $=$
- This is called *comparison-based sorting*
- The fastest sorting algorithms we have seen take $O(N \lg N)$ time
- We would like to find a faster sorting algorithm, if one exists
- Alternatively we would like to prove there is no faster comparison-based sorting algorithm

Decision Trees

We can draw a binary *decision tree* of any sorting algorithm, such as this decision tree for insertion sort with three elements:

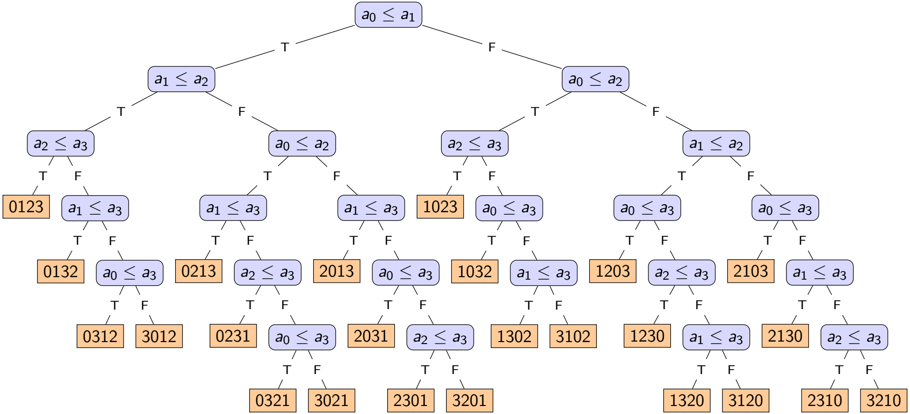


Decision Trees

- The number of permutations of a list of N elements is $N!$
- Therefore the decision tree must have $N!$ leaves
- The worst case for the algorithm is the height of the decision tree

Decision Trees

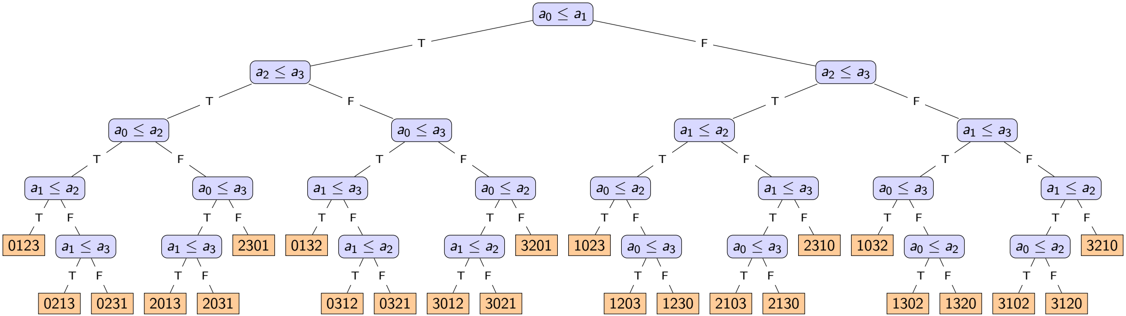
The decision tree for insertion sort is very unbalanced and has $O(N^2)$ height:



Note that this is the previous tree with all leaves expanded into insertion processes.

Decision Trees

Whereas the decision tree for merge sort is well balanced for $4! = 24$ leaves:



Note that this is *not* the same tree structure as merge sort itself.

Lower Bound on Comparison-Based Sorting

- A binary tree of height h may have at most 2^h leaves
- Therefore any binary tree with $N!$ leaves must be at least $\lg N!$ in height
- So comparison-based sorting must perform at least $\lg N!$ comparisons in the worst case
- But how does $O(\lg N!)$ compare to $O(N \lg N)$?

Lower Bound on Comparison-Based Sorting

Theorem

$O(\lg N!) \subseteq O(N \lg N)$

Proof.

In the limit as $N \to \infty$:

$$N! < N^N$$
$$\lg N! < \lg N^N$$
$$\lg N! < N \lg N$$

Therefore $O(\lg N!) \subseteq O(N \lg N)$, as desired. $\square$

Recall $O(f(N))$ is the *set* of functions *bounded above* by $f(N)$.

Lower Bound on Comparison-Based Sorting

Theorem

$O(\lg N!) \supseteq O(N \lg N)$

Proof.

In the limit as $N \to \infty$:

$$
\begin{aligned}
N!^2 &= 1N \times 2(N-1) \times 3(N-2) \cdots \times (N-1)2 \times N1 \\
N!^2 &> N^N \\
\lg N!^2 &> \lg N^N \\
2 \lg N! &> N \lg N
\end{aligned}
$$

Therefore $O(\lg N!) \supseteq O(N \lg N)$, as desired. $\square$

Lower Bound on Comparison-Based Sorting

Theorem

$O(\lg N!) = O(N \lg N)$

Proof.

From above we have

$$O(\lg N!) \subseteq O(N \lg N)$$
$$O(\lg N!) \supseteq O(N \lg N)$$

which both hold only if $O(\lg N!) = O(N \lg N)$, as desired. $\square$

Corollary

$\lg N! \in \Omega(N \lg N)$

Recall $\Omega(f(N))$ is the set of functions bounded below by $f(N)$.

Lower Bound on Comparison-Based Sorting

- Determining the correct permutation requires at least $\lg N! \in \Omega(N \lg N)$ comparisons in the worst case
- Any comparison-based sorting strategy that uses less than $O(N \lg N)$ comparisons therefore can not be correct for all inputs
- We have comparison-based sorting algorithms with a worst-case complexity of $O(N \lg N)$
- An optimal comparison-based sorting algorithm will therefore require $\Theta(N \lg N)$ time in the worse case
- Merge sort and heapsort have optimal asymptotic complexities for comparison-based sorting

Recall $\Theta(f(N))$ is the set of functions bounded above *and* below by $f(N)$.