Lecture 4

Sorting algorithms

Plan for today

1. Applications of sorting
2. Insertion sort
3. Merge sort
4. Quicksort
5. Heaps and heapsort
6. Lower bounds
7. Bucketsort

Applications

Application I: Closest pair

- Given n numbers, find the pair which are closest to each other.
- Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an O(n) linear scan completes the job.

Application II: Element uniqueness

- Given a set of n items, are they all unique or are there any duplicates?

- Sort them and do a linear scan to check all adjacent pairs. This is a special case of closest pair above.

Application III: Mode

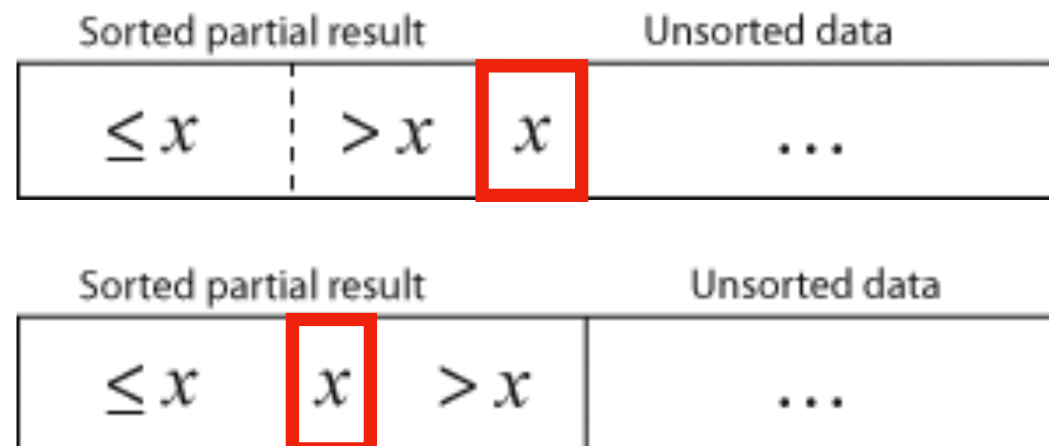
- Given a set of n items, which element occurs the largest number of times? More generally, compute the frequency distribution.
- Sort them and do a linear scan to measure the length of all adjacent runs. The number of instances of k in a sorted array can be found in $O(\log n)$ time by using binary search to look for the positions of both $k - \varepsilon$ and $k + \varepsilon$.

Application IV: Selection

- What is the k-th largest item in the array?
- Once the keys are placed in sorted order in an array, the k-th largest can be found in constant time by simply looking in the k-th position of the array.
- There is a linear time algorithm for this problem, but the idea comes from partial sorting.

Insertion sort

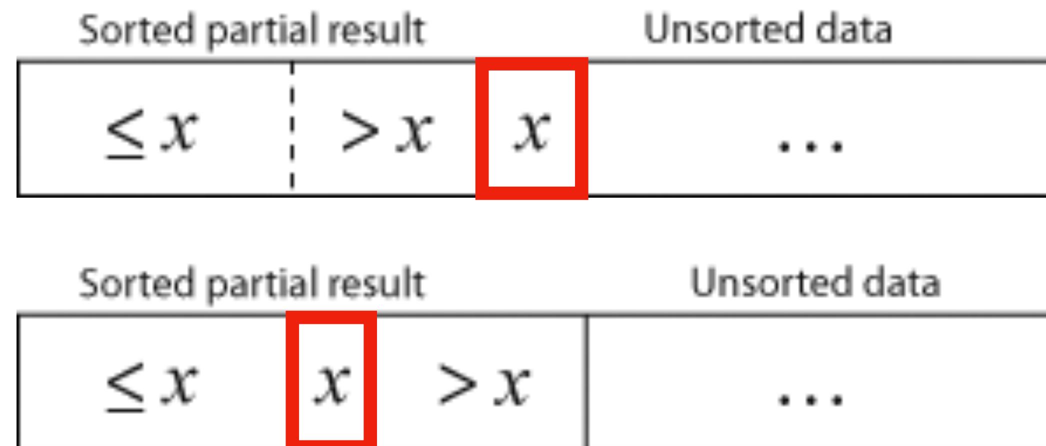
Insertion sort



Animation

```
InsertionSort(A)
1 i ← 1
2 while i < length(A) do
3    j ← i
4    while j > 0 and A[j-1] > A[j] do
5        swap A[j] and A[j-1]
6        j ← j - 1
7    endWhile
8    i ← i + 1
9 endWhile
```

Insertion sort



Lemma: Insertion sort outputs a sorted array.

Theorem: Insertion sort takes $\Theta(n^2)$ time.

Merge sort

Merge sort

- Recursive algorithms are based on reducing large problems into small ones.
- A nice recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively, and then interleaving the two sorted lists to totally order the elements.

Animation

Merge sort

```
MergeSort(L, low, high)
1 if (low < high) then do
2     middle = (low+high)/2
3     mergesort(L,low,middle)
4     mergesort(L,low,middle)
5     merge(L, low, middle, high)
6 endIf
```

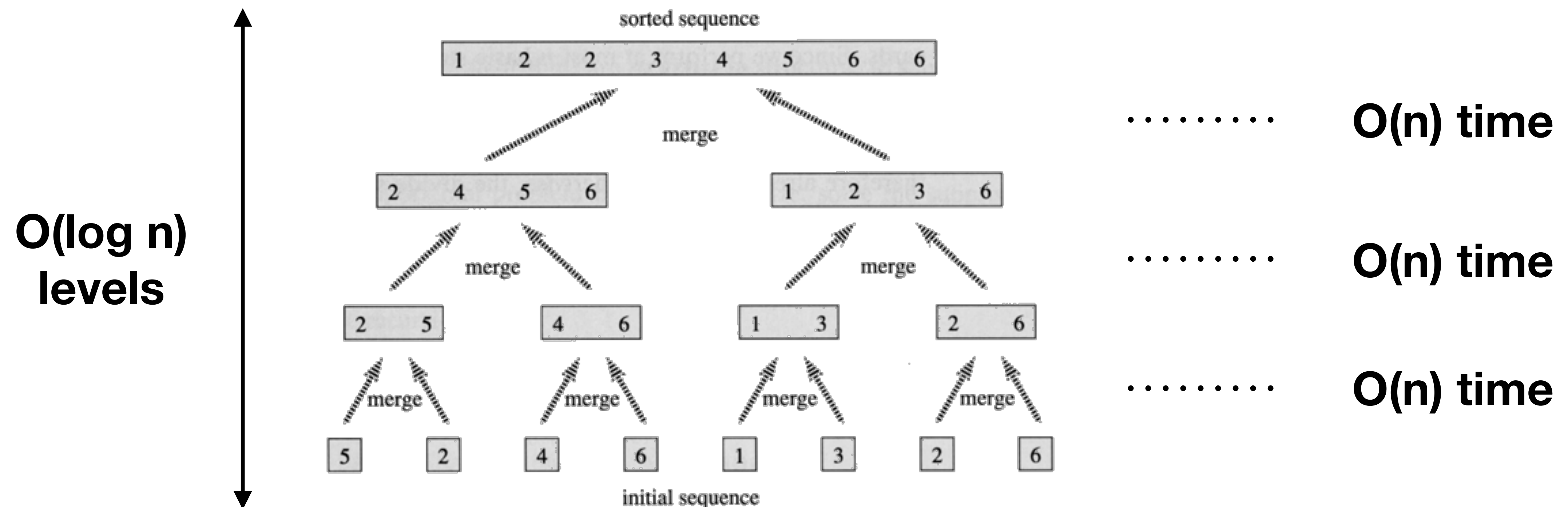
Merge sort

- The efficiency of mergesort depends upon how efficiently we merge the two sorted halves L_1, L_2 into a single sorted list L.
- You had this question in Homework 1.
- If L_1 or L_2 is empty, we copy the other list to L.
- Otherwise let x be the minimum of the head elements in L_1 and L_2. We pop x and push it to L, and apply the procedure to the new lists L_1 and L_2 recursively.
- **Example:** [4, 5, 8] and [6,7]

Merge sort

Theorem: Merge sort takes O(n log n) time.

Proof:



Thus the worst case time is O(n log n).

Merge sort

Theorem: Merge sort takes $O(n \log n)$ time.

One more proof:

$T(n) = 2\,T(n/2) + O(n)$

By Master theorem, the worst case time is $O(n \log n)$.

Quicksort

Quicksort

- In practice, the fastest sorting algorithm is Quicksort, which uses partitioning as its main idea.

Example: pivot about 10

Before: 17 12 6 19 23 8 5 10

After: 6 8 5 10 23 19 12 17

- Partitioning places all the elements less than the pivot in the left part of the array, and all elements greater than the pivot in the right part of the array. The pivot fits in the slot between.
- Note that the pivot element ends up in the correct place in the total order!

Quicksort

- We can partition an array about the pivot in one linear scan, by maintaining three sections: < pivot, > pivot, and unexplored.

Animation

- As we scan from left to right, we move the left bound to the right when the element is less than the pivot, otherwise we swap it with the rightmost unexplored element and move the right bound one step closer to the left.

Quicksort

Since the partitioning step consists of at most n swaps, takes time linear in the number of keys. But what does it buy us?

1. The pivot element ends up in the position it retains in the final sorted order.

2. After a partitioning, no element flops to the other side of the pivot in the final sorted order.

Thus we can sort the elements to the left of the pivot and the right of the pivot independently, giving us a recursive sorting algorithm!

Quicksort

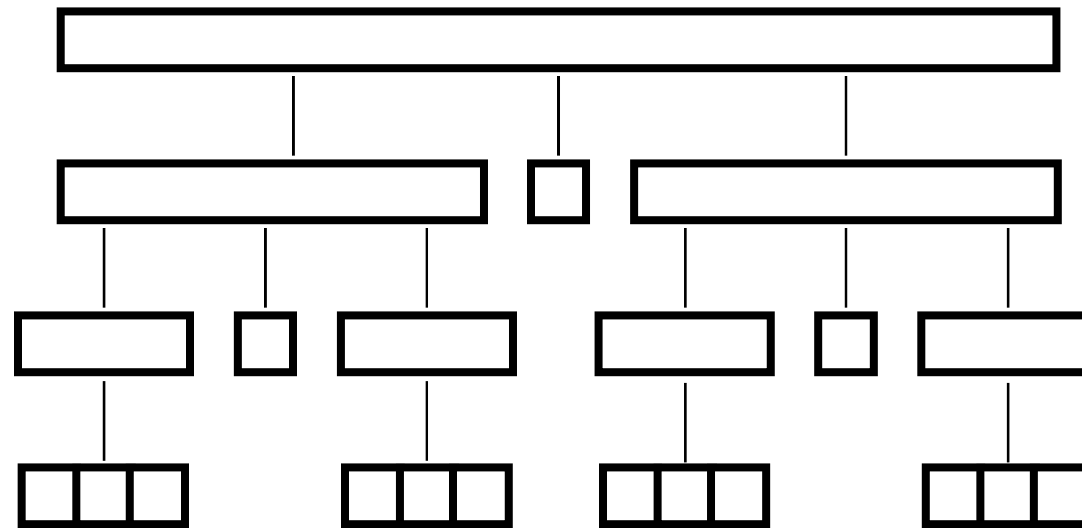
```
QuickSort(A, low, high)
1 if (low < high) then do
2    pivot-location = Partition(A,low,high)
3    QuickSort(A,low, pivot-location - 1)
4    QuickSort(A, pivot-location+1, high)
5 endIf
```

```
Partition(A, low, high)
1 pivot = A[low]
2 leftwall = low
3 for i = low+1 to high do
4   if (A[i] < pivot) then do
5       leftwall = leftwall+1
6       swap(A[i],A[leftwall])
7   endIf
8 endFor
9 swap(A[low],A[leftwall])
```

Quicksort

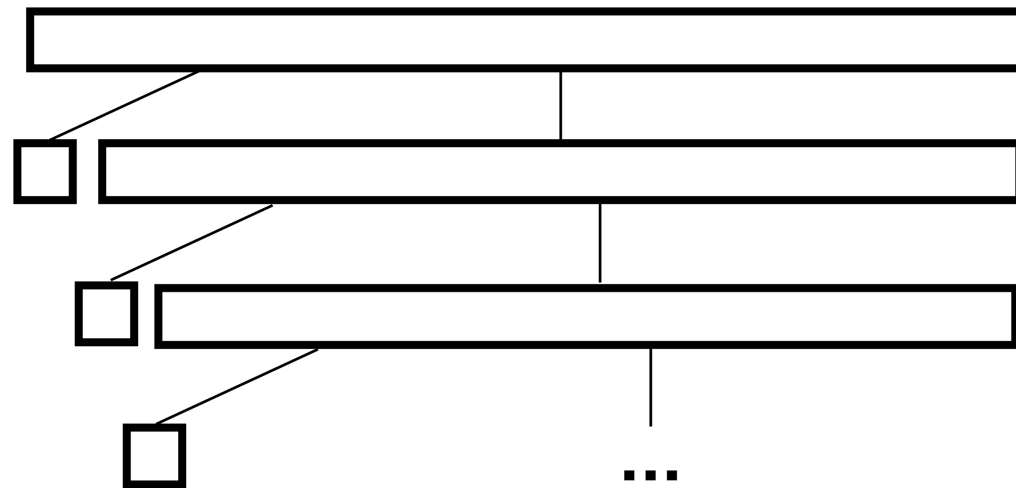
- Since each element ultimately ends up in the correct position, the algorithm correctly sorts. But how long does it take?
- The best case for divide-and-conquer algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size n/2.
- The partition step on each subproblem is linear in its size. Thus the total effort in partitioning the 2^k problems of size $n=2^k$ is $O(n)$.

Quicksort



The total partitioning on each level is O(n), and it takes lg n levels of perfect partitions to get to single element subproblems. When we are down to single elements, the problems are sorted. Thus total time in the best case is O(n lg n).

Quicksort



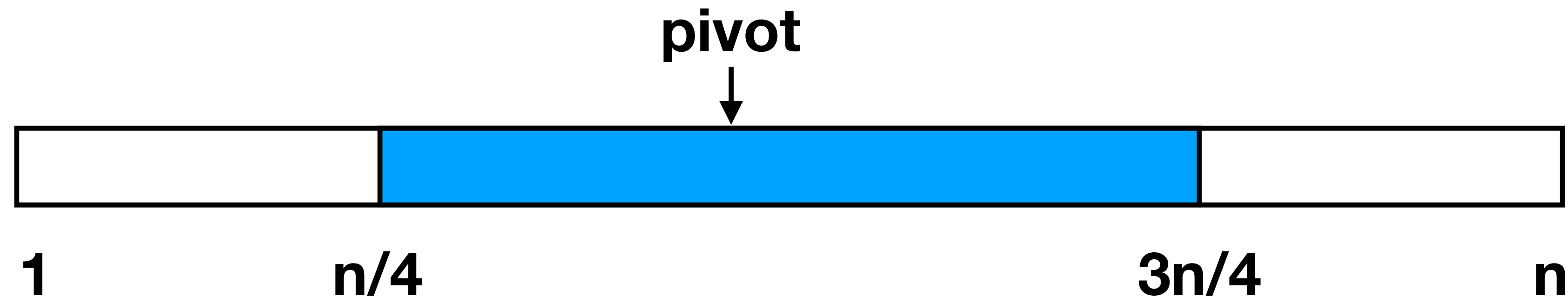
Suppose instead our pivot element splits the array as unequally as possible. Thus instead of n/2 elements in the smaller half, we get zero, meaning that the pivot element is the biggest or smallest element in the array.

Quicksort

- Now we have n-1 levels, instead of lg n, for a worst case time of $\Theta(n^2)$, since the first n/2 levels each have n/2 elements to partition.
- To justify its name, Quicksort had better be good in the average case. Showing this requires some intricate analysis.

Intuition: the average case for quicksort

- Suppose we pick the pivot element at random in an array of n keys.



- Half the time, the pivot element will be from the center half of the sorted array.

- Whenever the pivot element is from positions n/4 to 3n/4, the larger remaining subarray contains at most 3n/4 elements.

Intuition: the average case for quicksort

- If we assume that the pivot element is always in this range, what is the maximum number k of partitions we need to get from n elements down to 1 element?

- Since $(3/4)^k n = 1$, we have $k < 3 \log n$

Intuition: the average case for quicksort

- How often when we pick an arbitrary element as pivot will it generate a decent partition?
- Since any number ranked between n/4 and 3n/4 would make a decent pivot, we get one half the time on average.
- If we need 3 log n levels of decent partitions to finish the job, and half of random partitions are decent, then on average the recursion tree to quicksort the array has 6 log n levels.
- Each level takes O(n) time, so the total running time of quicksort is O(n log n) in the average case.

Average-case analysis of quicksort

Theorem: The average-case running time of quicksort is O(n log n).

Proof: See the blackboard.

Heaps and heap sort

Binary heap

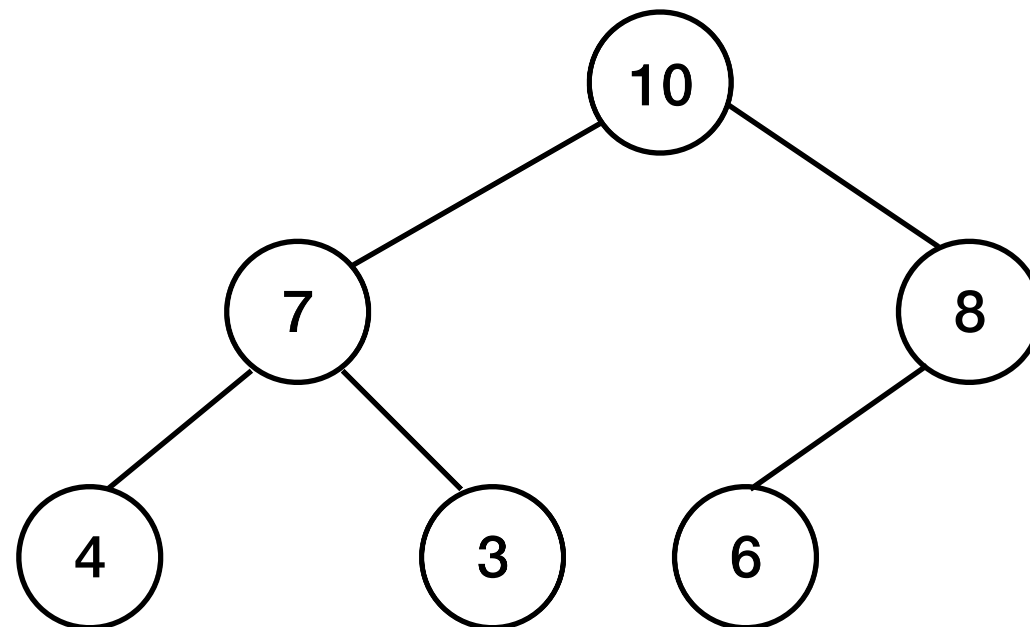
A binary heap is defined to be a binary tree with a key in each node such that:

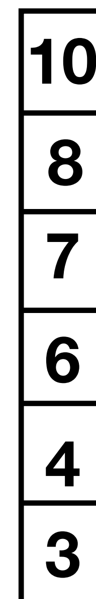
1. All leaves are on, at most, two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
3. The key in root is $\geq$ all its children, and the left and right subtrees are again binary heaps.

Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.

Binary heap

Heaps maintain a partial order on the set of elements which is weaker than the sorted order (so it can be efficient to maintain) yet stronger than random order (so the maximum element can be quickly identified).





Binary heap

Watch as

- We build the heap by repeated insertion.
- Embed it in an array.
- Then repeatedly remove the maximum to sort:

Animation

The partial order defined by the heap structure is weaker than sorting, which explains why it is easier to build: linear time if you do it right.

Binary heap

- The most natural representation of this binary tree would involve storing each key in a node with pointers to its two children.

- However, we can store a tree as an array of keys, using the position of the keys to implicitly satisfy the role of the pointers.

- The left child of k sits in position $2k$ and the right child in $2k+1$. The parent of k is in position $[k/2]$.

Binary heap

- Heaps can be constructed incrementally, by inserting new elements into the left-most open spot in the array.

- If the new element is greater than its parent, swap their positions and recur.

- Since all but the last level is always filled, the height h of an n element heap is bounded because:

$$2^{h+1} - 1 \geq n,$$

- So $h = \lfloor \log n \rfloor$. Doing n such insertions really takes $\Theta(n \log n)$ time, because the last n/2 insertions require $\Theta(\log n)$ time each.

Binary heap

- However, a heap can be constructed in O(n) time as well.
- Given two heaps and a fresh element, they can be merged into one (`Heapify`).

```
Build-heap(A)
1 n ← |A|
2 for i = [n/2] to 1 do
3    Heapify(A, i)
4 endFor
```

Binary heap

```
Heapify(A, i)
1 l = left_child(i), r = right_child(i)
2 largest = i
3 if (l ≤ n && A[l] > A[largest]) then largest = l endIf
4 if (r ≤ n && A[r] > A[largest]) then largest = r endIf
5 if (largest != i) then
6     swap(A[i], A[largest])
7     Heapify(A, largest)
8 endIf
```

Binary heap

- In a full binary tree on n nodes, there are at most $[n/2^{h+1}] + 1$ nodes of height h, so the cost of building a heap is:

$$\sum_{h=0}^{[\log n]} ([n/2^{h+1}] + 1)\, O(h) = n \cdot O(\underbrace{\sum_{h=0}^{[\log n]} h / 2^{h+1}}_{H})$$

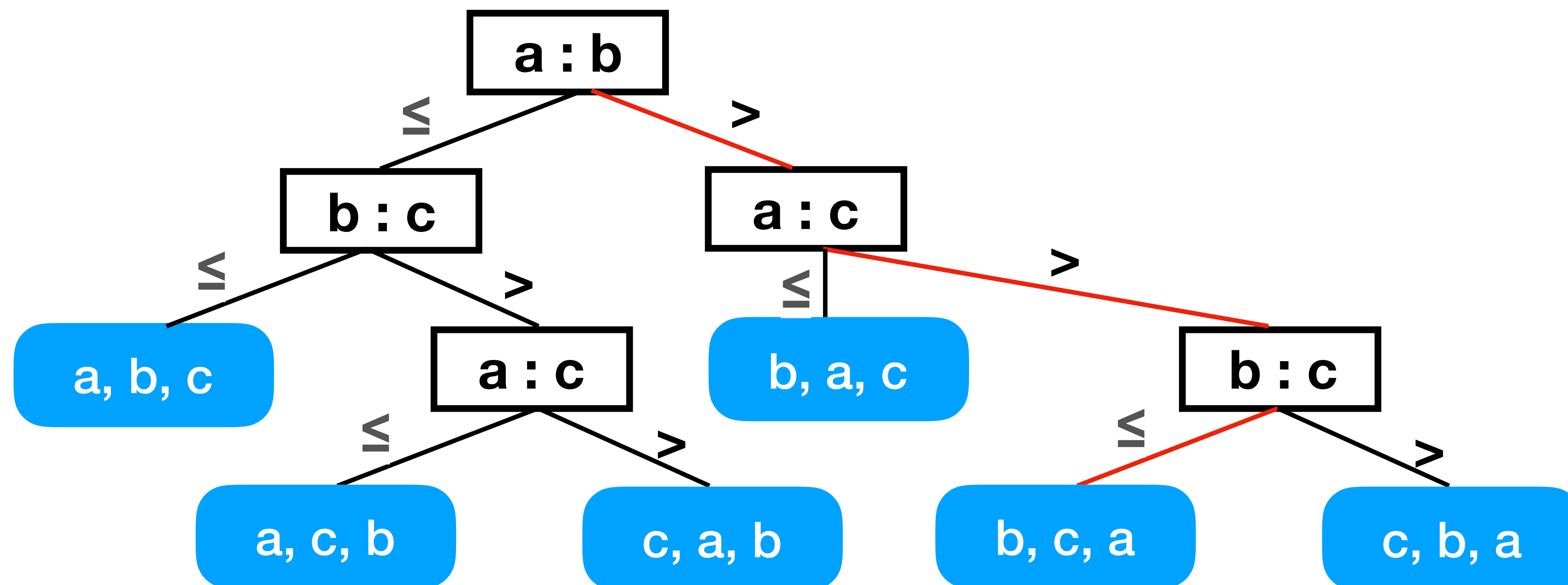
- $H \leq 2$ (see the blackboard).
- Hence, a heap can be constructed in O(n) time.

Lower bound

Can we sort in o(n log n)?

- Any comparison-based sorting program can be thought of as defining a decision tree of possible executions.
- Running the same program twice on the same permutation causes it to do exactly the same thing, but running it on different permutations of the same data causes a different sequence of comparisons to be made on each.

Decision tree



Example: a = 4, b = 2, c = 3

Claim: the minimum height of a decision tree is the worst-case complexity of comparison-based sorting.

Lower bound

Lemma: The height of any decision tree is $\Omega(n \log n)$.

Proof: Since any two different permutations of n elements requires a different sequence of steps to sort, there must be at least n! different paths from the root to leaves in the decision tree.

Thus there must be at least n! different leaves in this binary tree.

Since a binary tree of height h has at most 2^h leaves, we know that $n! \leq 2^h$, or $h \geq \lg(n!)$.

In TD1 we showed that $\lg(n!) = \Omega(n \log n)$.

Lower bound

Claim: the minimum height of a decision tree is the worst-case complexity of comparison-based sorting.

Lemma: The height of any decision tree is $\Omega(n \log n)$.

Theorem: Any comparison-based sorting algorithm must use $\Omega(n \log n)$ time.

[Beyond this course] The lower bound holds even for average-case time complexity and even for randomised algorithms.

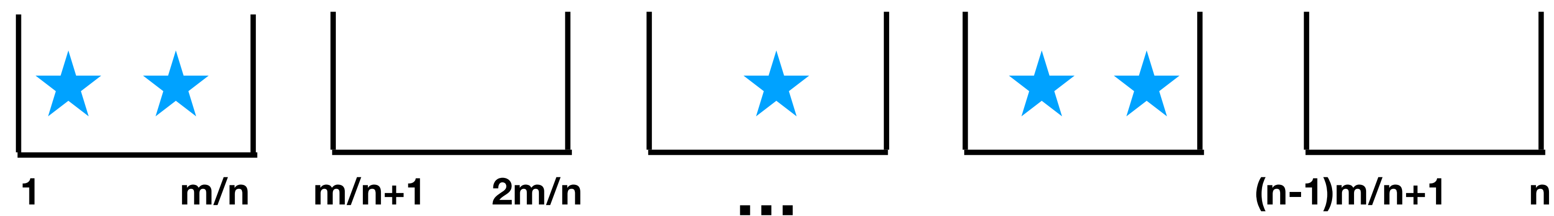
Bucket sort

Bucket sort

- All the sorting algorithms we have seen assume binary comparisons as the basic primitive, questions of the form “is x before y?”.
- But how would you sort a deck of playing cards?
- Most likely you would set up 13 piles and put all cards with the same number in one pile. With only a constant number of cards left in each pile, you can use insertion sort to order by suite and concatenate everything together.
- If we could find the correct pile for each card in constant time, and each pile gets O(1) cards, this algorithm takes O(n) time.

Bucket sort

Suppose we are sorting n numbers, where each number is in [1,m]. Suppose also that the numbers are approximately uniformly distributed. We can set up n buckets:



Given an input number x, it belongs in bucket [xn/m]+1. If we use an array of buckets, each item gets mapped to the right bucket in O(1) time. Sort each bucket using a quadratic-time algorithm.

Bucket sort

Theorem: On average, bucket sort takes O(n) time.

Proof:

With uniformly distributed keys, the expected number of items per bucket is 1. Thus sorting each bucket takes O(1) time!

The total effort of bucketing, sorting buckets, and concatenating the sorted buckets together is O(n).

For formal proof, see the blackboard.

“People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.”

– Donald Knuth