Video 1. Lower bound

k - number of leaves $\Omega(n!)$

$h = \Omega(\log_2 k) = \Omega(\log_2(n!))$

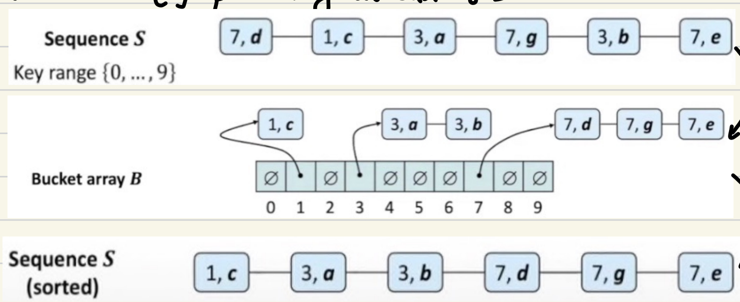
$\log_2(n!) \geq \log_2\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2}\log_2\frac{n}{2} \Rightarrow \Omega(n\log_2 n)$

Video 2. Bucket Sort

input: sequence S, integer keys $[0, N-1]$

B - array of N sequences $B[\text{key}] \cdot S[\text{entry}]$

$i: 0 \to N-1$ $B[i]$: put entry at end of S



$O(n+N)$

$O(n+N)$

stable sorting – preserving relative order od same key items

Video 3. Radix Sorts

N = radix

composite keys: d-tuples od elementary keys key $= (k_1, k_2, \dots, k_d)$

backward processing: LSD (least significant digit first)

↳ applying bucket sort d times $= O(d(n+N))$

forward processing: MSD (most significant digit first)

↳ applying bucket sort to each bucket of the previous

same complexity might be faster

Sorting algorithms

	Stable	In-place	Best-case time	Worst-case time	Space	Application
Selection sort	Yes	Yes	$O(n^2)$	$O(n^2)$	$O(1)$	Small sequences, but insertion is better.
Insertion sort	Yes	Yes	$O(n+m)$	$O(n^2)$	$O(1)$	Small sequences. It's $O(n)$ for nearly sorted (since # inversions m is small).
Heap sort	No	Yes	$O(n \log n)$	$O(n \log n)$	$O(1)$	Small to mid-size sequences that fit into memory. Slower than quick, merge sort.
Quick sort	No	Yes	$O(n \log n)$*	$O(n^2)$	$O(\log n)$*	General purpose if space is tight and stability is not a concern.
Merge sort	Yes	No	$O(n \log n)$	$O(n \log n)$	$O(n)$	General purpose for very large data that doesn't fit into main memory.
Radix sort	Yes	No	$O(d(n+N))$	$O(d(n+N))$	$O(n+N)$	Integers, strings, other d-tuples. If $d(n+N) \ll n \log n$, radix are faster than comparison-based algorithms.

Video 4. **Selection**

Algorithm quickSelect(S, k)
Input: Sequence S of n elements, and an integer $k \in \{1, \dots, n\}$
Output: The k^{th} smallest element of S
If $n == 1$ **then**
  return the (first) element of S
Pick a random (pivot) element x of S and divide S into three sequences:
  L, storing the elements in S less than x
  E, storing the elements in S equal to x
  R, storing the elements in S greater than x
If $k \le |L|$ **then**
  return quickSelect(L, k)
  else if $k \le |L| + |E|$ **then**
    return x
  else
    return quickSelect(R, $k - |L| - |E|$)

```
public static int selectInPlace(int[] array, int k) {
  return selectInPlace(array, 0, array.length-1, k-1);
}

private static int selectInPlace(int[] array, int left, int right, int i) { //i index of k-th
  if (left == right)
    return array[left];       // if there's only one element, return that element

  // select random pivot index between left and right (same as quick sort)
  int pivotIdx = randomPivot(left, right);
  // swap pivot with last element, partition, assign updated pivot index (same as quick sort)
  pivotIdx = partition(array, left, right, pivotIdx);

  if (i == pivotIdx)
    return array[i];                                   // k-th element is the pivot
  else if (i < pivotIdx)
    return selectInPlace(array, left, pivotIdx - 1, i);    // k-th element is <= pivot
  else
    return selectInPlace(array, pivotIdx + 1, right, i);   // k-th element is >= pivot
}
```

best case: $n + \frac{n}{2} + \frac{n}{4} + \dots + \frac{n}{2^{\lceil \log_2 n \rceil}} = n \sum_{i=0}^{\lceil \log_2 n \rceil} \left(\frac{1}{2}\right)^i = O(n)$ space: $O(\log_2 n)$

worst case: $n + (n-1) + (n-2) + \dots + 1 = O(n^2)$ space: $O(n)$

'good' partition = 1:3

$t(n)$ = (best case) $O(n)$ running time

$g(n)$ - # conseq. bad calls | $E(f(x))$ - expected time

$t(n) \le bn\,g(n) + t\left(\frac{3n}{4}\right)$, $b \ge 1$

$E(t(n)) \le E\left(bn\,g(n) + t\left(\frac{3n}{4}\right)\right) = bn\,E(g(n)) + E\left(t\left(\frac{3n}{4}\right)\right)$

$E(t(n)) \le 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} \left(\frac{3}{4}\right)^i \rightarrow O(n)$