Lower Bound for Sorting Complexity

- **Theorem 2.30**: Any algorithm that sorts by comparing only pairs of elements must use at least

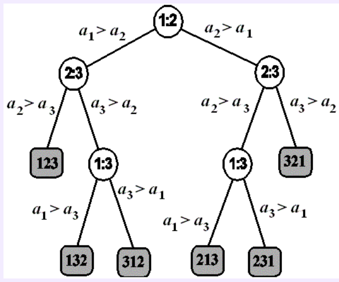
$$\lceil \log_2(n!) \rceil \cong n \log_2 n - 1.44n$$

comparisons in the worst case (that is, for some "worst" input sequence) and in the average case

  - Stirling's approximation of the factorial ($n!$):

$$1 \cdot 2 \cdot \ldots \cdot n \equiv n! \quad \geq \quad \left(\frac{n}{e}\right)^n \sqrt{2\pi n} \approx 2.5 n^{n+0.5} e^{-n}$$

Decision Tree for Sorting n Items



Decision tree for $n = 3$:

- $i{:}j$ - a comparison of a_i and a_j
- ijk - a sorted array $(a_i\, a_j\, a_k)$
- $n!$ permutations $\Rightarrow$ $n!$ leaves

Sorting in descending order of the numbers

Decision Tree for Sorting n Items

- Decision tree for $n = 3$: an array $A = \{a_1, a_2, a_3\}$
- Example: $\{a_1 = 35, a_2 = 10, a_3 = 17\}$
  - Comparison $1{:}2$ $(35 > 10) \rightarrow$ left branch $a_1 > a_2$
  - Comparison $2{:}3$ $(10 < 17) \rightarrow$ right branch $a_2 < a_3$
  - Comparison $1{:}3$ $(35 > 17) \rightarrow$ left branch $a_1 > a_3$
- Sorted array $132 \rightarrow \{a_1 = 35, a_3 = 17, a_2 = 10\}$

Decision Tree

Lemma: Decision tree of height h has $L_h \leq 2^h$ leaves

Proof by mathematical induction:

- $h = 1$: any tree of height 1 has $L_1 \leq 2^1$ leaves
- $h-1 \rightarrow h$:
  - Let any tree of height $h - 1$ have $L_{h-1} \leq 2^{h-1}$ leaves
  - Any tree of height h consists of a root and two subtrees of height at most $h - 1$
  - Therefore, $L_h = L_{h-1} + L_{h-1} \leq 2^{h-1} + 2^{h-1} = 2^h$

Worst-Case Complexity of Sorting

- **Theorem 2.32**: The worst-case complexity of sorting n items by pairwise comparisons is $\Omega(n \log n)$
- **Proof**:
  - Any decision tree of height h has at most 2^h leaves (see Lemma, Slide 4)
  - The least height h such that $L_h = 2^h \geq n!$ leaves is

$$h \geq \log_2(n!) \cong n \log_2 n - 1.44\, n$$

Bucket Sort (Exercise 2.6.2)

Let all integers to sort in an array a of size n be in the **fixed range** $[1,\ldots,q_{max}]$

1. Introduce a counter array t of size q_{max} and set its entries initially to zero
2. Scan through a to accumulate in the counters $t[i]$; $i = 0,\ldots,q_{max}-1$, how many times each item $i + 1$ is found in a
3. Loop through $0 \leq i \leq q_{max}-1$ and output $t[i]$ copies of integer $i + 1$ at each step

Bucket Sort (Exercise 2.6.2)

Worst- and average-case time complexity of bucket sort is $\Theta(n)$ provided that $q_{\max}$ is fixed

- $q_{\max} + n$ elementary operations to first set t to zero and then count how many times $t[i]$ each item $i + 1$ is found in a
- $q_{\max} + n$ elementary operations to successively output the sorted array a by repeating $t[i]$ times each entry $i + 1$

Theorem 2.30 **does not hold** under additional constraints!

Data Search: Efficiency

- Data record ⇔ Specific **key**
- Goal: to find all records with **key**s matching a given search **key**
- Purpose:
  - to access information in the record for processing, or
  - to update information in the record, or
  - to insert a new record or to delete the record

Types of Search

- Static search: **unalterable databases**
  - Given a data structure D of records and a search key k, either return the record associated with k in D or indicate that k is not found, without altering D
  - If k occurs more than once, return any occurrence
    - **Examples**: searching a phone directory or a dictionary
- Dynamic search: **alterable databases**
  - Records may be inserted or removed

Static Sequential Search (SSS)

- **Lemma 3.3**: Both successful and unsuccessful SSS have worst- and average-case complexity $\Theta(n)$
  - **Proof**: the unsuccessful search explores each of n keys, so the worst- and average-case time is $\Theta(n)$; the successful search examines n keys in the worst case and $n/2$ on the average, which is still $\Theta(n)$
    - Sequential search is the only option for an unsorted array and for linked-list data structures of records

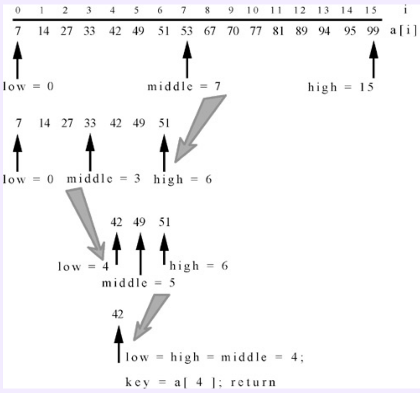
Static Binary Search $O(\log n)$

- Ordered array: $\mathbf{key}_0 < \mathbf{key}_1 < \ldots < \mathbf{key}_{n-1}$
- Compare the search **key** with the record $\mathbf{key}_i$ at the middle position $i = \lfloor (n-1)/2 \rfloor$
  - if $\mathbf{key} = \mathbf{key}_i$, return i
  - if $\mathbf{key} < \mathbf{key}_i$ or $\mathbf{key} < \mathbf{key}_i$, then it must be in the 1st or in the 2nd half of the array, respectively
- Apply the previous two steps to the chosen half of the array iteratively (**repeating halving principle**)

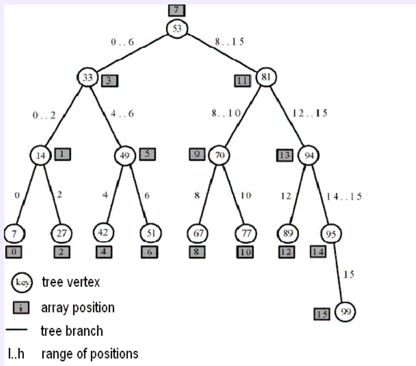
Pseudocode of Binary Search

```
begin BinarySearch (an integer array a of size n, a search key)
   low ← 0;  high ← n – 1
   while low ≤ high do
      middle ← ⌊( low + high ) / 2⌋
      if a[ middle ] < key  then  low  ←  middle + 1
      else if a[ middle ] > key then  high  ←  middle – 1
      else return  middle end if
   end while
   return ItemNotFound
end BinarySearch
```

Binary search: detailed analysis



Comparison structure: the binary (search) tree



Worst-Case Complexity $\Theta(\log n)$ of Binary Search

- Let $n = 2^k - 1$; $k = 1,2,\ldots$, then the binary tree is complete (each internal node has 2 children)
  - The tree height is $k - 1$ since the tree is **balanced**
  - Each tree level l contains 2^l nodes for $l = 0$ (the root), $1, \ldots, k - 2, k - 1$ (the leaves)
- $l + 1$ comparisons to find a key of level l
- **The worst case**: $k = \log_2(n + 1)$ comparisons so that the time complexity is $\Theta(\log n)$

Average-Case Complexity $\Theta(\log n)$ of Binary Search

Lemma 3.9: The average-case complexity of binary search in a balanced binary tree is $\Theta(\log n)$

Proof: $k = \lceil \log_2(n + 1) \rceil - 1$ is the depth of the tree

At least half of the nodes in the tree have the depth at least $k - 1$

The average depth over all nodes is at least $k/2$ which is $\Omega(\log n)$

Expected search time for **an arbitrary binary search tree** is equal to the average tree height $\Theta(\log n)$

Interpolation Search

- Improvement of binary search if it is possible to guess where the desired key sits
  - **Example**: the search for C or X in a phone directory
  - Practical if the sorted keys are almost uniformly distributed over their range
- BS: the middle position $m = \left\lfloor \frac{l + r}{2} \right\rfloor = l + \lceil 0.5(r - l) \rceil$
- IS: the predicted position

$$m = l + \lceil \rho(r - l) \rceil \equiv l + \left\lceil \frac{k - A[l]}{A[r] - A[l]}(r - l) \right\rceil$$

Dynamic Binary Tree Search

- Static binary search is converted into a **dynamic binary tree search** by allowing for insertion and deletion of data records
- **Binary tree search** makes actual use of the **binary search tree data structure**
  - The data structure is constructed by linking data records
  - Any node of a binary search tree may be removed