Advanced Algorithms
I.1. Algorithm Design and Analysis

Henry Förster

Chair of Efficient Algorithms

Algorithmic Revolution

KIND OF NEW: Machines learn algorithms based on information without explicit instructions

Even difficult problems can be solved that way!

Application scenarios where the definition is fuzzy can be solved!
(E.g., the machine should provide a „human-like“ response)

Why do we still discuss classic algorithms?

Shooting Sparrows with Cannons

Image generated with Microsoft Designer.

- *Efficient algorithms…*
  - solve *problems directly*
  - require *less computational ressources*
  - can run *on restrictive hardware*
  - can be *cheaper to operate*
  - require *no expensive training* (cost, emissions, …)
- *Also, it is fun to solve puzzles!*
  - And understand *why* the solution works!

Welcome to Advanced Algorithms!

- Hi, I‘m *Henry*!
  - *PhD* and *PostDoc* at *University of Tübingen*
  - *Research Focus:* Efficient Algorithms for *Visualizing Relational Data,*
    Properties of *Geometric and Topological Graphs,*
    Evaluating *Visualization Techniques*
  - This course is *not (completely) ad-hoc:* taught *similar classes in Tübingen*
- *How about you?*
- *Disclaimer:* Today is *my very first day* here!
  - *moodle* will be available *later today*
  - *Exam date* will be announced *later today on moodle* and *Thursday in class*

Course Organization

- *Lecture:* Tuesday 10:15 – 11:45
- Again, there will be a *moodle*
  - Slides, assignment sheets, forum, announcements, …
- *Assignment sheets*
  - Available *after lecture* (this week later today when *moodle is running*)
  - *Voluntary*, but *graded for feedback*
  - Hand-in via *moodle until Tuesday before lecture*, possible *in groups*
- *Discussion Session:* Thursday, 10:15 – 11:45 (same room)
  - *Per request:* Solutions for *assignment sheet from last week*, *questions* regarding the lecture, guidance on *current assignment sheet*
  - *This week:* Announcements regarding *moodle* and *exam date*
- *Written Exam:* Date TBA, 120 mins
  - Cheat sheet allowed (A4, both sides)

Course Contents

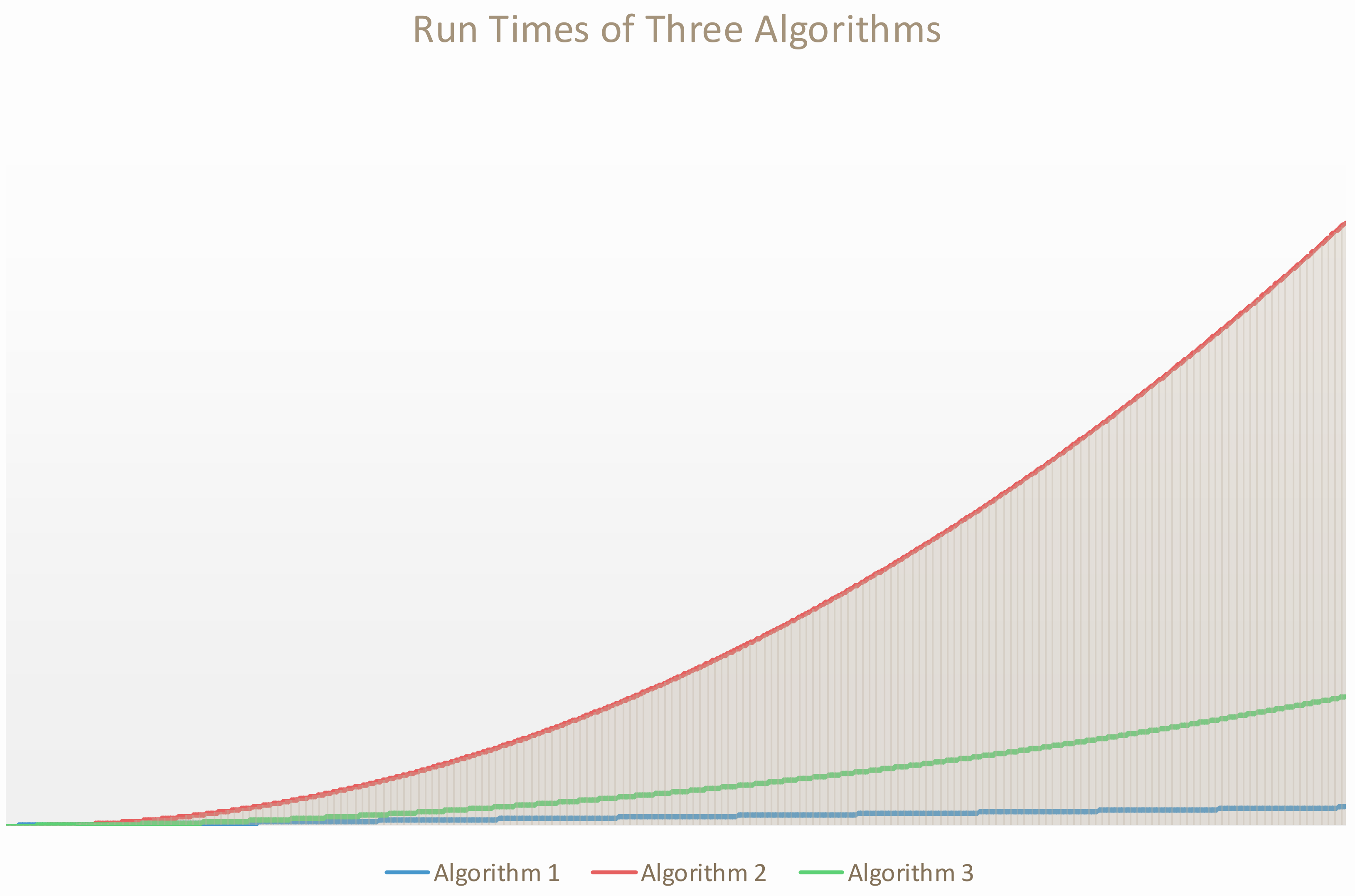
I. Algorithm Design and Analysis	***1. Recap:*** **Run time and Correctness, Landau Notation, Divide and Conquer** *2. Advanced Design Concepts:* Dynamic Programming, Greedy *3. Amortized Analysis*
II. Advanced Data Structures	*1. Fibonacci-Heaps* *2. Union-Find Data Structure*
III. Graph Algorithms	*1. Maximum Flow & Maximum Matching* *2. Push-Relabel Algorithm*
IV. Geometric Algorithms	*1. Sweep Line Method* *2. Randomized Incremental Construction*
V. Linear Programming	*1. Properties and Duality* *2. Algorithms for Linear Programming*
VI. Approximation Algorithms	*1. APX* *2. PTAS*
VII. Parameterized Algorithms	*1. FPT* *2. W[1]*

Goals for Today

At the end of this lecture, you can …

- ☐ prove the *correctness* of algorithms using suitable *invariants and induction*
- ☐ provide *bounds on run times* using the *Landau notation*
- ☐ *resolve recurrences* by applying the *master theorem*
- ☐ design *recursive* and *divide-and-conquer algorithms*

Comparing Run Times



Landau Notation (or Big-O Notation)

- *Analysis Goals:* Give a good estimate for the run time!
  - *No bias* on *size of data*
  - *Independent* of assumptions on *data distribution*
  - *Stable* with respect to *implementation details*
- *Worst-Case Asymptotic Bounds* expressed in *Landau Notation*
  - Let $f, g\colon \mathbb{N} \to \mathbb{R}_+$
  - f is *at most order* of g or $f \in O(g)$ or $f = O(g)$ iff
$$\exists c > 0 \colon \exists n_0 > 0 \colon \forall n \geq n_0 \colon f(n) \leq c \cdot g(n)$$
$$\Leftrightarrow \limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$
  - We also say g is *upper bound* for f
- *Does this kind of definition fulfill the goals?*

Landau Notation (or Big-O Notation)

- *More Useful Definitions:*
  - f is *at least order* of g or $f \in \Omega(g)$ or $f = \Omega(g)$ iff

    $$\exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : f(n) \geq c \cdot g(n) \Leftrightarrow \liminf_{n \to \infty} \frac{f(n)}{g(n)} > 0$$

  - f is *order* of g or $f \in \Theta(g)$ or $f = \Theta(g)$ iff $f \in O(g)$ and $f \in \Omega(g)$
  - Ω is used to establish *lower bounds*, Θ for *tight bounds*
- *Notation for imprecise bounds:*
  - f is *order strictly greater* than g or $f \in \omega(g)$ or $f = \omega(g)$ iff

    $$\forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : f(n) > c \cdot g(n) \Leftrightarrow \liminf_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

  - f is *order strictly lesser* than g or $f \in o(g)$ or $f = o(g)$ iff

    $$\forall c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : f(n) < c \cdot g(n) \Leftrightarrow \limsup_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Running Example for Today: SORTING

- *Input:* An *unsorted set A* of n distinct *comparable* elements
  - *Assumption:* A is implemented an *array*
  - *Elements* are taken from *universe U* with associated *abstract total order* $\preccurlyeq$

3	5	7	0	4	2	6	1

- *Output:* A *n-tuple $S = (s_1, s_2, \dots, s_n)$* such that
  - S contains *exactly the elements of A*, i.e., $\bigcup_{i=1}^{n} s_i = A$
  - S is *sorted w.r.t. to* $\preccurlyeq$, i.e., $\forall i \in \{1, \dots, n-1\}\colon s_i \preccurlyeq s_{i+1}$
  - *Assumption:* S is implemented as an *array*

0	1	2	3	4	5	6	7

- *Which sorting algorithms do you know? What are their run times?*

Recursive Bubble Sort

- *One strategy for sorting:*
  1. Find the *largest element a^** of A and put it *at position $|A|$*
  2. Sort the elements of $A \setminus \{a^*\}$ *recursively* (i.e., the *first $|A| - 1$ elements* of the array)
  3. Return A

- *Paradigm:* An algorithm that *calls itself* to solve *related problems* or *subproblems* is called *recursive algorithm* or *recursion*.
  - *Base Case:* Trivial solution *without self-call*

```
BubbleSort(A,n)
{
    if (n=1)
    {
        return A;
    }
    for i←1 to n-1
    {
        if A[i] > A[i+1]
        {
             swap(A[i],A[i+1]);
        }
    }
    BubbleSort(A,n-1);
    return A;
}
```

Recursive Bubble Sort – Run Time

- *Run Time:*
  - n comparisons
  - potentially n *swaps*
  - *Recursive call* on $n \leftarrow n-1$
  - Expression as *recurrence*:
    $$T(n) \le T(n-1) + 2n \qquad T(1) \le 2$$
- *Solving the recurrence:*
  - *Insert recursive formulation* a few times:
    $$\begin{aligned} T(n) &= T(n-1) + 2n \\ &= T(n-2) + 2(n-1+n) \\ &= T(n-3) + 2(n-2+n-1+n) = \cdots \end{aligned}$$
  - Educatedly *guess closed form*:
    $$T(n) = 2\sum_{i=1}^{n} i = n(n+1)$$

```
BubbleSort(A,n)
{
   if (n=1)
   {
      return A;
   }
   for i←1 to n-1
   {
      if A[i] > A[i+1]
      {
         swap(A[i],A[i+1]);
      }
   }
   BubbleSort(A,n-1);
   return A;
}
```

Recursive Bubble Sort – Run Time

- *Status Quo:*
  - *Given:* $T(n) = T(n-1) + 2n \qquad T(1) = 2$
  - *Guess:* $T(n) = n(n+1)$
- *Next Up:* Prove inductively!
  - *Induction Base:* $T(1) = 1(1+1) = 2$
  - *Induction Hypothesis:*
    For a *fixed* $n \geq 1$, $T(n) = n(n+1)$
  - *Inductive Step:*

$$\begin{aligned} T(n+1) &= T(n) + 2(n+1) \\ &=^{IH} n(n+1) + 2(n+1) \\ &= (n+2)(n+1) \qquad \square \end{aligned}$$

```
BubbleSort(A,n)
{
   if (n=1)
   {
      return A;
   }
   for i←1 to n-1
   {
      if A[i] > A[i+1]
      {
         swap(A[i],A[i+1]);
      }
   }
   BubbleSort(A,n-1);
   return A;
}
```

Recursive Bubble Sort – Correctness

- *Still to show:* The algorithm is correct!
- *Definition:* A *loop invariant* is a *property* which *remains true* if it was true *before entering* the loop.
- Useful tool for *proving correctness*!
  - *Initialisation:* Establish that the invariant holds before the loop
  - *Continuation:* Show that the loop invariant remains true in the i-th iteration assuming that it holds after the $(i-1)$-th.
  - *Termination:* Use the loop invariant for the remainder of the proof.

```
BubbleSort(A,n)
{
    if (n=1)
    {
        return A;
    }
    for i←1 to n-1
    {
        if A[i] > A[i+1]
        {
            swap(A[i],A[i+1]);
        }
    }
    BubbleSort(A,n-1);
    return A;
}
```

Recursive Bubble Sort – Correctness

- *For our for-loop:* After *i-th iteration*, `A[i+1]` is the *greatest of the first $i + 1$* elements.
- *Initialisation:* Before the first iteration ($i = 1$), `A[1]` is greatest of the first 1 elements.
- *Continuation:*
  - `A[i]` is greatest amongst the first i elements *by assumption of the invariant before the iteration*
  - Thus, *either* `A[i]` *or* `A[i+1]` is the greatest
  - *We swap if necessary!*
- *Termination:* `A[n]` is *greatest element of A*
  - *Remainder of the proof:* Induction

```
BubbleSort(A,n)
{
   if (n=1)
   {
      return A;
   }
   for i←1 to n-1
   {
      if A[i] > A[i+1]
      {
          swap(A[i],A[i+1]);
      }
   }
   BubbleSort(A,n-1);
   return A;
}
```

Recursive vs. Iterative Algorithms

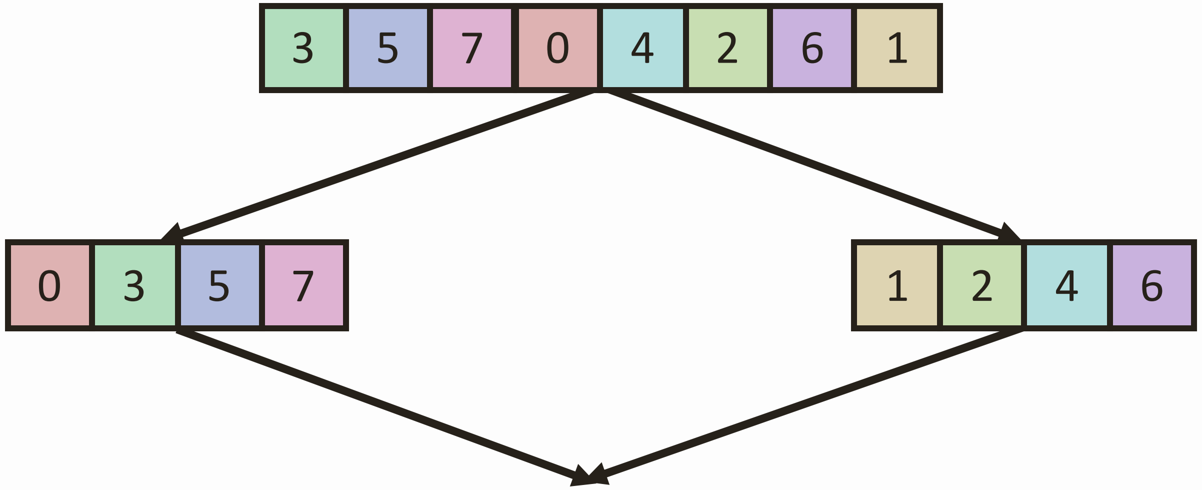
- *Iterative version:* Recursive calls are replaced by a loop!
- Works because the *signatures of calls are known in advance*
- *In general:* Recursion is *more powerful* than iteration!
  - *Recall:* Halting problem and Incomputability
  - But also *more difficult to analyze, implement,* …

```
BubbleSort_iter(A,n)
{
   for j←1 to n-1
   {
      for i←1 to n-j
      {
         if A[i] > A[i+1]
         {
             swap(A[i],A[j]);
         }
      }
   }
   return A;
}
```

```
BubbleSort_rec(A,n)
{
   if (n=1)
   {
      return A;
   }
   for i←1 to n-1
   {
      if A[i] > A[i+1]
      {
          swap(A[i],A[i+1]);
      }
   }
   BubbleSort(A,n-1);
   return A;
}
```

Merge Sort

- *Another strategy for sorting:*
  1. Sort *first half A_1* and *second half A_2 independently* yielding S_1 and S_2
  2. Compare *first elements of S_1 and S_2* and move the lesser in S
  3. Repeat 2. *until sorted*



Merge Sort – Analysis

- *Paradigm:* A *divide-and-conquer algorithm* is a *recursive algorithm* that constructs a solution based on *optimal recursively constructed subsolutions.*
- *Correctness (Sketch)*
  - Prove correctness of *merge phase via invariant*
  - *Induction* on n (assume power of 2)
- *Run Time:*

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n \qquad T(1) = 1$$

  - Solvable with *Master Theorem*

```
MergeSort(A)
{
    if (n=1)
    {
        return A;
    }
    S1←MergeSort(A[1,…,n/2]);
    S2←MergeSort(A[n/2+1,…,n]);
    i1,i2 ← 1;
    for (i←1 to n)
    {
        if(S1[i1] < S2[i2])
        {
            S[i] = S1[i1];
            i1 = i1+1;
        }
        else
        {
            S[i] = S2[i2];
            i2 = i2+1;
        }
    }
    return S;
```

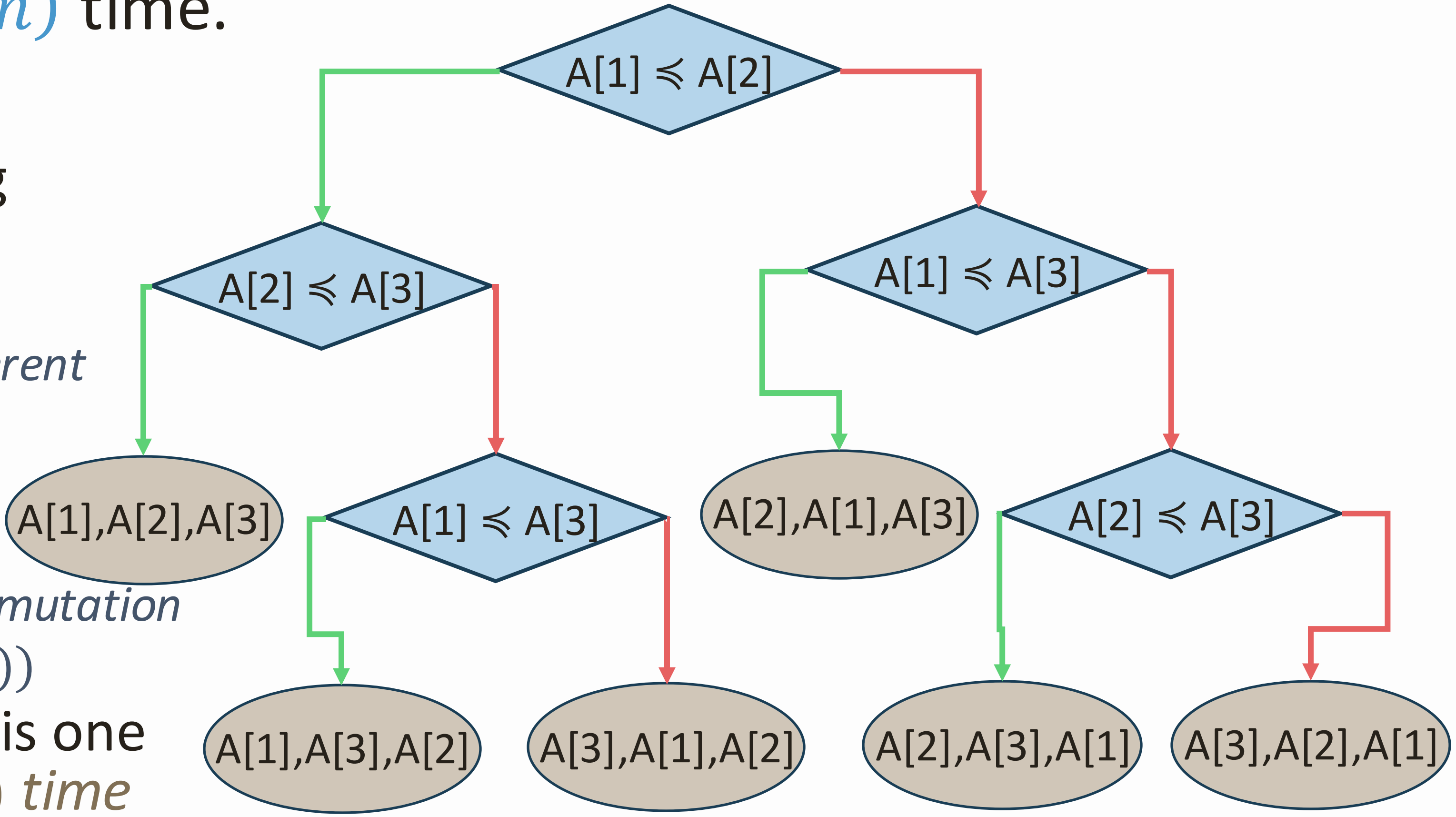
Master Theorem

- *Theorem:* Let $a, b \geq 1$ *constant* and let $f(n), T(n) \geq 0$ such that
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$
  - If $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for $\varepsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
  - If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \cdot \log_b n\right)$
  - If $f(n) = O\left(n^{\log_b a + \varepsilon}\right)$ for $\varepsilon > 0$ and $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for $c < 1$ and n sufficiently large, then $T(n) = \Theta(f(n))$
- *Merge Sort:* $T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n$
  1. *Identify a, b and $f(n)$:* $a = 2, b = 2, f(n) = c \cdot n$
  2. *Compute $\log_b a$:* $\log_2 2 = 1$
  3. *Compare $f(n)$ and $n^{\log_b a}$:* $f(n) = \Theta(n) = \Theta(n^{\log_b a})$

  $\Rightarrow$ $T(n) = \Theta(n \log n)$

Run Time Lower Bound for SORTING

- *Theorem: Comparison-based* sorting requires $\Omega(n \log n)$ time.
- *Proof:*
  - *Comparison-based* sorting algorithms can
    - *Compare* two elements
    - *Write* an element to a *different position*
  - Consider the *decision tree* of *any* algorithm
    - n! *leaves*, one for *each permutation*
    - thus, its *height is* $\Omega(\log(n!))$
  - *At each inner node*, there is one comparison ⇒ $\Omega(n \log n)$ *time*



SORTING – Summary

- *Theorem:* SORTING can be solved in $\Theta(n \log n)$ *time*.
  - *Bubble Sort* takes $O(n^2)$ time.
- *Remark:* There are *non-comparison-based* sorting algorithms.
  - *Examples:* Radix Sort, Bucket Sort, Counting Sort
  - The *lower bound* can be extended to these!
  - *But:* More efficient if data contains mostly *elements with short encoding* or *duplicates*.