Lecture Notes on Data Structures and Algorithms: Analysis of Algorithms

Conrado Martínez
U. Politècnica Catalunya

April 1, 2016

Part I

Analysis of Algorithms

Complexity of Algorithms

- Complexity of an algorithm = computational resources it consumes: execution time, memory space
- Analysis of algorithms $\rightarrow$ Investigate the propieties of the complexity of algorithms
  - Compare alternative algorithmic solutions
  - Predict the resources that an algorithm or data structure will use
  - Improve exisiting algorithms and data structures and guide the design of novel algorithms and DS

Complexity of Algorithms

In general terms, given an algorithm A with input set $\mathcal{A}$, its complexity or cost (in time, in memory space, in I/Os, etc.) is a function T from $\mathcal{A}$ to $\mathbb{N}$ (or $\mathbb{Q}$ or $\mathbb{R}$, depending on what we want to study):

$$\begin{aligned} T : \mathcal{A} &\to \mathbb{N} \\ \alpha &\to T(\alpha) \end{aligned}$$

Characterizing such a function is too complex and the huge amount of information it yields cannot be handled, and is impractical.

Worst-, Best-, Average-case Complexity

Let $\mathcal{A}_n$ denote the set of inputs of size n and $T_n : \mathcal{A}_n \to \mathbb{N}$ the restriction of T to $\mathcal{A}_n$.

- *Best-case cost*:

$$T_{\text{best}}(n) = \min\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Worst-case cost*:

$$T_{\text{worst}}(n) = \max\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Average-case cost*:

$$\begin{aligned} T_{\text{avg}}(n) &= \sum_{\alpha \in \mathcal{A}_n} \Pr(\alpha)\, T_n(\alpha) \\ &= \sum_{k \geq 0} k \,\Pr(T_n = k). \end{aligned}$$

Worst-, Best-, Average-case Complexity

1. For all $n \geq 0$ and for all $\alpha \in \mathcal{A}_n$

$$T_{\text{best}}(n) \leq T_n(\alpha) \leq T_{\text{worst}}(n).$$

2. For all $n \geq 0$

$$T_{\text{best}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{worst}}(n).$$

Worst-, Best-, Average-case Complexity

In general we will only study the worst-case complexity:

1. Provides a guarantee on the complexity of the algorithm, the cost will never exceed the worst-case cost
2. It is easier to compute than the average-case cost

Part I

Analysis of Algorithms

Rates of Growth

A fundamental feature of the cost of an algorithm (a function, in general) is its rate of growth

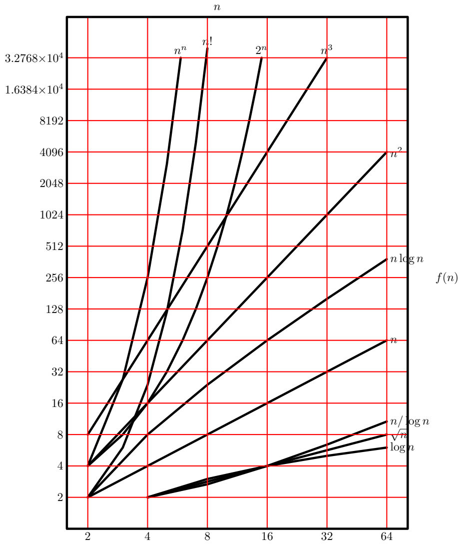
Example

1. Linear: $f(n) = a \cdot n + b \Rightarrow f(2n) \approx 2 \cdot f(n)$
2. Quadratic: $q(n) = a \cdot n^2 + b \cdot n + c \Rightarrow q(2n) \approx 4 \cdot q(n)$

We say that linear and quadratic functions have different rates of growth. We can also say that they are of different orders of magnitude.

Rates of Growth

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	262144
5	32	160	1024	32768	$6.87 \cdot 10^{10}$
6	64	384	4096	262144	$4.72 \cdot 10^{21}$
			$\cdots$		
ℓ	N	L	C	Q	E
$\ell + 1$	$2N$	$2(L+N)$	$4C$	$8Q$	E^2



Source: G. Valiente

Asymptotic Notation: Big-Oh

Constant factors and lower order terms are irrelevant as far as the rate of growth of a function is concerned: for instance, $30n^2 + \sqrt{n}$ has the same rate of growth as $2n^2 + 10n \Rightarrow$ asymptotic notation

Definition

Given a function $f : \mathbb{R}^+ \to \mathbb{R}^+$ the class $\mathcal{O}(f)$ (big-Oh of f) is

$$\mathcal{O}(f) = \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists n_0\, \exists c\, \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

In words, a function g is in $\mathcal{O}(f)$ if there exists a constant c such that $g < c \cdot f$ for all n from some value n_0 onwards.

Asymptotic Notation: Big-Oh

Although $\mathcal{O}(f)$ is a set of functions, people often write $g = \mathcal{O}(f)$ instead of $g \in \mathcal{O}(f)$. However, note that $\mathcal{O}(f) = g$ is nonsensical.

Basic properties of the $\mathcal{O}$ notation:

1. If $\lim_{n\to\infty} g(n)/f(n) < +\infty$ then $g = \mathcal{O}(f)$
2. It is reflexive: for all $f : \mathbb{R}^+ \to \mathbb{R}^+$, $f = \mathcal{O}(f)$
3. It is transitive: if $f = \mathcal{O}(g)$ and $g = \mathcal{O}(h)$ then $f = \mathcal{O}(h)$
4. For all positive constants $c > 0$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

Asymptotic Notation: Big-Oh

Since constant factors are irrelevant for the asymptotic notation we will systematically omit them: for instance, we will talk about $\mathcal{O}(n)$, not about $\mathcal{O}(4 \cdot n)$ (it is the same class); we will not express the base of logarithms unless they appear in an exponent, hence we will write $\mathcal{O}(\log n)$; we can change from one base to another multiplying by apropriate factor:

$$\log_c x = \frac{\log_b x}{\log_b c}$$

Asymptotic Notation: Omega and Theta

Other asymptotic notations include Ω (omega) and Θ (zita). Ω defines the set of functions with rate of growth is bounded from below by the rate of growth of the given function:

$$\Omega(f)=\{g:\mathbb{R}^+\to\mathbb{R}^+ \mid \exists n_0\, \exists c>0\, \forall n\geq n_0 : g(n)\geq c\cdot f(n)\}$$

Ω is reflexive and transitive; if $\lim_{n\to\infty} g(n)/f(n) > 0$ then $g = \Omega(f)$. On the other hand, Ω and $\mathcal{O}$ are related as follows: if $f = \mathcal{O}(g)$ then $g = \Omega(f)$, and vice-versa.

Asymptotic Notation: Omega and Theta

We will often say that $\mathcal{O}(f)$ is the class of function that grow no faster than f. Analogously, $\Omega(f)$ is the class of functions that grow at least as fast as f.
Finally,

$$\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$$

is the class of functions with the same rate of growth as f.
Θ is reflexive and transitive, as the other notations, but it is also symmetric: $f = \Theta(g)$ if and only if $g = \Theta(f)$. If $\lim_{n\to\infty} g(n)/f(n) = c$ for some c, $0 < c < \infty$ then $g = \Theta(f)$.

Asymptotic Notation

Additional properties of the asymptotic notations (set inclusions are strict):

1. For any to constants α and β, with $\alpha < \beta$, if f is an increasing function then $\mathcal{O}(f^{\alpha}) \subset \mathcal{O}(f^{\beta})$.
2. For any two constants a and $b > 0$, if f is an increasing function then $\mathcal{O}((\log f)^{a}) \subset \mathcal{O}(f^{b})$.
3. For any constant $c > 1$, if f is an increasing function $\mathcal{O}(f) \subset \mathcal{O}(c^{f})$.

Asymptotic Notation

Conventional operations like sums, substractions, division, etc. can be extended to classes of functions (as defined by asymptotic notations) as follows:

$$A \otimes B = \{h \mid \exists f \in A \wedge \exists g \in B : h = f \otimes g\},$$

where A and B are two sets of functions. Expressions of the form $f \otimes A$, where f a function, denote $\{f\} \otimes A$.
With these conventions we can now write expressions such as $n + \mathcal{O}(\log n)$, $n^{\mathcal{O}(1)}$, or $\Theta(1) + \mathcal{O}(1/n)$.

Asymptotic Notation: Rule of the sums and products

Rule of sums:

$$\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\}).$$

Rule of products:

$$\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g).$$

Similar rules hold for $\mathcal{O}$ and Ω.

Part I

Analysis of Algorithms

Analysis of Iterative Algorithms

1. The cost of an elementary operation (e.g., comparing two integers) is $\Theta(1)$.
2. If the cost of the fragment S_1 is f and that of S_2 is g then the cost of $S_1; S_2$ is $f + g$ (sequential composition).
3. If the cost of S_1 is f, that of S_2 is g and the cost of evaluating the Boolean expression B is h then the worst-case cost of

   if B **then** S_1
   elseS_2
   end if

   is $\mathcal{O}(\max\{f + h, g + h\})$.

Analysis of Iterative Algorithms

4. If the cost of S in the i-th iteration is f_i, the cost of evaluating B is h_i and the number of iterations is g, then the cost of T of

   while B **do**

   $\quad S$

   end while

   is
   $$T(n) = \sum_{i=1}^{i=g(n)} f_i(n) + h_i(n).$$
   If $f = \max\{f_i + h_i\}$ then $T = \mathcal{O}(f \cdot g)$.

Analysis of Iterative Algorithms

```
// example of use:
//   vector<int> my_vector = read_data();
//   cout << "min = " << minimum(v.begin(), v.end()) << endl;

template <class Elem, class Iter>
Elem minimum(Iter beg, Iter end) {
  if (beg == end) throw NullSequenceError;
  Elem min = *beg; ++beg;
  for (Iter curr = beg; curr != end; ++curr)
     if (*curr < min) min = *curr;
  return min;
}
```

Example (Finding the minimum)

If a comparison between two `Elem`'s or an assignment of an `Elem` to a variable (e.g., `min = *curr`) are elementary operations then

1. In the worst-case, the body of the `for` loop takes time $\Theta(1)$; the increment of iterators is also $\Theta(1)$
2. Comparing two iterators is $\Theta(1)$ since we need only to check that they "point" to the same object
3. If the length of the sequence is n ($n = \texttt{end}-\texttt{beg}$) then the loop is executed $n-1$ times. Applying the rule of products we have then

$$F(n) = (n-1)\cdot\Theta(1) = \Theta(n)\cdot\Theta(1) = \Theta(n)$$

Analysis of Iterative Algorithms

```
typedef vector<double> Row;
typedef vector<Row> Matrix;

// we can use the newly defined operator like this: Matrix C = A * B;
// Pre: A[0].size() == B.size()
Matrix operator*(const Matrix& A, const Matrix& B) {
       if (A[0].size() != B.size()) throw IncompatibleMatrixProduct;
       int m = A.size();
       int n = A[0].size();
       int p = B[0].size();
       Matrix C(m, Row(p, 0.0)); // C = m x p matrix initialize to 0.0
       for (int i = 0; i < m; ++i)
          for (int j = 0; j < p; ++j)
             for (int k = 0; k < n; ++k)
                C[i][j] += A[i][k] * B[k][j];
       return C;
}
```

Example (Matrix multiplication)

The algorithm above computes the matrix product of $A = (A_{ij})_{m \times n}$ and $B = (B_{ij})_{n \times p}$ using its definition:

$$C_{ij} = \sum_{k=0}^{n} A_{ik} \cdot B_{kj}$$

Analysis of Iterative Algorithms

Example (Matrix multiplication (cont'd))

1. The body of the innermost **for** loop (on k) has cost $\Theta(1)$. Thus the body of the second **for** loop (on j) is, applying the rule of products, $\Theta(n)$.
2. Similarly the body of the outermost loop (on i) has cost $\Theta(p \cdot n)$.
3. Thus the cost of the three nested loops is $\Theta(m \cdot p \cdot n)$.
4. The other parts of the algorihm have cost $\Theta(m \cdot p)$. By the rule of sums, the overall cost of the algorithm is $\Theta(m \cdot n \cdot p)$.
5. For square matrices, seting $N = m = n = p$, the cost of the algorithm is $\Theta(N^3)$.

Analysis of Iterative Algorithms

```cpp
template <class T, class Comp = std::less<T>>
void insertion_sort(vector<T>& A, Comp smaller) {
  int n = A.size();
  for (int i = 1; i < n; ++i) {
    // put A[i] into its place in A[0..i-1]
    T x = A[i]; int j = i - 1;
    while (j >= 0 and smaller(x, A[j])) {
      A[j+1] = A[j];
      --j;
    };
    A[j] = x;
  }
}
```

Example (Insertion sort)

Insertion sort is one of the so-called *elementary sort algorithms*. It is very easy to understand and to program. Its running time for any instance is both $\Omega(n)$ and $\mathcal{O}(n^2)$. In particular, the best-case is $\Theta(n)$ and the worst-case is $\Theta(n^2)$.

Analysis of Iterative Algorithms

```
template <class T, class Comp = std::less<T>>
void insertion_sort(vector<T>& A, Comp smaller) {
  int n = A.size();
  for (int i = 1; i < n; ++i) {
    // put A[i] into its place in A[0..i-1]
    T x = A[i]; int j = i - 1;
    while (j >= 0 and smaller(x, A[j])) {
      A[j+1] = A[j];
      --j;
    };
    A[j] = x;
  }
}
```

Example (Insertion sort (cont'd))

1. The `while` can make any number of iterations from 0 (when the vector is already sorted) to i (when the vector is in reverse order). Its cost is $\Theta(i)$ assuming that the cost of the comparison `smaller` is $\Theta(1)$, and the assignment between elements of class `T` takes also constant time.
2. Thus the cost of the `for` loop in the worst-case is

$$\sum_{i=1}^{n-1} \Theta(i) = \Theta\left(\sum_{i=1}^{n-1} i\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2)$$

Analysis of Iterative Algorithms

```
template <class T, class Comp = std::less<T>>
void insertion_sort(vector<T>& A, Comp smaller) {
  int n = A.size();
  for (int i = 1; i < n; ++i) {
    // put A[i] into its place in A[0..i-1]
    T x = A[i]; int j = i - 1;
    while (j >= 0 and smaller(x, A[j])) {
      A[j+1] = A[j];
      --j;
    };
    A[j] = x;
  }
}
```

Example (Insertion sort (cont'd))

3. A quick upper bound follows by observing that the cost of the `while` loop is $\mathcal{O}(i) = \mathcal{O}(n)$, hence the cost of the algorithm is $\mathcal{O}(n^2)$.
4. The cost of the `for` loop is $\Theta(n)$ in the best case, since the cost of the i-th iteration in the best case is $\Theta(1)$.
5. The average cost of the algorithm is also $\Theta(n^2)$, assuming each of the $n!$ possible initial orderings of the vector is equally likely. The inner `while` loop will perform, on average, $\approx i/2$ iterations when inserting $A[i]$.

Part I

Analysis of Algorithms

Analysis of Recursive Algorithms

The cost $T(n)$ (worst-, best-, average-case) of a recursive algorithm satisfies a recurrence: an equation where T appears in both sides, with $T(n)$ depending on $T(k)$ for one or more values $k < n$. Recurrences appear often in one of the two following forms:

$$T(n) = a \cdot T(n-c) + f(n),$$
$$T(n) = a \cdot T(n/b) + f(n).$$

First correspond to algorithms where the non-recursive part has cost $f(n)$ and they make a recursive calls on inputs of size $n - c$, for some constant c.
Second corresponds to algorithm with non-recursive cost $f(n)$ making a recursive calls on inputs of size (approx.) n/b, where $b > 1$.

Analysis of Recursive Algorithms

Theorem

Let $T(n)$ satisfy the recurrence

$$T(n) = \begin{cases} g(n) & \text{if } 0 \leq n < n_0 \\ a \cdot T(n-c) + f(n) & \text{if } n \geq n_0, \end{cases}$$

where n_0 is a constant, $c \geq 1$, $g(n)$ is an arbitrary function, and $f(n) = \Theta(n^k)$ for some constant $k \geq 0$.
Then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < 1 \\ \Theta(n^{k+1}) & \text{if } a = 1 \\ \Theta(a^{n/c}) & \text{if } a > 1. \end{cases}$$

Analysis of Recursive Algorithms

Theorem

Let $T(n)$ satisfy the recurrence

$$T(n) = \begin{cases} g(n) & \text{if } 0 \leq n < n_0 \\ a \cdot T(n/b) + f(n) & \text{if } n \geq n_0, \end{cases}$$

where $a \geq 1$, $b > 1$ and n_0 constants, $g(n)$ is an arbitrary function and $f(n) = \Theta(n^k)$ for some constant $k \geq 0$.
Let $\alpha = \log_b a$. Then

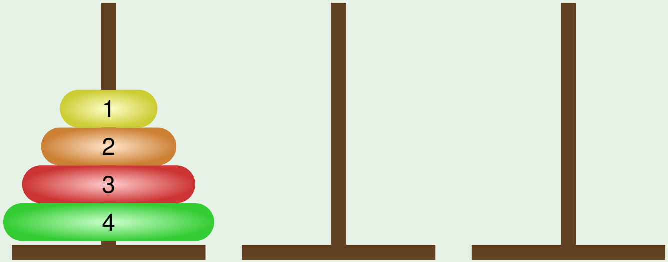
$$T(n) = \begin{cases} \Theta(n^k) & \text{if } \alpha < k \\ \Theta(n^k \log n) & \text{if } \alpha = k \\ \Theta(n^\alpha) & \text{if } \alpha > k. \end{cases}$$

The conditions $\alpha < k$, $\alpha = k$ and $\alpha > k$ are equivalent to $a < b^k$, $a = b^k$ and $a > b^k$, respectively.

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

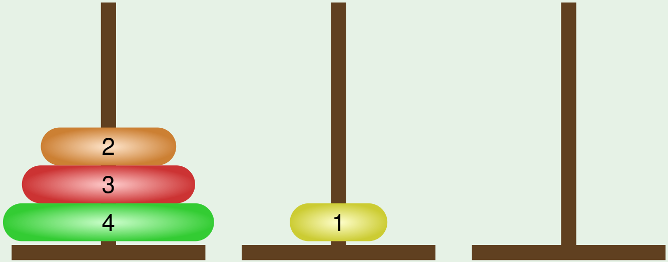


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

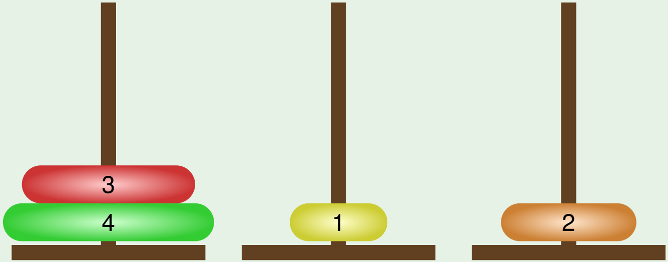


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

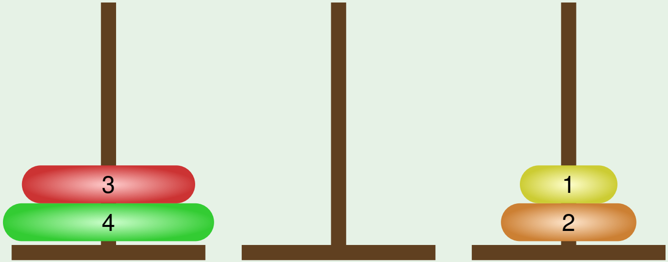


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

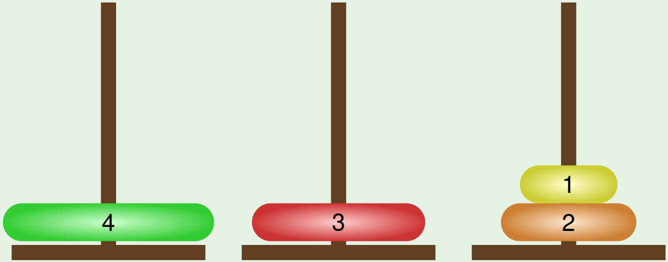


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

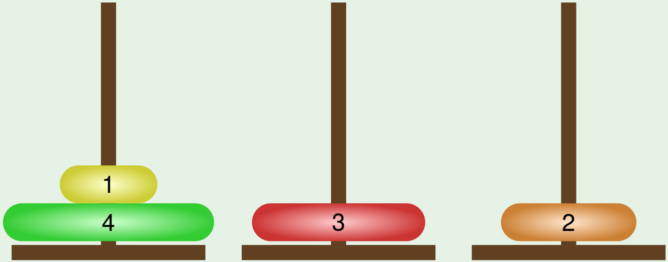


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

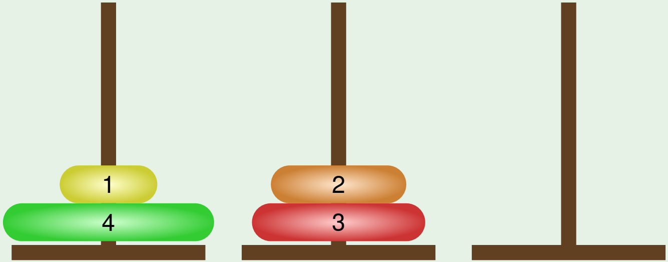


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

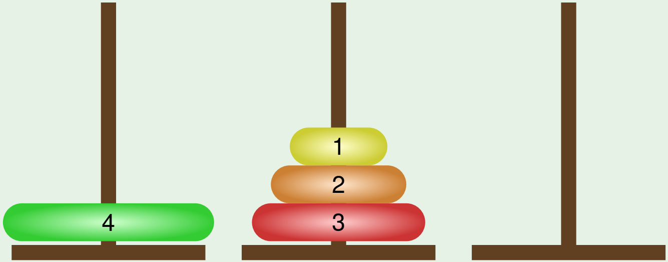


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

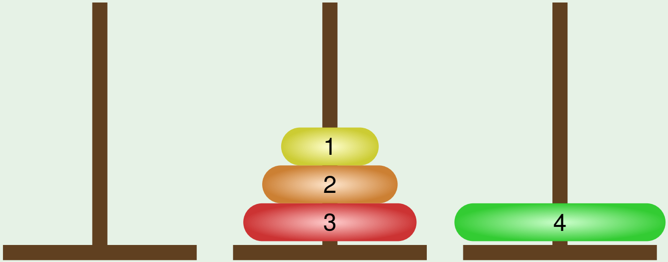


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

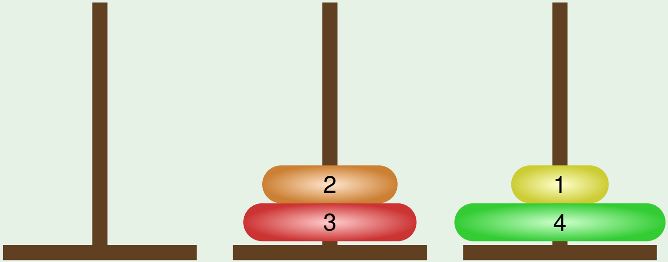


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

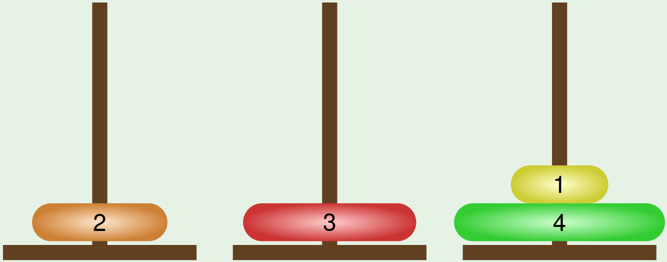


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

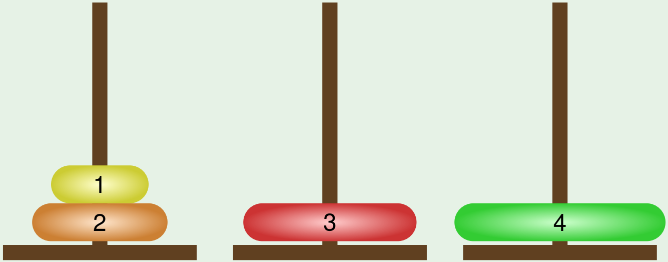


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

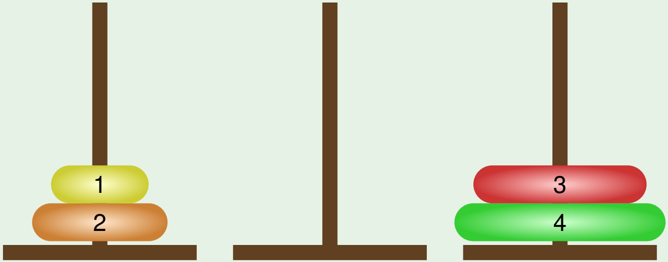


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

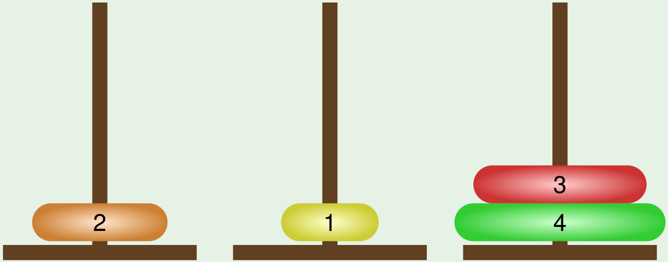


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

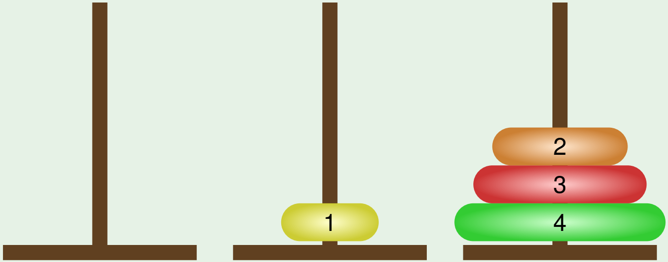


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

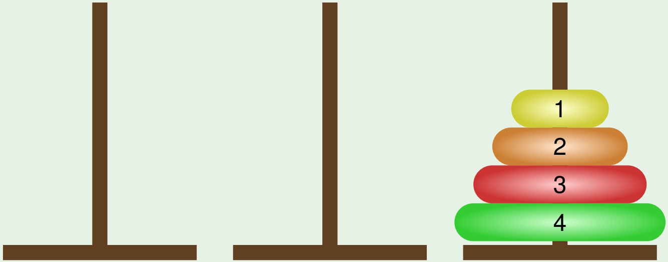


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.

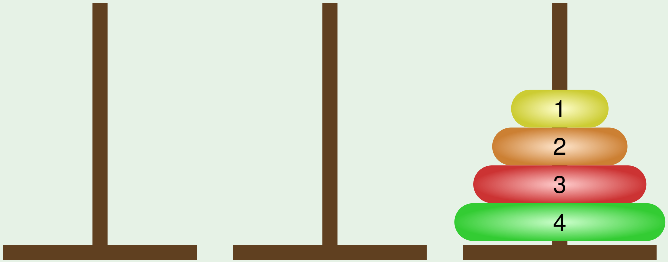


Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have n disks of decreasing diameters with a hole in their center and three poles A, B and C. The n disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



Source: B. Damman, M. Hofmann (TEXample.net)

Analysis of Recursive Algorithms

```
typedef char pole;

// Initial call: hanoi(n,'A', 'B', 'C');

void hanoi(int n, pole org, pole aux, pole dst) {
     if (n == 1)
        cout << "Move from " << org << " to " << dst << endl;
     else {
        hanoi(n - 1, org, dst, aux);
        // move the largest disk
        cout << "Move from " << org << " to " << dst << endl;
        hanoi(n - 1, aux, org, dst);
     }
}
```

Example (Towers of Hanoi (cont'd))

The cost $f(n)$ of the non-recursive part is $\Theta(1)$, and for $n \leq n_0 = 1$ the cost is also $\Theta(1)$. The recurrence that describes the cost $H(n)$ of `hanoi` is

$$H(n) = 2H(n-1) + \Theta(1), \quad \text{if } n > 1$$

and $H(1) = \Theta(1)$. Applying the theorem for "substractive" recurrences with $a = 2$ and $c = 1$ we get $H(n) = \Theta(2^n)$. Indeed, it can be easily shown that exactly $M_n = 2^n - 1$ single moves are necessary (and sufficient) to move the n disks from A to C.

Analysis of Recursive Algorithms

Example (Powers)

Given three positive integers x, y and $m > 1$, compute $x^y \bmod m$.

- For any y_1, y_2 such that $y_1 + y_2 = y$,

  $$x^y \mod m = ((x^{y_1} \bmod m) \cdot (x^{y_2} \bmod m)) \bmod m,$$

  that is, we can take $\bmod m$ in intermediate steps to avoid dealing with very large numbers
- If we compute x^y, either iteratively or recursively, using the identity $x^y = x \cdot x^{y-1}$ for $y > 0$, we end up with an algorithm making $\Theta(y)$ products $\Rightarrow$ exponential in the size of the input (we need $\approx \log_2(x) + \log_2(y) + \log_2 m$ bits)

Analysis of Recursive Algorithms

Example (Powers)

Given three positive integers x, y and $m > 1$, compute $x^y \bmod m$.

- For any y_1, y_2 such that $y_1 + y_2 = y$,

  $$x^y \mod m = ((x^{y_1} \bmod m) \cdot (x^{y_2} \bmod m)) \bmod m,$$

  that is, we can take $\bmod m$ in intermediate steps to avoid dealing with very large numbers
- If we compute x^y, either iteratively or recursively, using the identity $x^y = x \cdot x^{y-1}$ for $y > 0$, we end up with an algorithm making $\Theta(y)$ products $\Rightarrow$ exponential in the size of the input (we need $\approx \log_2(x) + \log_2(y) + \log_2 m$ bits)

Analysis of Recursive Algorithms

```
int power(int x, int y, int m) {
    if (y == 0) return 1;
    int p = power(x, y/2, m);
    if (y % 2 == 0)
       return (p * p) % m;
    else
       return (((p * p) % m) * x) % m;
}
```

Example (Powers (cont'd))

The cost $P(y)$ (measured as the number of arithmetical operations) of `power` satisfies the following recurrence[a]

$$P(y) = P(y/2) + \Theta(1),$$

and $P(0) = 0$; we can solve the recurrence using the theorem for "divisive" recurrences with $k = 0$, $a = 1$ and $b = 2$; since $\alpha = \log_2 1 = 0 = k$ the solution is $P(y) = \Theta(\log y) \Rightarrow$ linear number of products in the size of the input

[a]Ceilings and floors can be safely ignored; the actual recurrence is $P(y) = P(\lceil y/2 \rceil) + \Theta(1)$.