Advanced Algorithmics and Artificial Intelligence

Olivier Baudon

Université de Bordeaux

March 2021

Objectives

- Deepen your understanding of programs
- Provide knowledge of widely-used algorithms
- Give some insight into theoretical issues of Computer Science

Prerequisites

- Mathematics: elementary analysis and linear algebra
- Programming practice in C and Java: loops, recursive functions, classes, generics
- Arrays: searching and sorting algorithms
- Data structures: strings, lists, stacks, queues, trees

Outline

1. Design and Analysis of Algorithms
2. Advanced Data Structures
3. Graph Algorithms
4. AI

Organization

Lectures

- Olivier Baudon, UB
- `olivier.baudon@labri.fr`

- 10 $\times$ 2h - lectures

Organization

Lectures

- Olivier Baudon, UB
- `olivier.baudon@labri.fr`
- 10 $\times$ 2h - lectures

Training

- Quan Thanh Tho, HCMUT
- `qttho@cse.hcmut.edu.vn`
- 9 $\times$ 3h-training sessions

Organization

Lectures

- Olivier Baudon, UB
- `olivier.baudon@labri.fr`
- 10 × 2h - lectures
- 3h final exam (2/3)

Training

- Quan Thanh Tho, HCMUT
- `qttho@cse.hcmut.edu.vn`
- 9 × 3h-training sessions
- exams and/or projects (1/3)

Bibliography

T. J. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein.
Introduction to Algorithms.
The MIT Press, third edition, 2009.

Shimon Even.
Graph Algorithms.
Computer Science Press, 1979.

S. Russel and P. Norvig.
Artificial Intelligence, a modern approach.
Prentice Hall Ed, 2009.

Schedule

- Design and Analysis of Algorithms
- Data Structures
- Graph Algorithms
- AI

- Elementary Analysis

- Elementary Analysis
  - Classification of Algorithms
  - Design Methods
  - Analysis of Algorithms

Classification of Algorithms

- **Problem types**
  computing, sorting, searching, optimization...
- **Data types**
  numbers, strings, lists, trees, graphs...
- **Implementation**
  iterative algorithms, recursive algorithms...
- **Design methods**
  divide-and-conquer, dynamic programming, greedy algorithms...

- Elementary Analysis
  - Classification of Algorithms
  - Design Methods
  - Analysis of Algorithms

```
int search(int x, sortedArray A of size n) {
    int start = 0, end = n-1, middle;
    while (start <= end) {
         middle = (start + end)/2;
         if (x == A[middle])
            return FOUND ;
         if (x < A[middle])
            end = middle - 1; // restrict to lower half
         else
            start = middle + 1; // restrict to upper half
     }
     return NOT_FOUND ;
}
```

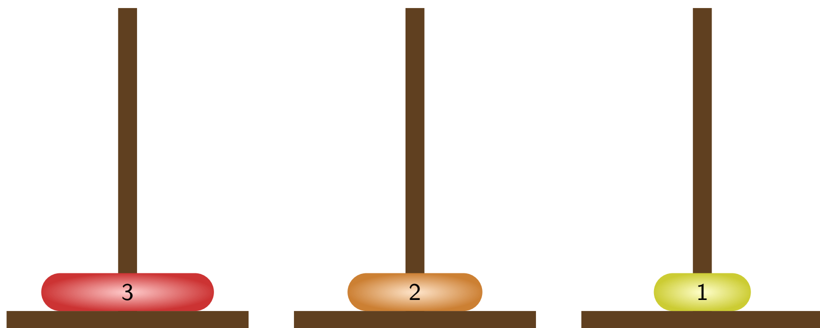
Divide-and-conquer: The Towers of Hanoi

Authors: Martin Hofmann and Berteun Damman

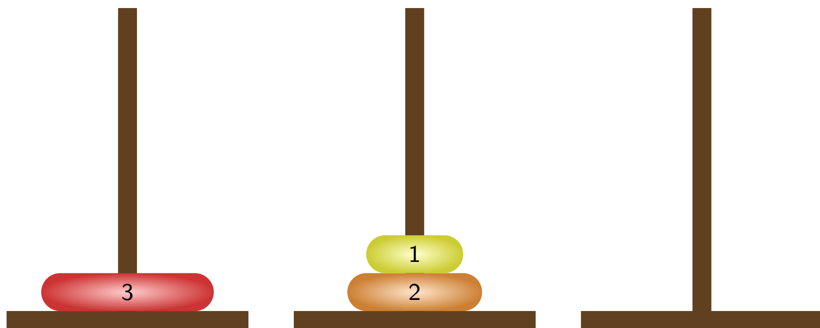
Divide-and-conquer: The Towers of Hanoi

Moved disc from pole 1 to pole 3.

Divide-and-conquer: The Towers of Hanoi

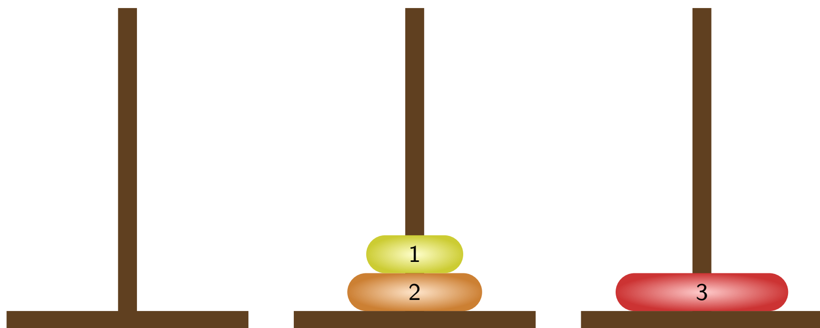


Moved disc from pole 1 to pole 2.



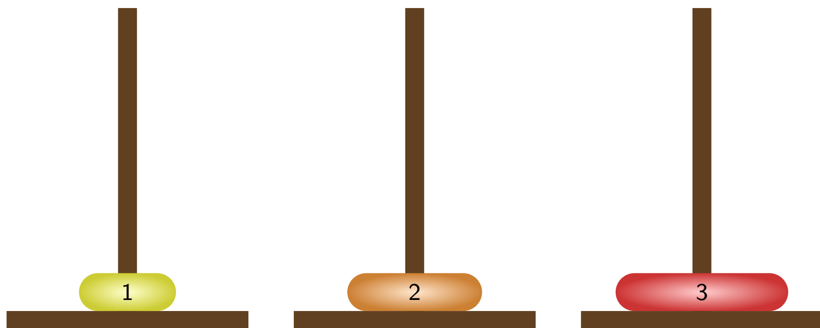
Moved disc from pole 3 to pole 2.

Divide-and-conquer: The Towers of Hanoi



Moved disc from pole 1 to pole 3.

Divide-and-conquer: The Towers of Hanoi



Moved disc from pole 2 to pole 1.

Divide-and-conquer: The Towers of Hanoi

Moved disc from pole 2 to pole 3.

Divide-and-conquer: The Towers of Hanoi

Moved disc from pole 1 to pole 3.

Divide-and-conquer: The Towers of Hanoi

```
void hanoi(int n,int from,int to,int other) {
  if (n > 1)
      hanoi(n-1,from,other,to);

  printf("Move disc from pole \%d to pole \%d.\n",from,to);

  if (n > 1)
      hanoi(n-1,other,to,from);
}
```

Divide-and-Conquer: MergeSort

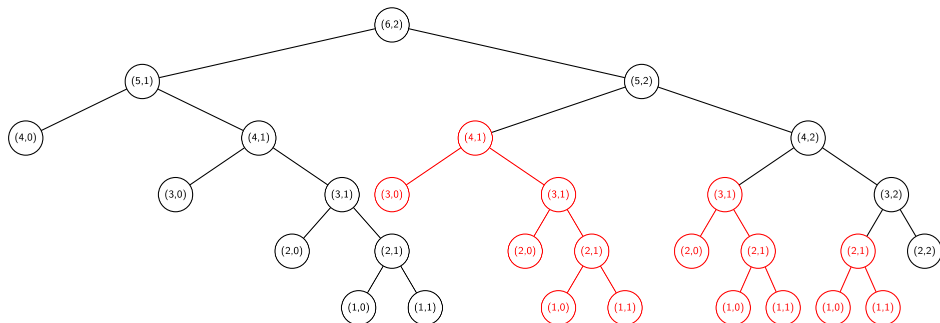
```
sort array A of size n:
   if (n ⩾ 2) {
      sort the first half of A;
      sort the second half of A;
      merge the two sorted halves;
   }
```

n \ p	0	1	2	3	4	5	6
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

```
binomial(n, p) = binomial(n-1, p-1) + binomial(n-1, p)
```

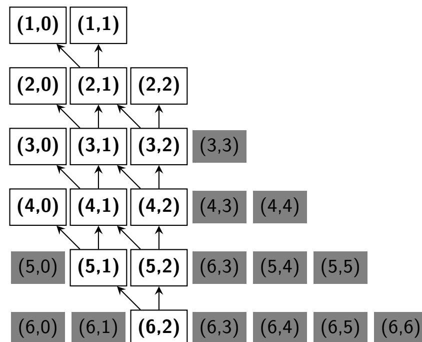
Recursive Computation of Binomial Coefficients

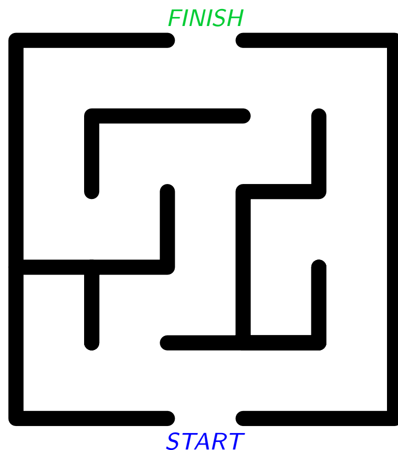
```
int binomial(int n, int p) {
  if (p == 0 || p == n)
      return 1;
  return binomial(n-1, p-1) + binomial(n-1, p);
}
```



Dynamic Programming: Binomial Coefficients

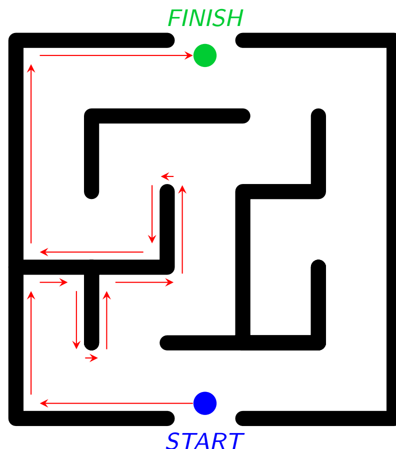
```
int binomial(int n, int p) {
    if (table[n][p] > 0)
        return table[n][p];
    if (p == 0 || p == n)
        return table[n][p] = 1;
    return table[n][p] = binomial(n-1, p-1) + binomial(n-1, p);
}
```



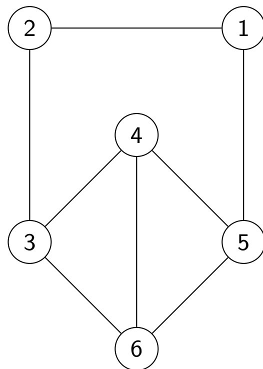


Algorithm: follow the wall on your left.

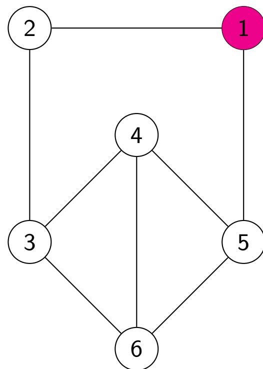
Greedy Algorithms: Maze



Algorithm: follow the wall on your left.

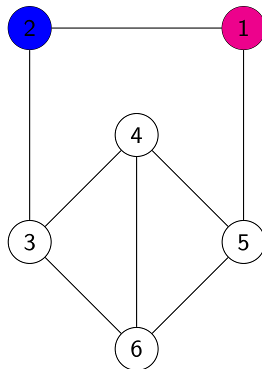


Algorithm: colour each node with the first available colour.



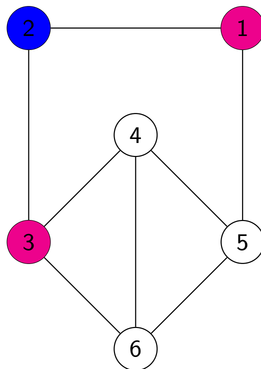
Algorithm: colour each node with the first available colour.

Greedy Algorithms: Graph Colouring

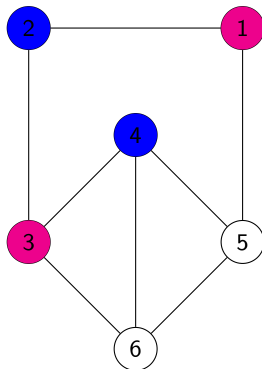


Algorithm: colour each node with the first available colour.

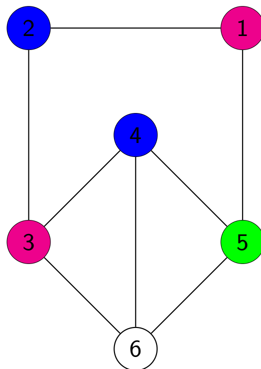
Greedy Algorithms: Graph Colouring



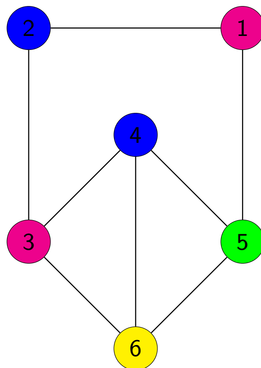
Algorithm: colour each node with the first available colour.



Algorithm: colour each node with the first available colour.



Algorithm: colour each node with the first available colour.



Algorithm: colour each node with the first available colour.

- Elementary Analysis
  - Classification of Algorithms
  - Design Methods
  - Analysis of Algorithms

What to analyze?

A "good" algorithm should be

- effective
- efficient
- economic

What to analyze?

A "good" algorithm should be

- effective $\rightarrow$ *prove correctness*
- efficient
- economic

What to analyze?

A "good" algorithm should be

- effective $\rightarrow$ *prove correctness*
- efficient $\rightarrow$ *estimate run-time*
- economic

What to analyze?

A “good” algorithm should be

- effective $\rightarrow$ *prove correctness*
- efficient $\rightarrow$ *estimate run-time*
- economic $\rightarrow$ *estimate memory needed,*
  but also cost of work and maintenance

Proving Correctness: Dichotomic Search

```
int search(int x, sortedArray A of size n) {
   int start = 0, end = n-1, middle;
   while (start <= end) {
        middle = (start + end) / 2;
        if (x == A[middle])
           return FOUND ;
        if (x < A[middle])
           end = middle - 1; // restrict to lower half
        else
           start = middle + 1; // restrict to upper half
    }
    return NOT_FOUND ;
}
```

- Termination
  `"end - start"` strictly decreases between successive executions of the `while` loop
- Partial correctness follows from **loop invariant**
  - `A[i]` $<$ `x` if `i` $<$ `start`
  - `A[i]` $>$ `x` if `i` $>$ `end`

Proving Correctness: Merge Sort

```
sort array A of size n:
   if (n ⩾ 2) {
      sort the first half of A;
      sort the second half of A;
      merge the two sorted halves;
   }
```

- Termination
  the size of the array to sort strictly decreases in the recursive calls
- Partial correctness
  - if $n < 2$: obvious
  - otherwise, can be deduced from the correctness of the recursive calls

Proving Correctness: The Towers of Hanoi

```
void hanoi(int n, int from, int to, int other) {
  if (n > 1)
      hanoi(n-1, from, other, to);

  printf("Move disc from pole \%d to pole \%d.\n",from,to);

  if (n > 1)
      hanoi(n-1, other, to, from);
}
```

- Termination
  The argument `n` strictly decreases in the recursive calls.
- Partial correctness
  - obvious if `n` $\leq$ 1
  - otherwise, can be deduced from the correctness of the recursive calls, slightly modifying the problem (poles may contain other discs)

Proving Correctness: Binomial Coefficients

```
int binomial(int n, int p) {
    if (table[n][p] > 0)
        return table[n][p];
    if (p == 0 || p == n)
        return table[n][p] = 1;
    return table[n][p] = binomial(n-1,p-1) + binomial(n-1,p);
}
```

- Termination
  The sum `n+p` of the arguments strictly decreases in the recursive calls.

- Partial correctness
  We prove by induction that `binomial(n,p)` returns $\binom{n}{p}$, and that at each step of the algorithm, if `table[i][j]` $>$ 0, then `table[i][j]` is equal to $\binom{i}{j}$
  - obvious if `table[n][p]` $\neq$ 0 and either `p = 0` or `p = n`
  - otherwise, can be deduced from the correctness of the recursive calls

Asymptotic Complexity: big O, Omega and Theta

T_n: run-time of an algorithm, depending on n: size of the input, is

- $\mathbf{O(f(n))}$ "big O": asymptotic upper bound
  if for some constant $c > 0$, $T_n \leqslant c \times f(n)$ when n is large enough
- $\mathbf{\Omega(f(n))}$ "Omega": asymptotic lower bound
  if for some constant $c > 0$, $T_n \geqslant c \times f(n)$ when n is large enough
- $\mathbf{\Theta(f(n))}$ "Theta": asymptotic equivalent (up to a multiplying factor)
  if T_n is both $O(f(n))$ and $\Omega(f(n))$

Common Asymptotic Behaviours: Names

- **Linear**: $\mathbf{\Theta(n)}$
- **Quadratic**: $\mathbf{\Theta(n^2)}$
- **Logarithmic**: $\mathbf{\Theta(\log n)}$
- **Polynomial**: $\mathbf{\Theta(n^k)}$, for some $k > 0$
- **Exponential**: $\mathbf{\Theta(k^n)}$, for some $k > 1$
- **Factorial**: $\mathbf{\Theta(n!)}$

Common Asymptotic Behaviours: Figures

linear n	**quadratic** n^2	**logarithmic** $\lceil \log_2 n \rceil$	**exponential** 2^n	**factorial** $n!$
1	1	0	2	1
2	4	0	4	2
5	25	3	32	120
10	100	4	$1,024$	$3,628,800$
100	$10,000$	7	1.2×10^{30}	9.3×10^{157}
$1,000$	$1,000,000$	10	10^{301}	4×10^{2567}
$10,000$	$100,000,000$	14	1.9×10^{3010}	2.8×10^{35659}

Complexity of Iterative Algorithms: Dichotomic Search

```
int search(int x, sortedArray A of size n) {
    int start = 0, end = n-1, middle;
    while (start <= end) {
        middle = (start + end)/2 ;
        if (x == A[middle])
           return FOUND;
        if (x < A[middle])
           end = middle - 1; // restrict to lower half
        else
           start = middle + 1; // restrict to upper half
     }
     return NOT_FOUND ;
}
```

Complexity: $\mathbf{O}(\log_2 \mathbf{n})$
(at most $\lceil \log_2 n \rceil$ executions of the `while` loop, and each execution is $O(1)$)

```
search element x in sorted array A of size n:
   determine in which half x is to be searched for;
   search x in this half;
```

$$T_n = c + T_{\frac{n}{2}} \textit{ is } \mathbf{O}(\log_2 \mathbf{n})$$

```
search element x in unsorted array A of size n:
   search x in the first half of A;
   if (not found)
      search x in the second half of A;
```

$$T_n = c + 2 \times T_{\frac{n}{2}} \textit{ is } \mathbf{O}(\mathbf{n}) \textit{ but not } \mathbf{O}(\log_2 \mathbf{n})$$

Complexity of Recursive Algorithms: MergeSort

```
sort array A of size n:
   if (n ⩾ 2) {
      sort the first half of A;
      sort the second half of A;
      merge the two sorted halves;
   }
```

$$
\begin{aligned}
T_n &= 2 \times T_{\frac{n}{2}} + c \times n \\
&= 2(2 \times T_{\frac{n}{4}} + c \times \tfrac{n}{2}) + c \times n \\
&= \cdots \\
&= c \times (n + 2\frac{n}{2} + \cdots + 2^k \frac{n}{2^k}) \textit{ where } k = \log_2 n \\
&= c \times n \times \log_2 n
\end{aligned}
$$

is $\mathbf{\Theta(n \log_2 n)}$

Complexity of Recursive Algorithms: QuickSort

```
sort array A[first..last] {
   if (first >= last) return;
   pivot = A[first];
   i = first; j = last;
   while (i <= j) {
       if (A[j] > pivot) j--;
       else { swap(A[i],A[j]); i++; }
   }
   sort A[first..j];
   sort A[i..last];
}
```

- Worst case: $T_n = c \times n + T_{n-1} = c \times (n + (n-1) + \cdots + 1)$ is $\mathbf{O(n^2)}$
- Best case: $T_n = c \times n + 2T_{\frac{n}{2}}$ is $\mathbf{O(n \log_2 n)}$