COMP 3761: Algorithm Analysis and Design

Shidong Shan

BCIT

Learning outcomes

- Describe the relationship between recursion/iteration and induction
- Evaluate the effect of recursion on space complexity
- Recognize algorithms as recursive or iterative
- Convert between recursive and iterative solutions
- Draw a recursion tree, and relate the depth to the number of recursive calls, and the size of the runtime stack
- Solve basic recurrence equations.

Recurrence relations

- A function calls itself (usually on a smaller instance of the problem)
- Correctness of a recurrence relation can be proved by mathematical induction
- Recursion is an extremely useful and powerful design technique
- e.g., used in Divide and Conquer strategy.

Proof by mathematical induction

- A style of proof to establish the truth of a given statement over an infinite set (usually natural numbers n)
- Basic steps:
  1. **base case**: show that the simplest case is true
  2. **inductive hypothesis**: assume the statement holds for some arbitrary element $k < n$
  3. **inductive step**: show that the statement holds for the successor $k + 1$
  4. **conclusion**.
- Exercise:
  Use mathematical induction to show

$$F(n) = F(n-1) * n, \quad F(n) = n!$$

Iterative vs. Recursive Factorial Function $n!$

- Iterative version
  if $n = 0$, return 1;
  else return $1 * 2 * 3 * ... * (n - 1) * n$;

- Recursive version
  if $n = 0$, return 1;
  return $n * (n - 1)!$;

Analyzing recursive algorithms

1. Decide on a parameter for input size.
2. Identify the algorithm's basic operation.
3. Check if the number of times the basic operation is executed varies on different inputs of the same size.
4. Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic operation is executed.
5. Solve the recurrence or establish the order of growth of its solution.

Analysis of Recursive Factorial

- Input size:
- Basic operation:
- Recurrence relation:

Backward substitution

- To solve recurrence uniquely, we need to specify:
  - a recurrence relation
  - initial condition(s)
- Example: Given $M(n) = M(n-1) + 1, M(0) = 0$, solve for $M(n)$.

Thinking recursively

- If the instance is simple, solve the instance directly
- else
  - divide instance into new sub-instances $i_1, i_2, \ldots, i_k$
  - call the function itself to solve $i_1, i_2, \ldots, i_k$
  - re-assemble the subproblem solutions to solve the entire problem.

Example 1: Counting # of binary bits in a number n

- iterative version: BinIter(n)

  count $\leftarrow 1$
  while $n > 1$
  count $\leftarrow$ count $+ 1$;
  $n \leftarrow \lfloor n/2 \rfloor$
  return count

- recursive version: BinRec(n):

  if $n = 1$, return 1
  else return BinRec $(\lfloor n/2 \rfloor) + 1$

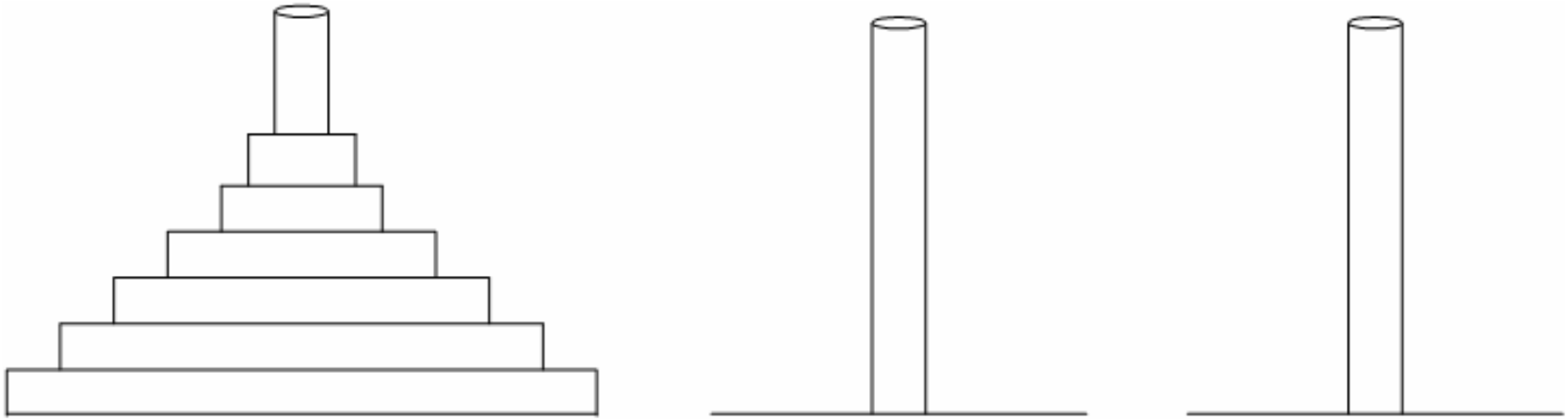
Smoothness rule

Example: Given $A(1) = 0$, solve $A(n) = A(\lfloor n/2 \rfloor) + 1$.

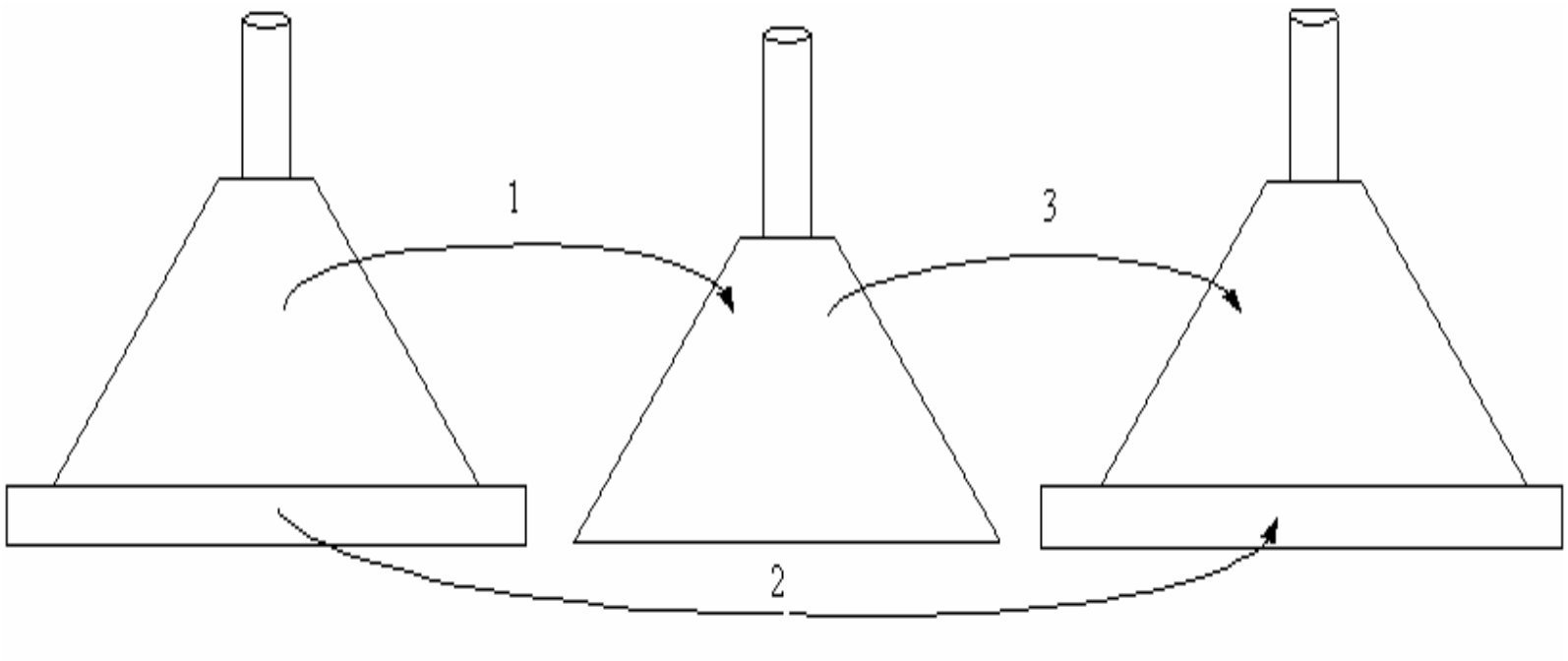
- direct backward substitutions do not work when $n \neq 2^k$.
- under certain assumptions, if $t(n) \in O(g(n))$ for $n = 2^k$, then $t(n) \in O(g(n))$ for all n. (see Appendix B)
- solve the recurrence for $n = 2^k$, using backward substitutions.

Example 2: Tower of Hanoi Puzzle

- Problem: n disks of different sizes and three pegs
- Initially all disks are on the Peg 1 in order of size
- Goal: to move all disks to Peg 3 using Peg 2 as an auxiliary
- Constraint: move one disk at a time, cannot put a larger one on top of a smaller one.



Recursive solution



Analysis of recursive Hanoi Towers

- Input size:
- Basic operation:
- Recurrence relation:
- solve the recurrence relation:

$$M(n) = 2M(n-1) + 1, \quad \text{for } n > 1, \quad M(1) = 1.$$

Fibonacci numbers

- $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$
- The Fibonacci recurrence:

$$\begin{aligned} F(n) &= F(n-1) + F(n-2) \\ F(0) &= 0 \\ F(1) &= 1 \end{aligned}$$

- Algorithm: $F(n)$
  if $n \leq 1$ return n
  else return $F(n-1) + F(n-2)$

Trace the Fibonacci function calls

- The Fibonacci recurrence looks very simple
- not trivial to solve this recurrence
- difficult to trace the function calls
- each recursive step involves two recursive calls to the function

Class exercise:

Trace the calls made by $F(5)$ using a recursion tree:

Space complexity of recursion

- each function call generates an *activation record* holds the values of arguments and local variables
- *run-time stack* is the memory space for the activation records
- The height of the recursion tree is the maximum number of activation records on the stack at any given time
- If height $>$ maximum activation records allowed, **stack overflow** occurs, the program will crash
- Use the height of the recursion tree to determine how much memory is required for the recursive function call
- The number of nodes in the recursion tree represents the total number of function calls, which determine the running time.

Exercise

Write an iterative function (function uses a **for** or **while** loop) to compute the *n*th Fibonacci number.

Decrease by one

- $f(n)$ = time required to reduce an instance to a smaller one and to extend the solution of the smaller instance to a solution of the larger instance.
- recurrence equation:

$$T(n) = T(n-1) + f(n)$$

- solve by backward substitutions
- solution:

$$T(n) = T(0) + \Sigma_{j=1}^{n} f(j)$$

Note: $f(n)$ determines the efficiency class of $T(n)$.

Decrease by a constant factor

- recurrence equation:

$$T(n) = T(n/b) + f(n)$$

- use smoothness rule to solve for $n = b^k$
- solution:

$$T(b^k) = T(1) + \Sigma_{j=1}^{k} f(b^j)$$

Divide and conquer

- recurrence equation:

$$T(n) = aT(n/b) + f(n)$$

- use smoothness rule to solve for $n = b^k$
- solution:

$$T(b^k) = a^k[T(1) + \Sigma_{j=1}^{k} f(b^j)/a^j]$$

- See more on this later in the course.

Exercises 2.4 #1

Solve the following recurrence relations

- $x(n) = x(n-1) + 5 \text{for } n > 1, \quad x(1) = 0$
- $x(n) = 3x(n-1) \text{for } n > 1, \quad x(1) = 4$
- $x(n) = x(n/2) + n \text{for } n > 1, \quad x(1) = 1$