Algorithms and Data Structures

Recursion

(c) Marcin Sydow

Topics covered by this lecture:

- Recursion: Introduction
  - Fibonacci numbers, Hanoi Towers, ...
- Linear 2nd-order Equations
- Important 3 cases of recursive equations (with proofs)
- QuickSort Average Complexity (Proof)
- Master Theorem

Recursion

e.g.: $n! = (n-1)!n$

- Mathematics: recurrent formula or definition
- Programming: function that calls itself
- Algorithms: reduction of an instance of a problem to a smaller instance of the same problem ("divide and conquer")

Warning: should be well founded on the trivial case:

e.g.: $0! = 1$

Example

base:

Fibonacci(0) = 0
Fibonacci(1) = 1

step:

Fibonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)

0,1,1,2,3,5,8,13,21,34,...

Note: some elder definitions define the Fibonacci sequence as starting with 1 (i.e. 1,1,2,3,5,...) omitting the leading 0 term, but in this course we use the more common definition starting with 0.

Recursion as an Algorithmic Tool

A powerful method for algorithm design

It has positive and negative aspects, though:

- (positive) very compact representation of an algorithm
- (negative) recursion implicitly costs additional memory for keeping the recursion stack

Example

What happens on your machine when you call the following function for n=100000?

```
triangleNumber(n){
 if (n > 0) return triangleNumber(n-1) + n
 else return 0
}
```

Iterative version of the above algorithm would not cause any problems on any reasonable machine.

In final implementation, recursion should be avoided or translated to iterations whenever possible (not always possible), due to the additional memory cost for keeping the recursion stack (that could be fatal...)

Hanoi Towers

A riddle:

Three vertical sticks A, B and C. On stick A, stack of n rings, each of different size, always smaller one lies on a bigger one. Move all rings one by one from A to C, respecting the following rule "bigger ring cannot lie on a smaller one" (it is possible to use the helper stick B)

Hanoi Towers - number of moves

How many moves are needed for moving n rings?
(hanoi(n) = ?)

This task can be easily solved with recurrent approach.

Hanoi Towers - number of moves

How many moves are needed for moving n rings?
(hanoi(n) = ?)

This task can be easily solved with recurrent approach.

If we have 1 ring, we need only 1 move (A -> C). For more rings, if we know how to move n-1 top rings to B, then we need to move the largest ring to C, and finally all rings from B to C.

Hanoi Towers - number of moves

How many moves are needed for moving n rings?
(hanoi(n) = ?)

This task can be easily solved with recurrent approach.

If we have 1 ring, we need only 1 move (A -> C). For more rings, if we know how to move n-1 top rings to B, then we need to move the largest ring to C, and finally all rings from B to C.

Thus, we obtain the following recurrent equations:
base:
hanoi(1) = 1

step:
hanoi(n) = hanoi(n-1) + 1 + hanoi(n-1) = 2*hanoi(n-1) + 1

Solving Recurrent Equations

2 general methods:

1. expanding to sum
2. generating functions

illustration of the method 1:

$$hanoi(n) = 2 * hanoi(n-1) + 1 =$$
$$2 * (2 * hanoi(n-2) + 1) + 1 = ... = \sum_{i}^{n-1} 2^i = 2^n - 1$$

(method 2 is outside of the scope of this course)

A general method for solving 2nd order linear recurrent equations

Assume the following recurrent equation:

$s_n = as_{n-1} + bs_{n-2}$

Then, solve the following *characteristic equation*:
$x^2 - ax - b = 0$.

1. single solution r: $s_n = c_1 r^n + c_2 n r^n$
2. two solutions r_1, r_2: $s_n = c_1 r_1^n + c_2 r_2^n$

for some constants c_1, c_2
(that can be found by substituting $n = 0$ and $n = 1$)

Illustration of the Theorem

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)
Fibonacci(0) = Fibonacci(1) = 1

Fibonacci(50) = ?

Illustration of the Theorem

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)
Fibonacci(0) = Fibonacci(1) = 1

Fibonacci(50) = ?

From the last theorem it can be shown that:

$$Fibonacci(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

(the "Binet's" formula)

(BTW: it is incredible, but this is always a natural number!)

Illustration of the Theorem

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)
Fibonacci(0) = Fibonacci(1) = 1

Fibonacci(50) = ?

From the last theorem it can be shown that:

$$Fibonacci(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

(the "Binet's" formula)

(BTW: it is incredible, but this is always a natural number!)

Lets compute Fibonacci(50)...

Illustration of the Theorem

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)
Fibonacci(0) = Fibonacci(1) = 1

Fibonacci(50) = ?

From the last theorem it can be shown that:

$$Fibonacci(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

(the "Binet's" formula)

(BTW: it is incredible, but this is always a natural number!)

Lets compute Fibonacci(50)... over 12 billion!

Illustration of the Theorem

Finonacci(n+1) = Fibonacci(n) + Fibonacci(n-1)
Fibonacci(0) = Fibonacci(1) = 1

Fibonacci(50) = ?

From the last theorem it can be shown that:

$$Fibonacci(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

(the "Binet's" formula)

(BTW: it is incredible, but this is always a natural number!)

Lets compute Fibonacci(50)... over 12 billion!
more precisely: 12 586 269 025

Other Important Special Cases

Some types of recurrent equations are quite frequently encountered in algorithmics.

I.e. time complexity function of some important algorithms is in the form of a recurrent equation of such type

We show 3 of them with simple solutions (on rank of complexity)

Case 1

$t(1) = 0$

$t(n) = t(n/2) + c$; $n>0$, $c \in N$ is a constant

($n/2$ means $\lfloor (n/2) \rfloor$ or $\lceil (n/2) \rceil$)

example of algorithm?

Case 1

t(1) = 0
t(n) = t(n/2) + c; n>0, $c \in N$ is a constant
($n/2$ means $\lfloor (n/2) \rfloor$ or $\lceil (n/2) \rceil$)

example of algorithm?

proof: (substitute $n = 2^k$)

$$t(2^k) = t(2^{k-1}) + c = t(2^{k-2}) + c + c = t(2^0) + kc = kc = clog(n)$$

solution: $t(n) = c(log(n)) = \Theta(log(n))$ (logarithmic)

example of algorithm:
binSearch (a version that assumes that the sequence contains the key, since $t(1) = 0$)

Case 2

$t(1) = 0$

$t(n) = t(\lfloor (n/2) \rfloor) + t(\lceil (n/2) \rceil) + c$; $n>0$, $c \in N$ is a constant

example of algorithm?

Case 2

$t(1) = 0$
$t(n) = t(\lfloor (n/2) \rfloor) + t(\lceil (n/2) \rceil) + c$; $n>0$, $c \in N$ is a constant

example of algorithm?

proof: (substitute $n = 2^k$)

$$t(2^k) = 2t(2^{k-1}) + c = 2(2t(2^{k-2}) + c) + c =$$
$$2^2(t(2^{k-2})) + 2^1 c + 2^0 c =$$
$$2^k t(2^0) + c(2^{k-1} + 2^{k-2} + ... + 2^0) = 0 + c(2^k - 1) = c(n-1)$$

solution: $t(n) = c(n-1) = \Theta(n)$ (linear)

example: maximum in sequence

Case 3

$t(1) = 0$

$t(n) = t(\lfloor (n/2) \rfloor) + t(\lceil (n/2) \rceil) + cn$; $n>0$, $c \in N$ is a constant

example of algorithm?

Case 3

$t(1) = 0$
$t(n) = t(\lfloor (n/2) \rfloor) + t(\lceil (n/2) \rceil) + cn$; $n>0$, $c \in N$ is a constant

example of algorithm?
proof: (substitute $n = 2^k$)

$t(2^k) = 2t(2^{k-1}) + c2^k = 2(2t(2^{k-2}) + c2^{k-1}) + c2^k =$
$2^2 t(2^{k-2}) + c2^k + c2^k = 2^k t(2^0) + kc2^k = 0 + cnlog(n)$
solution: $cn(log(n)) = \Theta(nlog(n))$ (linear-logarithmic)

example of algorithm: mergeSort

Completing the Proofs

We solved the equations only for exact powers of 2, i.e. $n = 2^k$. The asymptotic bounds, however, will hold in general, due to the following lemma:

If non-decreasing functions: $t(n) : N \rightarrow N$ and $f(x) : R \rightarrow R$ satisfy:

- $t(2^k) = \Theta(f(2^k))$, for $k \in N$
- $\exists_{x_0>0} \exists_{c>0} \forall_{x \geq x_0} f(2x) \leq cf(x)$

Then $t(n) = \Theta(f(n))$.

What functions satisfy the second condition?
$(x, logx, xlogx, x^2, 2^x)$?

Simple proofs presented on the last few slides are based on: Banachowski, Diks, Rytter "Introduction to Algorithms", Polish 3rd Edition, WNT, 2001, pp.20-21 and p.43; (BDR)

Example - the Average Quicksort's Complexity

Lets solve the following recurrent equation:
$A(0) = A(1) = 0$
$A(n) = (n+1) + \frac{1}{n}(\sum_{s=1}^{n}(A(s-1) + A(n-s)))$; $n > 1$

(The equation represents the average time complexity of some version of quickSort, that can be found e.g. in BDR, with assumption that input data is uniformly distributed among all permutations of n elements)

$$A(n) = \frac{2}{n}\sum_{s=1}^{n} A(s-1) + (n+1)$$

Transform the above to the two following equations:
$nA(n) = 2\sum_{s=1}^{n} A(s-1) + n(n+1)$
$(n-1)A(n-1) = 2\sum_{s=1}^{n-1} A(s-1) + (n-1)n$

Average QuickSort's Complexity, cont.

Lets subtract the 2nd equation from the first:

$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2n$

$nA(n) = (n+1)A(n-1) + 2n$

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1}$$

Now, lets expand the last equation:

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1} = \frac{a(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} =$$

$$= \frac{A(1)}{2} + 2/3 + 2/4 + ... + \frac{2}{n+1} = 2(1 + 1/2 + 1/3 + ... + 1/n - 3/2)$$

Thus,

$A(n) = 2(n+1)(1 + 1/2 + 1/3 + ... + 1/n - 3/2)$

Harmonic Number (cont. of the proof)

$A(n) = 2(n+1)(1+1/2+1/3+...+1/n-3/2)$

The sum $1+1/2+1/3+...+1/n$ is called the (n+1)-th *harmonic number*, denoted as H_{n+1}

It can be proved that asymptotically the following holds:

$H_n = ln(n) + \gamma + O(n^{-1})$, where $\gamma \approx 0,5772156...$ is called the *Euler's constant*.

Thus, finally we obtain:

$A(n) = (\frac{2}{log(e)})(n+1)log(n) + O(n) = \frac{2}{log(e)} nlog(n) + O(n) = \Theta(nlog(n))$ (the factor $2/log(e) \approx 1.44$)

This ends the proof of $\Theta(nlog(n))$ average time complexity of quickSort

Master Theorem - Introduction

(Pol.: "twierdzenie o rekurencji uniwersalnej")

A universal method for solving recurrent equations of the following form:

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

where $a \geq 1, b > 1$: *constants*, $f(n)$ is asymptotically positive

It can represent time complexity of a recurrent algorithm that divides a problem to a sub-problems, each of size n/b and then merges the sub-solutions with the additional complexity described by $f(n)$

E.g. for mergeSort $a = 2, b = 2, f(n) = \Theta(n)$

Master Theorem (Pol.: "Twierdzenie o rekurencji uniwersalnej")

Assume, $T(n) : N \rightarrow R$ is defined as follows:

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

where $a \geq 1, b > 1$: *constants*, n/b denotes $\lfloor (n/b) \rfloor$ or $\lceil (n/b) \rceil$ and $f(n) : R \rightarrow R$ is asymptotically positive

Then $T(n)$ can be asymptotically bounded as follows:

1. if $f(n) = O(n^{log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$
2. if $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a} lgn)$
3. if $f(n) = \Omega(n^{log_b a + \epsilon}$ for some $\epsilon > 0$, and if asymptotically $af(n/b) \leq cf(n)$ for some $c < 1$ ("regularity" condition), then $T(n) = \Theta(f(n))$

(Proof in CLR 4.4)

Interpretation and "Gaps" in the Master Theorem

Lets interpret the Master Theorem. To put it simply, it compares $f(n)$ with $n^{log_b a}$ and states that the function of the higer rank of complexity determines the solution:

1. if $f(n)$ is of *polynomially* lower rank than $n^{log_b a}$, the latter dominates
2. if $f(n)$ and $n^{log_b a}$ are of the same rank, the *lgn* coefficient occurs
3. if $f(n)$ is of *polynomially* higher rank than $n^{log_b a}$ and satisfies the "regularity" condition, the former function represents the rank of complexity

Some cases are not covered by the Master Theorem, i.e. for functions $f(n)$ that fall into "gaps" between conditions 1-2 or 2-3 or that do not satisfy the "regulartity" condition. In such cases the theorem cannot be applied.

Questions/Problems:

- Positive and negative aspects of recursion as an algorithmic tool
- Fibonacci numbers
- Hanoi Towers
- General methods for solving recursive equations
- How to solve linear 2nd-order equations
- Important 3 cases of recursive equations
- How to solve recursive equations satisfying one of the cases in Master Theorem

Thank you for your attention