Computer Programming

Recursion

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 9

Contents

Recursion

- a function is recursive if it calls itself
- its definition includes itself
- there must exist a special case without auto-call
    - otherwise we would have an infinite recursion
- related to mathematical induction
- any loop can be transcribed into a recursive function
- Turing-complete = we can simulate a Turing machine by using only recursion (no loops)

Advantages of recursion

- simple recursive formulas can describe very complex phenomena
- enables a short description of some solutions
- it is a key element for advanced problem solving techniques
  - backtracking, divide et impera, dynamic programming all use recursion in some way
- once a recursive formula is obtained in can be easily transformed into its iterative version

Disadvantages of recursion

- it is a different way of thinking, based on hierarchy and not on linear construction
- function calls in C save information on the stack, so deep recursion can fill up the stack
- if we are not careful during implementation, a subproblem can be calculated multiply times, if it is not saved

Approaches for writing recursive functions

- try to answer the question: if we have the result for a smaller problem, how can we construct the result for the big problem?
- divide the problem:
    - separate the first item from the solution and the rest
    - separate the last item from the solution and the rest
    - divide the solution in the middle
- unroll the procedure for constructing the solution from the last step towards the first step, in reverse order
- solve simple cases

What to avoid

- the usage of global or static variables
  - these usually hold partial results
  - one of the main goals of recursion is avoid secondary effects
- adding additional parameters to the function for saving partial results
- the usage of loops
- calling iterative library functions to calculate for us

Simple example

- let s(n) denote the sum of the first n natural numbers
- it can be defined recursively:
- $s(n) = 0$, if $n < 1$
- $s(n) = n + s(n-1)$, if $n \geq 1$
- notice the special case which does not contain self-reference
- this is checked before the recursive call

Array sum - three types of division

```
int sum_last(int* a, int n){
  if (n == 1)
    return a[n-1];
  else
    return a[n-1] + sum_last(a, n-1);
}

int sum_first(int* a, int n){
  if (n == 1)
    return a[0];
  else
    return a[0] + sum_first(a+1, n-1);
}

int sum_mid(int* a, int l, int r){
  if (l == r)
    return a[l];
  int m = (l+r)/2;
  return sum_mid(a, l, m) + sum_mid(a, m+1, r);
}
```

Program 9.1 - Array sum

```c
#include <stdio.h>

int sum_last(int* a, int n);
int sum_first(int* a, int n);
int sum_mid(int* a, int l, int r);

int main(){
  int a[] = {1, 2, 3, 4};
  int n = sizeof(a)/sizeof(int);
  printf("%d %d %d",
    sum_last(a, n),
    sum_first(a, n),
    sum_mid(a, 0, n-1));
  return 0;
}
```

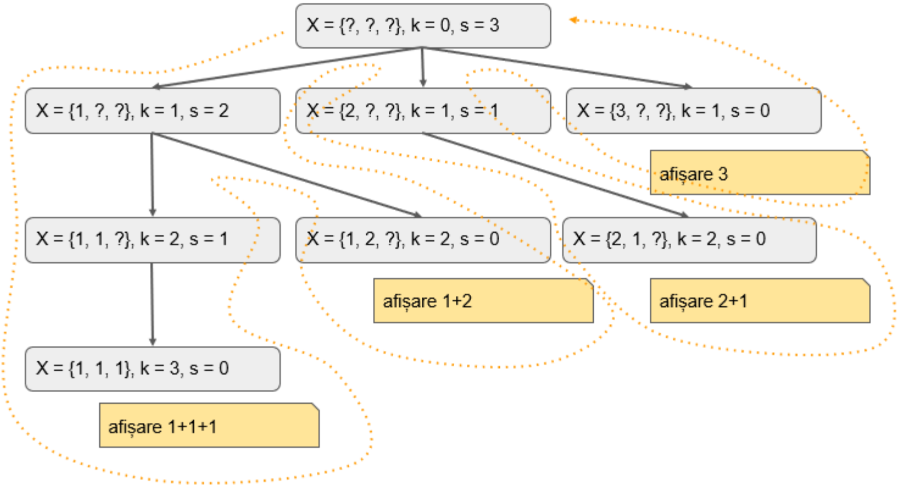
- functions defined on the previous slide
- we send a pointer and array size
- or a pointer and the interval
- take care to use 0-indexing
- which method is faster? why?

Program 9.2 - Compositions

```c
#include <stdio.h>
void compositions(int* X, int k,
      int s){
  if (s == 0){
    for(int i=0; i<k-1; i++)
      printf("%d + ", X[i]);
    printf("%d\n", X[k-1]);
    return;
  }
  for(int x = 1; x <= s; x++){
    X[k] = x;
    compositions(X, k+1, s-x);
  }
}
int main(){
  int n = 3;
  int X[n];
  compositions(X, 0, n);
  return 0;
}
```

- compositions = number of ways to write n as a sum of natural numbers
- X stores the current solution
- k is the number of terms
- s is the remaining sum
- recursive backtracking = generate all solutions
- recursive formula? closed-form?

Compositions - search tree visualization



Execution stack revisited

- it is the memory zone dedicated to local variables, function arguments, function return addresses
- all function calls are pushed to the top of the stack
- when a function is done, its variables are popped from the top of the stack
- has a typical dimension of a few MBs
- can be changed from settings, but in general, it is not recommended to have large variables on the stack, or deep recursion

Program 9.3 - Execution stack + recursion

```c
#include <stdio.h>
void f(int n){
  if (n > 1)
    f(n/2); //B
  printf("%d", n&1);
}
int main(){
  f(11); //A
  return 0;
}
```

f	$n = 1$	return to line B	print 1
f	$n = 2$	return to line B	print 0
f	$n = 5$	return to line B	print 1
f	$n = 11$	return to line A	print 1
main			

Contents

Recursion on strings - recommendations

- recursive functions on strings can be extremely compact
- most only need the initial string pointer as input
- this is because the null-character marks the end of the string
- first, handle the special cases which are usually empty strings or strings of small length
- then write the recursive formula
- do not use functions from the library (strlen, strcpy) because these may traverse the whole string

Simple examples - string length and character count

```
int strlen_rec(char* s){
  if (s[0] == 0)
    return 0;
  return 1 + strlen_rec(s+1);
}

int strcnt_rec(char* s, char c){
  if (!*s)
    return 0;
  return (*s == c) + strcnt_rec(s+1, c);
}
```

- if s points to the null-character we have the empty string
- the length of the empty string is 0
- otherwise the length is 1 more than the length of the string starting from the second character

- if s points to the null-character we have the empty string
- the empty string does not contain the character c
- otherwise, we check if the first character is c
- add this to the result on the string without the first

Complex example - substring search

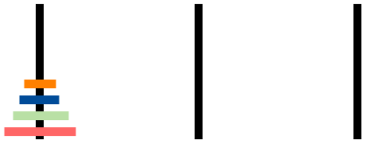
```
1  int strstr_rec(char* t, char* s, int matched)
2  {
3    //printf("%s %s %d\n", t, s, matched);
4    if (*s == 0)
5      return 0;
6    if (*t == 0)
7      return -1;
8    if (*t == *s)
9    {
10     int r = strstr_rec(t+1, s+1, matched+1);
11     if (r >= 0)
12       return r;
13   }
14   else
15   {
16     int r = strstr_rec(t+1-matched, s-matched, 0);
17     if (r >= 0)
18       return r+1;
19   }
20   return -1;
21 }
```

- *matched* stores the number of characters matched so far
- needed when we restart from a bad partial match
- if from line 8: continue matching or reset

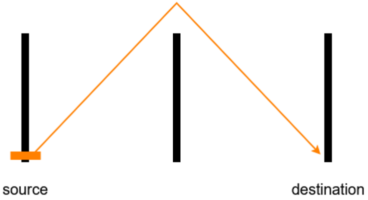
Contents

Problem definition

- define the problem with n disks:
  - there are 3 pegs
  - n disks are placed on the left-most peg, in decreasing order of their diameter
  - move the disks to another peg
  - you are not allowed to place a large disk on a smaller one
- let H(n) denote the minimal number of moves required

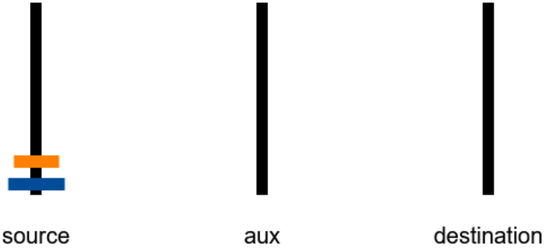


Small case $n = 1$



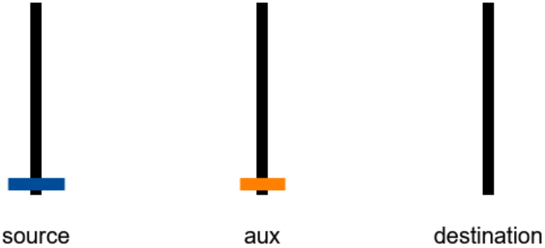
For $n = 1$ we just move the single disk from the source peg to the destination peg. So $H(1) = 1$.

Small case $n = 2$



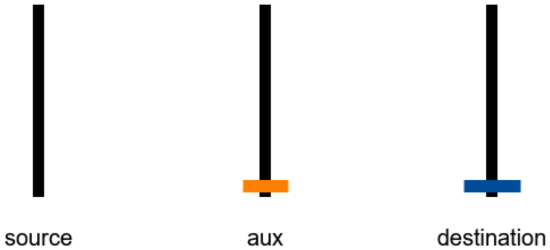
For $n = 2$ we need to use the intermediate peg (aux).
starting position

Small case n = 2



For n = 2 we need to use the intermediate peg (aux).
step 1

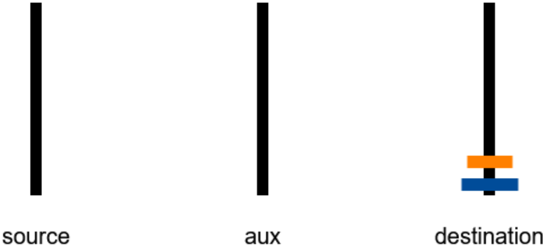
Small case $n = 1$



For $n = 2$ we need to use the intermediate peg (aux).

step 2

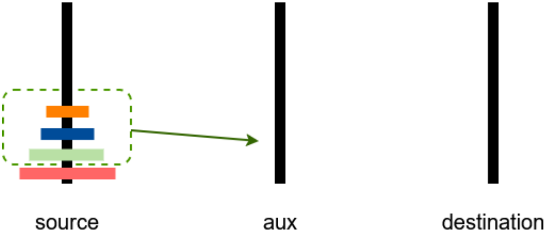
General n



For $n = 2$ we need to use the intermediate peg (aux).
step 3

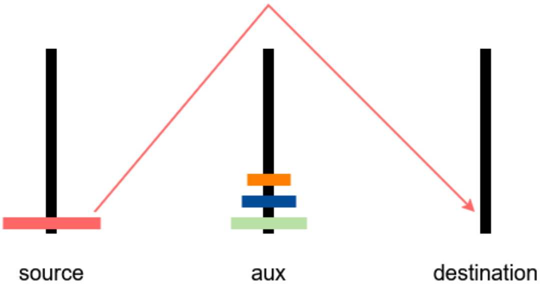
We can see that $H(2) = 3$.

General n



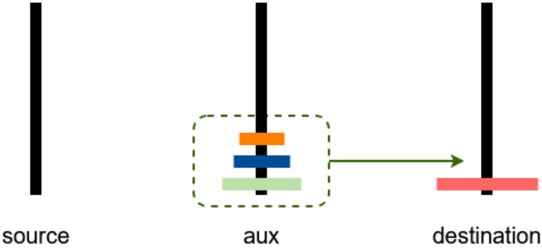
For general n, we need to move all the smaller disks from the larger disk the intermediate peg.

General n



Then move the large disk to the destination peg.

General n



Finally, move the n-1 smaller disks to the destination peg.

Notice that moving n-1 smaller disks is the same problem,
but the roles of the pegs change.

Program 9.4 - Hanoi Towers

```c
#include <stdio.h>

void Hanoi(int n, char a, char b, char c){
  if (n == 1)
    printf("%c -> %c\n", a, c);
  else{
    Hanoi(n-1, a, c, b);
    printf("%c -> %c\n", a, c);
    Hanoi(n-1, b, a, c);
  }
}

int main(){
  Hanoi(3, 'A', 'B', 'C');
  return 0;
}
```

- a = source peg
- b = intermediate peg
- c = destination peg
- what is the recurrence for H(n)?

Problem definition

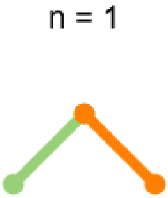
- we define a mountain range of length n
  - start from the ground level
  - use exactly 2n moves (n pairs)
  - each move is of unit length in the up-right or down-right direction
  - we are not allowed to move below the ground level
  - it must end at the ground level
- let M(n) count the mountain ranges of length n

Approach outline

General approach for solving harder problems

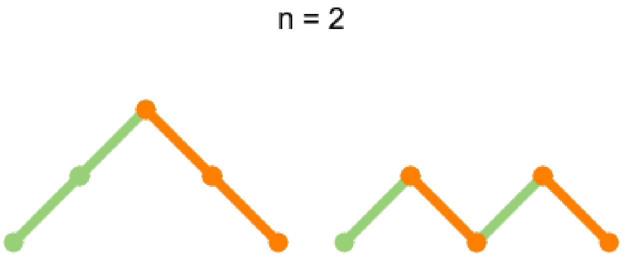
- draw some examples
- find the answer for small n
- come up with a recursive formula for large n
  - reduce the large problem to a smaller instance
  - combine the answers from smaller instances
- write a recursive function
- *memoize* the function
- transform it into the iterative version

Small case n = 1



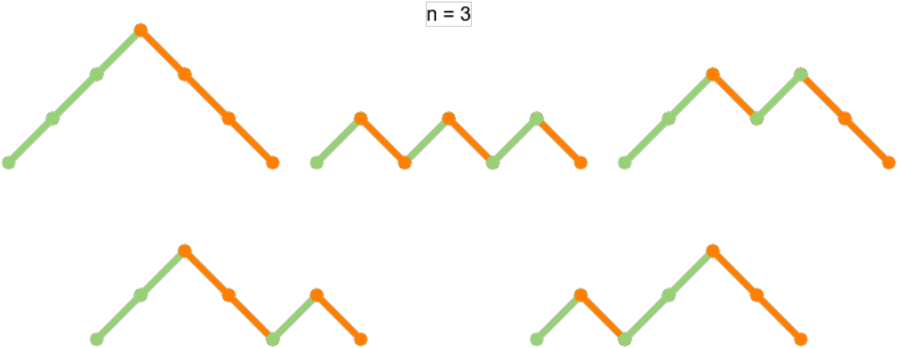
$M(1) = 1$

Small case n = 2



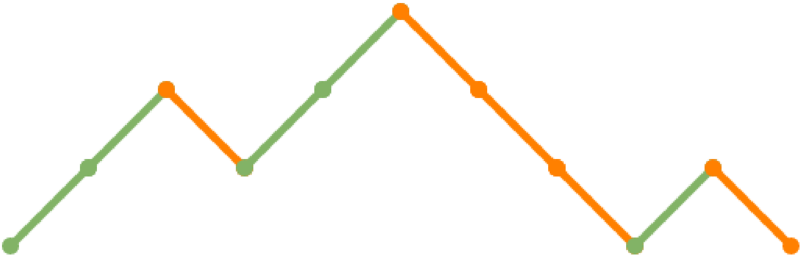
M(2) = 2

Small case n = 2



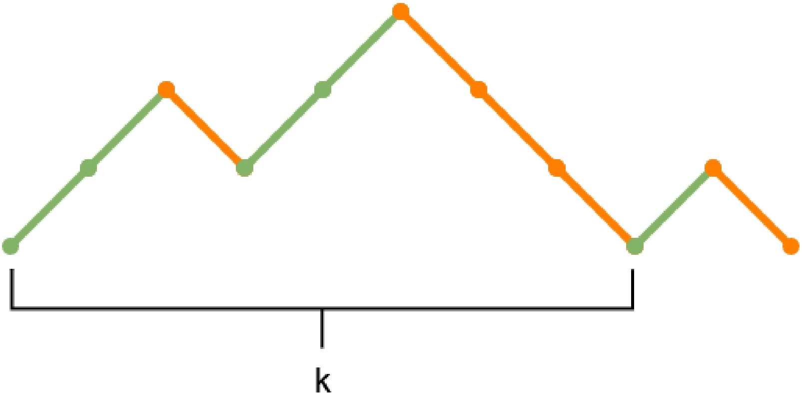
$$M(3) = 5$$

General n



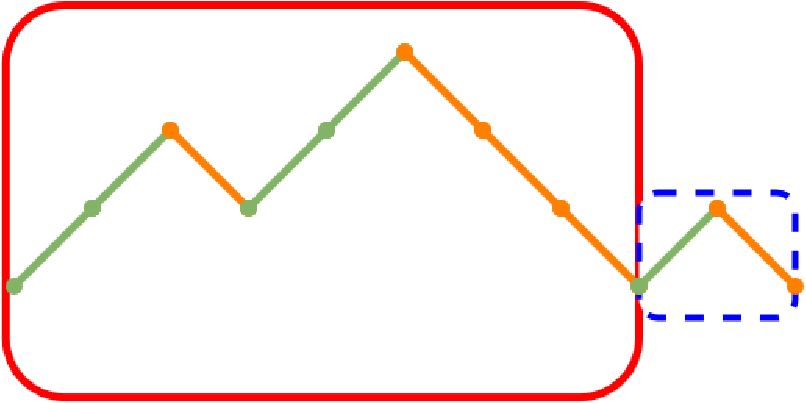
Consider the general case for large n

General n - first return



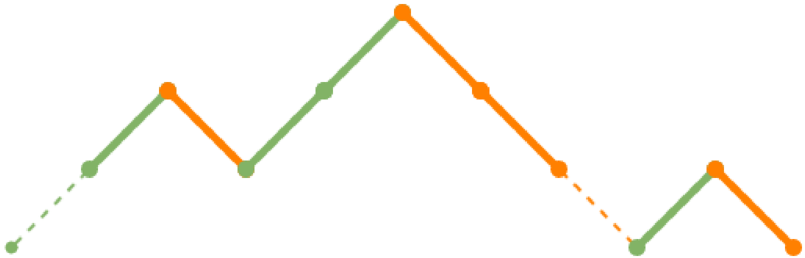
Find k such that the mountain range first returns to ground level after 2k moves

General n - split



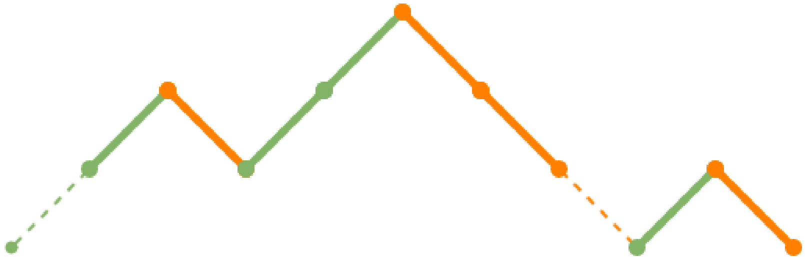
This splits the range into two smaller ranges (red and blue)

General n - split



Chop off the first and last segment from the left mountain range.
We are left with a mountain range of size k-1 and one with size n-k.

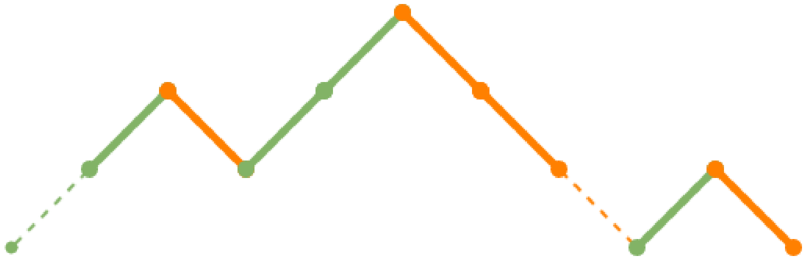
General n - split



Both parts are counted by the same function, so the number of ways to draw them is:

$$M(k-1)M(n-k)$$

General n - formula



Because k can be any value between 1 and n we have the formula for M(n) considering all possible return positions:

$$M(n) = \sum_{k=1}^{n} M(k-1)M(n-k)$$

Edge cases

- what should we use for $M(0)$?
- this is the number of empty mountain ranges, with no movement
- this value appears in our previous formula when $k = 1$ or $k = n$
- it makes sense to set $M(0) = 1$

Recursive function

```
long long M(int n){
  if (n < 2)
    return 1;
  long long r = 0;
  for(int k = 1; k <= n; k++)
    r += M(k-1)*M(n-k);
  return r;
}
```

- already slow for $n = 20$, why?

Memoized recursive function

```
long long M(int n, long long* memo){
  if (n < 2)
    return 1;
  if (memo[n] != -1)
    return memo[n];
  long long r = 0;
  for(int k = 1; k <= n; k++)
    r += M(k-1, memo)*M(n-k, memo);
  memo[n] = r;
  return r;
}
//memo allocated + initialized with -1
```

- avoid recalculation of the same sub-problem

Iterative function

```c
long long M(int n){
  long long memo[n+1];
  memset(memo, 0, sizeof(memo));
  memo[0] = 1;
  for(int i = 1; i <= n; i++){
    for(int k = 1; k <= i; k++)
      memo[i] += memo[k-1]*memo[i-k];
  }
  return memo[n];
}
```

- we can calculate from small to large n

Conclusion

- the values of M(n) are the Catalan numbers
- closed form exists:

$$M(n) = \binom{2n}{n} / (n + 1)$$

- this formula can be deduced from the recursive formula
- what changes if we can go below the ground level?
- what if we can move horizontally?
- why do we chop off segments from the left mountain range?