# Algorithms and Data Structures

*Lecture 4:*

*Recursion,*
*Dynamic programming,*
*Divide & Conquer*

*Sequence Alignment, Quicksort*

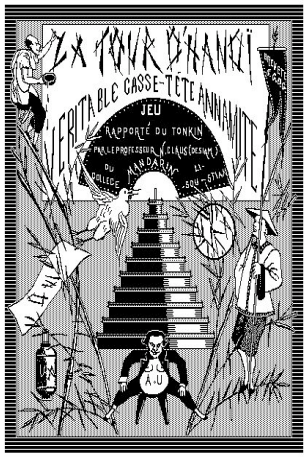
Verónica Gaspes

veronica.gaspes@ide.hh.se

*www.hh.se/staff/vero/itads*



# Tower of Hanoi




- Tower of Hanoi puzzle, marketed in 1883 by Professor N. CLAUS (DE SIAM), an anagram pseudonym for Professor Édouard LUCAS (D'AMEINS).
- The game consists of demolishing the tower level by level, and reconstructing it in a neighboring place, conforming to the rules given.



# Tower of Hanoi

Move all plates from peg A to peg C

Plates can be moved one by one from one peg to another peg

At no stages should a smaller plate come below a bigger plate

An extra peg B can be used.



# Tower of Hanoi

Move N-1 smallest discs to pole B.

Move largest disc to pole C.

Move N-1 smallest discs to pole C.

# Recursion

A *recursive method* is a method that directly or indirectly makes a call to itself.

```
void hanoi(int n, char from, char to, char h){
    if(n>0){
      hanoi(n-1,from,h,to);
      System.out.println(from+" --> "+to);
      hanoi(n-1,h,to,from);
    }
}
```

The *recursive calls* on *values closer to 0*.



# Recursion

```
void hanoi(int n, char from, char to, char help){
    if(n>0){
      hanoi(n-1,from,help,to);
      System.out.println(from+" --> "+to);
      hanoi(n-1,help,to,from);
    }
}
```

The Base Case. Allways have at least one case that is solved *without recursion*. In `hanoi`, `0` and all negative integers are base cases: do nothing!

Progress towards the base case. All recursive calls must be done with *arguments that get closer* to the base case. In `hanoi`, when calling with a positive integer `x`, the recursive calls are with `x-1`

*You gotta believe!* Always assume that the *recursive calls work*. And complete the solution for the actual value!



# Fibonacci numbers

Consider the following sequence of numbers

1 1 1+1 2 1+2 3 2+3 5 3+5 8 5+8 13 8+13 2

Strange as it seems it has very nice properties, it occurrs in many places and has magazines dedicated to it!

We can define the $n-th$ element of the sequence:

$$fib(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases}$$



# Fibonacci numbers

A program that computes the $n-th$ fibonacci number:

```
int fib(int n){
  if(n==0||n==1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

# Fibonacci numbers

Nice, all the *rules* are followed (base cases, progression, belief!)

BUT! The recursive calls are *overlapping*:

To compute `fib(5)` we call `fib(4)` and `fib(3)`
To compute `fib(4)` we call `fib(3)` and `fib(2)`
to compute `fib(3)` we call `fib(2)` and `fib(1)`
To compute `fib(2)` we call `fib(1)` and `fib(0)`

This leads to *very inefficient* programs!

More about this later today, first

a good use of recursion . . .



# Divide and Conquer

A problem solving technique that leads to recursive solutions.

A divide and conquer algorithm is an efficient recursive algorithm that consists of 2 parts:

*Divide*: Smaller problems are solved recursively (except the base cases!)

*Conquer*: The solution to the original problem is formed from the solutions to the subproblems.

Hopefully all subproblems are much smaller than the original one and the subproblems do not overlap!



# Divide & Conquer and Sorting

Sort an array using *Divide and Conquer*:

To sort an array of size $N$.

Divide the array into two halves.

Recursively sort the two parts.

Put together the sorted parts to a sorted whole.

What to do for putting together depends on how we choose to divide



# Quicksort

*To sort an array of size* $10$.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 13 | 81 | 92 | 43 | 31 | 65 | 57 | 26 | 75 | 0 |

*Divide the array in two halves.*

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 81 | 92 | 43 | 31 | 65 | 57 | 26 | 75 | 0 | Pivot? |

Partition

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 0 | 26 | 43 | 57 | 31 | 65 | 92 | 75 | 81 |

*Recursively sort the two parts. (Believe! Quicksort)*

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 13 | 26 | 31 | 43 | 57 | 65 | 75 | 81 | 92 |

*Put together the sorted parts.*

# Quicksort

Auxiliary methods

1. Find a good pivot. An element in the array that has more or less as many elements smaller as it has larger in the array.

   Find it in constant time!

   Median of 3 among `a[low]`, `a[mid]` and `a[high]`

2. Partition. All smaller than the pivot to the left, all larger to the right.

   Loop through the array from `low` upwards and from `high` downwards.

   Stop on elements that are on the wrong half.

   Exchange elements when needed and continue looping until all elements are in the proper half.



# Quicksort

```
void quicksort(T [ ] a, int low, int high ){
    if( small array )
        insertionSort( a, low, high );
    else{
        int middle = ( low + high ) / 2;
        sort low, middle, high
        partition
        quicksort( a, low, i - 1 );
         // Pivot at i
        quicksort( a, i + 1, high );
    }
}
```



# Quicksort

Small array

`low + CUTOFF > high`

where `CUTOFF` can be around 10.



# Quicksort

sort `low`, `middle`, `high`

```
if( a[ middle ].compareTo( a[ low ] ) < 0 )
   swapReferences( a, low, middle );
if( a[ high ].compareTo( a[ low ] ) < 0 )
   swapReferences( a, low, high );
if( a[ high ].compareTo( a[ middle ] ) < 0 )
   swapReferences( a, middle, high );
```

# Quicksort

Partition

```
// Place pivot at position high - 1
swapReferences( a, middle, high - 1 );
T pivot = a[ high - 1 ];
// Begin partitioning
int i, j;
for( i = low, j = high - 1; ; ){
   while( a[ ++i ].compareTo( pivot ) < 0 );
   while( pivot.compareTo( a[ --j ] ) < 0 );
   if( i >= j ) break;
   swapReferences( a, i, j );
}
// Restore pivot
swapReferences( a, i, high - 1 );
```



# Quicksort - analysis

$T(N)$ *time to sort an array of size N*

Divide it into two halves takes $\mathcal{O}(c)$ to pick the pivot and $\mathcal{O}(N)$ to partition. So division is $\mathcal{O}(N)$.

Recursively sort the two parts will take $T(N_{small}) + T(N_{large})$

Put together the solutions do nothing!

So

$$T(N) = T(N_{small}) + T(N_{large}) + \mathcal{O}(N)$$



# Quicksort - analysis

$$T(N) = T(N_{small}) + T(N_{large}) + \mathcal{O}(N)$$

If we manage to divide the array in equal sized parts we will get

$$T(N) = 2T(\tfrac{N}{2}) + N \ = 4T(\tfrac{N}{4}) + 2\tfrac{N}{2} + N \ = \dots$$
$$= NT(1) + Nlog(N)$$

$T(N)$ is $\mathcal{O}(Nlog(N))$ if we manage to find a good pivot in constant time!

Compare with $\mathcal{O}(N^2)$ for insertion sort!



# Fibonacci's problem

- Look once more at the definition of $fib(n)$:

$$fib(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases}$$

- An *obvious* java program

```
int fib(int n){
  if(n==0||n==1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

leads to an *explosion* of recursive calls with values being computed once and again!

# The Problem

```
fib(8)
    fib(7)
        fib(6)
            fib(5)
                fib(4)
                fib(3)
            fib(4)
                fib(3)
                fib(2)
        fib(5)
            fib(4)
                fib(3)
                fib(2)
            fib(3)
                fib(2)
                fib(1)
```

```
fib(6)
    fib(5)
        fib(4)
            fib(3)
            fib(2)
        fib(3)
            fib(2)
            fib(1)
    fib(4)
        fib(3)
            fib(2)
            fib(1)
        fib(2)
            fib(1)
            fib(0)
```

and it is not over . . .



# Memoaization

Whenever we have to compute a value, check in a memo whether we already have computed it!

This means that when we compute a value for the first time, we have to record it in a memo!

```
BigInteger [] memo;

BigInteger fib(int n){
  if(n == 0 || n == 1)
     memo[n]=BigInteger.ONE;
  else
     if(memo[n].equals(BigInteger.ZERO))
        memo[n]=fib(n-2).add(fib(n-1));
  return memo[n];
}
```



# Bottom-up: Dynamic programming

It is easy to realize that we can fill the array from the base cases and forward! And that we only need 2 values any point!

```
BigInteger fibIt(int n){
  BigInteger fn_1 = BigInteger.ONE;
  BigInteger fn_2 = BigInteger.ZERO;
  while(n-- > 0){
    fn_1 = fn_1.add(fn_2);
    fn_2 = fn_1.subtract(fn_2);
  }
  return fn_1;
}
```

In short from a *recursive* formulation of the problem to an *iterative* program that recalls computed values that are further needed



# Sequence Comparison

A more advanced application of dynamic programming

A widely applied topic: file comparisson, spelling correction, information retrieval and searching for similarities among biosequences.

- How similar are the strings `VERONICA` and `MARTIN`?
- How similar are spinach and rice? (according to peptide sequences of Triosephosphate Isomerase):
  - `CNGTKESITKLVSDLNSATLEAD__VDVVVAPPFVYIDQVKSSLTGRVEISA`
  - `CNGTTDQVDKIVKILNEGQIASTDVVEVVVSPPYVFLPVVKSQLRPEIQVAA`
- And monkeys and humans?
  - `MNGRKQNLGELIGTLNAAKVPAD__TEVVCAPPTAYIDFARQKLDPKIAVAA`
  - `MNGRKQSLGELIGTLNAAKVPAD__TEVVCAPPTAYIDFARQKLDPKIAVAA`

# Minimal Edit Distance

One such string comparison problem can be stated as

Align two strings in such a way that the number of commands needed to transform one into the other is minimal.

```
VERONICA
MARTIN__
```

requires 7 changes (editing commands)
while

```
VERONICA
MART_IN_
```

requires only 6!
as well as

```
VERONICA
MAR_TIN_
```



# Minimal Edit Distance

Or, more formally

*Given 2 strings compute an alignment that minimizes the edit distance between them*

For strings $a$ and $b$, the distance $\delta(a, b)$ is

$$\delta(a, b) = \Sigma\delta(a_i, b_i)$$

for the aligned strings (possibly with gaps) where

$$\delta(a_i, b_i) = \begin{cases} 0 & \text{if } a_i = b_i \\ 1 & \text{if } a_i \neq b_i \end{cases}$$



# Minimal Edit Distance

**First attempt**

Enumerate all alignments and their distances and choose an alignment with minimum distance.

Unfortunately . . . there are too many!
For strings of lengths $m$ and $n$ there are

$$\frac{(m+n)!}{m!n!}$$

alignments and for $n = m = 150$ this is approximately $10^{90}$!



# Minimal Edit Distance

**Second attempt**

Based on the observation that *Any prefix of the optimal alignment is an optimal alignment of prefixes* use the recursion

$$\mu(i,j) = \begin{cases} j & \text{for } i = 0 \\ i & \text{for } j = 0 \\ min \begin{cases} \mu(i-1,j)+1 \\ \mu(i,j-1)+1 \\ \mu(i-1,j-1)+\delta(a_{i-1},b_{j-1}) \end{cases} & \end{cases}$$

where $\mu(i, j)$ is the minimal cost of aligning the prefixes of $a$ and $b$ of lengths $i$ and $j$ respectively. Base cases correspond to empty prefixes, indexes in the strings are $0 \ldots m-1, 0 \ldots n-1$.

# Minimal Edit Distance

**Third attempt - Dynamic Programming**

Each step of the recursion requires *3 values*. Try to find a way of recording the values in a bottom-up fashion.

| | ” | v | e | r | o | n | i | c | a |
|---|---|---|---|---|---|---|---|---|---|
| ” | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| m | 1 | 1 | 2 | 3 | | | | | |
| a | 2 | 2 | 2 | 3 | | | | | |
| r | 3 | 3 | 3 | 2 | | | | | |
| t | 4 | | | | | | | | |
| i | 5 | | | | | | | | |
| n | 6 | | | | | | | | ? |

” v ve ver vero veron

” _ __ ___ ____ _____

veroni veronic veronica



# Minimal Edit Distance

**Dynamic Programming**

The matrix can be filled in different ways so that the values needed in the computation are available:

- Row by row ... 
- Column by column ...
- Antidiagonal by antidiagonal



# Dynamic Programming

- The problem is stated as an *optimization* problem.
- Optimal values are *defined recursively*.
- *Efficient solutions* are derived *memorizing already computed values* (using *dynamic porgramming*)
- In some problems, e.g. sequence alignment, not only the optimal value is of interest, but also how it is achieved.

```
>java SequenceAlignment1 veronica martin
6
veronica
mar-ti-n
```

In this case extra space must be used to trace it back



# Tracing back an alignment

When a value is chosen for $\mu(i,j)$ by taking

$$min \begin{cases} \mu(i-1,j)+1 \\ \mu(i,j-1)+1 \\ \mu(i-1,j-1)+\delta(a_{i-1},b_{j-1}) \end{cases}$$

we record also *the coordinates* of the chosen alternative:

| | ” | v | e | r | o | n | i | c | a |
|---|---|---|---|---|---|---|---|---|---|
| ” | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| m | 1 | 1 | 2 | 3 | | | | | |
| a | 2 | 2 | 2 | 3 | | | | | |
| r | 3 | 3 | 3 | 2 | | | | | |

We record that for $\mu(3,3)$ we come from cell $(2,2)$

# Tracing back an alignment

We have to do this for each cell in the matrix, we need a matrix of

```
class Coord{
  int i, j;
  Coord(int x, int y){
    i=x;j=y;
  }
}
```

We fill both matrices during the same traversal of all possible alignments

The optimal alignment is then recovered by tracing the coordinates back from the value corresponding to the alignment of the complete strings.



# Sequence Alignment in Bioinformatics

DNA and proteins are built as long chains of chemichal components *(biosequences)* conventionaly denoted by letters

`A G C T` for ADN

`A C D E F G H I K L M N P Q R S T V W Y` for proteins

Biosequences are compared in the hope that what holds for a sequence also holds for *similar* sequences.

The way of comparing biosequences is by finding *good alignments*

Alignments are good when they *maximize similarity*



# Sequence Alignment in Bioinformatics

## Score matrices

Similarity between biosequences is built up from how similar the letters are.

There is not only match/mismatch but *matrices* that describe how similar each pair of letters is

This is related to how likely it is that a letter is the result of a mutation from some ancestor

There are *many!* computed *score matrices*: *(e.g. gonnet)*

| | C | S | T | P | A | G | N | D | E |
|---|---|---|---|---|---|---|---|---|---|
| C | 12 | 0 | 0 | -3 | 0 | -2 | -2 | -3 | -3 |
| S | 0 | 2 | 2 | 0 | 1 | 0 | 1 | 0 | 0 |

. . . . . .



# Sequence Alignment in Bioinformatics

Dynamic programming made sequence alignment feasible.

Many optimizations have been proposed: to *minimize the space* required for computations; heuristics that *reduce* the portion of $\mu$ that is explored

There are now search engines for huge databases: **BLAST** the Basic Local Alignment Search Tool.

Original sources:

*A general method appplicable to search for similarities in the amino acid sequence of two proteins* by Needleman and Wunch, JBL 1970.

*Identification of common molecular subsequences* by Smith and Waterman, JBL 1981.

*Basic Local Alignment Search Tool* by Altschul et al., JBL 1990.