Lecture 24
Methods for algorithm design
TDDD86: DALP

Print version of the lecture *Data structures, algorithms and programming paradigms*
6 december 2016

Jalil Boudjadar, Tommy Färnqvist. IDA, Linköping University

Content

Innehåll

1 Decomposition

Divide and conquer, söndra och härska

- A general paradigm for algorithms design::
  - Split up the input S into 2 or more disjoint subsets $S_1, S_2, \ldots$
  - Solve partial problems recursively
  - Combine the solutions to partial problems to a solution to S
- Base case is a subproblem of constant size
- Analysis can be performed using recurring relations

Example: Matrix multiplication

$$C = AB \Leftrightarrow \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Suppose we have $n \times n$-matrices and $n = 2^k$ for any k

function MUL(A, B, n)
  if $n = 1$ **then return** $A \cdot B$
  $C_{11} =$ADD(MUL$(A_{11}, B_{11}, \frac{n}{2})$,MUL$(A_{12}, B_{21}, \frac{n}{2}), \frac{n}{2})$
  $C_{12} =$ADD(MUL$(A_{11}, B_{12}, \frac{n}{2})$,MUL$(A_{12}, B_{22}, \frac{n}{2}), \frac{n}{2})$
  $C_{21} =$ADD(MUL$(A_{21}, B_{11}, \frac{n}{2})$,MUL$(A_{22}, B_{21}, \frac{n}{2}), \frac{n}{2})$
  $C_{22} =$ADD(MUL$(A_{21}, B_{12}, \frac{n}{2})$,MUL$(A_{22}, B_{22}, \frac{n}{2}), \frac{n}{2})$
  return C

Analysis

$$\left.\begin{array}{l} T(1) = 1 \\ T(n) = 8T\left(\frac{n}{2}\right) + 4 \cdot \left(\frac{n}{2}\right)^2 \end{array}\right\} \Rightarrow T(n) \in O(n^3)$$

Example: Matrix multiplication (Strassen)

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Can be calculated by $m_1 = (a_{12} - a_{22}) \cdot (b_{21} + b_{22})$ $m_2 = (a_{12} + a_{22}) \cdot (b_{11} + b_{22})$ $m_3 = (a_{11} - a_{21}) \cdot (b_{11} + b_{12})$ $m_4 = (a_{11} + a_{12}) \cdot b_{22}$ $m_5 = a_{11} \cdot (b_{12} - b_{22})$ $m_6 = a_{22} \cdot (b_{21} - b_{11})$ $m_7 = (a_{21} + a_{22}) \cdot b_{11}$
$c_{11} = m_1 + m_2 - m_4 + m_6$ $c_{12} = m_4 + m_5$ $c_{21} = m_6 + m_7$ $c_{22} = m_2 - m_3 + m_5 - m_7$

Analysis
A total of 7 multiplications and 18 additions/subtractions. The multiplication of 2 $n \times n$-matrices takes time

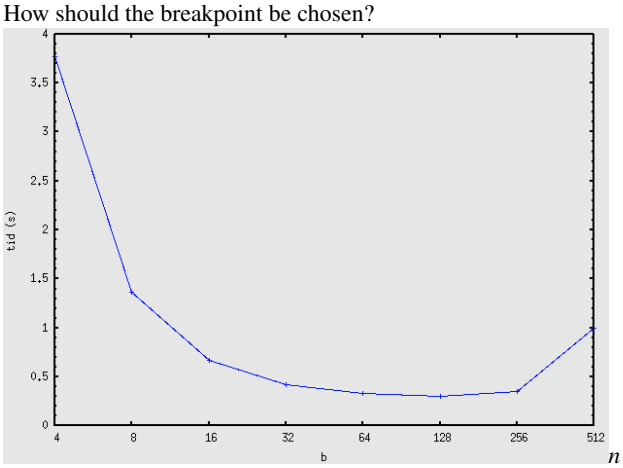
$$\left.\begin{aligned} T(1) &= 1 \\ T(n) &= 7T\left(\tfrac{n}{2}\right) + 18 \cdot \left(\tfrac{n}{2}\right)^2 \end{aligned}\right\} \Rightarrow T(n) \sim n^{2.81}$$

24.5

Strassen in practice

- Use Strassen's matrix multiplication algorithm for large matrices ($n > b$).
- Use standard matrix multiplication for smaller matrices ($n \leq b$)

How should the breakpoint be chosen?



$n = 512$

24.6

Example: Multiplication of binary numbers

$$x = \underbrace{x_{n-1}x_{n-2}\ldots x_{\frac{n}{2}}}_{a}\underbrace{x_{\frac{n}{2}-1}\ldots x_1x_0}_{b} = a \cdot 2^{\frac{n}{2}} + b$$

$$y = \underbrace{y_{n-1}y_{n-2}\ldots y_{\frac{n}{2}}}_{c}\underbrace{y_{\frac{n}{2}-1}\ldots y_1y_0}_{d} = c \cdot 2^{\frac{n}{2}} + d$$

$$x \cdot y = a \cdot c \cdot 2^n + (a \cdot d + b \cdot c)2^{\frac{n}{2}} + b \cdot d$$

Suppose that $n = 2^k$

function MULT(x, y, k)
  if $k = 1$ **then return** $x * y$
  else
    $[a, b] \leftarrow x$
    $[c, d] \leftarrow y$
    $p_1 \leftarrow$MULT$(a, c, k-1)$
    $p_2 \leftarrow$MULT$(b, d, k-1)$
    $p_3 \leftarrow$MULT$(a, d, k-1)$
    $p_4 \leftarrow$MULT$(b, c, k-1)$
    return $p_1 \cdot 2^n + (p_3 + p_4) \cdot 2^{\frac{n}{2}} + p_2$

Analysis

$$T(1) = \Theta(1) \text{ and}$$
$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n)$$
$$\Rightarrow T(n) \in O(n^{\log_2 4})$$

24.7

Example: Smarter multiplication (Karatsuba)

$A \leftarrow a \cdot c$ $B \leftarrow b \cdot d$ $C \leftarrow (a+b) \cdot (c+d)$ $D \leftarrow A \cdot 2^n + (C-A-B) \cdot 2^{\frac{n}{2}} + B$ $D = a \cdot c \cdot 2^n + (a \cdot d + b \cdot c) \cdot 2^{\frac{n}{2}} + b \cdot d = x \cdot y$

function SMARTMULT(x, y, k)
  if $k \leq 4$ **then return** $x * y$
  else
    $[a,b] \leftarrow x$
    $[c,d] \leftarrow y$
    $A \leftarrow$SMARTMULT$(a,c,k-1)$
    $B \leftarrow$SMARTMULT$(b,d,k-1)$
    $C \leftarrow$SMARTMULT$(a+b,c+d,k-1)$
    return $A \cdot 2^n + (C-A-B) \cdot 2^{\frac{n}{2}} + B$

Analysis

$$\left.\begin{array}{l} T(4) = \Theta(1) \\ T(n) = 3 \cdot T\left(\frac{n}{2}\right) + \Theta(n) \end{array}\right\} \Rightarrow T(n) \in O(n^{\log_2 3}) \in O(n^{1,58})$$

24.8

2 Total search

Exhaustive search, total search

- Go through all possible solutions and check if the intended solution exists. This is best done recursively
- Often the number of possible solutions is exponential, and then it can only be used for small n. Total search is a method that one can takes as a last resort.

The hardest part of the total search is to make sure that you go through every possible solution one (and preferably not more than one) time.

Sometimes even before building a solution one can see that it is not a possible solution. Then, one can just ignore it and instead go back and construct the next possible solution. This is called backtracking.

24.9

TSP – salesman problem

TSP: Traveling salesman problem

what is the shortest possible route that visits each city exactly once and returns to the origin city?

Different variants of TSP:

- General TSP instance: graph with edge weights
- Euclidean TSP in dimension d. The cities given as coordinates in $\mathbb{R}^d$

24.10

Example: Solving general TSP with total search

function TSP$(n, d[1 \ldots n, 1 \ldots n])$
  $minlen \leftarrow \infty$
  for $i \leftarrow 1$ **to** n **do** $visited[i] \leftarrow$ **false**
  for $i \leftarrow 1$ **to** n **do**
    $perm[1] \leftarrow i$; $visited[i] \leftarrow$ **true**
    CHECKPERM$(2,0)$
    $visited[i] \leftarrow$ **false**
  return $minlen$
procedure CHECKPERM$(k, length)$
  if $k > n$ **then**
    $totalLength \leftarrow length + d[perm[n], perm[1]]$

```
        if totalLength < minlength then
            minlength ← totalLength
    else
        for i ← 1 to n do
            if ¬visited[i] then
                perm[k] ← i; visited[i] ← true
                CHECKPERM(k+1, length+d[perm[k−1], i])
                visited[i] ← false
```

24.11

3 Dynamic programming

Dynamic programming

A method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions

- Determine the structure of the optimal solution
- Set up recursion for the optimal value
- Calculate the optimal values of partial problems from small to large
- Construct an optimal solution (poss. With additional information stored in step 3)

24.12

Dynamic programming

- Applicable to issues that at first glance seem to require a lot of time (perhaps exponentially):
  - simple subproblems: a subproblrm can be defined in terms of some few variables
  - The optimality of a subproblem: the globally optimal value can be defined in terms of optimal subproblems
  - Overlapping of partial problems: partial problems are not independent, instead they overlap (and therefore should be constructed bottom-up)

24.13

Calculation the Fibonacci number

A serie of numbers calculated via a set of functions $F_0, F_1, ...$

- Fibonacci number is a recursive definition:$F_0 = 0 F_1 = 1 F_i = F_{i-1} + F_{i-2}$ for $i > 1$
- A recursive algorithm (first attempt):

```
function BINARYFIB(k)
    if k < 2 then
        return k
    else
        return BINARYFIB(k−1)+BINARYFIB(k−2)
```

24.14

Analysis of Fibonacci with binary recursion

- Let n_k denote the number of recursive calls made of BinaryFib(k)
  - $n_0 = n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
- The value at least doubled for every other value n_k, i.e. $n_k > 2^{k/2}$. The number of calls grows exponentially!

24.15

A better algorithm for Fibonacci number

- Use dynamic programming instead:

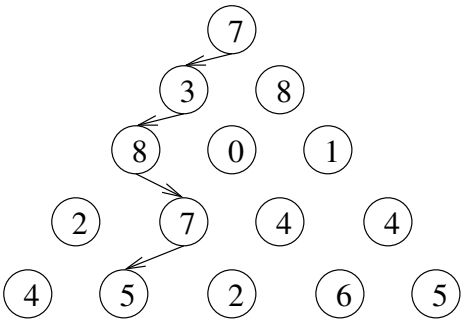
```
function DYNPROGFIB(k)
    int F[0..k]
    F[0] = 0
    F[1] = 1
    for i = 2 to k do
        F[i] = F[i-2] + F[i-1]
    return F[k]
```

- The execution time is $O(k)$

24.16

Example: Maximum triangle path

Problem: find the path from top to bottom that maximizes the sum of its individual numbers.



$n =$ the number of rows It exists 2^{n-1} paths to try $a_{ij} =$ element j in row i

24.17

Dynamic programming solution

- Optimal solution structure path made up of sub-paths
- RecursionLet $V[i,j] =$ the value of the best path from a_{ij} down to row n

$$V[i,j] = \begin{cases} a_{ij} & \text{if i=n (last row)} \\ a_{ij} + \max(V[i+1,j], V[i+1,j+1]) & \text{otherwise} \end{cases}$$

- Calculation

for $j = 1$ **to** n **do**
  $V[n,j] \leftarrow a_{nj}$
for $i = n-1$ **downto** 1 **do**
  for $j = 1$ **to** i **do**
    $V[i,j] \leftarrow a_{ij} + \max(V[i+1,j], V[i+1,j+1])$
return $V[1,1]$

Time complexity: $O(n^2)$

24.18

Partial sequences

- A *partial sequence* of a string $x_0x_1x_2\ldots x_{n-1}$ is a string in the form $x_{i_1}x_{1_2}\ldots x_{i_k}$, where $i_j < i_{j+1}$
- Not the same as substring!

Example: ABCDEFGHIJK

- Partial string: CDEF
- Partial sequence: ACEGIJK
- Partial sequence: DFGHK
- Not a partial sequence: DAGH

24.19

The longest common subsequence problem (LCS)

- Given 2 strings X and Y, LCS is the problem of finding the longest common subsequence of X and Y
- Has applications in testing the matching between DNA strands (the alphabet is {A, C, G, T })

Example
ABCDEFG and XZACKDFWGH have ACDFG as a the longest common subsequence

24.20

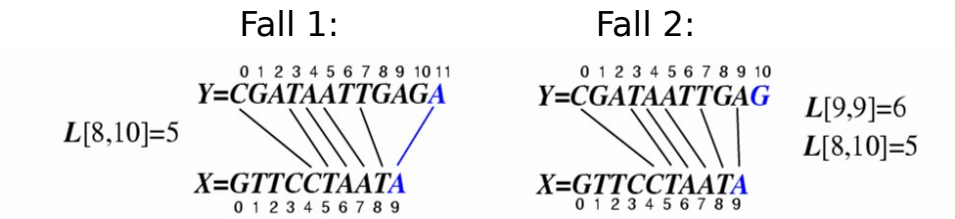
A bad way to approach the LCS problem

- A brute-force solution
  - Enumerate all partial sequences of X
  - Test which also are partial sequences of Y
  - Select the longest
- Analysis
  - If X has length n it contains 2^n partial sequences
  - This algorithm requires exponential time!

24.21

Dynamic programming and LCS-problem

- Let $L[i,j]$ be the length of the longest common subsequence of $X[0\ldots i]$ and $Y[0\ldots j]$
- Allow -1 as the index, so that $L[-1,k]=0$ and $L[k,-1]=0$ to indicate that *null*-subsequence of X or Y does not match at all the second string
- Then we can define $L[i,j]$ in the general case as follows:
  - If $X[i]=Y[j]$, $L[i,j]=L[i-1,j-1]+1$ (we have a hit here)
  - If $X[i]\neq Y[j]$, $L[i,j]=\max\{L[i-1,j],L[i,j-1]\}$ (we have no hit here)



An algorithm for LCS

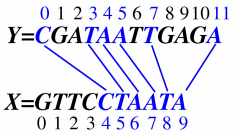
function LCS($X,Y,|X|=n,|Y|=m$)
  for $i=-1$ **to** n-1 **do**
    $L[i,-1]=0$
  for $j=0$ **to** m-1 **do**
    $L[-1,j]=0$
  for $i=0$ **to** $n-1$ **do**
    for $j=0$ **to** $m-1$ **do**
      if $X[i]=Y[j]$ **then**
        $L[i,j]=L[i-1,j-1]+1$
      else
        $L[i,j]=\max\{L[i-1,j],L[i,j-1]\}$
  return array L

Visualization of the LCS-algorithm

m

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**

n



24.24

Analysis of the LCS-algorithm

- We have two nested loops
  - The outer iterates n times
  - The inner iterates m times
  - In each iteration of the inner loop, a constant amount of work is performed
  - Thus, the total execution time is $O(nm)$
- The answer is in $L[n,m]$ (and subsequence can be created from the table L)

24.25

4 Greedy algorithms

Greedy algorithms

Algorithms that solve a piece of the problem at a time. In each step performed, the algorithm gives better return.

- The greedy method is a general paradigm for algorithm design based on the following:
  - configurations: different choices, collections or values to find
  - objective function: configurations assigned a score that we want to maximize or minimize
- It works fine when applied on problems with the *greedy-choice*-property:
  - a globally optimal solution can always be found by a series of local improvements from an initial configuration

For many problems greedy algorithms do not provide optimal solutions but maybe decent approximate solutions.

24.26

Give change

- You need to give x cents in a change, using coins of values 1, 5, 10 and 25 cents. What is the minimum number of coins needed to provide the change?
- Greedy approach:
  - Take as many 25-cent coins as possible, then
  - Take as many 10-cent coins as possible, then
  - Take as many 5-cent coins as possible, then
  - Take as many 1-cent coins as needed to finish.
- Example: 99 cent = $3*25$ cent + $2*10$ cent + $0*5$ cent + $4*1$ cent
- Is it optimal?
- 1, 5, 10 and 25 cent: greedy algorithm provides optimum
- 1, 6 and 10 cent: greedy algorithm does not provide optimum

24.27