nice code

def whatisthis(n):
    return
    (4 << n*(3+n)) // ((4 << 2*n) - (2 << n) - 1)
    & ((2 << n) - 1)

source: Paul Hankin

Fibonacci Numbers

definition of the Fibonacci Numbers \( F_n \):

\[
\begin{align*}
F_1 &= 1 \\
F_2 &= 1 \\
F_n &= F_{n-1} + F_{n-2}
\end{align*}
\]

Algorithm \( \text{fib}(n) \):

\[
\begin{align*}
\text{if } n &= 1 \text{ or } n = 2 \text{ then} \\
\quad &\text{return } 1 \\
\text{else} \\
\quad &x := \text{fib}(n - 1) \\
\quad &y := \text{fib}(n - 2) \\
\quad &\text{return } x + y
\end{align*}
\]
analysis of the naive recursive algorithm

exponential-time algorithm

see the recursion tree (for example for fib(6))

recurrence equation

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \text{ or } n = 2 \\
T(n - 1) + T(n - 2) + 2 & \text{if } n \geq 3 
\end{cases} \]

\( T(n) \in \mathcal{O}(2^n) \)

analytical bound exponential in golden ratio

use memoization

pre-processing:
take array \( r[1 \ldots n] \) and initialize it at 0 for all indices

init: array at 1 and at 2 gets value 1

for-loop: array at 3 up to \( n \) gets value computed from previous two entries
gives linear-time algorithm

bottom-up approach using memoization

what we used

**Algorithm** `fib(n)`:

```plaintext
new array r[1 \ldots n]
r[1] := 1
r[2] := 1
for i := 3 to n do
    r[i] := r[i - 1] + r[i - 2]
return r[n]
```

analysis: \( \mathcal{O}(n) \)

(there also is a logarithmic algorithm)

reuse of easier sub-problems

order computation in order to be able to reuse

systematic storage of results

optimal substructures

overlapping subproblem
dynamic programming is not always the right way
give in graph longest cycle-free path
sub-path of cycle-free path is not necessarily longest

overview
- Fibonacci numbers
- more knapsack 01
- longest common subsequence
- graph representations
- all pairs shortest paths
- matrix multiplication: not yet done in lecture 11
- material

knapsack01: algorithm

$S$ consists of $n$ items with $b_i$ and $w_i$; $W$ is the max total weight

**Algorithm 01Knapsack($S$, $W$):**

new $B[0 \ldots n, 0 \ldots W]$

for $w := 0$ to $W$
do

$B[0, w] := 0$

for $k := 1$ to $n$
do

$B[k, 0] := 0$

for $w := 1$ to $W$
do

if $w_k \leq w$ then

$B[k, w] := \max(B[k - 1, w], B[k - 1, w - w_k] + b_k)$

else

$B[k, w] := B[k - 1, w]$

knapsack01: application

- item 1 with $w_1 = 3$ and $b_1 = 9$
- item 2 with $w_2 = 2$ and $b_2 = 5$
- item 3 with $w_3 = 2$ and $b_3 = 5$
- max total weight $W = 4$

- $B[0, 0] = 0$, $B[0, 1] = 0$, $B[0, 2] = 0$, $B[0, 3] = 0$, $B[0, 4] = 0$
- $B[1, 0] = 0$, $B[1, 1] = 0$, $B[1, 2] = 0$, $B[1, 3] = 9$, $B[1, 4] = 9$

make a table; be ready to say something about the origin of an entry
for every $k = 0, \ldots, n$ we consider $S_k$

for every $S_k$ we consider $w = 0, \ldots, W$

time complexity in $nW$

if $W = 2^n$ then not so nice

this is a pseudo-polynomial algorithm

one does not expect a polynomial algorithm will be found

alternative where only one array is used

Algorithm 01Knapsack($S, W$):

\begin{algorithmic}
\For{$w := 0$ to $W$}
  \State $B[w] := 0$
\EndFor

\For{$k := 1$ to $n$}
  \For{$w := W, \ldots, w_k$}
    \If{$B[w - w_k] + b_k > B[w]$}
      \State $B[w] := B[w - w_k] + b_k$
    \EndIf
  \EndFor
\EndFor
\end{algorithmic}

overview

- Fibonacci numbers
- more knapsack 01
- longest common subsequence
- graph representations
- all pairs shortest paths
- matrix multiplication: not yet done in lecture 11
- material

longest common subsequence

for instance used in bioinformatics

to see similarity between two DNA-sequences

subsequence: obtained by removing elements

example:

$\langle A, B, C, B, D, A, B \rangle$ and $\langle B, D, C, A, B, A \rangle$

have longest common subsequence $\langle B, C, B, A \rangle$
problem
given two finite sequences over some alphabet:
\[ X = \langle x_1, \ldots, x_m \rangle \text{ and } Y = \langle y_1, \ldots, y_n \rangle \]
find (the length of) a maximum-length common subsequence of \( X \) and \( Y \)
abbreviation: LCS = longest common subsequence
brute-force approach
consider every subsequence of \( X \)
check whether it is also a subsequence of \( Y \)
keep track of the longest success found
this is in \( O(2^m) \)

towards a recursive algorithm
towards a dynamic programming algorithm

towards a recursive algorithm
notation:
\[ X_i = \langle x_1, \ldots, x_i \rangle \]
\( c[i, j] \) length of LCS of \( X_i \) en \( Y_j \)

- \( c[i, j] = 0 \) if \( i = 0 \) or \( j = 0 \)
- \( c[i, j] = c[i-1, j-1] + 1 \) if \( i, j > 0 \) and \( x_i = y_j \)
  example: \( X = \langle A, B, C, A, B \rangle \) and \( Y = \langle B, A, B \rangle \)
- \( c[i, j] = \max\{c[i, j-1], c[i-1, j]\} \) if \( i, j > 0 \) and \( x_i \neq y_j \)
  example: \( X = \langle A, B \rangle \) and \( Y = \langle A, B, D \rangle \)
  \( X = \langle A, B, D, B \rangle \) and \( Y = \langle A, B \rangle \)

reuse substructures
Algorithm LCS(X, Y):
    new array C[0...m, 0...n]
    for i := 0 to m do
        C[i, 0] := 0
    for j := 0 to n do
        C[0, j] := 0
    for i := 1 to m do
        for j := 1 to n do
            if \( x_i = y_j \) then
                \( C[i, j] := C[i - 1, j - 1] + 1 \)
            else
                \( C[i, j] := \max\{C[i, j - 1], C[i - 1, j]\} \)
    return C

example

\( X = \langle A, B, C, B, D, A, B \rangle \)
\( Y = \langle B, D, C, A, B, A \rangle \)
or
\( X = \langle A, B, C, B, D, A, B \rangle \)
\( Y = \langle B, D, C, A, B, A \rangle \)

overview

- Fibonacci numbers
- more knapsack 01
- longest common subsequence
- graph representations
- all pairs shortest paths
- matrix multiplication: not yet done in lecture 11
- material
graphs

A graph $G$ consists of nodes $V$ and edges $E$ between a pair of nodes:

$G = (V, E)$

In a directed graph edges are directed (arrows)

In an undirected graph edges can be traversed in either direction

In a weighted graph every edge has an associated weight:

$w : E \rightarrow \mathbb{R}^+$

properties of graphs

Simple:

No parallel edges and no self-loops

Connected:

Between every pair of distinct nodes is a path

adjacency-list representation

Array $Adj$ of $|V|$ lists

For every node there is an index in the array $G.Adj[u]$

At every index there is a list consisting of all neighbours of the node

Memory use: in $\Theta(V + E)$

Direct access to neighbours

Is $uv$ an edge? Requires traversing list at $Adj[u]$

Is it easy to consider weighted or directed graphs?

adjacency-matrix representation

Matrix-entry $a_{ij} = 1$ if and only if there is an edge $(i, j)$

Memory use: in $\Theta(V^2)$

Is $uv$ an edge? Direct

Is it easy to consider weighted or directed graphs?
overview

- Fibonacci numbers
- more knapsack 01
- longest common subsequence
- graph representations
- all pairs shortest paths
- matrix multiplication: not yet done in lecture 11
- material

shortest paths

given a weighted graph

given a start node $s$, find for all vertices $v$ a shortest path from $s$ to $v$

all-pair shortest path:
for all $s$ and for all $v$, find a shortest path from $s$ to $v$

all pairs shortest paths

give distance table
we can apply $V$ times a single source shortest path algorithms
or: dynamic programming algorithm
let the number of stop-overpossibilities increase in every step

all pair shortest paths algorithm

input: a graph with $n$ vertices and a weight function

init: a $n \times n$ matrix with distance 0 from a node to itself, $w(u, v)$ for edges $(u, v)$, and $\infty$ for the rest

loop: let in every iteration the possibilities for stop-overs increase, from no stopovers, to stopovers in $\{v_1\}$ to stopovers in $V = \{v_1, \ldots, v_n\}$
all pair shortest path: example

Graph $G$ with $n$ vertices in $V$ and $m$ edges in $E$ and weight function $w(i,j)$.

**Algorithm** allPairShortestPaths($G, W$):

1. **matrix** $D^0 : n \times n$
2. for $i := 1 \text{ to } n$ do
   1. for $j := 1 \text{ to } n$ do
      1. $d_{ij}^0 := w(i,j)$
3. for $k := 1 \text{ to } n$ do
   1. **matrix** $D^k : n \times n$
   2. for $i := 1 \text{ to } n$ do
      1. for $j := 1 \text{ to } n$ do
         1. $d_{ij}^k := \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$
4. return $D^k$

all pair shortest paths: time complexity

overview

- Fibonacci numbers
- more knapsack 01
- longest common subsequence
- graph representations
- all pairs shortest paths
- matrix multiplication: not yet done in lecture 11
- material
matrix multiplication

input: matrix $A$ with $p$ rows and $q$ columns
and matrix $B$ with $q$ rows and $r$ columns

$A\text{.\hspace{1em}rows and } A\text{.\hspace{1em}columns}$ etc defined

**Algorithm** matrixMultiply($A$, $B$):

new matrix $C : p \times r$

for $i := 1$ to $p$ do

for $j := 1$ to $r$ do

$c_{ij} := 0$

for $k := 1$ to $q$ do

$c_{ij} := c_{ij} + a_{ik}b_{kj}$

time complexity determined by $p \cdot q \cdot r$

brute force?

consider all possibilities for putting parentheses

calculate per possibility the number of steps

take best option

number of possibilities is Catalan number, so exponential, almost $4^n$

matrix multiplication

matrix multiplication is associative, that is, $(AB)C = A(BC)$

how to put parentheses partly determines the time complexity

example:

$$A = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$(A \cdot B) \cdot C$ requires $6 + 6$ multiplication steps

$A \cdot (B \cdot C)$ requires $2 + 3$ multiplication steps

how to find the best parentheses configuration?

matrix multiplication: dynamic programming

problem:
give optimal parentheses configuration for $A_1 \cdot \ldots \cdot A_n$

dimension of $A_i$: $d_{i-1} \times d_i$

idea for $m_{i,j}$: number of steps for optimal parentheses for $A_i \cdot \ldots \cdot A_j$

example: assume optimal parentheses are as follows:

$(A_1 \cdot \ldots \cdot A_i) \cdot (A_{i+1} \cdot \ldots \cdot A_n)$

then number of steps is $m_{i,i} + m_{i+1,n}$ plus the steps for the last multiplication

definition of $m_{i,j}$:

$$m_{i,j} = \min_{i \leq k < j} \{ m_{i,k} + m_{k+1,n} + d_{i-1} \cdot d_k \cdot d_j \}$$

dimensions as follows:

$A_i \cdot \ldots \cdot A_k : d_{i-1} \times d_k$

$A_{k+1} \cdot \ldots \cdot A_j : d_k \times d_j$
Algorithm matrixChain(d):
   new table m[1...n, 1...n]
   for i := 1 to n do
      m[i,i] := 0
   for l := 2 to n do
      for i := 1 to n - l + 1 do
         j := i + l - 1
         m[i,j] := \infty
         for k := i to j - 1 do
            q := m[i,k] + m[k+1,j] + d_{i-1}d_kd_j
            if q < m[i,j] then
               m[i,j] := q
   return m

• $d_0 = 2$, $d_1 = 1$, $d_2 = 2$, $d_3 = 4$, $d_4 = 3$.

• $A_1$ is $2 \times 1$, $A_2$ is $1 \times 2$, $A_3$ is $2 \times 4$, $A_4$ is $4 \times 3$.

• $m[1,1] = 0$, $m[2,2] = 0$, $m[3,3] = 0$, $m[4,4] = 0$
• $m[1,2] = 4$, $m[2,3] = 8$, $m[3,4] = 24$
• $m[1,3] = \min(0 + 8 + 8, 4 + 0 + 16) = 16$, $m[2,4] = \min(0 + 24 + 6, 8 + 0 + 12) = 20$
• $m[1,4] = \min(0 + 20 + 6, 4 + 24 + 12, 16 + 0 + 24) = 26$
• optimal: $A_1 \cdot ((A_2 \cdot A_3) \cdot A_4)$.

overview

• Fibonacci numbers
• more knapsack 01
• longest common subsequence
• graph representations
• all pairs shortest paths
• matrix multiplication: not yet done in lecture 11

extra material

• homepage of S. Rao Kosaraju
• wiki of Robert Tarjan