Data Structures and Algorithms
Week 8

Dynamic programming

- Fibonacci numbers
- Optimization problems
- Matrix multiplication optimization
- Principles of dynamic programming
- Longest Common Subsequence

Algorithm design techniques

- Algorithm design techniques so far:
  - Iterative (brute-force) algorithms
    - For example, insertion sort
  - Algorithms that use efficient data structures
    - For example, heap sort
  - Divide-and-conquer algorithms
    - Binary search, merge sort, quick sort

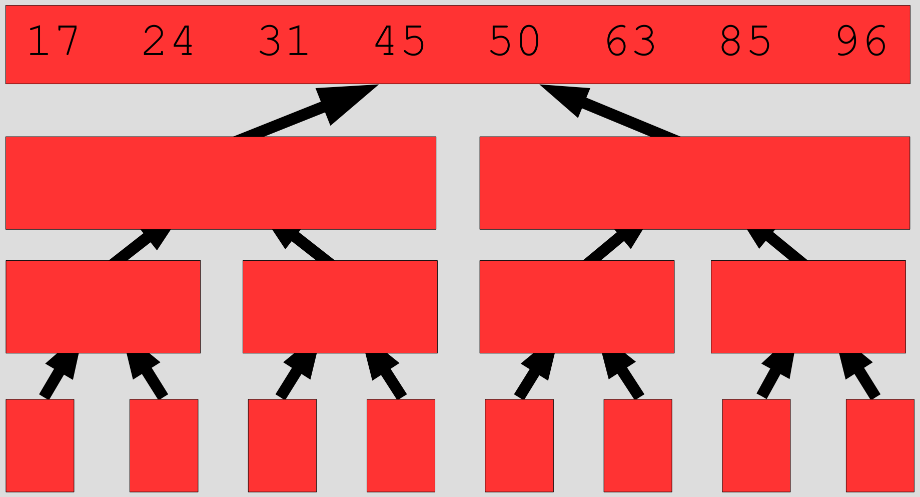
Divide and Conquer

- *Divide and conquer* method for algorithm design:
  - **Divide**: If the input size is too large to deal with in a simple manner, divide the problem into two or more disjoint subproblems
  - **Conquer**: Use divide and conquer recursively to solve the subproblems
  - **Combine**: Take the solutions to the subproblems and “merge” these solutions into a solution for the original problem

Divide and Conquer/2

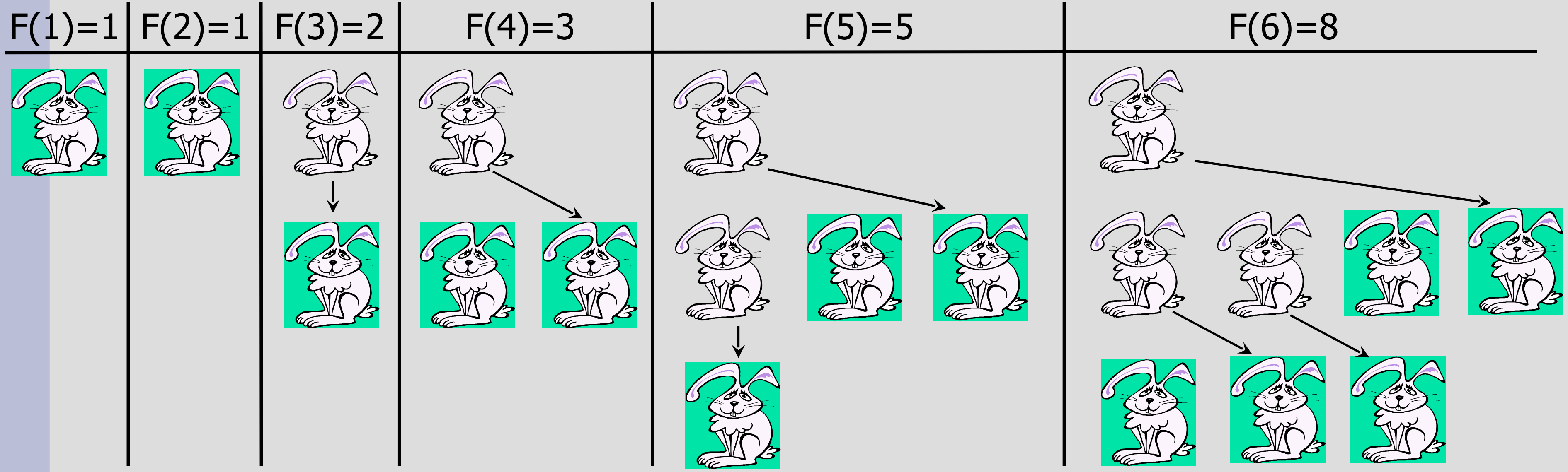
- For example, **MergeSort**
- The subproblems are independent and non-overlapping

```
Merge-Sort(A, l, r)
    if l < r then
        m := (l+r)/2
        Merge-Sort(A, l, m)
        Merge-Sort(A, m+1, r)
        Merge(A, l, m, r)
```



Fibonacci Numbers

- *Leonardo Fibonacci* (*1202*):
  - A rabbit starts producing in the 2nd year after its birth and produces one child each generation.
  - How many rabbits will there be after *n* generations?



Fibonacci Numbers/2

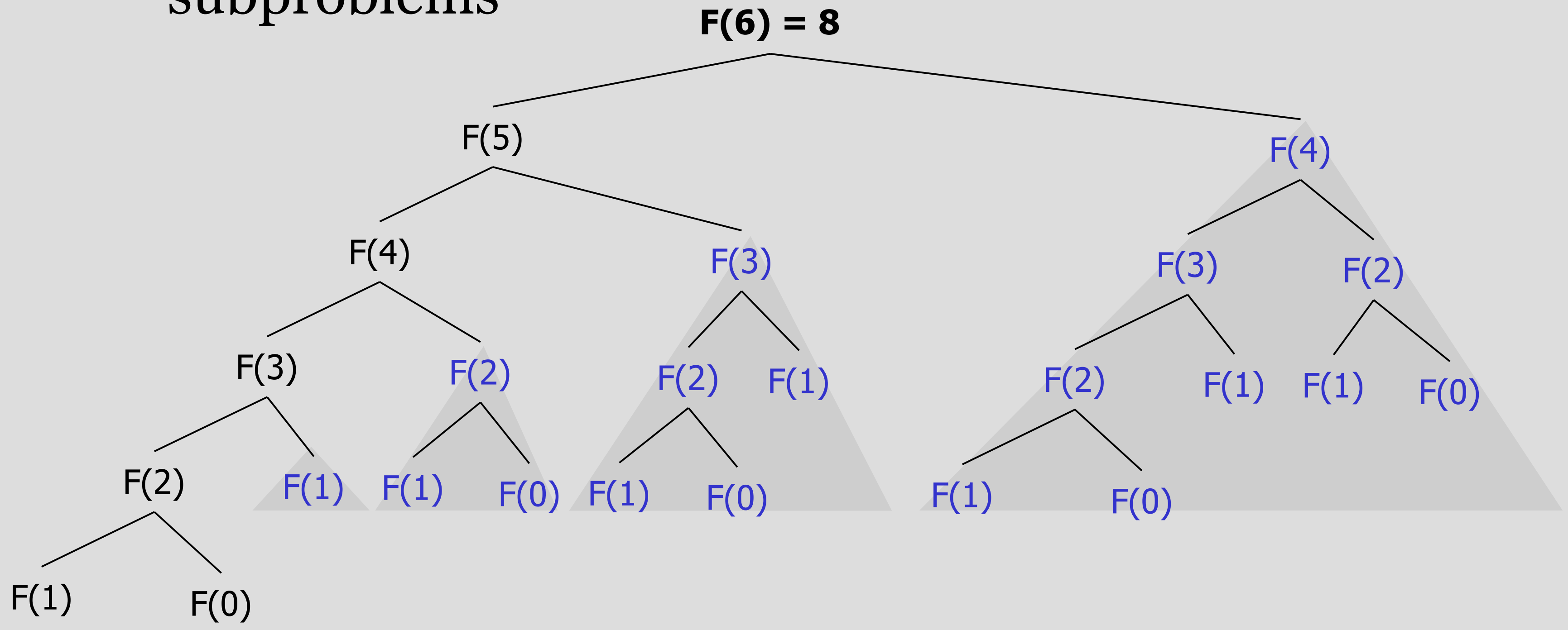
- $F(n) = F(n-1) + F(n-2)$
- $F(0) = 0, F(1) = 1$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

```
FibonacciR(n)
01 if n ≤ 1 then return n
02 else return FibonacciR(n-1) + FibonacciR(n-2)
```

- Straightforward recursive procedure is slow!

Fibonacci Numbers/3

- We keep calculating the same value over and over!
  - Subproblems are overlapping – they share sub-subproblems



Fibonacci Numbers/4

- How many summations are there $S(n)$?
  - $S(n) = S(n-1) + S(n-2) + 1$
  - $S(n) \geq 2S(n-2) + 1$ and $S(1) = S(0) = 0$
  - Solving the recurrence we get

    $S(n) \geq 2^{n/2} - 1 \approx 1.4^n$
- Running time is *exponential!*

Fibonacci Numbers/5

- We can calculate $F(n)$ in *linear* time by **remembering solutions of solved sub-problems** (= *dynamic programming*).
- Compute solution in a bottom-up fashion
- Trade space for time!

```
Fibonacci(n)
01 F[0] := 0
02 F[1] := 1
03 for i := 2 to n do
04    F[i] := F[i-1] + F[i-2]
05 return F[n]
```

Fibonacci Numbers/6

- In fact, only two values need to be remembered at any time!

```
FibonacciImproved(n)
if n ≤ 1 then return n
Fim2 := 0
Fim1 := 1
for i := 2 to n do
   Fi := Fim1 + Fim2
   Fim2 := Fim1
   Fim1 := Fi
return Fi
```

History

- Dynamic programming
  - Invented in the 1950s by *Richard Bellman* as a general method for optimizing multistage decision processes
  - The term “programming” refers to a tabular method.
  - Often used for optimization problems.

Optimization Problems

- We have to choose one solution out of many.
- We want the solution with the optimal (minimum or maximum) value.
- Structure of the solution:
  - It consists of a sequence of choices that were made.
  - What choices have to be made to arrive at an optimal solution?
- An algorithm should compute the optimal value plus, if needed, an optimal solution.

Multiplying Matrices

- Two matrices, A – $n \times m$ matrix and B – $m \times k$ matrix, can be multiplied to get C with dimensions $n \times k$, using nmk scalar multiplications

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} \ldots & \ldots & \ldots \\ \ldots & c_{22} & \ldots \\ \ldots & \ldots & \ldots \end{pmatrix} \qquad c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}$$

- Problem: Compute a product of many matrices efficiently
- Matrix multiplication is *associative:* $(AB)C = A(BC)$

Multiplying Matrices/2

- The parenthesization matters
- Consider A×B×C×D, where
  - A is 30×1, B is 1×40, C is 40×10, D is 10×25
- Costs:
  - $((AB)C)D$ = 1200 + 12000 + 7500 = 20700
  - $(AB)(CD)$ = 1200 + 10000 + 30000 = 41200
  - $A((BC)D)$ = 400 + 250 + 750 = 1400
- We need to optimally parenthesize A_1 x A_2 x … A_n where A_i is a d_{i-1} x d_i matrix

Multiplying Matrices/3

- Let $M(i,j)$ be the *minimum* number of multiplications necessary to compute $A_{i..j} = A_1 \times \dots \times A_n$
- Key observations
  - The outermost parenthesis partitions the chain of matrices (i,j) at some k, $(i \le k < j)$:
    $(A_i \dots A_k)(A_{k+1} \dots A_j)$
  - The optimal parenthesization of matrices (i,j) has optimal parenthesizations on either side of k: for matrices (i,k) and $(k+1,j)$

Multiplying Matrices/4

- We try out all possible *k*:

$$M(i,i) = 0$$

$$M(i,j) = \min_{i \le k < j} \left\{ M(i,k) + M(k+1,j) + d_{i-1} d_k d_j \right\}$$

- A direct recursive implementation is exponential – there is a lot of duplicated work.
- But there are only few different sub-problems (*i,j*): one solution for each choice of i and j (i<j).

Multiplying Matrices/5

- Idea: store the optimal cost $M(i,j)$ for each subproblem in a 2d array $M[1..n,1..n]$
  - Trivially $M(i,i) = 0$, $1 \leq i \leq n$
  - To compute $M(i,j)$, where $i - j = L$, we need only values of M for subproblems of length $< L$.
  - Thus we have to solve subproblems in the increasing length of subproblems: first subproblems of length 2, then of length 3 and so on.
- To reconstruct an optimal parenthesization for each pair (i,j) we record in $c[i, j]=k$ the optimal split into two subproblems (i, k) and $(k+1, j)$

Multiplying Matrices/6

```
DynamicMM
for i := 1 to n do
    M[i,i] := í
for L := 1 to n-1 do
    for i := 1 to n-L do
        j := i+L
        M[i,j] := ❥
        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
                c[i,j] := k
return M, c
```

Multiplying Matrices/7

- After the execution: $M[1,n]$ contains the value of an optimal solution and c contains optimal subdivisions (choices of k) of any subproblem into two subsubproblems
- Let us run the algorithm on the four matrices:

  A_1 is a 2x10 matrix,

  A_2 is a 10x3 matrix,

  A_3 is a 3x5 matrix,

  A_4 is a 5x8 matrix.

Multiplying Matrices/8

- Running time
  - It is easy to see that it is $O(n^3)$ (three nested loops)
  - It turns out it is also $\Omega(n^3)$
- Thus, a reduction from exponential time to polynomial time.

Memoization

- If we prefer recursion we can structure our algorithm as a recursive algorithm:

```
MemoMM(i,j)
1.   if i = j then return 0
2.   else if M[i,j] < ∞ then return M[i,j]
3.   else for k := i to j-1 do
4.            q :=  MemoMM(i,k)+
                       MemoMM(k+1,j) + $d_{i-1}d_kd_j$
5.               if q < M[i,j] then
6.                   M[i,j] := q
7.     return M[i,j]
```

- Initialize all elements to ∞ and call **MemoMM(i,j)**

Memoization/2

- Memoization:
  - Solve the problem in a top-down fashion, but record the solutions to subproblems in a table.
- Pros and cons:
  - ☹ Recursion is usually slower than loops and uses stack space (not a relevant disadvantage)
  - ☺ Easier to understand
  - ☺ If not all subproblems need to be solved, you are sure that only the necessary ones are solved

Dynamic Programming

- In general, to apply dynamic programming, we have to address a number of issues:
  - Show **optimal substructure** – an optimal solution to the problem contains optimal solutions to sub-problems
    - Solution to a problem:
      - Making a choice out of a number of possibilities (look what possible choices there can be)
      - Solving one or more sub-problems that are the result of a choice (characterize the space of sub-problems)
    - Show that solutions to sub-problems must themselves be optimal for the whole solution to be optimal.

Dynamic Programming/2

- Write a recursive solution for the value of an optimal solution
  - M_{opt} = $\text{Min}_{\text{over all choices } k}$ {(Combination of M_{opt} of all sub-problems resulting from choice *k*) + (the cost associated with making the choice *k*)}
- Show that the number of different instances of sub-problems is bounded by a polynomial

Dynamic Programming/3

- Compute the value of an optimal solution in a bottom-up fashion, so that you always have the necessary sub-results pre-computed (or use memoization)
- Check if it is possible to reduce the space requirements, by “forgetting” solutions to sub-problems that will not be used any more
- Construct an optimal solution from computed information (which records a sequence of choices made that lead to an optimal solution)

Longest Common Subsequence

- Two text strings are given: *X* and *Y*
- There is a need to quantify how similar they are:
  - Comparing DNA sequences in studies of evolution of different species
  - Spell checkers
- One of the measures of similarity is the length of a Longest Common Subsequence (LCS)

LCS: Definition

- *Z* is a subsequence of *X* if it is possible to generate *Z* by skipping some (possibly none) characters from *X*
- For example: *X* =“ACGGTTA”, *Y*=“CGTAT”, LCS(*X*,*Y*) = “CGTA” or “CGTT”
- To solve LCS problem we have to find “skips” that generate LCS(*X*,*Y*) from *X* and “skips” that generate LCS(*X*,*Y*) from *Y*

LCS: Optimal Substructure

- We make Z to be empty and proceed from the ends of X_m="$x_1 x_2 \ldots x_m$" and Y_n="$y_1 y_2 \ldots y_n$"
  - If $x_m = y_n$, append this symbol to the beginning of Z, and find optimally LCS(X_{m-1}, Y_{n-1})
  - If $x_m \neq y_n$,
    - Skip either a letter from X
    - or a letter from Y
    - Decide which decision to do by comparing LCS(X_m, Y_{n-1}) and LCS(X_{m-1}, Y_n)
  - Starting from beginning is equivalent.

LCS: Recurrence

- The algorithm can be extended by allowing more “editing” operations in addition to *copying* and *skipping* (e.g., changing a letter)
- Let c[i,j] = LCS(X_i, Y_j)

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- Note that the conditions in the problem restrict sub-problems (if xi = yi we consider xi-1 and yi-1, etc)

LCS: Algorithm

```
LCS-Length(X, Y, m, n)
for i := 1 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
                        b[i,j] := "copy"
    else if c[i-1,j] $\geq$ c[i,j-1] then
      c[i,j] := c[i-1,j]
      b[i,j] := "skipX"
    else c[i,j] := c[i,j-1]
         b[i,j] := "skipY"
return c, b
```

LCS: Example

- Lets run:
  X =“GGTTCAT”, Y=“GTATCT”

- What is the running time and space requirements of the algorithm?
- How much can we reduce our space requirements, if we do not need to reconstruct an LCS?

Next Week

- Graphs:
  - Representation in memory
  - Breadth-first search
  - Depth-first search
  - Topological sort