Algorithm Design Techniques

- Dynamic Programming
  - Given a solution of a problem using smaller sub-problems, e.g., a recursive solution
  - Useful when the same sub-problems show up again and again in the solution

A simple case: Computing Fibonacci Numbers

- Recall $F_n = F_{n-1} + F_{n-2}$ and $F_0 = 0$, $F_1 = 1$
- Recursive algorithm:
  - `Fibo(n)`
    - if $n = 0$ then return(0)
    - else if $n = 1$ then return(1)
    - else return(Fibo(n-1)+Fibo(n-2))

Call tree - start

Memo-ization

- Remember all values from previous recursive calls
- Before recursive call, test to see if value has already been computed
- Dynamic Programming
  - Convert memo-ized algorithm from a recursive one to an iterative one
Fibonacci - Dynamic Programming Version

- FiboDP(n):
  - F[0] ← 0
  - F[1] ← 1
  - for i=2 to n do
    - F[i] = F[i-1] + F[i-1]
  - endfor
  - return(F[n])

Dynamic Programming

- Useful when
  - same recursive sub-problems occur repeatedly
  - Can anticipate the parameters of these recursive calls
  - The solution to whole problem can be figured out with knowing the internal details of how the sub-problems are solved
  - principle of optimality

List partition problem

- Given: a sequence of n positive integers s₁,...,sₙ and a positive integer k
- Find: a partition of the list into up to k blocks:
  - s₁,...,sᵢ₁ | sᵢ₁+1,...,sᵢ₂ | sᵢ₂+1,...,sᵢⱼ-1 | sᵢⱼ-1+1,...,sₙ
  - so that the sum of the numbers in the largest block is as small as possible.
  - i.e. find spots for up to k-1 dividers

Greedy approach

- Ideal size would be P = \( \sum_{i=1}^{n} \frac{s_i}{k} \)
- Greedy: walk along until what you have so far adds up to P then insert a divider
- Problem: it may not exact (or correct)
  - 100 200 400 500 900 | 700 600 | 700 600
  - sum is 4800 so size must be at least 1600.

Recursive solution

- Try all possible values for the position of the last divider
- For each position of this last divider
  - there are k-2 other dividers that must divide the list of numbers prior to the last divider as evenly as possible
    - s₁,...,sᵢ₁ | sᵢ₁+1,...,sᵢ₂ | sᵢ₂+1,...,sᵢⱼ-1 | sᵢⱼ-1+1,...,sₙ
    - recursive sub-problem of the same type

Recursive idea

- Let M[n,k] the smallest cost (size of largest block) of any partition of the n into k pieces.
- If between the jth and i+1st is the best position for the last divider then
  - M[n,k] = max ( M[i,k-1] , \( \sum_{j=i+1}^{n} s_j \) )
- In general
  - M[n,k] = minᵢ max ( M[i,k-1] , \( \sum_{j=i+1}^{n} s_j \) )
## Time-saving - prefix sums

- Computing the costs of the blocks may be expensive and involved repeated work

- Idea: Pre-compute prefix sums
  - \( p[1] = s_1 \)
  - \( p[2] = s_1 + s_2 \)
  - \( p[3] = s_1 + s_2 + s_3 \)
  - \( \ldots \)
  - \( p[n] = s_1 + s_2 + \ldots + s_n \)

  - cost: \( n \) additions, space \( n \)
  - Length of block \( s_{i+1} + \ldots + s_j \) is just \( p[j] - p[i] \)

## Linear Partition Algorithm

**Partition** \( \text{Partition}(S, k) \):

- \( p[0] \leftarrow 0; \) for \( i = 1 \) to \( n \) do \( p[i] \leftarrow p[i-1] + s_i \)
- for \( i = 1 \) to \( n \) do \( M[i, 1] \leftarrow p[i] \)
- for \( j = 1 \) to \( k \) do \( M[1, j] \leftarrow s_1 \)
- for \( i = 2 \) to \( n \) do
  - for \( j = 2 \) to \( k \) do
    - \( M[i, j] \leftarrow \infty \)
    - for \( pos = 1 \) to \( i-1 \) do
      - \( s \leftarrow \max(M[pos, j-1], p[i] - p[pos]) \)
      - if \( M[i, j] > s \) then
        - \( M[i, j] \leftarrow s; D[i, j] \leftarrow pos \)

\( \text{D}[i, j] \leftarrow \text{value of pos where min is achieved} \)