4 Dynamic Programming

[CLRS2, Chapter 15] (mostly) or [Par, Section 2.2] (different examples)

4.1 Coin changing problem (general)

[BB, Chapter 8.2]

Problem: We are given a set of coin denominations $d_1 < d_2 < \ldots < d_m$ and a value S. We have an infinite supply of each of the coins. We want to pay out a sum S with the least possible number of coins or determine that it is not possible.

How it can be not possible? $m = 1$, $d_1 = 2$, we can pay out only even sums.

Idea: As in Bentley's problem, solution 4, try to work from the "smaller" problems to "bigger" ones.

Let $coins[i]$ be the smallest number of coins to pay out sum i or ∞, if it is not possible.

Easy case: $coins[0] = 0$ (we do not need any coins to pay out 0)

Harder case: To compute $coins[i]$ for $i > 0$, we have to consider all possibilities for the first coin. When we pay out the first coin then the rest must be paid out optimally (otherwise we could "swap" that part of the solution for an optimal one and get a better solution overall).

$$coins[i] = 1 + \min_{j:d_j \leq i} coins[i - d_j]$$

Pseudocode:

```
coins[0]:=0;
for i:=1 to S do
  min:=infinity;
  for j:=1 to m do
    if d[j]<=i and coins[i-d[j]]<min then
      min:=coins[i-d[j]];
  coins[i]:=1+min;

return coins[S];
```

Time: $\Theta(mS)$

Example: $d = (1, 3, 4, 5)$, $S = 7$

```
        0   1   2   3   4   5   6   7
      -------------------------------
coins:  0   1   2   1   1   1   2   2
                        ^           |
                        +-----------+
```

How to recover the solution? Introduce new array where we remember the choices which we have made while determining each value of coin[i].

Note: To recover the whole solution it is sufficient to remember only the last choice as the previous choices can be recovered from the subproblems.

```
coins[0]:=0;
for i:=1 to S do
  min:=infinity;
  for j:=1 to m do
    if d[j]<=i and coins[i-d[j]]<min then
      min:=coins[i-d[j]];
*     minchoice:=j;
  coins[i]:=1+min;
* choice[i]:=minchoice;

// recover solution
if coins[S]=infinity then
  write 'No solution!';
else
  while S>0 do
    write d[choice[S]];
    S:=S-d[choice[S]];
```

We have discovered "dynamic programming"!

4.2 Dynamic programming - summary

[CLRS2, Chapter 15.3]

A bit of history

- Dynamic programming introduced by Richard Bellman in 1955.
- The word "programming" at the time referred to filling values of a big matrix in some prespecified way and looking for the solution somewhere in the cells of that matrix.
- Note the common meaning of word "programming" in both "linear programming" and "dynamic programming".

Let us look at the dynamic programming from the point of view of "filling a matrix".

How to design a dynamic programming algorithm?

1. **Identify subproblem.** Usually the structure of subproblems corresponds to a big (1D, 2D, 3D, ...) matrix. We need to:
   - give matrix dimensions
   - give the precise meaning to each cell in the matrix
   - specify, where to find an answer to the whole problem

   e.g.: *In coin changing we had a 1D matrix coins*$[1..S]$*, where coins*$[i]$ *meant "the number of coins necessary to pay sum* i *(or* ∞*, if sum* i *cannot be paid)". Answer will be contained in coins*$[S]$*.*

2. **Define a subproblem in terms of smaller subproblems.** How to compute value of every cell in the matrix from other cells in the matrix?
   e.g.: $coins[i] = 1 + \min_{j:d_j \leq i} coins[i - d_j]$

3. **Set base cases.** Which cells cannot be computed using formula(s) from the previous step? What values should they contain?
   e.g.: $coins[0] = 0$

4. **Choose an order of evaluation.** In what order we need to fill the matrix so that in every step we have all information needed to compute the cell's value?
   e.g.: *In case of coin changing, left-to-right.*

Note: You need to try to start working from the first step. If you get "stuck" during the design, you need to find out why did you get stuck and then change the design of your table in step 1 and start again from scratch. Not all definitions of subproblem lead to a succesful dynamic programming algorithm.

How to recover solution from dynamic programming?

1. **Keep track of which subproblems were used to solve each large problem.** Which case in formula we used to compute the value of the cell?
   e.g.: *In coin changing we keep track with which "first coin" we were able to find the best solution.*

2. **Trace back through the table.**
   e.g.: *If the first coin was i in subproblem j, then the next bit of solution can be found in a cell corresponding to a subproblem $j - i$.*

4.3 Memoization

Another implementation of the recurrence formula for coin changing.

Pseudocode 1:

```
function coins(i):
  // base cases
  if (i=0) then return 0;

  // recursion:
  min:=infinity;
  for j:=0 to m do
    if (d[j]<=i) then
      smaller_sol:=coins(i-d[j]);
      if smaller_sol<min then min:=smaller_sol;

  return 1+min;

// ------------
// main program
return change_coins(S);
```

Example: Draw the recursion tree for coin system $(1, 2)$ and $S = 5$. Note that many subproblems are solved multiple times.

Time: exponential in S
For coin system (1,2):
How many leaves there are in the tree?
$L(0) = 1, L(1) = 1, L(n) = L(n-1) + L(n-2)$; thus $L(S) = F_S = \Theta((\frac{1+\sqrt{5}}{2})^S)$.
How many nodes there are in the tree?
$L(S) \leq N(S) \leq 3L(S)$ and therefore $N(S) = \Theta(L(S)) = L(S) = F_S = \Theta((\frac{1+\sqrt{5}}{2})^S)$
How much time we spend in each node? $\Theta(1)$ (because $m = 2$)
Total: $\Theta((\frac{1+\sqrt{5}}{2})^S)$.

Idea: Avoid multiple computation of the same subproblem

Pseudocode 2:

```
function coins(i):
* if (coins[i] is initialized) return coins[i]
  // base cases
  if (i=0) then return 0;

  // recursion:
  min:=infinity;
  for j:=0 to m do
    if (d[j]<=i) then
      smaller_sol:=coins(i-d[j]);
      if smaller_sol<min then min:=smaller_sol;

* coins[i]=1+min;
  return 1+min;

// ------------
// main program
return change_coins(S);
```

Time: $O(mS)$

- computation of each subproblem takes $\Theta(m)$ (not counting recursive calls)
- each subproblem is computed at most once and there are S subproblems.

This is another form of dynamic programming called memoization.

Advantages: Only subproblems that are needed are computed.

Disadvantages: More complicated code, more complicated analysis, overhead for recursion.

4.4 Longest common subsequence

[CLRS2, Chapter 15.4]

Motivation:

- How similar two computer files are? What is behing UNIX `diff` command?
- How similar are humans and chimpanzees? One possible way of looking at this problem is to look at the similarities between their DNA sequences. From our point of view, the DNA sequences are strings over the alphabet $\{A, C, G, T\}$. The following problem gives one way of defining how similar the two DNA sequences are.
- How similar two assignment submissions are? Let's not go there right now...

Notation: A *subsequence* of a string X is a string which can be obtained by deleting some of the characters from X.

Problem: **Longest common subsequence.** We are given two strings $X = x_1 \dots x_m$ and $Y = y_1 \dots y_n$. Find the longest string Z which is a subsequence of both X and Y.

Example:
ALGORITHM
LOGARITHM

Subproblem: $C[i, j]$ is the length of the longest common subsequence of the strings $X[1 \dots i]$ and $Y[1 \dots j]$. The answer of the problem can thus be found in $C[m, n]$.

Recurrence: Let $Z = z_1 z_2 \dots z_k$ be the LCS of $X[1 \dots i]$ and $Y[1 \dots j]$.

- If $x_i = y_j = c$ then $z_k = c$ and $Z[1 \dots k-1]$ is the LCS of $X[1 \dots i-1]$ and $Y[1 \dots j-1]$.

  Justification:

  - Assume that $z_k \neq c$. Then we can append Z with c obtaining longer common subsequence — contradiction (we assumed Z is the LCS)
  - Assume that $Z[1 \dots k-1]$ is not the LCS of $X[1 \dots i-1]$ and $Y[1 \dots i-1]$. Then we can take the LCS and replace $Z[1 \dots k-1]$ with it obtaining longer common subsequence of $X[1 \dots i]$, $Y[1 \dots j]$.

- If $x_i \neq y_j$ then $Z[1 \dots k]$ is the LCS of:
  - either $X[1 \dots i-1]$, $Y[1 \dots j]$,
  - or $X[1 \dots i]$, $Y[1 \dots j-1]$.

  Justification: Because $x_i \neq y_j$, the last character of the LCS is either not x_i or not y_j (maybe neither).

  - If $z_k \neq x_i$ then Z must be the LCS of $X[1 \dots i-1]$ and $Y[1 \dots j]$. (If there was a longer common subsequence, we could use it instead and obtain a better solution.)
  - Similarly, if $z_k \neq y_j$ then Z must be the LCS of $X[1 \dots i]$, $Y[1 \dots j-1]$.

$$C[i,j] = \begin{cases} C[i-1, j-1] + 1, & \text{if } x_i = y_j, \\ \max\{C[i-1, j], C[i, j-1]\}, & \text{otherwise.} \end{cases}$$

Base cases: The formula cannot be used if $i = 0$ or if $j = 0$. Clearly, $C[0, j] = 0$, $C[i, 0] = 0$, $C[0, 0] = 0$.

Order of evaluation: To compute formula for $C[i, j]$, we need to know $C[i-1, j], C[i, j-1], C[i-1, j-1]$. Computing the cells from the top-most corner by rows will ensure that these values will be ready before computing $C[i, j]$.

Pseudocode:

```
// base cases
for i:=0 to m do C[i,0]:=0;
for j:=0 to n do C[0,j]:=0;

// filling the matrix
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-1,j],C[i,j-1]);

return C[m,n];
```

Time: $\Theta(mn)$

Example:

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

To recover solution: We need to remember which choice we used to compute $C[i, j]$.

$$D[i,j] = \begin{cases} \text{upleft}, & \text{if } C[i,j] = 1 + C[i-1, j-1] \\ \text{up}, & \text{if } C[i,j] = C[i-1, j] \\ \text{left}, & \text{if } C[i,j] = C[i, j-1] \end{cases}$$

(you can easily add the corresponding lines of code to the pseudocode above)

How do we use values of D?

```
row:=m; col:=n;
lcs:="";

while (row>0 and col>0) do
  if (D[row,col]=upleft) then
    // X[row]=Y[col]
    lcs:=lcs.X[row];
    row:=row-1; col:=col-1;

  else if (D[row,col]=up) then
    row:=row-1;

  else if (D[row,col]=left) then
    col:=col-1;

reverse lcs;
return lcs;
```

4.5 Knapsack problem

[BB, Chapter 8.4]

Problem: We are given n objects and a knapsack. Each object i has a positive weight w_i and a positive value v_i. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.

Example:
Objects (weight,value): (6,6), (5,5), (5,5)
$W = 10$
Optimal solution: (5,5), (5,5)

4.5.1 First try

Note: The problem is very similar to coin changing – why not try the same approach

Subproblem: $V[j]$ – the maximum value of the objects which can fit into the knapsack of size j.
Solution will be found in $V[j]$.

Recurrence: (Try every possibility for the first element in the array)

$$V[j] = \max_{i\,:\,w_i \le j} \{v_i + V[j - w_j]\}$$

What is wrong with this recurrence?

4.5.2 Second try

Subproblem: $V[i, j]$ – the maximum value of the objects which can fit into the knapsack of size j and including only objects $1, \ldots, i$.
Solution will be found in $V[n, W]$.

Recurrence: (We either use the object i or not.)

$$V[i,j] = \max\{V[i-1,j], V[i-1,j-w_i] + v_i\}$$

Base cases: $V[0,j] = 0$

Order of computation: Row-order from top-left to botom-right corner.

```
for j:=0 to W do
  V[0,j]:=0;

for i:=1 to n do
  for j:=1 to W do
    sol:=V[i-1,j];
    if (w[i]<=j) then
      othersol:=V[i-1,j-w[i]]+v[i];
      if (othersol>sol) then
        sol:=othersol;
    V[i,j]:=sol;

return V[n,W];
```

Notes about the code: we can make it more memory efficient.
Note that to compute cell $V[i,j]$, we need only the cells from the previous line and to the left of $V[i-1,j]$.

```
for j:=0 to W do
* V[j]:=0;

for i:=1 to n do
* for j:=W downto 1 do
*   sol:=V[j];
    if (w[i]<=j) then
*     othersol:=V[j-w[i]]+v[i];
      if (othersol>sol) then
        sol:=othersol;
*   V[j]:=sol;

return V[W];
```

More simplification...

```
for j:=0 to W do
* V[j]:=0;

for i:=1 to n do
* for j:=W downto 1 do
    if (w[i]<=j) then
*     othersol:=V[j-w[i]]+v[i];
*     if (othersol>V[j]) then
*       V[j]:=othersol;

return V[W];
```

4.6 Shortest triangulation of convex polygon

[CLR1, Chapter 16.4]

Computational geometry. Computational geometry is a branch of computer science that studies algorithms for solving geometric problems. It has applications in computer graphics, robotics, VLSI design, computer-aided design, statistics, numerical computations, and other areas.

How old is computational geometry? It is a new discipline but much older than computers. Perhaps the first work in the area was work of Emile Lemoine (1902) who studied how many operations you need to achieve different constructions with compass and straight-edge ruler.

...

Convex polygon. Polygon is *convex*, if for any two points on its boundary, all points on the line segment drawn between them are contained in the polygon's interior.

Convex polygon can be represented by listing its vertices in clockwise order.

Triangulation. Every convex polygon can be split into disjoint triangles by a set of non-crossing *chords* – segments connecting non-adjacent vertices of the polygon.

Length of triangulation is the length of all chords in the triangulation plus the length of the polygon boundary.

Note: Triangulations are important in both graphics and numerical algorithms applications. They decompose possibly complex polygon into a set of simple shapes (triangles) allowing for simpler methods for dealing with such polygons. We may want to find a triangulation for which some parameter is optimized – in our example we will be looking for the shortest triangulation.

Problem: Given is a convex polygon by listing its vertices in clockwise order $(v_1, v_2, \ldots, v_n)$. Find triangulation which has the shortest length.

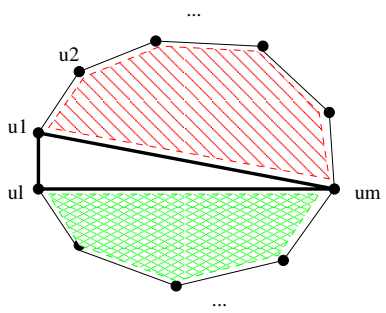
Notation: $d(v_i, v_j)$ – Euclidian distance of vertices v_i, and v_j.

Solution by dynamic programming.

Subproblem: $t[u_1, \ldots, u_l]$ – the shortest length of triangulation for polygon $(u_1, u_2, \ldots, u_n)$, where $u_1, \ldots, u_l$ is a subsequence of $v_1, \ldots, v_n$.

Note: There are 2^n of such subproblems. This is too many; even if each of them could be computed in a constant time, the algorithm would still run in exponential time. Let's not worry about it right now.

Recurrence: Edge (u_l, u_1) is part of a triangle in an optimal triangulation – denote the third vertex of that triangle u_m. The cost of such triangulation must be $d(u_1, u_l) + t[u_1, \ldots, u_m] + t[u_m, \ldots, u_l]$.



Thus trying all possibilities for a vertex u_m we have:

$$t[u_1, \ldots, u_l] = \min_{1<m<l} \{d(u_1, u_l) + t[u_1, \ldots, u_m] + t[u_m, \ldots, u_l]\} \tag{1}$$

Base cases: We cannot use the formula when we have less than three vertices (we never have less than two):

$$t[a, b] = d(a, b)$$

Note: To compute $t[v_1, \ldots, v_n]$ using the formula above we only need to compute subproblems of the form $t[v_i, v_{i+1}, \ldots, v_j]$ for all $i < j$. Thus we need to compute only $O(n^2)$ subproblems.

Denote $T[i, j] := t[v_i, v_{i+1}, \ldots, v_j]$ and $D[i, j] = d(v_i, v_j)$. We can rewrite the recurrence (1) as follows:

$$T[i, j] = \min_{i<m<j} \{D[i, j] + T[i, m] + T[m, j]\}$$

$$T[i, i + 1] = D[i, i + 1]$$

Order of computation: The recursive formula always use subproblems with smaller number of points. Therefore we can order computation by $j - i$ (from smallest polygons to the largest).

```
// base case - j=i+1
for i:=1 to n-1 do
  T[i,i+1]:=D[i,i+1];

for delta:=2 to n-1 do
  // cases where j-i=delta
  for i:=1 to n-delta do
    j:=i+delta;
    T[i,j]:=infinity;
    // try all possible triangles v_i,v_j,v_m
    for m:=i+1 to j-1 do
      cost:=D[i,j]+T[i,m]+T[m,j];
      if cost<T[i,j] then
*       T[i,j]:=cost;

return T[1,n];
```

Time: $\Theta(n^3)$

Solution recovery: This time a little bit tricky – we have always TWO subproblems from which the optimal solution is constructed. We will use recursive function.

Let $M[i,j]$ is m used to compute T[i,j] (note that the pseudocode above can be easily modified at line marked by * to compute M as well).

```
function give_solution(i,j)
  output edge (i,j);
  if j>i+1 then
    give_solution(i,M[i,j]);
    give_solution(M[i,j],j);
```