NOPT042 Constraint programming: Tutorial 6 – Scheduling

In [1]:
```
%load_ext ipicat
```

```
Picat version 3.5#5
```

Example from the last tutorial: Swimmers

(From: W. Winston, Operations Research: Applications & Algorithms.)

In medley swimming relay, a team of four swimmers must swim 4x100m, each swimmer using a different style: breaststroke, backstroke, butterfly, or freestyle. The table below gives their average times for 100m in each style. Which swimmer should swim which stroke to minimize total time?

Swimmer	Free	Breast	Fly	Back
A	54	54	51	53
B	51	57	52	52
C	50	53	54	56
D	56	54	55	53

Write a general model, generate larger instances, and try to make your model as efficient as possible.

In [2]:
```
!cd swimmers && picat instances.pi
```

```
{A,B,C,D}
{Free,Breast,Fly,Back}
{{54,54,51,53},{51,57,52,52},{50,53,54,56},{56,54,55,53}}
{Swimmer1,Swimmer2,Swimmer3,Swimmer4,Swimmer5}
{Style1,Style2,Style3,Style4,Style5,Style6,Style7}
{{46,49,54,64,53,55,55},{54,60,55,47,64,65,49},{65,52,48,60,61,61,62},{59,57,54,47,5
0,50,58},{46,64,50,45,48,57,62}}
```

In [3]:
```
!cd swimmers && picat swimmers.pi
```

```
Primal model:
Swimmer A is swims Fly
Swimmer B is swims Back
Swimmer C is swims Free
Swimmer D is swims Breast

Dual model:
Style Free is swum  by C
Style Breast is swum  by D
Style Fly is swum  by A
Style Back is swum  by B

Channeling model:
Swimmer A is swims Fly
Swimmer B is swims Back
Swimmer C is swims Free
Swimmer D is swims Breast
or in the dual view
Style Free is swum  by C
Style Breast is swum  by D
Style Fly is swum  by A
Style Back is swum  by B
```

In [4]:
```
!cat swimmers/swimmers.pi
```

```
import cp.

main =>
    cl(sample_instance),
    solve_primal,
    solve_dual,
    solve_channeling.

solve_primal =>
    sample_instance(SwimmerNames, StyleNames, Times),
    primal_model(StyleOfSwimmer, Times, TotalTime),
    solve([$min(TotalTime)], StyleOfSwimmer),
    println("\nPrimal model:"),
    foreach(I in 1..SwimmerNames.length)
        printf("Swimmer %w is swims %w\n", SwimmerNames[I], StyleNames[StyleOfSwimme
r[I]])
    end.

solve_dual =>
    sample_instance(SwimmerNames, StyleNames, Times),
    dual_model(SwimmerOfStyle, Times, TotalTime),
    solve([$min(TotalTime)], SwimmerOfStyle),
    println("\nDual model:"),
    foreach(J in 1..StyleNames.length)
        printf("Style %w is swum  by %w\n", StyleNames[J], SwimmerNames[SwimmerOfSty
le[J]])
    end.

solve_channeling =>
    sample_instance(SwimmerNames, StyleNames, Times),
    channeling_model(StyleOfSwimmer, SwimmerOfStyle, Times, TotalTime),
    solve([$min(TotalTime)], StyleOfSwimmer ++ SwimmerOfStyle),
    println("\nChanneling model:"),
    foreach(I in 1..SwimmerNames.length)
        printf("Swimmer %w is swims %w\n", SwimmerNames[I], StyleNames[StyleOfSwimme
r[I]])
    end,
    println("or in the dual view"),
    foreach(J in 1..StyleNames.length)
        printf("Style %w is swum  by %w\n", StyleNames[J], SwimmerNames[SwimmerOfSty
le[J]])
    end.


primal_model(StyleOfSwimmer, Times, TotalTime) =>
    N = Times.length,
    %K = Times[1].length,
    K = N,
    StyleOfSwimmer = new_array(N),
    StyleOfSwimmer :: 1..K,
    all_different(StyleOfSwimmer),

    TimeOfSwimmer = new_array(N),
    foreach(I in 1..N)
        matrix_element(Times, I, StyleOfSwimmer[I], TimeOfSwimmer[I])
        % matrix_element(Matrix, I, J, Entry)
```

```
    end,
    TotalTime #= sum(TimeOfSwimmer).


dual_model(SwimmerOfStyle, Times, TotalTime) =>
    N = Times.length,
    K = N,
    SwimmerOfStyle = new_array(K),
    SwimmerOfStyle :: 1..N,
    all_different(SwimmerOfStyle),

    TimeOfStyle = new_array(K),
    foreach(J in 1..K)
        matrix_element(Times, J, SwimmerOfStyle[J], TimeOfStyle[J])
    end,
    TotalTime #= sum(TimeOfStyle).


channeling_model(StyleOfSwimmer, SwimmerOfStyle, Times, TotalTime) =>
    primal_model(StyleOfSwimmer, Times, TotalTime),
    dual_model(SwimmerOfStyle, Times, TotalTime),
    assignment(StyleOfSwimmer, SwimmerOfStyle).  %chanelling constraint
```

Modelling functions

In general, how to model a function (mapping) $f : A \to B$? Let's say $A = \{1, \ldots, n\}$ and $B = \{1, \ldots, k\}$.

- as an array:

  ```
  F = new_array(N),
  F :: 1..K.
  ```

- *injective*: `all_different(F)`
- *dual model*: switch the role of variables and values (not a function unless F injective, see above): a partition of A into classes labelled by B, to each element of B map a set of elements of A. In Picat we can model set as their characteristic vectors. The sum of the vectors should be all 1s. More on modelling with sets later.
- *surjective*: No characteristic vector can be all 0s.
- *partial function*: a dummy value for undefined inputs
- *channelling*: combine the primal and dual models, if it is a bijection, then use `assignment(F, FInv)`

In [5]:
```
!picat functions.pi 4 4
```

```
{1,2,3,4}
{1,2,3,4}
```

Scheduling

Example: moving

A simple scheduling problem: Four friends are moving. The table shows how much time and how many people are necessary to move each item. Schedule the moving to minimize total time. (Adapted from R. Barták's tutorial; check the SICStus Prolog model.)

Item	Time (min)	People
piano	45	4
chair	10	1
bed	25	3
table	15	2
couch	30	3
cat	15	1

In [6]:
```
!cat moving/instance.pi
```

```
instance(NumPeople, Items, Duration, People) =>
    NumPeople = 4,
    Items = ["piano", "chair", "bed", "table", "couch", "cat"],
    Duration = [45, 10, 25, 15, 30, 15],
    People = [4, 1, 3, 2, 3, 1].
```

The `cummulative` global constraint

For the above problem we can use the following global constraint:

```
cumulative(StartTimes, Durations, Resources, Limit)
```

which means that we have `Limit` of resource available, each item starts at `StartTimes[i]`, takes `Durations[i]` time and consumes `Resources[i]` of the resource.

Homework: moving

Generalize the moving problem from the tutorial to include:

- trolleys: some items require a one or more trolleys to be moved; we have a certain amount of trolleys
- precedence: some items need to be moved before other items (given as a list of pairs), `[item1,item2]` means that moving of item1 must be finished before moving of item2 starts.

See the `hw-instance.pi`. Your model should accept a filename, e.g.

```
picat moving.pi hw-instance.pi
```

The autograder will only test for the presence of the optimal time (in minutes) in the output but include also some reasonable output of the schedule.

In [7]:
```
!cat moving/hw-instance.pi
```

```
% moving with trolleys and precedence (for the homework)
instance(NumPeople, NumTrolleys, Items, Duration, People, Trolleys, Precedence) =>
    NumPeople = 4,
    NumTrolleys = 2,
    Items = ["piano", "chair", "bed", "table", "couch", "cat", "fridge"],
    Duration = [50, 10, 25, 15, 30, 15, 60],
    People = [4, 1, 3, 2, 3, 1, 2],
    Trolleys = [2, 0, 2, 1, 2, 0, 1],
    Precedence = [["couch", "cat"], ["fridge", "cat"], ["bed", "piano"]]. % finish m
oving the couch before starting moving the cat
```