OptSeqによる
スケジューリング問題
のモデリングと事例

Log Opt Co., Ltd.

目次1

- スケジューリング最適化ソルバーOptSeqの概要，特徴
- Pythonインタフェイスの使い方
- Pythonインタフェイスを用いた例題

Part1
- **航空機離陸問題1：作業，モード，時間制約**
- **航空機離陸問題2：資源制約**

Part2
- **並列ショップスケジューリング1:資源制約（複数資源）**
- **並列ショップスケジューリング2：複数モード**

Part3
- **お家を早く造ろう：（任意の形の）一般化資源制約**

Part4
- **航空機離陸問題3：再生不能資源**
- **航空機離陸問題4：同時開始**

目次2

- Pythonインタフェイスを用いた例題
  - Part5
    - 並列ショップスケジューリング3：並列作業
  - Part6
    - 納期遅れ最小化問題1：納期遅れ
    - 納期遅れ最小化問題2：納期遅れ，一般化資源
    - 納期遅れ最小化問題3：納期遅れ，作業の中断
  - Part7
    - ジョブショップスケジューリング：並列作業，作業中断，資源の占有
  - Part8
    - 段取りのある生産問題1：状態変数
    - 段取りのある生産問題2：状態変数
  - Part9
    - 初期解の設定方法
- 適用例

概要と特徴

スケジューリングモデルとは

- 作業（activity；活動）：遂行すべき仕事のこと.
  別名，オペレーション，ジョブ，仕事，タスク.
  作業時間や納期の情報が付加される.
- 時間制約（time constraint）：作業と他の作業の間の時間関係を表す制約.
- 資源（resource）：作業を行うときに必要とされるもので，その量に限りがあるもの.
  機械(machine）はジョブによって占有される資源.
  その他，人員，材料，お金なども全部資源.

スケジューリング最適化ソルバーOptSeqとは

大規模なスケジューリング問題を高速に解くためのソルバー

- 茨木先生（京都大学名誉教授）と野々部先生（法政大学）の開発したメタヒューリスティクスを基礎
- トライアルバージョン（１５変数まで）
  http://logopt.com/OptSeq/OptSeq.htm

1. 簡易モデリング言語（テキスト）によるモデル作成
2. Pythonからの呼び出し
3. ライブラリ呼び出しによる利用が可能
   （C++, Visual Basic, C# ）

OptSeqの特徴

- 誰でも効率の良い定式化を簡単に書ける
  - 混合整数最適化ソルバーと異なり定式化の強弱によらない
- スケジューリング問題に特化したソルバー
- スケジューリング問題の考え方で自然なモデル化ができる
- 様々な実務的な付加条件に対応
  - 作業の多モード化
  - 一般化時間制約
  - 一般化資源制約，再生不能資源制約
  - 中断可能な作業と作業の並列化
  - 状態変数

Pythonインタフェイス

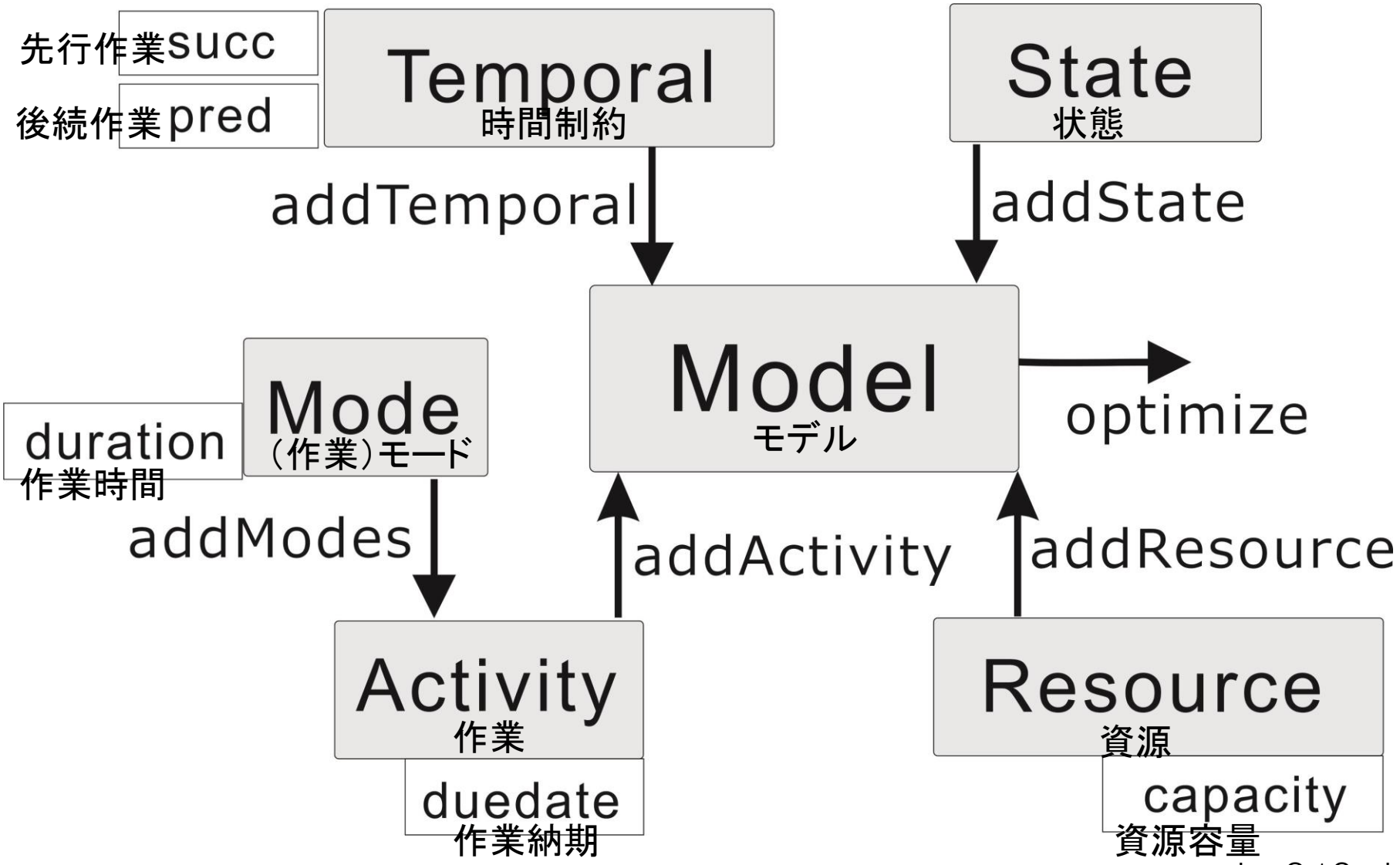
optseq.pyモジュールとは

- スケジューリング最適化ソルバーOptSeqを，Pythonから直接呼び出して，求解するためのモジュール
- すべてPythonで書かれたクラスで構成
- ソースコードはユーザに公開されているので，ユーザが書き換え可能

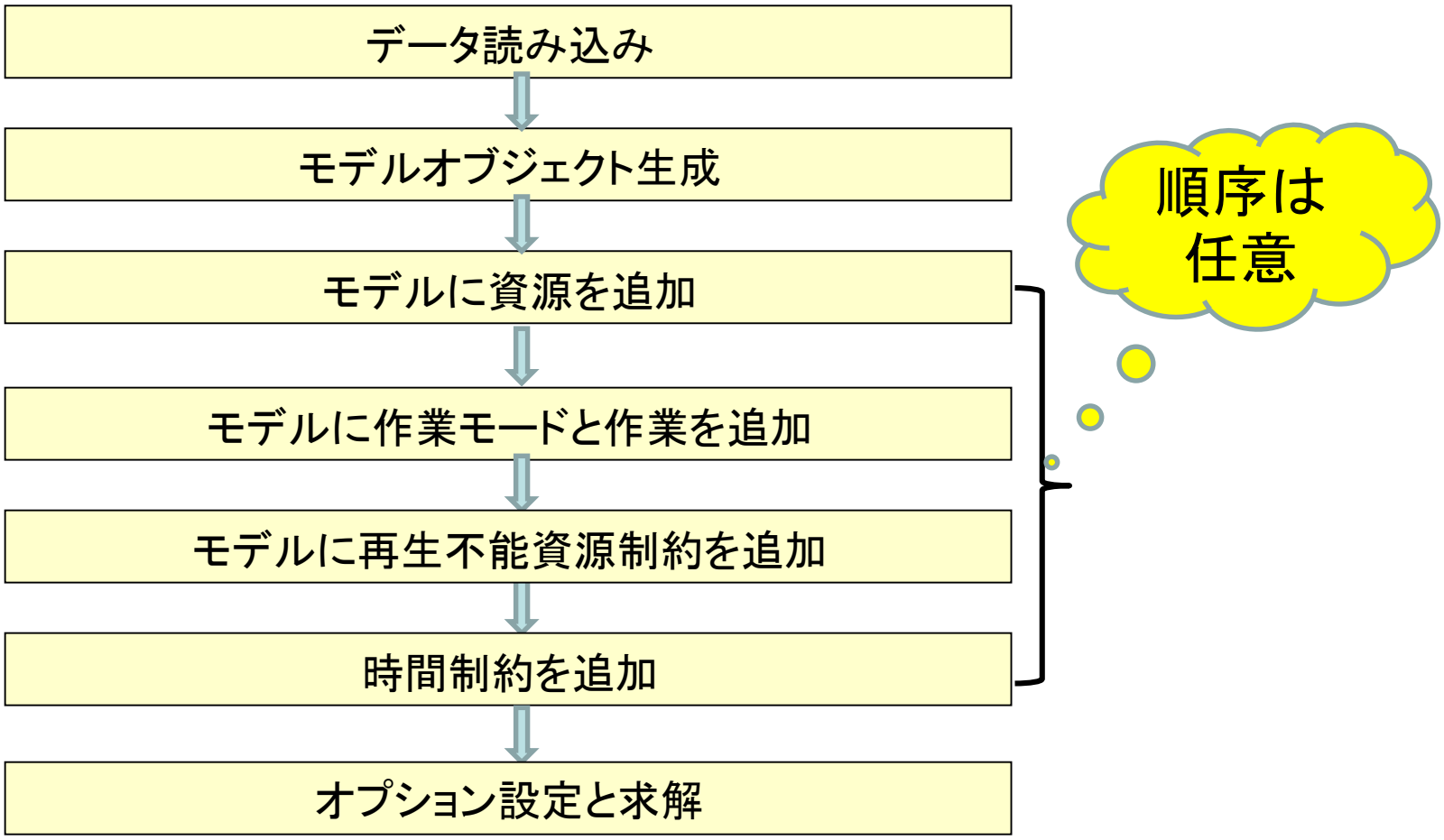
Pythonモジュールのクラス

- モデルクラス　ー　Model
- 作業クラス　ー　Activity
- モードクラス　ー　Mode
- 時間制約クラス　ー　Temporal
- 資源クラス　ー　Resource
- 状態クラス　ー　State
  - モデル内の状態変数を表すためのクラス
- パラメータクラス　ー　Params
  - アルゴリズムのパラメータを設定するためのクラス

クラス間の関係と主要なメソッド・属性



optseq.pyモジュールを用いてのモデル作成手順



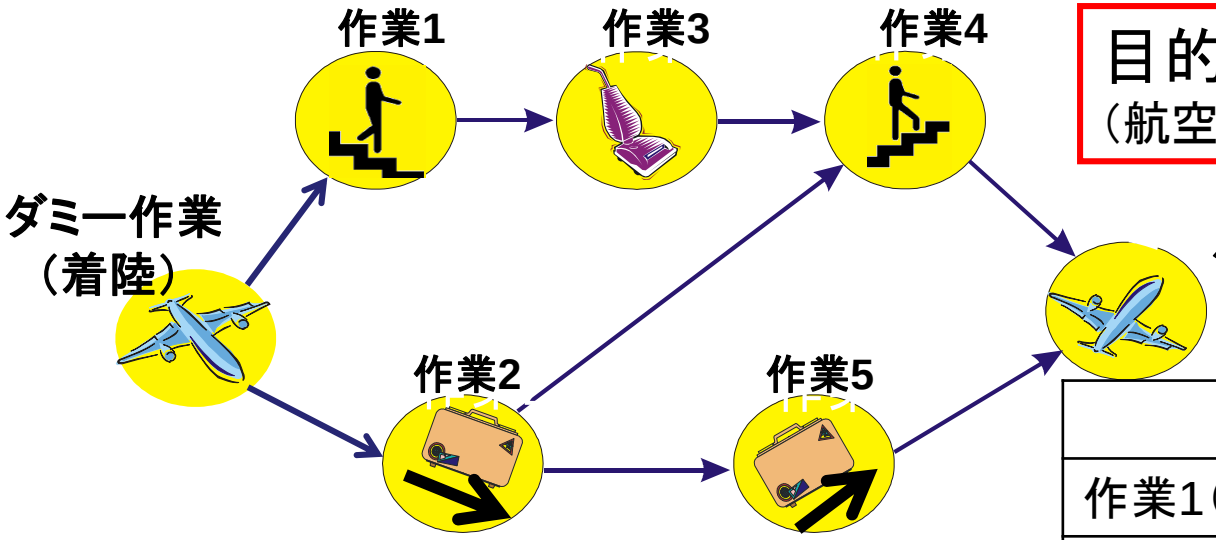
例題Part 1

航空機離陸問題1
航空機離陸問題2

時間制約，資源制約

航空機離陸問題1
時間制約

- PERT: Program Evaluation and Review Technique,第二次世界大戦のポラリス潜水艦の建造で利用.（ORの古典）



目的：完了時刻最小化
（航空機をなるべく早く離陸させる）

矢印は作業間の先後順位を表す.
eg：作業1が終了しないと作業3は開始できない

	作業時間
作業1（乗客降ろし）	13
作業2（荷物降ろし）	25
作業3（機内清掃）	15
作業4（乗客搭乗）	27
作業5（荷物積み込み）	22

Pythonインターフェイスによる記述(1)
(航空機離陸問題1:Example1.py)

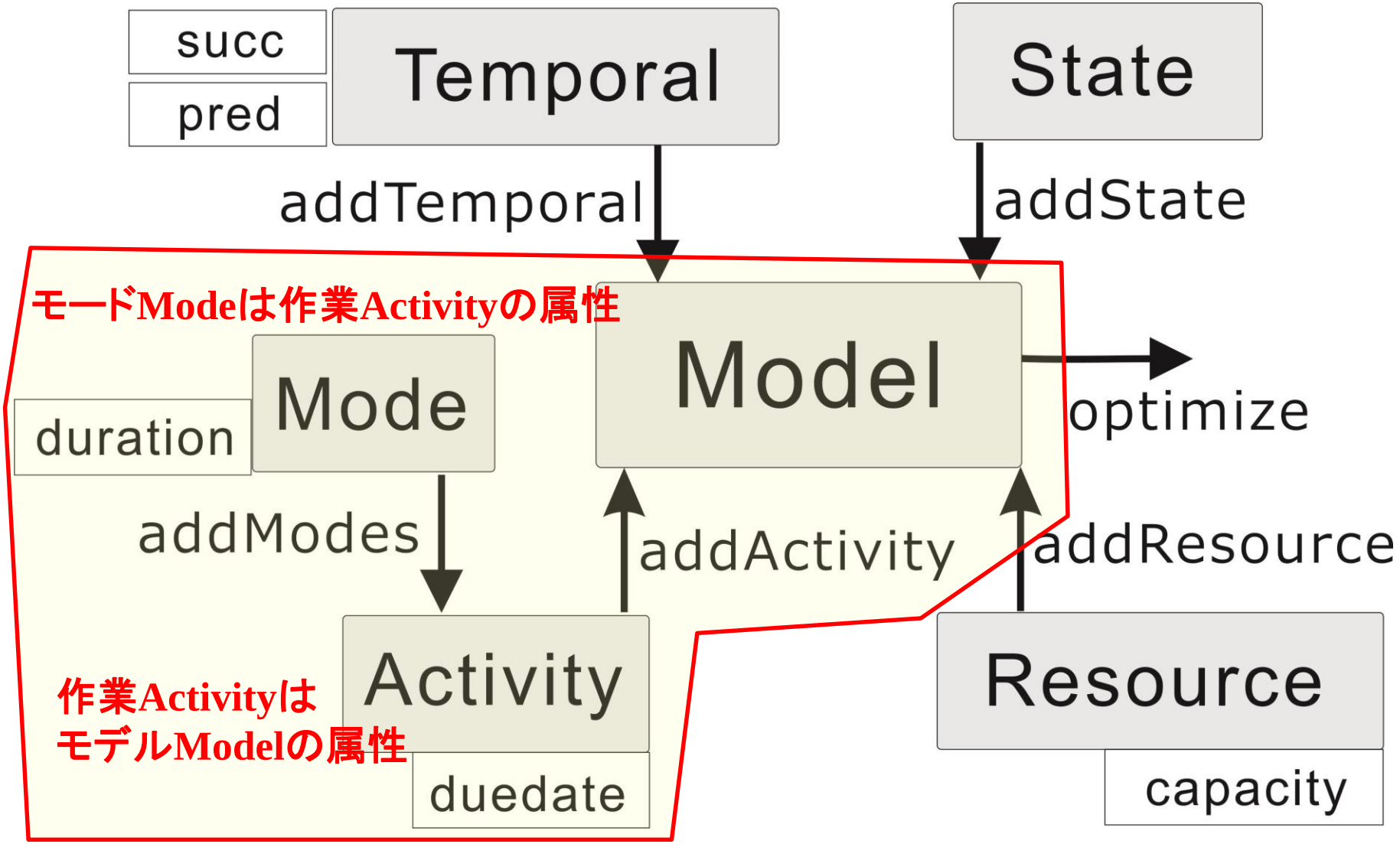
無料配布
例題集の
ファイル名

```
from optseq import *    #optseq.pyモジュールを読み込む
m1=Model()              #m1と言うモデルオブジェクトを生成
#作業時間を辞書で表す
duration ={1:13, 2:25, 3:15, 4:27, 5:22 }
act={} #作業を保管するための辞書を準備
mode={} #作業モードを保管するための辞書を準備
```

5:22:作業5の通常の作業時間が22分であることを表す.

Pythonインターフェイスによる記述(2)

(航空機離陸問題1:Example1.py)



Pythonインターフェイスによる記述（3）
（航空機離陸問題1：Example1.py）

```
for i in duration:
    #作業をモデルm1に追加（作業はモデルの属性であるため）
    act[i]=m1.addActivity('Act[{0}]'.format(i))
    #作業モードを保管するための辞書modeに作業モードオブジェクトを追加
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    #上で作成した各作業のモードオブジェクトを作業へ追加（モードは作業の属性であるため）
    act[i].addModes(mode[i])
```

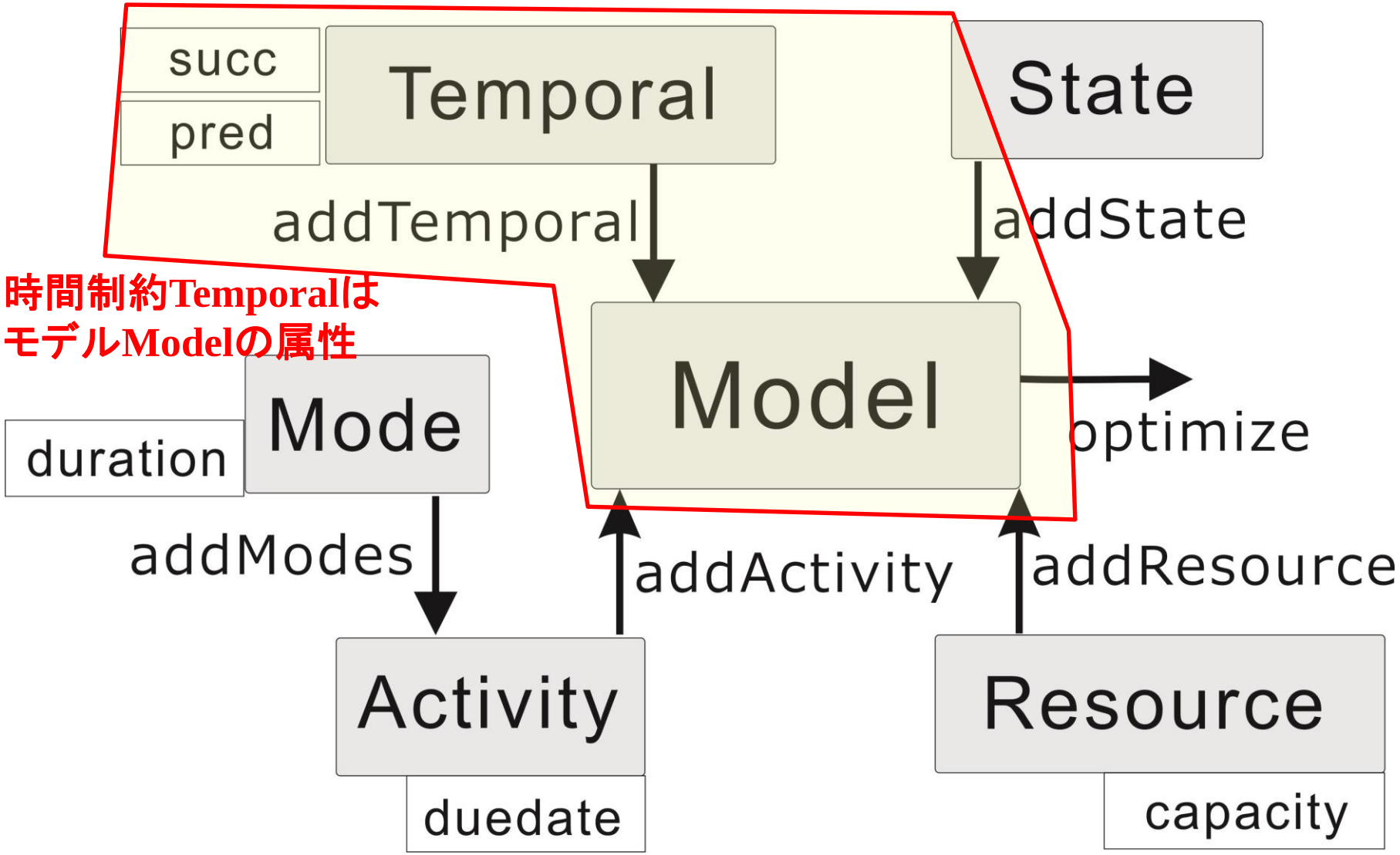
作業追加メソッド addActivity(作業名)

モードオブジェクト作成：Mode(モード名,モード作業時間)

モード追加メソッドaddModes(モードオブジェクトをカンマ区切りで入力)

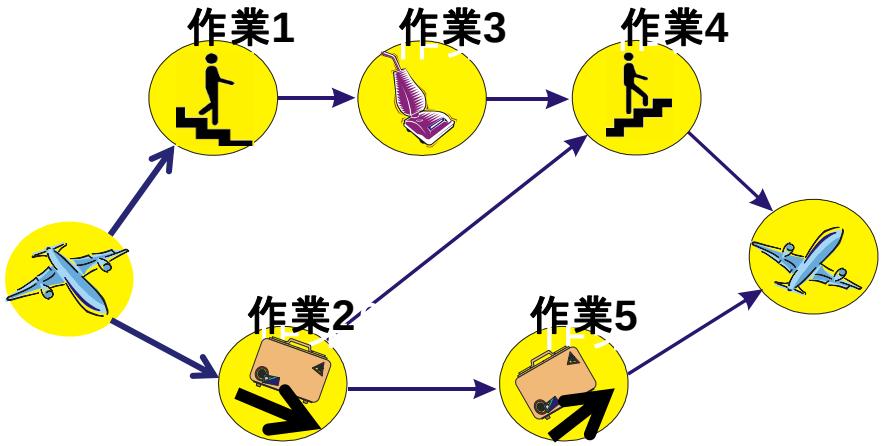
Pythonインターフェイスによる記述(4)

(航空機離陸問題1:Example1.py)



Pythonインターフェイスによる記述(5)
(航空機離陸問題1:Example1.py)

```
#モデルへ時間制約を追加
m1.addTemporal(act[1],act[3])
m1.addTemporal(act[2],act[4])
m1.addTemporal(act[2],act[5])
m1.addTemporal(act[3],act[4])
```



時間制約追加メソッドaddTemporal(
先行作業, 後続作業 , '制約タイプ' ,時間ずれ)

規定値:CS　　規定値:0

CSはcompletion, start:先行作業終了後に後続作業を開始することを表す.

*時間制約についてはExample8で詳しく説明する

Pythonインターフェイスによる記述(6)
(航空機離陸問題1:Example1.py)

```
m1.Params.TimeLimit=1          #求解時間設定

m1.Params.Makespan=True        #最大完了時刻最小化の場合True,
                                 納期遅れ最小化などその他の場合
                                 False(規定値)を設定する.

m1.optimize()                  #モデルを求解する
```

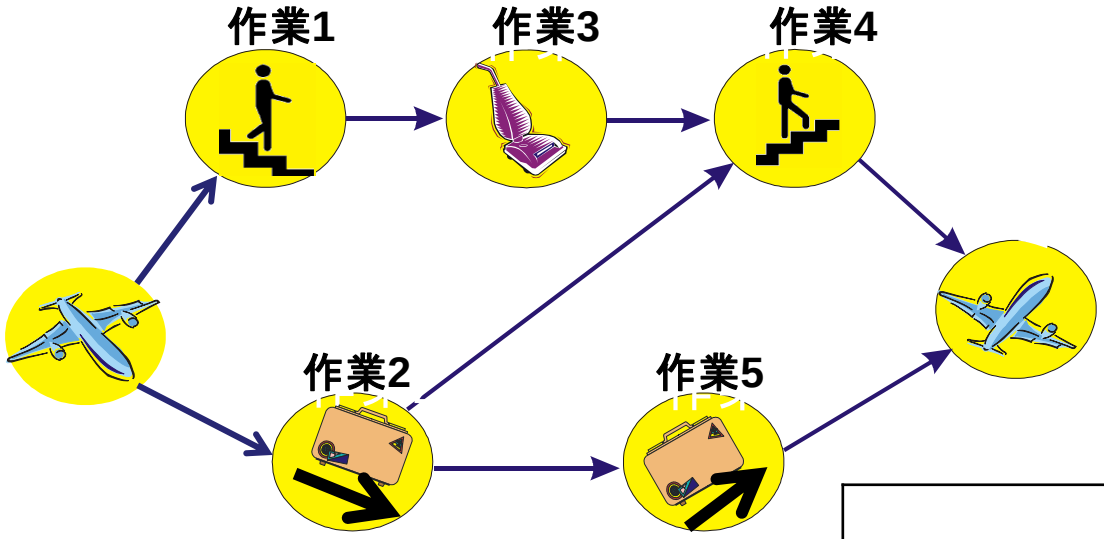
最適解

航空機離陸問題2

資源制約

目的：完了時刻最小化

航空機離陸問題1に作業員資源制約を追加



	作業時間
作業1（乗客降ろし）	13
作業2（荷物降ろし）	25
作業3（機内清掃）	15
作業4（乗客搭乗）	27
作業5（荷物積み込み）	22

Pythonインターフェイスによる記述（1）
（航空機離陸問題2：Example2.py）

```
from optseq import *    #optseq.pyモジュールを読み込む
m1=Model()              #m1と言うモデルオブジェクトを生成
#作業時間を辞書で表す
duration ={1:13, 2:25, 3:15, 4:27, 5:22 }

#モデルに再生可能資源追加
res=m1.addResource('worker',capacity=1)
```

モデル資源追加メソッドaddResource (資源名, 使用可能資源量})

```
act={} #作業を保管するための辞書を準備
mode={} #作業モードを保管するための辞書を準備
```

Pythonインターフェイスによる記述（2）
（航空機離陸問題2：Example2.py）

```
for i in duration:
    act[i]=m1.addActivity('Act[{0}]'.format(i))
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])

    #作業モードへの資源使用量追加
    mode[i].addResource(res,requirement=1)
    act[i].addModes(mode[i])   #作業へのモード追加
```

モード資源追加メソッドaddResource (資源, 使用資源量})

Pythonインターフェイスによる記述（3）
（航空機離陸問題2：Example2.py）

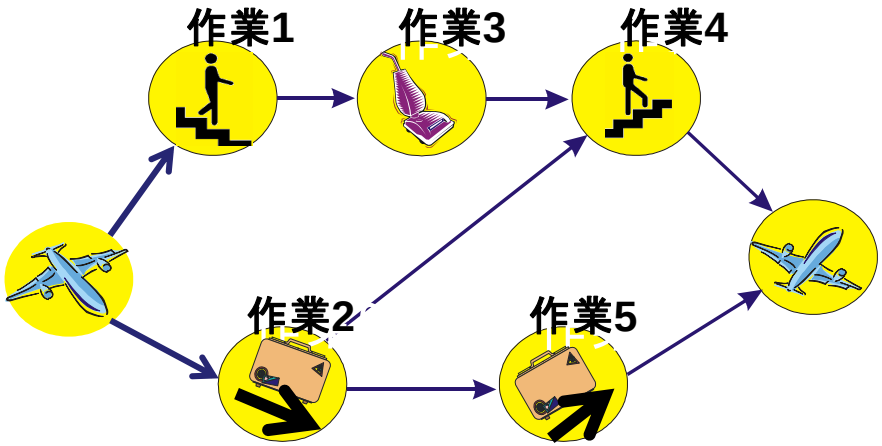
#モデルへの時間制約の追加

m1.addTemporal(act[1],act[3])

m1.addTemporal(act[2],act[4])

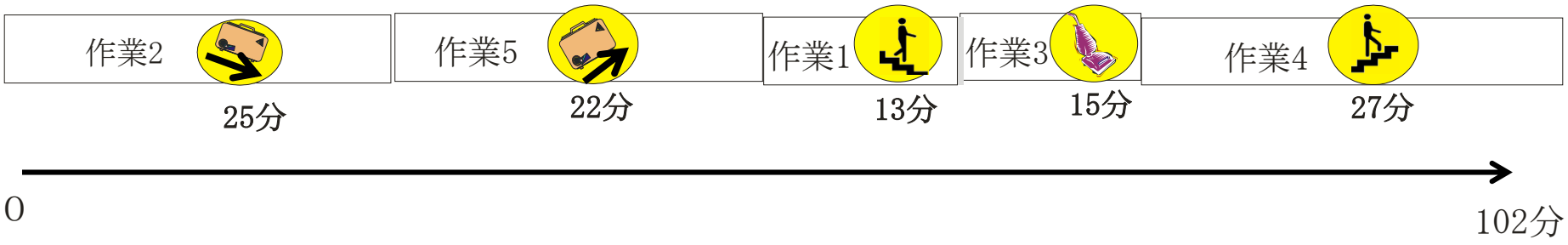
m1.addTemporal(act[2],act[5])

m1.addTemporal(act[3],act[4])



Pythonインターフェイスによる記述(4)
(航空機離陸問題2:Example2.py)

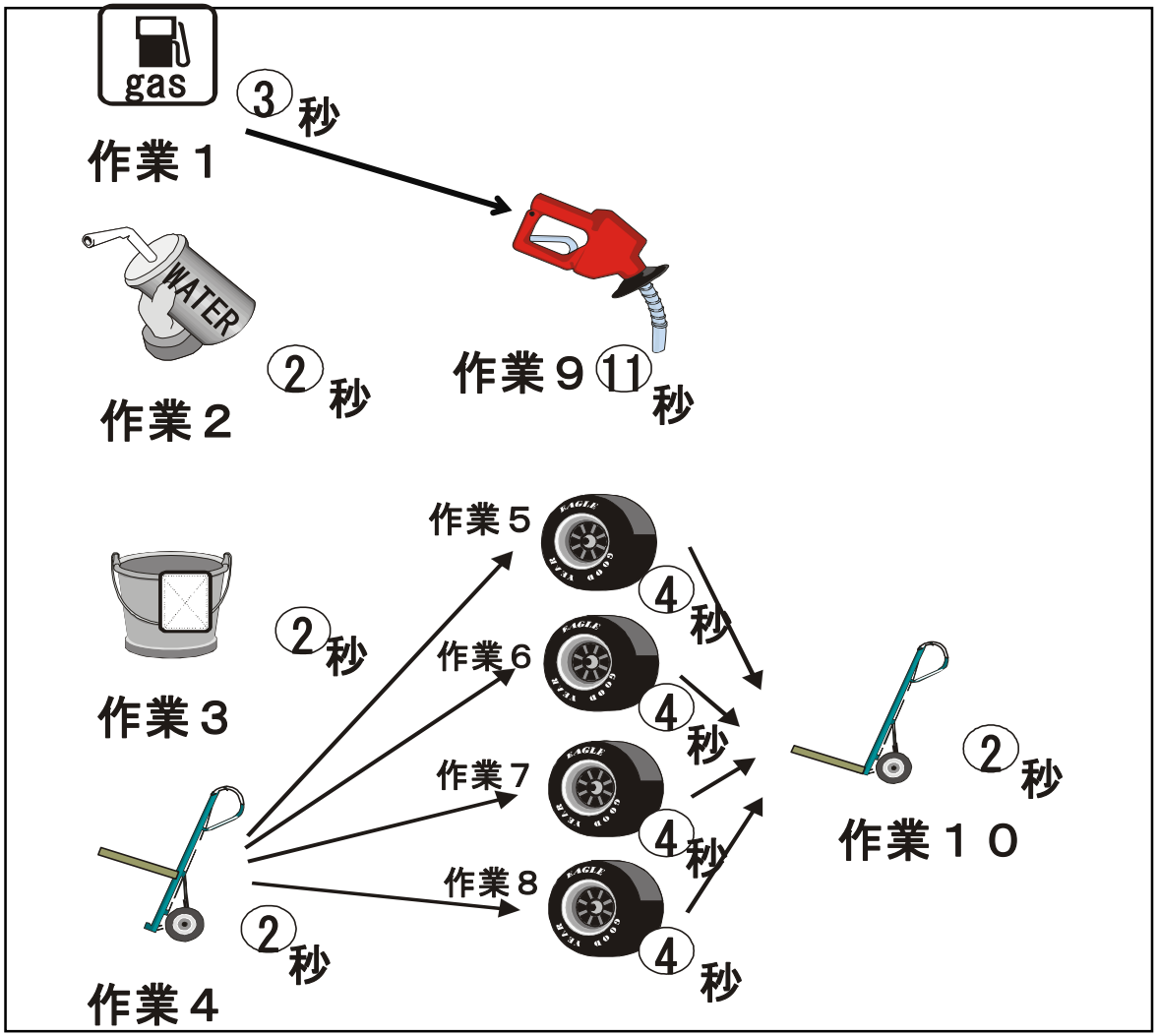
```
m1.Params.TimeLimit=1          #求解時間設定

m1.Params.Makespan=True        #最大完了時刻最小化の場合True,
                                 納期遅れ最小化などその他の場合
                                 False(規定値)を設定する.

m1.optimize()                  #モデルを求解する
```



例題Part 2

並列ショップスケジューリング問題1
並列ショップスケジューリング問題2
複数資源，作業モード

並列ショップスケジューリング1



３人のピットクルー
（**資源制約**）

Pythonインターフェイスによる記述（1）
（並列ショップスケジューリング1：Example3.py）

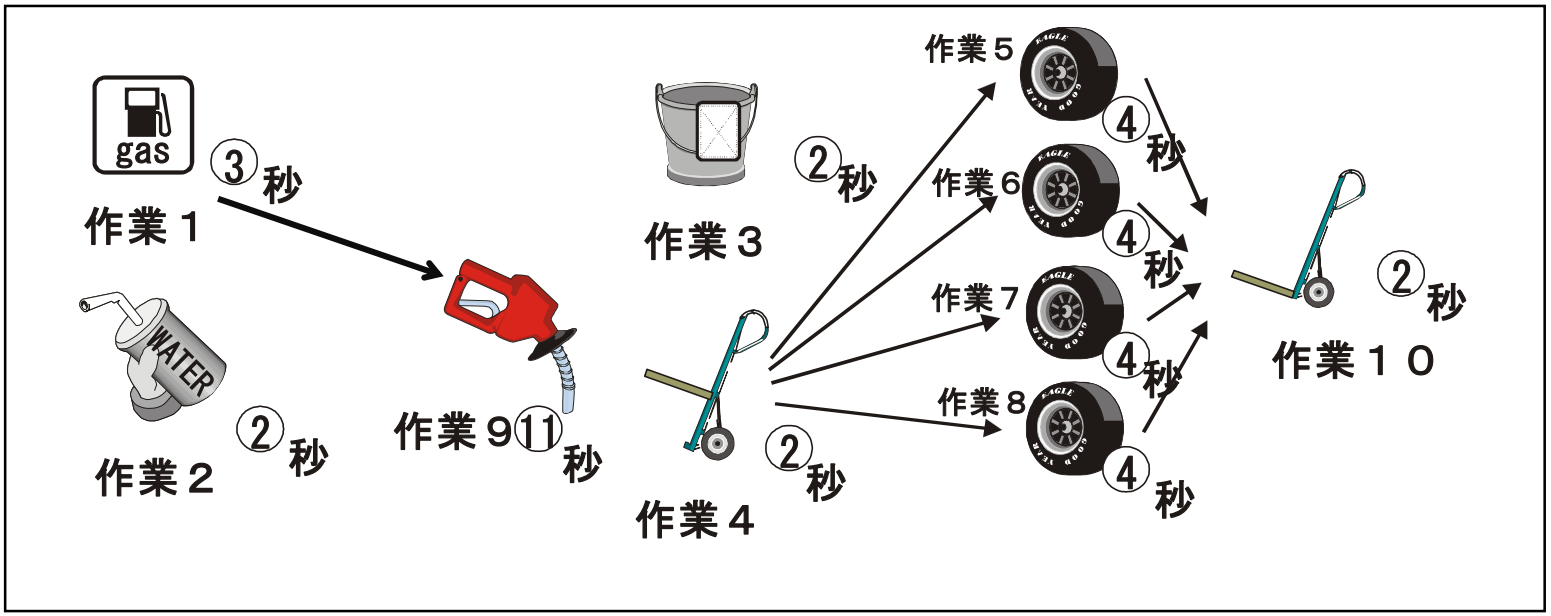
```
m1=Model()

#各作業の作業時間データ
duration ={1:3, 2:2, 3:2, 4:2, 5:4, 6:4, 7:4, 8:4, 9:11, 10:2 }

#使用可能資源（3人のピットクルー）を定義
res=m1.addResource('worker',capacity={(0,"inf"):3})

act={}
mode={}
```

３人のピットクルー
（資源制約）

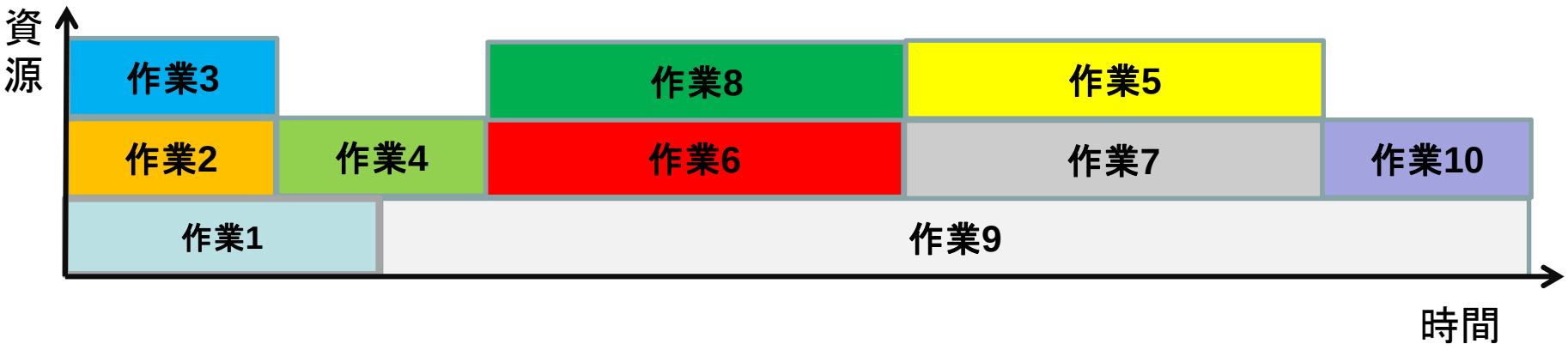
Pythonインターフェイスによる記述（2）
（並列ショップスケジューリング1：Example3.py ）

```
#作業作成＋作業にモード追加
for i in duration:
    act[i]=m1.addActivity("Act[{0}]".format(i))
   mode[i]=Mode("Mode[{0}]".format(i),duration[i])
   mode[i].addResource(res,{(0,"inf"):1})
   act[i].addModes(mode[i])

#モデルに時間制約を追加
m1.addTemporal(act[1],act[9])
for i in range(5,9):
   m1.addTemporal(act[4],act[i])
   m1.addTemporal(act[i],act[10])
```

Pythonインターフェイスによる記述(3)
(並列ジョブスケジューリング1:Example3.py)

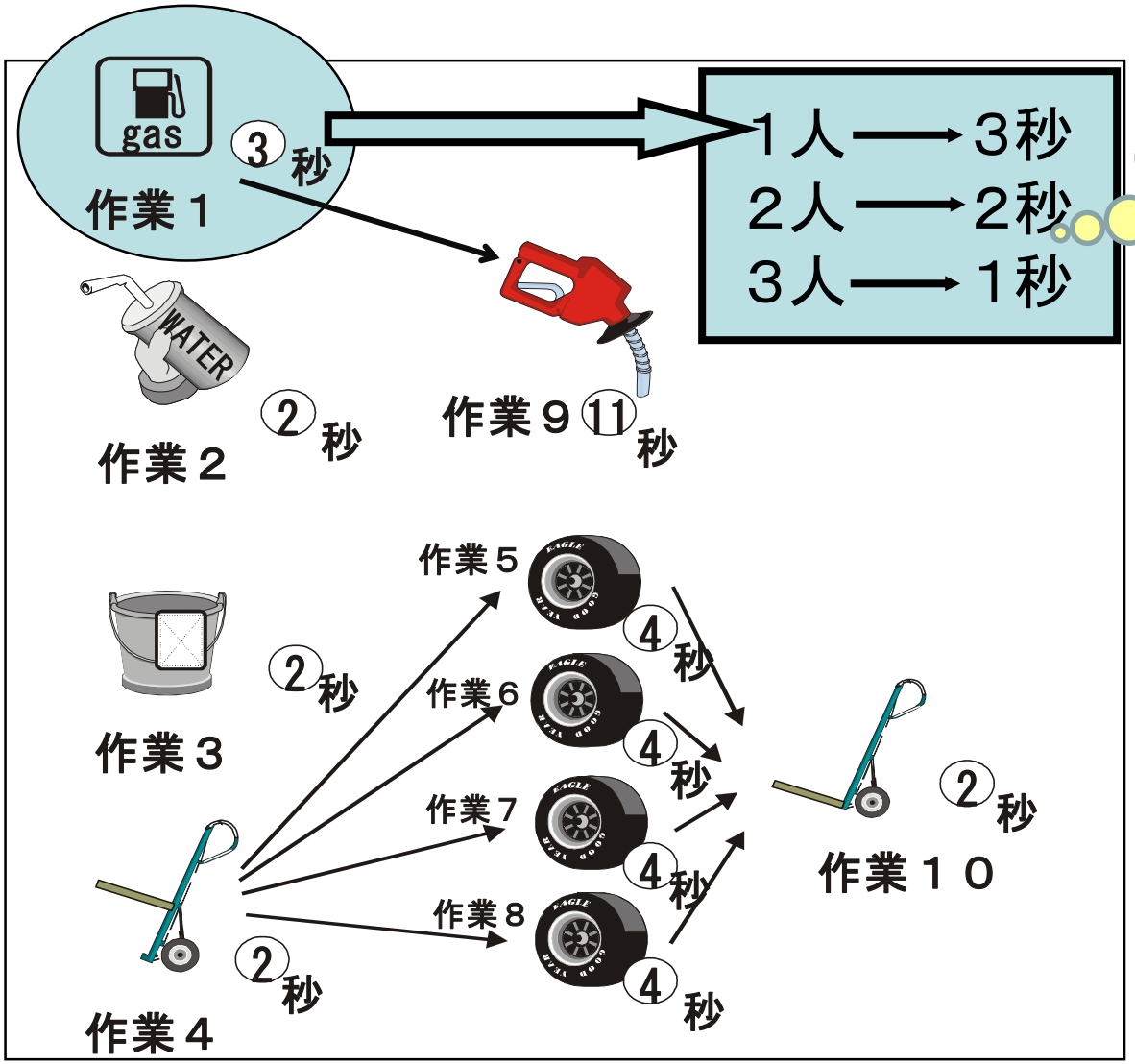


結果



並列ショップスケジューリング2

-- 複数モード



作業1は作業する人数（資源量）によって作業時間が異なる

作業1を3つのモードで表す

３人のピットクルー（資源制約）

作業の複数モード

どのような場合に使用するか？

作業の処理の仕方が複数ある場合

↓

モードごとに，作業時間，資源使用量などを設定

↓

作業に複数個のモード追加

↓

作業はいずれかのモード1つのモードを選択することによって処理される

使用例：缶ジュースを購入する作業

作業時間，使用資源が異なる場合：

- コンビニモード：作業時間20 分，お金資源120 円，人資源1 人
- 自販機モード：作業時間5 分，お金資源120 円，人資源1 人
- スーパーモード：作業時間40 分，お金資源100 円，人資源1 人，車資源1 台

作業を行う人数（や処理を行う機械の台数）によって作業時間が短縮される場合：

モード1：作業時間3 秒，人資源1 人
モード2：作業時間2 秒，人資源2 人
モード3：作業時間1 秒，人資源3 人

Pythonインターフェイスによる記述（1）
（並列ショップスケジューリング2：Example4.py）

```
m1=Model()

#各作業の作業時間データ
duration ={1:3, 2:2, 3:2, 4:2, 5:4, 6:4, 7:4, 8:4, 9:11, 10:2 }

#使用可能資源（3人のピットクルー）を定義
res=m1.addResource('worker',capacity=3)

act={}
mode={}
```

３人のピットクルー
（資源制約）

Pythonインターフェイスによる記述(2)-1
(並列ショップスケジューリング2:Example4.py)

```
for i in duration:  #作業作成＋作業にモード追加
    act[i]=m1.addActivity('Act[{0}]'.format(i))    #モデルに作業追加
    if i==1:
        mode[1,1]=Mode('Mode[1_1]',3)
        mode[1,1].addResource(res,1)
        mode[1,2]=Mode('Mode[1_2]',2)
        mode[1,2].addResource(res,2)
        mode[1,3]=Mode('Mode[1_3]',1)
        mode[1,3].addResource(res,3)

        act[i].addModes(mode[1,1],mode[1,2],mode[1,3])
    else:       …
```

モード名　作業時間

作業1に対しては3つのモードを定義

#作業員1人を使用する場合の作業時間は3秒であることを表すモード

#作業員2人を使用する場合の作業時間は2秒であることを表すモード

#作業員3人を使用する場合の作業時間は1秒であることを表すモード

複数個のモードはカンマ区切りで追加

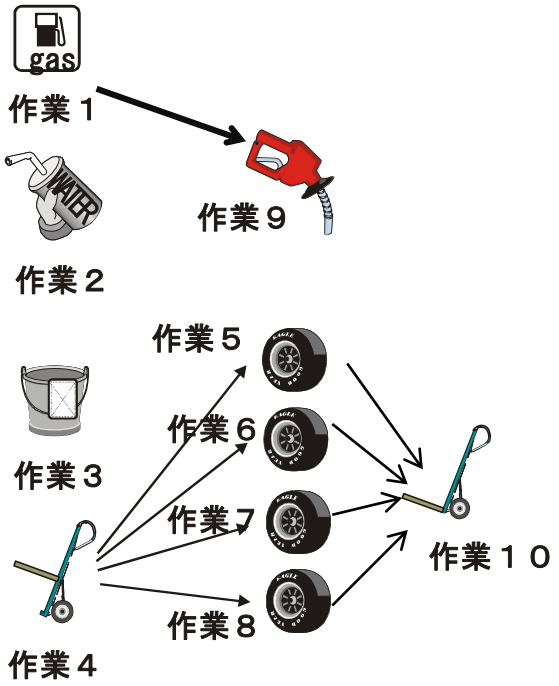
作業1に3つのモード追加

Pythonインターフェイスによる記述（2）-2
（並列ショップスケジューリング2：Example4.py ）

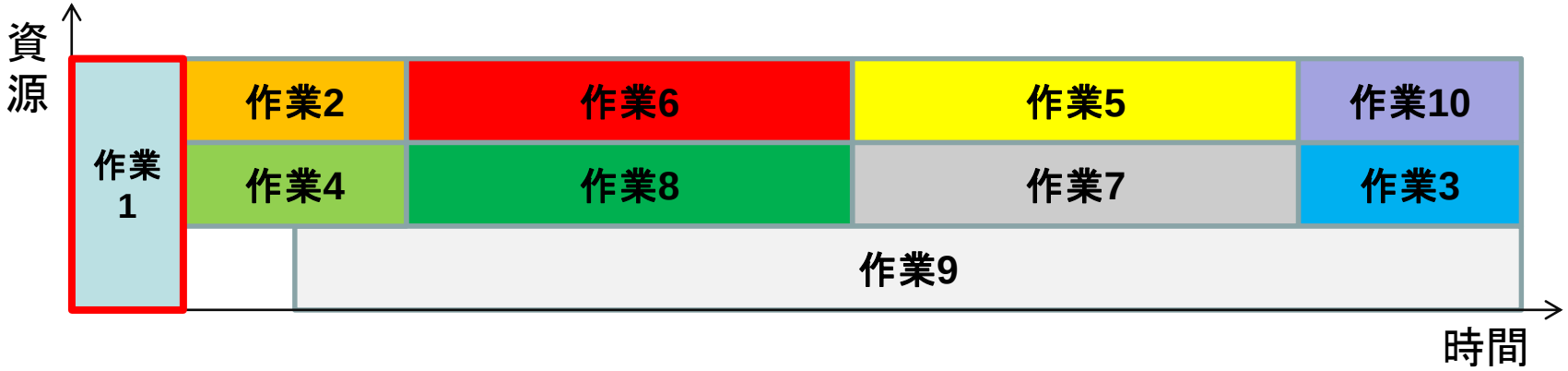
```
else: #作業1以外の作業にモードと資源追加
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    mode[i].addResource(res,1)   #作業員1人を使用
    act[i].addModes(mode[i])
```

Pythonインターフェイスによる記述（3）
（並列ジョブスケジューリング2：Example4.py ）

```
#モデルに時間制約を追加
m1.addTemporal(act[1],act[9])
for i in range(5,9):
    m1.addTemporal(act[4],act[i])
    m1.addTemporal(act[i],act[10])
```



作業1を3人で同時処理：



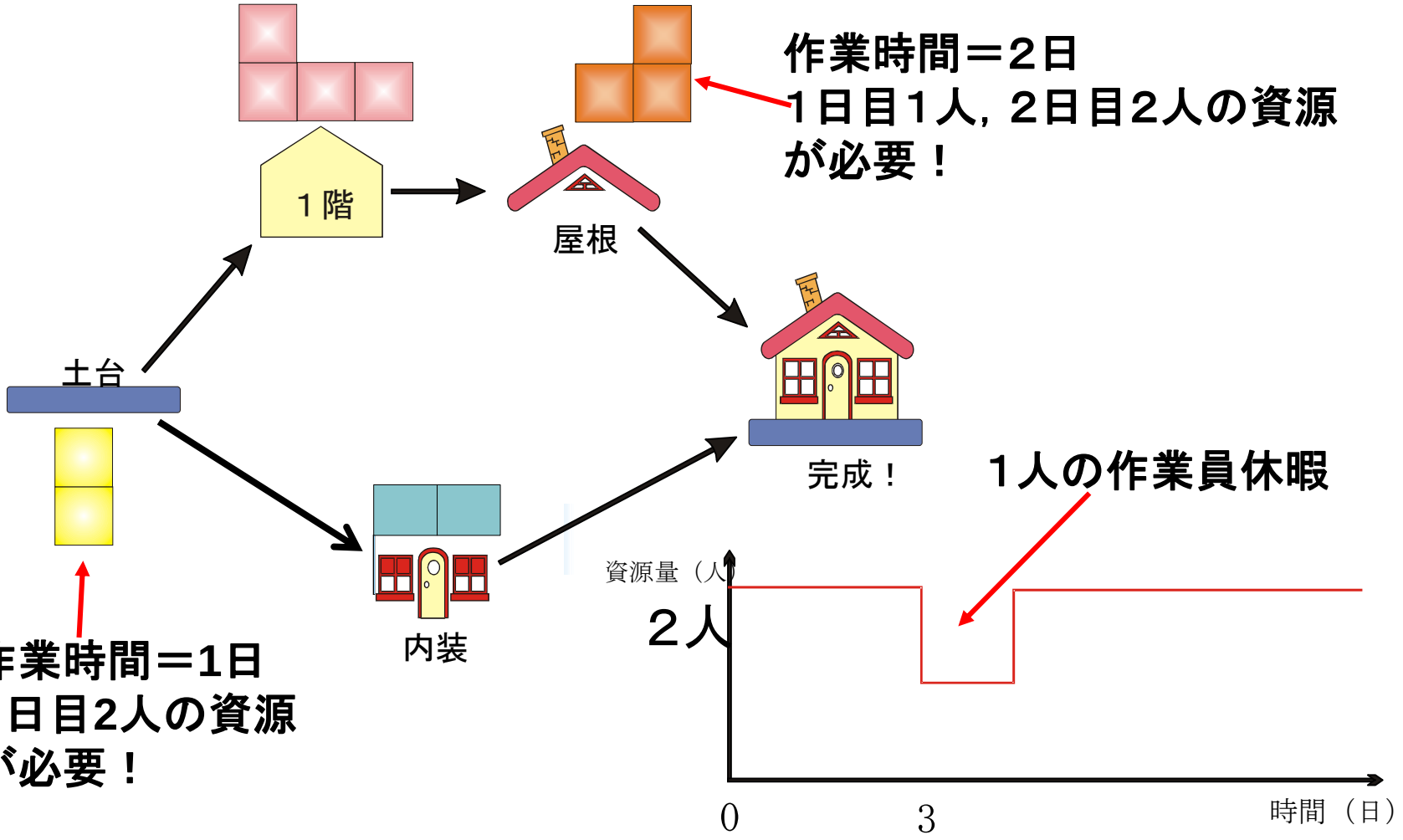
例題Part 3

お家を早く造ろう！

一般化資源制約付き問題

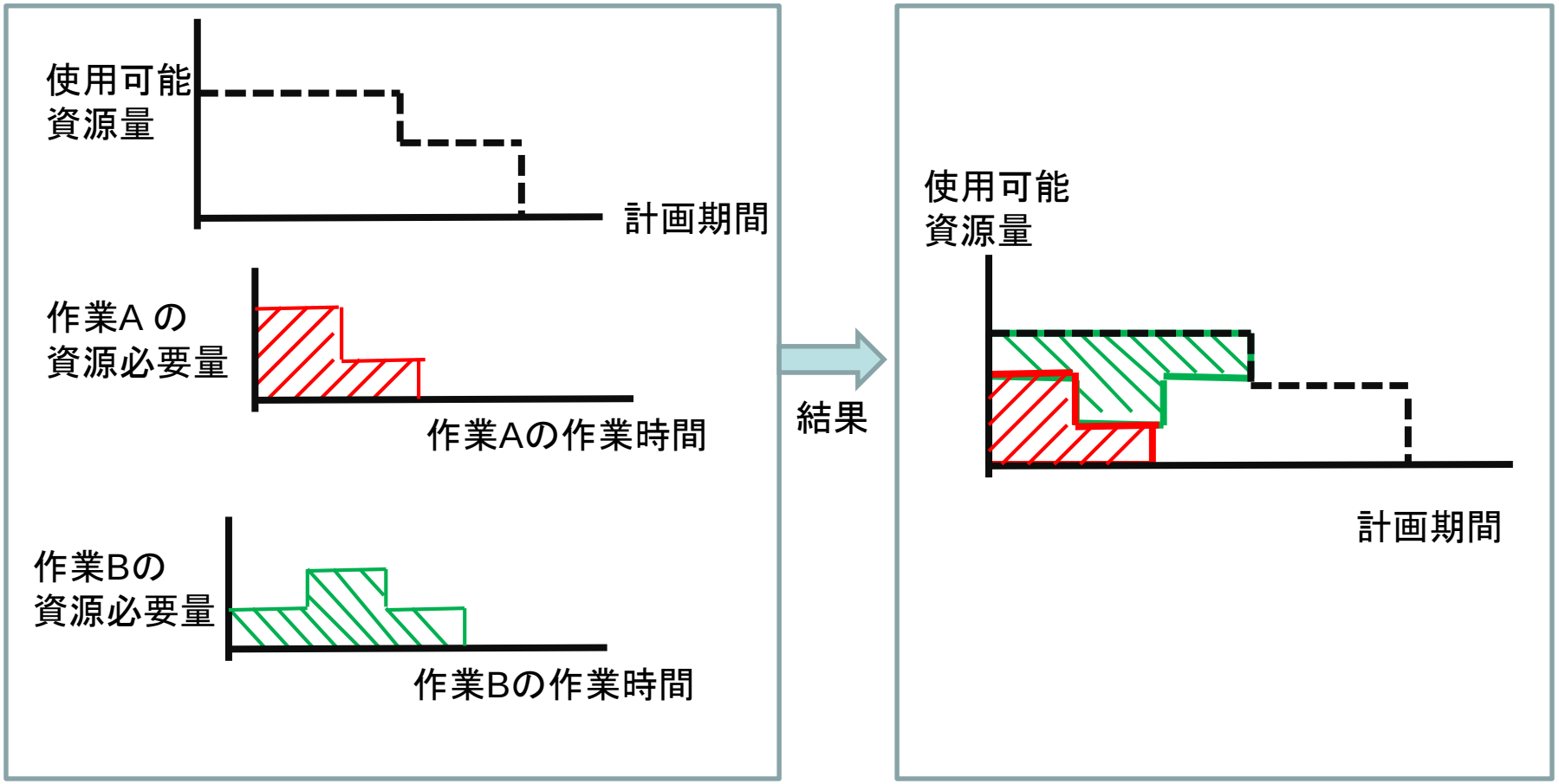
お家を早く造ろう！

一般化資源制約



一般化資源制約

- 任意の形の資源制約を追加可能

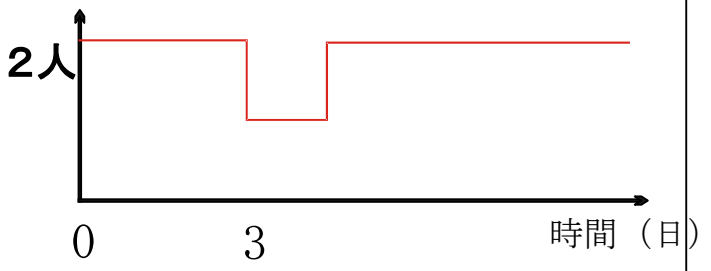


Pythonインターフェイスによる記述（1）
（お家を早く造ろう：Example5.py）

```
m1=Model()
duration ={1:1,2:3,3:2,4:2}  #各作業の作業時間
#モデルのメソッドaddResourceで使用可能資源を追加
res=m1.addResource('worker')
#資源に作業員資源使用可能量を追加
res.addCapacity(0,2,2)
res.addCapacity(2,3,1)
res.addCapacity(3,'inf',2)
act={}
mode={}
```

資源使用可能量が複数の区分に分かれている場合，
資源量追加メソッドaddCapacityを用いると便利

作業開始時刻　作業終了時刻　必要資源量

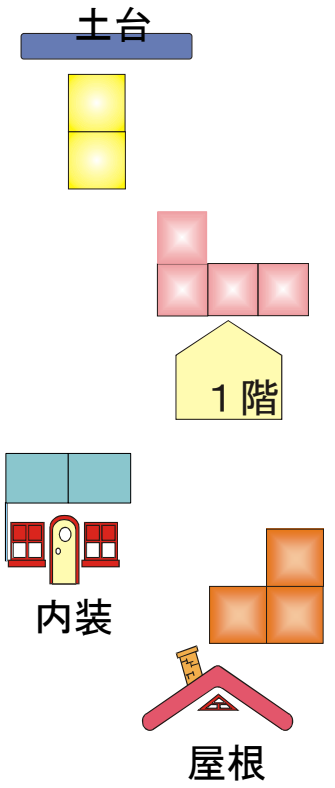


Pythonインターフェイスによる記述（2）
（お家を早く造ろう：Example5.py）

```
act, mode, req={}, {}, {}
```

作業開始時刻　作業終了時刻　必要資源量

```
req[1]={(0,1):2 }              #土台作りの作業員必要量
req[2]={(0,1):2 ,(1,3):1}      #一階作りの作業員必要量
req[3]={(0,2):1 }              #内装の作業員必要量
req[4]={(0,1):1,(1,2):2 }      #屋根作りの作業員必要量
for i in duration:
    act[i]=m1.addActivity('Act[{0}]'.format(i))
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    #作業モードに資源を追加
    mode[i].addResource(res, requirement= req[i])
    act[i].addModes(mode[i])
```



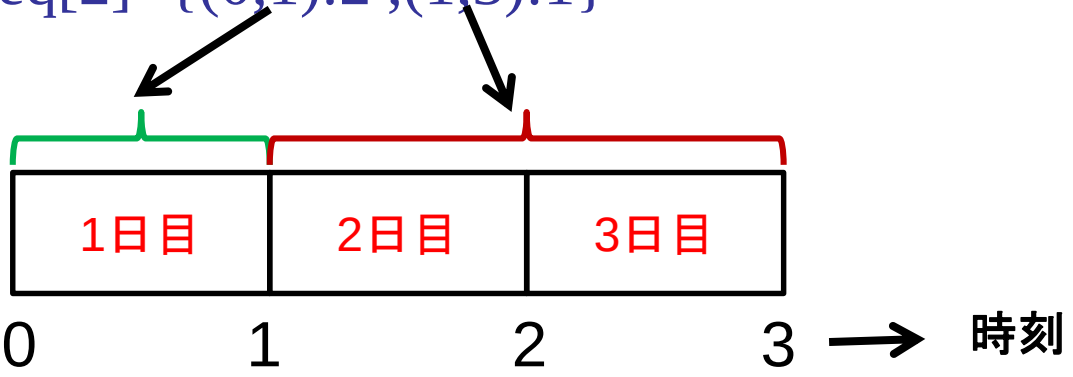
資源使用量が複数の区分に分かれている場合，
資源使用量（requirement）は，辞書形式で入力

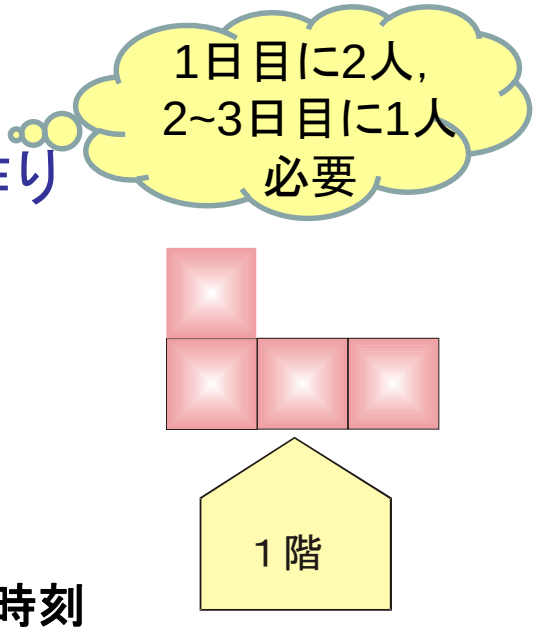
時刻表現の注意点

- 時刻表現の使用場所
  - 資源の使用開始時刻，使用終了時刻
  - 中断可能作業の開始時刻，終了時刻（後述）
- 時刻表現の特徴
  - 0から開始
  - 開始時刻～終了時刻未満を表す

例： お家を早く造ろう(Example5.py)の1階作り

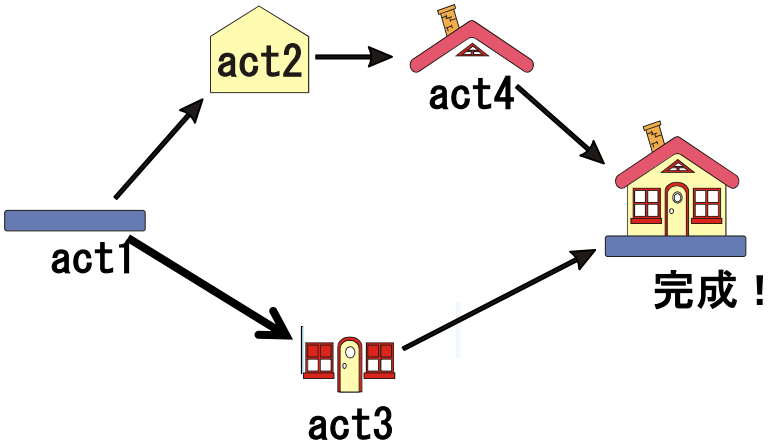
req[2]={(0,1):2 ,(1,3):1}



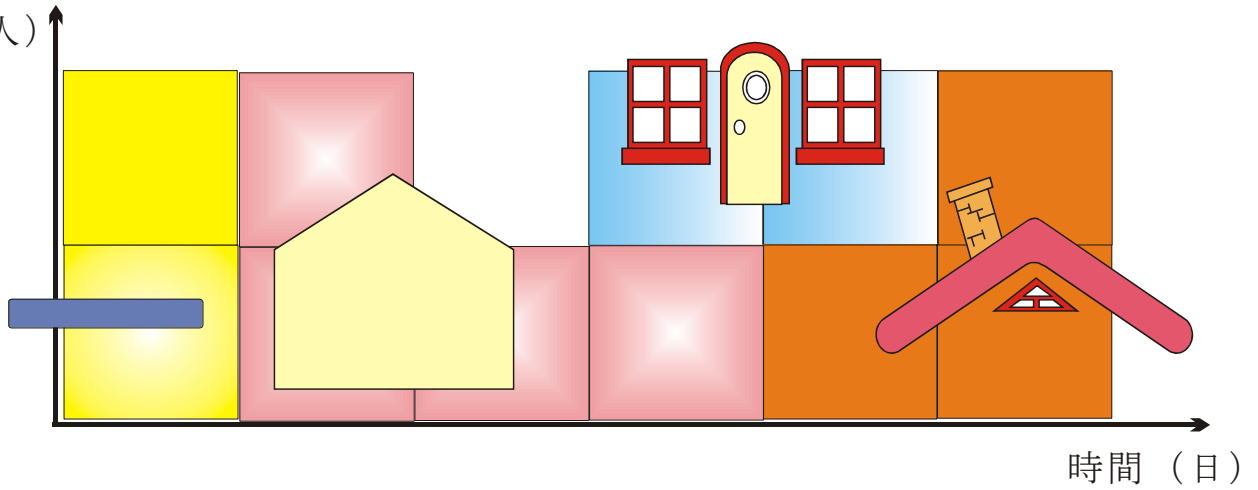


Pythonインターフェイスによる記述（3）
（お家を早く造ろう：Example5.py）

```
#時間制約を追加
m1.addTemporal(act[1],act[2])
m1.addTemporal(act[1],act[3])
m1.addTemporal(act[2],act[4])
```



結果



例題Part 4

航空機離陸問題3

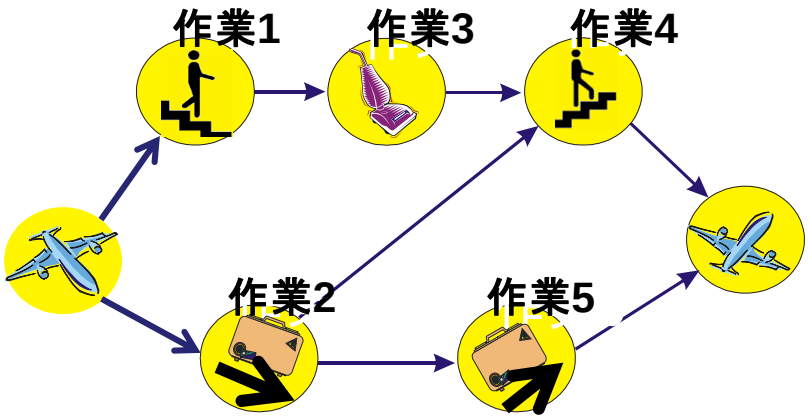
再生不能資源

航空機離陸問題4

同時開始

航空機離陸問題3
-再生不能資源

目的:完了時刻最小化



航空機離陸問題1に再生不能資源制約を追加

使用可能なお金は全部で4万円
作業時間は費用をかけることによって短縮できる
再生不能資源で表現

	作業時間	特急作業時間	特急費用
作業1(乗客降ろし)	13	10	1万円
作業2(荷物降ろし)	25	20	1万円
作業3(機内清掃)	15	10	1万円
作業4(乗客搭乗)	27	25	1万円
作業5(荷物積み込み)	22	20	1万円

どの作業時間で作業を行うかは

複数モード

Pythonインターフェイスによる記述（1）
（航空機離陸問題3：Example7.py）

```
Jobs=[1,2,3,4,5]
#通常の作業時間データdurationAと再生不能資源を使用する場合
    の作業時間durationBを作成する.
durationA = {1:13, 2:25, 3:15, 4:27, 5:22 }
durationB = {1:10, 2:20, 3:10, 4:25, 5:20 }

m1=Model()        #m1と言うモデルオブジェクトを生成.
act={}            #作業を保管するための辞書を準備
mode={}           #作業モードを保管するための辞書を準備
```

Pythonインターフェイスによる記述（2）
（航空機離陸問題3：Example7.py）

```
for i in Jobs:
    mode[i,1]=Mode('Mode[{0}][1]'.format(i),durationA[i])
    mode[i,2]=Mode('Mode[{0}][2]'.format(i),durationB[i])
    act[i]=m1.addActivity('Act[{0}] '.format(i))
    act[i].addModes(mode[i,1],mode[i,2])
```

作業1の通常モード
と特急モードを定義

複数モードを追加する場合，
カンマ区切りでモードを追加

Pythonインターフェイスによる記述（3）
（航空機離陸問題3：Example7.py）

```
#モデルに再生不能資源を追加する
res=m1.addResource('money',rhs=4,direction='<=')
```

モデルへの資源追加メソッドaddResource (資源名，右辺(rhs)，制約向き(direction))

```
#再生不能資源制約左辺追加
for i in Jobs:
    res.addTerms(1,act[i],mode[i,2])
```

作業act[i]をmode[i][2]で行う場合，再生不能資源を1単位使用することを表す

再生不能資源左辺追加メソッドaddTerms(再生不能資源使用量,作業,再生不能資源を使うモード)

再生不能資源制約

- 任意の不等式で記述可能(≦, ≧, ＝)
  - お金や原材料などの資源の上限を表す
    - 使用した原材料の量≦原材料の上限
  - 多目的のスケジューリング問題

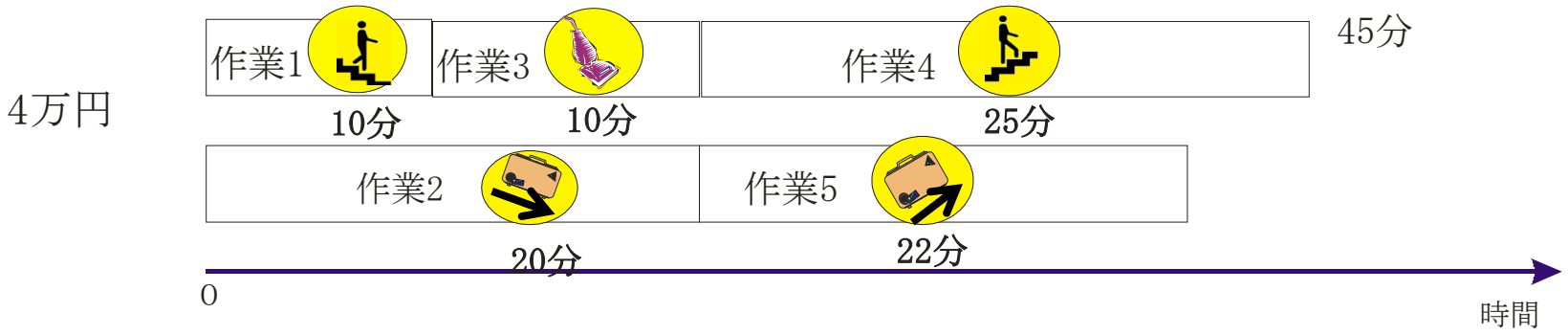
    例えば，生産量下限を考慮しながら原材料の使用量上限も考慮したい.
    - 原材料の使用量≦原材料上限

      生産量≧生産量下限
- 任意の正数重みを付けることが可能
  - 各再生不能資源制約に重みを付けることで，制約の重要度を表現することも可能

Pythonインターフェイスによる記述(4)
(航空機離陸問題3:Example7.py)

```
#モデルに時間制約を追加する
m1.addTemporal(act[1],act[3])
m1.addTemporal(act[2],act[4])
m1.addTemporal(act[2],act[5])
m1.addTemporal(act[3],act[4])
```

再生不能資源を4単位使用する場合:

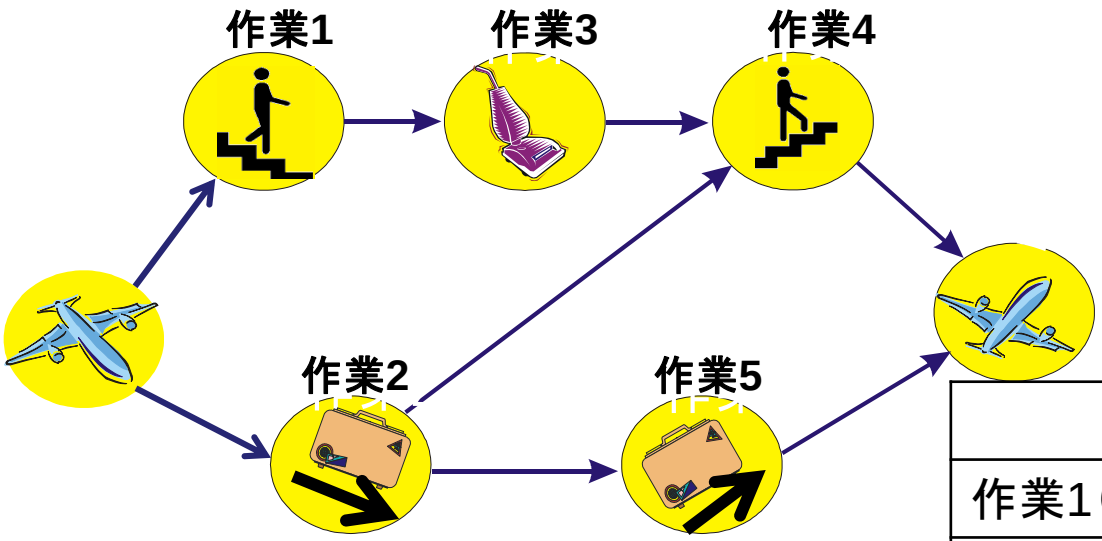


航空機離陸問題4

同時開始

目的:完了時刻最小化

航空機離陸問題1に同時開始制約を追加



作業3と作業5は同時開始する必要あると仮定

矢印は作業間の先後順位を表す.
eg:作業1が終了しないと作業3は開始できない

	作業時間
作業1(乗客降ろし)	13
作業2(荷物降ろし)	25
作業3(機内清掃)	15
作業4(乗客搭乗)	27
作業5(荷物積み込み)	22

Pythonインターフェイスによる記述（1）
（航空機離陸問題4：Example8.py）

```
m1=Model()        #m1と言うモデルオブジェクトを生成.
duration = {1:13, 2:25, 3:15, 4:27, 5:22 }#各作業の作業時間
act={}                #作業を保管するための辞書を準備
mode={}           #作業モードを保管するための辞書を準備

for i in durationA:
    #作業をモデルm1に追加
    act[i]=m1.addActivity("Act[{0}]".format(i))
    mode[i]=Mode("Mode[{0}]".format(i),durationA[i])
    act[i].addModes(mode[i]) #作業にモード追加
```

Pythonインターフェイスによる記述（2）
（航空機離陸問題4：Example8.py）

```
#時間制約追加
m1.addTemporal(act[1],act[3])
m1.addTemporal(act[2],act[4])
m1.addTemporal(act[2],act[5])
m1.addTemporal(act[3],act[4])

#作業3と作業5を同時開始する制約追加
m1.addTemporal(act[3],act[5],"SS",0)
m1.addTemporal(act[5],act[3],"SS",0)
```

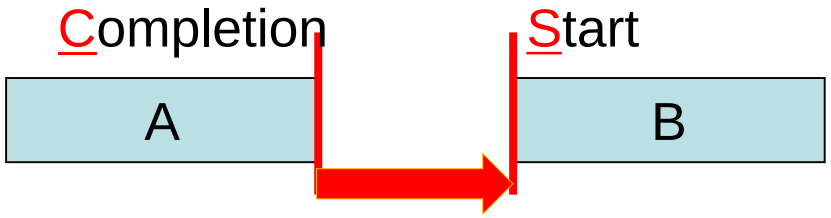
作業3が先行作業の場合と作業5が先行作業の場合の2種類制約を同時記入する必要がある

時間制約追加メソッドaddTemporal(
先行作業, 後続作業 , ‘制約タイプ’ ,時間ずれ)

規定値：CS　　規定値：0

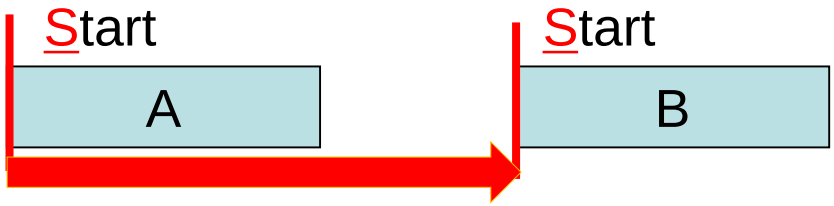
時間制約タイプ（type）

type CS



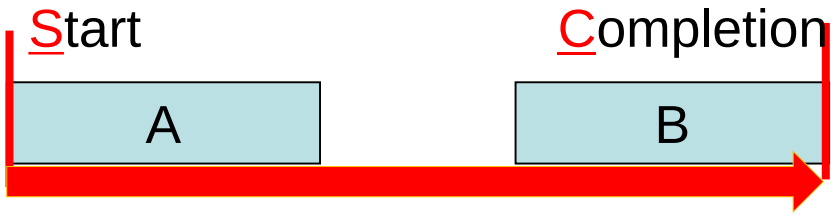
作業A終了後に作業B
が開始することを表す

type SS



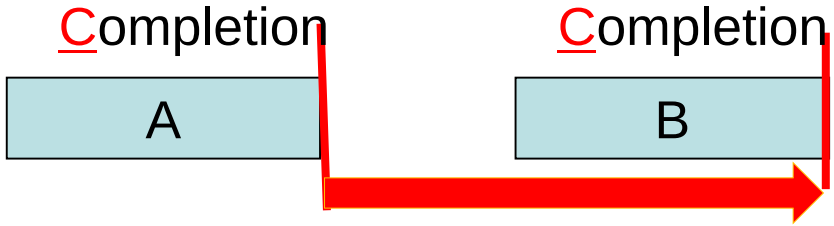
作業A開始後に作業B
が開始することを表す

type SC



作業A開始後に作業B
が終了することを表す

type CC

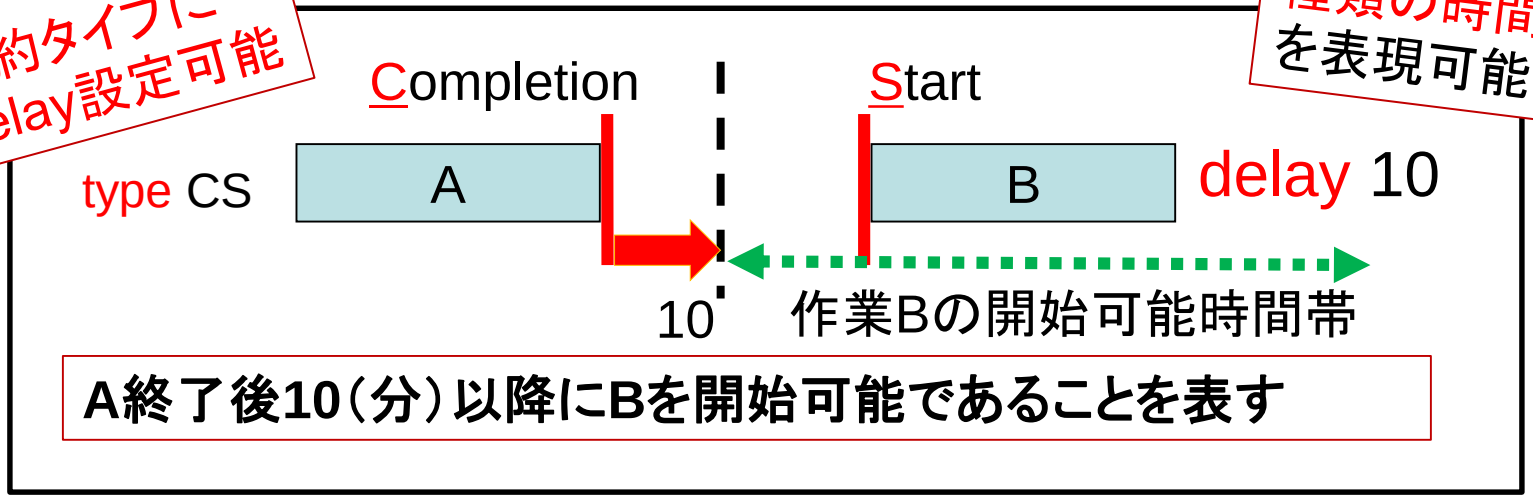


作業A終了後に作業B
が終了することを表す

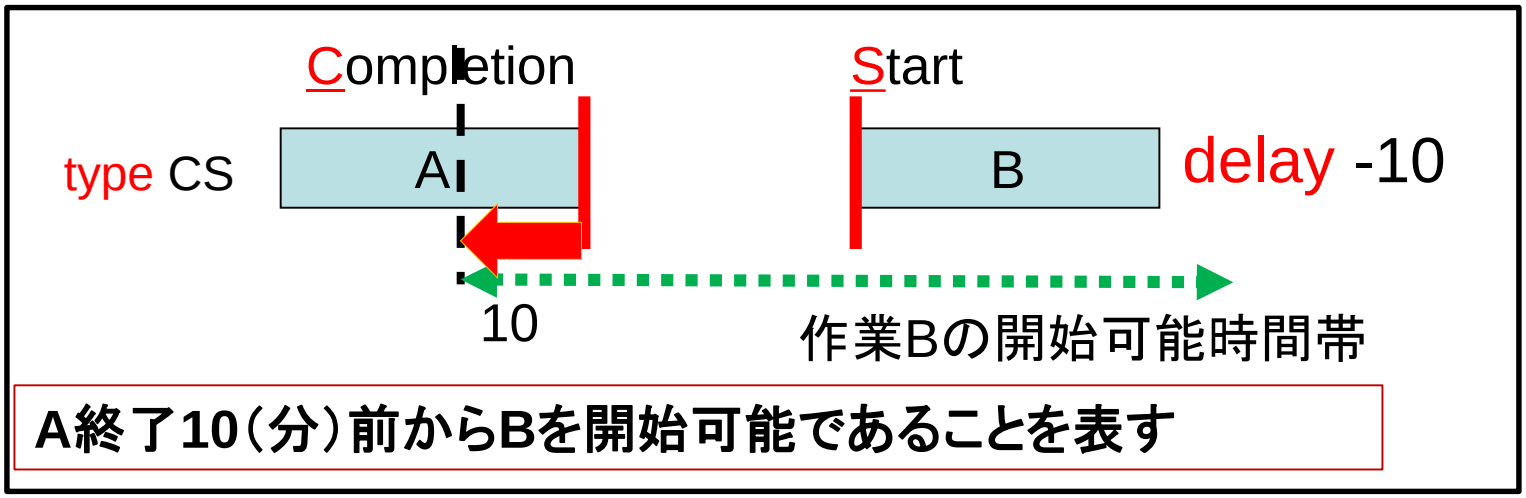
時間制約の時間ずれ（delay）

すべての時間
制約タイプに
delay設定可能

AとB間の様々な
種類の時間制約
を表現可能



A終了後10（分）以降にBを開始可能であることを表す



A終了10（分）前からBを開始可能であることを表す

時間制約の使用例(1)

- **航空機離陸問題1（Example8.py）を例に時間制約の設定方法を紹介**

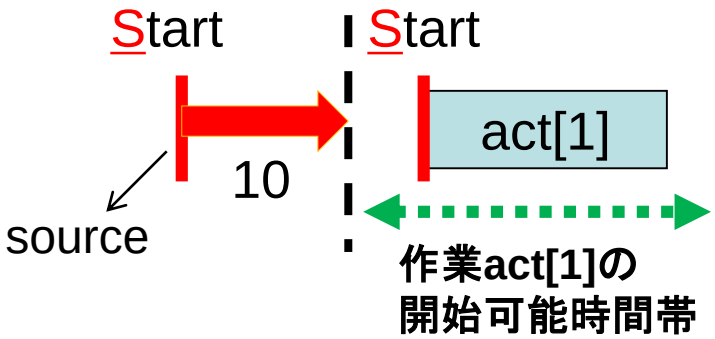
#作業1の開始時刻を計画期間開始時刻の10分後に固定する制約

m1.addTemporal('source',act[1],'SS',10)

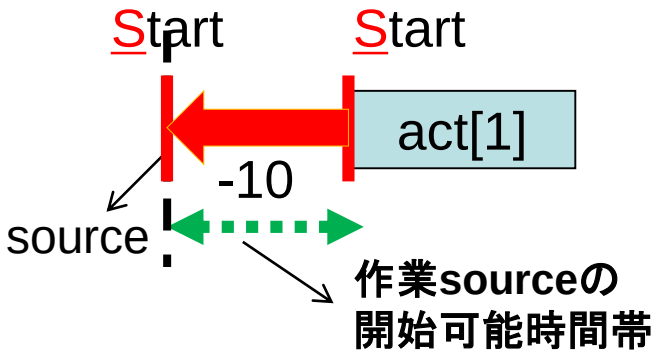
m1.addTemporal(act[1],'source','SS',-10)

sourceは始点を表すダミー作業を表す

('source',act[1],'SS',10)



(act[1],'source','SS',-10)



時間制約の使用例(2)

- **航空機離陸問題1**（Example8.py）**を例に時間制約の設定方法を紹介**

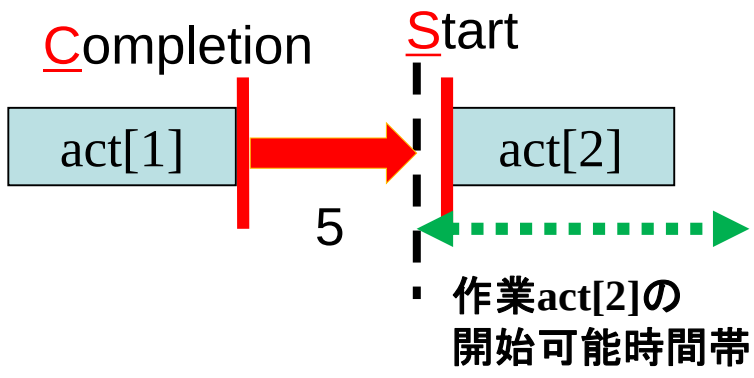
#作業2は作業1終了5分～10分の間に開始しなければならないことを表す制約

m1.addTemporal(act[1],act[2],'CS',5) ⇨ act[1]Completion+5≦act[2]Start

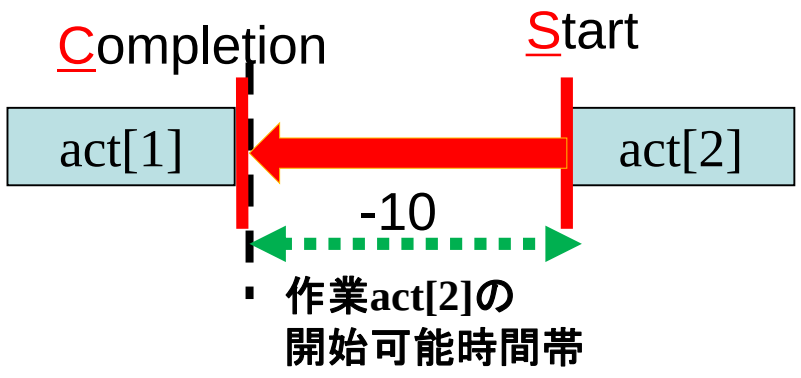
m1.addTemporal(act[2],act[1],'SC',-10) ⇨ act[2]Start-10 ≦act[1]Completion

act[1]Completion+5 ≦ act[2]Start ≦ act[1]Completion+10

(act[1],act[2],'CS',5)



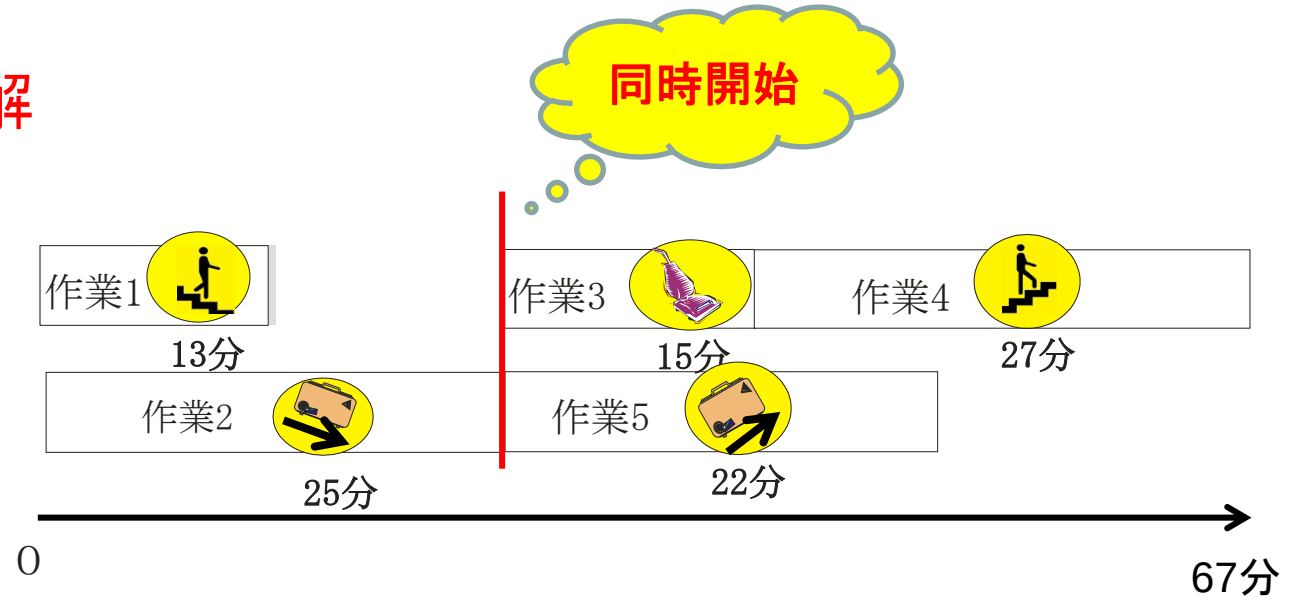
(act[2],act[1],'SC',-10)



Pythonインターフェイスによる記述(3)
(航空機離陸問題4:Example8.py)

```
m1.Params.TimeLimit=1          #求解時間設定
m1.Params.Makespan=True        #最大完了時刻最小化の場合True
m1.optimize()                  #モデルを求解する
```

最適解

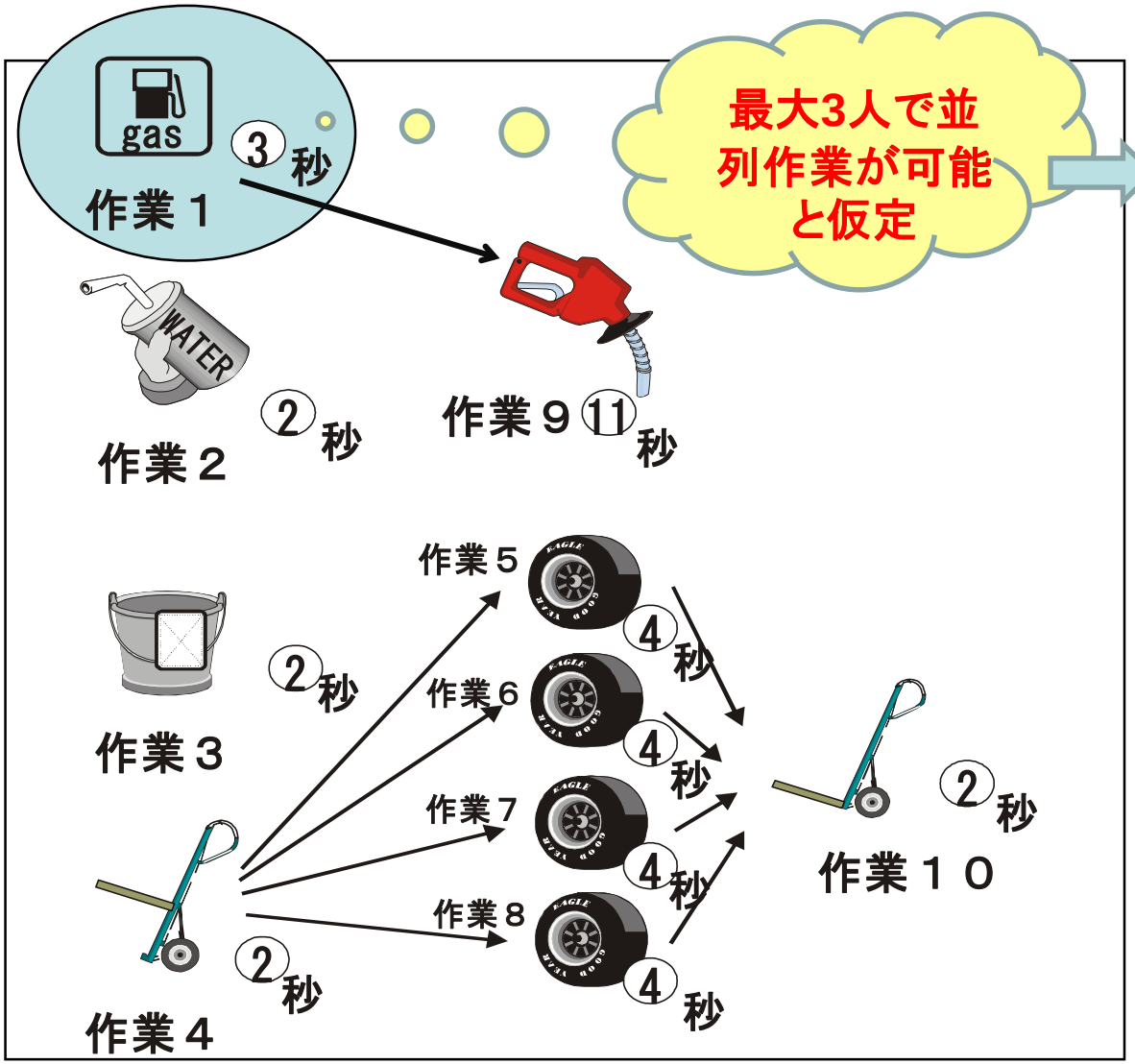


例題Part 5

並列ショップスケジューリング問題3
作業の並列実行

並列ショップスケジューリング3

-- 作業の並列実行



最大3人で並列作業が可能と仮定

作業1を並列処理可能と定義する

３人のピットクルー（資源制約）

作業の並列処理

作業1（作業時間は3秒）
最大3人（3単位）で並列作業可能

作業1を3つの小作業に分割可能と仮定

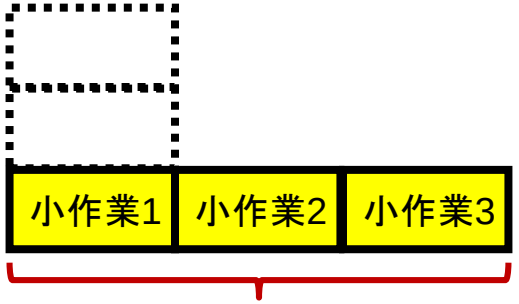
addParallel(1,1,3)

並列開始小作業開始番号

並列終了小作業終了番号

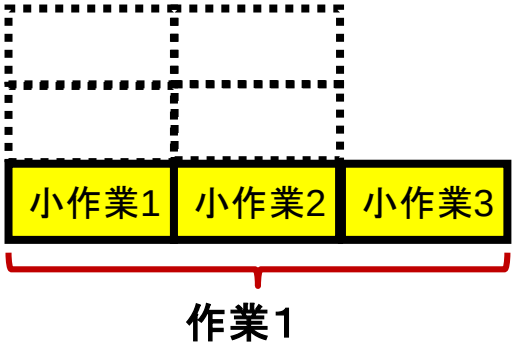
最大並列数

小作業1は最大3人で並列作業可能



addParallel(1,2,3)

小作業1と小作業2が最大3人で並列作業可能

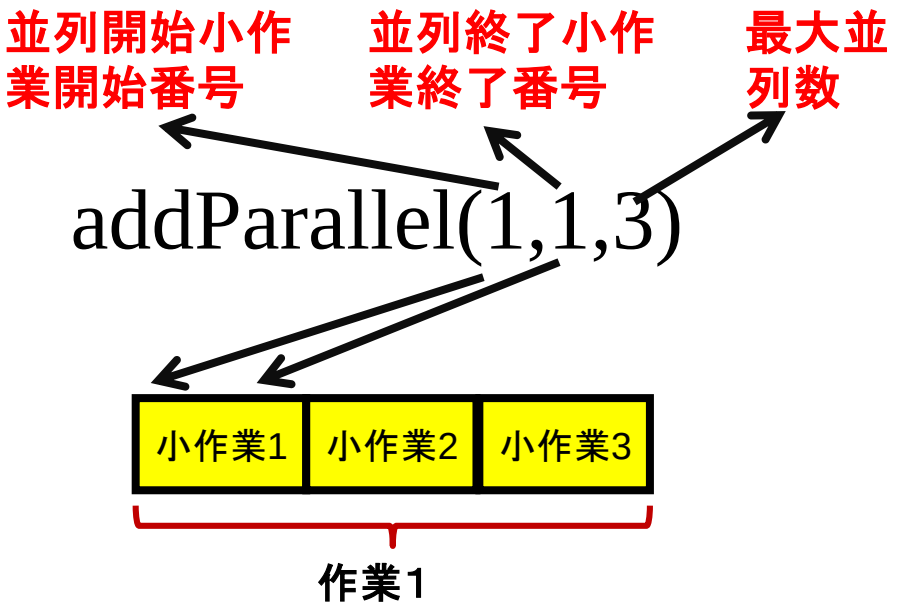


作業番号表現の注意点

- 作業番号表現の使用場所
  - 並列作業を表すときの小作業
- 作業番号表現の特徴
  - 1から開始

資源使用開始，終了などを表す**時刻は0から開始する**ので注意

例： 並列ジョブスケジューリング問題(Example10.py)の作業1の小作業が最大3人で並列作業可能



Pythonインターフェイスによる記述（1）
（並列ショップスケジューリング3：Example10.py ）

その他の条件，制約は並列ショップスケジューリング1のExample4.pyと同じ.

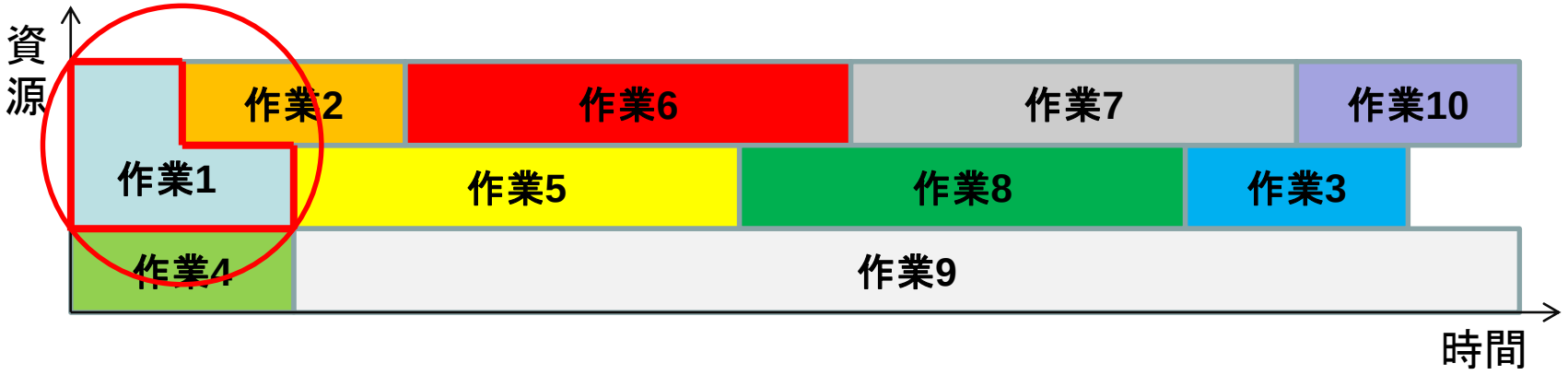
```
for i in duration:
    act[i]=m1.addActivity('Act[{0}]'.format(i))
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    mode[i].addResource(res,1)
    if i==1:
        mode[i].addParallel(1,1,3)
    act[i].addModes(mode[i])
    …
```

#作業1を最大3人で同時に行うことを作業の並列モードを追加することで表現する.

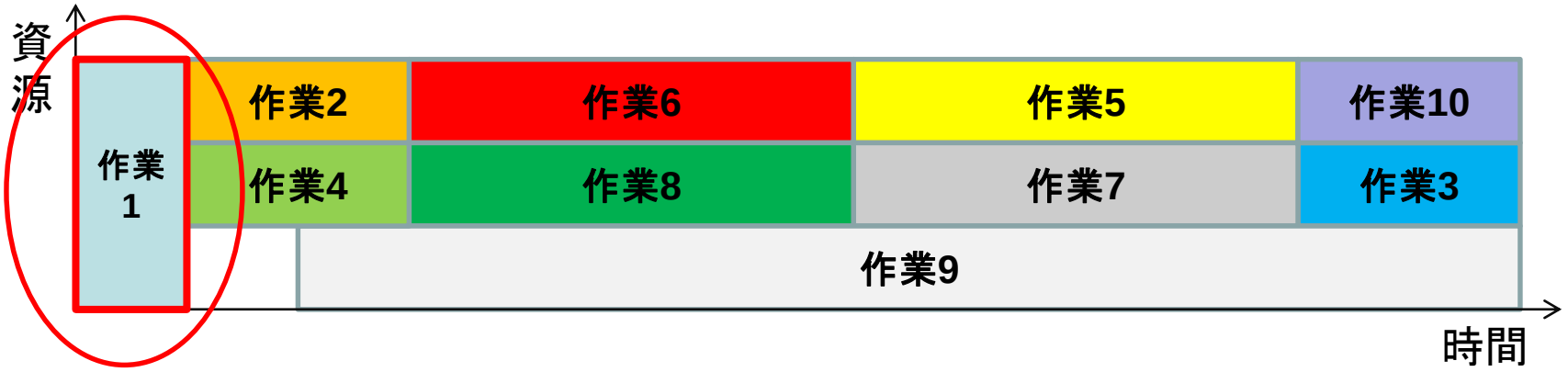
並列モード追加：.addParallel (
並列開始小作業番号,並列終了小作業番号, 最大並列数)

並列ショップスケジューリング結果

作業1を並列モードで処理：Example10.py

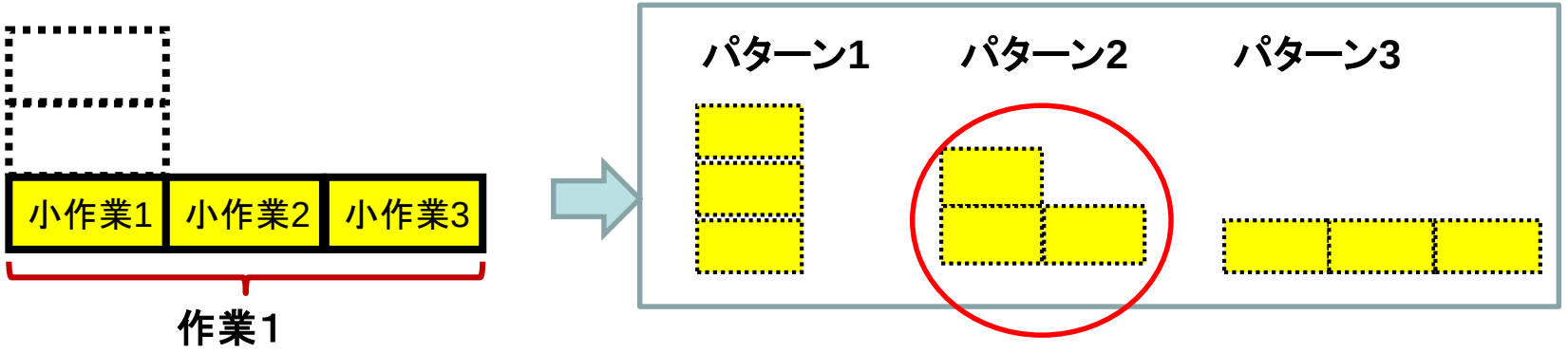


作業1を3人で同時処理：Example4.py

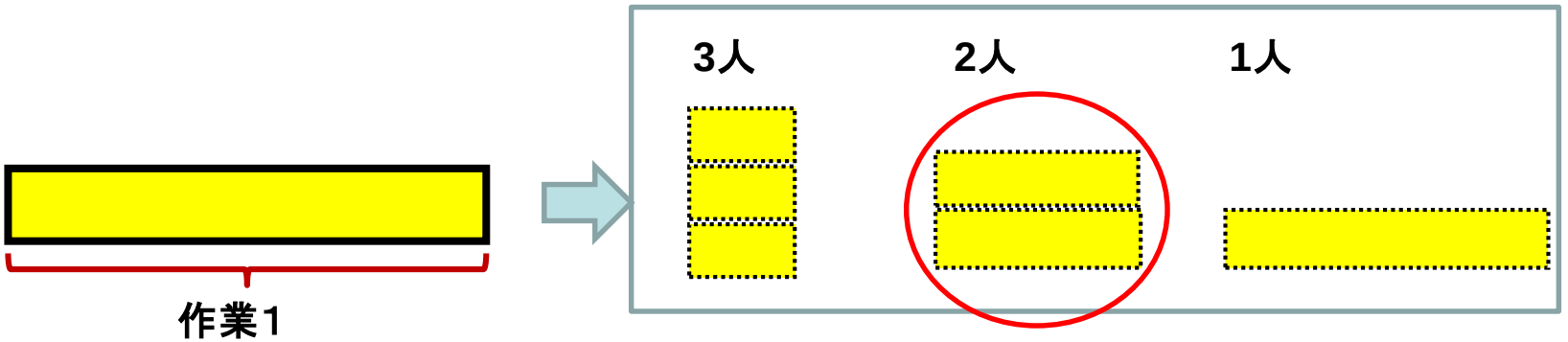


複数モードと並列の違い

- 作業1の小作業1が最大3人で並列処理可能と仮定する場合



- 作業1を複数モード（1人の場合3秒，2人の場合2秒，3人の場合3秒）で処理可能と仮定する場合



例題Part 6

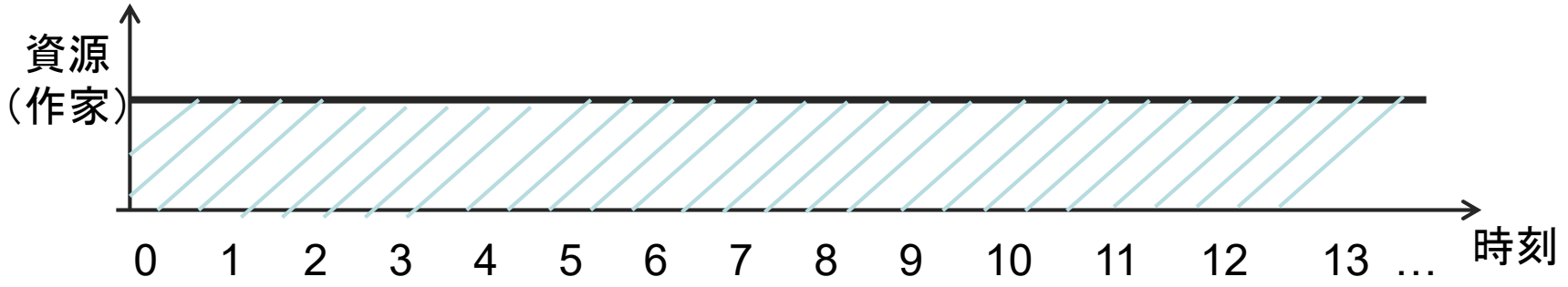
納期遅れを最小にしよう1
納期遅れを最小にしよう2
納期遅れを最小にしよう3

納期，一般化資源，作業の途中中断

納期遅れを最小にしよう1
--納期

- 1人の作家（休みなしに働けると仮定）
- 4社（A，B，C，D）から依頼された原稿を書く（作業）
- 納期遅れの和が最小になるように，スケジュールを組みたい．

会社名	A社	B社	C社	D社
作業時間	1日	2日	3日	4日
納期	5日後	9日後	6日後	4日後



Pythonインターフェイスによる記述（1）
（納期遅れ最小化問題1：Example6.py）

```
m1=Model()        #m1と言うモデルオブジェクトを生成

due={1:5,2:9,3:6,4:4}      #各社の納期
duration={1:1, 2:2, 3:3, 4:4 }    #各社の原稿の作業時間

act={}
mode={}
```

Pythonインターフェイスによる記述(2)
(納期遅れ最小化問題1:Example6.py)

```
#資源を追加する
res=m1.addResource("writer")
res.addCapacity(0,"inf",1)

#作業，作業モード追加
for i in duration:
    act[i]=m1.addActivity('Act[{0}]'.format(i),duedate=due[i])
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    mode[i].addResource(res,{(0,'inf'):1})
    act[i].addModes(mode[i])
```

作業追加メソッドaddActivity(作業名，納期，作業重み(規定値1))

Pythonインターフェイスによる記述(3)
(納期遅れ最小化問題1:Example6.py)

```
#パラメータ設定と求解
m1.Params.TimeLimit=1
m1.Params.Makespan=False
m1.optimize()
```

納期遅れ最小化を目的とするため,パラメータの **Makespan** は **False** .

結果

会社名	A社	B社	C社	D社
作業時間	1日	2日	3日	4日
納期	5日目	9日目	6日目	4日目
納品日	5日目	10日目	8日目	4日目
納期遅れ	0日	1日	2日	0日

納期遅れ和:3日

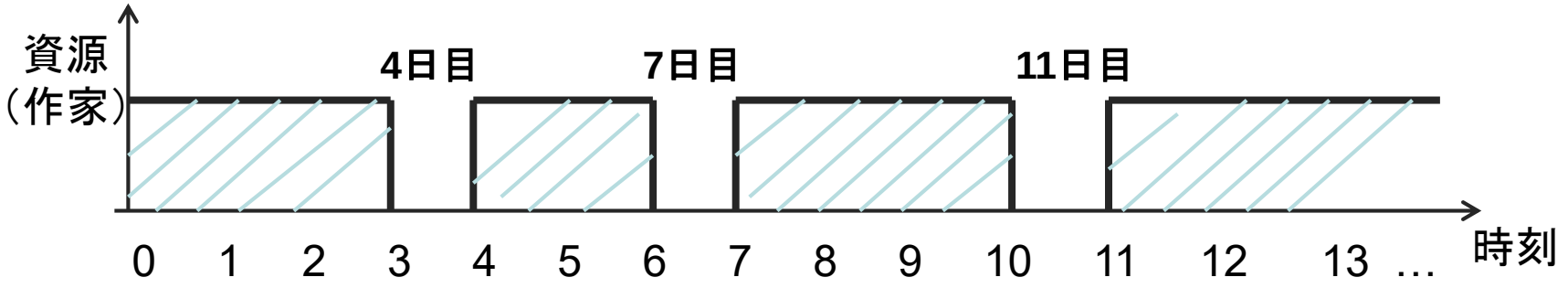
納期遅れを最小にしよう2！

--納期，一般化資源

休み追加

- 1人の作家（4日目，7日目，11日目は休み）
- 4社（A，B，C，D）から依頼された原稿を書く（作業）
- 納期遅れの和が最小になるように，スケジュールを組みたい．

会社名	A社	B社	C社	D社
作業時間	1日	2日	3日	4日
納期	5日後	9日後	6日後	4日後

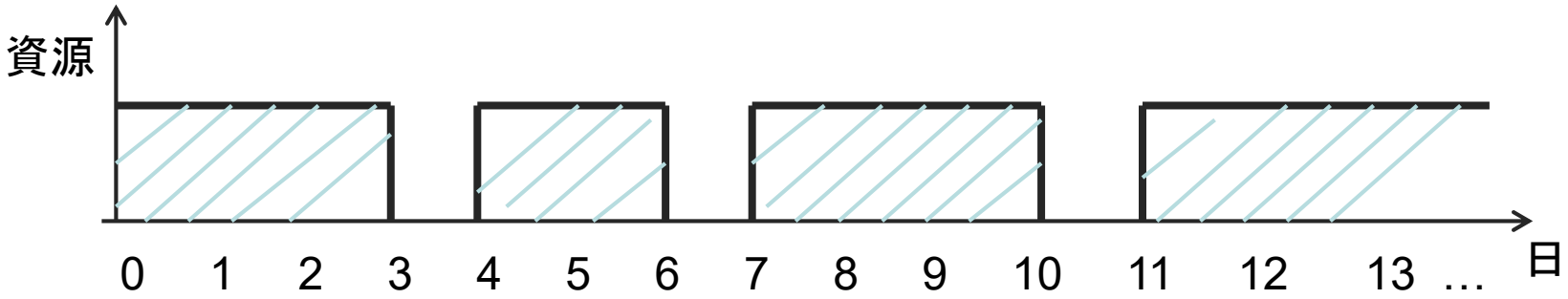


Pythonインターフェイスによる記述（1）
（納期遅れ最小化問題2：Example9_1.py）

```
m1=Model()        #m1と言うモデルオブジェクトを生成

due={1:5,2:9,3:6,4:4}      #各社の納期
duration={1:1, 2:2, 3:3, 4:4 }    #各社の原稿の作業時間

act={}
mode={}
```

Pythonインターフェイスによる記述（2）
（納期遅れ最小化問題2：Example9_1.py）

```
#資源を追加する
res=m1.addResource('writer')
res.addCapacity(0,3,1)
res.addCapacity(4,6,1)
res.addCapacity(7,10,1)
res.addCapacity(11,'inf',1)
```



Pythonインターフェイスによる記述（3）
（ 納期遅れ最小化問題2：Example9_1.py ）

```
for i in duration:        #作業，作業モード追加
    act[i]=m1.addActivity('Act[{0}]'.format(i),duedate=due[i])
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    mode[i].addResource(res,{(0,'inf'):1})
    act[i].addModes(mode[i])
```

作業追加メソッドaddActivity（作業名，納期，作業重み(規定値1)）

Pythonインターフェイスによる記述(4)
(納期遅れ最小化問題2:Example9_1.py)

```
#パラメータ設定と求解
m1.Params.TimeLimit=1
m1.Params.Makespan=False
m1.optimize()
```

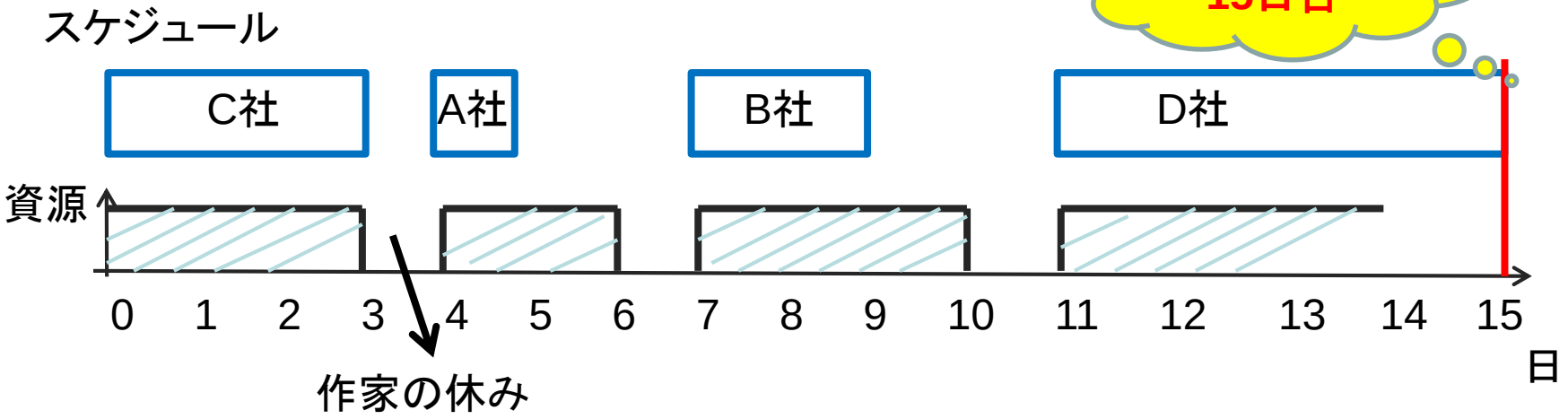
納期遅れ最小化を目的とするため,パラメータの **Makespan** は **False** .

納期遅れ最小化問題2(結果)

会社名	A社	B社	C社	D社
作業時間	1日	2日	3日	4日
納期	5日目	9日目	6日目	4日目
納品日	5日目	9日目	3日目	15日目
納期遅れ	0日	0日	0日	11日

納期遅れ和:11日

作業終了は
15日目

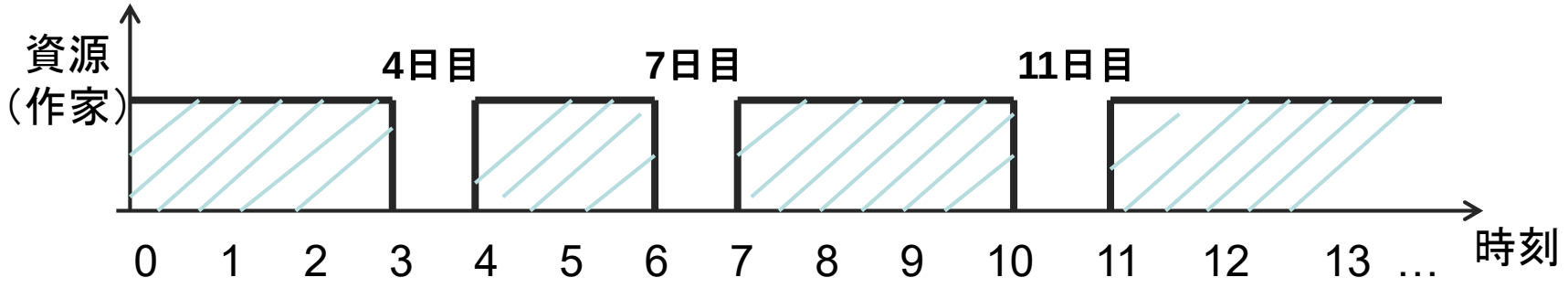


納期遅れを最小にしよう3！

-- 納期，作業の途中中断

- 1人の作家（4日目，7日目，11日目は休み）
- 4社（A，B，C，D）から依頼された原稿を書く（作業）
- 納期遅れの和が最小になるように，スケジュールを組みたい．
- 各作業は最大1日まで途中中断可能 ← 追加制約

会社名	A社	B社	C社	D社
作業時間	1日	2日	3日	4日
納期	5日後	9日後	6日後	4日後

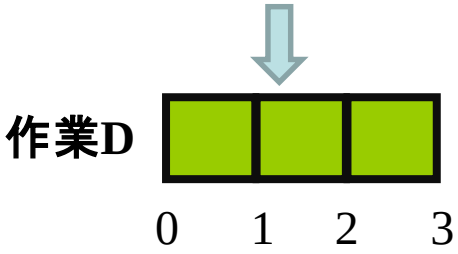


作業の中断1

作業の中断：緊急の作業が入ってくるなどいま行っている作業を途中で中 断して，別の（緊急で行わなければならない）作業を行った後に，再び中断していた作業を途中から行う．

- **作業Dの作業時間は3時間**
- **作業Dは途中で1時間まで中断可能**

→ **作業Dを1時間単位で3つの小作業に分割**



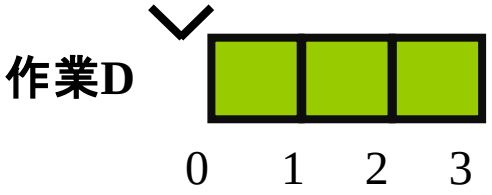
addBreak(0,3,1)

- **中断可能な小作業の開始時刻**
- **中断可能な小作業の終了時刻**
- **最大中断可能時間**

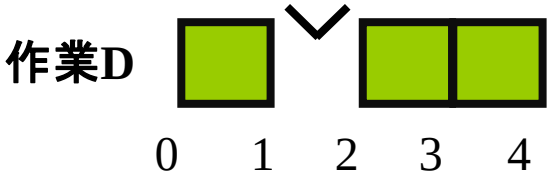
例えば，作業Dは前の2時間の内，最大1時間まで中断可能である場合 → addBreak(0,2,1)

作業の中断2

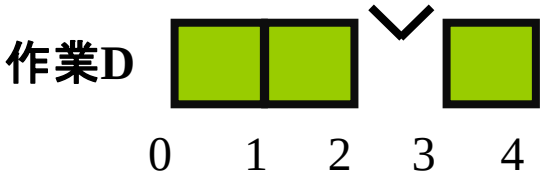
addBreak(0,3,1)の場合，
下記の中断のパターンの中から1つ選択される



作業Dが開始する前に中断



作業Dが開始し，1時間経過後中断



作業Dが開始し，2時間経過後中断

Pythonインターフェイスによる記述（1）
（納期遅れ最小化問題3：Example9_2.py ）

その他の条件，制約は納期遅れ最小化問題1のExample9_1.pyと同じ.

```
for i in duration:        #作業，作業モード追加
    act[i]=m1.addActivity('Act[{0}]'.format(i),duedate=due[i])
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    mode[i].addResource(res,{(0,'inf'):1})
    mode[i].addBreak(0,duration[i],1)

    act[i].addModes(mode[i])
```

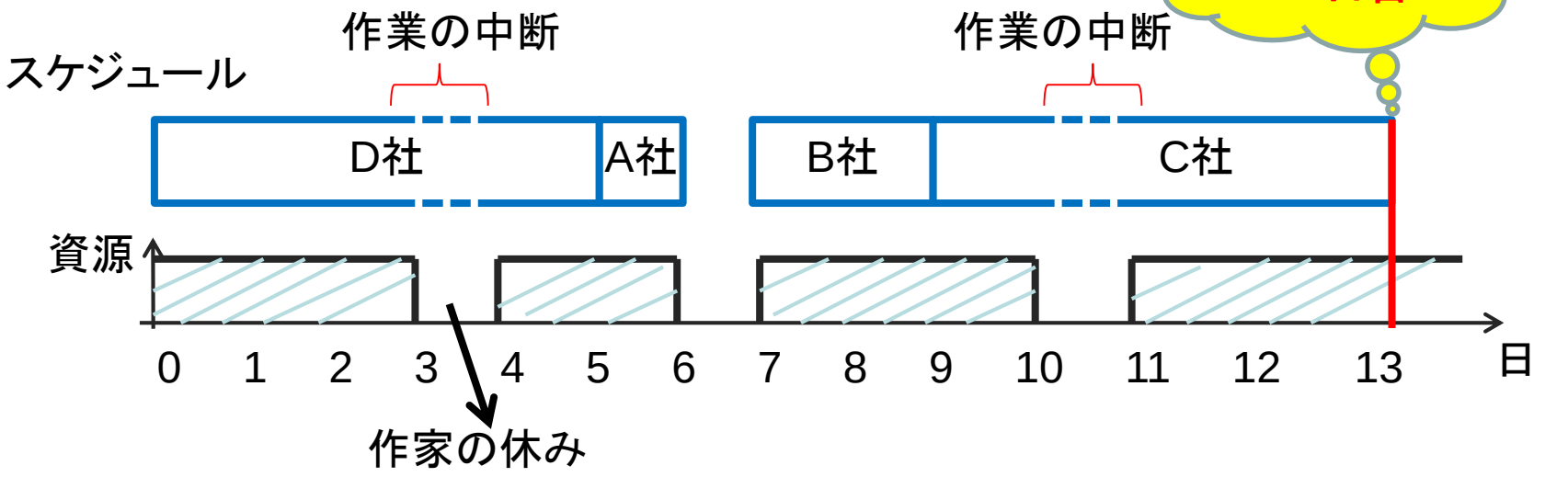
中断追加メソッドaddBreak（作業開始時刻，作業終了時刻，最大中断可能時間）

納期遅れ最小化問題3(結果)

会社名	A社	B社	C社	D社
作業時間	1日	2日	3日	4日
納期	5日目	9日目	6日目	4日目
納品日	6日目	9日目	13日目	5日目
納期遅れ	1日	0日	7日	1日

納期遅れ和:9日

作業終了は
13日目



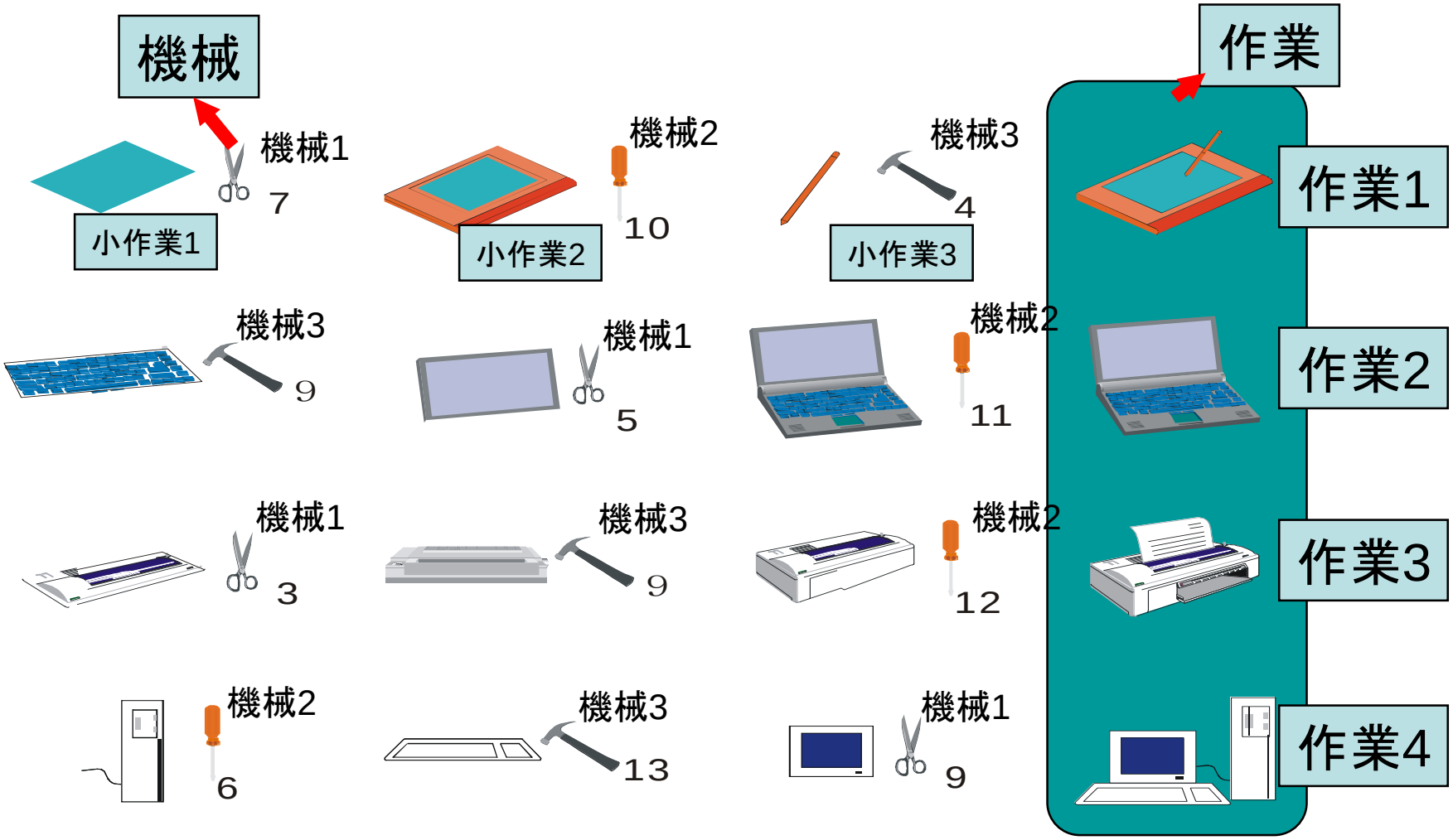
例題Part 7

ジョブショップスケジューリング
並列作業，作業中断，資源の占有

ジョブショップスケジューリング

--並列作業，作業中断，資源の占有

ジョブ（作業）ごとに機械（工程）順が異なっても良い！



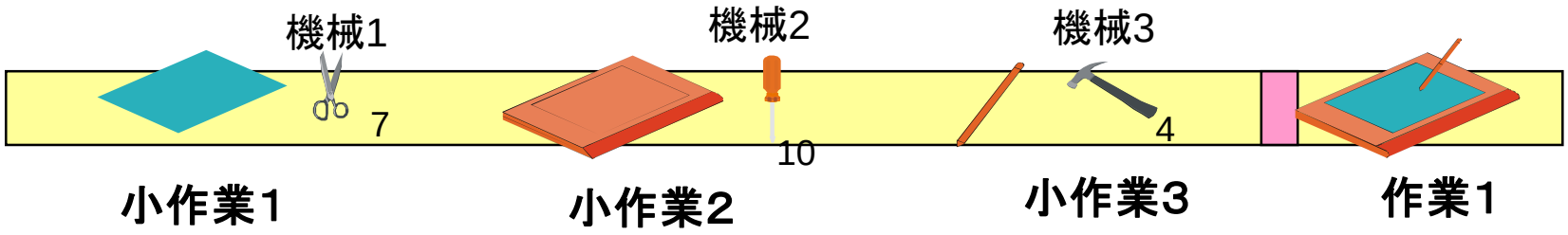
ジョブショップスケジューリング

- 各作業は，最初の２日間は作業員の作業が必要
- 各作業は小作業1,2,3の順に処理される
- 各作業は１日目の作業後休むことが可能
- 作業員は平日（週５日間）のみ作業可能
- 作業員は１日最大２人作業可能
- 機械1での作業は，最初の1日は2個まで並列処理可能
- 機械２はexpressモードで処理可能；処理時間は４日間に短縮可能
- expressモードは最大１回しか使用できない
- 機械１での作業２は，作業１が完了した直後に行う（他の作業が間に入らない）

Pythonインターフェイスによる記述（1）
（ジョブショップスケジューリング問題：Example11.py）

```
m1=Model()
# 3種類の機械；各機械は1つの作業しか処理できない
machine={}
for j in range(1,4):
    machine[j]=m1.addResource("machine[{0}]".format(j),capacity
    ={(0,"inf"):1})

#  作業員は平日のみ作業可能；平日の作業可能な作業員上限2人
manpower=m1.addResource("manpower")
for t in range(9):
    manpower.addCapacity(t*7,t*7+5,2)
```

Pythonインターフェイスによる記述(2)
(ジョブショップスケジューリング問題:Example11.py)



小作業:1　機械:1

作業:1　作業時間:7

```
JobInfo={ (1,1):(1,7), (1,2):(2,10), (1,3):(3,4),
        (2,1):(3,9), (2,2):(1,5), (2,3):(2,11),
        (3,1):(1,3), (3,2):(3,9), (3,3):(2,12),
        (4,1):(2,6), (4,2):(3,13), (4,3):(1,9)
    }
```

Pythonインターフェイスによる記述（3）
（ジョブショップスケジューリング問題：Example11.py）

```
# 機械2はexpressモードで処理可能
# expressモードが1回しか使えないことを資源の使用可能量(rhs) =1で表現
budget=m1.addResource(“budget_constraint”,rhs=1)

#   express の作業時間は4日，機械2を1単位使用
express=Mode(“Express”,duration=4)
express.addResource(machine[2],{(0,"inf"):1},"max")

#  最初の2日間は作業員が1人必要
express.addResource(manpower,{(0,2):1})

#  1日目作業後休憩可能
express.addBreak(1,1)
```

Pythonインターフェイスによる記述（4）
（ジョブショップスケジューリング問題：Example11.py）

```
#作業へのモードの追加とモデルへの作業の追加
act={}
mode={}
for (i,j) in JobInfo:
    act[i,j]=m1.addActivity("Act[{0}][{1}]".format(i,j))
    mode[i,j]=Mode("Mode[{0}][{1}]".format(i,j),duration=JobInfo[i,j][1])
    mode[i,j].addResource(machine[JobInfo[i,j][0]],{(0,"inf"):1},"max")
    mode[i,j].addResource(manpower,{(0,2):1})
    mode[i,j].addBreak(1,1)

    if JobInfo[i,j][0]==1:
        #機械1の最初の1日は2個まで並列処理可能
        mode[i,j].addParallel(1,1,2)
```

次のページへ続く

Pythonインターフェイスによる記述（5）
（ジョブショップスケジューリング問題：Example11.py）

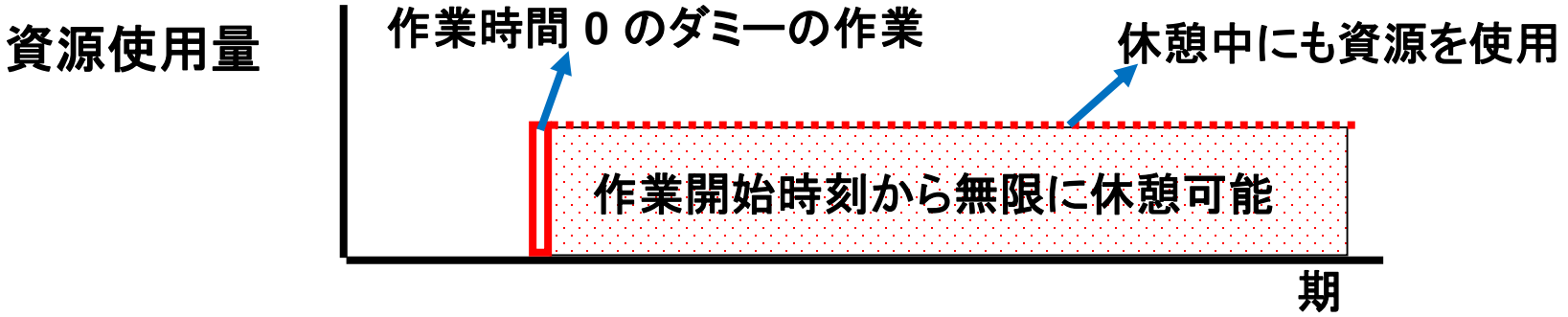
```
    if JobInfo[i,j][0]==2:
        #機械2は通常モードとExpressモードがある
        act[i,j].addModes(mode[i,j],express)
        #Express モードはbudgetを1使う
        budget.addTerms(1,act[i,j],express)
    else:
        act[i,j].addModes(mode[i,j]) #その他の場合モードは1つ

#各作業は子作業1,2,3の作業順で処理される
for i in range(1,5):
    for j in range(1,3):
        m1.addTemporal(act[i,j],act[i,j+1])
```

Pythonインターフェイスによる記述（6）
（ジョブショップスケジューリング問題：Example11.py）

機械1での作業2は，作業1が完了した直後に行う（他の作業が間に入らない）

ダミーの作業を用いて表現

```
d_act=m1.addActivity("dummy_activity")  # ダミーの作業を追加
d_mode=Mode("dummy_mode")               # ダミーのモードを追加
d_mode.addBreak(0,0)                    # 作業開始時刻から無限に休憩可能
d_mode.addResource(machine[1],{(0,0):1},"break")  # 休憩中にも資源を使用
d_act.addModes(d_mode)                  # 作業にモードを追加
```



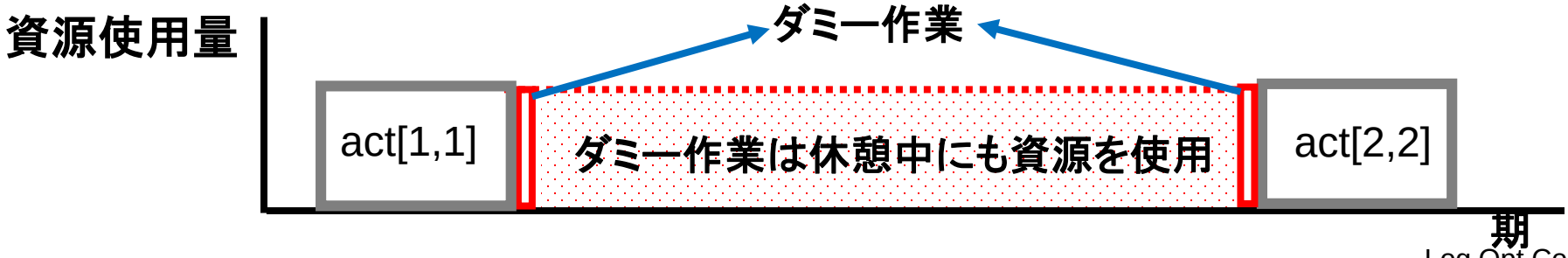
Pythonインターフェイスによる記述（7）
（ジョブショップスケジューリング問題：Example11.py）

時間制約追加

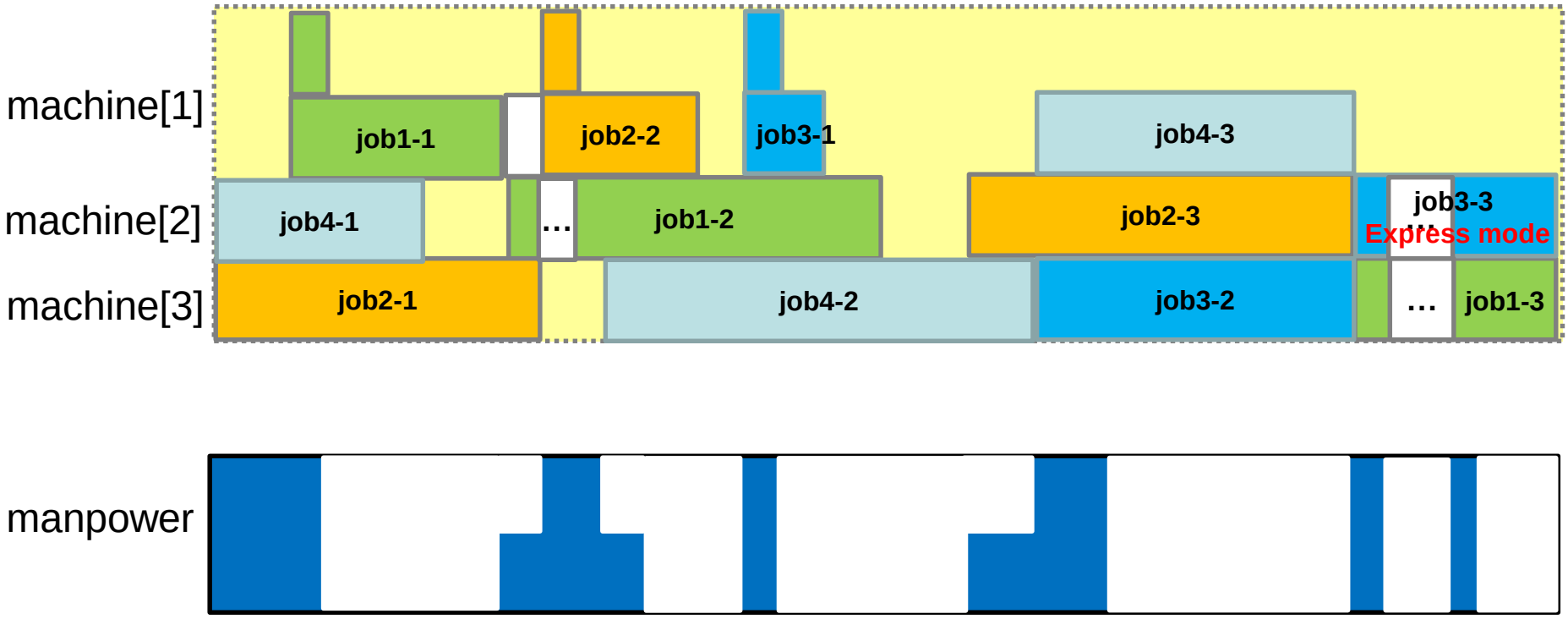
```
#作業1の小作業1(act[1,1]) が終了しないとダミー作業 (d_act) が開始できない
m1.addTemporal(act[1,1],d_act,tempType=“CS”)

#ダミー作業 (d_act) が開始しないと作業1の小作業1 (act[1,1]) が終了できない
m1.addTemporal(d_act,act[1,1],tempType="SC")

m1.addTemporal(d_act,act[2,2],tempType="CS")   # (1) , (2) と同様
m1.addTemporal(act[2,2],d_act,tempType="SC")   # (1) , (2) と同様
```

結論：　作業1の小作業1 はダミー作業の直後に開始

ダミー作業は作業1の小作業1 の直後に開始



ジョブショップスケジューリング結果



例題Part 8

段取りを考慮した生産問題1
段取りを考慮した生産問題2
状態変数

段取りを考慮した生産問題1

--状態変数

段取りを状態変数を用いて表現しよう！

Type1	Type2
Type1-作業1	Type2-作業1
Type1-作業2	Type2-作業2
Type1-作業3	Type2-作業3

- Type1とType2には各3つの作業がある
- 各作業の作業時間は3時間
- すべての作業は1つの機械で処理する
- 同じTypeの作業間の段取り時間は1時間
- 異なるTypeの作業間の段取り時間は5時間

状態変数

状態変数：時刻によって変化可能な非負の整数変数

- ある作業がある処理モードで開始された直後に変化させることができる.
  - モードが実行される前の状態
  - モードが実行される後の状態

状態変数の使い方

- 状態変数を定義

  状態インスタンス＝モデルインスタンス.addState(‘状態名称’)

- 状態の値を設定（特定の時刻にある値に変化させる）

  状態インスタンス.addValue(time=時刻,value=値)

- モードインスタンスに状態を追加

  モードインスタンス.addState(状態インスタンス,モードが実行される前の値,モードが実行された後の値)

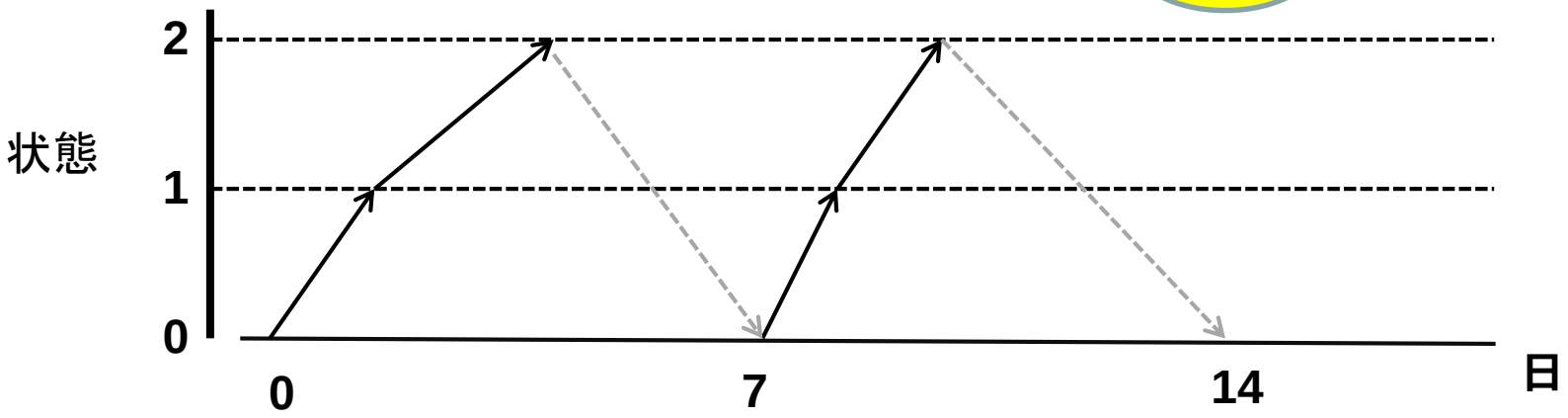
状態変数の使用例

- 1週間に高々2回だけあるモードを実行できない

```
state = model . addState ( )
state.addValue ( time=0, value=0 )
state.addValue ( time=7, value=0 )
state.addValue ( time=14, value=0 )
. . .
mode = Mode( 'mode' )
mode.addState ( state , 0 , 1)
mode.addState ( state , 1 , 2)
```

時刻0,7,14には状態を強制的に0に戻すことを表す

モード実行によって状態は2まで変更可能



Pythonインターフェイスによる記述（1）
（段取りを考慮した生産問題1：Example12.py）

```
m1=Model()
duration={(1,1):3,(1,2):3,(1,3):3,(2,1):3,(2,2):3,(2,3):3} #作業時間
setup={(1,1):1,(1,2):5,(2,1):5,(2,2):1} #Type間の段取り時間
act={}
act_setup={} #段取り作業
mode={} #作業モード
mode_setup={} #段取り作業モード
rs=m1.addResource("machine",1)
s1=m1.addState("Setup_State")
s1.addValue(time=0,value=1)
```

モデルに状態を追加：
addState(状態名)

初期状態を定義：. addValue(初期時刻，初期状態)

Pythonインターフェイスによる記述（2）
（段取りを考慮した生産問題1：Example12.py ）

```
#段取り作業モード
for (i,j) in setup:
    mode_setup[i,j]=Mode("Mode_setup"+str(i)+str(j),setup[i,j])
    mode_setup[i,j].addState(s1,i,j)
```

段取り作業モードに状態変化を追加：
. addState(**状態,モードが実行される前の値,モードが実行された後の値**)

```
mode_setup[i,j].addResource(rs,{(0,"inf"):1}) #段取り中も資源を使う
```

Pythonインターフェイスによる記述（3）
（段取りを考慮した生産問題1：Example12.py ）

```
for (i,j) in duration:
    act_setup[i,j]=m1.addActivity("Setup"+str(i)+str(j),autoselect=True)
```

addActivity(段取り作業名，autoselect=True)

```
    act_setup[i,j].addModes(mode_setup[1,i],mode_setup[2,i])

 #作業と作業モード追加
    act[i,j]=m1.addActivity("Act"+str(i)+str(j))
    mode[i,j]=Mode("Mode"+str(i)+str(j),duration[i,j])
    mode[i,j].addResource(rs,{(0,"inf"):1},"break")
    act[i,j].addModes(mode[i,j])
```

状態を使用する場合：
autoselectをTrueに設定した方が良い解を見つけることができる.

Pythonインターフェイスによる記述（4）
（段取りを考慮した生産問題1：Example12.py ）

```
#時間制約
for (i,j) in duration:
    m1.addTemporal(act_setup[i,j],act[i,j],"CS")
    m1.addTemporal(act[i,j],act_setup[i,j],"SC")

m1.Params.TimeLimit=1
m1.Params.Makespan=True
m1.optimize()
```

制約1：
各作業の段取りが終了後に
各作業が開始できる

段取り作業と作業の間に他の
作業を入れないための制約

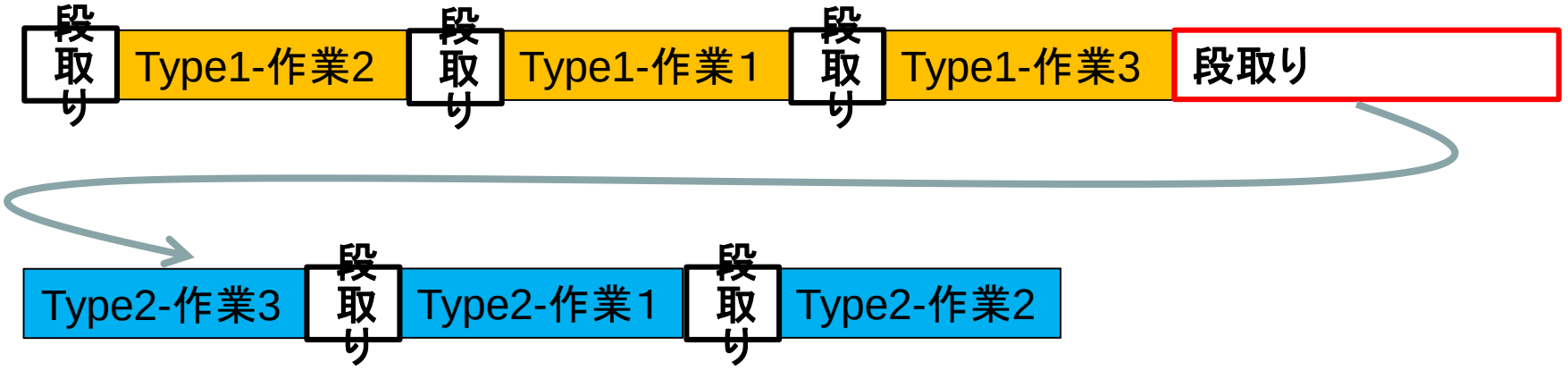
制約2：
各作業が開始する前に各作
業の段取りが終了する

制約1：
$end(act\_setup)+0 \leqq start(act)$

制約2：
$start(act)+0 \leqq end(act\_setup)$

下線の部分が同じであるため，両方の制約は等号である必要がある．

段取りを考慮した生産問題1 結果

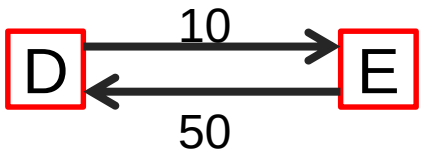


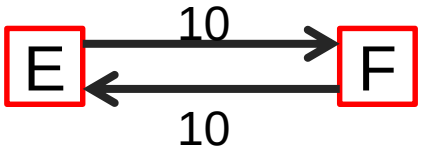
段取りのある生産問題2
-- 状態変数

目的:完了時刻最小化(段取り時間の最小化)になるような生産順番を決めたい

ライン1

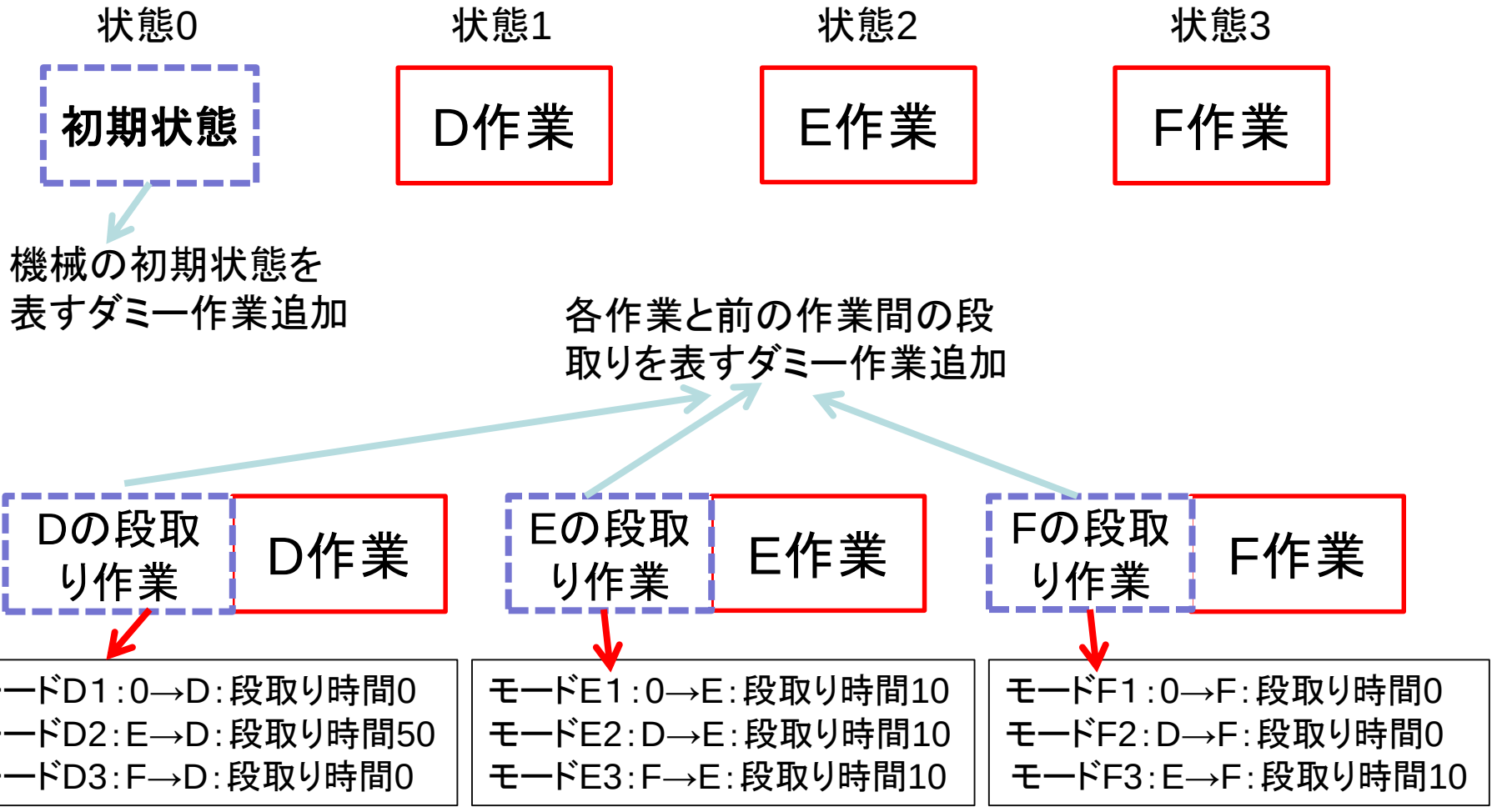
同一の生産ラインでD,E,Fを生産





D,EとE,F間の切り替えには段取り時間がかかる.

段取りを状態で表す


状態0
状態1
状態2
状態3
初期状態
D作業
E作業
F作業
機械の初期状態を
表すダミー作業追加
各作業と前の作業間の段
取りを表すダミー作業追加
Dの段取り作業
D作業
Eの段取り作業
E作業
Fの段取り作業
F作業
モードD1：0→D：段取り時間0
モードD2：E→D：段取り時間50
モードD3：F→D：段取り時間0
モードE1：0→E：段取り時間10
モードE2：D→E：段取り時間10
モードE3：F→E：段取り時間10
モードF1：0→F：段取り時間0
モードF2：D→F：段取り時間0
モードF3：E→F：段取り時間10

Pythonインターフェイスによる記述（1）
（段取りのある生産問題2：Example13.py）

```
m1=Model()
duration={'D':30,'E':30,'F':30}     #作業時間
setup={('D','E'):10,('E','D'):50,    #各作業間の段取り時間
        ('E','F'):10,('F','E'):10,
        ('start','D'):0,('start','E'):10,
        ('start','F'):0,('D','F'):0,('F','D'):0}
s={'D':1,'E':2,'F':3,'start':0}    #各作業の状態
rs=m1.addResource('line1',1)      #生産ラインを資源として追加
```

Pythonインターフェイスによる記述（2）
（段取りのある生産問題2：Example13.py）

```
#作業と作業モードを追加
act={}
mode={}
for i in duration:
    act[i]=m1.addActivity('Act_{0}'.format(i))
    mode[i]=Mode('Mode_{0}'.format(i),duration[i])
    mode[i].addResource(rs,{(0,'inf'):1})
    act[i].addModes(mode[i])
s1=m1.addState('Setup_State')
s1.addValue(time=0,value=s['start'])
```

モデルに状態を追加：
addState(状態名)

初期状態を定義：. addValue(初期時刻，初期状態)

Pythonインターフェイスによる記述(3)
(段取りのある生産問題2:Example13.py)

```
#段取り作業モード作成
mode_setup={}
for (i,j) in setup:
    mode_setup[i,j]=Mode('Mode_setup_{0}_{1}'.format(i,j),setup[i,j])
    mode_setup[i,j].addState(s1,s[i],s[j])
```

段取り作業モードに状態変化を追加:
. addState(**状態,モードが実行される前の値,モードが実行された後の値**)

```
    if setup[i,j] != 0:
        mode_setup[i,j].addResource(rs,{(0,setup[i,j]):1})  #段取り時間が0でない場合, 段取り作業中も機械が使用されることを表す.
```

Pythonインターフェイスによる記述（4）
（段取りのある生産問題2：Example13.py）

```
#段取り作業をモデルに追加
act_setup={}
for k in duration:
    act_setup[k]=m1.addActivity('Setup_{0}'.format(k),autoselect
    =True)
    for (i,j) in setup:
        if k==j:
            act_setup[k].addModes(mode_setup[i,j])  #段取り作業に
                                                    段取りモードを追加
```

addActivity(段取り作業名，autoselect=True)

Pythonインターフェイスによる記述（5）
（段取りのある生産問題2：Example13.py）

```
#時間制約の追加
for j in act_setup:
    m1.addTemporal(act_setup[j],act[j],'CS')
    m1.addTemporal(act[j],act_setup[j],'SC’)

m1.Params.TimeLimit=1
m1.Params.OutputFlag=True
m1.Params.Makespan=True
m1.optimize()
```

結果：

例題Part 9

初期解の設定法

初期解の設定方法

- 初期解を設定せずに求解成功すると自動的にoptseq_best_act_data.txtというファイルが生成される.
  このファイルが初期解ファイルである.

- optseq_best_act_data.txtがある環境で，モデルの属性Params.InitialをTrueに設定する.
  e.g.：モデルインスタンス.Params.Initial=True

例題は次のページ

初期解設定例

並列ジョブスケジューリング問題
optseq_best_act_data.txt

source ---
Act[1] Mode[1_3]
Act[2] ---
Act[4] ---
Act[7] ---
Act[5] ---
Act[9] ---
Act[6] ---
Act[8] ---
Act[3] ---
Act[10] ---
sink ---

並列ジョブスケジューリング問題:
Example4.py

```
m1.Params.TimeLimit=1
m1.Params.Initial=True
m1.Params.Makespan=True
m1.optimize()
m1.write('chart4.txt')
```

初期解設定

適用例

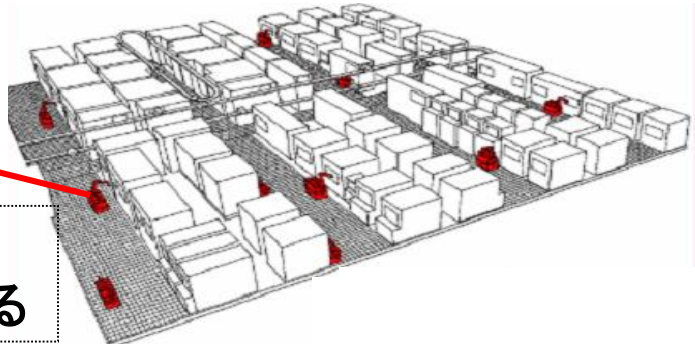
適用例

- 半導体工場の自動制御
  - 最大･最小の時間制約の利用
- 線路の競合回避
  - 資源制約の利用
- 化学生産スケジューリング
  - ダミー作業，中断中の資源利用
- 配送計画
  - 再生不能資源，複数モード，ルート生成法
- 会議スケジューリング
  - ダミーの利用

半導体工場の自動制御

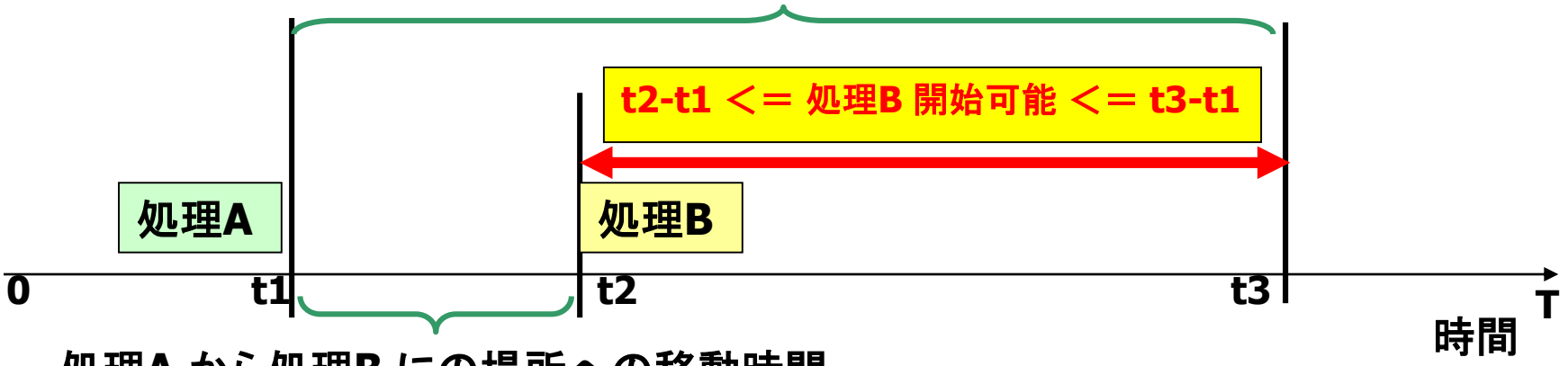
目標：搬送時間短縮
　　AGV間の衝突回避

半導体工場



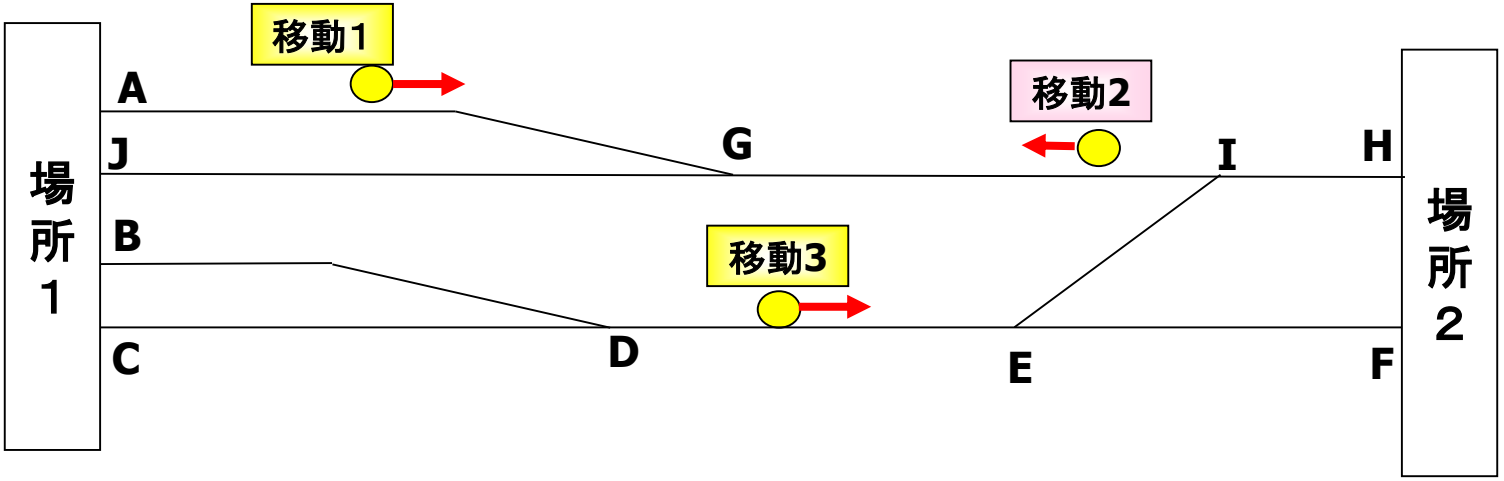
作業の順位：　処理A → 処理B
処理A と処理B の間には下記のような制約がある

処理A 終了後時刻 t3までに処理B を開始しなければならない



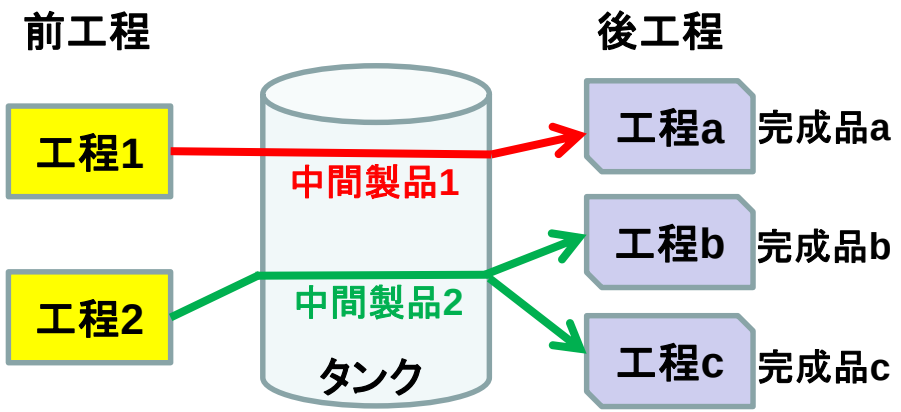
処理A から処理B にの場所への移動時間

線路上での競合回避



- 作業：複数のものを線路上で運ぶ．移動1，移動2など．
- 線路区間（A-Bなど）内では高々1つの移動作業しか行うことができない．
  -- 資源（線路）の使用可能上限が1で表現
- 線路上で停止することもある．
  -- OptSeqの作業の中断（Break）で表現可能
  -- 中断中も線路資源を使用可能と定義可能

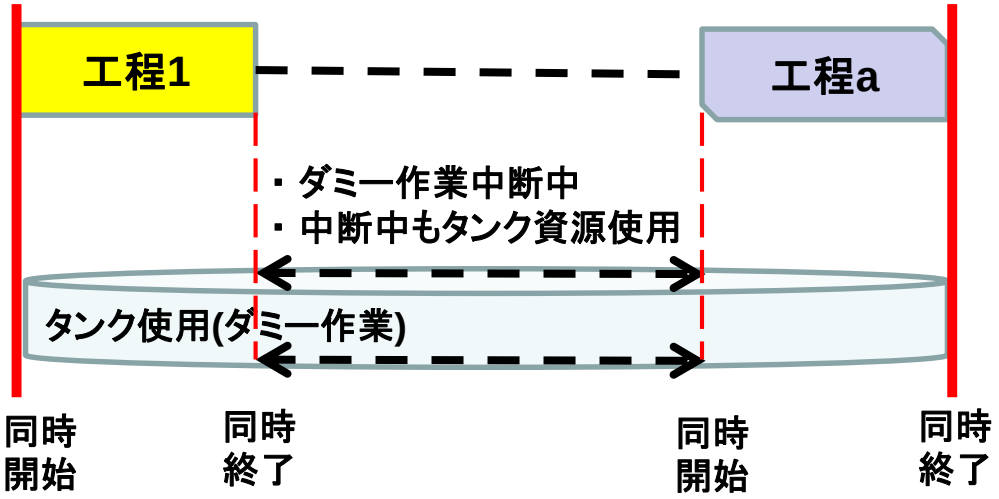
化学生産スケジューリング



・ 前工程で生産される中間製品はタンクに保存され，後工程の生産で使用される.

・ タンクには1種類の中間製品しか保存できない.（工程1から工程aが終了するまで，工程2は開始できない.）

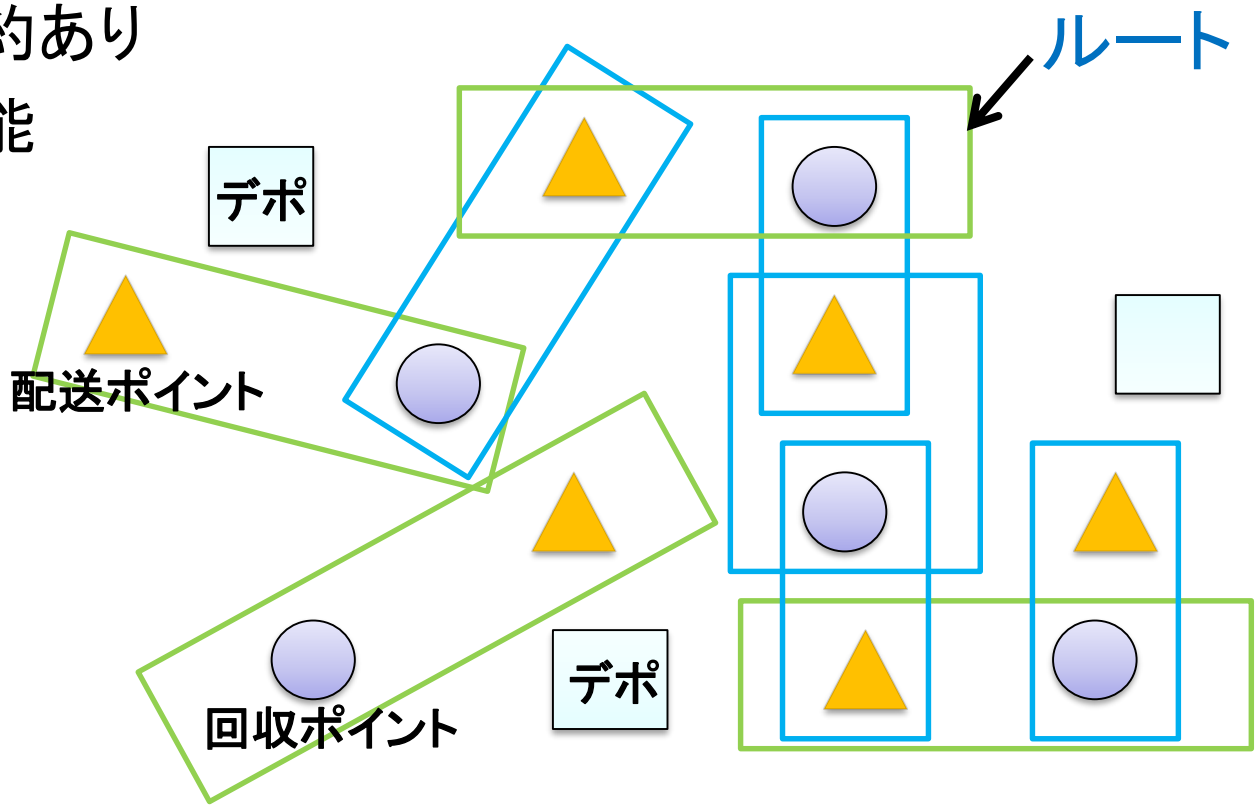
OptSeqのダミー作業と中断可能で工程1と工程aをタンクでつなぐことができる.



・ 工程1終了後工程aがすぐ開始しない場合は，タンク使用を表すダミー作業は中断可能と定義する

配送計画1

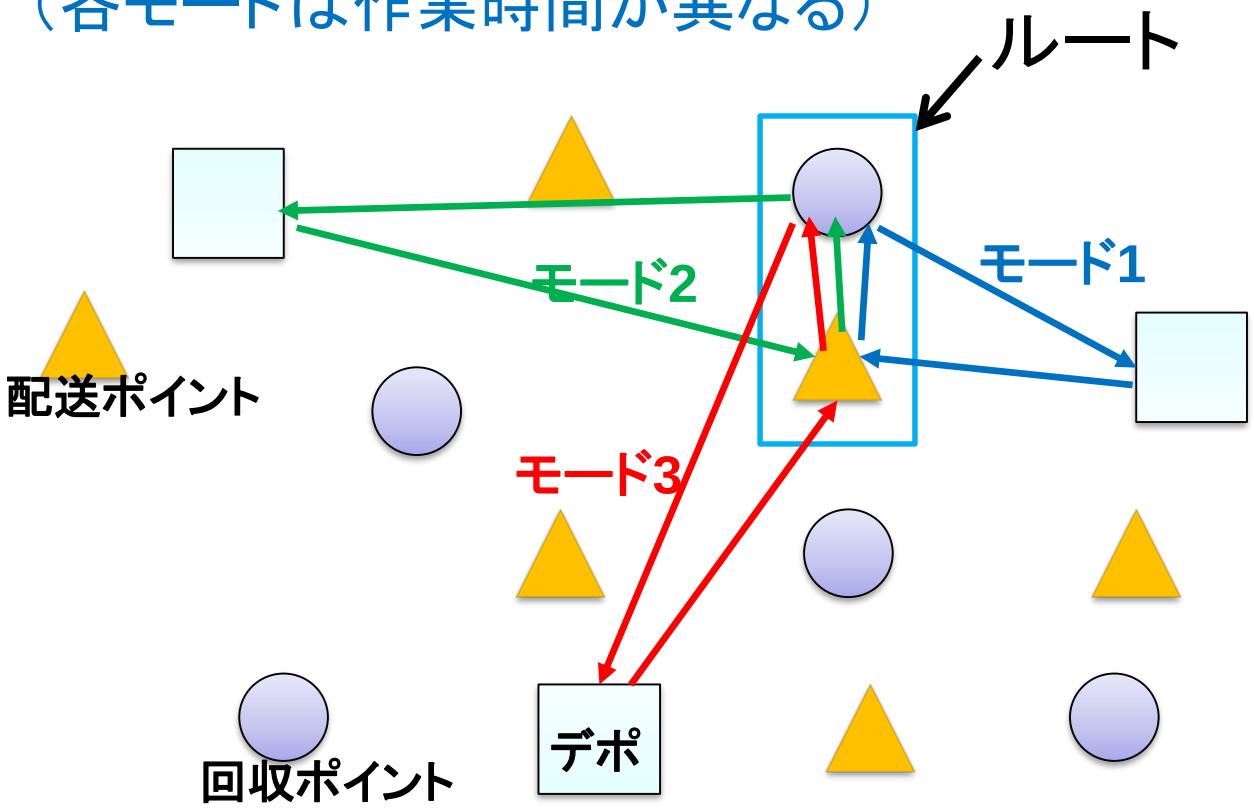
- デポから開始し，配送ポイントへのモノの配送，回収ポイントからのモノの回収を同時に行う
- トラック数，容量制限あり
- 時間指定制約あり
- 複数回転可能



使用可能な
ルートを生成

配送計画2

再生可能資源:トラック
再生不能資源:配送(回収)ポイント
複数モード:どのデポからまわるかをモード
（各モードは作業時間が異なる）



会議スケジューリング1

会議

- 長さ
- 優先順位(偉い役員参加会議優先)
- 開始可能時間
- 納期
- 毎週同じ時間に行う会議がある

会議室

- TV会議室
- 普通会議室
- 外部(費用がかかる)

メンバー

- 役員，秘書，報告者．．．
- 仲が悪い人同士はできるだけ同じ会議に参加しないようにする．
- 代理人を使用可能なメンバーがいる．

代理人使用可能な人のダミーを作成

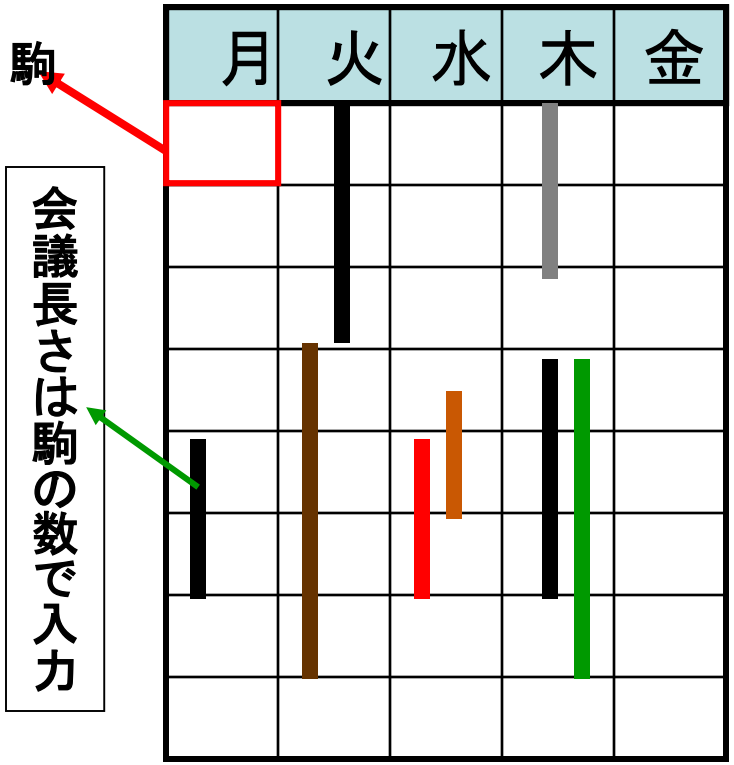
計画期間： 週，月

目的： 計画期間内の実行可能なスケジュール作成，コスト最小化

会議スケジューリング２

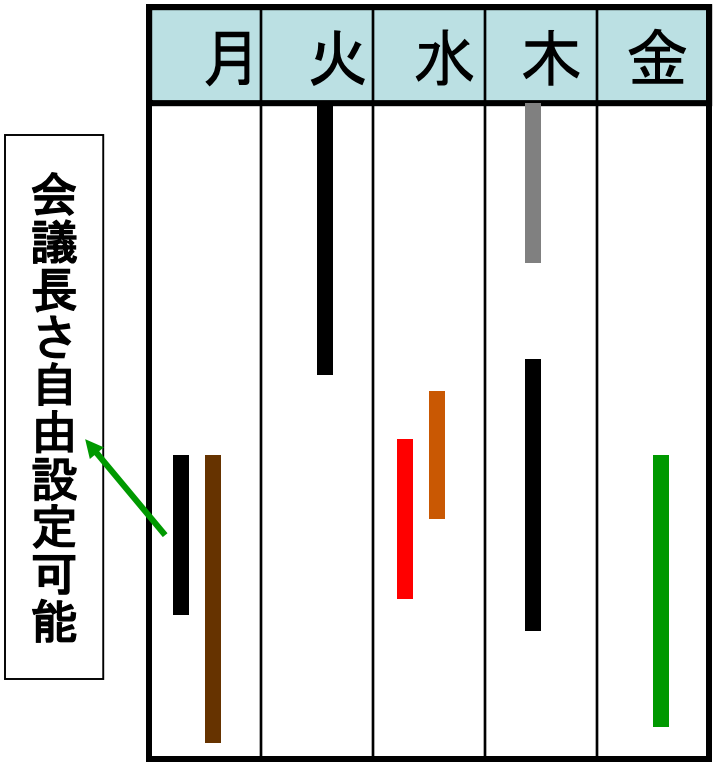
SCOP

離散時間：　期間をn個の駒に分割



OptSeq

連続時間



その他の適用例

- 工場内の各種生産スケジューリング
  - 作業間の時間制約，資源制約，段取りなど考慮
- 造船工場など巨大建造物の生産スケジューリング
  - 資源の占有，作業間の時間制約など考慮
- 点検作業のスケジューリング
  - 作業員のレベルや作業間の先後制約，作業の優先順位など考慮