Advanced algorithms
Exercises and solutions

X. Hilaire
hilairex@esiee.fr

April 7, 2019

1 Introduction – complexity

1.1 Factorial

1.1.1 Statement

Show that

$$n! = \omega(2^n)$$

.

1.1.2 Solution

We need to show that for any $a > 0$, there exists an integer n_0 such that the inequality

$$n! \geq a.2^n$$

holds true for any $n \geq n_0$. Or, equivalenty,

$$\frac{n!}{2^n} \geq a$$

where a can be as large as desired. To show this, suffice to see that

$$\begin{aligned} n! &= 1 \times 2 \times 3 \times ... \times n \\ &\geq 2 \times 3^{n-2} \end{aligned}$$

for any $n \geq 3$. Hence, we have that

$$\frac{n!}{2^n} \geq \frac{2 \times 3^{n-2}}{2^n} = \frac{1}{2} \times \left(\frac{3}{2}\right)^{n-2} \geq a$$

which is always true for n large enough.

1.2 Statement

Let $f, g : \mathbb{N}^+ \to \mathbb{N}^+$ be two functions. Prove or invalidate each of the following claims:

1. if $f(n) = O(g(n))$ then $g(n) = O(f(n))$
2. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$ (assuming that it exists)
3. if $f(n) = O(g(n))$ then $\log(f(n)) = O(\log(g(n))$
4. if $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$
5. $f(n) = O(f^2(n))$
6. if $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$
7. $f(n) = \Theta(f(n/2))$
8. $f(n) + o(f(n)) = \Theta(f(n))$

1.3 Solution

1. False. Counter-example: $f(n) = n$, and $g(n) = n^2$. We well have $f(n) = O(n^2)$, since this is equivalent to $n \leq an^2$, or $1/n \leq a$, so suffice to choose $a = 1$. However, the converse would require $n^2 \leq an$, or $n \leq a$ to hold true, that is, a can't be a constant.

2. False. Counter-example: $f(n) = 1$, $g(n) = n$. Then $\min(f(n), g(n)) = 1$ whatever n, but clearly, $f(n) + g(n) = 1 + n \neq O(1)$, so $f(n) + g(n) \neq \Theta(\min(f(n), g(n)))$ as well.

3. True provided $g(n) > 1$ for n large enough. Since we know that $f(n) = O(g(n))$, there must exist constants $n_0 > 0$ and $a > 0$ such that

$$\begin{aligned} \forall n \geq n_0, \quad f(n) &\leq ag(n) \\ \log f(n) &\leq \log a + \log g(n) \end{aligned}$$

   On the other hand, $\log f(n) = O(\log g(n))$ means there would exist $b > 0$ such that

$$\log f(n) \leq b \log g(n)$$

   for n large enough. Can we find a value of b that would keep both inequalities true? Indeed,

$$\begin{aligned} &\log a + \log g(n) \leq b \log g(n) \\ \Leftrightarrow &\frac{\log a}{\log g(n)} + 1 \leq b \end{aligned}$$

   tells us that b should be chosen as $1 + \frac{\log a}{\inf_n g(n)}$ to ensure this, and this is possible only if $g(n) > 1$ for n large enough. If g periodically hits 1 as n increases, then the claim is false.

4. True. Again, $f(n) = O(g(n))$ means there exist $a > 0$ such that $f(n) \leq ag(n)$ holds true for n large enough. On the other hand, $\log f(n) = O(\log g(n))$ means we can find b such that

$$\begin{aligned} \log f(n) &\leq b \log g(n) \\ f(n) &\leq g(n)^b \end{aligned}$$

holds true for n large enough. To conclude, suffice to see that

$$\begin{aligned} & ag(n) \leq g(n)^b \\ \Leftrightarrow & a \leq g(n)^{b-1} \\ \Leftrightarrow & \log a \leq (b-1)g(n) \\ \Leftrightarrow & \frac{\log a}{g(n)} + 1 \leq b \end{aligned}$$

so $b > 1 + \log a$ is always a possible choice.

5. True. Indeed, $f(n) \leq af^2(n)$ is equivalent to $a \geq \frac{1}{f(n)}$, so suffice to choose $a = 1$ to keep the inequality happy.

6. True. If $f(n) = O(g(n))$, then there exist $a > 0$ and $n_0 > 0$ such that

$$\begin{aligned} n \geq n_0 \Rightarrow & f(n) \leq ag(n) \\ \Leftrightarrow & g(n) \geq \frac{1}{a} f(n) \end{aligned}$$

which is the very definition of Ω with the rescaling constant chosen as $1/a$.

7. False. Counter-example: $f(n) = \exp n$. Then $\exp n \leq a \exp(n/2)$ implies $a \geq \exp(n/2)$, so $a \to \infty$ as $n \to \infty$, a contradiction to the assumption it has to be constant.

8. True. Indeed,

$$\begin{aligned} f(n) + o(f(n)) &\leq f(n) + af(n) \\ &= (1+a)f(n) \end{aligned}$$

for some constant $a > 0$ and n large enough. Then, very clearly

$$(1+a)f(n) \leq bf(n) \Rightarrow b \geq 1 + a$$

shows the validity of the equality in O, and

$$(1+a)f(n) \geq cf(n) \Rightarrow c \leq 1 + a$$

that in Ω, and the result follows.

2 Divide and conquer

2.1 Recurrences

2.1.1 Statement

Find the Ω and O bounds for each of the following recurrences (floors $\lfloor . \rfloor$ have been omitted to simplify notations):

- $T(n) = 2T(n/3) + n^2$
- $T(n) = T(n-1) + n$
- $T(n) = T(n-1) + 1/n$
- $T(n) = T(n-1) + T(n-2) + n/2$

- $T(n) = T(n/2) + T(n/3)$
- $T(n) = 2T(n/2) + \log n$
- $T(n) = T(\sqrt{n}) + 1$
- $T(n) = \sqrt{n}T(\sqrt{n}) + n$

2.1.2 Solution

- $T(n) = 2T(n/3) + n^2 = \theta(n^2)$ by copy pasting the calculation derived in the proof of the Master theorem, and identifying $a = 2, b = 3, d = 2$
- $T(n) = T(n-1) + n = T(n-2) + n + n - 1 = ... = T(1) + \frac{n(n-1)}{2} = \Theta(n^2)$
- $T(n) = T(n-1) + \frac{1}{n} = \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + ... + T(1) = T(1) + H(n) - 1$, by definition of the harmonic number. But for positive integers $x = n$, $\log(x) = \int_1^x \frac{dt}{t}$ admits a double bounding as
$$-1 + \sum_{i=1}^{n} \frac{1}{i} \leq \log n \leq \sum_{i=1}^{n} \frac{1}{i}$$
which show $H(n) = \Theta(\log(n))$
- $T(n) = T(n-1)+T(n-2)+n/2 : \Omega(n) = O(n) = F_n$, by application of the substitution method and the definition of the Fibonacci numbers. The $n/2$ term does not dominate, just as in the proof of the Master theorem.
- $T(n) = T(n/2)+T(n/3)$: What can be said is $2T(n/3) < T(n/2) < 2T(n/2)$. A copy-paste of the calculation derived in the proof of the Master theorem gives $\Omega(n^{\log_3(2)}) \approx \Omega(n^{0.63})$ for the left hand side, and $O(n)$ on the right-hand side.
- $T(n) = 2T(n/2) + \log n$. Assume $n = 2^p$ for the time being. It is easily seen that $T(n) = 2^p T(n/2^p) + \sum_{i=0}^{p-1} 2^i \log(n/2^i)$. The last term evaluates to
$$\begin{aligned}
&\sum_{i=0}^{p-1} 2^i \log(\frac{n}{2^i}) \\
&= (2^p - 1)\log n - \sum_{i=0}^{p-1} i2^i \log 2 \\
&= (2^p - 1)\log n - (2 - 2^{1+p} + p2^p)\log 2 \\
&= (n-1)\log n - (n(\log n - 2) + 2)\log 2
\end{aligned}$$
Thus, $T(n) = n + (n-1)\log n - (n(\log n - 2) + 2)\log 2 = \Theta(n \log n)$. If $n \neq 2^p$, then there exists k such that $2^k < n < 2^{k+1}$. Since both $T(n = 2^k)$ and $T(n = 2^{k+1})$ are $\Theta(n \log n)$, if follows that $T(n) = \Theta(n \log n)$ too.
- $T(n) = T(\sqrt{n}) + 1$

$$\begin{aligned}
T(n) &= T(n^{\frac{1}{2}}) + 1 \\
&= T(n^{\frac{1}{4}}) + 1 + 1 \\
&= T(n^{\frac{1}{8}}) + 1 + 1 + 1 \\
&= ... \\
&= T(n^{1/2^h}) + h
\end{aligned}$$

Two answers are possible then:

- Either we should consider that the argument n in $T(n)$ is non integer, and $\frac{1}{2^h}$ will never reach 0 for any h, even arbitrary large : in this case, the recurrence relation is diverging, and it does not admit any Ω or O bounds
- Or this argument has to be integer, which assumes n must be a power of 2. In this case, we should stop recursing avec immediately after solving $T(n = 2)$, for which there is a solution in h:

$$\begin{aligned} n^{1/2^h} &< 2 \\ 2^h &> \frac{\log n}{\log 2} \\ h \log 2 &> \log\left(\frac{\log n}{\log 2}\right) \end{aligned}$$

which gives $T(n) = \Theta(\log \log n)$

- $T(n) = \sqrt{n}T(\sqrt{n}) + n$

2.2 Faster Fibonacci numbers

2.2.1 Statement

Can you think of a faster method `fibo3` to compute F_n? Sketch:

- Show that for any $n \geq 1$:

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

- Show that $O(\log n)$ matrix multiplications suffice to compute F_n
- Show that all computations involve at most $O(n)$ bits
- Conclude that the running time of `fibo3` must be no more than $O(n^2 \log n)$
- Assuming that 2 n-bits integers can be multiplied in $O(M(n))$ time, can you prove that `fibo3` is in $O(M(n))$ time?

2.2.2 Solution

By definition, we know that $F_{n+1} = F_n + F_{n-1}$. This reads, in matricial form,

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \mathbf{M} \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

which repeated n times, gives

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \mathbf{M}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Our problem is therefore to compute $\mathbf{M}^n$, for any value of n. We can do this thanks to the following function (C-like style, where `Matrix` is a special type for handling matrices):

```
Matrix power(Matrix M, unsigned n)
{
  Matrix P;

  /* trivial cases */
  if (n == 0) return I;  /* the identity matrix */
  if (n == 1) return M;

  /* non-trivial cases */
  P= power(M, n/2);

  if (n % 2 == 0)  /* n is even */
    return P*P;
  else
    return P*P*M;  /* n is odd */
}
```

We would initially call `power(M,n)`, with `n` as the desired value. If we count in terms of multiplications, say $T(n)$ for the input argument n, we have that

$$T(n) = T(n/2) + O(1)$$

as the function calls `power(M, n/2)` only *once*, and that the final results may eventually be multiplicated by $\mathbf{M}$, which is only an extra multiplication, so $O(1)$. The above recurrence relation resolves to $T(n) = O(\log n)$, the desired result.

We could also have proposed the following procedural code:

```
Matrix power(Matrix M, unsigned n)
{
  Matrix R = I; /* the result, initialized to identity */
  Matrix T = M; /* used to store the highest power of 2 of M */

  while (n > 0)
  {
    if (n % 1 == 1)
        R = R * T;

    T = T * T;     /* next square of T */
    n= n / 2;
  }

  return R;
}
```

which considers the binary decomposition of $n = \sum_i 2^i b_i$: bit b_i tells us whether the next power of 2 of M should be included in the result or not, which is what `n % 1 == 1` is testing. As n can not be made of more than $\log_2 n$ bits, the `while` loop runs in $O(\log n)$ steps too, and involves 2 multiplications each time. This agains leads to the same result.

Regarding the number of bits needed to compute $\mathbf{M}^n$, we know that $F_{n+1} = F_{n-1} + F_n$ and that $(F_n)_n$ is an increasing sequence. So we have that

$$F_{n+1} \leq 2F_n$$

which repeated n times give

$$F_n \leq 2^n$$

The higest number stored in $\mathbf{M}^n$ can therefore be no larger than 2^n, which requires n bits to be stored. We have 4 such numbers, a constant. So the number of bits needed is well bounded by $O(n)$. Since the nave multiplications of 2 n-bits integer takes $O(n^2)$ time (n additions of n-bit integers), and that no more than $\log(n)$ such multiplications may be involved, the overal complexity must be at worst $O(n^2 \log n)$.

Could we conclude that `fibo3` is $O(M(n))$ in time? Strictly speaking, it depends on $M(n)$, which is expected to be less than quadratic, but certainly more than linear. Let us suppose $M(n) \leq \alpha n^c$, where $c > 1$ is some constant. Then:

$$\begin{aligned} T(n) &= T(n/2) + \alpha n^c \\ &= T(n/4) + \alpha(n^c + (n/2)^c) \\ &= T(n/8) + \alpha(n^c + (n/2)^c + (n/4)^c) \\ &= ... \\ &= T(1) + \alpha n^c \left(\sum_{i=0}^{\lfloor \log_2(n) \rfloor - 1} 1/2^i \right)^c \end{aligned}$$

The last summand may involve an arbitrary large number of terms, but can never reach the value of 2. Therefore $T(n) \leq T(1) + 2^c \alpha n^c$, which shows $T(n) = O(M(n))$ for any $c > 1$.

2.3 Pr Sempron

2.3.1 Statement

Professor Sempron has n chipsets, not all of them are in good state. He can only test them pairwise, and according to the following:

- if a chip is good, its opinion on the other chip will always be: positive if the other chip is good, negatve otherwise
- if a chip is damaged, its opinion on the other chip is unpredictable
- a test if positive if and only if the opinions of both the chips are positive

1. Show it is impossible to diagnose which chips are reliable if more than $n/2$ of them are damaged.
2. Assuming that more than $n/2$ chips are good, show that only $\Theta(n)$ operations are sufficient to find a good chip amongst the n
3. Show that $\Theta(n)$ operations are sufficient to diagnose the good chips, still assuming more than $n/2$ of them are good.

2.3.2 Solution

1) For convenience and clarity, we think of the problem in terms of unoriented graphs: a chipset is represented by a node, and there is an edge between a pair of nodes (n_1, n_2) if and only if n_1's opinion about n_2 is that it is good *and* n_2's opinion about n_1 is that it is good. Three important facts follow :

1. There must be an edge between *any* pair (n_1, n_2) of good chipsets, for otherwise, at least one of them would answer negatively, tnus violating its definition.

2. Consequence of point 1: the subset of good chipset must form a clique (a graph with maximal connectivity)

3. The subset of damaged chipset must be disconnected from the clique of good chipsets: any edge $(good, bad)$ would link a good and a bad chipset together, which is not possible as the good one has to answer negatively in such case.

Appart from being disconnected from the set of good chipsets, the set of damaged chipset has no reason to assume any partical shape, as Fig. 1 shows. In particular, if it is *not* a clique, then it is identified as the one representing bad chipsets, even if it contains more than $n/2$ of them.

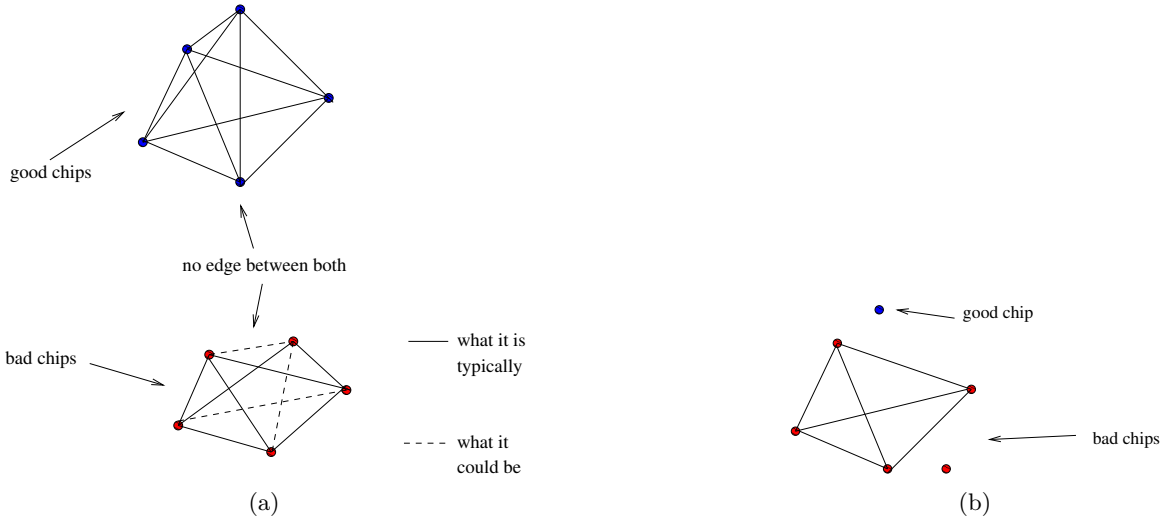

Figure 1: Pr Sempron's problem. (a) results of testing all possible pairs of chipsets; ((b) why the set of bad chipsets can't be chosen to be majoritary

However, in the worst case this set may perfectly be a clique too (all chipsets provide a fake answers). In this latter case, we are facing two cliques with no means of distinguishment... except if we add the requirement that one has to be majoritary. However, that one can't be the sef of damaged chipsets because of the counter-example shown in Fig. 1 (b) : 1 good chip only making a (degenerated) clique, n-2 damaged chipsets forming a clique, and 1 damaged chipset not connected to any other one. It is still impossible to distinguish which one of the isolated node is good or damaged. The result follows.

2) Call G the set of good chipsets, and D that of damaged ones. We pretend that Alg. 1 actually provides the answer to the problem in $\Theta(n)$ time.

Why works? The underlying idea is very simple: if you test any pair of chipsets (c_1, c_2), and find that the test is positive, then c_1 and c_2 must be of the same nature: either both good or both damaged. To see this, check that i) two good chipsets do have to answer positively, for otherwise the property that a good chipset always reports the truth about the other would be violated, and ii) the only other possible case for a positive test is that both are damaged. So in such case, we have 2 chipsets of the same nature, though we do not know (yet) this nature. It is therefore useless to consider both of them any further: suffice to keep one of them, and save the information that another chipset is of the same nature than that one.

On the contrary, let us assume that the test is negative. Then *at least* one of them is damaged. This means that one of the inequalities $|G| - 1 > |D| - 1$ or $|G| > |D| - 2$ must

Algorithm 1 Solving Pr. Sempron's problem.

1. Init: let $S := \{(c_1, 1), (c_2, 1), ..., (c_n, 1)\}$, where $c_1, ..., c_n$ represent the chipsets.
2. If $S = \{(c, .)\}$, then claim that chipset c is good, and stop.
3. Set $Z := \emptyset$.
4. Choose any two pairs of couples $(c_1, n_1), (c_2, n_2)$ from S such that $n_1 \leq n_2$. Remove these couples from S.
5. Test c_1 against c_2. If the test is positive, add $(c_2, n_1 + n_2)$ to Z.
6. If the test is negative and $n_1 \neq n_2$, add $(c2, n_2 - n_1)$ to Z
7. If $|S| \geq 2$, then iterate to step 4.
8. Set $S := S \cup Z$, and iterate to step 2.

hold in this case, therefore the hypothesis $|G| > |D|$ is preserved if we drop these two pairs (get them definitely away from the problem, without changing its solution).

Repeated applications of the following test leaves us to one of the following cases:

- We have "bags" of chipsets of size n_1 and n_2, and find that that any pair of representative tests positively: this amounts to saying that we have a single "bag" of size $n_1 + n_2$, all of the same nature
- We find that the test is negative. In that case, $n_2 - n_1$ pairs can be removed without changing the answer of the problem.

Thus, $|G| > |D|$ always holds in our method, and at step 8, $|S|$ becomes at worst $\lfloor |S-1|/2 \rfloor + 1$ in case n is odd, and there is only one more good chipset than damaged ones. Thus,

$$T(n) = T(n/2) + O(n)$$

holds, which resolves to $T(n) = O(n)$ by the Master theorem. For the lower bound, suffice to see that the best possible case when iterating from step 8 to step 2, is when all the pairs drawn at step 4 are of different nature, except maybe 1. Yet, this requires n testings, so $\Theta(n)$ time is justified.

NOTE: in a divide-and-conquer fashion, this amounts to go at finest level first (all chipsets tested pairwise), then build the higher level from conlusions drawn from the actual level. Though it leads to the same tree, the divide-and-conqer paradigm is somewhat different there, in the sense the method is better described as "bottom-up" than "top-down".

3) If we can select a good chipset thanks to question 2), and know that it is a good, then that chipset can be used to diagnose all other ones.. in linear time in all cases, so $\Theta(n)$ time again.

3 Dynamic programming

3.1 Tailcoat's problem

3.1.1 Statement

You are given a piece of fabric with integer dimensions $X \times Y$. You have a set of n template objects, each of which requires a piece of fabric with integer dimensions $x_i \times y_i$ to be copied.

If you produce a copy of object i, your profit is c_i; you can produce as many copies of any object you want, or none. You have a machine that can cut any piece of fabric into two pieces, either horizontally or vertically. Propose an algorithm which tells you how to maximize your profit.

3.1.2 Solution

In this solution, we make the assumption[1] that $x_i \leq y_i$ for all objects , and that the same holds for any piece of fabric at any time ($X \leq Y$).

Denote by S_i the set of objects whose either x dimension exactly equals i, and by $P(x, y)$ the best profit we can make for any piece of fabric with integer dimensions $x \times y$.

For the moment, assume that the dimensions of the piece of fabric are strictly greated than those of the largest object, in such a way that we *must* make at least a cut.

Suppose we know an optimal solution to the problem, and consider the first cut we are going to make. Two cases may occur:

- The optimal solution says we must cut alongside of the X axis, k units away from a border: we remain with 2 pieces of fabric A and B, whose dimensions are $X \times k$, and $X \times (Y - k)$
- The optimal solution says we must cut alongside of the Y axis, k units away from a border: we remain with 2 pieces of fabric A and B, whose dimensions are $k \times Y$, and $(X - k) \times Y$

The profit we make in each case is always $P(A) + P(B)$, and both $P(A)$ and $P(B)$ must correspond to optimal solutions too, for otherwise it would mean there would exist another solution yielding a strictly greater sum, contradicting our optimality hypothesis.

As we don't know neither in which direction, nor how far we should cut, we consider all possibilities, and write

$$P_x(x, y) = \max_{1 \leq k \leq \lfloor \frac{x}{2} \rfloor} P(x - k, y) + P(k, y) \tag{1}$$

$$P_y(x, y) = \max_{1 \leq k \leq \lfloor \frac{y}{2} \rfloor} P(x, y - k) + P(x, k) \tag{2}$$

to represent the optimal profit we make by cutting an $x \times y$ piece of fabric alongside of the X and Y axes, respectively. Only the best of this profit is of interest.

Let us come back now to the case our hypothesis on the dimensions of the piece of fabric does not hold. There are two ways to unassert it:

- The dimensions of the piece of fabric are exactly those of an object i whose profit is maximal. We don't need any cut, and this case our profit is just p_i.
- The piece of fabric is too small to cover any object. We don't need any cut as weel, and in this case our profit is 0.

To handle these latter cases, we define a $X \times Y$ matrix P_0 as

$$P_0(x, y) = \begin{cases} p_i & \text{if } x_i = x \text{ and } y_i = y \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

It then always holds that

[1] w.l.o.g, indeed simply rotate any object of 90 degrees to see the problem remains unchanged

$$P(x,y) = \min\{P_x(x,y), P_y(x,y), P_0(x,y)\} \tag{4}$$

How to turn (4) into concrete code?

1. We don't really need P_0: suffice to initiaize a single matrix P from P_0's values using (3), then update it for all possible couples (x,y) in case P_x or P_y does better ☞ the following code does so, in lines 10, and 59–61
2. We can't compute $P_y(x,y)$ before all $P_y(x,j), j < y$ are computed ☞ line 64: for x fixed, we compute all successive y starting from x itself (recall $y \geq x$ by convention)
3. We can't compute $P_x(x,y)$ before all $P_x(i,y), i < x$ are computed ☞ line 63 increments x after all $P(x, y \leq x)$ are computed
4. We don't need to compute $P(x,y)$ with $x > y$, as $P(x,y) = P(y,x)$ ☞ functions `Get_P` and procedure `Set_P` check which ceil of $P(x,y)$ or $P(y,x)$ should be accessed, and are systemtically used.
5. Procedures `Px` and `Py` implemenent (1) and (2) and return the best cut index and profit in `Best_K` and `Best_P`

```
1  procedure Dp_tailor is
2
3  type Cut is record
4     Xcut : Natural := 0; -- 0 means no cut // to Y
5     Ycut : Natural := 0; -- 0 means no cut // to X
6  end record;
7
8  Fx : constant Positive := 5; --  X dim. of the initial piece of fabric
9  Fy : constant Positive := 7; --  Y dim. ...
10 P : array(1..Fx,1..Fy) of Natural := (others => (others => 0));
11 Sol : array(1..Fx,1..Fy) of Cut; -- the solution
12 Kx, Ky, Cx, Cy : Natural;
13
14 function Get_P(X,Y : in Positive) return Natural is
15 begin
16    if (X <= Y) then
17       return P(X,Y);
18    else
19       return P(Y,X);
20    end if;
21 end Get_P;
22
23 procedure Set_P(X,Y : in Positive; Val : in Natural) is
24 begin
25    if (X <= Y) then
26       P(X,Y) := Val;
27    else
28       P(Y,X) := Val;
29    end if;
30 end Set_P;
31
32 procedure Px(X,Y: in Positive; Best_K,Best_P : out Natural) is
```

```
33 begin
34    Best_P := 0;
35    Best_K := 0;  -- no cut (possible, or optimal)
36    for K in 1..X/2 loop
37       if Get_P(K,Y)+Get_P(X-K,Y) > Best_P then
38          Best_P := Get_P(K,Y)+Get_P(X-K,Y);
39          Best_K := K;
40       end if;
41    end loop;
42 end Px;
43
44 procedure Py(X,Y: in Positive; Best_K, Best_P : out Natural) is
45 begin
46    Best_P := 0;
47    Best_K := 0; -- no cut (possible, or optimal)
48    for K in 1..Y/2 loop
49       if Get_P(X,K)+Get_P(X,Y-K) > Best_P then
50          Best_P := Get_P(X,K)+Get_P(X,Y-K);
51          Best_K := K;
52       end if;
53    end loop;
54 end Py;
55
56 begin
57    -- place a few random objects with profit
58    --
59    Set_P(1,1,1);
60    Set_P(1,2,3);
61    Set_P(2,2,10);
62
63    for X in 1..Fx loop
64       for Y in X..Fy loop
65          Px(X,Y,Kx,Cx);
66          Py(X,Y,Ky,Cy);
67          if (Cx > Get_P(X,Y)) then
68             Set_P(X,Y,Cx);
69             Sol(X,Y).Xcut := Kx;
70          end if;
71          if (Cy > Get_P(X,Y)) then
72             Set_P(X,Y,Cy);
73             Sol(X,Y).Ycut := Ky;
74          end if;
75       end loop; -- Y
76    end loop; -- X
77
78    -- print the solution
79    for Y in 1..Fy loop
80       for X in 1..Fx loop
81          Put(Natural'Image(P(X,Y)) & ":(" &
82              Natural'Image(Sol(X,Y).Xcut) & "," &
83              Natural'Image(Sol(X,Y).Ycut) & ") ");
84       end loop;
```

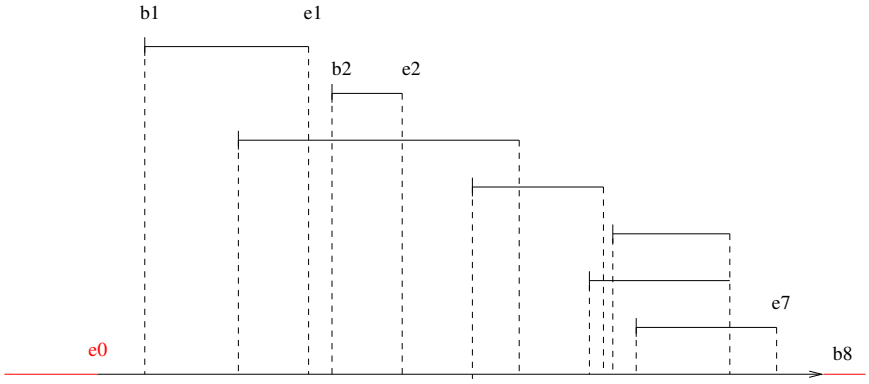

Figure 2: Task scheduling problem.

```
85          New_Line;
86      end loop;
87
88  end Dp_Tailor;
```

Remark 1. *Solving a single subproblem $x \times y$ amounts to computing a single element $P(x, y)$, and takes $\Theta(x + y)$ time as we have to examine all possible cuts of this piece of fabric. As we have exactly XY elemeets to compute to reach the solution, the overall complexity is $\Theta(X^2Y + Y^2X)$. It is polynomial in X and Y, but exponential in the number of bits it takes to encode X and Y, so the real complexity, expressed as the* size *of its input as usual, is indeed exponential. One talks about "pseudo-polynomial" solutions in such cases.*

Remark 2. *If all objects have their $x_i = 1$, then this problem is exactly the integer unbounded knapsack problem. We only have to examine all possible cuts perpendicular to the Y axis only in this case, which leads to a complexity of $\Theta(Y^2)$. Again, and as expected, the complexity is exponential in the number of bits it takes to encore Y. DP solutions might therefore be exponential in time, not contradicting the NP-completeness nature of a problem.*

3.2 Task scheduling with penalty

3.2.1 Statement

Suppose you have a machine that can process only a single task at a time. You have n such tasks $a_1, ..., a_n$. The duration of task j is t_j seconds, and its (absolute) execution deadline is d_j. If you terminate task a_i before d_j, you earn p_i Euros; otherwise, you earn nothing.

Propose an algorithm to find the scheduling that maximise your benefit.

You may assume that all datations are integers, and that any task may be schedulded more than once.

3.2.2 Solution

For convenience, we assume that the tasks are sorted by increasing deadline dates, as shown in Fig. 2. This will cost $O(n \log n)$ time. We also assume that if task i is schedulded at time t, it terminates slightly before $t + d_i$, in such a way that a new task can begin exactly at that time. We denote by $P(t)$ the best profit we can make at time t.

The underlying principle that we shall apply in both cases is that since tasks are independent one to each other, the optimal solution to the problem at any time t does not depend on how the optimal profit has be obtained at passed values $t = 1, ..., t - 1$, but only on the values of these profit. Consider task i: at any time $t \leq d_i - t_i$, we have to make a decision:

- Either we decide to schedule it : this implies that no other task can be scheduled from t to $t + t_i$, and the profit at $t + t_i$ will be that already obtained at t, plus b_i, the profit for achieving task i
- Or we decide not to schedule it : this implies that either the profit remains unchanged, or that we schedule another task (if we can).

In other words, it always holds that the optimal profit $P(t)$ at any time t obeys

$$P(t) = max(P(t-1), \max_{i \in S(t)} b_i + P(t - t_i)) \tag{5}$$

where $S(t) = \{1 \leq i \leq n : t + t_i \leq d_i\}$ is the set of tasks still schedulable at time t, and with the convention that $P(t) = -\infty$ for any integer $t < 0$. This results into the following $O(nn_d + n \log n)$ time algorithm, which applies Eq. 5 every second to update an array of profit P, and keeps track of the related task scheduled in a separate array T:

Algorithm 2 Scheduling tasks using dynamic programming

```
Initialization: P(i) := 0, T(i) := 0 for all 1 ≤ i ≤ d_n
for t := d_1..d_n do
  maxi := P(t − 1)
  for i ∈ S(t) do
    if maxi < P(t − t_i) + b_i then
      maxi := P(t − t_i) + b_i
      T(t) := i;
    end if
  end for
  P(t) := maxi;
end for
```

Practically, $S(n)$ can be implemented at no extra cost, as a function returning a pointer to the head of a list of remaining tasks, the latter being updated each time t equals a new deadline. Also, values of $T(n)$ are only meaningful when they come with a change of profit.

Remark: a simpler algorithm can also be derived in case any task i is scheduldable only once with fixed starting date $t = d_i - t_i$. In that case, it is only needed to update the profit for every "event" on the time axis in Fig. 2. Thus, the outer loop "for t := $d_1..d_n$" should be replaced by $1..2n$ – although there might pairs of identical ending or starting dates; yet, the number of events to be examined remains $2n$. The final complexity is therefore $O(n \log n + 2n) = O(n \log n)$.

3.3 Subsequence of maximum sum

3.3.1 Statement

A contiguous subsequence of a list T is a subsequence made up of consecutive elements of T. For instance, if T=$\{1, -3, 2, 7, 8, 0, 3\}$, then $\{-3, 2, 7\}$ is a subsequence, but $\{2, 7, 3\}$ is not. Give a linear time algorithm to determine the contiguous subsequence of T whose sum is maximal. *Hint: consider subsequences ending exactly at position j.*

3.3.2 Solution

As suggested by the hint in the statement, let us consider the subsquence S_j with maximum sum ending exactly at index $j \in [1 : N]$, assuming that $N = |S| \geq 1$. Because the ending

index of S_j is j (it *must* include $T(j)$), the only unknown is its starting index, which we shall denote by b_j.

Suppose we know b_j, and consider T_{j+1}, the best subsequence ending exactly at index $j+1$, and starting at some (unknown) index b_{j+1}. We claim the following result:

$$b_{j+1} = \begin{cases} b_j & \text{if } \sum_{i=b_j}^{j} T(i) \geq 0 \\ j+1 & \text{otherwise} \end{cases} \tag{6}$$

Proof. Denote by $\tau_j = \sum_{i=b_j}^{j} T(i)$, the sum conveyed by S_j. If both $\tau_j \geq 0$ and $T(j+1) \geq 0$, then $\tau_{j+1} = \tau_j + T(j+1) \geq \tau_j$, so we have improved the previous subsequence, without it be possible to improve it more: b_j is actually the best starting index for subsequence S_j, so $\tau_j + T(j+1)$ must be the maximum sum we can get in such case. Thus, $b_{j+1} = b_j$. If $\tau_j \geq 0$ but $T(j+1) < 0$, adding $T(j+1)$ to the former subsequence does indeed not improve the sum, however we can't do better: because S_{j+1} must end at index $j+1$, the maximal partial sum comes from S_j, and this partial sum can't be improved by definition of b_j. So $b_{j+1} = b_j$ again.

This proves the first case of the result. On the other hand, the second case corresponds to $\tau_j < 0$, and in this case, $T(j+1) + \tau_j < T(j+1)$ whatever the value of $T(j+1)$. This implies that the starting index of S_{j+1} should be $j+1$, in other words, S_{j+1} is just $T(j+1)$ itself.

Eq. (6) provides a means to compute b_{j+1} given b_j in $O(1)$ time, and obviously, $b_1 = 1$. Thus, repeated applications of Eq. (6) justify the $O(N)$ time algorithm: suffice to compute all b_j (and τ_j), and return that of maximal τ_j. However, we need not compute all of them: suffice to keep values for the best known subsequence at any time. This is what the following code does:

```
1  with Ada.Text_IO;
2  use Ada.Text_IO;
3
4  procedure Dp_Subseq is
5
6  T : constant array(1..8) of Integer := (10,-20,3,8,0,-2,-4, 7);
7  b : Positive := 1; -- starting index for subseq. ending at i
8  bs, be : Positive := 1; -- best known starting and ending indices
9
10 Sum : Integer := T(1);  -- sum of T(b..i)
11 Best : Integer := T(1); -- sum of T(bs..be)
12
13 begin
14    for i in T'First+1..T'Last loop
15
16       if Sum > 0 then
17         Sum := Sum+T(i);
18       else
19         Sum := T(i);
20         b := i;
21       end if;
22
23       if Best < Sum then  -- current partial sum better than best known?
24          bs := b; be := i;     -- => update
25          Best := Sum;
26       end if;
27    end loop;
```

```
28
29      Put("Best subsequence = T(" & Positive'Image(bs) & ".." & Positive'Image(be)
30      Put_Line(") , sum = " & Integer'Image(Best));
31  end Dp_Subseq;
```

4 Greedy algorithms

4.1 A long journey

4.1.1 Statement

You are going on a long journey between Antwerpen and Napoli. Once the tank of your car is filled, you know you can do at most n km. You have a roadmap that tells you where the fuel stations are located, and you would like to make as few stops as possible. Give an efficient algorithm to solve this problem.

Suggested steps:

1. Show that if your current position is p, and x and y are two consecutive towns you must visit, then the solution of refilling your tank at town y can't be worse that refilling at town x.
2. As a result, justify the greedy algorithm.

4.1.2 Solution

Denote by x_i, $x = 1, ..., k$ the absolute distance of fuel station i from Antwerpen, and put $x_0 =$ Antwerpen, and $x_{k+1} =$ Napoli. First observe that if $\exists i \in [2:n] : x_i - x_{i-1} < n$, then don't move and remain home! We shall obviously assume $x_{i+1} - x_i \leq n, \quad \forall i \in [2:n]$, so that there is always exist a next fuel station less than n km away from the last one.

The procedure is indeed very simple:

```
procedure Ga_Car is

   N : constant Integer := 8;
   K : constant Integer := 7;
   X : array(0..K+1) of integer := (0, 2, 5, 8, 11, 18, 29, 34, 45);
   I,D : Integer := 0;

begin
   while I < K+1 loop
      D := X(I);
      while (I < K+1) and (X(I)-D < N) loop
         I := I+1;  -- look ahead for next station
      end loop;
      if I < K+1 then
         Put_Line("Make a break at " & Integer'Image(X(I-1)));
      end if;
   end loop;
end Ga_Car;
```

Put briefly, when you leave station i, you always make a break at the farthest fuel station j ahead from you, but not further away than n km. Why works?

- If you stop at $k < j$, then your tank full, so you can reach station j equally well. The only difference is : you did an extra stop. So solution k is not better than solution j.
- If you wish to stop further than j, then the condition $x(j) - x(i) \leq n$ is broken.

As a summary, solution j cannot be improved, and since going from j to n is no dependant of how you reached j, this terminates the justification.

4.2 Scheduling lectures

4.2.1 Statement

Suppose you have a set of n lectures that need be scheduled in classrooms. Each lecture has fixed (non-modifiable) starting and ending times. You would like to use as few classrooms as possible to schedule all lectures.

1. Describe an naive $O(n^2)$ algorithm to determine the scheduling of lectures
2. Try to improve this solution to an $O(n \log n)$ time algorithm, and possibly $O(n)$ under the condition that all lectures start and end on exact hours, and that the algorithm is to be run daily.

4.2.2 Solution

Let $l_i = (s_i, e_i)$ represent lecture i, starting at date s_i and ending at date e_i. A naive solution could consist in maintaining a list C of classrooms as follows:

- for each lecture l:
  - search C for a classroom either unused or free at s_i
  - schedule lecture i in class C
- end for

A worst case occurs when there is as many lectures as classroom, and the beginning of lecture i is less than ending time of all other already schedulded lectures. In that case, the inner loop is $O(n)$, so overall $O(n^2)$.
We may improve this time to $O(n \log n)$ as follows. First, consider a quadruplet (i, d, b, c) where:

- i is a lecture index number
- d is either b_i or e_i
- b is true if and only if $d = b_i$
- c is a classroom number

To build this new solution, we need an array T of $2n$ such quadruplets, and a list F initially containing all classrooms numbers. The algorithm is given at Alg. 3.
The algorithm works roughly like this:

- T stores all events, which consists of all dates, either of beginning or of end of all lectures
- We walk T chronologically:

Algorithm 3 Greedy scheduling of lectures

```
for i=1..n do
  T[2i] := (i, b_i, true, 0)
  T[2i + 1] := (i, e_i, false, 0)
end for
Sort T by increasing d's first, then "increasing" b's (false < true)
for i=1..2n do
  if T[i].b is false then
    Pop the first element f in front of F
    Schedule lecture i in classroom f : T[i].c := f
  else
    Push integer T[i].c in front of F
  end if
end for
```

- If we find that the next event is the end of a course ($T[i].b$ is false), then we declare the classroom in which it was schedulded as free, and put it back *in front* of F.
- Otherwise, it is the beginning of a new course, so we schedule it in the first free classroom we find, and make the classroom unavailable til its end.

Note that in case several lectures begin and and at the same time, lectures who end are favoured and processed first: this, indeed, ensures optimality of the solution, as

- a classroom can not be reinserted in F earlier than the the end of the current lecture schedulded in it
- if two classroom are available for a new course, one already used, the other not, then the fact we use F as a LIFO ensures the former will be favoured over the latter.

The inner loop is in $O(n)$ time, and the complexity is dominated by that of sorting T, that is, $O(n \log n)$. If enumerating all possible date is not too costly (dates are typically integer hours of a day), it may even be possible to sort T in linear time from the date histogram.

4.3 Permutation

4.3.1 Statement

Suppose you have two sequences of n positive numbers $A = \{a_i\}_{i=1}^n$ and $B = \{b_i\}_{i=1}^n$. You are free to reorganize them as you want, after what, you get a profit of $\Pi_{i=1}^n a_i^{b_i}$. Give a strategy to maximize your profit.

Suggested steps:

1. Show (by contradiction) that if $x = \max_i a_i$ and $y = \max_i b_i$, then x^y must be a factor of the profit.
2. As a result, derive the optimal strategy.

4.3.2 Solution

Suffice to sort both a and b by decreasing order.

For the clarity of the proof, and w.l.o.g, we shall assume so. So a_1 is the smallest value of a whereas a_n is the largest. Likewise for b.

First of all, observe that maximizing the profit or its log are identical problems. Therefore we can try to maximize

$$P = \sum_{i=1}^{n} b_i \log a_i$$

equally well. Let us show by contradiction that $b_n \log a_n$ must be a term of P. Suppose $\log a_n$ be multiplied by something lower than b_n, say b_j, with both $j < n$ and $b_j < b_n$. The partial profit (for indices n and j) for this solution is

$$P_j = b_j \log a_n + b_n \log a_j$$

Simply observe that permutating indices n and j leads to a profit of

$$P_n = b_n \log a_n + b_j \log a_j$$

As a consequence

$$\begin{aligned} P_n - P_j &= (b_n - b_j) \log a_n - (b_n - b_j) \log a_j \\ &= (b_n - b_j) \log \frac{a_n}{a_j} \\ &> 0 \end{aligned}$$

because $a_n > a_j$ and $b_n > b_j$ by hypothesis. So P_n is a better solution than P_j whatever j is, provided $b_j < b_n$, a contradiction.

Obviously, if the maximum value of b occurs more than once in the sequence, then permutating makes no difference, but does not contradict the claim a_n must be expoentiated by the maximum value of b.

Since $b_n \log a_n$ must be a term of P, simply a_n and b_n from a and b, and recursively apply the above result to all suproblems of size $n-1$ down to 1 to conclude a and b have to be sorted as claimed.

4.4 Customer service

4.4.1 Statement

A service has n customers waiting to be served, and can only serve one at a time. Customer i will spend d_i minutes in the service, and will have to wait $\sum_{j=1}^{i-1} d_j$ minutes before being served. The penality for making customer i wait m minutes is mp_i, where $p_i > 0$ is some constant. You would like to schedule the customers, that is, find a permutation $\phi : [1:n] \mapsto [1:n]$ so as to minimize the overall penalty $P(\phi) = \sum_{i=1}^{n} p_{\phi(i)} \sum_{j=1}^{i-1} d_{\phi(j)}$.

1. Consider 3 customers C_1,C_2,C_3, with service duration 3,5,7, and priorities 6, 11, 9. Among the possible schedulings $\phi_1(1) = 1, \phi_1(2) = 3, \phi_1(3) = 2$ and $\phi_2(1) = 3, \phi_2(2) = 1, \phi_2(3) = 2$, which one is preferable?

2. Consider two schedulings ϕ_1 and ϕ_2, identical everywhere except that ϕ_1 makes customer j served immediately after customer i, while ϕ_2 does just the opposite. What does $\Delta = P(\phi_1) - P(\phi_2)$ equal to?

3. Derive the expression of an evaluation function f which associates a number to any customer i and decides whether $\Delta > 0$ or not.

4. Derive an algorithm for this problem, and justify it. Complexity?

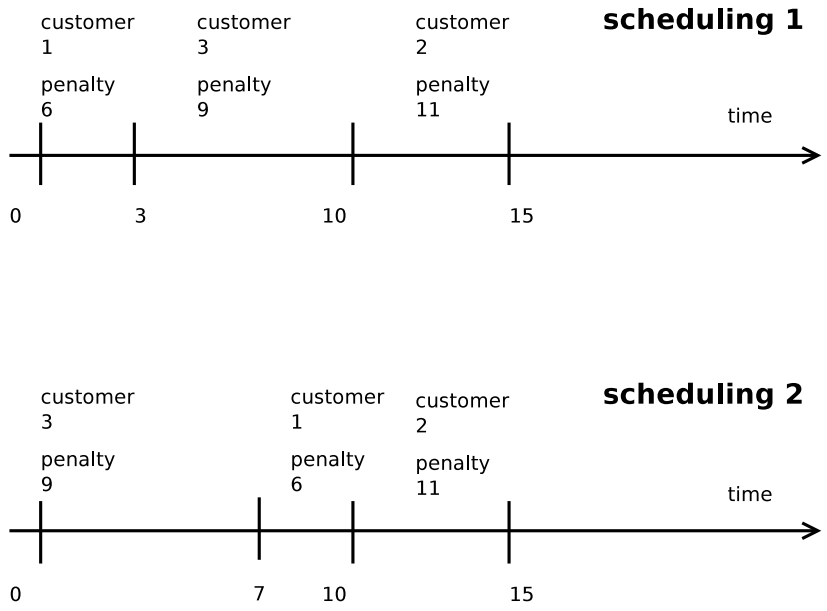

Figure 3:

4.4.2 Solution

Question 1

See Fig. 3. For the first scheduling,

$$P_1 = 0 \times 6 + 3 \times 9 + 11 \times 10 = 137$$

For the second scheduling,

$$P_2 = 0 \times 9 + 6 \times 7 + 11 \times 10 = 152$$

Thus, the first arrangement is better.

Question 2

See Fig. 4. We may first observe that:

- should we schedule customer i or j at rank k, the time at which we'll do that does only depend on the past, and neither on i nor j. It is some time T.
- in a similar way, what will be schedulded starting from rank $k+2$ does actually not depend on how schedulded i and i : the starting time is always $T' = T + d_i + d_j$, as $+$ is a commutative operator.

If we schedule customer i first, then he will wait T minutes, and will cost Tp_i to the service. Then will come customer j, who will wait $T + d_i$, and will cost $(T + d_i)p_j$. The total cost is therefore

$$P(\phi_1) = Tp_i + (T + d_i)p_j$$

On the other hand, if we schedule customer j first, and j next, then we get a cost of

$$P(\phi_2) = Tp_j + (T + d_j)p_i$$

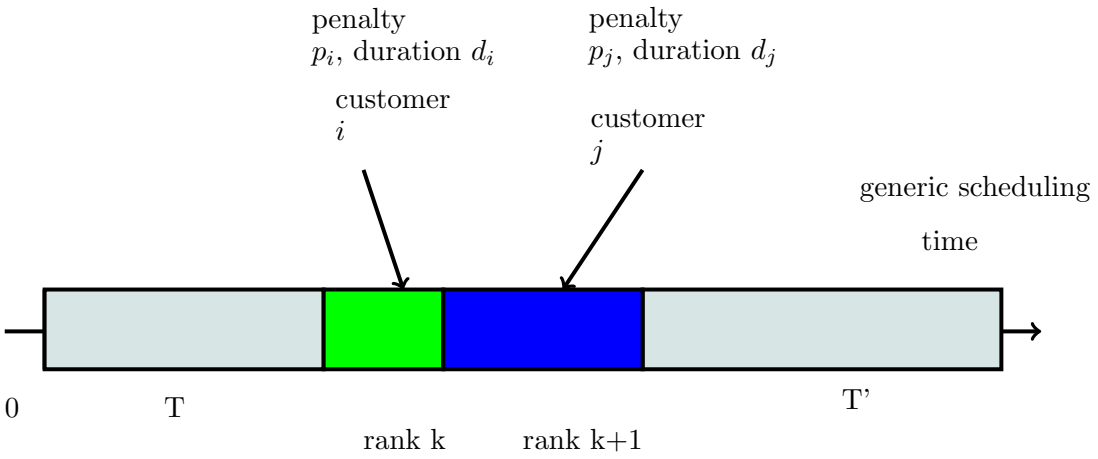

Figure 4:

The difference between both is

$$\begin{aligned}\Delta = P(\phi_1) - P(\phi_2) &= Tp_i + Tp_j + d_i p_j \\ &\quad - Tp_j - Tp_i - d_j p_i \\ &= d_i p_j - d_j p_i\end{aligned}$$

Question 3

We have

$$\begin{aligned}\Delta > 0 \Leftrightarrow & d_i p_j > d_j p_i \\ & \frac{d_i}{d_j} - \frac{p_i}{p_j} > 0\end{aligned}$$

We may then define

$$f(i) = \frac{d_i}{d_j} - \frac{p_i}{p_j}$$

The function should be seen as an $f_j(i)$, as it depends on j. However, its sign will reveal whether customer i and j are ranked in the correct order, or if they should be swapped. Formally:

$$f_j(i) > 0 \Leftrightarrow \text{customers } i, j \texttt{ must be swapped} \tag{7}$$

Question 4

Eq. 7 gives a condition to determine whether two customers ranked consecutively must be swapped or not, and it works everywhere : this is the very definition requested for the sorting operator in any sorting algorithm.

Hence, to solve the problem, suffice to sort according to f. The algorithm is greedy in that it selects first the customer with lowest possible cost. Any sortging algorithm can do the job, for instance, mergesort. Its complexity is $O(n \log n)$.

4.5 Change and coins

4.5.1 Statement

Consider the problem of giving back change on n cents using as few coins as possible.

1. Give a greedy algorithm that gives back change using coins of 50, 20, 10, 5 and 1 cents. Show that it is optimal. *Hints: you should do this in a case-by-case fashion: by considering all possible values of n in [1:100], compare, in each case, how many coins the greedy solution requires to how many a concurrent can require at best.*

2. Suppose now you have $k+1$ coins whose values are powers of some constant $\gamma > 0$, that is, $1, \gamma, \gamma^2, ..., \gamma^k$. Prove that your greedy algorithm is still optimal. *Suggested steps:*

   (a) *Show that if h is the highest integer for which the numbers of coins with value γ^h differ between the greedy and a concurrent solution, then it is impossible that the concurrent solution returns more coins of that value than the greedy.*

   (b) *Taking this result in account, show that any concurrent solution will return at least one more coin than the greedy solution. Hints: i) try to see what happens for $\gamma = 2$; ii) generalize for any $\gamma > 0$, and remember that for any $r \neq 1$,*

$$\sum_{i=0}^{n} r^n = \frac{r^{n+1} - 1}{r - 1}$$

3. Give a set of values for the coins for which the greedy solution is *not* optimal.

4. Give an optimal $O(nk)$ algorithm which gives back change whatever the values of the coins – but assuming there is always a coin of 1 cent. *Suggested steps:*

   (a) *Suppose the optimal solution involves the subproblem of giving back change on $p < n$ cents. Which other subproblem does this implies?*

   (b) *Using this, and by considering p can span all possible values of the coins, give the recurrence relation.*

   (c) *Derive the code which will fill the table. Simply check its complexity is $O(nk)$.*

4.5.2 Solution

Question 1 Put $v_1 = 1$, $v_2 = 5$, $v_3 = 10$, $v_4 = 20$, $v_5 = 50$. A code for the greedy algorithm could merely look as follows:

```
procedure Gr_Coins is

V : constant array(1..5) of Positive := (1, 5, 10, 20, 50);
N,R : natural;

begin
   Put("Give n > 0: ");
   Get(Item => N);

   for I in reverse V'Range loop
      R := N / V(I);  -- integer division
      Put_Line("Give " & natural'Image(R) & " coins of value " & Natural'Image(V(I)));
      N := N rem V(I);  -- integer remainder of N divided by V(I)
   end loop;
end Gr_Coins;
```

Why is it optimal? Consider the two expansions $n = \sum_{i=1}^{5} g_i v_i$ and $n = \sum_{i=1}^{5} c_i v_i$, that correspond to the greedy, and to a concurrent (non-greedy) solution, respectively. Let h be the highest integer such that $c_h \neq g_h$. Observe the following two points:

h	v_h	$n \in$	concurrent uses .. coins	greedy uses at most .. coins
1	1	$[1:5[$	N/A	N/A
2	5	$[5:10[$	n	$n-4$
3	10	$[10:20[$	k	$k-1$ since some subset must sum up to 10
4	20	$[20:50[$	k	$k-1$ since some subset must sum up to 20
5	50	$[50:100[$	k	$k-2$ since some subset must sum up to 50

Table 1: Simplication cases for the greedy solution

1. By definition of the division, we necessarily have $c_h < g_h$, for otherwise we would have $c_h \geq g_h + 1$ and the concurrent expansion would exceed n.
2. Inequality $g_i v_i < v_{i+1}$ holds for $1 \leq i \leq 4$, for otherwise the greedy algorithm would be able to put at least one additional coin of value v_{i+1}, a contradiction.

So $g_i < v_{i+1}/v_i$ from point 2, and $c_h < g_h$ from point 1, which implies $c_h < g_h < v_{h+1}/v_h$.
If we apply this double inequality with possible values for h and the additional constraint that $n \leq 99$, we get that $g_h \leq 1$ for $h = 2, 3$, and $g_4 \leq 2$. So there is always room so that the greedy solution can be distinguished with at least a coin of value v_h, and the concurrent solution must have fewer coins of that value.

Consider now an optimal solution that does include a coin of value c_h. Then the other coins represent the optimal solution to the problem of giving change on $n - v_h$ cents : if that was not true, it would mean there would exist a solution for giving change on $n - v_h$ cents using fewer coins, contradicting the optimality hypothesis of the whole solution without that it be possible to remove the coin with value v_h itself. This proves the optimal substructure of the problem.

To prove optimality, suffice to show that the concurrent solution, which includes *fewer* coins of value v_h, can always be improved to the greedy solution for each value of h. This is summarized in table 1 in a case-by-case fashion, as different values of h yield different possible intervals for n.

Question 2 We keep the same notations as in question 1. Inequalities established there still apply and will prove useful:

$$c_h < g_h \tag{8}$$

$$\forall i \in [1:k], \quad c_i < v_{h+1}/v_h = \gamma \tag{9}$$

Remind that (9) results from the assumption of optimality of the concurrent solution. We shall show that this assumption leads to a contradiction.

Denote by S_g and S_c the expansions of n according to the greedy and the concurrent solutions:

$$S_g = g_h \gamma^h + \sum_{i=0}^{h-1} g_i \gamma^i \tag{10}$$

$$S_c = c_h \gamma^h + \sum_{i=0}^{h-1} c_i \gamma^i \tag{11}$$

From (9) and (11) we get

$$\begin{aligned} S_c &\leq c_h\gamma^h + (\gamma - 1)\sum_{i=0}^{h-1}\gamma^i \\ &\leq c_h\gamma^h + (\gamma - 1)\frac{\gamma^h - 1}{\gamma - 1} \\ &\leq (c_h + 1)\gamma^h - 1 \end{aligned} \tag{12}$$

From (10), we immediately get

$$S_g \geq g_h\gamma^h \tag{13}$$

Combining (12) and (13) together, we arrive at

$$\begin{aligned} S_c - S_g &\leq (c_h + 1)\gamma^h - 1 - g_h\gamma^h \\ &\leq (c_h - g_h + 1)\gamma^h - 1 \\ &< 0 \end{aligned}$$

The last line holds because of (8), and $\gamma > 1$ (otherwise, distinguishing powers of γ is irrelevant). This contradicts $S_p > S_g$, therefore *any* concurrent solution cannot be better than the greedy solution.

Question 3 A counter-example can be easily built by choosing $n = p \times v_{h-1}$ exactly, and $v_h = v_{h-1} + 1$. For instance, $\{v_i\} = \{8, 7, 5, 1\}$, and $n = 35$:

$$\begin{aligned} n = 35 &= 4 \times 8 + 3 \times 1 && \rightarrow \text{ 7 coins} \\ &= 5 \times 7 && \rightarrow \text{ 5 coins} \end{aligned}$$

Question 3 Since the greedy choice does not always apply, the only exploitable property of the problem is that of optimal substructure: if an optimal solution to the problem of giving change on n cents involves a coin of value v, then the other coins represent the solution to the problem for giving change on $n - v$ cents, and must be optimal too.

The only problem now is that we don't know which coin should be included in the solution; so we try them all, discarding all those whose value is greater than n. In addition, the trivial case is when we have to give change on $n = 0$ cents, with no coins at all as solution.

Denote by $C(n)$ the "cost" for giving change on n cents, which equals the number of coins we use. This gives the relation :

$$\begin{aligned} \forall n \geq 1, \quad C(n) &= \min_{i \in [1:k], v_i \leq n} C(v_i) + C(n - v_i) \\ &= \min_{i \in [1:k], v_i \leq n} 1 + C(n - v_i) \end{aligned}$$

because the cost for giving change on exactly v_i cents is always 1 (1 coin of that value and nothing else). In addition, we have the trivial case

$$C(0) = 0$$

We can tabulate both in a DP program, as done in the following code:

```
K : constant Positive := 4;
V : constant array(1..K) of Positive := (1, 3, 8, 9);

procedure Tabulate(N : in Natural) is

   -- C and Coins are fully initialized to 0
   --
   C : array(0..N) of Natural := (others => 0);
   Coins: array(0..N,1..K) of Natural := ( others => (others => 0));
   Best_Sum, Best_I, I : Natural := Natural'Last;

begin
   for J in 1..N loop -- give change on J coins
      I := 1;
      Best_Sum := Natural'Last;
      while V(I) <= J loop
         if 1+C(J-V(I)) < Best_Sum then
            Best_Sum := 1+C(J-V(I));
            Best_I := I;
         end if;
         I:=I+1;
         exit when I > K;
      end loop;

      -- solution for C(J) found
      C(J) := Best_Sum;
      for I in 1..K loop
         Coins(J,I) := Coins(J-V(Best_I),I);
      end loop;
      Coins(J,Best_I) :=Coins(J,Best_I)+1;
   end loop;

end Tabulate;
```

In this code we have taken advantage of the inner loop on J to also tabulate an extra array nammed Coins, which represent the optimal number of coins needed to give change on each value of n.

The complexity is obviously $O(nk)$ as the outer loop runs over the different values of n, and the inner loop examines k different values of v_i each time.

4.6 Party

4.6.1 Statement

Your boss asks you to organize a party in which new colleagues can meet. So that the party be successful, you believe it reasonable not to invite someone if he/she knows more than $n - p$ persons out of the n, or fewer than p. However, you would like to invite as many people as possible.

1. Propose an algorithm to solve this problem for $p = 1$. Complexity? *Hints:*

   - Model the problem as a graph: each vertex is a person, each edge represent the fact two persons know each other

- A subgraph is obtained by removing a given vertex in a graph. Observe that any vertex can't be connected to more vertices in any subgraph than it is in the original graph.
- Show that knowing $n-1$ persons in the graph of knowledges is equivalent to not knowning 1 in the graph's complement.

2. Can you generalize your solution to $p \geq 2$?

4.6.2 Solution

Question 1

We represent the problem with an unoriented graph $G = (V, E)$: $v \in V$ represents person v, while $e = (u, v) \in E \subseteq V \times V$ encodes the fact that persons u and v know each other. Recall that the degree $d_V(x)$ of a vertex x is the number of edges it is involved in, or equivalently, the number of other vertices it is connected to.

For $p = 1$, we are looking for the largest possible subgraph $G' = (V', E')$ of G, satisfying the following two conditions:

C1 $\forall x \in V', \exists y \in V' : y \neq x \Rightarrow (x, y) \in E'$

C2 $\forall x \in V', \exists y \in V' : y \neq x \Rightarrow (x, y) \notin E'$

Without discussing the unicity of the solution (we just want one), we pretend that Alg. 4 computes a correct solution to the problem. Its principle is very simple : initially, we set $V' = V$, then we examine in turn each node x of V' and remove it if either $d(x) = 0$ or $d(x) = |V'| - 1$ hold, until stability. Why does this work? All relies, indeed, on this obvious result:

Lemma 1. *Let G be a graph, H a subgraph of G, and x a vertex of H. Then $d_H(x) \leq d_G(x)$.*

In other words, a vertex cannot be more connected in a subgraph than it is in a given graph.. which falls under common sense. A consequence is that if a node is disconnect in a graph, then so is it any subgraph as well.

Since $\exists y \in V' : y \neq x \Rightarrow (x, y) \in E'$ is equivalent to $d_{V'}(x) = 0$, and that $d_{V'}(x) = 0 \Rightarrow d_{V''}(x) = 0$ for any $V'' \subset V'$ in virtue of the lemma, we conclude that all nodes with degree 0 of G cannot belong to the solution because they don't satisfy C1, and this holds whatever C2 says or think of them.

V' is therefore at best equal to $V'' = \{x \in V : d(x) > 0\}$, and V'' is obviously the largest possible subset of V satisfying condition C1.

Let us now turn to C2. Consider V'' as computed before, and $x \in V''$. If x satisfies C2, then we are done. If it does not, it means it is connected to all other nodes of V''. Consider then $\overline{G''} = (V'', V'' \times V'' \setminus E'')$, the complementary graph of V'' in its set of edges. In this graph, x is connected to nothing. So $d_{\overline{G''}}(x) = 0$, and by virtue of the lemma, $d_{\overline{G'''}}(x) = 0$ holds too for any subgraph G''' of G''. This means there is no subgraph of G'' which can contain x and be a solution at the same time. Therefore, x cannot belong to the solution and has to be removed.

To conclude, observe that a vertex x which violates C1 can be removed at any time, without contradicting $d_{V''}(x) = 0$, and independently of V''. This justify that "submitting" V'' to C1 for removal, *then* to C2 for removal, is equivalent to checking both conditions, and possibly remove,node by node in a single loop.

Algorithm 4 The largest successful party for $p = 1$

```
V' := V, n := |V'|
continue := true
for x ∈ V' do
  if d(x) = 0 or d(x) = n − 1 then
    for y ∈ V' : (x, y) ∈ E' do
      Set d(y) := d(y) − 1
      E' := E' \ {(x, y)}
    end for
    n := n − 1, V' := V' \ {x}
  end if
end for
Invite V' to the party
```

If G is given in the form of a sorted planar map[2] then we can iterate on the neighbours of any node in logiarthmic time, so line $E' := E' \setminus \{(x, y)\}$ takes $O(\log n)$ time at worst, which occurs when G is a clique. This line would be executed $O(n)$ times for the loop on y's, which in turn will be executed $O(n)$ times by the loop on x's. Altogether, the the algorithm is therefore $O(|V^2| \log |V|)$. In the best case (the graph is a line), there is no removal at all, so $\Omega(|V|)$.

Question 2

The above proof can be easily extended when $p \geq 2$:

- For condition C1, if $d_G(x) < p$, then $d_H(x) < p$ too for any subgraph H of G; so x has to be removed
- For condition C2, if $d_{\bar{G}}(x) < p$, then $d_{\bar{H}}(x) < p$ too for any subgraph H of G; so x has to be removed too.

The **if** condition $d(x) = 0$ or $d(x) = n - 1$ simply has to be replaced by $d(x) < p$ or $d(x) \geq n - p$. The complexities remain unchanged.

5 NP-completeness

5.1 ZOE

5.1.1 Statement

Consider ZOE (zero-one equations): given an $m \times n$ matrix $\mathbf{A}$ filled with 0's or 1's, is there an n-vector $\mathbf{x}$ filled of 0's or 1's only such that $\mathbf{Ax} = \mathbf{1}$?

1. Show that ZOE is NP-complete.
2. Infer that ILP (integer linear programming) is NP-complete too.

5.1.2 Solution

Question 1) ZOE $\in \mathcal{P}$: the certificate is just the solution itself, and it takes $O(mn)$ binary multiplications $+O(n)$ comparisons to check the validity of any solution.

[2]To each vertex is attached the set of pointers to edges the vertex is involved in, which is furthermore assumed as sorted

ZOE is NP-hard : we shall show this by reducing ZOE to n-SAT. Let us first rewrite $\mathbf{Ax} = \mathbf{1}$ this way :

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \tag{14}$$

Consider, for the moment, the i-th row of this linear system of equations alone:

$$\sum_{j=1}^{n} a_{ij} x_j = 1 \tag{15}$$

From now on, we assume that i is fixed and focus only on line i. Let S be the set of indices j for which $a_{ij} = 1$, and put, for convenience, $y_u = x_{iS_u}$ for all $u = 1, \dots, |S|$. Equation (15) sum can be rewritten as

$$\sum_{u=1}^{|S|} y_u = 1 \tag{16}$$

Because the y_u's can only be 0 or 1 (just as the x_i's), we can think of this last inequality in terms of logics: with a slight abuse of notation, it merely says that only one y_u variable must be true, while all others need be false. If $|S_i| = k$, this is general k-XOR condition :

$$\begin{aligned} F(y_1, \dots, y_u) = \quad & (y_1 \wedge \neg y_2 \wedge ... \wedge \neg y_k) \vee \\ & (\neg y_1 \wedge y_2 \wedge \neg y_3 \wedge ... \wedge \neg y_k) \vee \\ & (\neg y_1 \wedge ... \wedge \neg y_{k-1} \wedge y_k) \end{aligned}$$

which is in disjunctive normal form (DNF). To recast the problem as a $k-SAT$ problem, we need to convert $F(y_1, \dots, y_u)$ into a CNF; this unfortunately results in a conjunction with 2^k clauses before simplification (as shown in class), or an $O(2^n)$ in worst case, and our reduction to k-SAT would this be invalid (remind, our 1-to-1 transformation needs be polynomial in time).

There is a more straightforward way to rewrite F using a polynomial number of clauses only: suffice, indeed, to see that $F(y_1, \dots, y_u)$ is true if and only if i) one of the y_u's is true, and ii) all *pairs* (y_a, y_b) are false whenever $a \neq b$. If all pairs are false, then so will be all triplets; and so will be all quadruplets; and so on. It is therefore easily seen that

$$\begin{aligned} F(y_1, \dots, y_u) = \quad & (y_1 \vee ... \vee y_k) \\ & \wedge \neg(y_1 \wedge y_2) \wedge \neg(y_1 \wedge y_3) \wedge ... \wedge \neg(y_1 \wedge y_k) \\ & \wedge \neg(y_2 \wedge y_3) \wedge \neg(y_2 \wedge y_4) \wedge ... \wedge \neg(y_2 \wedge y_k) \\ & \wedge ... \\ & \wedge \neg(y_{k-1} \wedge y_k) \end{aligned}$$

At last, we apply De Morgan's law on all clauses $\neg(y_u \wedge y_v)$ to get $\neg(y_u \wedge y_v) = \neg y_u \vee \neg y_v$, then

$$\begin{aligned} F(y_1, \dots, y_u) = \quad & (y_1 \vee ... \vee y_k) \wedge \\ & (\neg y_1 \vee \neg y_2) \wedge (\neg y_1 \vee \neg y_3) \wedge ... \wedge (\neg y_1 \vee \neg y_k) \\ & (\neg y_1 \vee \neg y_2) \wedge (\neg y_1 \vee \neg y_3) \wedge ... \wedge (\neg y_1 \vee \neg y_k) \\ & \wedge ... \\ & \wedge (\neg y_{k-1} \wedge \neg y_k) \end{aligned}$$

which is the conjunction of a k-CNF with $O(k^2) = O(n^2)$ additional 2-CNFs, so overall, a k-CNF whenever $k \geq 2$. Since we have m independent lines to satify, the resulting CNF will be at worst an n-CNF with $O(mn^2)$ clauses, a polynomial number of both m and n.

We have thus found a procedure to *rewrite* any instance of ZOE as an instance of n-SAT in polynomial time, and our transformation works forwards, but also backward. Therefore, ZOE $\geq_p$ n-SAT, and since n-SAT is NP-hard, so must be ZOE. ■

Question 2) Integer linear programming (ILP) is a problem of the form

$$\begin{array}{ll} \text{maximize} & \mathbf{b}^t\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \leq \mathbf{c} \end{array}$$

where $\mathbf{A}$ is a matrix with integer coefficients, $\mathbf{x}$, $\mathbf{b}$, and $\mathbf{c}$ are vectors with integer coefficients, and t denotes transposition. It is an optimization problem. Its corresponding decision version amounts to determining whether the objective function can be larger than some constant k. In other words: does the system of inequalities

$$\begin{cases} \mathbf{b}^t\mathbf{x} \geq k \\ \mathbf{A}\mathbf{x} \leq \mathbf{c} \end{cases} \tag{17}$$

admits a solution in $\mathbf{x}$ or not? The latter problem is clearly in $\mathcal{NP}$: the certificate is just the solution $\mathbf{x}$ itself, and if A is an $m \times n$ matrix, it will take $O(mn)$ multiplications and addition to check the correctness of the solution. To show that ILP is NP-hard, consider ILP problems of the form :

$$\begin{cases} (1 \dots 1)^t\mathbf{x} \geq 0 \\ \mathbf{A}\mathbf{x} \leq \mathbf{1} \\ -\mathbf{A}\mathbf{x} \leq -\mathbf{1} \\ \mathbf{0} \leq \mathbf{x} \\ \mathbf{x} \leq \mathbf{1} \end{cases} \tag{18}$$

The last two lines are equivalent to

$$\mathbf{A}\mathbf{x} = \mathbf{1} \tag{19}$$

and the former two constraint $\mathbf{x}$ to have components in $\{0, 1\}$. That is, a general ZOE problem, whose solution leaves the first line of (18) always s In other words, any ILP problem of the form (18) can be rewritten as one of the form (19), and conversely. Therefore, there is at least a subset of ILP problems for which $ZOE \leq_P ILP$ holds true. But ZOE is NP-hard, so ILP can be claimed to be NP-hard too.

5.2 Bonnie and Clyde

5.2.1 Statement

Bonnie and Clyde have just robbed a bank. They have a loot consisting of n dollars, and they would like to split it into 2 parts of equal size. State whether splitting the loot is an NP-complete problem or not in each of the following cases:

- they have coins of x and y dollars
- they have coins of 1,2,4,8,... 2^i dollars
- they have bearer bonds of arbitrary value
- same, but don't mind if the difference between the 2 parts is less that 100 dollars

5.2.2 Solution

We first notice that for the 3 first questions, the problem is not NP-hard whenever $n \neq 2p$, simply because the answer to the decision problem in such cases is negative. We shall therefore assume $n = 2p$ in the sequel, and focus on that case. Clearly, it takes a polynomial number of multiplications and additions to check whether a proposal will sum to p or not, or is in $[p - 100 : p + 100]$ (last question). Thus in all cases, the solutions can be checked in polynomial time, therefore the algorithms are in $\mathcal{NP}$, and it remains to examine their NP-hardness.

1) They have a coins of x dollars, and b of y dollars. We assume $n = a + b = 2p$. Suppose Bonnie receives u coins of value x and v of value y. Then the question amounts to determining whether

$$x(a - u) + y(b - v) = p$$

has a solution in $0 \leq u \leq a$, $0 \leq v \leq b$ or not. This can be answered in $O(ab)$ time by testing all possible pairs. So the problem is not NP-hard, therefore not NP-complete, in this case.

2) They have coins of 1,2,4,8,... 2^i dollars. The problem is not NP-hard. Still assuming $n = 2p$, we can write the loot and p as

$$n = \sum_{j=0}^{i} \alpha_j 2^j, \qquad p = \sum_{j=0}^{i} \beta_j 2^j$$

where the β_i's are the coefficients of the binary expansion of p. Since we want $n = 2p$, and the binary decompositions of p and n are unique, our problem is therefore to determine whether these two decompositions can be considered as equivalent by possibly relocating exceeding coins of value 2^i to coins of larger values $2^j, j > i$, if this situation ever happens. We claim that the following (greedy) algorithm precisely achieves this:

- Init: pack every coin in a separate bag (see why further in the text).
- For j=0,...,i do
  - If $\alpha_j < 2\beta_j$ or $\alpha_j - 2\beta_j$ is even, then claim the loot is not separable, and stop.
  - If $\beta_j > 0$, then give 1 bag of value 2^j to Bonnie, and another one to Clyde.
  - Pack the $\alpha_j - 2\beta_j$ remaining bags with value 2^j, *by pairs*, into new bags. Put these new bags along with those of value 2^{j+1} and update $\alpha_{j+1} \leftarrow \alpha_{j+1} + \alpha_j/2 - \beta_j$
- Claim that the loot has been separated.

Why works? The binary decomposition of p is unique. At iteration 0, if p is even, then its lowest bit has to be 1, so $\beta_0 = 1$. This tells us that Bonnie *must* receive a coin of value 1, just as Clyde, for otherwise the split would be unfair. Two cases may then arise:

- It is not feasible because $\alpha_0 < 2$ (we have no, or just one coin of value 1). Then the problem has no solution : *all* other coins have a value ≥ 2, therefore any combination of them will be a multiple of 2, but never 1.
- On the contrary, if $\alpha_0 \geq 2$, then we can give a coin of value 1 to Bonnie, and another to Clyde. But we might be left with a new problem: what to do of the remaining coins, if any? As we said earlier, *all* the remaining bags now have a value of at least 2. This implies two things:

1. We don't need to care of coins of value 1 any longer: the "unit" is now 2, not 1. So we group pairs of coins of value 1 into bags of value 2, and put these new bags alongst with those of value 2 : this is a new problem, but strictly identical in nature to the former one.
2. If we have an even number of remaining coins, again, we find ourselves with 1 coin of value 1 and we are stuck: it cannot be part of the solution.. so the problem, again, has no solution.

If $\beta_0 = 0$, then there are no coin to distribute, but the problem remains the same (what to do of the remaining coins of value 1, if any?), and our tests involving $\alpha_j - 2\beta_j$ still work.

We then start a new iteration, and face exacly the same problem with packs of value 4/2 instead of 2/1.. a.s.o, til the end. In short, the correctness of this algorithm relies on the fact that the question of the "precision" of the split, at any iteration j, must be answered immediately, and does not depend on subsequent arrangements of the loot.

The algorithm is clearly polynomial in time : i times operations involving at worst substractions.

3) They have bearer bonds of arbitrary value. Let $v_1, ..., v_n$ be the values of these bonds. The loot is separable if and only if we can find $x_1, ..., x_n \in \{0, 1\}$, such that

$$\sum_{i=1}^{n} x_i v_i = p \tag{20}$$

Assume that all v_i's can be coded using at most k bits. Call $\mathbf{v}_1, ..., \mathbf{v}_n \in \mathbb{Z}^k$ the colon vectors whose components are the binary decompositions of $v_1, ..., v_n$, and $\mathbf{p}$ that of p. Then Eq. (20) is immediately rewritten as

$$\begin{pmatrix} \mathbf{v_1} & \mathbf{v_2} & ... & \mathbf{v_n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix} = \begin{pmatrix} p_1 \\ \vdots \\ p_k \end{pmatrix} \tag{21}$$

This is almost a $k \times n$ ZOE problem, except that there might be indices i for which $p_i = 0$, a violation of the statement of ZOE. To correct that, let us consider one such i. The corresponding row implies that all variables x_j for which $A_{ij} \neq 0$ be set to 0: for otherwise, the equality would just not hold. Call S_j the set of indices of these variables. Then, the set of all variables which need to be set to 0 is nothing but

$$V := \bigcup_{j:p_j=0} S_j$$

a set whose size is arbitrary, and does only depend on $\mathbf{A}$ and $\mathbf{p}$. Thus, solving (21) amounts to solving a true $(k - |V|) \times n$ ZOE problem, obtained by taking (21), and droping all rows and variables for which $p_i = 0$. Because $|V|$ is independent of k (rows of $\mathbf{A}$ for which $p_i \neq 0$ can still be chosen arbitrarilly), there exist instances of the Bonnie and Clyde problem (BC) such that $ZOE \leq_p BC$. But ZOE is NP-hard, so BC has to be NP-hard too.

4) Same, but don't mind if the difference between the 2 parts is less that 100 dollars. This does not change the complexity of the problem, which is still NP-hard. The question addressed now is: does

$$\mathbf{Ax} = \mathbf{z_0}$$

admit a solution $\mathbf{x}$ in $\{0, 1\}^k$ for any $\mathbf{z_0}$ being the binary representation of $p-100$, $p-99$, ..., $p+100$? If the problem were not NP-hard, that is, in $\mathcal{P}$, that would mean *all* of its instances

could be solvable in polynomial time. However, for arbitary p large enough, all of them are known to be NP-hard, as instances of problems of the question 3. A contradiction. So the problem remains NP-hard.

5.3 Hitting set

5.3.1 Statement

Consider the HITTING SET problem: given n sets of elements $S_1, \ldots, S_n$, and a contant $k > 0$, find a set S such that $|S| \leq k$ and $\forall i = 1, \ldots, n : S \cap S_i \neq \emptyset$. Is HITTING SET NP-hard or not?

5.3.2 Solution

NOTE : the related optimization version is : given n set of values $S_1, \ldots, S_n$, find the smallest possible set S such that $\forall i = 1, \ldots, n : S \cap S_i \neq \emptyset$.

Let us show that this is nothing but a disguised ILP problem. Consider the union of all subsets. Say it is made of p values $v_1, \ldots, v_p$. Now define $x_1, ..., x_p$ as

$$x_i = \begin{cases} 1 & \text{if } x_i \text{ should be in } S \\ 0 & \text{if if should not} \end{cases}$$

for any i. Likewise, define $\mathbf{A} = (a_{ij})$ to be a $p \times p$ 0-1 matrix such that

$$a_{ij} = \begin{cases} 1 & \text{if } x_j \in S_i \\ 0 & \text{otherwise} \end{cases}$$

We are looking for an integer vector $\mathbf{x} = (x_1, ..., x_p)^t$ satisfying

$$\begin{cases} (1, ..., 1)\mathbf{x} \leq k \\ \mathbf{A}\mathbf{x} \geq (1, ..., 1)^t \end{cases}$$

This system of equations is identical to that of (17), except that sign must be changed in both lines[3]. Any instance of HITTING SET is therefore equivalent to an ILP problem. But ILP is NP-hard, and so must be HITTING SET. ■

[3]But we made no assumptions regarding the sign of k, $\mathbf{b}$, or $\mathbf{c}$ for ILP