Theoretical Computer Science 454 (2012) 72–80

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

The complexity of Euler's integer partition theorem

Cristian S. Calude [a,*], Elena Calude [b], Melissa S. Queen [c]

[a] Department of Computer Science, The University of Auckland, New Zealand

[b] Institute of Information and Mathematical Sciences, Massey University at Auckland, New Zealand

[c] Computer Science Department, Dartmouth College, NH, USA

ARTICLE INFO

Keywords:
Complexity of mathematical problems
Euler's integer partition theorem

ABSTRACT

Euler's integer partition theorem, which states that *the number of partitions of an integer into odd integers is equal to the number of partitions into distinct integers*, ranks 16 in Wells' list of the most beautiful theorems (Wells, 1990) [15]. In this paper, we use the algorithmic method to evaluate the complexity of mathematical statements developed in Calude et al. (2006) [5] and Calude and Calude (2009, 2010) [6,7] and to show that Euler's theorem is in class $\mathfrak{C}_{U,3}$, the same complexity class as the Riemann hypothesis.

1. Euler's integer partition theorem

The number of ways of writing an integer as a sum of n positive integers, where the order of addends is ignored, is denoted by $P(n)$. By $P(n|\text{odd parts})$ and $P(n|\text{distinct parts})$ we denote the number of ways of writing the integer as a sum of n odd/distinct positive integers. By convention, partitions are usually ordered from largest to smallest. Some examples are presented in Table 1.

Leonhard Euler is credited (see [1] p. 2) to have proved in 1748 the theorem bearing his name: no matter how long we extend Table 1, there will always be as many items in the left column as in the right one. In other terms, *the number of partitions of an integer into odd integers is equal to the number of partitions into distinct integers*.

Euler's integer partition theorem is a Π_1-statement, i.e. a statement of the form $\forall n\, P(n)$, where $P(n)$ is a unary computable predicate; hence its complexity can be evaluated with the coarse-grained method developed in [5–7] and outlined below.

2. The complexity measure

The complexity measure for Π_1-statements used in this paper[1] is defined by means of register machine programs which implement a universal self-delimiting Turing machine U. The machine U (which is fully described in [7]) has to be *minimal* in the sense that none of its instructions can be simulated by a program for U written with the remaining instructions.

To every Π_1-problem $\pi = \forall m P(m)$ we associate the algorithm $\Pi_P = \inf\{n : P(n) = \text{false}\}$ which systematically searches for a counterexample for π using the predicate P. There are many programs (for U) which implement Π_P; without loss of generality, any such program will also be denoted by Π_P. Note that π is true iff $U(\Pi_P)$ never halts.

* Corresponding author. Tel.: +64 21 2411 454.
E-mail addresses: cristian@cs.auckland.ac.nz (C.S. Calude), Melissa.S.Queen.13@dartmouth.edu (M.S. Queen).
URLs: http://www.cs.auckland.ac.nz/~cristian (C.S. Calude), http://www.massey.ac.nz/~ecalude (E. Calude).

[1] We are not aware of any other complexity measure for Π_1-statements.

0304-3975/$ – see front matter
doi:10.1016/j.tcs.2012.03.023

Table 1
Integer partitions into odd integers versus integer partitions into distinct integers.

n	Odd parts	$P(n\|\text{odd parts})$	Distinct parts	$P(n\|\text{distinct parts})$
1	1	1	1	1
2	$1+1$	1	2	1
3	$1+1+1$		3	
	3	2	$2+1$	2
4	$1+1+1+1+1$		4	
	$3+1$	2	$3+1$	2
5	$1+1+1+1+1$		5	
	$3+1+1$		$4+1$	
	5	3	$3+2$	3
6	$1+1+1+1+1+1$		6	
	$3+1+1+1$		$5+1$	
	$3+3$		$4+2$	
	$5+1$	4	$3+2+1$	4

Motivated by Occam's Razor principle of parsimony, see [2], we define the complexity (with respect to U) of a Π_1-problem π to be the length of the shortest program Π_P – defined as above – where minimisation is calculated for all possible representations of π, where $\pi = \forall n P(n)$[2]:

$$C_U(\pi) = \min\{|\Pi_P| : \pi = \forall n P(n)\}.$$

Because the complexity C_U is incomputable,[3] we work with upper bounds for C_U. As the exact value of C_U is not important, following [7], we classify Π_1-problems into the following classes:

$$\mathfrak{C}_{U,n} = \{\pi : \pi \text{ is a } \Pi_1\text{-problem}, C_U(\pi) \leq n \text{ kbit}\}.$$

This method has been applied to a variety of problems, including Fermat's last theorem, the Goldbach conjecture, the four-colour problem, the Riemann hypothesis, and the Hilbert 10th problem. The complexity method we use does not always match the "intuitive complexity" of those problems well; a typical example is the low complexity of Fermat's last theorem. One reason is that our complexity takes into consideration only one way – a brute-force search for a counterexample – from infinitely many ways to solve the problem.

3. A universal prefix-free binary Turing machine

Here, we briefly describe the syntax and the semantics of a register machine language which implements a minimal universal prefix-free binary Turing machine U; it is a refinement, constructed in [7], of the language in [5]; see also [10]. In principle, any universal (Turing-complete) language can be used here. However, the language has to be in some sense "natural". For example, no specific problem should be coded in an unreasonable simple way; also, the language syntax should not 'force' programs to be artificially long.

To be able to have meaningful comparison between mathematical problems, one has to use the same language or have a fine-tuned simulation between languages used for the evaluation of different problems. To make the paper as self-contained as possible, we succinctly present the adopted language.

Any register program (machine) uses a finite number of registers, each of which may contain an arbitrarily large non-negative integer. By default, all registers, named with a string of lower-case or upper-case letters, are initialised to 0. Instructions are labelled sequentially, beginning with 1. The register machine instructions are listed below. Note that in all cases R2 and R3 denote either a register or a non-negative integer, while R1 must be a register.

=R1, R2, R3

If the content of R1 and R2 being equal, then the execution continues at the R3-th instruction of the program. If the contents of R1 and R2 are not equal, then execution continues with the next instruction in sequence. If the content of R3 is outside the scope of the program, then we have an illegal branch error.

2 For C_U it is irrelevant whether π is known to be true or false. In particular, the program containing the single instruction halt is not a Π_P program, for any P.

3 This follows from the undecidability of the halting problem; see [4].

Table 2
Special characters.

Special characters	Code	Special characters	Code
,	ε	+	111
&	01	!	110
=	00	%	100

&R1, R2

The content of register R1 is replaced by R2.

+R1, R2

The content of register R1 is replaced by the sum of the contents of R1 and R2.

!R1

One bit is read into the register R1, so the content of R1 becomes either 0 or 1. Any attempt to read past the last data bit results in a run-time error.

%

This is the last instruction for each register machine program before the input data. It halts the execution in two possible states: either it halts successfully or it halts with an under-read error.

A *register machine program* consists of a finite list of labelled instructions from the above list, with the restriction that the halt instruction appears only once, as the last instruction of the list. The input data (a binary string) follows immediately after the halt instruction. A program not reading the whole data or attempting to read past the last data bit results in a run-time error. Some programs (as the ones presented in this paper) have no input data; these programs cannot halt with an under-read error.

The instruction =R,R,n is used for the unconditional jump to the nth instruction of the program.

For longer programs, it is convenient to distinguish between the main program and some sets of instructions called "routines" which perform specific tasks for another routine or the main program. The call and call-back of a routine are executed with unconditional jumps.

4. Binary coding of programs

In this section, we develop a systematic efficient method to uniquely code the register machine programs in binary. To this aim, we use a prefix-free coding as follows.

The binary coding of special characters (instructions and comma) is given in Table 2 (ε is the empty string).

For registers, we use the prefix-free code $\text{code}_1 = \{0^{|x|}1x \mid x \in \{0,1\}^*\}$. The register codes are chosen to optimise the length of the program, i.e. the most frequent registers have the smallest code_1 length. For non-negative integers, we use the prefix-free code $\text{code}_2 = \{1^{|x|}0x \mid x \in \{0,1\}^*\}$. The instructions are coded by self-delimiting binary strings as follows.

(1) & R1,R2 is coded in two different ways depending on R2[4]:

$$01\text{code}_1(\text{R1})\text{code}_i(\text{R2}),$$

where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

(2) + R1,R2 is coded in two different ways depending on R2:

$$111\text{code}_1(\text{R1})\text{code}_i(\text{R2}),$$

where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

(3) = R1,R2,R3 is coded in four different ways depending on the data types of R2 and R3:

$$00\text{code}_1(\text{R1})\text{code}_i(\text{R2})\text{code}_j(\text{R3}),$$

where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer, $j = 1$ if R3 is a register and $j = 2$ if R3 is an integer.

(4) !R1 is coded by

$$110\text{code}_1(\text{R1}).$$

(5) % is coded by

$$100.$$

[4] As $x\varepsilon = \varepsilon x = x$, for every string $x \in \{0,1\}^*$, in what follows we omit ε.

All codings for instruction names, special symbol comma, registers, and non-negative integers are self-delimiting; the prefix-free codes used for registers and non-negative integers are disjoint. The code of any instruction is the concatenation of the codes of the instruction name and the codes (in order) of its components; hence, the set of codes of instructions is prefix free. The code of a program is the concatenation of the codes of its instructions, so the set of codes of all programs is prefix free too.

5. The counting algorithm

We use a slightly modified form of Algorithm 7 in [13, p. 13], which generates all integer partitions: for each of them we test whether the partition uses odd or distinct integers, and count accordingly. We present the algorithm in the following commented pseudo-code format; labels L0, L2, L6, L13, L18, L23 correspond to the program in Table 9.

```
L0    for N in (2, 3, ...) do
        P ← 0              //tally of odd partitions
        R ← 0              //tally of distinct partitions

        //generate the starting array A
        A[N] ← N
        for I in (1 ... N − 1) do
          A[I] ← 0
        end for

        //check if the array A is ordered
L2      a ← 0              //the previous element
        for I in (1 ... N) do
          if a > A[I] then
            goto L6        //not ordered
          end if
          a ← A[I]
        end for
        goto L13           //array was ordered

        //generate the next partition
L6      V ← 0              //counts the leading ones
        for I in (1 ... N) do
          if A[I] = 0 then
            continue
          else if A[I] = 1 then
            A[I] ← 0
            V + +
            continue
          else if A[I] > 1 then
            A[I] ← A[I] − 1
            A[I − 1] ← V + 1
            goto L2  //next partition has been made, is it ordered?
          end if
        end for
        goto L23       //all elements are 1, we are at the last partition

        //check if the partition has all odd elements
L13     for I in (1 ... N) do
          if A[I] = 0 then
            continue
          else if A[I] mod 2 = 0 then
            goto L18  //found a non-zero even element
          end if
        end for
        P + +          //all elements were checked, they were all odd
```

```
          //check if the partition has all distinct elements
L18     a ← 0      //previous element
        for I in (1 .. N) do
          if A[I] = 0 then
            continue
          else if A[I] = a then
            goto L6     //found two identical non-zero elements
          end if
          a ← A[I]
        end for
        R++           //all elements were checked and distinct
        goto L6
L23     if P = R then
          continue
        else
          HALT
        end if
      end for
```

Other possible algorithms for generating all integer partitions are discussed in [11,12,14]. They may be used in an attempt to improve the complexity bound obtained in this paper.

6. Routines

There are two types of routine: (a) 1-routines, that is, routines that do not use any other routines, and (b) 2-routines, that is, routines that call other routines. In general, unary 1-routines or 2-routines use the register a for input, b for keeping track of returning to the calling environment, and c for storing the result. Binary 1-routines or 2-routines use a and b for input, c for the return address, and d for the result.

As the registers are shared between the main program and routines, care must be taken so that the content of a register is not changed inadvertently. There are various ways to deal with this problem. One is to reserve the letters from a to h to 1-routines and to use aa, ab, ac, ...in 2-routines or the main program (see [10]). The approach used in the following examples is that 1-routines use single-letter names, 2-routines use double-letter names, where the second letter comes from the first letter of the routine, and the main program uses capital letters for registers. As all 2-routines have different first-letter names, there is no danger of using the same register in a routine that calls another routine using the same register name. The upside of this approach is the guarantee that the values in registers are the correct ones; the downside is that the number of necessary registers tends to increase (this fact can be mitigated at the end by safely reusing a few registers).

We present three routines used in the program: CMP, SUBT1, and DIV2. All routines are presented in a "human-readable" register machine code, along with comments and the corresponding binary code (using the encoding described in Section 4). Note that in the binary representation the instruction labels have been replaced with their actual line numbers, and those numbers may change depending on where the routine is in the larger program.

The compare routine CMP takes as input two non-negative integers, stored in registers a and b, and produces its result in register d according to the formula $d = \mathrm{CMP}(a, b) = 1$ if $a < b$, 0 if $a = b$, and 2 if $a > b$. It then returns to c.

The subtraction routine SUBT1 takes an integer stored in register a with $\mathtt{a} \geq 1$ as input, and produces the result a−1 in register d. It then returns to c.

The routine DIV2 takes as input a single value, stored in register a. It computes $\lfloor \frac{\mathtt{a}}{2} \rfloor$ and stores this back into a ($\mathtt{a} = \mathtt{a}/2$). It also computes a mod 2, and stores the result in c. Finally, it returns to the address stored in register b.

Concatenating the (prefix-free) binary codes of instructions inserted in the last column of Table 5, we get the binary string

01001000100101010001001011000101101010001101110101100100011
00100101101100011101101011001000011010001010110101001011011

which uniquely codes the routine DIV2. It has 119 bits.

7. Register machine language implementation of arrays

We use the coding for the array data structure library developed by Dinneen [10], which represents arrays (lists) using a single register variable. An integer element a_i within an array A is represented as a sequence of a_i bits 0; the bit 1 is used as a (leading) separator or deliminator of the array elements. If there are no 1s (e.g. the register has value 0) then we have an array of size 0.

Table 3
Routine CMP.

Label	Instruction	Comments	Binary representation
CMP	& d, 0		01 00111 100
	= a, b, c	// a==b	101 010 00110 00101
	& e, 0		01 011 100
LCP1	& d, 1		01 00111 101
	= a, e, c	// a<b	101 010 011 00101
	& d, 2		01 00111 11010
	= b, e, c	// a>b	101 00110 011 101
	+ e, 1		100 011 101
	= a, a, LCP1		101 010 010 1110100

Table 4
Routine SUBT1.

Label	Instruction	Comments	Binary representation
SUBT1	& d, 0		01 00111 100
LS1	& e, d		01 011 00111
	+ e, 1		100 011 101
	= e, a, c	// d+1=a	101 011 010 00101
	& d, e		01 00111 011
	= a, a, LS1		101 010 010 11010

Table 5
Routine DIV2.

Label	Instruction	Comments	Binary representation
DIV2	& f, a		01 00100 010
	& a, 0		01 010 100
LD1	& c, 0	// c=0 when a is even	01 00101 100
	& e, a		01 011 010
	+ e, e	// calculate 2a	100 011 011
	= e, f, b	// 2a==f, so a is halved	101 011 00100 00110
	& c, 1	// c=1 when a is odd	01 00101 101
	+ e, 1		100 011 101
	= e, f, b	// 2a+1==f, so a is halved	101 011 00100 00110
	+ a, 1	// otherwise, a++ and	100 010 101
	= a, a, LD1	// continue looping	101 010 010 11011

In this representation, there is the freedom to interpret the array as starting either at the left (most significant bits) or the right (least significant bits) of the string. While both interpretations are equally effective, it is much easier to add bits to the right of a string (double the number to add a 0, or double and add 1 to add a 1), and thus a natural choice often arises. Because it was important for this implementation to be easily able to add elements to the beginning of an array, we decided to use a right-to-left interpretation. For example, the array $A = [a_1, a_2, a_3, a_4] = [6, 1, 0, 4]$ is represented in binary as 100001101000000 or in decimal as 17216.

Next we present three routines dealing with arrays: PREPEND, ELM and RPL. They are used in the main program presented in Table 9.

The 1-routine PREPEND, presented in Table 6, adds `b` to the start of array `a`, then returns to `c`. It alters `a` directly. Using PREPEND to add 2 to array [6, 1, 0, 4] produces [2, 6, 1, 0, 4] coded by the string 100001101000000100.

The use of generic array `a`, rather than the specific array `A` used by the main program, is necessary because this routine is used in RPL.

The 2-routine ELM (Table 7) takes as input a number in register `I` and an array in register `A`. It extracts the `I`th element from the array `A`, and as output stores it into register `d` (`d = A[I]`). It returns to the line number stored in register `c`.

The 2-routine RPL (Table 8) takes as input a number stored in register `I` and an array stored in register `A` (the same input as for routine ELM). It replaces the `I`th element with value `b` (`A[I] = b`), and has no formal output other than the changed array (it alters `A` directly). Upon completion, it returns to the line number stored in register `c`.

Table 6
Routine PREPEND.

Label	Instruction	Comments
PREPEND	`& e, 0`	`// loop counter`
	`+ a, a`	
	`+ a, 1`	`// add 1 as element separator`
LP1	`= e, b, c`	`// loop until we have added b zeros`
	`+ a, a`	
	`+ e, 1`	
	`= a, a, LP1`	

Table 7
Routine ELM.

Label	Instruction	Comments
ELM	`& ae, a`	
	`& be, b`	
	`& ce, c`	
	`& d, 1`	`// counter`
	`& a, A`	`// make a copy of the array`
LE3	`= I, d, LE6`	`// if we have counted enough 1s, goto L6`
	`& b, LE4`	
	`= a, a, DIV2`	`// otherwise, halve a`
LE4	`+ d, c`	`// add last bit to our counter`
	`= a, a, LE3`	
LE6	`& d, 0`	`// reset the counter; now we are counting 0s`
LE7	`& b, LE8`	
	`= a, a, DIV2`	
LE8	`= c, 1, LE10`	`// if we got to a 1, then exit`
	`+ d, 1`	`// otherwise increment the counter`
	`= a, a, LE7`	
LE10	`& a, ae`	
	`& b, be`	
	`& c, ce`	
	`= a, a, c`	`// restore and return`

8. The program $\Pi_{\text{IntegerPartition}}$

We are now ready to present (see Table 9) the main program $\Pi_{\text{IntegerPartition}}$, which is based on the counting algorithm presented in Section 5 and uses all routines described in Section 6 (CMP, SUBT1, DIV2), Tables 3–5, and Section 7 (PREPEND, ELM and RPL), Tables 6–8.

The register machine program for Euler's integer partition theorem – which is obtained from the main program in Table 9 in which all symbolic names of routines are replaced by their respective codes – consists of 158 instructions having 2396 bits; hence it is in $\mathfrak{C}_{U,3}$. In this way, Euler's integer partition theorem is in the same complexity class as the Riemann hypothesis [9] (although the Riemann hypothesis seems slightly more complex, as it contains 178 instructions and 2745 bits; see [9]) and less complex than the four-colour theorem, which is in $\mathfrak{C}_{U,4}$ [8].

9. Final comments

The main source of the complexity of Euler's integer partition theorem comes from the necessity to work with arrays. Initially we had used Cantor's bijection for coding arrays; this resulted in the theorem being in the complexity class $\mathfrak{C}_{U,7}$, a too crude and unintuitive result. Using Dinneen's coding for arrays [10] – which we learned about after finishing a first draft of the paper – and further optimisations, we succeeded in decreasing the complexity to class $\mathfrak{C}_{U,3}$.

With more work one could probably decrease the complexity of Euler's integer partition theorem to $\mathfrak{C}_{U,2}$, but probably not to $\mathfrak{C}_{U,1}$.

Occam's Razor principle motivates the complexity of Π_1-statements used in this paper. The same principle leads, under some general assumptions, to a learning algorithm which produces hypotheses that with high probability will be predictive of future observations [3]. Is there any relation between the complexity of Π_1-statement and its learnability?

To conclude, we cite a question posed by a referee: How does the complexity of Euler's integer partition theorem established in this paper compare with the complexity of related integer partition theorems, e.g. Rogers–Ramanujan identities?

Table 8
Routine RPL.

Label	Instruction	Comments
RPL	& ar, a	
	& br, b	
	& cr, c	
	& ir, I	
	& I, N	// counter
	& fr, 0	// new array
LR1	= I, 0, LR7	// have reached end of the array
	= I, ir, LR5	// have reached the element to replace
	& c, LR2	
	= a, a, ELM	
LR2	& b, d	// b = A[i]
	& a, fr	
	& c, LR3	
	= a, a, PREPEND	// adds element onto our new array
LR3	& fr, a	
	& a, I	
	& c, LR4	
	= a, a, SUBT1	
LR4	& I, d	// I--
	= a, a, LR1	// loop
LR5	& b, br	// insert the new element
	& a, fr	
	& c, LR3	
	= a, a, PREPEND	// loop back
LR7	& a, ar	
	& b, br	
	& c, cr	
	& A, fr	
	& I, ir	
	= a, a, c	

Table 9
Main program $\Pi_{\text{IntegerPartition}}$.

Label	Instruction	Label	Instruction	Label	Instruction
MAIN	& N, 1		+ I, 1	L13	& I, 0
L0	+ N, 1		= a, a, L3	L14	= I, N, L17
	& P, 0	L22	+ R, 1		+ I, 1
	& R, 0	L6	& V, 0		& c, L15
	& I, 1		& I, 0		= a, a, ELM
	& a, 0	L7	= I, N, L23	L15	= d, 0, L14
	& b, N		+ I, 1		& a, d
	& c, L1		& c, L8		& b, L16
	= a, a, PREPEND		= a, a, ELM		= a, a, DIV2
L1	= I, N, L2	L8	= d, 0, L7	L16	= c, 0, L18
	+ I, 1		& a, d		= c, 1, L14
	& b, 0		& c, L9	L17	+ P, 1
	& c, L1		= a, a, SUBT1	L18	& I, 0
	= a, a, PREPEND	L9	& b, d		& a, 0
L2	& A, a		& c, L10	L19	= I, N, L22
L24	& I, 1		= a, a, RPL		+ I, 1
	& a, 0	L10	+ V, 1		& c, L20
L3	= I, N, L13		= d, 0, L7		= a, a, ELM
	& c, L4		& a, I	L20	= d, 0, L19
	= a, a, ELM		& c, L11		= a, d, L6
L4	& b, d		= a, a, SUBT1		& a, d
	& c, L5	L11	& I, d		= a, a, L19
	= a, a, CMP		& b, V	L23	= P, R, L0
L5	= d, 2, L6		& c, L24		%
	& a, b		= a, a, RPL		

Acknowledgement

We thank the referees for very useful comments, which led to a better presentation.

References

[1] G.E. Andrews, K. Eriksson, Integer Partitions, Cambridge University Press, 2004.
[2] R. Ariew, Ockham's Razor: A Historical and Philosophical Analysis of Ockham's Principle of Parsimony, Champaign-Urbana, University of Illinois, 1976.
[3] A. Blumer, A. Ehrenfeucht, D. Haussler, M.K. Warmuth, Occam's Razor, Information Processing Letters 24 (6) (1987) 377–380.
[4] C.S. Calude, Information and Randomness: An Algorithmic Perspective, 2nd ed., Springer-Verlag, Berlin, 2002.
[5] C.S. Calude, E. Calude, M.J. Dinneen, A new measure of the difficulty of problems, Journal for Multiple-Valued Logic and Soft Computing 12 (2006) 285–307.
[6] C.S. Calude, E. Calude, Evaluating the complexity of mathematical problems. Part 1, Complex Systems 18-3 (2009) 267–285.
[7] C.S. Calude, E. Calude, Evaluating the complexity of mathematical problems. Part 2, Complex Systems 18-4 (2010) 387–401.
[8] C.S. Calude, E. Calude, The complexity of the Four Colour Theorem, LMS Journal of Computation and Mathematics 13 (2010) 414–425.
[9] E. Calude, The complexity of Riemann's Hypothesis, Journal for Multiple-Valued Logic and Soft Computing 18 (3–4) (2012) 257–265.
[10] M.J. Dinneen, A program-size complexity measure for mathematical problems and conjectures, in: M.J. Dinneen, B. Khoussainov, A. Nies (Eds.), Computation, Physics and Beyond, in: LNCS, vol. 7160, Springer, Heidelberg, 2012, pp. 81–93.
[11] J. Kelleher, Encoding partitions as ascending compositions, Ph.D. Thesis, University College Cork, 2006.
[12] D. Knuth, The Art of Computer Programming, Pre-Fascicle 3b: Generating all partitions, http://www-cs-faculty.stanford.edu/%7Eknuth/fasc3b.ps.gz, (version of 10 December 2004).
[13] D. Stanton, D. White, Constructive Combinatorics, Springer-Verlag, New York, 1986.
[14] A. Zoghbi, I. Stojmenovic, Fast algorithms for generating integer partitions, International Journal of Computer Mathematics 70 (1998) 319–332.
[15] D. Wells, Are these the most beautiful?, The Mathematical Intelligencer 12 (3) (1990) 37–41.