Een eenvoudig algoritme om permutaties te genereren

Daniel von Asmuth

Inleiding

Er zijn in de vakliteratuur verschillende manieren beschreven om alle permutaties van een verzameling te generen. De methoden in dit artikel zijn grotendeels ontleend aan Sedgewick [1].

In het dagelijks leven noteren we getallen meestal in het decimale stelsel: een getal *g* geschreven als $c_n c_{n-1} ... c_2 c_1 c_0$ representeert een waarde van $g = c_n \times b^n + c_{n-1} \times b^{n-1} ... + c_2 \times b^2 + c_1 \times b^1 + c_0 \times b^0$ met als grondtal b = 10. Computers hanteren doorgaans het binaire getallenstelsel, met 2 als grondtal; programmeurs gebruiken soms het octale (grondtal 8) en hexadecimale (grondtal 16) stelsel.

Om permutaties te tellen is het handiger om het factoriale stelsel te gebruiken waarin het n^e cijfer wordt genoteerd op basis van grondtal *n*, zodat $c_n c_{n-1} ... c_2 c_1 c_0 = c_n \times (n+1)! + c_{n-1} \times n! ... + c_2 \times 3! + c_1 \times 2! + c_0 \times 1!$ (een technisch probleem in dit verhaal is dat we tellen vanaf 1 i.p.v. 0).

Het programmaatje in listing 1a geeft een manier om volgens het factoriale stelsel van n! naar 1 af te tellen.

Listing 1a. Factoriaal tellen (recursief)

```
#include    <stdio.h>
#define     n           4

int         array[ n + 1];
/* nulde element is ongebruikt */

/* drukt inhoud van het array af  */
void        print( void)
{
  int       i;

  for( i = 1; i <= n; i++)
    printf( "%01d ", array[ i]);
  printf( "\n");
}

void    loop_( int i, int j)
{
  if( i <= n)
  {
    if( j <= n - i)
    {
      array[ i] = j;
      loop( i + 1, 0);
      loop( i, j + 1);
    }
  }
  else
      print();
}
```

Listing 1b. Factoriaal tellen (iteratief)

```
void loopje( void)
{
  int i1, i2, i3, i4;

  for( i1 = 0; i1 < 4; i1++)
  {
    array[1] = i1;
    for( i2 = 0; i2 < 3; i2++)
    {
      array[2] = i2;
      for( i3 = 0; i3 < 2; i3++)
      {
        array[3] = i3;
        for( i4 = 0; i4 < 1; i4++)
        {
          array[4] = i4;
          print();
        }
      }
    }
  }
}
```

```
/* het hoofdprogramma begint hier */
int         main( void)
{
  loop_( 1, 1);
  return 0;
}
```

Listing 2. Factoriaal tellen

```
0 0 0 0
0 0 1 0
0 1 0 0
0 1 1 0
0 2 0 0
0 2 1 0
1 0 0 0
1 0 1 0
1 1 0 0
1 1 1 0
1 2 0 0
1 2 1 0
2 0 0 0
2 0 1 0
2 1 0 0
2 1 1 0
2 2 0 0
2 2 1 0
3 0 0 0
3 0 1 0
3 1 0 0
3 1 1 0
3 2 0 0
3 2 1 0
```

Listing 3. Permutaties

```
4 1 2 3
4 2 1 3
4 3 1 2
4 3 2 1
4 1 3 2
4 2 3 1
3 4 2 1
3 4 1 2
2 4 1 3
1 4 2 3
2 4 3 1
1 4 3 2
3 1 4 2
3 2 4 1
2 3 4 1
1 3 4 2
2 1 4 3
1 2 4 3
3 1 2 4
3 2 1 4
2 3 1 4
1 3 2 4
2 1 3 4
1 2 3 4
```

Recursie biedt vaak een elegante en beknopte manier om een algoritme te formuleren en te analyseren; iteratieve functies worden door de computer vaak sneller verwerkt. Ter illustratie volgen twee functies in de programmeertaal C, die de faculteit van een getal uit rekenen. De dubbel recursieve loop() functie kan eveneens in iteratieve vorm geschreven worden, zoals in listing 1b, maar als we de constante *n* willen veranderen in 5 hebben we al een extra for-lus nodig.

Listing 4a. Faculteiten berekenen

```
/* recursieve variant */
int             fac( int n)
{
  if( n > 1)
    return n * fac( n - 1);
  else
    return 1;
}
```

Listing 4b. Faculteiten berekenen

```
/* iteratieve variant */
int             fac_( int n)
{
  int           i;
  int           r;

  r = 1;
  for( i = n; i > 1; i--)
    r = i * r;

  return r;
}
```

Analyse van een algoritme

In ons algoritme om alle permutaties van 1 ... n af te drukken doet de loop() functie niets anders dan factoriaal tellen, met twee keer het verwisselen van twee array elementen. Ten behoeve van de analyse worden de permutaties in onderstaand programma in een andere volgorde gegenereerd.

Listing 5

```
#include    <stdio.h>
#define     n           4

int         array[ n + 1];          /* nulde element wordt niet gebruikt */

void        print( void)            /* drukt de inhoud van het array af
*/
{
  int       i;

  for( i = 1; i <= n; i++)
    printf( "%01d ", array[ i]);
  printf( "\n");
}

void        fill( void)             /* initialiseert het array met 1 .. n
*/
{
  int       i;

  for( i = 1; i <= n; i++)
    array[ i] = i;
}

void        swap( int i, int j)     /* verwisselt inhoud van a[i] en a[j]*/
{
  int       t;

  t = array [i];
  array[ i] = array[ j];
  array [j] = t;
}

void        loop( int i, int j)   /* genereert alle permutaties van 1..n */
{
  if( i > 0)
  {
    if( j >= i)
    {
      loop( i, j - 1);
      swap( i, j);
      loop( i - 1, n);
      swap( i, j);
    }
  }
  else
    print();
}

int         main( void)          /* het programma begint hier            */
{
  fill();
  loop( n, n);
  return 0;
}
```

We gaan nu op informele wijze na dat de functie loop() alle permutaties van 1 ... n genereert door stap voor stap de code na te lopen en waar de functie zichzelf recursief aanroept met inductie het resultaat te verifiëren.

Lemma 1

Na aanroep van 'loop()', ongeacht de argumenten, heeft 'array' de zelfde inhoud als daarvoor.

Bewijs

Als je de code van 'loop()' volgt blijkt de bewering voor twee van de drie vertakkingen te kloppen. Voor de hoofdtak zien we dat telkens een paar elementen van 'array[]' wordt verwisseld en later nog eens. Als in de recursieve aanroepen van 'loop()' evenmin iets verandert, dan is de bewering in alle gevallen waar.

De waarheid van lemma 1 blijkt uit een eenvoudige vorm van inductie: 'loop()' doet óf niets, óf het roept de hoofdtak aan, waarin twee keer 'swap()' en twee keer 'loop()' wordt aangeroepen, en daarin dan weer... zodat zelfs als de functie zichzelf oneindig keer herhaalt de inhoud van de rij 'array[]' nog niet verandert.

□

Lemma 2

Uitvoeren van 'loop(i, j)' doet niets als j < i.

Dit is in te zien door de code van de loop() functie na te lopen.

Lemma 3

Uitvoeren van 'loop(0, j)' zal de inhoud van 'array[]' afdrukken (ongeacht de waarde van j).

Idem dito.

Lemma 4

Uitvoeren van 'loop(1, j)' zal *j* permutaties afdrukken waarin het 1^e element is verwisseld met respectievelijk het 1^e, 2^e, ... j^e.

We gaan dit na door de functieaanroep uit te werken.

Bewijs

<u>Basis</u>

loop(1, 1)

↳	loop(1, 0)	*doet niets volgens lemma 2*
	swap(1, 1)	*verwisselt 1^e en 1^e element*
	loop(0, n)	*drukt de permutatie af volgens lemma 3*
	swap(1, 1)	*herstelt de uitgangssituatie*

<u>Inductie</u>

loop(1, j + 1)

↳	loop(1, j)	*volgens de inductiehypothese*
	swap(1, j + 1)	*verwisselt 1^e en $(j+1)^e$ element*
	loop(0, n)	*drukt de permutatie af volgens lemma 3*
	swap(1, j + 1)	*herstelt de uitgangssituatie*

□

Lemma 5

'loop(i, i)' heeft het zelfde effect als 'loop(i - 1, n)'

Bewijs

loop(i, i)

↳ loop(i, i - 1) *doet niets volgens lemma 2*
swap(i, i) *heeft geen effect*
loop(i - 1, n)
swap(i, i) *heeft geen effect*

□

Op basis van lemma 4 analyseren we de eerste recursieve functieaanroep en daarna de tweede.

Lemma 6

De functie 'loop(i, n)' zal (voor i < n) (n - i + 1) maal 'loop(i - 1, n)' aanroepen, nadat het i^e element verwisseld is met respectievelijk het i^e, $(i+1)^e$, ... n^e.

Bewijs

Basis

loop(i, i)

↳ loop(i, i - 1) *doet niets volgens lemma 2*
swap(i, i) *verwisselt i^e en i^e element*
loop(i - 1, n)
swap(i, i) *herstel*

Inductie

loop(i, j + 1)

↳ loop(i, j) *volgens de inductiehypothese*
swap(i, j + 1) *verwisselt i^e en $(j+1)^e$ element*
loop(i - 1, n)
swap(i, j + 1) *herstel*

□

Lemma 7

De functie 'loop(i, n)' zal (n) × (n - 1) × (n - 2) × ... × (n - i + 1) permutaties afdrukken met
het 1^e element verwisseld met het 1^e, 2^e, 3^e, ... n^e
het 2^e element verwisseld met het 2^e, 3^e, ... n^e
.....
het i^e element verwisseld met het i^e, $(i+1)^e$, $(i+2)^e$, ... n^e

Bewijs

Basis

loop(1, n) *drukt volgens lemma 4* n *permutaties af waarin het 1^e element is verwisseld met respectievelijk het 1^e, 2^e, 3^e, ... n^e*

Inductie

loop(i + 1, n)

↳ loop(i + 1, n - 1) } *verwisselt volgens lemma 6 het i^e element met respectievelijk het*
swap(i + 1, n) } *$(i+1)^e$, $(i+2)^e$, ... n^e*
loop(i , n) *inductiehypothese*
swap(i + 1, n) *herstel*

□

Uit lemma 7 volgt uiteindelijk dat loop(n, n) alle permutaties van 1 □ n zal afdrukken.

Rekentijd

Op de bovenstaande manier kunnen we nagaan hoeveel stappen de 'loop()' functie kost om tot de conclusie te komen dat de hoofdtak n × (n - 1) × (n - 2) × ... × 1 keer wordt uitgevoerd met tweemaal swap() per iteratie en drie of vier vergelijkingen. We sjoemelen een beetje door het feit dat elke aanroep van print() *n* stappen kost weg te laten.

We kunnen argumenteren dat er geen alternatief is dat niet tenminste *n* faculteit stappen nodig heeft en per permutatie minstens één paar elementen moet verwisselen, zodat dit algoritme in de buurt van een optimale oplossing komt.

Er is nog ruimte voor verbetering: door de eerste aanroep van swap() te vervangen door onderstaand swap1() en de tweede door swap2() zijn nog maar vijf van de zes assignments nodig.

Sorteren

Een array sorteren komt neer op het zoeken van de juiste permutatie: door de loop() functie te vervangen door sorteer() uit listing 7 verkrijgen we een alternatieve sorteerfunctie. De performance is niet goed, maar kan worden verbeterd door de beide recursieve aanroepen parallel uit te voeren.

Listing 6. Swap() functie versnellen

```
int             reserve[ n + 1];

void            swap1( int i, int j)
{
  reserve[ i] = array [i];
  array[ i] = array[ j];
  array [j] = reserve[ i];
}

void            swap2( int i, int j)
{
  array[ j] = array[ i];
  array [i] = reserve[ i];
}
```

Listing 7. Array sorteren

```
void        sorteer( int i, int j)
{
  if( i > 0)
  {
    if( j >= i)
    {
      if( array[j] >= array[i])
      {
        swap( i, j);
        sorteer( i - 1, n);
      }
      sorteer( i, j - 1);
    }
  }
}
```

Willekeurige permutaties

Als we maar de i^{e} permutatie hoeven te berekenen is de rekentijd minder van belang, zolang we maar niet alle voorgaande permutaties hoeven af te lopen. Hieronder is de loop dient de main() functie slechts als demonstratie. De rekentijd kan uiteraard worden verbeterd door de faculteiten te tabelleren in plaats van telkens te berekenen. Onderstaande vorm werkt slechts voor kleine waarden van *n*, maar kan eenvoudig worden aangepast voor gebruik van 'bignums'.

Listing 8. Willekeurige Permutaties

```
void            permute( int i, int j)

{
  int           a;
  int           b;

  if( j > 1)
```

```
  {
    a = i / fac( j - 1);
    b = i % fac( j - 1);

    permute( b, j - 1);
    swap( a + 1, j);
  }
}

int             main( void)
{
  int           i;

  for( i = 0; i < fac( n); i++)
  {
    fill();
    permute( i, n);
    print();
  }
  return 0;
}
```

De essentie van de permute() functie uit listing 7 is dat ze het binaire argument *i* converteert naar factoriale notatie. De variabele *a* wordt het j^{e} cijfer en de rest *b* wordt recursief geconverteerd.
De werking is beter te begrijpen is door te kijken hoe de positie van de vieren opschuift. Als permute() vanuit main() met j = n wordt aangeroepen wordt het n^{e} cijfer verwisseld met het (a+1)^{e} nadat het 1^{e} t/m (n-1)^{e} recursief zijn gepermuteerd, waarbij *b* het volgnummer binnen het blok van (n-1)! permutaties van 1 ... n-1 voorstelt.

Literatuur

[1] Sedgewick, R. Permutation Gerneration Methods. *Computing Surveys,* Vol 9, No 2, Juni 1977, 137 - 164