Programmeermethoden

Universiteit Leiden
The Netherlands

Arrays (vervolg)

Walter Kosters en Jonathan Vis

week 8: 28 oktober–1 november 2024

www.liacs.leidenuniv.nl/~kosterswa/pm/

week	onderwerp	boek	dictaat
28 okt–1 nov	Arrays (vervolg)	5	4.2,op31/36
4–8 nov	Arrays (vervolg 2)	5	4.2,op37/43
11–15 nov	Pointers	10	3.12,op44/46,52/55
18–22 nov	Recursie	13	3.10,op57/61
25–29 nov	Datastructuren	17	5,op47/51
2–6 dec	Algoritmen, Python		
9–13 dec	Oude tentamens . . .		

op = opgaven uit opgavendictaat; zelf maken, antwoorden: zie website.

In rood: de weken met een deadline op de maandag erna.

Tentamen: donderdag 16 januari 2025, 13:00–16:00 uur; Gorlaeus BW.0.17/18/29/30/31/40. Inschrijven!

Derde programmeeropgave — 1

Programmeermethoden 2024
Derde programmeeropgave: Life

De *derde* programmeeropgave van het vak Programmeermethoden in het najaar van 2024 heet *Life*; zie ook het zevende werkcollege, en lees geregeld deze pagina op WWW.

De opgave

Het is de bedoeling om een C++-programma te maken dat de gebruiker in staat stelt *Life* te spelen via een menu-systeem. Dat betekent dat de gebruiker van het programma kan kiezen uit een aantal mogelijkheden, de zogeheten *opties*. Er is één submenu, waarin ook weer enkele opties zijn. De bedoeling is dat het hele menu op één regel staat, onder de wereld (zie verderop).
De opties worden gekozen door de eerste letter van de betreffende optie in te toetsen (gevolgd door Enter), bijvoorbeeld een `s` of `S` om te stoppen. Uiteraard wordt een en ander duidelijk en ondubbelzinnig aan de gebruiker meegedeeld. Gebruik *geen* recursie!

Alle door de gebruiker ingetoetste symbolen moeten gecontroleerd worden, dat wil zeggen dat er binnen redelijke grenzen geen foute invoer geaccepteerd wordt. Zo zal het intoetsen van bijvoorbeeld `q` of `&` in het hoofdmenu genegeerd worden. Verder moet bij getalleninvoer karakter voor karakter ingelezen worden (met `cin.get ( );` als je elders ook nog `cin >> ...` gebruikt krijg je overigens soms problemen met "hangende Enter's"; gebruik dus overal `cin.get ( )`). Er moet ook op gelet worden dat er geen te grote getallen worden ingevoerd. Schrijf dus een geschikte functie `leesgetal` die de gelezen karakters (cijfers) omzet in een getal (tip: negeer alle "voorloop-Enter's"; verwerk alles tot en met de eerstvolgende enter, en maak hiervan zo goed mogelijk een getal, van een maximale grootte; zo kan `abc123defg999h`, als je een getal kleiner dan 10000 wilt, bijvoorbeeld verwerkt worden tot `1239`), en een functie `leesoptie` die netjes één karakter inleest en Enter's afhandelt! Aan de gebruiker mogen "redelijke" beperkingen worden gevraagd, bijvoorbeeld dat de in te voeren getallen maximaal vier cijfers hebben. Het programma moet dan echter wel bestand zijn tegen pogingen meer dan vier cijfers in te voeren. Ook het invoeren van letters in plaats van cijfers moet geen problemen opleveren. Houd het simpel!

Life is een cellulaire automaat, in 1970 bedacht door John Horton Conway. Zie verder het college, Wikipedia of hier, en Johan Bontes' implementatie [tarball van GitHub-versie, zie "Binary"; patterns]. In een 2-dimensionaal (zeg) 1000 bij 1000 rooster, de *wereld*, beginnen we met een eindig aantal levende vakjes oftewel cellen. Een levend vakje met minder dan 2 of meer dan 3 buren van de 8 (horizontaal, verticaal en diagonaal) gaat dood (uit eenzaamheid of juist overbevolking), met precies 2 of 3 levende buren overleeft het. In een dood vakje met precies 3 levende buren ontstaat leven. Dit leidt tot de volgende *generatie*. Let erop dat dit voor alle vakjes tegelijk gebeurt!
Eigenlijk moet het geheel zich afspelen op een oneindig rooster, maar we kiezen voor de eindige variant. Om moeilijkheden te voorkomen, spreken we af dat de rand van onze wereld altijd uit dode cellen blijft bestaan. De gebruiker ziet altijd een klein gedeelte van de wereld, de *view* geheten. Steeds staan de coordinaten van het punt linksboven genoemd. De hoogte en breedte van de view zijn member-variabelen (zie verderop), zeg 25 en 80. Liefhebbers mogen ze eventueel wijzigen in het parameter-submenu.

In het hoofdmenu zijn de volgende opties aanwezig:

1. Stoppen.
2. Heelschoon. Maak de wereld leeg (alle cellen gaan dood).
3. Schoon. Maak de view leeg (alle cellen in de view gaan dood).
4. Verschuif de view naar links, boven, rechts, of onder.
5. Parameters. Dit leidt tot een submenu om de parameters in te stellen, zie onder.
6. Random. Vul de view met random dode en levende cellen. De rest van de wereld blijft onveranderd.
7. Toggle. Klapt levend en dood om voor de cel op de "cursorpositie"; deze laatste kan met vier toetsen omhoog/omlaag/naar links/rechts (bijvoorbeeld W/A/S/Z) gewijzigd worden, waarbij de coordinaten steeds getoond worden.
8. Glidergun. Vul de view met een glidergun. Lees de configuratie in uit een file. (Als dit "hard-coded" wordt gedaan kost dat een halve punt.)
9. Een. Er wordt één generatie gedaan.
10. Gaan. Er worden een hele serie generaties gedaan — en allemaal getoond (zonder Enter's).

Steeds wordt de view getoond, in het begin ruwweg het midden van de wereld. Voor de optie Random moet een zelfgemaakte random-generator (nou ja, random) gebruikt worden, zie Hoofdstuk 3.9.3 uit het dictaat, gedeelte "aantekeningen bij de hoorcolleges".

Er zijn verschillende parameters, in te stellen via het gelijknamige submenu:

1. De *verschuivings-stapgrootte* van de view. Deze parameter wordt gebruikt als de view in één van de vier richtingen verschuift. Beeld de echte rand van de wereld ook duidelijk af, zodra deze in beeld is.
2. Het *percentage* cellen dat levend moet zijn bij de optie Random (bij benadering).
3. De *twee verschillende karakters* die op het scherm gebruikt worden voor levende en dode cellen.

Kies zelf redelijke grenswaarden voor deze parameters. En denk natuurlijk aan de optie "Terug naar het hoofdmenu".

De bedoeling is een klasse (`class`) `life` te maken, met daarin onder meer functies die ieder voor zich een menuoptie afhandelen. De parameters zijn typisch membervariabelen. Gebruik nog geen eigen headerfiles, alles moet deze keer in één file staan.

Opmerkingen
Gebruik geschikte (member)functies. Bij deze opgave mogen bij elke functie (zelfs `main`) tussen begin-{ en eind-} *hooguit circa 30* niet al te volle regels staan! Elke functie dient van commentaar voorzien te zijn, bij voorkeur één regel boven de functie. Let op goed parametergebruik: alle parameters, met uitzondering van membervariabelen, in de heading doorgeven, en de variabele-declaraties zowel bij `main` als bij de andere functies aan het begin. De enige te gebruiken headerfiles zijn in principe `iostream`, `fstream`, `cstdlib` en `string`. Zeer ruwe indicatie voor de lengte van het C++-programma: 400 à 500 regels. Denk aan het infoblokje.

Uiterste inleverdatum: maandag 11 november 2024, 18:00 uur.

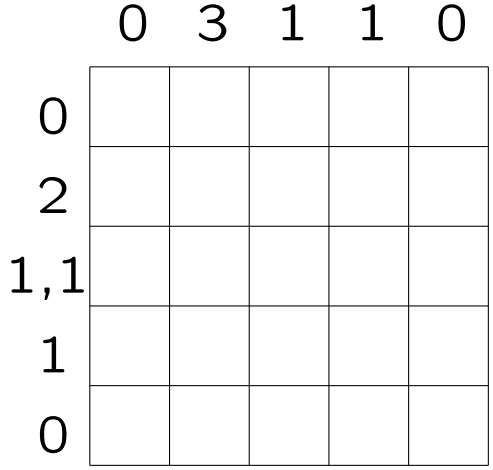
Manier van inleveren:

1. Digitaal de C++-code inleveren via Brightspace > Course Tools > Assignments. Stuur geen executable's, LaTeX-files of PDF-files, lever alleen één C++-file digitaal in!
2. Doe een print van het verslag in de doos bij kamer Gorlaeus BM.2.07.

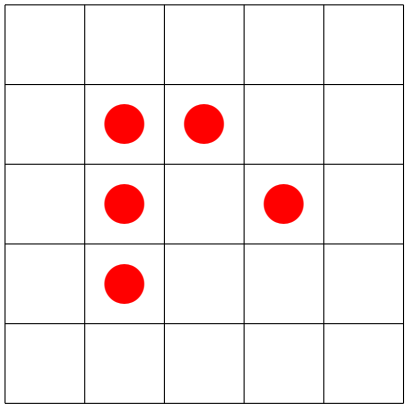
De laatst voor de deadline ingeleverde versie wordt nagekeken.
Overal duidelijk datum en namen van de (maximaal twee) makers vermelden, in het bijzonder als commentaar in de eerste regels van de C++-code.
Het *verslag* (uiteraard weer in LaTeX, zie de eerdere opgaven) moet het volgende bevatten: een beschrijving van het programma (waarin Life kort maar duidelijk wordt uitgelegd), een interessante Life-configuratie (bijvoorbeeld van internet; uiteraard met een nette citatie = referentie ("`\cite`")) met een plaatje van een niet al te zwart screenshot (in Linux: Shift-PrintScreen) van het eigen programma waarin deze configuratie in beeld is, een beschrijving van punten waarop het programma faalt (indien van toepassing), en een tabel met gewerkte uren, uitgesplitst per week en per persoon. Geef ook minstens één andere referentie, bijvoorbeeld naar werk van Conway. Zie hier hoe je een plaatje verwerkt, en hoe een citatie = referentie gemaakt wordt (en zo ziet dat eruit). Te gebruiken compiler: als hij maar C++ vertaalt; het programma moet in principe op een Linux-machine draaien. Het programma wordt doorgaans nagekeken met behulp van de compiler die (uiteraard) in het commentaar bovenin het programma vermeld staat. Normering: layout 1; commentaar (inclusief verslag) 2; modulariteit (OOP, functies) 3; werking 4. Eventuele aanvullingen en verbeteringen: lees de huidige WWW-bladzijde: `www.liacs.leidenuniv.nl/~kosterswa/pm/op3pm.php`.

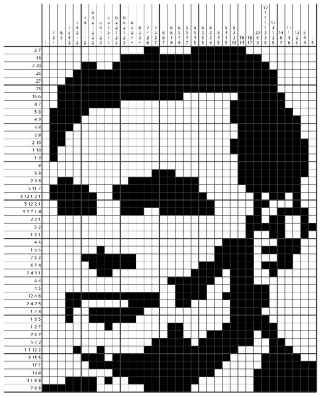
www.liacs.leidenuniv.nl/~kosterswa/pm/op3pm.php

Japanse puzzels (Nonogrammen) zien er zo uit:



glider





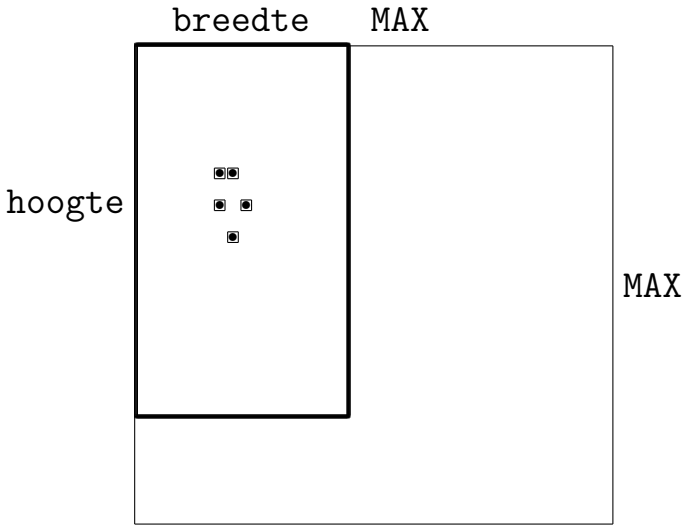
Naast iedere rij en boven ($\rightarrow$ onder) iedere kolom staan in volgorde de lengtes van aaneengesloten series rode blokjes.

Voor Life/Nonogram/LightsOut: 2-dimensionale arrays (matrices)!

www.liacs.leidenuniv.nl/~kosterswa/pm/op3pm.php

Een klasse `nonogram` voor Life ziet er ± zo uit:

```
class life {
  public:
    life ( );  // constructor
    void drukaf ( );
    void vulrandom ( );
    void maakschoon ( );
    void zetpercentage ( );
    // ...
  private:
    bool deWereld[MAX][MAX]; // array!!!
    int percentage;
    int hoogte;
    // ...
};//life (let op de punt-komma hier)
```



klasse object

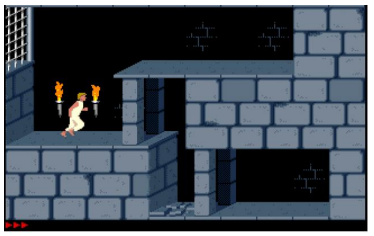
↓ ↓

```
life L;

L.drukaf ( );
```

Maak member-functies als (zie ook verderop):

```
// laat de Life-wereld zien
void life::drukaf ( );
  ...
}//life::drukaf
```



YouTube

en

```
// stel percentage in tussen 0 en 100
void life::zetpercentage ( ) {
  percentage = leesGetal (100);
}//life::zetpercentage
```

waarbij de zelfgemaakte functie `int leesGetal (int maxi)` een geheel getal, maximaal `maxi`, van toetsenbord inleest.

Voor een Life-wereld is een 2-dimensionaal array nodig:

```
bool deWereld[MAX][MAX];
```

Er geldt: `deWereld[i][j]` is `true` precies dan als rij `i` (van boven) en kolom `j` (van links) "levend" is.
En dit allemaal in een klasse `life`, met methoden als `void life::drukaf ( )`.

Maak eerst een *menu* en de functie `leesGetal`. Zie de tips:

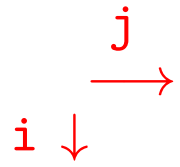
www.liacs.leidenuniv.nl/~kosterswa/pm/pmwc7.php

invoer moet $<$ 10000 zijn (bijvoorbeeld):

`\n\nabc123$%@@rr45xx\n` $\longrightarrow$ 1234 OF . . .
↖ Enter

Een basisfunctie is dus het afdrukken van een Life-configuratie:

```
// laat de Life-configuratie zien; eerste poging
void life::drukaf ( );
  int i, j; // voor rijen en kolommen
  for ( i = 0; i < hoogte; i++ ) {
    for ( j = 0; j < breedte; j++ ) {
      if ( deWereld[i][j] )
        cout << " X"; // <== later "levendkarakter"
      else
        cout << " .";
    }//for j
    cout << endl;
  }//for i
}//life::drukaf
```



En is linksboven $(0, 0)$?

We willen willekeurige (random) getallen maken. Dit gaat met het volgende recept, de lineaire congruentie methode (zie Knuth). Kies een startwaarde x (de "seed"). Pas dan herhaald toe

$$x \leftarrow (a \cdot x + c) \text{ modulo } m$$

met vaste a, c en m; en voor modulo lees: % uit C++.

Vaak: $c = 1$, m een macht van 10, en a modulo $200 = 21$ (zie dictaat). Dan krijg je zoveel mogelijk verschillende getallen, voordat het zich gaat herhalen.

Met $x = (5 * x + 1) \,\%\, 8$ krijg je 0–1–6–7–4–5–2–3–0–...
En $x = (3 * x + 1) \,\%\, 8$ geeft 0–1–4–5–0–...

```
// geef random getal tussen 0 en 999
int randomgetal ( ) {
  static int getal = 42;                    // (*)
  getal = ( 221 * getal + 1 ) % 1000;  // niet aan knoeien
  return getal;
}//randomgetal
```

(*) Een `static` variabele is locaal, wordt eenmalig geinitialiseerd, en blijft behouden tussen functie-aanroepen.

En een random getal uit $\{1, 2, 3, 4, 5, 6\}$ aanmaken? Doe:

```
  cout << 1 + randomgetal ( ) % 6 << endl;
```

Of misschien beter `1 + randomgetal ( ) / 167` ?

In `cstdlib` zit de RNG ("(pseudo)Random Number Generator") `rand ( )`, en `srand (...)` zet de seed.

Met `int A[8];` maak je een array met 8 gehele getallen:
`A[0]`, `A[1]`, `A[2]`, `A[3]`, `A[4]`, `A[5]`, `A[6]`, `A[7]`.

↑ array-index

array-element

We vullen het array:

```
A[0] = 1;
for ( int i = 1; i < 8; i++ ) A[i] = 2 * A[i-1];
```

En we drukken het af:

```
for ( int i = 0; i < 8; i++ ) cout << A[i] << " ";
```

Dat geeft `1 2 4 8 16 32 64 128 .` Op `A[5]` staat 32.

Eendimensionale arrays

```
const int MAX = 1000;
int A[MAX]; // of int A[1000];
// declareert/definieert een array bestaande uit
// MAX integers A[0], A[1], ..., A[MAX-1]
```

en for-loop

```
for ( int i = 0; i < MAX; i++ )
  A[i] = 0;
// zet alle array-elementen op 0
```

Er is een subtiel verschil tussen *declareren* en *definiëren*.

... en functies

beginadres grootte

↓ ↓

```
void kwadraat (int B[ ], int n) { // geen & nodig!!
  for ( int i = 0; i < n; i++ )
    B[i] = i * i;
}//kwadraat

// vult de array-elementen B[0] tot en met B[n-1]
// met de eerste n kwadraten: 0 tot en met (n-1)^2
```

met aanroep: `kwadraat (A,MAX);` of `kwadraat (A,500);`. Array `A` verandert, ook al is het een call-by-value parameter!

Hoe sorteer je een array oplopend? Een eerste idee is: zet herhaald de "kleinste" vooraan.

```
void simpelsort (int C[ ], int n) {
  int voorste, kleinste, plaatskleinste, k;
  for ( voorste = 0; voorste < n; voorste++ ) {
    plaatskleinste = voorste;
    kleinste = C[voorste];
    for ( k = voorste + 1; k < n; k++ )
      if ( C[k] < kleinste ) {
        kleinste = C[k];
        plaatskleinste = k;
      }//if
    if ( plaatskleinste > voorste )
      wissel (C[plaatskleinste],C[voorste]);
  }//for
}//simpelsort
```

zoek (plaats)kleinste van C[voorste],...,C[n-1]

Een voorbeeld van de werking van simpelsort:

```
0  1  2  3  4  5  6  (n=7)

3  8  7  5  2  4  9
2| 8  7  5  3  4  9
2  3| 7  5  8  4  9
2  3  4| 5  8  7  9
2  3  4  5| 8  7  9
2  3  4  5  7| 8  9
2  3  4  5  7  8| 9
2  3  4  5  7  8  9|
```

Bubblesort

En nog een (slechte) sorteermethode:

```
void bubblesort (int A[ ], int n) {
  int i, j;
  for ( i = 1; i < n; i++ )
    for ( j = 0; j < n - i; j++ )
      if ( A[j] > A[j+1] )
        wissel (A[j],A[j+1]);  // (*)
}//bubblesort
```

YouTube

Bij (*):

```
void wissel (int & a, int & b) {
  int hulp = a; a = b; b = hulp; }//wissel
```

of (zonder functie `wissel`):

```
  { int temp = A[j]; A[j] = A[j+1]; A[j+1] = temp; }
```

Een voorbeeld van de werking van simpelsort (links) en bubblesort (rechts):

```
0  1  2  3  4  5  6  (n=7)      0  1  2  3  4  5  6

3  8  7  5  2  4  9              3  8  7  5  2  4  9
2| 8  7  5  3  4  9              3  7  5  2  4  8 |9
2  3| 7  5  8  4  9              3  5  2  4  7 |8  9
2  3  4| 5  8  7  9              3  2  4  5 |7  8  9
2  3  4  5| 8  7  9              2  3  4 |5  7  8  9
2  3  4  5  7| 8  9              2  3 |4  5  7  8  9
2  3  4  5  7  8| 9              2 |3  4  5  7  8  9
2  3  4  5  7  8  9|
```

Bubblesort doet bij een rij met n elementen

$$(n-1)+(n-2)+\ldots+3+2+1 = n(n-1)/2$$

vergelijkingen tussen array-elementen. Het is een $O(n^2)$ ("orde n^2") algoritme — en dat is niet zo fijn.

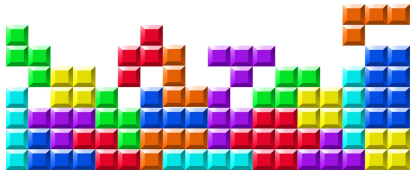
Dezelfde analyse geldt voor "simpelsort" = Selection sort.

Later meer over zoeken en sorteren ... het kan namelijk beter = sneller!

http://www.sorting-algorithms.com/

Insertion sort (invoegsorteer)

```
// sorteer array A (met n integers) oplopend
// met behulp van insertion sort (Opgave 54)
void invoegsorteer (int A[ ], int n) {
  int i, j, temp;
  for ( i = 1; i < n; i++ ) {  // zet A[i] goed in
    temp = A[i];      // reeds gesorteerd beginstuk
    j = i - 1;
    while ( ( j >= 0 ) && ( A[j] > temp ) ) {
      A[j+1] = A[j];
      j--;
    }//while
    A[j+1] = temp;
  }//for
}//invoegsorteer
```



link

Insertion sort: werking

3	8	14	17	32	33	37	16	12	58	7	1	34	6	9

0 1 2 ... $i-1$ i ... $n-1$

okee

In de i^{de} ronde is
$A[0]$ tot en met $A[i-1]$
van array A (met n elementen)
al gesorteerd en wordt $A[i]$
in het beginstuk op de juiste
plek "ingevoegd".

Het aantal vergelijkingen tussen array-elementen dat dit sorteeralgoritme doet hangt af van het invoerrijtje (met n elementen).

In het slechtste geval (worst case) kost het

$$1 + 2 + 3 + \ldots + i - 1 + \ldots + n - 1 = \frac{1}{2}n(n-1)$$

vergelijkingen om het array oplopend te sorteren, bijvoorbeeld als het beginarray aflopend gesorteerd is.

Het beste geval (best case) treedt op als het beginarray reeds oplopend gesorteerd is: $n - 1$ vergelijkingen.

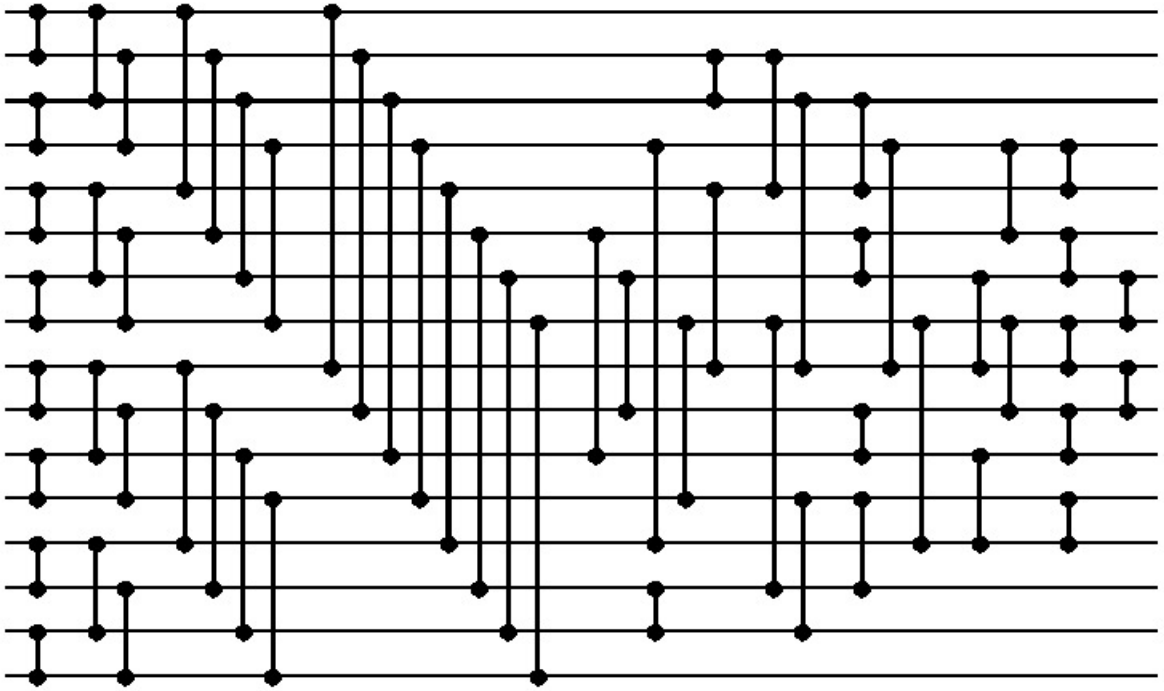
Shellsort ⋈

↑

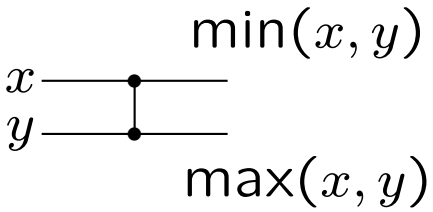
geen tentamenstof

```
void Shellsort (int A[ ], int n) {
  int i, j, h = n; // spronggrootte h
  while ( h > 1 ) {
    // A is h-gesorteerd
    h = h / 2;  // er bestaan betere keuzes dan / 2
    for ( i = h; i < n; i++ ) {
      // insertion sort op deelrijtjes
      temp = A[i]; j = i - h;
      while ( ( j >= 0 ) && ( A[j] > temp ) ) {
         A[j+h] = A[j];
         j = j - h;
      }//while
      A[j+h] = temp;
    }//for
  }//while
  // h = 1: A is nu 1-gesorteerd = gesorteerd
}//Shellsort
```

Er zijn allerlei andere benaderingen, zoals “counting sort” (voor 1323312132312 3), en sorteernetwerken (GPU’s):



vergelijker:



sorteert
16 getallen
60 vergelijkers
10 tijdstappen

sorteermethode	aantal vergelijkingen (worst case)
Selection sort	$O(n^2)$
Bubblesort	$O(n^2)$
Insertion sort	$O(n^2)$
Shellsort	$O(n\sqrt{n})$ (of nog beter?)
Quicksort	$O(n \lg n)$

Deze sorteeralgoritmen zijn alle gebaseerd op het doen van array-vergelijkingen (Selection sort = simpelsort); $\lg n = {}^2\log n = \log_2 n$.

Stelling: Elk sorteeralgoritme gebaseerd op het doen van array-vergelijkingen doet in het slechtste geval (de worst case) altijd *minstens* $\lg n! \approx c\, n \lg n$ vergelijkingen voor een array met n elementen.

zie de vakken Algoritmiek en Complexiteit . . .

```
void tabelletje (int n) {
  int i, j;
  for ( i = 1; i <= n; i++ ) { // buitenste loop
    cout << i << ": ";
    for ( j = 1; j <= i; j++ ) // binnenste loop
      cout << i * j << " ";
    cout << endl;
  }//for i
}//tabelletje
```

geeft, met `tabelletje (5);`:

```
1: 1
2: 2 4
3: 3 6 9
4: 4 8 12 16
5: 5 10 15 20 25
```

Tweedimensionale arrays (2D arrays, matrices)

```
const int m = 100;     // rijen
const int n = 50;      // kolommen
int A[m][n];           // of int A[100][50];
// declareert een tweedimensionaal array van m rijen
// en n kolommen, bestaande uit m*n integers
```

en dubbele for-loop

```
int i, j;
for ( i = 0; i < m; i++ )
  for ( j = 0; j < n; j++ )
    A[i][j] = 42;      // dus niet A[i,j]
// zet alle array-elementen op 42
```

... en functies

moet constante zijn!!
`B[ ][n]` mag ook (als `n const` is)

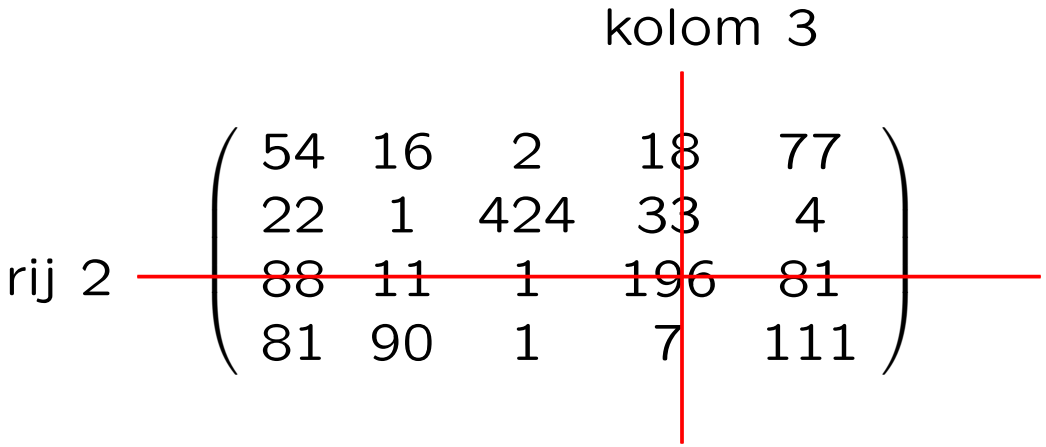
↓

```
int somarray (int B[ ][50], int zoveel) {
  int i, j, som = 0;
  for ( i = 0; i < zoveel; i++ )   // rijen
    for ( j = 0; j < 50; j++ )     // kolommen
      som += B[i][j];   // += betekent "ophogen met"
  return som;
}//somarray

// berekent de som van de elementen
// uit de eerste zoveel rijen van B
```

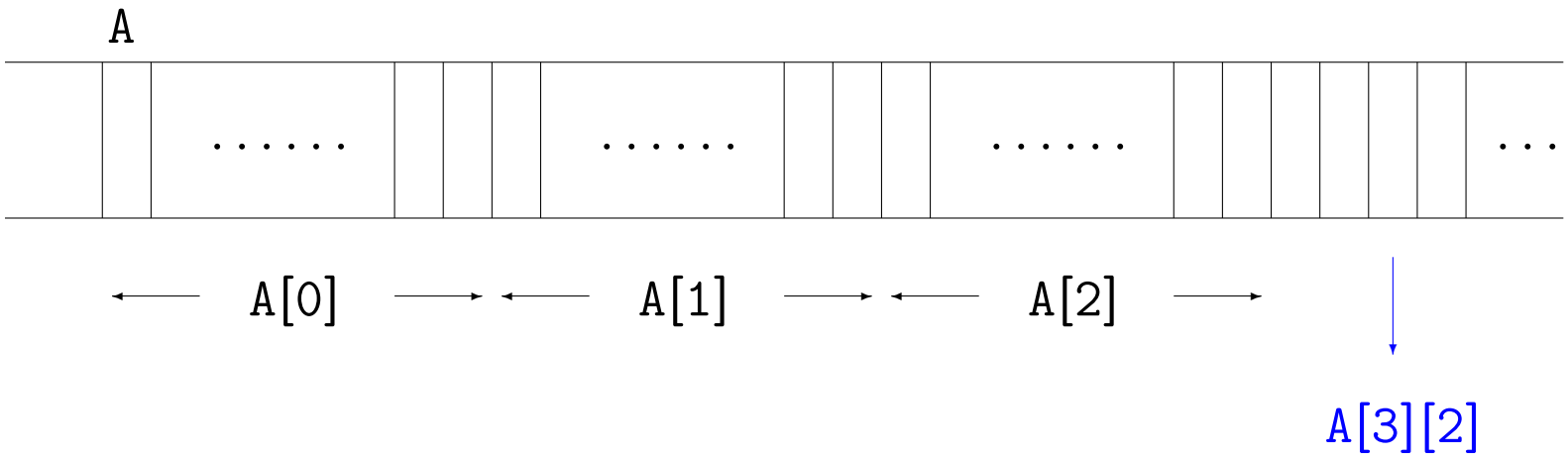
en aanroep: `antwoord = somarray (A,m);` of
`antwoord = somarray (A,20);` of ...

Een 4×5 array `A` (`int A[4][5];`) heeft 4 rijen en 5 kolommen:

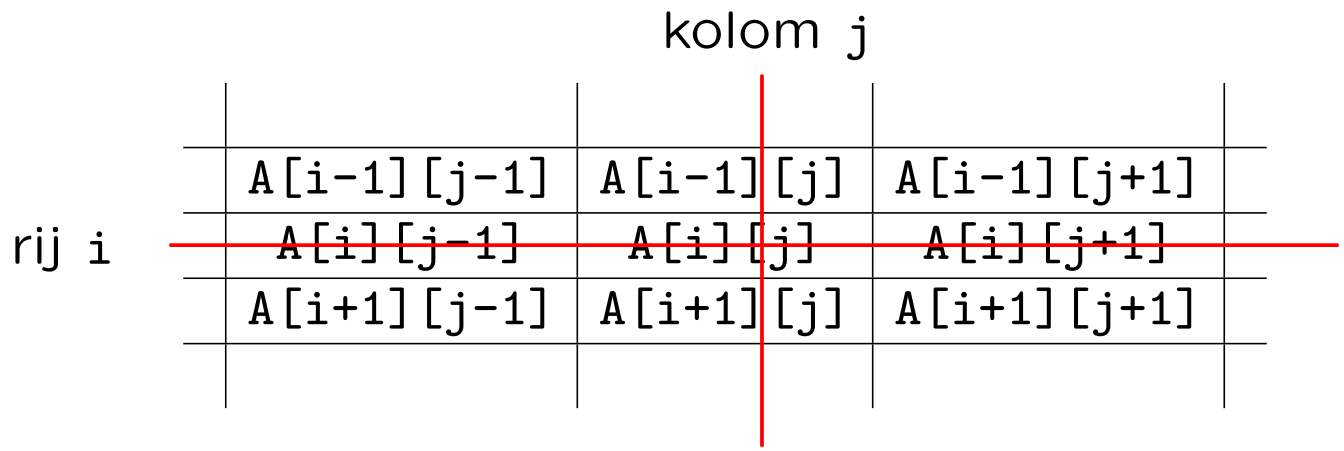


Op plek $(2, 3)$ staat het getal 196: `A[2][3]`.
Op plek $(0, 0)$ (dus linksboven) staat het getal 54: `A[0][0]`.
Rechtsonder staat `A[3][4]`: 111.
En het array-element `A[4][5]` "bestaat niet".

De rijen van zo'n 2-dimensionaal array `int A[m][n]` liggen achter elkaar in het geheugen:



Het adres van `A[3][2]` is `A + 3 * n + 2`, of eigenlijk preciezer `A + (3 * n + 2) * sizeof (int)`.
De `n` moet dus bekend, oftewel een `const`, zijn!



`m` × `n` array `int A[m][n];`

Array-elementen aan een rand hebben minder buren!

Als een functie de inhoud van het array moet veranderen is geen & nodig:

adres verandert niet ↓ dus ↓

```
void ikeerj (int A[ ][n], int m) {   // geen & !!!
  int i, j;
  for ( i = 0; i < m; i++ )
    for ( j = 0; j < n; j++ )
      A[i][j] = i * j;
}//ikeerj
```

↑ inhoud verandert wel

We bekijken nu `n` bij `n` vierkante matrices:

```
const int n = 42;

// C wordt de som van A en B, alle drie n bij n matrices
// optelling geschiedt elementsgewijs
void optellen (double A[ ][n], double B[ ][n],
               double C[ ][n]) {
  int i, j;
  for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ )
      C[i][j] = A[i][j] + B[i][j];
}//optellen
```

zie het vak Lineaire algebra . . .

We bekijken weer `n` bij `n` vierkante matrices:

```
const int n = 42;

// C wordt het (matrix-)product van A en B
void vermenigvuldigen (double A[ ][n], double B[ ][n],
                       double C[ ][n]) {
  int i, j, k;
  for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ ) {
      C[i][j] = 0;
      for ( k = 0; k < n; k++ )
        C[i][j] += A[i][k] * B[k][j];
    }//for j
}//vermenigvuldigen
```

Voorbeeld (we berekenen van $A \cdot B = C$ met name `C[1][0]`):

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

In het algemeen geldt:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] * B[k][j] \text{ ofwel } C_{ij} = \sum_{k=0}^{n-1} A_{ik} \cdot B_{kj}$$

Het product van twee matrices is overigens ook gedefinieerd voor niet-vierkante matrices A en B, mits maar geldt dat aantal kolommen van A = aantal rijen van B.

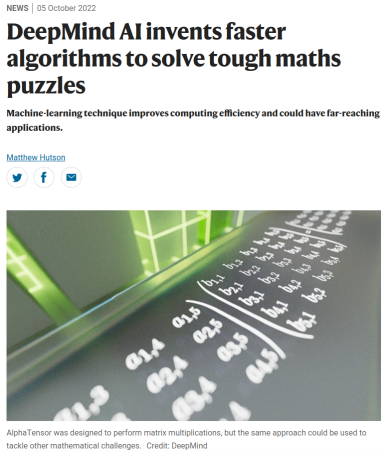
Matrixvermenigvuldiging: Strassen

Het gewone algoritme (links) kost $O(n^3)$ vermenigvuldigingen van array-elementen voor het product van twee $n \times n$ matrices; Strassen (rechts) "slechts" $O(n^{\log_2 7}) \approx O(n^{2.8})$:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

$$\begin{aligned} 19 &= 1*5+2*7 \\ 22 &= 1*6+2*8 \\ 43 &= 3*5+4*7 \\ 50 &= 3*6+4*8 \end{aligned}$$

$$\begin{aligned} M_1 &= (1+4)*(5+8) = 65 \\ M_2 &= (3+4)*5 = 35 \\ M_3 &= 1*(6-8) = -2 \\ M_4 &= 4*(7-5) = 8 \\ M_5 &= (1+2)*8 = 24 \\ M_6 &= (3-1)*(5+6) = 22 \\ M_7 &= (2-4)*(7+8) = -30 \\ 19 &= M_1+M_4-M_5+M_7 \\ 22 &= M_3+M_5 \\ 43 &= M_2+M_4 \\ 50 &= M_1-M_2+M_3+M_6 \end{aligned}$$



NEWS | 05 October 2022

DeepMind AI invents faster algorithms to solve tough maths puzzles

Machine-learning technique improves computing efficiency and could have far-reaching applications.

Matthew Hutson

AlphaTensor was designed to perform matrix multiplications, but the same approach could be used to tackle other mathematical challenges. Credit: DeepMind

Opgave 1 van het tentamen van 6 januari 2014:

In een array `int A[n]` staan `n` (een `const` > 0) gehele getallen.

a. Schrijf een C++-functie `hoevaak (A,X,n)` die teruggeeft hoe vaak het gehele getal `X` in het array `A` voorkomt.

b. Schrijf een Booleaanse C++-functie `uniek (A,n)` die precies dan `true` teruggeeft als geen enkel getal twee maal (of vaker) voorkomt in `A`, en anders `false`. Hierbij moet de functie van **a** *zinvol* gebruikt worden (hoe vaak komt `A[i]` voor?).

c. Schrijf een C++-functie `meest (A,n)` die het meest voorkomende getal uit `A` teruggeeft. Als er verschillende kandidaten zijn (bijvoorbeeld voor het array `17 12 30 12 42 30`) moet het kleinste getal dat het meest voorkomt worden geretourneerd. In het voorbeeld is dit `12` (dat even vaak voorkomt als `30`). Maak opnieuw gebruik van de functie van **a**.

d. Schrijf een C++-functie `sorteer (A,n)` die de getallen in `A` zodanig ordent dat voor alle getallen (behalve het laatste) geldt dat ze hooguit even vaak voorkomen als hun rechter buurman. Tip: pas de C++-code voor *bubblesort* eenvoudig aan; gebruik **a**.

e. Hoe vaak wordt de functie `hoevaak` aangeroepen in **d**?

zie werkcollege 31 oktober–1 november: 1a en 2a

- werk aan de derde programmeeropgave — de deadline is op maandag 11 november 2024
- bezoek daarom de colleges (video's), werkcolleges en vragenuren (de hele maandag)
- lees Savitch Hoofdstuk 5
- lees collegedictaat Hoofdstuk 3.8, 4.1 en 4.2
- maak opgaven 31/36 uit het opgavendictaat
- www.liacs.leidenuniv.nl/~kosterswa/pm/