PROGRAMMEREN 1 TAKE HOME TOETS

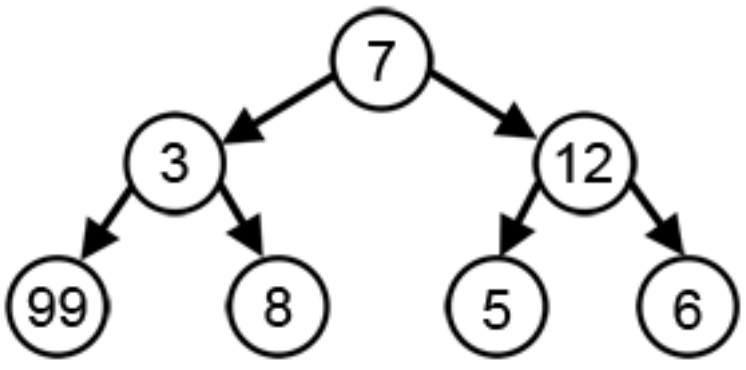
Rivka Vollebregt
12164968

7 april 2020

GREEDY

1) Wat zijn Greedy algoritmen en hoe werken ze?

   Een greedy algoritme is bedoeld om een probleem op te lossen door bij elke stap waar een beslissing moet worden genomen de optie te kiezen die het meest voordelig is. Dit klinkt alsof een greedy algoritme de beste oplossing voor een probleem geeft maar dit hoeft niet per se zo te zijn. Een greedy algoritme kijkt namelijk niet naar 'the big picture' en kiest niet voor welke kleine beslissingen samen de beste oplossing zijn voor het probleem, maar alleen naar de lokale beste oplossing voor een lokaal probleem. Zo kan het dus zijn dat als een greedy algoritme de taak heeft een snelste route te kiezen naar een bestemming, er bij een kruispunt zal worden gekozen voor de weg die rechtstreeks richting de bestemming gaat, ook als een andere weg uiteindelijk veel sneller is ondanks dat je daarvoor eerst een blokje om moet lopen. Een ander voorbeeld is een greedy algoritme dat door een netwerk van getallen, de route met de grootste som van getallen moet vinden zoals in de afbeelding beneden te zien is. Het greedy algoritme zal hier bij elke node voor het grootste getal kiezen en dus de route 7-12-6 nemen met als som 25, hoewel de 'beste' route in dit geval 7-3-99 met als som 109 is.



https://brilliant.org/wiki/greedy-algorithm/

2) Bij welke bedragen en munten werkt greedy.c niet met minimaal aantal munten?

Onderstaand is mijn code waarin ik andere munten heb gebruikt dan in de originele opdracht. Volgens dit algoritme bestaat bij een hoeveelheid change van 9, het wisselgeld uit 4 munten namelijk 6,1,1,1. Dit is niet de optimale oplossing omdat het in minder munten kan, namelijk in 3 munten: 4,4,1. Dit geld ook voor de situatie wanneer het wisselgeld bestaat uit de munten 4, 3 en 1 met het bedrag van 6. Greedy zal in die situatie als oplossing geven 3 munten: 4,1,1 terwijl de optimale oplossing 2 munten is: 3,3.
Dit is ook het geval bij het bedrag 12 euro en de munten 8, 6, 1. Als algemene regel valt vast te stellen als de derde munt 1 is, de tweede munt de helft is van het bedrag en de eerste munt hoger is, maar nog wel minimaal 2 lager dan het budget, greedy niet optimaal werkt.

```c
#include <cs50.h>
#include <stdio.h>
#include <math.h>

int main(void)
{
    int amount = 0;
    int count = 0;
    float dollars = 0;

    // prompt user for amount of change due
    do
    {
        dollars = get_float("Change owed: $ ");
    }
    while (dollars < 0);

    // convert input into cents
    dollars *= 1.0;
    amount = (int) round(dollars);

    // while amount is larger than coin give that coin as change
    while (amount >= 6)
    {
        count++;
        amount = amount - 6;
    }

    while (amount >= 4)
    {
        count++;
        amount = amount - 4;
    }

    while (amount >= 1)
    {
        count++;
        amount = amount - 1;
    }

    //print minimum number of coins for change
    printf("You receive %i coins \n", count);
}
```

LOOPS

1) Uitwerkingen van 4 loops opdrachten:

```
// Een algoritme dat een Mario Pyramide print met hoogte dat de
gebruiker invoert.

int n;

do
{
    n = get_int("enter height: ");
}
while (n < 0 || n > 23);

for (int row = 0; row < n; row ++)
{
    for (int column = n - row; column < 1; column--)
    {
        printf(" ");
    }

    for (int k = 0; k < row + 2; k++)
    {
        printf("#");
    }

    printf("\n");
}
```

```
// Een loop die alle laatste letters in de string tale print

string tale = "It was the best of times";

for (int i = 0; i < strlen(tale); i++)
{
    if (tale[i] == ' ')
    {
        printf("%c", tale[i - 1]);
    }
}

printf("%c\n", tale [strlen(tale)-1]);
```

```
// Een loop dat de gebruiker vraagt om een negatieve integer in
te voeren

int n;

do
{
     n = get_int(“enter negative integer: “);
}
while (n > 0);
```

```
// Een loop die het gemiddelde berekend van alle getallen in de
array numbers en deze opslaat in de variable avg

int numbers[] = {5, 7, 2, 4, 6};
int length = 5;
int avg;
int sum = 0;

for (int i = 0; i < length; i++)
{
    sum += numbers[i];
}

avg = (sum / length);
printf("%i\n",avg);
```

2) 2 varianten van de opdrachten in C-code:

```
// Dit programma is een variant op Arrays oefening 1: het
vraagt een target integer van de gebruiker en kijkt of deze
zich in de array bevindt.

#include <stdio.h>
#include <cs50.h>
#include <string.h>
#include <ctype.h>

int main(void)
{
    int numbers[] = {7, 3, 4, 5};
    int length = 4;

    int target = get_int("Which number are you looking for? ");

    for (int i = 0; i < length; i++)
    {
        if (numbers[i] == target)
        {

            printf("found\n");
            return true;
        }
    }

    printf("not found\n");
}


// Als de gebruiker de integer 7,3,4 of 5 intikt, is de
oplossing: found. Als de gebruiker iets anders dan deze
getallen in tikt is de oplossing: not found.
```

```
// Dit programma is een variant van de string bob zoals in de
opdrachten loops staat. Het print alle hoofdletters van een
string (in dit geval de string text).

#include <stdio.h>
#include <cs50.h>
#include <string.h>
#include <ctype.h>

int main(void)
{
    string text = "proGrammEren is LeUK als het lukT";

    for (int i = 0; i < strlen(text); i++)
    {
        if (isupper(text[i]))
        {
            printf("%c", text[i]);
        }
    }
}

// De oplossing is: GELUKT
```

ALGORITMEN

1) Selection sort

5	**8**	**2**	**1**	**3**
1	8	2	5	3
1	2	8	5	3
1	2	3	5	8
1	2	3	5	8
1	2	3	5	8

2) Bubble sort

6	**3**	**9**	**0**	**1**
3	6	9	0	1
3	6	0	9	1
3	6	0	1	9
3	0	6	1	9
3	0	1	6	9
0	3	1	6	9
0	1	3	6	9

3) Insertion sort

2	**7**	**8**	**1**	**9**
2	7	8	1	9
2	7	8	1	9
1	2	7	8	9
1	2	7	8	9

4) Merge sort

3	**1**	**9**	**5**	**2**
3	1	9	5	2
3	1	9	5	2
1	3	9	5	2
1	3	9	2	5
1	2	3	5	9

CODEKWALITEIT

1) Stijlmiddelen:
   Om de kwaliteit van mijn code te verbeteren probeer ik altijd om de code overzichtelijk te maken. Dit doe ik door te zorgen voor consistente whitespace regels tussen verschillende onderdelen in mijn code, bijvoorbeeld een whitespace regel tussen een blokje code dat het command-line-argument checkt en het volgende blokje dat iets uitrekent. Ook doe ik dit door indenting te gebruiken wat ervoor zorgt dat het meteen te zien is bij welk stuk code een bepaalde regel hoort, bijvoorbeeld in welke loop een printf command wordt uitgevoerd. Als ik een paar dagen of weken na het schrijven van een code weer terugkijk naar wat ik had gedaan en hoe de code in elkaar zit, is het handig om bij elk onderdeel comments te hebben staan zodat de structuur van de code makkelijker te begrijpen is. Ik vind het zelf altijd fijn om veel korte comments te gebruiken in plaats van een paar langere, maar het is wel belangrijk, zeker bij korte comments, om duidelijk te zijn.

   Iets dat erg verleidelijk is en ik in het begin ook vaak deed was korte namen voor variabelen gebruiken; in een dubbele for-loop gebruikte ik standaard i en j als variabelen. Hoewel het wat meer werk is, gebruik ik nu standaard wat langere namen voor variabelen om het duidelijker te maken wat er gebeurt in de code. Bijvoorbeeld bij de dubbele for-loop gebruik ik nu in plaats van i -> row en in plaats van j -> column. Bij een dubbele for-loop is i en j nog goed te begrijpen omdat het vaak wordt gebruikt, maar zeker bij andere variabelen zoals een counter of input van het command line argument gebruik ik altijd betekenisvolle namen zodat ik op een later moment of iemand anders die naar mijn code kijkt kan begrijpen waar het over gaat.

   Aan het begin van mijn code gebruik ik een header waarin ik een korte uitleg geef over de functie van de code en als dat nodig is ook de input en output van het programma beschrijf. Dit doe ik niet heel uitgebreid, want het wordt verder verduidelijkt door de comments.

   Wat ik ook altijd probeer te gebruiken in mijn code zijn loops in plaats van stukken code te kopiëren en plakken. Dit maakt de code net wat korter en scheelt ook tijd bij het debuggen omdat ik net een keer verkeerd heb gekopieerd en door de gekopieerde blokken niet zie waar de fout zit in de code, wat veel sneller te zien is in een loop. Wel probeer ik niet te veel in loops te doen om nesting te voorkomen.

   Ten slotte, gebruik ik als het kan of een globale variabele of maak ik de structuur van het programma zo dat de waarden van variabelen makkelijk aan te passen zijn. Als ik bijvoorbeeld een bepaald formaat 50 gebruik, zet ik aan het begin van het programma een variabele ‘formaat’ met de waarde 50 in plaats van overal waar formaat nodig is het getal 50 neer te zetten in de code. Als de code dan moet werken met een ander formaat hoef ik niet overal de 50 te veranderen maar kan ik de variabele ‘formaat’ boven aan de code een nieuwe waarde toekennen. Maar, als ik de variabele maar in 1 blokje code gebruik, zet ik deze niet helemaal bovenaan de code, maar net boven het blokje code waar het gebruikt wordt. Dit voorkomt dat bij een hele lange code er bovenaan een lijst met 30 variabelen staat wat ook niet overzichtelijk is.

2) Recover.c

```c
/**********************************************************************************
 * recover.c
 *
 * recovers jpeg images
 *
 * Programmeren 1
 *
 * Rivka Vollebregt
 * 12164968
 *
 * recover.c recovers jpeg files from a memory card that have been 'deleted'
 * the input is the data from the memory card and the output is the jpeg images
**********************************************************************************/
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    // error if the command line argument is missing or >1
    if (argc != 2)
    {
        printf("Usage: ./recover image\n");
        return 1;
    }

    // open card file with fopen
    FILE *input = fopen(argv[1], "r");

    if (input == NULL)
    {
        fprintf(stderr, "Could not open %s.\n", argv[1]);
        return 2;
    }

    unsigned char buffer[512];

    // create space for jpeg image filename
    char photo[8];

    // counter for number of jpeg's processed
    int counter = 0;

    // create pointer to output file
    FILE *output = NULL;
```

```c
    // repeat until end of card
    while (fread(buffer, 512, 1, input))
    {
        // check if start of a jpeg
        if (buffer[0] == 0xff && buffer[1] == 0xd8 && buffer[2] == 0xff &&
            (buffer[3] & 0xf0) == 0xe0)
        {
            // if already found jpeg, close old jpeg and start new jpeg
            if (counter > 0)
            {
                fclose(output);
            }

            // if not found, close previous file, open new
            sprintf(photo, "%03i.jpg", counter);

            // open image file and check if successful
            output = fopen(photo, "w");

            if (output == NULL)
            {
                fprintf(stderr, "Could not create %s.\n", photo);
                return 3;
            }

            // up 1 the number of jpeg's created
            counter++;
        }

        // if photo exists, write the block to it
        if (output != NULL)
        {
            // write to image file
            fwrite(buffer, 512, 1, output);
        }
    }

    // close any remaining files (end of card)
    fclose(output);
    fclose(input);

    return 0;
}
```

In recover.c ben ik de code begonnen met een fileheader waar een korte uitleg in staat over wat de functie is van de code. De rest van de uitleg over de functie heb ik in kleine comments in de code gezet boven elk blokje code waar het bij hoort, zoals te zien is in de blauwe regels. Tussen elk blokje code heb ik een whitespace regel gebruikt zodat het duidelijk is waar een nieuwe stuk code begint. Ook is er indenting aanwezig waardoor ik goed kan zien welk commando in welke loop hoort en dus wanneer het worden uitgevoerd.

Voor de variabelen heb ik namen gebruikt die representatief zijn voor de functie, zoals buffer en photo. In recover.c zijn niet veel variabelen en dus heb ik ze bovenaan mijn code gezet omdat ik dat beter vond staan dan direct boven het blokje code waar ze worden gebruikt. Dit had als bijkomend voordeel dat de variabele buffer bovenaan wordt aangemaakt en daar ook een grootte wordt toegekend zodat dit makkelijk te veranderen is mocht dat nodig zijn.

In recover.c heb ik geen gebruik gemaakt van for-loops, maar alleen van een while-loop, omdat dat het best paste bij de functie van de code.

Een verbeterpunt aan deze code is dat ik gebruik heb gemaakt van ‘magic numbers’ als 512 bij buffer en 8 bij photo. Ik had hier beter `char photo[length_of_char]` kunnen gebruiken.