Artificial Intelligence 1
Lab 1

Name1 (student number 1) & Name2 (student number 2)
Group name

day-month-year

Theory

Exercise 1

Exercise 2

Programming

Program description

The program asks for two numbers as inputs, and a search method, and looks for a way to produce the second number from the first, using the allowed operations and the given search method. It prints out the way of production, and the number of states enlisted and examined, if the program finds a solution within the given memory boundary.

Problem analysis

There were several functions to implement: The BFS and DFS methods had to be improved to give a more efficient solution, the heap had to be filled to the skeleton of the method, and the IDS method had to written from skratch. Also, the program had to be extended to print the series of actions to the solution as well.

Program design

The design of the problems, in order of the questions is the following:

- In the BFS method, the task 0 to 100 runs out of memory, and in the DFS method the task 0 to 1 runs out of memory, the others in question 1 can be produced. The reason is, that the solutions are only checked when visited, and not when generated, so the DFS only checks in the sequence of the last action (rightmost branch), and the BFS adds 1 level of depth to the complexity, that can be avoided. The solution is to check for goals when generating a node, and not only when visiting them. To implement this, we defined a helper function newValue that can be used to calculate the new values for every action iteratively, and also allows to check for solutions iteratively before enlisting the new node. Thus, a loop for node generation is implemented within the search functions, that loops through the "cases", the valid operations.

- To keep track of the preceding actions in every node generated, we defined a new struct called Operation, that stores an operatoin applied to a state, and the value after that. We also extended the struct State with an array of operations, and we add all the operations in the sequence here. When a solution is found, we add the last operation, and simply print out the sequence that is stored in the State's operation array (called path).

- For implementing the heap, we modified the heap functions from Algorithms and Data Structures to take a fringe as an argument. This way, the fringe's array could be used for a heap. We also modified the relations to store the minimum in the beginning. The heap functions are added to the fringe files. The heap order is based on the cost of the paths, as it is suggested in the fifth question. With the heap functions added, we only had to add enqueue and dequeue to the fringe functions. This gives and optimal path of steps 6 and cost 9 between 0 and 42.

```
C:\Users\Bálint\Desktop\Bálint\RUG\1D\AI1\find42\Find42>a.exe HEAP 0 42
Problem: route from 0 to 42
0  (+1)->1  (+1)->2  (*3)->6  (+1)->7  (*2)->14  (*3)->42
length = 6, cost = 9
goal reached (302 nodes visited)
#### fringe statistics:
 #size         :    1360
 #maximum size:    1360
 #insertions  :    1661
 #deletions   :     302
####
```

- The iterative deepening is basically a limited DFS with increasing limits, and the limited DFS is basically a DFS with a given limit. To use these properties, we did not implement the IDS in the fringe, but rather in the search function: we added an extra loop, that is only repeated when the mode is IDS, and an other if statement before generating nodes, that checks if the depth is within limit (that is always true in the other modes, since the limit in those is practically set to infinity). The extra loop increases the limit by 1 in every iteration, and runs it as a DFS, with the limit checker.

- Comparing the different methods, the following conclusions can be drown:

  – The DFS method uses relatively small memory, but it only examines the leftmost branch of the tree, and the children that are generated on that branch. Thus, in many cases, when the children is not accesible on that branch it reaches the memory limit without finding the solution, so it is not an optimal solution. However, if the solution is on that branch, DFS finds it the quickest. The memory complexity is O(6m) for 6 = branching factor and m = maximal depth, but with no limits it can be arbitrarily large. The time complexity is also O(6m), since it never returns from any states with no limits or constraints. Note that m is practically infinite, giving a high chance of extending the memory limit.

- The BFS method enqueues all the generated nodes, and calculates upon the oldest one, so it proceeds horizontally on the tree. This means, that it stores all the nodes on a depth level, giving high memory usage, and high time complexity (no ordering in this case), but is reliable in finding the solution. This gives us the time comlexity of $O(6^d)$, with branching factor 6 and the solution depth d. The memory complexity is also $O(6^d)$.
- The HEAP method works similarily as the BFS, but it uses ordering based on the cost, that is not neccessarily helps with finding the solution earlier, so it does not reduce time or memory complexity, but ensures, that the given answer is cost-optimal. The complexities here depend on the optimal path cost C*, giving complexities $O(6^{1+C*})$, where the minimal cost is 1.
- The IDS method runs int the order of the BFS, but inherits the low memory usage of DFS. The time complexity is the same order as in BFS, but in practice it is slightly higher, since many nodes are generated multiple times. However, the memory complexity is low, only in O(bd). If the task is to find a way, and not to find the optimal way, this is the most reliable, and most memory-efficient search method in this task.

Program evaluation

The program runs with no memory leaks. The complexity of the solution depends on which search method was used, as listed in the last point in the previous section.

Program output

The program outputs a sequence of operations starting from the first number to produce the second number. The outputs may differ using different modes, since most of the times there are multiple solutions, and the modes visit the nodes in different order. Only the heap method is optimal in the sense of giving the lowest cost path. The program also outputs the fringe statistics, making it simple to compare the methods.

Program files

search.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fringe.h"
#define RANGE 1000000

int newValue (int value, int cases){
	switch (cases){
		case 0:
			value += 1;
			return value;
		case 1:
			value -= 1;
			return value;
		case 2:
			value *= 2;
			return value;
		case 3:
			value *= 3;
			return value;
		case 4:
			value /= 2;
			return value;
		case 5:
			value /=3;
			return value;
	}
}
char *operationID(int cases){
	char *stringOperator;
	switch (cases){
		case 0:
			stringOperator = "+1";
			return stringOperator;
		case 1:
			stringOperator = "-1";
			return stringOperator;
		case 2:
			stringOperator = "*2";
			return stringOperator;
		case 3:
			stringOperator = "*3";
			return stringOperator;
		case 4:
			stringOperator = "/2";
			return stringOperator;
```

```
        case 5:
            stringOperator = "/3";
            return stringOperator;
    }
}

void updateState(State *s, State old, int cases, int value){
  s->cost = old.cost + (cases / 2) + 1;
  s->pathlen = old.pathlen + 1;
  s->depth = old.depth + 1;
  s->value = value;
  s->path = malloc((old.pathlen + 1) * sizeof(Operation));
  for (int i = 0; i<old.pathlen; i++){
    s->path[i] = old.path[i];
    }
  s->path[old.pathlen] = newOp(cases, value);
}

void printPath(int start, State s){
    printf("%d ", start);
    for (int i = 0; i< s.pathlen ; i++){
        printf(" (%s)->%d ", operationID(s.path[i].op),
          s.path[i].value);
    }
    printf("\nlength = %d, cost = %d\n", s.pathlen, s.cost);
}

Fringe insertValidSucc(Fringe fringe, int value, State old, int cases) {
  State s;

  if ((value <= 0) || (value > RANGE)) {
    /* ignore states that are out of bounds */
    return fringe;
  }
  updateState(&s, old, cases, value);
  return insertFringe(fringe, s);
}

void search(int mode, int start, int goal) {
  Fringe fringe;
  State state;
  int goalReached = 0;
  int visited = 0;
  int value;
  int limit = 9999999;                //set limit to infinity for non-IDS modes
  if (mode == IDS){limit = 1;}
  fringe = makeFringe(mode);


```

```
  do{                                                          //Extra loop for the iterative deepening, in other mode only 1 iteration
        //printf("limit is %d\n", limit);
        state.value = start;
        state.depth = 0;
        state.pathlen = 0;
        state.cost = 0;
          state.path = malloc(sizeof(Operation));
              fringe = insertFringe(fringe, state);
        while (!isEmptyFringe(fringe)) {
              /* get a state from the fringe */
              fringe = removeFringe(fringe, &state);
              visited++;
              /* is state the goal? */
              value = state.value;
              int valNew;
              /* insert neighbouring states */
              for (int cases = 0; cases < 6; cases++){
                      valNew = newValue(value, cases);
                      if (state.depth <= limit){   //in non-IDS this is always true
                              if (valNew == goal){
                                      goalReached = 1;
                                      State printState;
                                      updateState(&printState , state, cases, valNew);
                                      printPath(start, printState);
                                      free(state.path);
                                      free(printState.path);
                                      break;
                              }
                              fringe = insertValidSucc(fringe, valNew, state, cases);
                      }
              }
              if (goalReached){break;}
              free(state.path);
        }
      limit++;
      }while (mode == IDS&&!goalReached);
  if (goalReached == 0) {
    printf("goal not reachable ");
  } else {
    printf("goal reached ");
  }
  printf("(%d nodes visited)\n", visited);
  showStats(fringe);
  deallocFringe(fringe);
}
```

```c
int main(int argc, char *argv[]) {
  int start, goal, fringetype;
  if ((argc == 1) || (argc > 4)) {
    fprintf(stderr, "Usage: %s <STACK|FIFO|HEAP|IDS> [start] [goal]\n", argv[0]);
    return EXIT_FAILURE;
  }
  fringetype = 0;

  if ((strcmp(argv[1], "STACK") == 0) || (strcmp(argv[1], "LIFO") == 0)) {
    fringetype = STACK;
  } else if (strcmp(argv[1], "FIFO") == 0) {
    fringetype = FIFO;
  } else if ((strcmp(argv[1], "HEAP") == 0) || (strcmp(argv[1], "PRIO") == 0)) {
    fringetype = HEAP;
  } else if (strcmp(argv[1], "IDS") == 0)
        fringetype = IDS;
  if (fringetype == 0) {
    fprintf(stderr, "Usage: %s <STACK|FIFO|HEAP|IDS> [start] [goal]\n", argv[0]);
    return EXIT_FAILURE;
  }

  start = 0;
  goal = 42;
  if (argc == 3) {
    goal = atoi(argv[2]);
  } else if (argc == 4) {
    start = atoi(argv[2]);
    goal = atoi(argv[3]);
  }

  printf("Problem: route from %d to %d\n", start, goal);
  search(fringetype, start, goal);
  return EXIT_SUCCESS;
}
```

fringe.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include "fringe.h"
/////////////////////////////////////////////operation
```

```c
functions////////////////////////
Operation newOp(int op, int value){
        Operation o;
        o.value = value;
        o.op = op;
        return o;
}
////////////////////////////////////////////heap functions////////////////////////

int isEmptyHeap (Fringe h) {
        return (h.front == 1);
}

void heapEmptyError() {
        printf("heap empty\n");
        abort();
}

void doubleHeapSize (Fringe *fringe) {
        //printf("doubleing heap size\n");
        int newSize = 2 * fringe->size;
        fringe->states = realloc(fringe->states, newSize * sizeof(State));
        assert(fringe->states != NULL);
        fringe->size = newSize;
}

void swap(State *pa, State *pb) {
        State h = *pa;
        *pa = *pb;
        *pb = h;
}

void enqueue (State n, Fringe *fringe) {
        //printf ("enqueueing position\n");
        int fr = fringe->front;
        if ( fr >= 1 + fringe->size ) {            //size is only increased after enqueing
            doubleHeapSize(fringe);
        }
        fringe->states[fr] = n;
        upheap(fringe,fr);
        fringe->front++;
}

void upheap(Fringe *fringe, int n){
        //printf("upheap started\n");
        if (n<=1){return;}
        if ( fringe->states[n/2].cost>fringe->states[n].cost ) {
```

```
            swap(&(fringe->states[n]),&(fringe->states[n/2]));
            upheap(fringe,n/2);
        }
}

void downheap (Fringe *fringe, int n) {
        //printf("downheap started\n");
        int fr = fringe->front;
        int indexMax = n;
        if ( fr < 2*n+1 ) { /* node n is a leaf, so nothing to do */
            return;
        }
        if ( fringe->states[n].cost > fringe->states[2*n].cost ) {
            indexMax = 2*n;
        }
        if ( fr > 2*n+1 && fringe->states[indexMax].cost > fringe->states[2*n+1].cost ) {
            indexMax = 2*n+1;
        }
        if ( indexMax != n ) {
            swap(&(fringe->states[n]),&(fringe->states[indexMax]));
            downheap(fringe,indexMax);
        }
}

State dequeue(Fringe *fringe) {
        //printf("dequeue started\n");
        State n;
        if ( isEmptyHeap(*(fringe) ) ){
            //heapEmptyError();
            return n;
        }
        n = fringe->states[1];
        fringe->front--;
        fringe->states[1] = fringe->states[fringe->front];
        downheap(fringe,1);
        return n;
}

void printHeap(Fringe fringe){
        printf("heap is :");
        for (int i = 0; i< fringe.size; i++){
            printf("%d ", fringe.states[i]);
        }
        printf("\n");
}

/////////////////////////////////Fringe functions/////////////////////////////////////////////////

```

```c
Fringe makeFringe(int mode) {
  /* Returns an empty fringe.
   * The mode can be LIFO(=STACK), FIFO, or PRIO(=HEAP)
   */
  Fringe f;
  if ((mode != LIFO) && (mode != STACK) && (mode != FIFO) &&
      (mode != PRIO) && (mode != HEAP) && (mode != IDS)) {
    fprintf(stderr, "makeFringe(mode=%d): incorrect mode. ", mode);
    fprintf(stderr, "(mode <- [LIFO,STACK,FIFO,PRIO,HEAP])\n");
    exit(EXIT_FAILURE);
  }
  f.mode = mode;
  f.size = f.front = f.rear = 0; /* front+rear only used in FIFO mode */
  f.states = malloc(MAXF*sizeof(State));
  if (mode == HEAP||mode == PRIO){
          f.size = f.front = f.rear = 1;
          f.states[0].path = malloc(sizeof(Operation));
          }                                          // heap index starts from 1, easier to calculate, allocated memory for easy free
  if (f.states == NULL) {
        fprintf(stderr, "makeFringe(): memory allocation failed.\n");
    exit(EXIT_FAILURE);
  }
  f.maxSize = f.insertCnt = f.deleteCnt = 0;
  return f;
}



void deallocFringe(Fringe fringe) {
  State state;
  /* Frees the memory allocated for the fringe */
  while(!isEmptyFringe(fringe)){
         fringe = removeFringe(fringe, &state);
         free(state.path);
  }
  free(fringe.states);
}
void resetFringe(Fringe fringe){
        deallocFringe(fringe);
        fringe.states = malloc(MAXF*sizeof(State));
        fringe.front = fringe.rear = fringe.size = 0;
}

int getFringeSize(Fringe fringe) {
  /* Returns the number of elements in the fringe
   */
  return fringe.size;
}

```

```c
int isEmptyFringe(Fringe fringe) {
  /* Returns 1 if the fringe is empty, otherwise 0 */
  return (fringe.size == 0 ? 1 : 0);
}

Fringe insertFringe(Fringe fringe, State s, ...) {
  /* Inserts s in the fringe, and returns the new fringe.
   * This function needs a third parameter in PRIO(HEAP) mode.
   */
  // printf("%d\n", s.value);
  //int priority;
  //va_list argument;

  if (fringe.size == MAXF) {
    fprintf(stderr, "insertFringe(..): fatal error, out of memory.\n");
    exit(EXIT_FAILURE);
  }
  fringe.insertCnt++;
  switch (fringe.mode) {
  case LIFO: /* LIFO == STACK */
  case STACK:
  case IDS:
    fringe.states[fringe.size] = s;
    break;
  case FIFO:
    fringe.states[fringe.rear++] = s;
    fringe.rear %= MAXF;
    break;
  case HEAP:
  case PRIO: //using a heap for priority queue
        enqueue(s, &fringe);
        break;
  }

  fringe.size++;
  if (fringe.size > fringe.maxSize) {
    fringe.maxSize = fringe.size;
  }
  return fringe;
}

Fringe removeFringe(Fringe fringe, State *s) {
  /* Removes an element from the fringe, and returns it in s.
   * Moreover, the new fringe is returned.
   */
   //printf ("removing\n");
  if (fringe.size < 1) {
    fprintf(stderr, "removeFringe(..): fatal error, empty fringe.\n");
    exit(EXIT_FAILURE);
  }
```

```c
  fringe.deleteCnt++;
  fringe.size--;
  switch (fringe.mode) {
  case LIFO: /* LIFO == STACK */
  case STACK:
  case IDS:
    *s = fringe.states[fringe.size];
    break;
  case FIFO:
    *s = fringe.states[fringe.front++];
    fringe.front %= MAXF;
    break;
  case HEAP:
  case PRIO: //for priotity queue implementation we use a heap
        *s = dequeue(&fringe);
        break;
  }
  return fringe;
}

void showStats(Fringe fringe) {
  /* Shows fringe statistics */
  printf("#### fringe statistics:\n");
  printf(" #size      : %7d\n", fringe.size);
  printf(" #maximum size: %7d\n", fringe.maxSize);
  printf(" #insertions : %7d\n", fringe.insertCnt);
  printf(" #deletions : %7d\n", fringe.deleteCnt);
  printf("####\n");
}
```

fringe.h

```c

#ifndef FRINGE_H
#define FRINGE_H
#include <stdarg.h>

#define MAXF 500000 /* maximum fringe size */

#define LIFO 1
#define STACK 2
#define FIFO 3
#define PRIO 4
#define HEAP 5
#define IDS     6

////////////////////////////////////////Operation definition//////////////////////////////////////////////////
typedef struct Operation {
```

```
        int op;
        int value;
}Operation;

Operation newOp(int op, int value);
///////////////////////////////////////////////STATE DEFINITION///////////////////
typedef struct {
  int value;
  int cost;
  int pathlen;
  int depth;
  Operation *path;
} State;
///////////////////////////////////////////////////////////////////////////////////////

typedef struct Fringe {
  int mode;     /* can be LIFO(STACK), FIFO, or PRIO(HEAP)     */
  int size;     /* number of elements in the fringe            */
  int front;    /* index of first element in the fringe (FIFO mode) */
  int rear;     /* index of last element in the fringe (FIFO mode) */
  State *states; /* fringe data (states)                       */
  int insertCnt; /* counts the number of insertions            */
  int deleteCnt; /* counts the number of removals (deletions)  */
  int maxSize;  /* maximum size of the fringe during search    */
} Fringe;

Fringe makeFringe(int mode);
/* Returns an empty fringe.
 * The mode can be LIFO(=STACK), FIFO, or PRIO(=HEAP)
 */

void deallocFringe(Fringe fringe);
/* Frees the memory allocated for the fringe */

int getFringeSize(Fringe fringe);
/* Returns the number of elements in the fringe
 */

int isEmptyFringe(Fringe fringe);
/* Returns 1 if the fringe is empty, otherwise 0 */

Fringe insertFringe(Fringe fringe, State s, ...);
/* Inserts s in the fringe, and returns the new fringe.
 * This function needs a third parameter in PRIO(HEAP) mode.
 */

Fringe removeFringe(Fringe fringe, State *s);
/* Removes an element from the fringe, and returns it in s.
 * Moreover, the new fringe is returned.
```

```
 */

void resetFringe(Fringe fringe);

void showStats(Fringe fringe);
/* Shows fringe statistics */

///////////////////////////////////////////////HEAP FUNCTION
    DEFINITIONS//////////////////
int isEmptyHeap (Fringe h) ;
void heapEmptyError() ;
void doubleHeapSize (Fringe *fringe) ;
void swap(State *pa, State *pb) ;
void enqueue (State n, Fringe *fringe) ;
void upheap(Fringe *fringe, int n);
void downheap (Fringe *fringe, int n) ;
State dequeue(Fringe *fringe) ;
void printHeap();

#endif
```