Problem Set 1-a

Problem 1. There is a 5-digit number that satisfies 4*abcde = edcba, that is, when multiplied by 4 yields the same number read backwards. Write a C-program to find this number.

Problem 2. Write a C program to compute the matrix product of two matrices A and B.

Problem 3. Write a C program that outputs, in alphabetical order, all the distinct strings that use each of the characters 'c', 'a', 't', 'd', 'o', 'g' exactly once. How many strings does the program generate?

Problem 4. Write a C function that takes a positive integer n as argument and outputs a series of numbers according to the following process, until 1 is reached:

- If n is even, set n to n/2
- If n is odd, set n to 3*n+1

Problem 5. Define a data structure to store all information of a single ride with the Opal card. Here are two sample records:

Transaction number	Date/time	Mode	Details	Journey number	Fare Applied	Fare	Discount	Amount
642	Mon 24/07/2017 18:55	T	Central to Kings Cross	2	Off-peak	$3.46	$1.04	-$2.42
640	Mon 24/07/2017 09:50	B	Flinders St af Oxford St to Anzac Pde D opp UNSW	1		$1.43	$0.00	-$1.43

You may assume that individual stops (such as "Anzac Pde D opp UNSW") require no more than 31 characters.

Determine the memory requirements of your data structure, assuming that each integer and floating point number takes 4 bytes.

If you want to store millions of records, how would you improve your data structure?

Problem 6. The Fibonacci numbers are defined as follows:

Fib(1) = 1
Fib(2) = 1
Fib(n) = Fib(n-1)+Fib(n-2) for $n \geqslant 3$

Write a C program fibonacci.c that applies the process described in Q4 to the first 10 Fibonacci numbers. The output of the program should begin with

Fib[1] = 1
1
Fib[2] = 1
1
Fib[3] = 2
2

1
Fib[4] = 3
3
10
5
16
8
4
2
1

Problem 7. Write a C function that takes 3 integers as arguments and returns the largest of them. Your C function cannot use any control construct.

Problem 8. Write a C program that takes a sequence of integers from the keyboard, sorts them, and displays the sorted sequence on the screen, one integer per line. A non-integer indicates the end of sequence.

Problem Set 1-a Solutions

Problem 1. There is a 5-digit number that satisfies 4*abcde = edcba, that is, when multiplied by 4 yields the same number read backwards. Write a C-program to find this number.

Solution:

```
#include <stdio.h>
#define MIN 10000
#define MAX 24999    // solution has to be <25000
int main(void) {
  int a, b, c, d, e, n;
  for (n = MIN; n <= MAX; n++) {
    a = (n / 10000) % 10;
    b = (n / 1000) % 10;
    c = (n / 100) % 10;
    d = (n / 10) % 10;
    e = n % 10;
    if (4*n == 10000*e + 1000*d + 100*c + 10*b + a) {
      printf("%d\n", n);
    }
  }
  return 0;
}
```

Problem 2. Write a C program to compute the matrix product of two matrices A and B.

Solution:

```
#include <stdio.h>
#define M 4
#define N 4
#define P 4

// Function matrixProduct  computes a[][]*b[][], and stores the result in c[][]

 void matrixProduct(float a[M][N], float b[N][P], float c[M][P]) {
  int i, j, k;
  for (i = 0; i < M; i++) {
    for (j = 0; j < P; j++) {
          c[i][j] = 0.0;
          for (k = 0; k < N; k++) {
            c[i][j] += a[i][k] * b[k][j];
          }
    }
```

```
  }
}

int main()
{
   float a[M][N] = { {1, 1, 1, 1}, {2, 2, 2, 2},   {3, 3, 3, 3},  {4, 4, 4, 4}};
   float b[N][P] = { {1, 1, 1, 1}, {2, 2, 2, 2},  {3, 3, 3, 3}, {4, 4, 4, 4}};
   float c[M][P]; // To store result
   int i, j;

   matrixProduct(a, b, c);
   printf("Result matrix is \n");
   for (i = 0; i < M; i++)
     {
        for (j = 0; j < P; j++)
          printf("%f ", c[i][j]);
        printf("\n");
     }
   return 0;
}
```

Problem 3. Write a C program that outputs, in alphabetical order, all the distinct strings that use each of the characters 'c', 'a', 't', 'd', 'o', 'g' exactly once. How many strings does the program generate?

Solution:

```
#include <stdio.h>
int main(void) {
  char catdog[] = { 'a','c','d','g','o','t' };

  int count = 0;
  int i, j, k, l, m, n;
  for (i=0; i<6; i++)
    for (j=0; j<6; j++)
          for (k=0; k<6; k++)
            for (l=0; l<6; l++)
              for (m=0; m<6; m++)
                    for (n=0; n<6; n++)
                      if (i!=j && i!=k && i!=l && i!=m && i!=n &&
                            j!=k && j!=l && j!=m && j!=n &&
                            k!=l && k!=m && k!=n &&
                            l!=m && l!=n && m!=n) {
                        printf("%c%c%c%c%c%c\n", catdog[i], catdog[j],
                                                 catdog[k], catdog[l],
                                                 catdog[m], catdog[n]);
                        count++;
```

```
                    }
  printf("%d\n", count);
  return 0;
}
```

Problem 4. Write a C function that takes a positive integer n as argument and outputs a series of numbers according to the following process, until 1 is reached:

- If n is even, set n to n/2
- If n is odd, set n to 3*n+1

Solution:

```
void collatz(int n) {  // named after the German mathematician who invented this problem
  printf("%d\n", n);
  while (n != 1) {
    if (n % 2 == 0) {
          n = n / 2;
    } else {
          n = 3*n + 1;
    }
    printf("%d\n", n);
  }
}
```

Problem 5. Define a data structure to store all information of a single ride with the Opal card. Here are two sample records:

Transaction number	Date/time	Mode	Details	Journey number	Fare Applied	Fare	Discount	Amount
642	Mon 24/07/2017 18:55	T	Central to Kings Cross	2	Off-peak	$3.46	$1.04	-$2.42
640	Mon 24/07/2017 09:50	B	Flinders St af Ox-ford St to Anzac Pde D opp UNSW	1		$1.43	$0.00	-$1.43

You may assume that individual stops (such as "Anzac Pde D opp UNSW") require no more than 31 characters.

Determine the memory requirements of your data structure, assuming that each integer and floating point number takes 4 bytes.

If you want to store millions of records, how would you improve your data structure?

Solution:

```
typedef struct {
  int day, month, year;
} DateT;

typedef struct {
  int hour, minute;
} TimeT;

typedef struct {
  int   transaction;
  char  weekday[4];       // 3 chars + terminating '\0'
  DateT date;
  TimeT time;
  char  mode;             // 'B', 'F' or 'T'
  char  from[32], to[32];
  int   journey;
  char  faretext[12];
  float fare, discount, amount;
} JourneyT;
```

Memory requirement for one element of type JourneyT: 4 + 4 + 12 + 8 + 1 (+ 3 padding) + 2·32 + 4 + 12 + 3·4 = 124 bytes.

The data structure can be improved in various ways: encode both origin and destination (from and to) using Sydney Transport's unique stop IDs along with a lookup table that links e.g. 203311 to "Anzac Pde Stand D at UNSW"; use a single integer to encode the possible "Fare Applied" entries; avoid storing redundant information like the weekday, which can be derived from the date itself.

Problem 6. The Fibonacci numbers are defined as follows:

Fib(1) = 1
Fib(2) = 1
Fib(n) = Fib(n-1)+Fib(n-2) for n⩾3

Write a C program fibonacci.c that applies the process described in Q4 to the first 10 Fibonacci numbers. The output of the program should begin with

```
Fib[1] = 1
1
Fib[2] = 1
1
Fib[3] = 2
2
1
Fib[4] = 3
3
10
5
16
```

8
4
2
1

Solution:

```
#include <stdio.h>
#define MAX 10

void collatz(int n) {  // named after the German mathematician who invented this problem
  printf("%d\n", n);
  while (n != 1) {
    if (n % 2 == 0) {
          n = n / 2;
    } else {
          n = 3*n + 1;
    }
    printf("%d\n", n);
  }
}

int main(void) {
  int fib[MAX] = { 1, 1 };     // initialise the first two numbers
  int i;
  for (i = 2; i < MAX; i++) {  // compute the first 10 Fibonacci numbers
    fib[i] = fib[i-1] + fib[i-2];
  }

  for (i = 0; i < MAX; i++) {  // apply Collatz's process to each number
    printf("Fib[%d] = %d\n", i+1, fib[i]);
    collatz(fib[i]);
  }

  return 0;
}
```

Problem 7. Write a C function that takes 3 integers as arguments and returns the largest of them. Your C function cannot use any control construct.

Solution:

```
int max(int a, int b, int c) {
  int d = a * (a >= b) + b * (a < b);   // d is max of a and b
  return c * (c >= d) + d * (c < d);   // return max of c and d
}
```

Problem 8. Write a C program that takes a sequence of integers from the keyboard, sorts them, and displays the sorted sequence on the screen, one integer per line. A non-integer indicates the end of sequence.

Solution:

```
#include <stdio.h>
#define SIZE 250

void insertionSort(int array[], int n) {
  int i;
  for (i = 1; i < n; i++) {
    int element = array[i];              // for this element ...
    int j = i-1;
    while (j >= 0 && array[j] > element) {  // ... work down the ordered list
      array[j+1] = array[j];             // ... moving elements up
      j--;
    }
    array[j+1] = element;                // and insert in correct position
  }
}

int main(void) {
  int numbers[SIZE];
  int i, n=0;
  int done=1,rev;

 while (done) // Initialize the array numbers[] by receiving integers from keyboard
  {
    if (n==SIZE-1)
     break;
    printf("Type in a number \n");
    rev=scanf("%d", &numbers[n]);
    printf("numbers[%d]=%d\n", n, numbers[n]);
    if (rev<=0) // not an integer
       done=0;
    else n++;
  }
  insertionSort(numbers, n);
  for (i = 0; i <n; i++)
    printf("%d\n", numbers[i]);

  return 0;
}
```

Problem Set 1-b

Problem 1. Modify the stack implementation in the lecture notes (Stack.h and Stack.c) to implement a stack of integers.

Problem 2. Write a test program for your stack code in **Q**1 that does the following:

- initialise the stack
- prompt the user to input a number *n*
- check that *n* is a positive number
- prompt the user to input *n* numbers and push each number onto the stack
- use the stack to output the *n* numbers in reverse order

An example of the program executing could be

```
Enter a positive number: 3
Enter a number: 2017
Enter a number: 12
Enter a number: 24
24
12
2017
```

Problem 3. Modify your program in **Q**2 so that it takes the *n* numbers from the command line. An example of the program execution could be

```
prompt$./tester 2017 12 24
24
12
2017
```

Problem 4. A stack can be used to convert a positive decimal number *n* to a different numeral system with base *k* according to the following algorithm:

while n>0 **do**
  push *n*%*k* onto the stack
  n = *n* / *k*
end while

The result can be displayed by printing the numbers as they are popped off the stack. Example (*k*=2):

```
n = 13        --> push 1 (= 13%2)
n = 6  (= 13/2) --> push 0 (= 6%2)
n = 3  (=  6/2) --> push 1 (= 3%2)
n = 1  (=  3/2) --> push 1 (= 1%2)
n = 0  (=  1/2)
Result: 1101
```

Using your stack code in Q1, write a C program to implement this algorithm to convert to base *k*=2 a number given on the command line. Design a Makefile to compile this program along with the integer stack implementation.

An example of program compilation and execution could be

```
prompt$ make
gcc -Wall -Werror -c binary.c
gcc -Wall -Werror -c IntStack.c
gcc -o binary binary.o IntStack.o
./binary 13
1101
./binary 128
10000000
./binary 127
1111111
```

Problem 5. Implement a queue of integers in C using an array to store all the integers. All the function prototypes of the integer queue are defined in IntQueue.h as follows:

```
// Integer queue header file
void queueInit();       // set up an empty queue
int  isEmpty();     // check whether the queue is empty
void enqueue(int);  // insert int at the end of queue
int  dequeue();     // remove int from the front of queue
```

Problem 6. Given the following definition:

```
int data[12] = {5, 3, 6, 2, 7, 4, 9, 1, 8};
```

and assuming that *&data[0] == 0x10000*, what are the values of the following expressions?

data + 4
*data + 4
*(data + 4)
data[4]
*(data + *(data + 3))
data[data[2]]

Problem 7. Consider the following piece of C code:

```
typedef struct {
  int   studentID;
  int   age;
  char  gender;
  float WAM;
} PersonT;

PersonT per1;
PersonT per2;
PersonT *ptr;

ptr = &per1;
per1.studentID = 3141592;
ptr->gender = 'M';
ptr = &per2;
ptr->studentID = 2718281;
ptr->gender = 'F';
per1.age = 25;
per2.age = 24;
ptr = &per1;
per2.WAM = 86.0;
ptr->WAM = 72.625;
```

What are the values of the fields in the *per1* and *per2* record after execution of the above statements?

Note that ptr->t means the same as (*ptr).t

Problem 8. Write a C program that takes 1 command line argument and prints all its *prefixes* in decreasing order of length. You are *not* permitted to use any library functions other than printf(). You are also *not* permitted to use any array other than argv[].

An example of the program execution could be

```
prompt$ ./prefixes Programming
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
```

Problem Set 1-b Solutions

Problem 1. Modify the stack implementation in the lecture notes (Stack.h and Stack.c) to implement a stack of integers.

Solution:

IntStack.h

```
// Integer Stack ADO header file
void StackInit();     // set up empty stack
int  StackIsEmpty();  // check whether stack is empty
void StackPush(int);  // insert int on top of stack
int  StackPop();      // remove int from top of stack
```

IntStack.c

```
// Integer Stack ADO implementation
#include "IntStack.h"
#include <assert.h>

#define MAXITEMS 10

static struct {
   int item[MAXITEMS];
   int top;
} stackObject;  // defines the Data Object

void StackInit() {        // set up empty stack
   stackObject.top = -1;
}

int StackIsEmpty() {     // check whether stack is empty
   return (stackObject.top < 0);
}

void StackPush(int n) {  // insert int on top of stack
   assert(stackObject.top < MAXITEMS-1);
   stackObject.top++;
   int i = stackObject.top;
   stackObject.item[i] = n;
}
```

```
int StackPop() {          // remove int from top of stack
   assert(stackObject.top > -1);
   int i = stackObject.top;
   int n = stackObject.item[i];
   stackObject.top--;
   return n;
}
```

Problem 2. Write a test program for your stack code in **Q**1 that does the following:

- initialise the stack
- prompt the user to input a number *n*
- check that *n* is a positive number
- prompt the user to input *n* numbers and push each number onto the stack
- use the stack to output the *n* numbers in reverse order

An example of the program executing could be

```
Enter a positive number: 3
Enter a number: 2017
Enter a number: 12
Enter a number: 24
24
12
2017
```

Solution:

```
#include <stdio.h>
#include "IntStack.h"

int main(void) {
   int i, n, number;

   StackInit();

   printf("Enter a positive number: ");
   if (scanf("%d", &n) == 1 && (n > 0)) {   // test if scanf
successful and returns positive number
```

```
        for (i = 0; i < n; i++) {
        printf("Enter a number: ");
        scanf("%d", &number);
        StackPush(number);
        }
        while (!StackIsEmpty()) {
        printf("%d\n", StackPop());
        }
    }
    return 0;
}
```

Problem 3. Modify your program in **Q**2 so that it takes the *n* numbers from the command line. An example of the program execution could be

```
prompt$./tester 2017 12 24
24
12
2017
```

Solution:

```
#include <stdlib.h>
#include <stdio.h>
#include "IntStack.h"

int main(int argc, char *argv[]) {
    int i;

    StackInit();
    for (i = 1; i < argc; i++) {
        StackPush(atoi(argv[i]));
    }
    while (!StackIsEmpty()) {
        printf("%d\n", StackPop());
    }
    return 0;
}
```

Problem 4. A stack can be used to convert a positive decimal number *n* to a different numeral system with base *k* according to the following algorithm:

while n>0 **do**
  push *n*%*k* onto the stack
  n = *n* / *k*
end while

The result can be displayed by printing the numbers as they are popped off the stack. Example (*k*=2):

```
n = 13        --> push 1 (= 13%2)
n = 6  (= 13/2) --> push 0 (= 6%2)
n = 3  (=  6/2) --> push 1 (= 3%2)
n = 1  (=  3/2) --> push 1 (= 1%2)
n = 0  (=  1/2)
Result: 1101
```

Using your stack code in Q1, write a C program to implement this algorithm to convert to base *k*=2 a number given on the command line. Design a Makefile to compile this program along with the integer stack implementation.

An example of program compilation and execution could be

```
prompt$ make
gcc -Wall -Werror -c binary.c
gcc -Wall -Werror -c IntStack.c
gcc -o binary binary.o IntStack.o
./binary 13
1101
./binary 128
10000000
./binary 127
1111111
```

Solution:

```
#include <stdlib.h>
#include <stdio.h>
#include "IntStack.h"
```

```
int main(int argc, char *argv[]) {
   int n;

   if (argc != 2) {
      printf("Usage: %s number\n", argv[0]);
      return 1;
   }

   StackInit();
   n = atoi(argv[1]);
   while (n > 0) {
      StackPush(n % 2);
      n = n / 2;
   }
   while (!StackIsEmpty()) {
      printf("%d", StackPop());
   }
   putchar('\n');
   return 0;
}
```

Makefile

```
binary : binary.o IntStack.o
     gcc -o binary binary.o IntStack.o

binary.o : binary.c IntStack.h
     gcc -Wall -Werror -c binary.c

IntStack.o : IntStack.c IntStack.h
     gcc -Wall -Werror -c IntStack.c
```

Problem 5. Implement a queue of integers in C using an array to store all the integers. All the function prototypes of the integer queue are defined in IntQueue.h as follows:

```
// Integer queue header file
void queueInit();       // set up an empty queue
int  isEmpty();    // check whether the queue is empty
void enqueue(int);  // insert int at the end of queue
int  dequeue();    // remove int from the front of queue
```

Solution:

IntQueue.c

```
// Integer Queue ADO implementation
#include "IntQueue.h"
#include <assert.h>

#define MAXITEMS 10
static struct {
   int item[MAXITEMS];
   int top;
} queueObject;  // defines the Data Object

void QueueInit() {          // set up empty queue
   queueObject.top = -1;
}

int QueueIsEmpty() {        // check whether queue is empty
   return (queueObject.top < 0);
}

void QueueEnqueue(int n) {  // insert int at end of queue
   assert(queueObject.top < MAXITEMS-1);
   queueObject.top++;
   int i;
   for (i = queueObject.top; i > 0; i--) {
      queueObject.item[i] = queueObject.item[i-1]; // move all
elements up
   }
   queueObject.item[0] = n; // add element at end of queue
}

int QueueDequeue() {        // remove int from front of queue
   assert(queueObject.top > -1);
   int i = queueObject.top;
   int n = queueObject.item[i];
   queueObject.top--;
   return n;
}
```

Problem 6. Given the following definition:

```
int data[12] = {5, 3, 6, 2, 7, 4, 9, 1, 8};
```

and assuming that *&data[0] == 0x10000*, what are the values of the following expressions?

data + 4
*data + 4
*(data + 4)
data[4]
*(data + *(data + 3))
data[data[2]]

Solution:

data + 4	== 0x10000 + 4 * 4 bytes == 0x10010
*data + 4	== data[0] + 4 == 5 + 4 == 9
*(data + 4)	== data[4] == 7
data[4]	== 7
*(data + *(data + 3))	== *(data + data[3]) == *(data + 2) == data[2] == 6
data[data[2]]	== data[6] == 9

Problem 7. Consider the following piece of C code:

```
typedef struct {
  int   studentID;
  int   age;
  char  gender;
  float WAM;
} PersonT;

PersonT per1;
PersonT per2;
PersonT *ptr;
```

```
ptr = &per1;
per1.studentID = 3141592;
ptr->gender = 'M';
ptr = &per2;
ptr->studentID = 2718281;
ptr->gender = 'F';
per1.age = 25;
per2.age = 24;
ptr = &per1;
per2.WAM = 86.0;
ptr->WAM = 72.625;
```

What are the values of the fields in the *per1* and *per2* record after execution of the above statements?

Note that ptr->t means the same as (*ptr).t

Solution:

per1.studentID	== 3141592
per1.age	== 25
per1.gender	== 'M'
per1.WAM	== 72.625
per2.studentID	== 2718281
per2.age	== 24
per2.gender	== 'F'
per2.WAM	== 86.0

Problem 8. Write a C program that takes 1 command line argument and prints all its *prefixes* in decreasing order of length. You are *not* permitted to use any library functions other than printf(). You are also *not* permitted to use any array other than argv[].

An example of the program execution could be

```
prompt$ ./prefixes Programming
Programming
```

```
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
```

Solution:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   char *start, *end;

   if (argc == 2) {
      start = argv[1];
      end = argv[1];
      while (*end != '\0') {    // find address of terminating
'\0'
         end++;
      }
      while (start != end) {
         printf("%s\n", start); // print string from start to '\0'
         end--;                 // move end pointer up
         *end = '\0';           // overwrite last char by '\0'
      }
   }
   return 0;
}
```

Problem Set 2

Problem 1. Consider the following function:

```
/* Makes an array of 10 integers and returns a pointer to it */

int *makeArrayOfInts(void) {
  int arr[10];
  int i;
  for (i=0; i<10; i++) {
    arr[i] = i;
  }
  return arr;
}
```

Explain what is wrong with this function. Rewrite the function so that it correctly achieves the intended result using malloc().

Problem 2. Consider the following program:

```
#include <stdio.h>
#include <stdlib.h>

void func(int *a) {
  a = malloc(sizeof(int));
}

int main(void) {
  int *p;
  func(p);
  *p = 6;
  printf("%d\n",*p);
  free(p);
  return 0;
}
```

Explain what is wrong with this program.

Problem 3. Write a C-program that uses a dynamic array of unsigned long long int numbers (8 bytes, only positive numbers) to compute the *n*'th Fibonacci number, where *n* is given as command line argument. For example, **./fib 60** should result in 1548008755920.

Hint: The placeholder %lld (instead of %d) can be used to print an unsigned long long int. Remember that the Fibonacci numbers are defined as Fib(1) = 1, Fib(2) = 1 and Fib(*n*) = Fib(*n*-1)+Fib(*n*-2) for *n*≥3.

Problem 4. Describe in words how you would implement a *queue* ADT using a dynamic linked list. Which of the functions for the linked list implementation of a stack from the lecture need to be changed, and how?

Problem 5. Suppose that you have a stack *S* containing *n* elements and a queue *Q* that is initially empty. Describe how you can use *Q* to scan *S* in order to check if it contains a certain element *x*, with the additional constraint that your algorithm must return the elements back to *S* in their original order. You should **not** use an additional array or linked list — only use *S* and *Q*.

Problem 6. Write a C-program called **llbuild.c** that builds a linked list of integers from user input. The program works as follows:

- starts with an empty linked list called *all* (say), initialised to NULL
- prompts the user with the message "Enter a number: "
- makes a linked list node called *new* from user's response
- appends *new* to *all*
- asks for more user input and repeats the cycle
- the cycle is terminated when the user enters any non-numeric character
- on termination, the program generates the message "Finished. List is " followed by the contents of the linked list in the format shown below.

A sample interaction is shown as follows:

```
prompt$ ./llbuild
Enter an integer: 12
Enter an integer: 34
Enter an integer: 56
Enter an integer: quit
Finished. List is 12->34->56
```

Note that any non-numeric data 'finishes' the interaction. If the user provides no data, then no list should be output:

```
prompt$ ./llbuild
Enter an integer:#
Finished.
```

Problem 7. Extend the C-program in Q6 to split the linked list in two equally-sized halves and output the result. If the list has an odd number of elements, then the first list should contain one more element than the second.

Note that:

- your algorithm should be 'in-place' (so you are not permitted to create a second linked list or use some other data structure such as an array);
- you should not traverse the list more than once (e.g. to count the number of elements and then restart from the beginning).

An example of the program executing could be

```
prompt$ ./llsplit
Enter an integer: 421
Enter an integer: 456732
Enter an integer: 321
Enter an integer: 4
Enter an integer: 86
Enter an integer: 89342
Enter an integer: 9
Enter an integer: #
Finished. List is 421->456732->321->4->86->89342->9
First half is 421->456732->321->4
Second half is 86->89342->9
```

Problem Set 2 Solutions

Problem 1. Consider the following function:

```
/* Makes an array of 10 integers and returns a pointer to it */

int *makeArrayOfInts(void) {
  int arr[10];
  int i;
  for (i=0; i<10; i++) {
    arr[i] = i;
  }
  return arr;
}
```

Explain what is wrong with this function. Rewrite the function so that it correctly achieves the intended result using malloc().

Solution:

The function is erroneous because the array arr will cease to exist after the line return arr, since arr is local to this function and gets destroyed once the function returns. So the caller will get a pointer to something that doesn't exist anymore, and you will start to see garbage, segmentation faults, and other errors.

Arrays created with malloc() are stored in a separate place in memory, the heap, which ensures they live on indefinitely until you free them yourself.

The correctly implemented function is as follows:

```
int *makeArrayOfInts() {
  int *arr = malloc(sizeof(int) * 10);
  int i;
  for (i=0; i<10; i++) {
    arr[i] = i;
  }
  return arr;  // this is fine because the array itself will live on
}
```

Problem 2. Consider the following program:

```
#include <stdio.h>
#include <stdlib.h>

void func(int *a) {
  a = malloc(sizeof(int));
}

int main(void) {
  int *p;
  func(p);
  *p = 6;
  printf("%d\n",*p);
  free(p);
  return 0;
}
```

Explain what is wrong with this program.

Solution:

The program is not valid because func() makes a *copy* of the pointer p. So when malloc() is called, the result is assigned to the copied pointer rather than to p. Pointer p itself is pointing to random memory (e.g., 0x0000) before and after the function call. Hence, when you dereference it, the program will (likely) crash.

If you want to use a function to add a memory address to a pointer, then you need to pass the *address* of the pointer (i.e. a pointer to a pointer, or "double pointer"):

```
void func(int **a) {
  a = malloc(sizeof(int));
  *a = malloc(sizeof(int));
}

int main(void) {
  int *p;

  func(&p);
  *p = 6;
```

```
  printf("%d\n",*p);
  free(p);
  return 0;
}
```

Problem 3. Write a C-program that uses a dynamic array of unsigned long long int numbers (8 bytes, only positive numbers) to compute the *n*'th Fibonacci number, where *n* is given as command line argument. For example, **./fib 60** should result in 1548008755920.

Hint: The placeholder %lld (instead of %d) can be used to print an unsigned long long int. Remember that the Fibonacci numbers are defined as Fib(1) = 1, Fib(2) = 1 and Fib(*n*) = Fib(*n*-1)+Fib(*n*-2) for *n*≥3.

Solution:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s number\n", argv[0]);
    return 1;
  }

  int n = atoi(argv[1]);
  if (n > 0) {
    unsigned long long int *arr = malloc(n * sizeof(unsigned long long int));
    arr[0] = 1;
    arr[1] = 1;
    int i;
    for (i = 2; i < n; i++) {
        arr[i] = arr[i-1] + arr[i-2];
    }
    printf("%lld\n", arr[n-1]);
    free(arr);              // don't forget to free the array
  }
  return 0;
}
```

Problem 4. Describe in words how you would implement a *queue* ADT using a dynamic linked list. Which of the functions for the linked list implementation of a stack from the lecture need to be changed, and how?

Solution:

In the stack ADT, elements are added to ("push") and removed from ("pop") the beginning of the linked list. For a queue, we have two options: either we add ("enqueue") new elements at the end and continue to take elements off ("dequeue") from the beginning. Or we continue to add elements at the beginning and dequeue from the end. Operating on both ends will be more efficient if we use a datastructure with two pointers: one pointing to the first and one pointing to the last element of a list.

Problem 5. Suppose that you have a stack *S* containing *n* elements and a queue *Q* that is initially empty. Describe how you can use *Q* to scan *S* in order to check if it contains a certain element *x*, with the additional constraint that your algorithm must return the elements back to *S* in their original order. You should **not** use an additional array or linked list — only use *S* and *Q*.

Solution:

The solution is to use the queue Q to process the elements in two phases. In the first phase, we iteratively pop all the elements from S and enqueue them in Q, then dequeue the elements from Q and push them back onto S. As a result, all the elements are now in reversed order on S. In the second phase, we again pop all the elements from S, but this time we also look for the element x. By again passing the elements through Q and back onto S, we reverse the reversal, thereby restoring the original order of the elements on S.

Problem 6. Write a C-program called **llbuild.c** that builds a linked list of integers from user input. The program works as follows:

- starts with an empty linked list called *all* (say), initialised to NULL
- prompts the user with the message "Enter a number: "
- makes a linked list node called *new* from user's response
- appends *new* to *all*
- asks for more user input and repeats the cycle
- the cycle is terminated when the user enters any non-numeric character
- on termination, the program generates the message "Finished. List is " followed by the contents of the linked list in the format shown below.

A sample interaction is shown as follows:

prompt$. /llbuild

Enter an integer: **12**

Enter an integer: **34**

Enter an integer: **56**

Enter an integer: **quit**

```
Finished. List is 12->34->56
```

Note that any non-numeric data 'finishes' the interaction. If the user provides no data, then no list should be output:

```
prompt$. /llbuild
Enter an integer:#
Finished.
```

Solution:

```
// llbuild.c: create a linked list from user input, and print
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node {
  int data;
  struct node *next;
} NodeT;

NodeT *joinLL(NodeT *head1, NodeT *head2) {
// either or both head1 and head2 may be NULL
  if (head1 == NULL) {
    head1 = head2;
  } else {
    NodeT *p = head1;
    while (p->next != NULL) {
      p = p->next;
    }
    p->next = head2; // this does nothing if head2 == NULL
  }
  return head1;
}

NodeT *makeNode(int v) {
  NodeT *new = malloc(sizeof(NodeT));
```

```
  assert(new != NULL);
  new->data = v;
  new->next = NULL;
  return new;
}

void showLL(NodeT *list) {
  NodeT *p;
  for (p = list; p != NULL; p = p->next) {
    printf("%d", p->data);
    if (p->next != NULL)
      printf("->");
  }
  putchar('\n');
}

void freeLL(NodeT *list) {
   NodeT *p = list;
   while (p != NULL) {
      NodeT *temp = p->next;
      free(p);
      p = temp;
   }
}

int main(void) {
  NodeT *all = NULL;
  int data;

  printf("Enter an integer: ");
  while (scanf("%d", &data) == 1) {
    NodeT *new = makeNode(data);
    all = joinLL(all, new);
    printf("Enter an integer: ");
  }
  if (all != NULL) {
    printf("Finished. List is ");
```

```
    showLL(all);
    freeLL(all);
  } else {
    printf("Finished.\n");
  }
  return 0;
}
```

Problem 7. Extend the C-program in Q6 to split the linked list in two equally-sized halves and output the result. If the list has an odd number of elements, then the first list should contain one more element than the second.

Note that:

- your algorithm should be 'in-place' (so you are not permitted to create a second linked list or use some other data structure such as an array);
- you should not traverse the list more than once (e.g. to count the number of elements and then restart from the beginning).

An example of the program executing could be

```
prompt$ ./llsplit
Enter an integer: 421
Enter an integer: 456732
Enter an integer: 321
Enter an integer: 4
Enter an integer: 86
Enter an integer: 89342
Enter an integer: 9
Enter an integer: #
Finished. List is 421->456732->321->4->86->89342->9
First half is 421->456732->321->4
Second half is 86->89342->9
```

Solution:

The following solution uses a "slow" and a "fast" pointer to traverse the list. The fast pointer always jumps 2 elements ahead. At any time, if slow points to the i^{th} element, then fast points to

the $2 \cdot i^{th}$ element. Hence, when the fast pointer reaches the end of the list, the slow pointer points to the last element of the first half.

```
NodeT *splitList(NodeT *head) {   // returns pointer to second half
  assert(head != NULL);

  NodeT *slow = head;
  NodeT *fast = head->next;

  while (fast != NULL && fast->next != NULL) {
    slow = slow->next;
    fast = fast->next->next;
  }
  NodeT *head2 = slow->next;    // this becomes head of second half
  slow->next = NULL;            // cut off at end of first half
  return head2;
}
```

Problem Set 3-a

Problem 1 Show that if p(n) is a polynomial in n, then log p(n) is O(log n).

Problem 2 Show that $\sum_{i=1}^{n} i^2$ is O(n^3).

Problem 3 Show that $\sum_{i=1}^{n} \frac{i}{2^i}$ is O(1).

Problem 4 Show that $\sum_{i=1}^{n} \log i$ is O(n log n).

Problem 5 Consider the following algorithm:

```
Algorithm Unknown(A, B)
  Input: Arrays A and B each storing n>0 integers.
  Output: The number of elements in B equal to the sum of prefix sums in A.

  c=0;
  for i=0 to n-1 do
   { s=0;
    for j=0 to n-1 do
      { s=s+A[0];
        for k=0 to n-1 do
          s=s+A[k];
      }
    if ( B[i]=s )
      c=c+1;
   }
  return c;
```

What is the time complexity of this algorithm in big-Oh notation?

Problem 6 Consider the following algorithm:

```
Algorithm MatrixMultiplication(A, B)
    Input: Matrices A[m,k] and B[k,n].
    Output: A matrix C= A*B.

    for i=0 to m-1 do
     {
      for j=0 to n-1 do
        { c[i,j]=0;
          for p=0 to k-1 do
            c[i,j]+=A[i,p]*B[p,j];
        }
     }
    return C;
```

What is the time complexity of this algorithm in big-Oh notation?

Problem 7 Let p(x) be a polynomial of degree n, where $p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$. Design an O(n)-time algorithm for computing p(x).

Problem 8 The Tower of Hanoi is a classical problem which can be solved by recurrence. There are three pegs and N disks of different sizes. Originally, all the disks are on the left peg, stacked in

decreasing size from bottom to top. Our goal is to transfer all the disks to the right peg, and the rules are that we can only move one disk at a time, and no disk can be moved onto a smaller one. We can easily solve this problem with the following recursive algorithm: If N = 1, move this disk directly to the right peg and we are done. Otherwise (N >1), first transfer the top N-1 disks to the middle peg applying the algorithm recursively, then move the largest disk to the right peg, and finally transfer the N-1 disks on the middle peg to the right peg applying the algorithm recursively. Let T(N) be the total number of moves needed to transfer N disks. We have that T(1) = 1, and T(N) = 2T(N-1)+ 1. What is the time complexity of this algorithm in big-Oh notation?

Problem 9 The Towers of Providence is a variation of the classical Towers of Hanoi problem. There are four pegs, denoted A, B, C, and D, and N disks of different sizes. Originally, all the disks are on peg A, stacked in decreasing size from bottom to top. Our goal is to transfer all the disks to peg D, and the rules are that we can only move one disk at a time, and no disk can be moved onto a smaller one.

We can solve this problem with a recursive algorithm: If N = 1,move this disk directly to peg D, and we are done. Otherwise (N > 1), perform the following steps:

1. transfer the top N-2 disks on peg A to peg B applying the algorithm recursively;
2. move the second largest disk from peg A to peg C;
3. move the largest disk from peg A to peg D;
4. move the second largest disk from peg C to peg D;
5. transfer the N-2 disks on peg B to peg D applying the algorithm recursively.

Let T(N) be the total number of moves needed to transfer N disks. We have:
T(1) = 1; T(N) = 2T(N-2) + 3: What is the time complexity of this algorithm in big-Oh notation?

Problem Set3-a Solutions

Problem 1 Show that if p(n) is a polynomial in n, then log p(n) is O(log n).

Solution: Let $p(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$
$p(n) < \max\{|a_m|, |a_{m-1}|, \ldots, |a_0|\} (m+1)\, n^m$
$\log p(n) < \log(\max\{|a_m|, |a_{m-1}|, \ldots, |a_0|\} (m+1)) + m \log n$
Since m and a_i (i=0, 1, …, m) are constants, we have $O(\log p(n)) = O(\log(\max\{|a_m|, |a_{m-1}|, \ldots, |a_0|\}) (m+1) + O(m \log n) = O(\log n)$.

Problem 2 Show that $1^2 + 2^2 + \ldots + n^2$ is $O(n^3)$.

Solution 1: The area of the shape enclosed by the x-axis, the vertical line x=0, the vertical line x=n+1, and the curve $y=x^2$, is given by $\int_0^{n+1} x^2\, dx$. This shape contains every rectangle formed by the four lines x=i, x=i+1, y=0 and $y=i^2$ (i=1, 2, …, n). Therefore, we have the following inequality:

$$\sum_{i=1}^{n} i^2 < \int_0^{n+1} x^2 dx$$
$$\int_0^{n+1} x^2 dx = \frac{(n+1)^3}{3} = O(n^3)$$

Solution 2: $1^2 + 2^2 + \ldots + n^2 = n(n+1)(2n+1)/6 = n^3/3 + n^2/2 + n/6 = O(n^3)$.

Problem 3 Show that $\sum_{i=1}^{n} \frac{i}{2^i}$ is O(1).

Solution: Let $S = \sum_{i=1}^{n} \frac{i}{2^i}$. $s = \sum_{i=1}^{n} \frac{i}{2^i} = \sum_{i=1}^{n} \frac{1}{2^i} + \sum_{i=2}^{n} \frac{i-1}{2^i} = \sum_{i=1}^{n} \frac{1}{2^i} + \sum_{i=1}^{n-1} \frac{i}{2^{i+1}} < 1 + s/2$.
Therefore, S<2. As a result, $\sum_{i=1}^{n} \frac{i}{2^i}$ is O(1).
Comments: It is easy to prove $\sum_{i=1}^{n} \frac{1}{2^i} < 1$. Let $A = \sum_{i=1}^{n} \frac{1}{2^i}$. $2A = \sum_{i=1}^{n} \frac{2}{2^i} = \sum_{i=1}^{n} \frac{1}{2^{i-1}} = 1 + \sum_{i=1}^{n-1} \frac{1}{2^i}$
<1+A. Threfore, A<1.

Problem 4 Show that $\sum_{i=1}^{n} \log i$ is O(n log n).

Solution: $\sum_{i=1}^{n} \log i < n \log n = O(n \log n)$

Problem 5 Consider the following algorithm:

```
Algorithm Unknown(A, B)
  Input: Arrays A and B each storing n>0 integers.
  Output: The number of elements in B equal to the sum of prefix sums in A.

  c=0;
  for i=0 to n-1 do
   { s=0;
    for j=0 to n-1 do
      { s=s+A[0];
        for k=0 to n-1 do
          s=s+A[k];
      }
    if ( B[i]=s )
      c=c+1;
```

```
  }
return c;
```

What is the time complexity of this algorithm in big-Oh notation?

Solution: The number of primitives of each statement is shown as follows.

Statements	Number of primitive operations of each statement
c=0;	1
for i=0 to n-1 do	n
{ s=0;	n
for j=0 to n-1 do	n^2
{ s=s+A[0];	n^2
for k=0 to n-1 do	n^3
s=s+A[k];	n^3
}	
if (B[i]=s)	n
c=c+1;	n
}	
return c;	1

The total number of primitives is $2n^3+2n^2+4n+2=O(n^3)$.

Problem 6 Consider the following algorithm:

```
Algorithm MatrixMultiplication(A, B)
    Input: Matrices A[m,k] and B[k,n].
    Output: A matrix C= A*B.

    for i=0 to m-1 do
     {
      for j=0 to n-1 do
        { c[i,j]=0;
          for p=0 to k-1 do
            c[i,j]+=A[i,p]*B[p,j];
        }
     }
    return C;
```

What is the time complexity of this algorithm in big-Oh notation?

Solution: The number of primitives of each statement is shown as follows.

Statements	Number of primitive operations of each statement
for i=0 to m-1 do	m
{	
for j=0 to n-1 do	m*n
{ s=0;	m*n
for p=0 to k-1 do	m*n*k
c[i,j]+=A[i,p]*B[p,j];	m*n*k
c[i,j]=s;	m*n
}	
}	
return C;	1

The total number of primitives is m + m*n + m*n + m*n*k + m*n*k + m*n=O(m*n*k).

Problem 7 Let p(x) be a polynomial of degree n, where $p(x)= a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$. Design an O(n)-time algorithm for computing p(x).

Solution: Rewrite p(x) as $p(x) = (\ldots((a_nx + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$, which can be recursively computed as follows: $p_0=a_n$, $p_{i+1}=p_i x+ a_i$ (i=0, 1, …, n-1). The algorithm is as follows:

Algorithm computingPloynomial(x, n, A[])

Input: A polynomial where all the coefficients are stored in a one-dimensional array A of size n+1 and x stores the value of the variable.
Output: The result of the polynomial for x.

```
p= A[n];
for (i=1; i<n+1; i++)
  {
    p=p*x+A[n-i];
  }
return p;
```

Problem 8 The Tower of Hanoi is a classical problem which can be solved by recurrence. There are three pegs and N disks of different sizes. Originally, all the disks are on the left peg, stacked in decreasing size from bottom to top. Our goal is to transfer all the disks to the right peg, and the rules are that we can only move one disk at a time, and no disk can be moved onto a smaller one. We can easily solve this problem with the following recursive algorithm: If N = 1, move this disk directly to the right peg and we are done. Otherwise (N >1), first transfer the top N-1 disks to the middle peg applying the algorithm recursively, then move the largest disk to the right peg, and finally transfer the N-1 disks on the middle peg to the right peg applying the algorithm recursively. Let T(N) be the total number of moves needed to transfer N disks. We have that T(1) = 1, and T(N) = 2T(N-1)+ 1. What is the time complexity of this algorithm in big-Oh notation?

Solution: $T(N) = 2(2T(N-2) + 1) + 1$
$= 4T(N-2) + 2 + 1$
$= 4(2T(N-3) + 1) + 2 + 1$
$= 8T(N-3) + 4 + 2 + 1$
$= \ldots$
$= 2^{(N-1)}T(1) + 2^{N-2} + \ldots + 2+1$
$= 2^{(N-1)}T(1) + 2^{N-1} -1$
$= 2^N - 1$

Therefore, the time complexity of this algorithm is $O(2^N)$.

Problem 9 The Towers of Providence is a variation of the classical Towers of Hanoi problem. There are four pegs, denoted A, B, C, and D, and N disks of different sizes. Originally, all the disks are on peg A, stacked in decreasing size from bottom to top. Our goal is to transfer all the disks to peg D, and the rules are that we can only move one disk at a time, and no disk can be moved onto a smaller one.

We can solve this problem with a recursive algorithm: If N = 1,move this disk directly to peg D, and we are done. Otherwise (N > 1), perform the following steps:

1. transfer the top N-2 disks on peg A to peg B applying the algorithm recursively;
2. move the second largest disk from peg A to peg C;
3. move the largest disk from peg A to peg D;
4. move the second largest disk from peg C to peg D;
5. transfer the N-2 disks on peg B to peg D applying the algorithm recursively.

Let T(N) be the total number of moves needed to transfer N disks. We have:
T(1) = 1; T(N) = 2T(N-2) + 3: What is the time complexity of this algorithm in big-Oh notation?

Solution: $T(N) = 2(2T(N-4) + 3) + 3$

$= 4T(N - 4) + 3*(2 + 1)$
$= 4(2T(N - 6) + 3) + 3*(2 + 1)$
$= 8T(N - 6) + 3*(4 + 2 + 1)$
$= 2^i\ T(N-2i)+3(2^{i-1} + \ldots + 2+1)$

Let N-2i =1. We have i=(N-1)/2. Therefore,

$T(N) = 2^{(N-1)/2}T(1) + 3*(2^{(N-1)/2-1} + \ldots + 2+1)$
$= 2^{(N-1)/2}T(1) + 3*(2^{(N-1)/2} - 1)$
$= 2^{(N-1)/2} + 3*2^{(N-1)/2} - 3$
$= 2^2*2^{(N-1)/2} - 3$
$= 2^{(N+3)/2} - 3$

Therefore, the time complexity of this algorithm is $O(2^{(N+3)/2} - 3) = O(2^N)$.

Problem Set 3-b

Problem 1 For each node v in a tree T, let pre(v) be the rank of v in the preorder traversal of T, post(v) the rank of in the postorder traversal of T, depth(v) the depth of v, and desc(v) the number of descendants of v, not counting v itself. Derive a formula defining post(v) in terms of desc(v), depth(v), and pre(v), for each node v in T.

Problem 2 Give an O(n)-time algorithm for computing the depths of all the nodes of a tree T, where n is the number of nodes of T.

Problem 3 Describe, in pseudo code, a nonrecursive algorithm for performing the preorder traversal on a binary tree in linear time.

Problem 4 Describe, in pseudo code, a nonrecursive algorithm for performing the inorder traversal on a binary tree in linear time.

Problem 5 Describe, in pseudo code, a nonrecursive algorithm for performing the postorder traversal on a binary tree in linear time.

Problem 6 In the inorder traversal on a binary tree, a node v_i is the immediate inorder predecessor of a node v_j, if v_j is visited immediately after v_i in the inorder traversal. Describe a linear time algorithm for finding the immediate inorder predecessor of a node.

Problem 7 Let T be a binary tree. Define a **Roman node** to be a node v in T, such that the number of descendants in v's left subtree differs from the number of nodes in v's right subtree by at most 5. Describe a linear-time algorithm for finding each node v of T, such that v is not a Roman node, but all of v's descendants are Roman nodes.

Problem 8 Let T be a binary tree with n nodes. Define the lowest common ancestor (LCA) of two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself). Given two nodes v and w, describe an efficient algorithm for finding the LCA of v and w. What is the running time of your algorithm?

Problem Set 3-b Solution

Problem 1 For each node v in a tree T, let pre(v) be the rank of v in the preorder traversal of T, post(v) the rank of in the postorder traversal of T, depth(v) the depth of v, and desc(v) the number of descendants of v, not counting v itself. Derive a formula defining post(v) in terms of desc(v), depth(v), and pre(v), for each node v in T.

Solution: In the postorder traversal, when a node v is visited, all its descendants have been visited and all its ancestors have not been visited. On the other hand, in the preorder traversal, when a node v is visited, all its descendants have not been visited and all its ancestors have been visited. Therefore, we have post(v)= pre(v)+ desc(v)- depth(v).

Problem 2 Give an O(n)-time algorithm for computing the depths of all the nodes of a tree T, where n is the number of nodes of T.

Solution:

```
Algorithm computingDepths(v, d)
   {
     v.depth=d;
     if  v has no child
         return;
     else
        for  each child v′ of v do
             computingDepths(v′, d+1));
     return;
   }
```

To compute the depth of each node, simply call computeDepth(root, 0). This algorithm visits each node only once and each visit takes constant time. Therefore, it's running time is O(n).

Problem 3 Describe, in pseudo code, a nonrecursive algorithm for performing the preorder traversal on a binary tree in linear time.

Solution: We simulate the execution of the recursive algorithm for the preorder traversal by using a stack. In the recursive preorder traversal, we visit the root first, and recursively traverse the left subtree and then the right subtree. Initially, the stack contains the root only. Each time, we pop a node from the stack, visit it and push its right child and then the left child onto the stack. The algorithm is shown in pseudo code as follows:

```
Algorithm preOrder(v)
  {
    if  v = null
        return;
    Create an empty stack S;
    S.push(v);
    while S is not empty
```

```
        { v=S.pop();
          visit(v);
          if hasRight(v)
             S.push(right(v));
          if  hasLeft(v)
             S.push(left(v));
         }
  }
```

Time complexity analysis: Creating an empty stack takes O(1) time. The non-recursive algorithm pushes each node onto the stack once, pops it from the stack once and visits it once. Therefore, the algorithm takes O(n) time.

Problem 4 Describe, in pseudo code, a nonrecursive algorithm for performing the inorder traversal on a binary tree in linear time.

Solution: We simulate the execution of the recursive algorithm for the inorder traversal by using a stack. In the recursive inorder traversal, we recursively traverse the left subtree, visit the root, and then traverse the right subtree.

The non-recursive algorithm is show in pseudo code as follows:

```
    Algorithm inOrderTraversal(v)
       {
            Create an empty stack S;
            while S is not empty or v ≠ null
                if  v ≠ null   // keep going left until v=null
                  {  S.push(v);
                     v=left(v);
                   }
                else
                   {  v= S.pop();  // v has no left child, so visit it
                      visit(v);
                      v=right(v); // go to the right child
                    }
         }
```

Time complexity analysis: Creating an empty stack takes O(1) time. The non-recursive algorithm pushes each node onto the stack once, pops it from the stack once and visits it once. Therefore, the algorithm takes O(n) time.

Problem 5 Describe, in pseudo code, a nonrecursive algorithm for performing the postorder traversal on a binary tree in linear time.

Solution: We simulate the execution of the recursive algorithm for the postorder traversal by using a stack. In the recursive postorder traversal, we recursively traverse the left subtree, then traverse the right subtree, and lastly visit the root.

We introduce a variable named lastNodeVisited to keep track of the last node visited. The non-recursive algorithm is show in pseudo code as follows:

```
Algorithm postOrderTraversal(v)
    {
        Create an empty stack S;
        lastNodeVisited =null;
        while  S is not empty or v ≠ null
           if  v ≠ null   // push v and each left descendant onto the stack
             {   s.push(v);
                 v=left(v);
             }
           else
              { topNode=S.top();
                // S.top() returns the top node on the stack without removing it
                // if topNode has a right child not visited before,
                // then move to the right child of topNode
                if  topNode.right ≠ null and lastNodeVisited ≠ topNode.right
                    v=topNode.right;
                else // both left and right subtrees of topNode has been traversed
                    { visit(topNode);
                      lastNodeVisited = S.pop();
                    }
              }
       }
```

Time complexity analysis: Creating an empty stack takes O(1) time. The non-recursive algorithm pushes each node onto the stack once, pops it from the stack once peeks it (S.top()) once, and visits it once. Therefore, the algorithm takes O(n) time.

Problem 6 In the inorder traversal on a binary tree, a node v_i is the immediate inorder predecessor of a node v_j, if v_j is visited immediately after v_i in the inorder traversal. Describe a linear time algorithm for finding the immediate inorder predecessor of a node.

Solution: In the recursive inorder traversal, we recursively traverse the left subtree, visit the root, and then traverse the right subtree. We consider the following cases:

Case 1. v has a left child. If the left child of v does not have a right child, the left child of v is the IIP (Immediate Inorder Predecessor) of v. Otherwise, The rightmost descendant of the left child of v is the IIP of v.

Case 2. v does not have a left child. We further distinguish between the following three cases.

a. v is the right child of its parent. The IIP of v is its parent.
b. No ancestor of v is a right child. v has no IIP.
c. An ancestor of v is a right child. Find the first ancestor that is the right child of its parent, and the parent of this ancestor is the IIP.

The non-recursive algorithm is show in pseudo code as follows:

```
Algorithm inOrderPredecessor(v)
    {
      if v=null
          return null;
      if  left(v) != null   // v has a left child
        {
```

```
    v=left(v);  // the rightmost descendant of left(v) is the IIP
    while right(v)!=null
          v=right(v);
    return v;
   }
else
   {
    if v is root
       return null;
    while parent(v) != null
      {
        if  right(parent(v))=v  // v is the right child of its parent
            return parent (v);
        else
            v = parent(v);
       }
    return null ;  // no IIP
}
```

Time complexity analysis: only the nodes on the path between the immediate inorder predecessor and the node are visited and each visit takes O(1) time. Therefore, the algorithm takes O(h) time, where h is the height of the binary tree.

Problem 7 Let T be a binary tree. Define a **Roman node** to be a node v in T, such that the number of descendants in v's left subtree differs from the number of nodes in v's right subtree by at most 5. Describe a linear-time algorithm for finding each node v of T, such that v is not a Roman node, but all of v's descendants are Roman nodes.

Solution: The following solution is proposed by my student **Brian Price** of the 17s1 class. We use the recursive postorder traversal. For each node v, the algorithm returns a 2-tuple (m, l), where m is the size of the subtree rooted at v, and l is a linked list containing all the nodes in the subtree rooted at v each of which is not a Roman node, but all its descendants are Roman nodes.

```
Algorithm romanNode(v)
    {
      if !hasLeft(v) and !hasRight(v)
       {
         Create an empty list L;
       return (1, L);
      }
     else
      {
        if hasLeft(v)
          (N1, L1) = romanNode(left(v));
        else
          (N1, L1)=(0, null);
        if hasRight(v)
           (N2, L2) = romanNode(right(v));
        else (N2, L2)=(0, null);
        // visit this node
```

```
    if L1=null and L2=null and |N1 - N2| > 5:
      {
        // All the descendants of v are Roman nodes, so v is the first node
        // in the list
        Create empty list L;
        L.append(v);
        return (N1+N2+1, L);
      }
     // v is not a node in the list
    Concatenate L1 and L2 into a single linked list L;
    return (N1+N2+1, L);
  }
```

Time complexity analysis: this algorithm uses the postorder traversal, where each node is visited once and each visit takes constant time. Notice that concatenating two linked lists takes O(1) time by making the last element of one linked list point the first element of the other linked list. Therefore, the time complexity is O(n), where n is the number of nodes.

Problem 8 Let T be a binary tree with n nodes. Define the lowest common ancestor (LCA) of two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself). Given two nodes v and w, describe an efficient algorithm for finding the LCA of v and w. What is the running time of your algorithm?

Solution: Assume that each node has a visit flag that is initially 0.

```
Algorithm LCA(v, w)
  {
    x=v;
    /* Set the visit flag of each node on the path from v to the root */
    while ( x!=null )
      {
        x.visit=1;
        x=x.parent;
      }
    x=w;
    while ( x.visit=0 )  // search for the LCA
      x=x.parent;
    y=v;
    while ( y!=null)  // Reset the visit flag
      {
        y.visit=0;
        y=y.parent;
       }
    return x // x is the LCA of v and w

  }
```

The first while loop takes O(depth(v)) time. The second while loop takes O(depth(w)) time, and the third while loop takes O(depth(v)) time. Therefore, this algorithm takes O(h) time, where h is the height of the tree.

This problem can also be solved in O(n) time by using the postorder traversal, where n is the number of nodes in the tree.

Additional notes: The fastest algorithm for this problem was proposed by Harel and Tarjan [1]. Their algorithm takes constant time after a preprocessing that takes O(n) time, where n is the number of the nodes in the tree.

Refereneces:
[1] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing, 13(2): 338355, 1984.

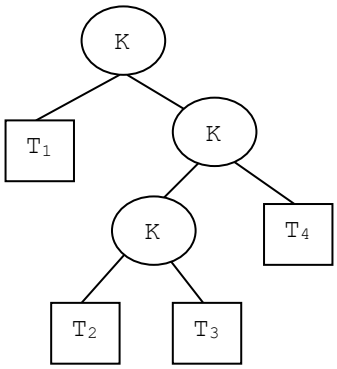
Problem Set 4

Problem 1 Consider a tree storing 100,000 entries. What is the worst-case height of T in the following cases?

a. T is an AVL tree.
b. T is a (2,4) tree.
c. T is a binary search tree
d. T is a splay tree

Problem 2 What does a splay tree look like if its entries are accessed in increasing order of their keys?

Problem 3 Assuming that *T*1, *T*2, *T*3, and *T*4 are AVL trees of height *h*, alter the following binary search tree to yield an AVL tree.



Problem 4 The method findAll(k) in a binary search tree T is to find a set of all the entries in T whose keys are equal to k. Design an algorithm for findAll(k) which runs in time O(h+s), where h is the height of T and s is the size of the set returned.

Problem 5 Show that after the tri-node restructuring operation on the subtree rooted at the node z, the new subtree is an avl tree.

Problem 6 Let T and U be (2,4) trees with n and m entries, respectively, such that all the entries in T have keys less than the keys of all the entries in U. Describe an O(log n + log m)-time algorithm for merging T and U into a single (2,4) tree.

Problem 7 Consider a sequence of keys (5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1). Draw the final result of inserting the entries with these keys in the given order into

a. An initially empty (2, 4) tree.
b. An initially empty red-black tree.

Problem 8 Consider a binary search tree where all the keys are distinct. Describe a modification to the binary search tree that supports the following two index-based operations in O(h) time, where h is the height of the tree:

- atIndex(i): return the entry in the binary search tree with index i.
- indexOf(e): return the index of entry e.

The index of an entry is the sequence number of the entry in the inorder traversal on the tree. For example, the indices of the first two entries with the smallest key and the second smallest key are 0 and 1, respectively.

Problem Set 4

Problem 1 Consider a tree storing 100,000 entries. What is the worst-case height of T in the following cases?

a. T is an AVL tree.
b. T is a (2,4) tree.
c. T is a binary search tree.
d. T is a splay tree.

Solution:

a. Let n(h) be the minimum number of nodes of an AVL tree with height h. We have:
$n(h)=n(h-1)+n(h-2)+1$ ($h\geq 2$), where $n(0)=1$ and $n(1)=2$.

The first 6 n(h)'s (h=0, 1, …, 5) are: 1, 2, 4, 7, 12, 20.
Recall that the first 9 Fibonacci numbers F_i (i=0, 1, … 8) are: 0, 1, 1, 2, 3, 5, 8, 13, 21.

Notice that $n(h) = F_{h+3} - 1$, where F_{h+3} is a Fibonacci number with $F_{h+3} = F_{h+2} + F_{h+1}$.
It is known that $F_h = [\varphi^h/5^{1/2}]$, where $\varphi=(1+5^{1/2})/2$ is the golden ratio.
Therefore, we have $n(h) = [\varphi^{h+3}/5^{1/2}] - 1$, and $h = \log(5^{1/2}(n(h)+1))/\log \varphi - 3$.
By applying the above formula, we get h = 22 for an AVL tree storing 100,000 entries.

b. In the worst-case height, a (2, 3) tree is a complete binary search tree, where there are 2^i nodes at depth i. Let n be the number of nodes, the height h satisfies the following constraint:
$n=2^0 + 2^1 +2^2+ \ldots + 2^h$ (1)
Multiplying both sides by 2, we have
$2n=2^1 + 2^2 +2^3+ \ldots + 2^h + 2^{h+1}$ (2)
So, we have $n=2^{h+1} -1$. Therefore, $h=\log (n+1)-1$.
Hence, the height of a (2,4) tree storing n entries is at most floor(log(n+1)-1), where floor(x) is the largest integer that is no larger than x. Therefore, the worst-case height of T is floor(log(100,001)-1) =15.

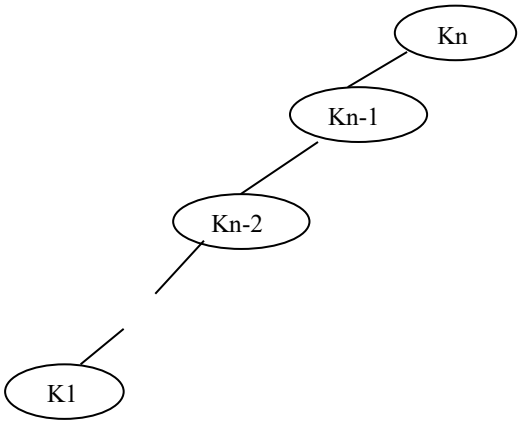
c. The worst-case height of T is 99,999.

d. The worst-case height of T is 99,999.

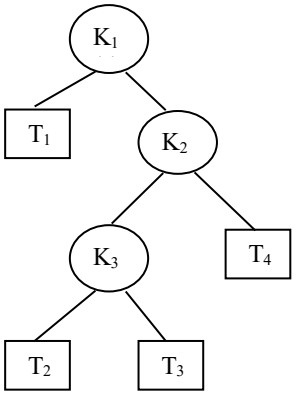
Problem 2 What does a splay tree look like if its entries are accessed in increasing order of their keys?

Solution: Assume that there are n entries in the splay tree and $k_i \leq k_{i+1}$ (i=1, 2, n-1), where k_i is the key of entry i. After entry 1 is assessed, it is moved to the root. Since k_1 is the smallest key, all other entries are in the right subtree of entry 1. Similarly, after entry 2 is accessed, entry 2 is moved to the root and entry 1 becomes the left

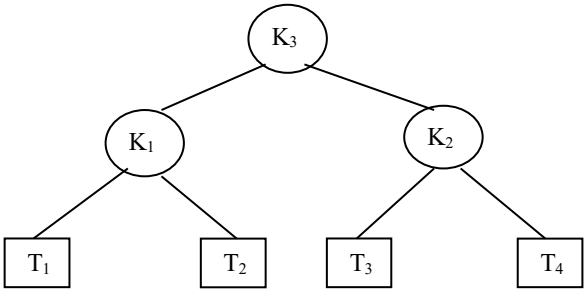
child of entry 2. After entry n is accessed, entry n is moved to the root and entry i-1 is the left child of entry i (i=2, 3, …, n). The resulting splay tree is shown as follows.



Problem 3 Assuming that *T*1, *T*2, *T*3, and *T*4 are AVL trees of height *h*, alter the following binary search tree to yield an AVL tree.



Solution:



Problem 4 Design an algorithm findAll(k) for finding all the entries with the key k in a binary search tree T, and show that it runs in time O(h+s), where h is the height of T and s is the size of the collection returned.

Solution: In a binary search tree, we need to make a rule for placing entries with duplicate keys. Entries with duplicate keys can be inserted in the left subtree or the right subtree of a node with the same key. Therefore, when we find the first node with the same key, we can either search left subtree or the right subtree for all the entries with the same key. However, in case of AVL trees, tri-node restructuring may spoil this order, that is, entries with duplicate keys may be stored in both the left subtree and the right subtree of the node with the same key.

The key idea of the algorithm for finding all the entries with the key k is that after we find the first node, we search the subtree rooted at the first node that contains all the entries with the key k. The algorithm is shown as follows:

```
Algorithm findAll(k)
        Input: The search key k, the root v of the binary search tree T
        Output: A list L containing all the entries with the key k
       {
         Create an empty list L;
         while ( v != NULL )
            {
              if (v.key=k)
                 {
                   findAllEntries(v, L);
                   return L;
                 }
              else
                 if ( v.key > k)
                    v=v.left;
                 else
                    v=v.right;
            }
          return L;
       }

Algorithm findAllEntries(v, L)
        Input: A node v with the key k and a List L
        Output: All the entries with the key k in the subtree rooted at v
       {
          // We use recursive pre-order traversal to find all the entries with the key k
in the subtree rooted at v
          if (v.key=k)
            {
              Add v to L;
              if ( v.left != NULL)
                 FindAllEntries(v.left, L)
              if (v.right !=NULL)
                 FindAllEntries(v.right, L);
            }
          return; // Traversal returns when v does not contain the key k
       }
```

Time complexity analysis:

1. Finding the first node with the key k takes O(h) time.
2. Since at 2s nodes are visited by findAllEntries(), traversing the subtree rooted at the first node takes O(s) time.

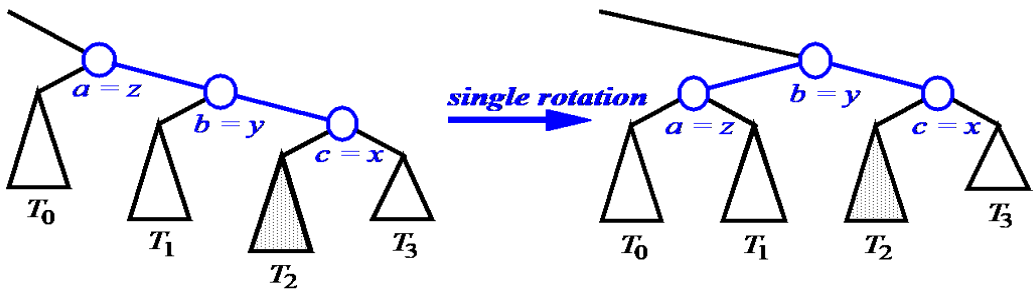
Therefore, this algorithm takes O(h+s) time.

Problem 5 Show that after the tri-node restructuring operation on the subtree rooted at the node z, the new subtree is an avl tree.

Proof: Tri-node restructuring occurs only when a new node is inserted into an avl tree or when a done is removed from an avl tree. Next, we consider the cases for inserting a node. The proof for deleting a node is similar.

Let height (v) be the height of a subtree v or the height of the subtree rooted at v. There are four cases:

Case 1:



By the definition of x, y, z, before tri-node restructuring we have the following equations:

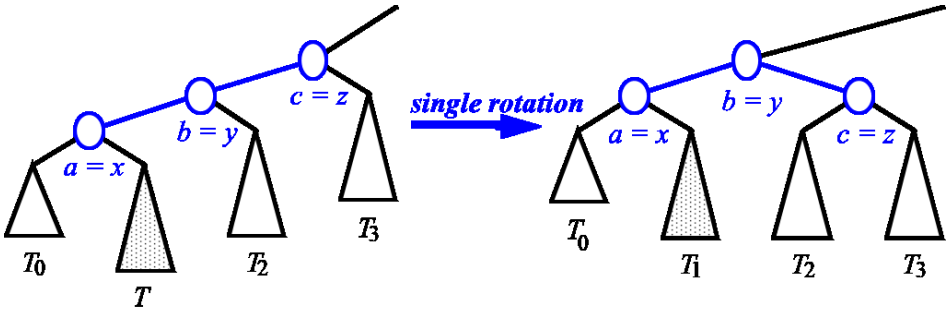
height(y)=height(T_0)+2 (1)
height(x)≤height(T1)+1 (2)
|height(T3)-height(T2)|≤1. (3)

From the above equations, we have:

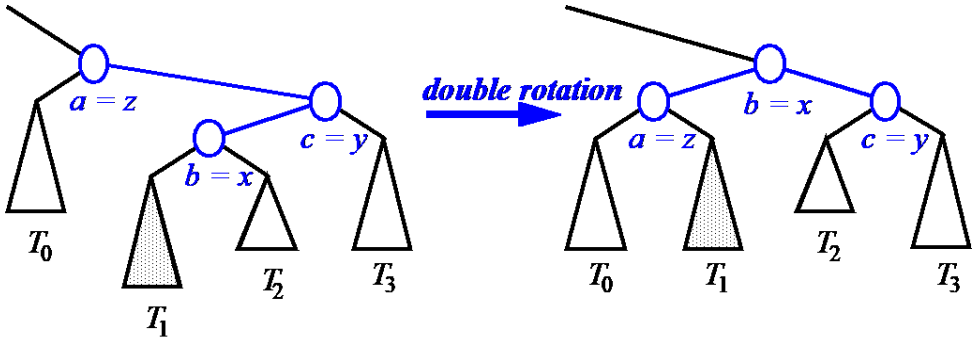
|height(T0)-height(T1)|≤1 (4)
|max{height(T0), height(T1)}-max{height(T2), height(T3)}|≤1 (5)

Equations (3) and (4) imply that in the new tree both z and x satisfy the height difference constraint. Equation (5) implies that in the new subtree, the root y satisfies the height difference constraint. Therefore, the new subtree rooted at y is an avl tree.

Case 2: This case is symmetrical to Case 1.



Case 3:



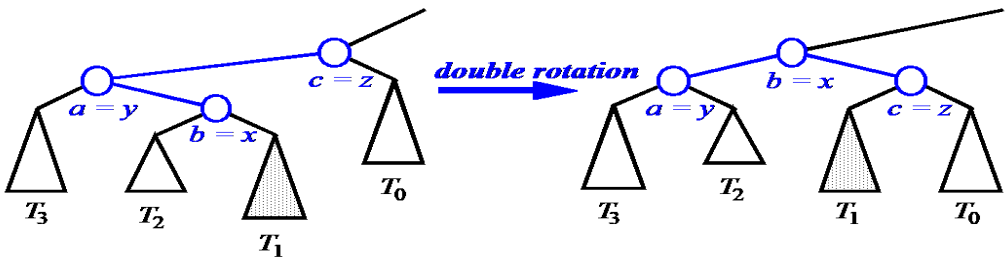
By the definition of x, y, z, before tri-node restructuring we have the following equations:

height(y)=height(T0)+2 (1)
height(x)≤height(T3)+1 (2)
|height(T1)-height(T2)|≤1. (3)

From the above equations, we have:

|height(T0)-height(T1)|≤1 (4)
|max{height(T0), height(T1)}-max{height(T2), height(T3)}|≤1 (5)

Case 4: This case is symmetrical to Case 3.



Problem 6 Let T and U be (2,4) trees with n and m entries, respectively, such that all the entries in T have keys less than the keys of all the entries in U. Describe an O(log n + log m)-time algorithm for merging T and U into a single (2,4) tree.

Solution: Let h_t and h_u be the heights of T and U, respectively. Consider two possible cases:

Case 1. $h_t \geq h_u$. Do the following:

a. Find the entry e with the smallest key in U.
b. Remove the entry e from U.
c. Let h'_u be the new height of U.
d. If $h_t > h'_u$, do the following:
    i. Insert the entry e into the rightmost node v of T at height $h'_u + 1$, and make the root of U the rightmost child of v.
    ii. If there is an overflow, handle the overflow.
e. If $h_t = h'_u$, do the following:
    i. Create a new node v, and insert the entry e into v.
    ii. Make T the left child of v and U the right child of v.

Case 2. $h_t < h_u$. This case is symmetrical to Case 1. Do the following:

f. Find the entry e with the largest key in T.
g. Remove the entry e from T.
h. Let h'_t be the new height of T
i. Insert the entry e into the leftmost node u of U at height $h'_t + 1$, and make the root of T the leftmost child of u.
j. If there is an overflow, handle the overflow.

Time complexity analysis:
Case 1:

a. Finding the entry e with the smallest key in U takes O(log m) time.
b. The remove operation on U takes O(log m) time.
c. The insert operation on T, including handling overflows, takes O(log n) time.

Therefore, the time complexity for this case is O(log n + log m).

Case 1:

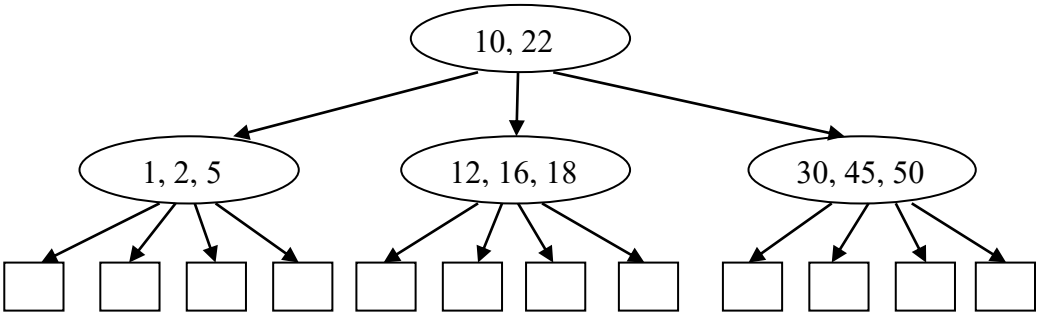
a. Finding the entry e with the largest key in T takes O(log n) time.
d. The remove operation on T takes O(log n) time.
e. The insert operation on U, including handling overflows, takes O(log m) time.

Therefore, the time complexity for this case is O(log n + log m).
Based on the above time complexity analysis, we can conclude that this algorithm takes O(log n + log m) time.

Problem 7 Consider a sequence of keys (5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1). Draw the final result of inserting the entries with these keys in the given order into

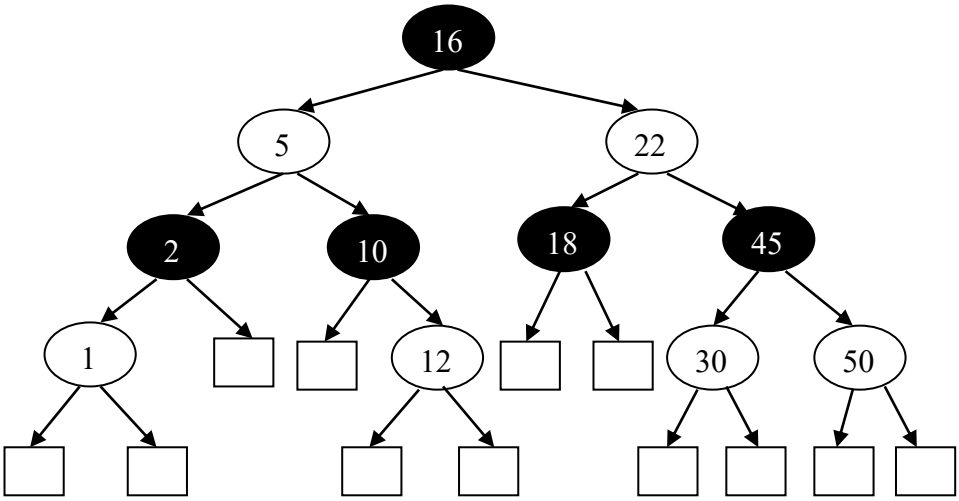
a. An initially empty (2, 4) tree.
b. An initially empty red-black tree.

Solution: a. The final (2, 4) tree:



Notice that the answer is not unique. When an overflow occurs, we can push either the second entry or the third entry to the parent, resulting in different trees.

b. The final red-black tree:



Problem 8 Consider a binary search tree where all the keys are distinct. Describe a modification to the binary search tree that supports the following two index-based operations in O(h) time, where h is the height of the tree:

- atIndex(i): return the entry in the binary search tree with index i.
- indexOf(e): return the index of entry e.

The index of an entry is the sequence number of the entry in the inorder traversal on the tree. For example, the indices of the first two entries with the smallest key and the second smallest key are 0 and 1, respectively.

Solution: For each node, we introduce a field named *size*, denoting the number of entries stored in the subtree rooted at this node. The field *size* of each relevant internal node needs to be updated when a new entry is inserted or an entry is removed.

- Key idea with atIndex(i): We use a recursive algorithm findIndex(i,v), where i is the relative index in the subtree rooted at v, and v is the current node. Given the current node v, if the target entry is in the left subtree, the relative index is unchanged. If the target entry is in the right subtree, the relative index in the right subtree is i-left(v).size-1.

- Key idea with indexOf(e): Since the index of an entry is its sequence number in the inorder traversal on the tree, the algorithms for indexOf(e) dynamically keeps track of the running number of inorder predecessors of the target entry.

Algorithm atIndex(i)
Input: The index i
Output: The position in the binary search tree of the entry at index i.

```
{
  if ( i>= the total number of entries in the tree )
     return IndexOutOfBound(); // error
  return findIndex(i, root);
}
```

Algorithm findIndex(i, v)
Input: The index i and the current node v
Output: The position in the binary search tree of the entry at index i.

```
{
  if ( (i=0 and left(v)=null) or left(v).size=i)
     return v;   // found
  if (i<left(v).size)   // in the left subtree and the relative index in the left
                        // subtree remains the same
     return findIndex(i, left(v));
  else   // in the right subtree and its relative index in the right subtree is
         //  i-left(v).size-1
     return findIndex(i-left(v).size-1, right(v));
}
```

Algorithm indexOf(e)
Input: An entry e
Output: The index of e.

```
{
  if ( root=null )
     return EntryDoesNotExist();  //  e is not found
  return indexOf(e, root, 0);
}
```

Algorithm indexOf(e, v, i)
Input: An entry e, the current node v, and the running number of inorder predecessors of v
Output: The index of e.

```
{
 if (v = null)
     return EntryDoesNotExist();  // e is not found
  if (v.key = the key of e)
     return i+left(v).size;   //  e is found
  if (v.key>the key of e)
    // e is in the left subtree, so the running number of inorder predecessors of
    // e is unchanged
      return indexOf(e, left(v), i);
  else   // e is in the right subtree and the running number of inorder
```

```
        // predecessors of e increases by left(v).size+1
    if (left(v)≠null)
        return indexOf(e, right(v), i+left(v).size+1);
    else
        return indexOf(e, right(v), i+1);
}
```

Time complexity analysis: Both algorithms traverse the tree along one path from the root to a leaf node in the worst case, and it takes O(1) time to visit each node. Therefore, both algorithms take O(h) time, where h is the height of the tree.

Problem Set 5

Problem 1 Show how to implement a stack ADT using only a priority queue and one additional integer variable.

Problem 2 Write an algorithm for updating the key of an item in a priority queue, and analyse its time complexity.

Problem 3 Given a heap T and a key k, give an algorithm to compute all the items in T with keys less than or equal to k. Your algorithm should run in time proportional to the number of items returned.

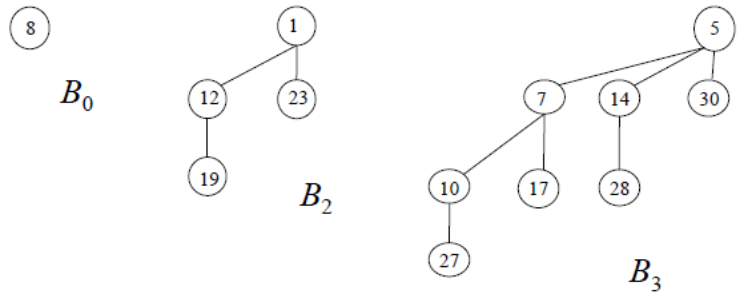
Problem 4 Qantas Airlines wants to give a first-class upgrade coupon to their top log n frequent flyers, based on the number of miles accumulated, where n is the total number of the airlines' frequent flyers. The algorithm they currently use, which runs in O(n log n) time, sorts the flyers by the number of miles flown and then scans the sorted list to pick the top log n flyers. Describe an algorithm that identifies the top log n flyers in O(n) time.

Problem 5 Suppose two binary trees, T1 and T2, hold entries satisfying the heap-order property, where no entry in each tree exists in the other tree. Describe a method for combining T1 and T2 into a tree T such that T's internal nodes hold the union of the entries in T1 and T2 and T also satisfies the heap-order property. Your algorithm should run in time O(h1+h2) where h1 and h2 are the respective heights of T1 and T2.

Problem 6 Give an alternative analysis of the bottom-up heap construction algorithm.

Problem 7 Prove that a binomial tree with 2^n nodes has $\binom{n}{i}$ nodes at depth i (0 ≤ i ≤n).

Problem 8 Consider the following binomial heap. Draw the resulting binomial heaps after inserting the keys 7, 12, 20, 24, 25 and 25, respectively.



Problem 9 In a computer game, all the players are divided into a number of groups. Each player can join one group only and is not allowed to join a different group later. Describe an algorithm for checking if two players are in the same group. What is the running time of your algorithm?

Problem Set 5

Problem 1 Show how to implement a stack ADT using only a priority queue and one additional integer variable.

Solution: Maintain a maxKey variable initialized to 0. On a push operation for element e, call insertItem(maxKey, e) and decrement maxKey. On a pop operation, call removeMinElement and increment maxKey.

Problem 2 Write an algorithm for updating the key of an item in a priority queue, and analyse its time complexity.

Solution: Assume the priority queue is based on a min-heap.

```
Algorithm updateKey(v)
 Input: a node v containing the item
 Output: the heap with the key updated
  {
    // upheap bubbling
    while (v!=NULL && v.key < v.parent.key)
        {
          swap the items of v and its parent;
          v=v.parent;
        }

   // downheap bubbling
   while (v!=NULL && v.key > min{v.left.key, v.right.key})
       {
         let u be the child of v with the smaller key;
        swap the items of v and u;
        v=u;
       }
}
```

Time complexity analysis: This algorithm performs either upheap bubbling or downheap bubbling. The loop body of each while loop takes O(1) time, and the number of iterations of each while loop is no more than h, where h is the height of the heap. Since h is O(log n), the time complexity of this algorithm is O(log n), where n is the number of items in the heap.

Problem 3 Given a heap T and a key k, give an algorithm to compute all the items in T with keys less than or equal to k. Your algorithm should run in time proportional to the number of items returned.

Solution:

```
Algorithm lessThanOrEqualToKEntries(H, v)
 Input: A heap H and a node v
 Output: A node list L that contains all the entries with keys less than k
  {
   if ( v.key≤k )
      {
        L.add((v.key, v.value));  // add the entry v to the list L
        if (v.leftchild !=null) LessThanOrEqualToKEntries(H, v.leftchild);
        if (v.rightchild !=null) LessThanOrEqualToKEntries(H, v.rightchild);
      }
 }
```

According to the heap order property, there is no node in T storing a key larger than k that has a descendent storing a key less than or equal to k. As a result, this algorithm takes O(n) time, where n is the number of entries returned.

Problem 4 Qantas Airlines wants to give a first-class upgrade coupon to their top log n frequent flyers, based on the number of miles accumulated, where n is the total number of the airlines' frequent flyers. The algorithm they currently use, which runs in O(n log n) time, sorts the flyers by the number of miles flown and then scans the sorted list to pick the top log n flyers. Describe an algorithm that identifies the top log n flyers in O(n) time.

Solution:

```
Algorithm TopKFlyers(A)
 Input: A list A of n flyers
 Output: An array B of the top log n flyers
  {
   Construct a heap H storing all the n flyers, where the key of each flyer P_i is 1/m_i
   (m_i is the number of miles P_i has flown);
   for (i=0; i< log n; i++)
      B[i]= H.removeMin();
   return B;
  }
```

Running time analysis: It takes O(n) time to construct a heap with n integers as keys by using bottom-up heap construction algorithm. removeMin() takes O(log n) time. Therefore, this algorithm takes $O(n + (\log n)^2)=O(n)$ time.

Problem 5 Suppose two binary trees, T1 and T2, hold entries satisfying the heap-order property, where no entry in each tree exists in the other tree. Describe a method for combining T1 and T2 into a tree T such that T's internal nodes hold the union of the entries in T1 and T2 and T also satisfies the heap-order property. Your algorithm should run in time O(h1+h2) where h1 and h2 are the respective heights of T1 and T2.

Solution:

Algorithm treeUnion(T1, T2)
Input: Two trees T1 and T2 that satisfy the heap-order property.
Output: A tree T that is the union of T1 and T2 and also satisfies the heap-order property.

```
{
  v=T1.removeMin();
  let v be the root of T;
  leftchild(v) = the root of T1;
  rightchild(v) = the root of T2;
  apply the down-heap bubbling to the tree T;
}
```

Running time analysis: T1.removeMin() takes O(h1) time. The down-heap bubbling to the tree T takes O(h2) time as only the down-heap bubbling to the subtree T2 is performed. All other operations take O(1) time. Therefore, this algorithm takes O(h1)+O(h2)+O(1)=O(h1+h2) time.

Problem 6 Give an alternative analysis of the bottom-up heap construction algorithm.

Solution: In the bottom–up heap construction, the number of merge operations is equal to the number of non–leaf nodes. The height of the heap is log n, where n is the total number of nodes. At each level i (i=0, 1, …, log n), the total number of nodes is 2^i. Each node v_k corresponds to one merge operation which takes O(log n – i) time, where log n –i is the height of the subtree rooted at v_k. Therefore, the total time of the heap construction is

$\Sigma_{i=0..\log n}\, 2^i(\log n - i) = \Sigma_{i=0..\log n}\, 2^{(\log n - i)}\, i = 2^{\log n}\Sigma_{i=0..\log n}\, 2^{-i}\, i = 2^{\log n}\Sigma_{i=0..\log n}\, i/2^i$.

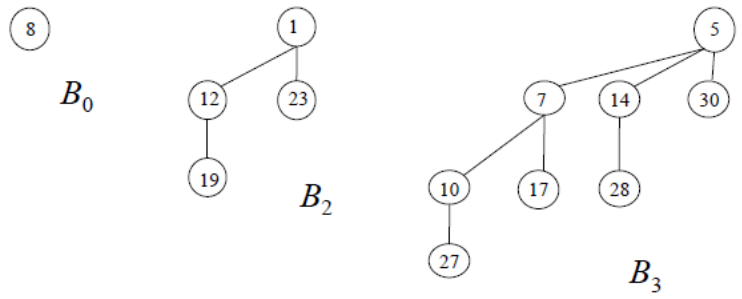
By using induction we can prove that $i \le 2^{i/2}$ holds for i>3. Therefore, we have $\Sigma_{i=0..\log n}\, i/2^i \le 1+3/8+\Sigma_{i=4..\log n}\, 1/2^{i/2} = 1+3/8+\Sigma_{i=4..\log n}\, (1/2^{1/2})^i < 1+3/8+1/(4-2\text{sqrt}(2)) < 2.5$. Hence, $2^{\log n}\Sigma_{i=0..\log n}\, i/2^i < 2.5*2^{\log n} = 2.5n = O(n)$.

Problem 7 Prove that a binomial tree with 2^n nodes has $\binom{n}{i}$ nodes at depth i ($0 \le i \le n$).

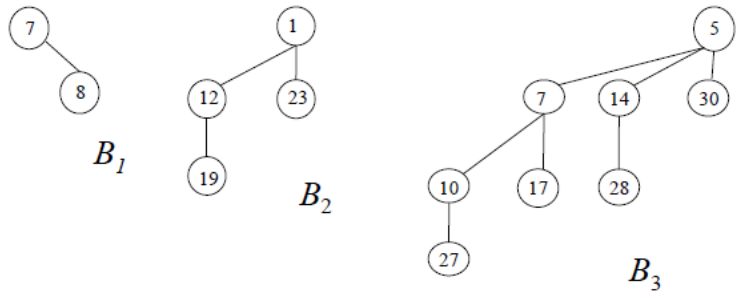
Proof: By the definition B_n, B_n consists of two B_{n-1} with the root with a larger key of one B_{n-1} being the child of the root of the other B_{n-1}. We prove it by induction. For n = 0, only the root of B_n is at depth 0. Therefore, the statement is true. Suppose in B_{n-1}, the number of nodes at depth i is $\binom{n-1}{i}$. Notice that the nodes at depth i − 1 of one B_{n-1} becomes the nodes at depth i for B_n. and the nodes at depth i-1 of the other B_{n-1} remain at the same level. Therefore, in B_n, the number of nodes at depth i is

$$\binom{n-1}{i} + \binom{n-1}{i-1} = \binom{n}{i}.$$

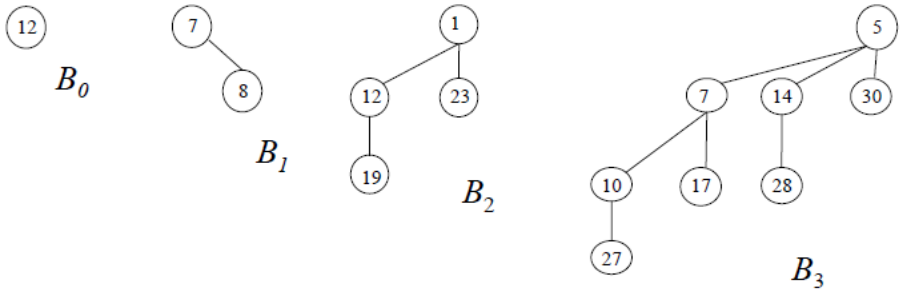
Problem 8 Consider the following binomial heap. Draw the resulting binomial heaps after inserting the keys 7, 12, 20, 24, 25 and 25, respectively.



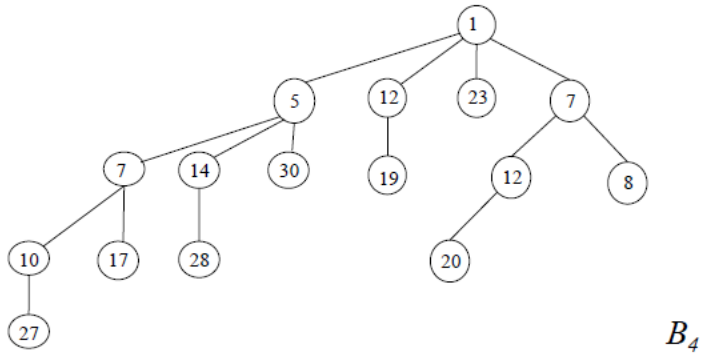
Solution: After insert 7:



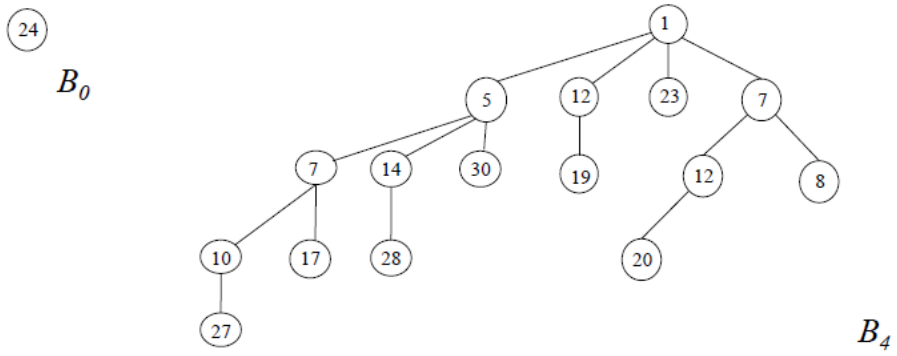
After Insert 12:



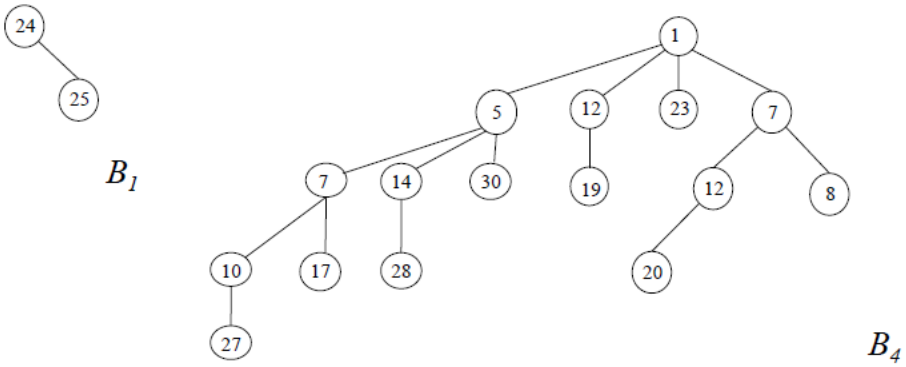
After insert 20:



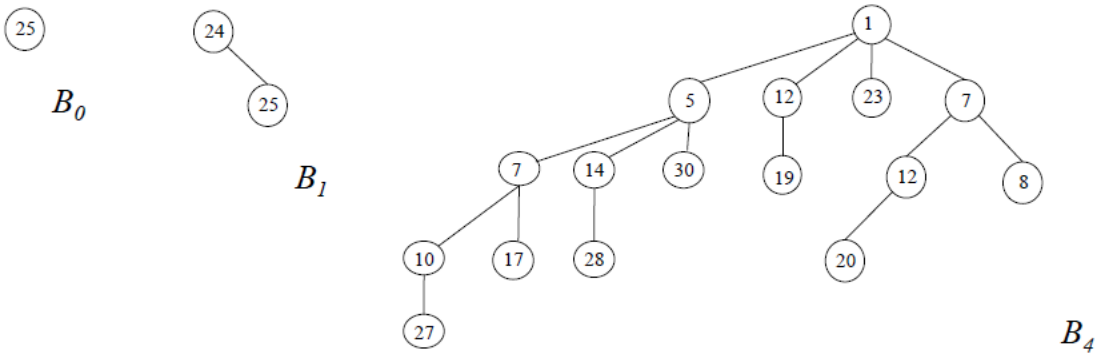
After insert 24:



After insert the first 25:



After insert the second 25:



Problem 9 In a computer game, all the players are divided into a number of groups. Each player can join one group only and is not allowed to join a different group later. Describe an algorithm for checking if two players are in the same group. What is the running time of your algorithm?

Solution: Use the disjoint set union-find data structure with union-by-size and path compression heuristics. The amortized complexity for checking if two players are in the same group is O(log* n).

Problem Set 6

Problem 1 Show how to modify the KPM pattern matching algorithm so as to find every occurrence of a pattern string P that appears as a substring in T, while still running in O(n+m) time. (Be sure to catch even those matches that overlap.)

Problem 2 A pattern P of length m is said to be a circular substring of a text T of length n if P is equal to the concatenation of a suffix of T and a prefix of T, where neither the suffix and nor the prefix is an empty string. For example, if T=aacabca, all the circular substrings of length 3 of T are caa and aaa. Give an O(n+m)-time algorithm for determining whether P is a circular substring of T.

Problem 3 Draw a standard trie and a compressed trie for the following set of strings:

{abab, baba,ccccc, bbaaaa, caa, bbaacc, cbcc, cbca}.

Problem 4 Draw the frequency array and Huffman tree for the following string:

"dogs do not spot hot pots or cats".

Problem 5 Give an efficient algorithm for deleting a string from a compressed trie and analyse its running time.

Problem 6 Give a sequence $S=(x_0, x_1, x_2, \ldots, x_{n-1})$ of numbers, describe an $O(n^2)$-time algorithm for finding a longest subsequence $T=(x_{i0}, x_{i1}, x_{i2}, \ldots, x_{ik-1})$ of the numbers, such that $i_j<i_{j+1}$ and $x_{ij}>x_{ij+1}$. That is, T is a longest decreasing subsequence of S.

Problem 7 Given a string s with repeated characters, design an efficient algorithm for rearranging the characters in s so that no two adjacent characters are identical, or determine that no such permutation exists. Analyse the time complexity of your algorithm.

Problem Set 6 Solutions

Problem 1 Show how to modify the KPM pattern matching algorithm so as to find every occurrence of a pattern string P that appears as a substring in T, while still running in O(n+m) time. (Be sure to catch even those matches that overlap.)

Solution: Instead of returning when a match is found, store the index and set i = i+1 and j = F(m-1).
The modified KMP algorithm is shown as follows.

Algorithm ***KMPMatch(T, P)***

```
{   F = failureFunction(P);
    i = 0;
    j = 0;
    Construct an empty stack S; // S stores the locations of all the occurrences of P in T
    while ( i < n )
      if ( T[i] = P[j] )
        { if ( j = m–1 )
            {
              S.push(i – j) ; //An occurrence and i-j is the location of this occurrence
              i=i+1;
              j=F(m-1);
            }
          else
            { i = i + 1;
              j = j + 1;}
            }
       else
         if ( j > 0 )
           j = F[j – 1];
         else
           i = i + 1;
   return S;  //  if S is empty, no match is found
}
```

Since constructing an empty stack and the push operation take O(1) time, the time complexity of this algorithm is still O(n+m).

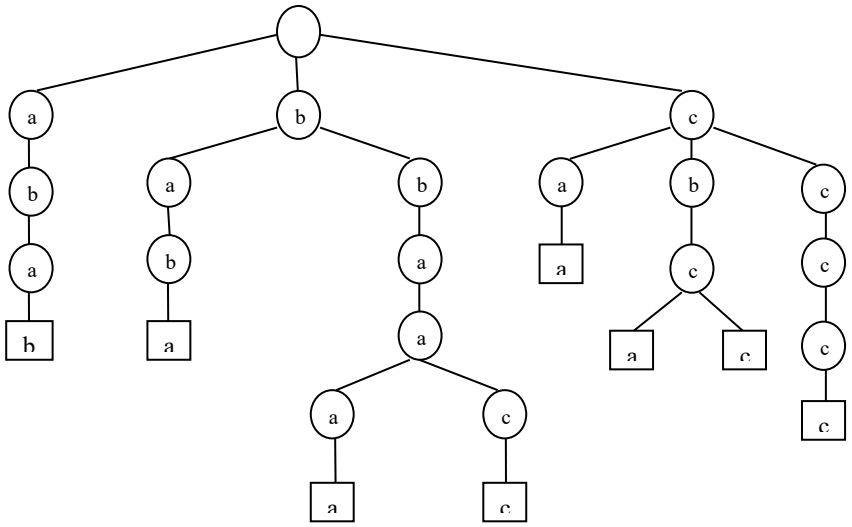
Problem 2 A pattern P of length m is said to be a circular substring of a text T of length n if P is equal to the concatenation of a suffix of T and a prefix of T, where neither the suffix and nor the prefix is an empty string. For example, if T=aacabca, all the circular substrings of length 3 of T are caa and aaa. Give an O(n+m)-time algorithm for determining whether P is a circular substring of T.

Solution: Generate a new text T′ = T[n−m+1. . . n-1]+T[0 . . .m-2]. Run KMP algorithm on T′ and P.

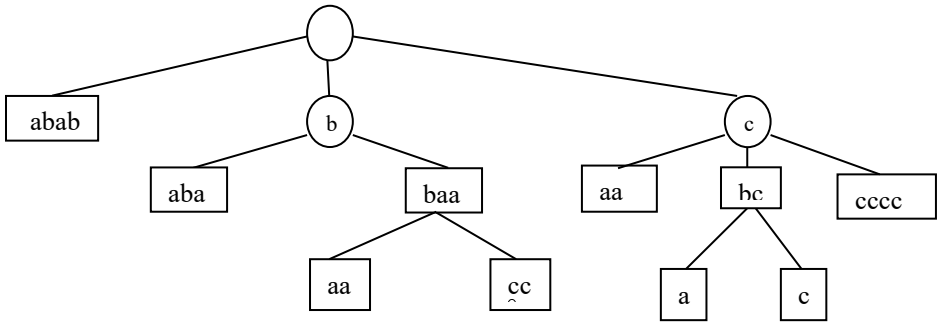
Problem 3 Draw a standard trie and a compressed trie for the following set of strings:

{abab, baba,ccccc, bbaaaa, caa, bbaacc, cbcc, cbca}.

Solution: The standard trie:



The compressed trie:



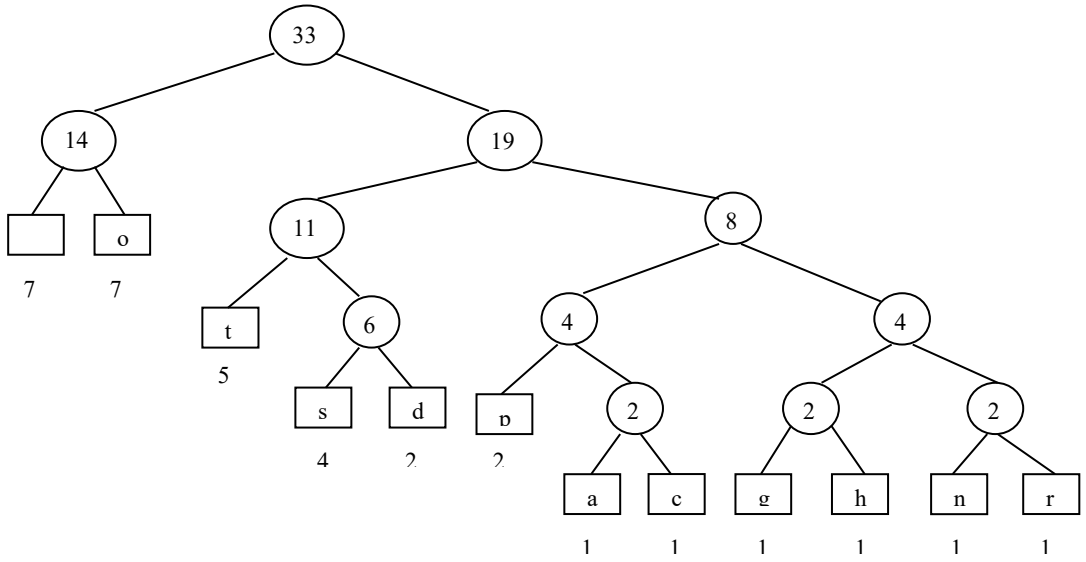
Problem 4 Draw the frequency array and Huffman tree for the following string:

"dogs do not spot hot pots or cats".

Solution: The frequency array:

Character		a	c	d	g	h	n	o	p	r	s	t
Frequency	7	1	1	2	1	1	1	7	2	1	4	5

The Huffman tree:



Problem 5 Give an efficient algorithm for deleting a string from a compressed trie and analyse its running time.

Solution:

```
Algorithm stringDeletion(T, s)
  Input: A compressed trie T and a string s
  Output: T without s
  {
   search the compressed trie for s; // work out the search procedure yourself
    if ( s was not found )
       {  print "s is not found";
          return 0;  // unsuccessful deletion
        }
    else
       {  let u be the node where s was just found;
          if ( s is not equal to the whole string ended at u or u has a child )
             {  print " s is a partial string";
                return 0;   // unsuccessful deletion
             }
          v=the parent of u;
          delete u;
          if ( v has a child c )
             { // p.string denotes the string stored at a node p
               v.string=v.string+c.string;
               delete node c;  // merge v and c into a single node
             }
          return 1;   // successful deletion
       }
```

```
}
```

Running time analysis: O(dm), where d is the size of the alphabet and m the depth of the external node that represents the string.

Problem 6 Give a sequence $S=(x_0, x_1, x_2, \ldots, x_{n-1})$ of numbers, describe an $O(n^2)$-time algorithm for finding a longest subsequence $T=(x_{i0}, x_{i1}, x_{i2}, \ldots, x_{ik-1})$ of the numbers, such that $i_j<i_{j+1}$ and $x_{ij}>x_{ij+1}$. That is, T is a longest decreasing subsequence of S.

Solution:

```
Algorithm subsequence(S)
  Input: A sequence S
  Output: The longest decreasing subsequence L of S
  {
   create an identical copy S1 of S;
   sort S1 in non-ascending order;
   remove all duplicates in S1 so that all numbers are distinct;
   find the longest common subsequence L of S and S1;
   return L;
  }
```

Time complexity analysis:

1. It takes O(n) time to create S1.
2. Sorting S1 using a heap-based priority queue takes O(n log n) time.
3. Removing all duplicates in S1 takes O(n) time.
4. Finding the longest common subsequence of S and S1 takes $O(n^2)$ time.

Therefore, this algorithm takes $O(n)+O(n \log n)+O(n^2)+O(n^2)=O(n^2)$ time.

Problem 7 Given a string s with repeated characters, design an efficient algorithm for rearranging the characters in s so that no two adjacent characters are identical, or determine that no such permutation exists. Analyse the time complexity of your algorithm.

Solution: We can use a greedy approach to solve this problem. The idea is to put the character with the highest frequency first. We use a priority queue based a max heap to store all characters with frequencies as keys where the highest frequency character is stored at the root. We repeatedly do the following until the priority queue becomes empty:

1. Remove the highest frequency character from the heap and append it to the result.
2. Decrease frequency of the character and temporarily move this character out of priority queue so that it is not picked next time.

```
Algorithm rearrangeString(s):
  Input:  a string s
  Output: a permutation of s such that no two adjacent chars are the same or false if no
such permutation exists
```

```
compute frequency of each char in s;
P=priority queue of distinct chars in S with frequency as key;
s_new=empty string;
c=removeMax(P); // remove the char with the max frequency
append c to s_new;
c.key=c.key-1;
while P is not empty
    {  d=removeMax(P);
       append d to s_new;
       d.key=d.key-1
       if  ( c.key>0 )
           insert(P, c);        // insert c back into the priority queue
       c=d;
    }
if ( c.key>0 )
  return false;
else
  return s_new;
```

Time complexity analysis:

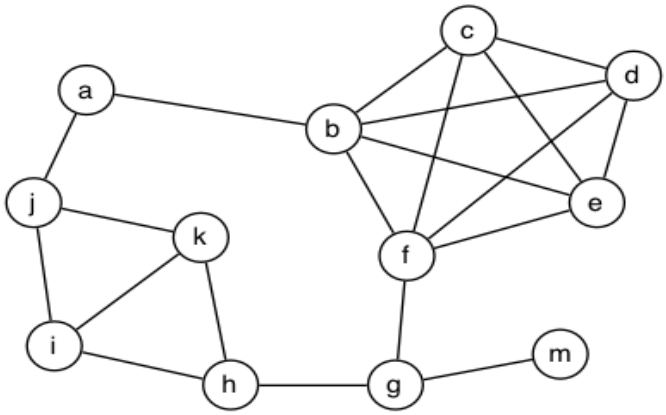
Let n be the size of the input string s.

1. Computing the frequencies of all characters in s takes O(n) time.
2. Creating a priority queue for all distinct characters using a heap-based priority queue takes O(n) time.
3. Both removeMax() and insert() take O(log n) time. Hence, the while-loop takes O(n·log n) time.

Therefore, the complexity of this algorithm is O(n·log n).

Problem Set 7

Problem 1. For the following graph,

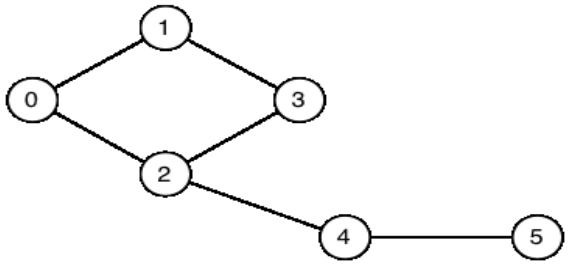


give examples of the smallest (but not of size/length 0) and largest of each of the following:

1. path
2. cycle
3. spanning tree
4. vertex degree
5. clique

Problem 2. Show how the following graph would be represented by

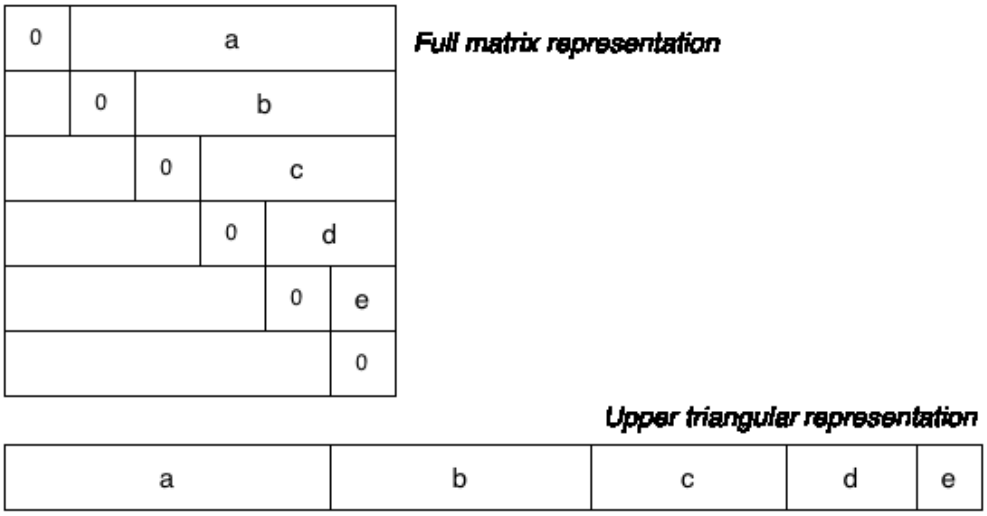
1. an adjacency matrix representation (V×V matrix with each edge represented twice)
2. an adjacency list representation (where each edge appears in two lists, one for v and one for w)



Problem 3. Consider the adjacency matrix and adjacency list representations for graphs. Analyse the storage costs for the two representations in more detail in terms of the number of vertices V and the number of edges E. Determine roughly the V:E ratio at which it is more storage efficient to use an adjacency matrix representation vs the adjacency list representation.

For the purposes of the analysis, ignore the cost of storing the GraphRep structure. Assume that: each pointer is 4 bytes long, a Vertex value is 4 bytes, a linked-list node is 8 bytes long and that the adjacency matrix is a complete V×V matrix. Assume also that each adjacency matrix element is 1 byte long. (Hint: Defining the matrix elements as 1-byte boolean values rather than 4-byte integers is a simple way to improve the space usage for the adjacency matrix representation.)

Problem 4. The standard adjacency matrix representation for a graph uses a full n×n matrix and stores each edge twice (at [v,w] and [w,v]). This consumes a lot of space, and wastes a lot of space when the graph is sparse. One way to use less space is to store just the upper (or lower) triangular part of the matrix, as shown in the diagram below:



The n×n matrix has been replaced by a single 1-dimensional array g.edges[] containing just the "useful" parts of the matrix.

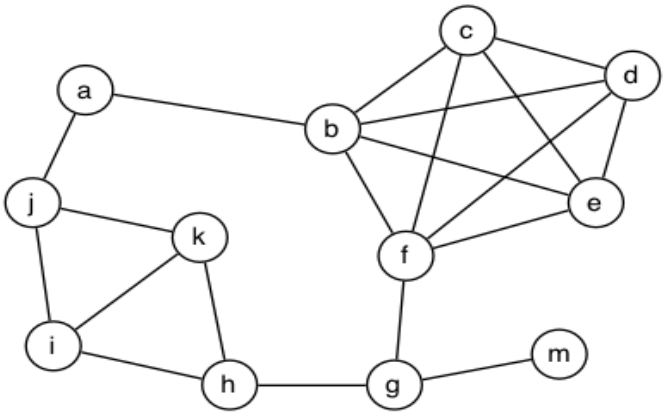
Accessing the elements is no longer as simple as g.edges[v][w]. Write pseudocode for a method to check whether two vertices v and w are adjacent under the upper-triangle matrix representation of a graph g.

Problem 5. Write an algorithm in pseudocode for computing the minimum and maximum vertex degree. Your algorithm should be representation-independent; the only function you should use is to check if two vertices v,w∈{0,⋯n-1} are adjacent in g. Determine the time complexity of your algorithm. Assume that the adjacency check takes constant time, O(1).

Problem 6. Write an algorithm in pseudocode for computing all 3-cliques ("triangles") of a graph g with n vertices. Your algorithm should be representation-independent; the only function you should use is to check if two vertices v,w∈{0,⋯n-1} are adjacent in g. Determine the time complexity of your algorithm. Assume that the adjacency check takes constant time, O(1).

Problem Set 7 Solutions

Problem 1. For the following graph,



give examples of the smallest (but not of size/length 0) and largest of each of the following:

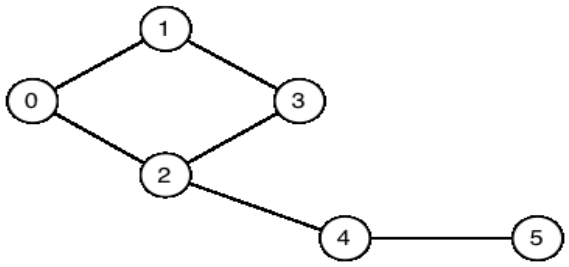
1. path
2. cycle
3. spanning tree
4. vertex degree
5. clique

Solution:

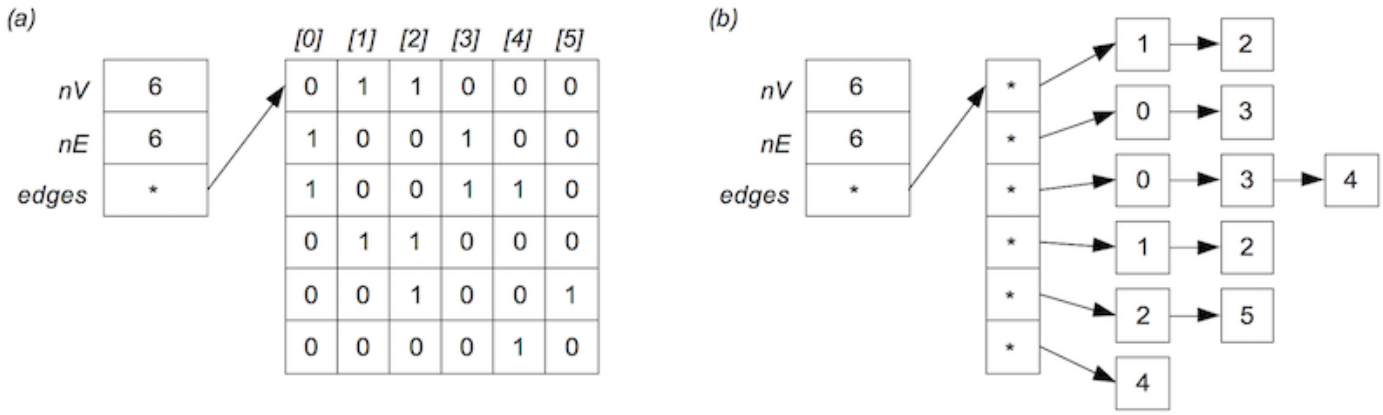
1. path
   smallest: any path with one edge (e.g. a-b or g-m)
   largest: some path including all nodes (e.g. m-g-h-k-i-j-a-b-c-d-e-f)
2. cycle
   smallest: need at least 3 nodes (e.g. i-j-k-i or h-i-k-h)
   largest: path including most nodes (e.g. g-h-k-i-j-a-b-c-d-e-f-g) (can't involve m)
3. spanning tree
   smallest: any spanning tree must include all nodes (the largest path above is an example)
   largest: same
4. vertex degree
   smallest: there is a node that has degree 1 (vertex m)
   largest: in this graph, 5 (b or f)
5. clique
   smallest: any vertex by itself is a clique of size 1
   largest: this graph has a clique of size 5 (nodes b,c,d,e,f)

Problem 2. Show how the following graph would be represented by

1. an adjacency matrix representation (V×V matrix with each edge represented twice)
2. an adjacency list representation (where each edge appears in two lists, one for v and one for w)



Solution:



Problem 3. Consider the adjacency matrix and adjacency list representations for graphs. Analyse the storage costs for the two representations in more detail in terms of the number of vertices V and the number of edges E. Determine roughly the V:E ratio at which it is more storage efficient to use an adjacency matrix representation vs the adjacency list representation.

For the purposes of the analysis, ignore the cost of storing the GraphRep structure. Assume that: each pointer is 4 bytes long, a Vertex value is 4 bytes, a linked-list node is 8 bytes long and that the adjacency matrix is a complete V×V matrix. Assume also that each adjacency matrix element is 1 byte long. (Hint: Defining the matrix elements as 1-byte boolean values rather than 4-byte integers is a simple way to improve the space usage for the adjacency matrix representation.)

Solution: The adjacency matrix representation always requires a V×V matrix, regardless of the number of edges, where each element is 1 byte long. It also requires an array of V pointers. This gives a fixed size of $V \cdot 4 + V^2$ bytes.

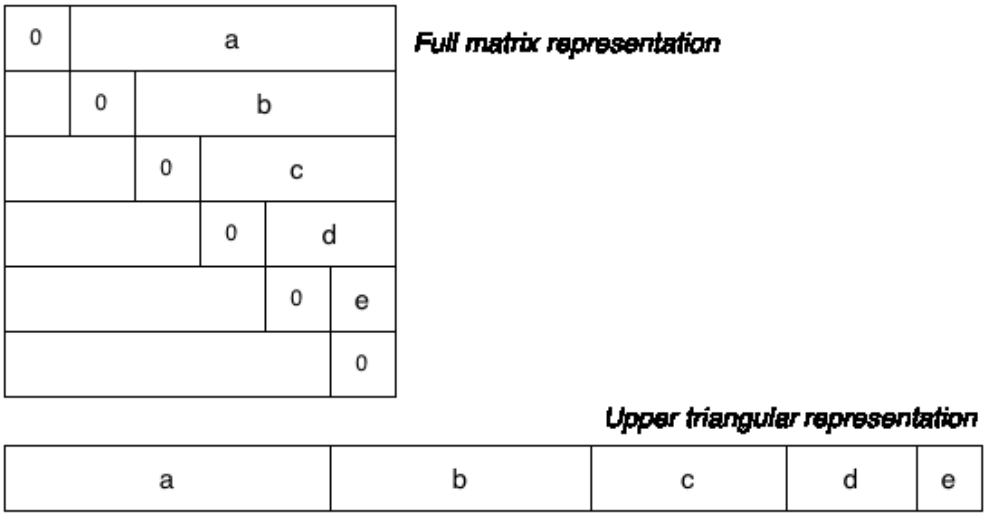
The adjacency list representation requires an array of V pointers (the start of each list), with each being 4 bytes long, and then one list node for each edge in each list. The total number of edge nodes is 2E (each edge (v,w)is stored twice, once in the list for v and once in the list for w).

Since each node requires 8 bytes (vertex+pointer), this gives a size of V·4+8·2·E. The total storage is thus V·4+16·E.

Since both representations involve V pointers, the difference is based on V^2 vs 16E. So, if $16E < V^2$ (or, equivalently, $E < V^2/16$), then the adjacency list representation will be more storage-efficient. Conversely, if $E > V^2/16$, then the adjacency matrix representation will be more storage-efficient.

To pick a concrete example, if V=20 and if we have less than 25 edges (= 20·20/16), then the adjacency list will be more storage-efficient, otherwise the adjacency matrix will be at least as storage-efficient.

Problem 4. The standard adjacency matrix representation for a graph uses a full n×n matrix and stores each edge twice (at [v,w] and [w,v]). This consumes a lot of space, and wastes a lot of space when the graph is sparse. One way to use less space is to store just the upper (or lower) triangular part of the matrix, as shown in the diagram below:



The n×n matrix has been replaced by a single 1-dimensional array g.edges[] containing just the "useful" parts of the matrix.

Accessing the elements is no longer as simple as g.edges[v][w]. Write pseudocode for a method to check whether two vertices v and w are adjacent under the upper-triangle matrix representation of a graph g.

Solution:

The following solution uses a loop to compute the correct index in the 1-dimensional edges[] array:

Algorithm adjacent(g,v,w)
Input: graph g in upper-triangle matrix representation
v, w vertices such that $v \neq w$

```
Output: true if v and w adjacent in g, false otherwise
  {
    if ( v>w )
       swap v and w;       // to ensure v<w

    chunksize=g.nV-1;
    offset=0;
    for  i=0..v-1 do
       {
         offset=offset+chunksize;
         chunksize=chunksize-1;
       }
    offset=offset+w-v-1;
    if ( g.edges[offset]=1)
          return true;
    else   return false;

   }
```

Alternatively, you can compute the overall offset directly via the formula $(nV-1)+(nV-2)+...+(nV-v)+(w-v-1)=nvV-v(v+1)/2+w-v-1$ (assuming that v<w).

Problem 5. Write an algorithm in pseudocode for computing the minimum and maximum vertex degree. Your algorithm should be representation-independent; the only function you should use is to check if two vertices $v,w \in \{0,\cdots n-1\}$ are adjacent in g. Determine the time complexity of your algorithm. Assume that the adjacency check takes constant time, O(1).

Solution:

The following algorithm uses two nested loops to compute the degree of each vertex. Hence its time complexity is $O(n^2)$.

```
Algorithm MinMaxDegree(g,n)
Input:  graph g of n vertices
Output: minimum and maximum vertex degree in g
 {
   min=n-1;
   max=0;
   for all vertices v∈g do
     { deg[v]=0;
       for all vertices w∈g, v≠w do
           if  v and w are adjacent in g
             deg[v]=deg[v]+1;
       if ( deg[v]<min )
          min=deg[v];
       if ( deg[v]>max )
```

```
            max=deg[v];
        }
    return min,max;
}
```

Problem 6. Write an algorithm in pseudocode for computing all 3-cliques ("triangles") of a graph g with n vertices. Your algorithm should be representation-independent; the only function you should use is to check if two vertices $v,w \in \{0,\cdots n-1\}$ are adjacent in g. Determine the time complexity of your algorithm. Assume that the adjacency check takes constant time, O(1).

Solution:

The following algorithm uses three nested loops to print all 3-cliques in order. Hence its asymptotic running time is $O(n^3)$.

```
Algorithm show3Cliques(g,n):
Input: graph g of n vertices numbered 0..n-1
Output: all 3-cliques
{
  for all i=0..n-3 do
      for all j=i+1..n-2 do
          if i and j are adjacent in g
              for all k=j+1..n-1 do
                  if i and k adjacent in g and j and k are adjacent in g
                     print i"-"j"-"k;
}
```

Problem Set 8

Problem 1 The BFS (Breadth-First Search) algorithm given in the lecture notes uses multiple lists. Modify the algorithm so that it uses only one queue to replace multiple lists.

Problem 2 Describe, in pseudo code, an O(n+m)-time algorithm for computing all the connected components of an undirected graph G with n vertices and m edges.

Problem 3 Given an undirected graph G and a vertex v_i, describe an algorithm for finding the shortest paths from v_i to all other vertices. The shortest path from a vertex v_s to a vertex v_t is a path from a vertex v_s to a vertex v_t with the minimum number of edges. What is the running time of your algorithm?

Problem 4 A connected undirected graph is said to be biconnected if it contains no vertex whose removal would divide G into two or more connected components. Give an O(n+m)-time algorithm for adding at most n edges to a connected graph G, with n>3 vertices and m>n-1 edges, to guarantee that G is biconnected.

Problem 5 An n-vertex directed acyclic graph G is **compact** if there is some way of numbering the vertices of G with the integers from 0 to n-1 such that G contains the edge (i, j) if and only if i<j, for all i, j in [0, n-1]. Give an $O(n^2)$-time algorithm for detecting if G is compact.

Problem 6 An Euler tour of a directed graph G with n vertices and m edges is a cycle that traverses each edge of G exactly once according to its direction. Such a tour always exists if G is connected and the in-degree equals to the out-degree for each vertex in G. Describe an O(n+m)-time algorithm for finding an Euler tour of such a directed graph.

Problem 7 An independent set of an undirected graph G=(V, E) is a subset I of V such that no two vertices in I are adjacent. That is, if u and v are in I, then (u, v) is not in E. A maximal independent set is an independent set such that if any additional vertex is added to it, it will not be an independent set. Give an efficient algorithm for finding a maximal independent set for an undirected graph, and analyse its time complexity.

Problem Set 8 Solutions

Problem 1 The BFS (Breadth-First Search) algorithm given in the lecture notes uses multiple lists. Modify the algorithm so that it uses only one queue to replace multiple lists.

Solution:

The main algorithm remains the same. The breadth-first search algorithm starting with a vertex is modified as follows:

```
Algorithm BFS(G, s)
 {
  Create an empty queue L;
   L.enqueue(s);
   setLabel(s, VISITED);
   while  ( ! L.isEmpty() )
    {
      v = L.dequeue();
       for all  e ∈ G.incidentEdges(v)
                    if  ( getLabel(e) = UNEXPLORED )
                      {   w = opposite(v,e);
                          if  ( getLabel(w) = UNEXPLORED )
                              {
                                 setLabel(e, DISCOVERY);
                                 setLabel(w, VISITED);
                                 L.enqueue(w);
                              }
                           else
                               setLabel(e, CROSS);
                       }
    }
 }
```

Problem 2 Describe, in pseudo code, an O(n+m)-time algorithm for computing all the connected components of an undirected graph G with n vertices and m edges.

Solution:

```
Algorithm connectedComponents(G)
  Input: An undirected graph G
  Output: All the connected components of G
  {
    for each vertex v in G do
       visited(v)=0;
    i=0;
    for each vertex v in G do
```

```
    {
      if ( visited(v)=0 )
        {
          create a new list Li ;  // Each Li stores a connected component
          perform the depth-first search or the width-first search starting with v;
          for each vertex u visited in the search do
             { add u to Li;
               visited(u)=1;
             }
          i=i+1;
        }
    }
}
```

Problem 3 Given an undirected graph G and a vertex v_i, describe an algorithm for finding the shortest paths from v_i to all other vertices. The shortest path from a vertex v_s to a vertex v_t is a path from a vertex v_s to a vertex v_t with the minimum number of edges. What is the running time of your algorithm?

Solution: For each vertex v we introduce a list Q(v) to store the shortest path from v_i to v. We can modify the breadth–first search algorithm given in Q1 to compute the shortest path from v_i to v as follows:

```
Algorithm BFS(G, vi)
 {
   for each vertex v of G do
       Create an empty list Q(v);
   Create an empty queue L;
   L.enqueue(vi);
   setLabel(vi, VISITED);
   while  ( ! L.isEmpty() )
    {
      v = L.dequeue();
       for all  e ∈ G.incidentEdges(v)
                    if  ( getLabel(e) = UNEXPLORED )
                      {  w = opposite(v,e);
                         if  ( getLabel(w) = UNEXPLORED )
                            {
                               setLabel(e, DISCOVERY);
                               setLabel(w, VISITED);
                               L.enqueue(w);
                              Q(w)=Q(v)+{v,w};
                            }
                         else
                              setLabel(e, CROSS);
                      }
    }
 }
```

The first for loop takes O(n) time, and "Q(w)=Q(s)+{v,w}" takes O(1) time. Hence, the running time of the algorithm is O(m+n).

Problem 4 A connected undirected graph is said to be biconnected if it contains no vertex whose removal would divide G into two or more connected components. Give an O(n+m)-time algorithm for adding at most n edges to a connected graph G, with n>3 vertices and m>n-1 edges, to guarantee that G is biconnected.

Solution: Number the vertices 0 to n − 1. Now add an edge from vertex i to vertex (i + 1) **mod** n, if that edge does not already exist. This connects all the vertices in a cycle, which is itself biconnected.

Time complexity analysis:
We can use an adjacency list to represent G.

1. It takes O(n) time to number all the n vertices.
2. Adding an edge from vertex i to vertex (i+1) mod n takes O(degree(i)) time, where degree(i) is the degree of vertex i. Since the total degree of all the vertices is 2m, where m is the number of edges in G. Therefore, the time complexity of adding all the edges is O(2m)=O(m).

As a result, the time complexity of this algorithm is O(n)+O(m)=O(n+m).

Problem 5 An n-vertex directed acyclic graph G is **compact** if there is some way of numbering the vertices of G with the integers from 0 to n-1 such that G contains the edge (i, j) if and only if i<j, for all i, j in [0, n-1]. Give an $O(n^2)$-time algorithm for detecting if G is compact.

Solution:

```
Algorithm compactGraphChecking(G)
  Input: A directed graph G
  Output: true if G is compact; or false
  {
    Perform topological sorting on G;
    Let TSN(vi) be the topological number of vertex vi.
    for each vertex vi in G do
      TSN(vi)= TSN(vi)-1;
    Let a[0:n-1] be an array of all the vertices sorted in increasing order of their
topological numbers;
    for i=0 to n-2 do
      for j=i+1 to n-1 do
          if  no edge exists from a[i] to a[j]
             return false;
    return true;
  }
```

Running time analysis: Topological sorting on G takes O(n+m) time, where e the number of edges of G. The array a[] can be obtained by modifying the topological sorting algorithm without changing its time complexity. The first **for** loop takes O(n) time. The nested **for** loop takes $O(n^2)$ time. Therefore, the total running time is $O(n+m)+O(n)+O(n^2)=O(n^2)$.

Problem 6 An Euler tour of a directed graph G with n vertices and m edges is a cycle that traverses each edge of G exactly once according to its direction. Such a tour always exists if G is connected and the in-degree equals to the out-degree for each vertex in G. Describe an O(n+m)-time algorithm for finding an Euler tour of such a directed graph.

Solution: Hierholzer proposed an efficient algorithm in 1873 which works as follows:

1. Arbitrarily select a starting vertex v.
2. Find a cycle C starting and ending at v such that C contains all the edges going into and out of v, and insert each edge of C into a doubly linked list L in the order of the cycle C.
3. Traverse L to find the first vertex v′ which has an outgoing unvisted edge. If such a vertex v′ is not found, the algorithm terminates. Otherwise, let (v′, v′′) be the next edge in L when v′ is found, and find a cycle C′ starting and ending at v′ such that C′ contains all the edges going into and out of v′, and insert each edge of C′ into L in the order of the cycle such that all the edges of C′ appear before (v′, v′′) in L. Repeat this step starting at the first edge of C′ in L.

This algorithm visits each vertex and each edge in O(1) time. Therefore, the time complexity is O(m+n), where m is the number of edges and n is the number of vertices in the graph. Since a directed graph with an Euler tour must be strongly connected, we have O(m+n)=O(m).

Problem 7 An independent set of an undirected graph G=(V, E) is a subset I of V such that no two vertices in I are adjacent. That is, if u and v are in I, then (u, v) is not in E. A maximal independent set is an independent set such that if any additional vertex is added to it, it will not be an independent set. Give an efficient algorithm for finding a maximal independent set for an undirected graph, and analyse its time complexity.

Solution: We can use a greedy algorithm to find a maximal independent set as follows:

1. Create an empty set maxSet.
2. Arbitrarily select a vertex v, and add v to maxSet.
3. Repeat the following until no vertex can be added to maxSet:
   - Find a vertex v′ that is in V and not adjacent to any vertex in maxSet, add v′ to maxSet.

Time complexity analysis: Assume that the adjacency matrix structure is used to represent the input graph. Adding one vertex to maxSet takes O(n) time. Notice that the maximum size of a maximal independent set is n, where n is the number of vertices in the graph. Therefore, the time complexity of this algorithm is $O(n^2)$.

Comments: The maximum independent set problem is different from this problem, and NP-complete.