English exam

Solution sketches to Written exam ID1020 Algorithms and Data structures

Tuesday 2019-10-22 kl 8.00-13.00

Examiner: Robert Rönngren
Allowed aiding material: None

Each sheet of paper handed in should have the following information:
1. Name and social security number (personnummer)
2. Number for the question
3. Page number

Only write solutions to a single question on each sheet of paper (does not apply for the multiple choice questions special answer sheet).

The exam consists of two parts. Part I covers five areas:
- The fundamentals
- Sorting
- Searching
- Graphs
- Applications

with two questions per area for the grade E.

Part II, which is not compulsory, contains four questions for grade A-D.

Solutions to each question will be graded with 0-10 points.

To pass Part I, i.e. get grade (E), you need a minimum of 12 points per area in Part I. For Fx the requirement is a minimum of 12 points on four of the areas and a minimum of 10 points on the remaining area. You need to pass Part I to be able to obtain a higher grade (A-D). Points from Part I is not transferred to Part II.

Points, answers and grading:

For multiple choice questions: Identify correct and incorrect propositions. Each correctly identified proposition gives 1 point. 10 points are awarded if all eight propositions are correctly. Your answers should be marked and handed in on the special answer sheet.

For the other questions: Complete solutions with motivations/explanations are required. That is, answers without motivations are normally graded with 0 points.

If you have reached grade E on Part I the following apply for higher grades from the score on Part II.

_ Grade D: 5-10 points
_ Grade C: 11-20 points
_ Grade B: 21-30 points
_ Grade A: 31-40 points

Solution suggestions will be posted on Canvas.

Part 1

NOTE: Many students seems to fail to read the instructions for the exam. Specifically how the exam will be graded which states that answers have to have a short motivation/explanation. Failing to provide explanations/motivations typically result in zero points for the answer

The Fundamentals:

Multiple choice question: The Fundamentals question 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. Recursion always uses stack space on the order of the size of the input
   FALSE: There is no general connection between the depth of the recursion and the size of the input

2. An advantage of Abstract Data Types (ADTs) is that one can change the internal implementation of the ADT without it having any effect on applications using it
   FALSE: While the services offered by the ADT should remain unaltered the performance in terms of memory usage and execution time could change which in turn could affect the application in terms of what sizes of problems that could be handled. Consider for example sorting where Insertionsort and Mergesort both are stable while one is in-place but $O(N^2)$ and the other is not in-place but O(Nlog(N)). They both sort data and preserve the relative order of elements with identical keys but do so using considerably different execution times and memory. (Object orientation is good fro facilitating code re-use but it does not mean that the internal implementation does not affect performance)

3. Given that the same elements have been inserted into a FILO queue and a stack, then a `dequeue`() operation on the queue and a `pop()` operation on the stack would return the same element
   TRUE: First-In Last-Out is the same as Last-In First-Out which is the queueing policy of a stack

4. An array based list implementation of a queue always has less memory overhead than a linked list implementation
   FALSE: First assume the data stored in the queue is represented by objects in JAVA and structs in C. The array will hold one reference/pointer per index. An element in a single linked list will, for JAVA have an overhead of an object and two references (a next reference and a reference to the data object) and in C only two pointers (next and data). In C the linked list uses less memory than the array if the array is less than half-full while for JAVA the overhead of an object determines the break-even point. If the object overhead is of the same size as a reference the linked list will use less memory if the array is less than one-third full.

5. The performance of an algorithm which has Big-Theta complexity can be assumed to be similar for all input of the same size (N)
   TRUE: Big-Theta means that the worst and best case performance is within a constant.

6. An algorithm that is O(Nlog(N)) may be less efficient than an $O(N^2)$ algorithm for some inputs
   TRUE: Example – Insertion sort is O(N) for input with few inversions while Merge- and Quicksort are O(Nlog(N)) for such input

7. A recursive algorithm can never use O(1) memory
   FALSE: If tail-recursion is supported recursion may use constant stack space

8. It is simpler to implement a LIFO-queue based on resizing arrays than it is to implement a FIFO-queue based on resizing arrays
   TRUE: In a stack (LIFO) we only access the top element of the stack. This means that all elements of the stack will be placed from index 0 and onwards, i.e. sequentially from the beginning of the array. When resizing the array it is therefore straight-forward to copy elements from the old array to the new. For a FIFO-queue we will add elements at one end and remove from the other. This will have the effect that the queue will move in the array so that the beginning of the queue could be located at the end of the array while the end of the queue may be located at the beginning of the array. Thus, when resizing the array we must check whether the elements of the queue need to be

reordered when copying the elements of the queue to the new array which is a little bit tricky and more time consuming.

The Fundamentals question 2:

Circular double linked lists with a "*sentinel*"-element (which marks the beginning/end of the list) can be used to implement different data types including queues. Assume you have access to a function/method as found below:

```
void addElement(element P, element N) // N is a reference/pointer to the element to be added
begin
  N.next = P.next;
  N.prev = P;
  P.next.prev = N;
  P.next = N;
end
```

In JAVA or C, implement the following:

As you could solve this in either JAVA or C we outline the solutions. If you are uncertain on things as lists check the chapter on Fundamentals in the book and the Preparatory lab in Canvas…

a) A data structure, `element`, to implement elements of the list containing an `integer` 2p

Create a class (JAVA) or a struct (C) element which contains

element next, prev; //pointers/references
integer data;

NOTE: Make sure you understand the difference between a data structure (how data is stored) and an algorithm (how data is manipulated/processed)

b) A method/function, `createList(?)`, to create an empty list 2p

An empty list consists of only a sentinel element where the next and prev pointers/references point to the sentinel element self. Use the same data type for the sentinel element as for other elements. Assume the sentinel is referred to by a reference/pointer **head**. The steps are:
1) Allocate memory for an element (referred to by **p**), in JAVA by a constructor, in C by an explicit call to malloc()
2) p.next = p.prev = p;
3) p.data = some number //the sentinel does not hold any data – it is just a marker – so it could be any number
4) return p;

c) A method/function, `insertFIFO(?)`, to insert an element in FIFO-order in a list 2p

In a FIFO-queue we add new elements last in the queue, i.e. after the last element in the queue which we find directly as head.prev

void insertFIFO(element head, element n)
{ addElement(head.prev, n); }

Note: The order in which the elements are stored in the list is of importance. It is reasonable to assume that applications could want to iterate over the elements in the list and for a FIFO-queue you would then expect to get the elements in FIFO-order when following the next references/pointers from the head, i.e. with the first element inserted to the queue as head.next

d) A method/function, `insertStack(?)`, to insert an element in a list implementing a stack 2p

the top of the stack should be at head.next, i.e. new elements should be inserted after head

void insertStack(element head, element n)
{ addElement(head, n); }

e) A method/function, `removeFirst(?)`, which removes and returns the first element of a list if it is non-empty 2p

```
element removeFirst(element head)
{ element first;
  if(head == null || head.next == head) return null; //empty list – no element to remove
  first = head.next;
  head.next = head.next.next;        //restore list
  head.next.prev = head;
  first.next = first.prev = first;       //reset pointers/references to avoid loitering and accidental use
  return first; }
```

Sorting

Multiple choice question: Sorting 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. A loop-invariant that shows that the entropy of the data sorted by a sorting algorithm is decreasing by each iteration of the algorithm can be used to show the correctness of the algorithm
   TRUE: Decreasing entropy (entropy is a term used in many areas is a measure of disorder) means that the disorder decreases and when the entropy becomes zero the elements of the array will be ordered (sorted)

2. Selection sort is among the fastest algorithms to sort data with few inversions
   FALSE: Selectionsort cannot make use of this fact while Insertionsort is near linear in time if the input has few inversions

3. An inversion is defined as two adjacent elements being out of order
   FALSE: An inversion is any two elements out of order regardless of whether they are adjacent (direct neighbors) or not

4. The execution time of a stable sorting method is insensitive to the properties of the input
   FALSE: Stability refers to that the sorting algorithm preserves the relative order between elements with identical keys. A counter example is Insertionsort which is stable but sensitive to the number of inversions in the input.

5. The performance of Quicksort, for general cases, depends on the quality of the random number generator in the system
   TRUE: The performance of Quicksort heavily depends on that the input is randomized. This means that we typically shuffle the input before sorting it and the quality (randomness) of the shuffle depends on the random number generator used which typically is the system random number generator

6. Mergesort is in-place
   FALSE: Mergesort typically use O(N) extra memory while in-place means that we use an additional memory of (1) or possibly O(log(N))

7. A priority queue implemented by a linked list has O(1) dequeue and O(N) enqueue operations
   TRUE: If the elements are kept sorted in the queue we can access the first element of the queue, the element with the highest priority directly in O(1) time while we would have to step through and compare O(N) elements when sorting a new element into the queue

8. It is not possible to sort N random 16-bit integers in less time than O(Nlog(N)) time
   FALSE: One can sort such data in O(N) time by simply counting the frequency (number of times each number occurs in the input) of the numbers, as shown at the lectures

Sorting question 2:

Show, step-by-step, how the following data: `[2,4,8,3,2]` is sorted by:

a) Insertionsort 2p

Required: Explain briefly how the sorting method works and show the steps performed when sorting the data (see the book). Note: Insertion sort swaps adjacent elements, it does not move elements longer steps at a time

b) Mergesort 2p

Required: Explain briefly how the sorting method works and show the steps (see the book)

Explain which sorting algorithms are used, and why these algorithms are used by the standard libraries in JAVA for sorting arrays of:

c) reference data types 2p
For reference data types, objects with identical keys (to sort by) may still be different why it may be important to preserve the relative order in the sorting process for such objects. Hence one would prefer to use a stable sorting method which is effective also for large inputs. Thus Mergesort is used.

d) primitive data types 2p
For primitive data types, identical keys are truly identical (i.e. one cannot distinguish one 4 from another 4) why it is not important to preserve the relative order in the sorting process. Hence one would prefer to use the most efficient (in time and memory) sorting method which is effective also for large inputs. Thus Quicksort is used.

Text/strings are normally implemented as one-dimensional arrays of characters. In languages as JAVA you also have access to how long, i.e. how many characters a string contains.

e) Describe in pseudo-code and words, how one can implement a `compareTo()`-method which compares two strings where the length of the strings are known 2p

The natural way of comparing strings is to compare them alphabetically which we do by comparing character by character. To do this properly one need to know the length of the strings.

(pseudo code, assume the characters are stored in a field called data, and the string length in a field length

the compareTo() interface means that we should return the following: if a < b return a negative number (-1), if a > b return a positive number (1), if a == b return 0)

```
int compareTo(string a, string b)
{
  // get the min length of the strings
  int i, minLength = (a.length< b.length) ? a.length: b.length;
  // check for difference in minLength parts of the strings (Unicode are small integer numbers…)
  for(i=0; i < minLength; i++)
     if(a.data[i] – b.data[i]) return a.data[i] – b.data[i]; //assumes a value of 0 is false and non-zero is true
  // strings are equal up to the min length
  // equal length?
if(a.length == b.length) return 0;
  //is a shorter than b?
  return (a.length < b.length) ? -1: 1;
}
```

Searching

Muliple choice question: Searching 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. Tree based Symbol Tables (STs) can use non-integer keys without the use of a separate translation table to translate symbolic (non-integer) key values to integer values
   TRUE: Tree based STs only need to be able to compare keys (while hash tables use keys as indices in an array, i.e. it needs to use integer keys or map non-integer keys to integers)

2. Hash table STs typically have amortized O(1) access times for both look-ups and insertions
   TRUE: see book

3. A general ST can be efficiently implemented by a sorted array and binary search
   FALSE: A general ST need to implement both inserts and look-ups. A ST implemented by a sorted array has the disadvantage that the array needs to be sorted if one or more keys has been inserted before a look-up can be performed. This kind of implementation is only effective if there are few inserts (or all inserts are done before the look.ups)

4. The time complexity of an ordinary binary search tree (not red-black or 2-3-tree) is O(N) for insertions and O(log(N)) for look-ups
   FALSE: A look-up in a binary search tree (BST) is only O(log(N)) if the tree is balanced which cannot be guaranteed for a BST. The worst time complexities for a BST are O(N) for inserts and look-ups. Hence one should in general avoid using BSTs (without balancing) as a basis for symbol tables.

5. A hash table based on linear probing uses less memory on average than a hash table based on separate chaining
   TRUE: In the average case – see the book

6. To find all keys associated with a specific value in a ST, i.e. performing a reverse look-up, is an O(N) operation in a hash table and an O(log(N)) operation on a tree based table
   FALSE: To find all keys associated with a specific value in a ST requires that one checks every key to see which value it is associated with, i.e. it requires inspecting each of the N keys - O(N) operations

7. In JAVA, hashcode and hash refer to the same thing
   FALSE: Hashcode is a 32-bit integer which is used as a basis for calculating the hash

8. A 2-3-tree is an efficient and easy to implement alternative to a left leaning red-black tree
   FALSE: A 2-3-tree is complex to implement which is the reason for why red-black trees (and AVL-tress) have been introduced

Searching question 2:

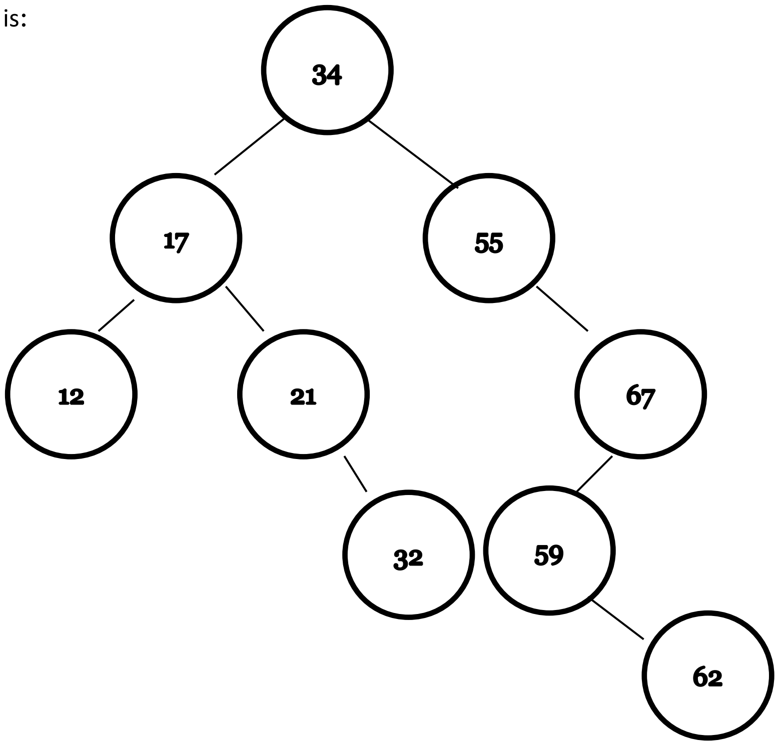
Use the following as input for the questions `[34, 17, 12, 55, 21, 67, 59, 62, 32]`

Show the result from building a Binary Search Tree from the input above 2p

Explain how the tree is built: i.e. that keys smaller than the key of a node go left, otherwise right

the resulting tree is:

a)



b) Show the result from printing the content of the tree from a) in prefix-order 2p

prefix: first print node content, process left sub-tree, process right sub-tree

34, 17, 12, 21, 32, 55, 67, 59, 62

c) Show the result from printing the content of the tree from a) in infix-order 2p

infix: process left-sub-tree, print node content, process right sub-tree

12, 17, 21, 32, 34, 55, 59, 62, 67

d) Show the result from printing the content of the tree from a) in postfix-order 1p

postfix: process left sub-tree, process right sub-tree, print node content
12, 32, 21, 17, 62, 59, 67, 55, 34

e) Show the result from building a hash table based on linear probing with the input above when the hash is calculated as `number%10` 3p

the size of the array in the hash table is 10 when we start. When the sixth key is inserted the array size will be doubled to 20 and all keys will be rehashed by the hash function *number%*20 (10 is chosen to facilitate your calculations, it would have been better to use a prime number such as 11 – but that would have made the calculations unnecessarily difficult). you should explain how the table is built in particular collision resolution! You should show the intermediate results and the en-result will be:
-, 21, 62, -, -, -, -, 67, -, -, -, -, 12, 32, 34, 55, -, 17, -, 59

Where we had one collision resolution when 32 is inserted as index 12 is already occupied by key 12 when inserting 32

Graphs

Multiple choice question: Graphs 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. A strongly connected component of a directed graph contains at least one cycle
   TRUE: follows from that we should be able to reach any vertex from any other vertex

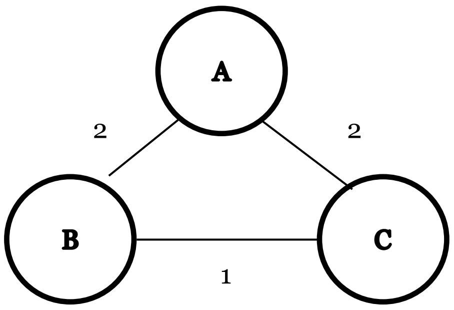
2. Depth-first search in an undirected graph with unit edge weights calculates the shortest path from a source vertex to a sink vertex
   FALSE: Depth-first-search finds one path (if it exists) which need not be the shortest path

3. A scheduling problem can be modeled by a directed graph where precedence constraints are modeled as zero weight edges
   TRUE: see the book. A job is modeled as two vertices with an interconnecting edge with a weight which represents the maximum time the job may take. Precedence constraints do not take any time, hence they are modeled as zero-weight edges (otherwise one could not calculate the time it takes to perform a series of jobs or the critical path)

4. A data structure similar to that used to implement a hash table with separate chaining is an efficient structure for implementing a sparse graph
   TRUE: An effective data structure to implement sparse graphs is an array indexed by vertices (keys in a hash table) where each index contains a bag of edges (values in a hash table)

5. There may be more than one minimum spanning tree with the root in a specific vertex in an edge weighted graph with arbitrary edge weights
   TRUE: If we can have different edges with equal weights there may be more than one equal weight spanning tree

   There are two MSTs staring in A:
   MST1: A,B,C
   MST2: A,C,B
   both with weight 3



6. To relax an edge in a graph processing algorithm means that the edge is identified as not being part of the path/tree that the algorithm is finding
   FALSE: To relax an edge means finding out whether the edge should be part or not of the path or tree the algorithm is building

7. The Bellman-Ford algorithm can be used to find Hamiltonian cycles in any edge weighted digraph with non-negative edge weights if such a cycle exists
   FALSE: There are no (known) algorithms to calculate Hamiltonian cycles

8. The shortest path between any two vertices in strongly connected component with one negative cycle in an edge weighted digraph must contain the negative cycle
   FALSE: Shortest path is not defined for paths containing negative cycles (it does not make sense as the path gets shorter for each iteration in a negative cycle, hence you should iterate infinitely in the negative cycle to get the shortest path possible but then you would never reach the destination (sink) vertex)

Question Graphs 2:

a) Which of the graph algorithms covered in the course is/are suitable to use in a navigation app? 2p

The typical problem in a navigation app is to find the shortest path, by time, distance etc., between two vertices in an edge weighted digraph (the graph is a digraph as we may have one-way streets/connections for vehicles or conveyor-belts for pedestrians). Further we can assume there will be no negative edge weights. Thus Dijkstra's algorithm to find shortest paths is the best choice of the algorithms covered in the course (Bellman-Ford is a less efficient alternative)

b) Describe what Prim's algorithm is used for and the principles for how it works. 3p

Prim's algorithm is a greedy algorithm to find a minimum spanning tree in an edge weighted graph. It works as follows:

- Start with vertex 0 (we select some vertex to be the staring vertex which we will number as vertex 0) and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Mark each visited vertex as visited
- Repeat until *V* – 1 edges.

It uses a priority queue where edges connected to the tree (but not yet in the tree) to be considered for relaxation is kept ordered by low edge weight.

c) Show if the following proposition is true or false:
"In a digraph the shortest path from vertex ***v*** *to vertex* ***w*** *must be part of any minimum spanning tree with the root in vertex* ***v****"* 3p

FALSE: For a counter example see the Graphs multiple choice question 5 above, The shortest path from A to C is the direct edge from A to C which is not part of the MST1: A, B, C. Note: a good strategy when faced with a problem of showing that something is true or false and you do not know the immediate answer is to try to find a counter example…

d) Identify which vertex/vertices could be the first (starting) vertex in a topological sort of the following digraph
TA-TB, TA-TF, TC-TA, TD-TF, TF-TE, TG-TE, TH-TG, TI-TH, TJ-TK, TJ-TL, TJ-TM, TL-TM 1p

A starting vertex is a vertex without any incoming edges, i.e. in-degree 0. That is TC, TD, TI, and TJ are all possible starting edges

e) Identify which vertex/vertices could be the last (end) vertex in a topological sort of the following digraph
TA-TB, TA-TF, TC-TA, TD-TF, TF-TE, TG-TE, TH-TG, TI-TH, TJ-TK, TJ-TL, TJ-TM, TL-TM 1p

An ending vertex is a vertex without any outgoing edges, i.e. out-degree 0. That is TB, TE, TK and TM may all be ending vertices.

Applications

Multiple choice question: Applications 1

Identify correct and incorrect statements. Mark your answers on the separate answer sheet for the multiple choice questions.

1. Software caching is an example of an execution time vs. memory usage trade-off
   TRUE: In software caching one stores/saves a value for later usage instead of re-calculating it, thus using some more memory to reduce execution time. An example is how hashcodes typically are implemented in JAVA where the calculation of the hashcode is time consuming. Thus the hashcode for a JAVA object is typically only calculated once when it is first accessed/used and then stored in the object to be immediately accessible for future use.

2. The callback mechanisms we have studied in the course means that we return to solve a sub-problem at a later stage in the execution of an algorithm
   FALSE: An example of a callback is when a sorting algorithm makes a call back to an object, i.e. calls the compareTo()-method of the object, to order objects

3. A linked list based priority queue is normally faster than a heap based priority queue when the number of elements in the queue is on the order of some ten elements
   TRUE: This is an example of where an O(N) algorithm may be faster than an O(log(N)) algorithm for small N (the big-Oh notation is asymptotic meaning you can only compare algorithms of different complexity classes by their big-Oh notation when N is large)

4. It is often necessary to understand the properties of the input to select the best algorithm for a given problem
   TRUE: To select the best algorithm you need to understand the properties of the input

5. It is not possible to create priority queue implementations which have amortized access times for both enqueue and dequeue operations that are lower than O(log(N))
   FALSE:For example the Calendarqueue (see Canvas) is an example of a (hash table like) structure which for many applications has amortized O(1) access time for enqueue and dequeue operations

6. Mergesort is, in practice, the fastest stable sorting algorithm
   TRUE: see book

7. Selectionsort is an example of a *Divide-and-conquer* algorithm
   FALSE: A divide-and conquer algorithm divides a large problem into smaller sub-problems which each will be solved separately and then the original problem is solved by combining the solutions to the sub-problems. Merge-and Quicksort are examples of such algorithms (while Mergesort is not)

8. The value associated with a key in an indexing application based on a Symbol Table is typically a FIFO queue.
   TRUE: An indexing application means that we build an index of where in a data set keys appear. An example could be an index of a book which tells where (on which pages) words/concepts occur. Typically one would like the index to present the occurrences in the order of appearance of the words/concepts in the text. Assuming the index is built by reading the text from the beginning to the end, a FIFO-queue is a good data structure to use. See the labs.

Question Applications 2:

a) Explain which data type is more effective to use in JAVA when sorting an array of integer numbers: `Integer` or `int`? 3p

To compare two int by < is (much) more efficient (less time consuming) than going through the objects using the call-back mechanism, compareTo() of Integers. Furthermore one could assume that an array of non-reference types often has better cache performance (i.e. all elements lies close to each other in memory) then an array of reference data (where the elements may be more dispersed in memory)

b) What are the big-Omega, big-Oh and big-Theta for the pieces of code below? 2p

A:

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    a[i][j] = a[i][j] * a[i][j];
```

B:

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    a[j][i] = a[j][i] * a[j][i];
```

For both A and B the inner for-loop will perform N iterations which will be performed N-times by the outer for-loop hence big-Omega, big-Oh and big-Theta will be N^2 for both A and B

c) What would you expect to find if you compared the execution times for the pieces of code above for large N? 3p

Since computer memory are one-dimensional, multi-dimensional arrays (matrices) has to be mapped to this one-dimensional memory. This is done by placing the rows after each other in memory. In the example A (above) we loop through the array row-wise which means that we will get good cache (and virtual memory) performance, i.e. good locality. In example B we will go through the array column-wise which will result in poor cache (and virtual memory) performance. Thus we would expect that example A will be (substantially – several orders of magnitude) faster to execute than example B for large N. Note: This example has been investigated in the lectures where the execution time for A was found to be ~1/100 compared to that of B for N=100000

d) Why is it preferable to use Insertionsort instead of Selectionsort when "CUTOFF" is implemented in Mergesort? 2p

For small amounts of data (sub-arrays with size ~10 elements) the overhead of doing recursive calls, both execution time and memory usage, is too large for it to be efficient to continue the recursive divide-and-conquer approach. Whereas an iterative method will be more effective. Insertionsort is stable while Selectionsort is not stable. So to preserve the property of stability Insertionsort is a better choice. Insertion sort also has better best case performance than Selectionsort while similar worst case performance, so on average it is also faster than Selctionsort.

Part 2

Complete solutions with motivations are required for all assignments. When grading the time and memory complexity of your solutions will be important.

Question 2.1:

a) Implement a program which calculates the **k** first numbers `(f(0), f(1) … f(k))` for the following algorithm in C or JAVA. 7p

```
f(0) = 1
f(1) = 1
f(2) = 1
f(n) = f(n-2) + f(n-3) for n >= 3
```

A naïve direct recursive implementation of the function will have a time complexity which is exponential. That is really bad and will only give 1p. A solution which saves results calculated to be reused at later stage, i.e. a dynamic programming approach will be linear in time (regardless if it is implemented as recursive or iterative) and will, if correct, give 7p Points will also be deducted if auxiliary (unnecessary) memory is used, unnecessary swaps are performed or the algorithm is unnecessarily complex.

b) What is the big-Omega, big-Oh and big-Theta time complexity for your implementation 3p

Depends on how it has been implemented. Correct identification of the complexity gives 3p in either case.
Note: big-Omega is not constant even if f(n) for n=0,1,2 can be calculated in constant time. Asymptotic complexity only holds for some N and upwards. For example Mergesort is not big-Omega(1) even if it can be used to sort an array of a single element in constant time.

Question 2.2:

a) Implement, in JAVA or C, a sorting algorithm which sorts an array of positive integers so that even numbers are sorted to come before (i.e. be placed on lower indices) than odd numbers in the array. 7p

See quick-sort partitioning. An effective linear time algorithm gives 7p.

b) What is the big-Omega, big-Oh and big-Theta time complexity for your implementation 3p

Depends on how it has been implemented. Correct identification of the complexity gives 3p in either case.

Question 2.3:

You are to implement a security system for a company which only should allow incoming traffic from certain IP-addresses to be let through. Your system should be able to answer questions on whether incoming traffic from an IP-address should be let through or not.

In this assignment you may use/base your solution on the algorithms and data structures introduced in the course.

The system should implement the API below:

```
void allowAddress(ipadress X)       //add address X to the addresses we allow incoming traffic from
boolean shouldAddressBeBlocked(ipadress X)    //should we allow incoming traffic from address X?
```

a) Describe your implementation/solution in words and pseudo code 7p

This is a simple ST-problem. Use a hash table as it has better time-complexity than tree based implementations. You should definitely not use an unbalanced BST.
allowAddress is implemented as insert a key (IP-address) and shouldAddressBeBlocked is implemented as a look-up. O(1) is needed to get 7p.

b) What is the big-Omega, big-Oh and big-Theta time complexity for your implementation/solution 3p

Depends on how it has been implemented. Correct identification of the complexity gives 3p in either case.

Question 2.4:

Your task is to design a system for scheduling real time systems. In real time systems there are jobs which are to be executed. Each job has a maximal execution time. A job may depend on other jobs which must have finished execution before the job can be started. For a real time system to be correct all jobs have to be finished within certain time constraints. Whether this is possible to achieve at all is limited by what is referred to as ”the critical path”. The critical path is the maximal time it may take from the first job is started to that the last job finishes.

In this assignment you may use/base your solution on the algorithms and data structures introduced in the course.

a) Describe how you represent jobs and dependencies (precedence constraints) between jobs 3p

The problem is modelled as an edge weighted digraph problem. A job is modelled as two vertices with an interconnecting edge where the weight represents the maximum time the job can take. Precedence constraints are represented by zero-weight edges.

b) Describe how you can assess if the jobs could be scheduled 3p

To assess if something is possible means that you should check if it is possible (or not). It is possible to find a schedule iff there are no cyclic dependencies in the graph. To check that there are no cycles could easiest be done by a modified depth-first-search. The schedule itself is found by doing a topological sort of the graph (describe how!)

c) Describe how you can calculate the length of the critical path 4p

As there should not be any negative edge weights we may calculate the critical path, defined as the longest path from a starting vertex to any ending vertex, by negating the edge weights and calculating the shortest paths from any starting vertex to any ending vertex in the topological sort. The shortest of these (negative edge weight) paths is the critical path. The length is easily calculated by simply negating the results (i.e. negating the edge weights back to positive weights).