Umeå University
Department of Computing Science
TDBC91 — Analysis of Algorithms
Examination: December 22, 2006

Name (printed): ______________________________

Swedish ID number: ______________________________

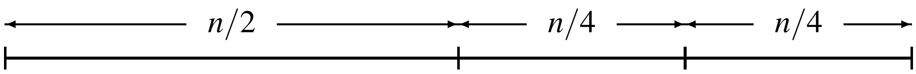
Computer User-ID: ______________________________

Signature: ______________________________

1. All answers must be written in English.
2. A pocket engineering calculator (miniräknare) and English/X - X/English dictionary may be used. No other help materials are allowed.
3. You must turn in this cover sheet together with the pages containing your answers. Collate the sheets in numerical order of the problems, and write on only one side of the paper.
4. The examination has a total of 400 points, which constitutes 40% of the course grade. The other 60% is based upon the obligatory exercises and problem sets, as explained in the course syllabus.
5. In many of the problem with multiple parts, the answer to later parts depends upon the answer to earlier parts. If the answer to an earlier part is wrong, and if that error makes the answer to a later part much easier or trivial, then full credit will not be awarded for that part even if the answer is correct relative to the previous error.
6. If you want to take a future examination to determine your final grade, and so do not want this examination graded, attach a sheet to your examination clearly stating this intention.
7. This examination consists of seven (7) problems on a total of four (4) pages.

(1: 50 points) Two distinct notions of asymptotic complexity were defined in the course notes and lectures, each recapturing the notion that function f is at least as complex as function g. They were represented as $f \in \Omega_1(g)$ and $f \in \Omega_2(g)$. Define each of Ω_1 and Ω_2, and briefly discuss the advantages and disadvantages of each. (It is *not* necessary that you remember which was called Ω_1 and which was called Ω_2. Just define and compare the two notions.) For full credit, your answer must include formal mathematical definitions, as well as less formal explanations of their respective meanings. Illustrate each with a simple example.

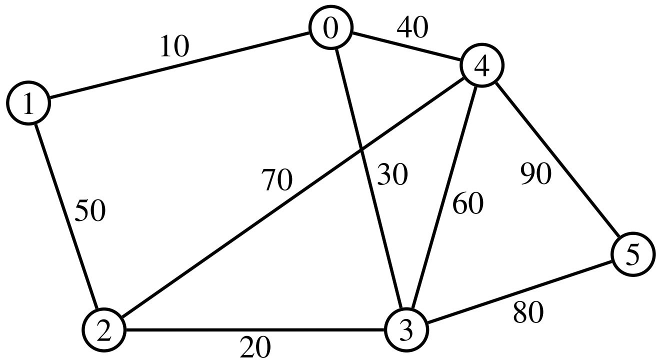
(2: 50 points) Consider a variation on the mergesort algorithm in which the array to be sorted is broken into *three* (instead of two) pieces, and then each piece is recursively processed using the same technique, with the three pieces then merged together. The three pieces are are not equal in size. Rather, a list of n elements is divided into one piece of size $n/2$, and two pieces of size $n/4$. (If n is not divisible by 4, then $n/2$ and $n/4$ must be rounded appropriately, but the details are not critical here.) The idea is recaptured in the picture below.



(a) Write down the recurrence relation which characterizes the time complexity of this algorithm. (You do **not** have to write down the algorithm; just give the recurrence relation which governs it.)

(b) Solve this recurrence relation, using whatever techniques are appropriate. You need not explicitly solve for constants, but you must produce the correct form of the solution. You must show intermediate work, including the characteristic polynomial and how you obtained it, as well as any variable transformations which were necessary in the solution.

(3: 60 points) Depicted below is an undirected graph with weighted edges.



(a) Show the sequence of edges which are considered, and which are selected, in applying Kruskal's algorithm for identifying a minimum spanning tree. Thus, you must show the edge considered, and whether or not it is selected, at each step. Represent each edge in your solution by the numerical labels on the vertices which are its endpoints.

(b) Repeat part (a) for Prim's algorithm.

(c) Kruskal's algorithm is a *formal greedy algorithm*, while Prim's is not. Explain clearly what is meant by a formal greedy algorithm, and why this definition holds for Kruskal's but not for Prim's approaches. State further the advantage of knowing that an algorithm is formal greedy.

(4: 60 points total)

(a) Give a concise but clear definition of the abstract data type *priority queue*. The following information must be provided.

   (i) The underlying type on which it operates.

   (ii) Each of its operations, including the domain and range (*i.e.*, input and output types), as well as a clear description of what the operation does.

   Do not describe implementation here in part (a); that issue is addressed in (b) below.

(b) Describe the data structure *min-heap*, including:

   (i) how it is implemented using arrays; and

   (i) how the operations of a priority queue are represented within it.

(c) State (proof is not required) the *worst case* and *average case* time complexity for each of the operations of a priority queue which is implemented as a min-heap as described in (b) above.

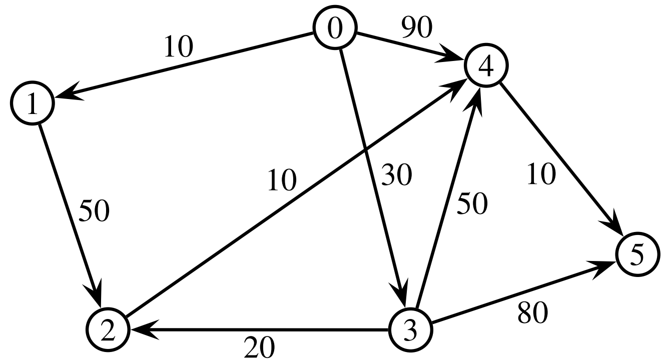
(d) State what is meant by an *adjustable priority queue*, emphasizing how it extends the notions defined in part (a). In particular, identify the operations which it adds to an ordinary priority queue.

Note: For part (b) above, the implementation must be the one presented in the book and course notes, or else an implementation for which all operations have a worst-case and average case time complexity which is no worse than for that of the implementation presented in the course.

(5: 60 points) Answer the following questions regarding Dijkstra's algorithm for the single-source shortest-path problem.

(a) Define what is meant by the *single-source shortest-path problem*.

(b) On the directed graph shown below, with source vertex 0, illustrate how Dijkstra's algorithm proceeds, step-by-step, in solving the problem. You can either put your solution in the table below, or else put a similar table with your solution on your answer sheet. If you fill in this table, then you must clearly indicate, on your answer sheet(s) for this problem, that the answer is to be found on this sheet.



Step	Other vertices allowed in path	To vertex 1	2	3	4	5	Nearest vertex
1							
2							
3							
4							
5							
6							

(c) Sketch how Dijkstra's algorithm can be solved efficiently using an adjustable priority queue. Your answer to this question may (and should) refer to your answer to problem 4(d) above.

(6: 60 points) Given are two polynomials

$$p_1(x) = \sum_{k=0}^{n_1-1} a_k \cdot x^k \qquad\qquad p_2(x) = \sum_{k=0}^{n_2-1} b_k \cdot x^k$$

Answer the following questions regarding the DFT and FFT.

(a) Define the *convolution* of p_1 and p_2, and state the time complexity of the naïve (straightforward) algorithm for solving this problem.

(b) Define the *pointwise multiplication* of p_1 and p_2, and state the time complexity of the naïve (straightforward) algorithm for solving this problem.

(c) Define the m^{th}-degree *discrete Fourier transform* (DFT) of p_1. If you cannot give a precise definition, you can still obtain partial credit for stating the size and nature of the transformation (*e.g.*, the family of basis functions which it uses).

(d) Repeat (c) for the *inverse DFT*.

(e) Give the formula which computes the convolution of p_1 and p_2 in terms of pointwise multiplication, the DFT, and the inverse DFT. Make sure that you indicate the degree of the DFT (and inverse DFT) which is required.

(f) The *Fast Fourier Transform* (FFT) is a special algorithm for computing the DFT. State its worst-case time complexity, and sketch briefly how it is used to realize efficient convolution of polynomials. Indicate also the overall complexity of this procedure for computing convolution.

(7: 60 points) Answer the following questions in the context of the 0/1-knapsack optimization problem.

(a) Define what is meant by an ε-*relative* approximation algorithm.

(b) Sketch a $1/(k+1)$-relative approximation for the 0/1-knapsack optimization problem which has worst-case time complexity $\Theta(n/(k+1))$. Here n is the problem size, while k is a parameter of the algorithm which you should define carefully. No proofs of complexity or correctness are required; just sketch the algorithm.

(c) Define what is meant by a *fully polynomial-time* approximation algorithm.

(d) Explain clearly why the algorithm identified in (b) is not fully polynomial time.

(e) Sketch a fully-polynomial time approximation algorithm for the 0/1-knapsack problem. State the worst-case complexity of this algorithm in terms of the problem size n and the parameter ε. No proofs of complexity or correctness are necessary. Three such algorithms were given in the course; any one of these is suitable as an answer.