CORE EXAMINATION
Department of Computer Science
New York University
January 23, 1998

This is the common examination for the M.S. program in CS. It covers core computer science topics: Languages and Compilers, Operating Systems, and Algorithms. The exam has two parts. The first part lasts three hours and covers the first two topics. The second part, given this afternoon, lasts one and one-half hours, and covers algorithms.

Attempt all of the questions. You are not required to take the algorithms section of the exam if you have passed the FOCS exam in the past.

Use the proper booklet for each question. Each booklet is marked with the Area and Question number, in the form PL&C1, PL&C2, PLC&C3, OS1, OS2, ALGS1, ALGS2, ALGS3. Use the appropriate booklet for each question. DO NOT put your name on the exam booklet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope. You can keep the exam.

Good luck!

Programming Languages and Compilers

Question 1

a) Describe briefly the purpose of an exception mechanism in a modern high level language (e.g. Ada, C++, or Java). Explain how a exception propagates in Ada.

b) What output does the following Ada program produce when executed?

```
with Text_Io; use Text_Io;
procedure demo is
   E : exception;

   procedure P (N : Integer) is
      F : exception;
   begin
      Text_Io.put (" Call P");
      if N = 2 then raise E: end if;
      if N = 3 then raise F; end if;
   exception
      when E => Text_Io.Put_Line ("E1");
      when others => Text_Io.Put_Line ("Others1");
   end P;

begin
   P (1); P (2); P (3); P (4);
exception
   when E => Text_Io.Put_Line ("E2");
   when others => Text_Io.Put_Line ("others2));
end;
```

Solution

a) Exceptions and exception handlers in both Ada and Java are designed to handle situations when the normal program execution has to be interrupted to perform some special processing. For example: error conditions (arithmetic overflow, out-of-bound array references, unexpected end-of-file), unpredictable conditions, tracing and monitoring, etc.

An exception can be raised either by a language-defined primitive operations (e.g. constraint-error on arithmetic or an index out of bounds), or by an explicit action in the user-code (raise_statement in Ada, throw in Java or C++). The effect of the raise, regardless of its origin, is to transfer control to the handler that is dynamically associated with the exception being raised.

The handler (if any) is determined by following the dynamic chain of sub-program activations leading up from the frame that raised the exception, until a handler is found for the exception being raised. If none is found, the program terminates.

b)The program prints:

```
Call P
Call P
E1
Call P
others1
Call P
```

Question 2

In the following, you are asked to write some code in the language of your choice. Your answer will be judged on the correctness, the clarity and the generality of your code in that language. Pseudo-code is not acceptable.

- In Ada, C, C++, or Java, write a type definition for a singly-linked list. Using your definition, write a code fragment in the same language that adds an element to an existing list.
- How can your definition be used to define lists with different element types (lists of integers, lists of lists of strings, etc.)?
- Can your definition be used to define heterogenous lists, that is to say lists that contain different element types? What are consequences of hetero- geneous lists for type-safety, in your chosen language?
- Using your definition, write a function or method that returns the length of a list, that is to say the number of elements it contains.

Solution

- Everyone can write a linked structure in one of the proposed languages. The link is expressed as a pointer type in C or C++ (T* for some T) or as an access type in Ada (type Link is access Node). In Java, reference semantics makes it possible to write the simple self-referential definition:

  ```
  class Node {
     int  value;
     Node next;
     ...
  }
  ```

- The most general solution is a generic one, in which the type of the item stored in each node of the list is a parameter to the definition. This means a generic package in Ada, or a class template in C++. No such facilities are available in C or Java. In C, one may use void* to indicate that the item type is arbitrary. The use of the root class Object in Java has the same effect.

- The C and Java solutions mentioned above allow the creation of arbitrary heterogeneous lists. In C++ and Ada, heterogeneous lists can be created only for types (or classes) related by inheritance. For example, in C++. one might define the element type to be T*, in which case any descendant of T can be stored in a list node.

  In all cases, heterogeneous lists are not statically typed, and type-checking may have to take place at run-time, either in the form of dynamic dispatching, or with explicit casts. This means that some kind of run-time identification must be available. Ada tagged types, C++ and Java classes carry type identification, so that the run-time can verify the legality of a cast or conversion. No such is available directly in C, so one would have to program explicitly the equivalent of discriminated types to encode type information.

- Writing a loop that traverses a list in not a big problem. If the list has an explicit header (rather than being represented by a pointer to its first node) it is reasonable to define a class member that holds the count of the number of elements in the list. The count must be maintained correctly under insertions and deletions, which is not necessarily cheap if these operations happen in the middle of the list.

Question 3

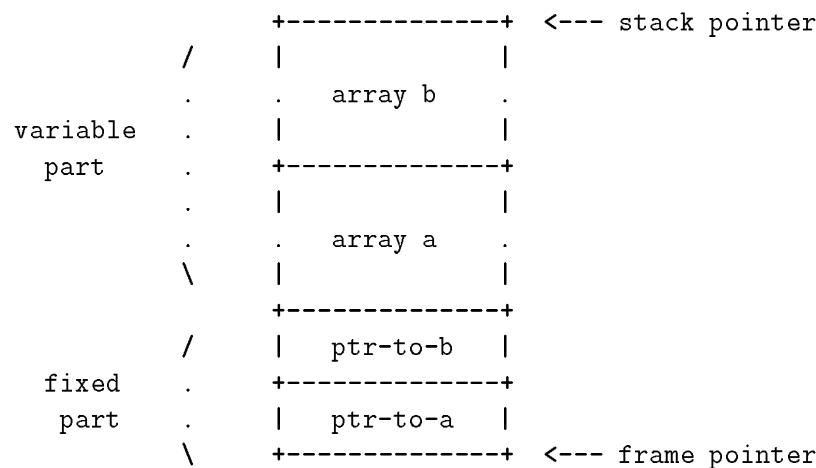
Suppose we want to add dynamic arrays to C, so we now allow declarations of the form:
int x (n);
where n is an expression. Such declarations are to be allowed only within functions, e.g.

```
int func1 (int param) {

    int twice [param * 2];
    int big   [param * param];

    ...  big [x] + twice [y] ...
};
```

- One possible implementation is to allocate such a variable on the stack frame of the function. Using a picture, show exactly how this would be done. Your answer should handle the general case of more than one such variable on a given stack.
- If the approach in 1 is followed, it may be necessary to allocate a frame pointer where none was previously needed. Explain.
- For the first such variable in a frame (or the only one if there is only one), an optimization is possible if the stack builds up in memory (but not if it builds down in memory). Explain using appropriate diagrams.
- Another approach is to allocate the array on the heap. What are the advantages and disadvantages of this approach.

Solution

- The stack frame must be divided into two parts, the fixed length part and the variable length part. The variable length objects are allocated in the variable length part (using the equivalent of alloca), and a pointer is allocated for each such array. The pointer is placed in the fixed part of the frame and initialized to point to the start of the array in the variable length part.
  An access to the array picks up the pointer and uses it to access the array.
  For example, suppose the stack builds up in memory (high memory at top of page), and there are two such arrays a and b, then the frame would look like:



- If the frame is fixed length, then, referring to the picture above, the fields in the frame can be addressed using (negative) offsets from the stack pointer. But if the frame has components whose length is not known

till execution time, then we have to have a separate frame pointer as shown in the diagram, and fields in the fixed part of the frame can be addressed using (positive) offsets from the stack pointer.

- Referring to the above diagram, the start of array a is actually at a fixed offset from the frame pointer, so it is no necessary to allocate a pointer to it, instead we can address it using this fixed offset. This is only possible if the start of the array is nearest to the frame pointer, i.e. if the stack builds up in memory. This is one of the only cases in which the choice of stacks building up and down in memory is not equivalent.

- In this approach we still alocate a pointer in the frame, but the referenced array is on the heap.

  The advantage is that the stack frame is still fixed length. This means that the frame pointer can be omitted, freeing up a register, and there are also other optimizations possible if all frame sizes are fixed, notably the possibility of a nested function "reaching up" to access the frame of its enclosing function directly without bothering to use the static chain if the language allows nested procedures (like GNU C).

  The disadvantage is that allocation on the heap is generally more expensive. Also to avoid memory leaks, the storage allocated for this purpose must be freed on exit from the function. This takes much more time than just popping the frame off the stack. Ensuring tha the memory is released is even more tricky if the language supports exceptions (like C++ or Ada) since then we have to ensure that the storage is released if an exception propagates through this frame.

Question 1

Consider a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty page is available or the replaced page is not modified, and 20 milliseconds if the replaced page is modified. Memory access time is 100 nanoseconds.
In the application program you are evaluating, the page to be replaced is modified 70% of the time. What is the maximum acceptable page-fault rate for an average access time of at most 200 nanoseconds?

Solution

Let the maximum fault-rate be F. Then:

$$200 nanosecs \geq (1-F) \times 100 nanosecs + 0.3_F \times 8 msecs + 0.7_F \times 20 msecs$$

Solving for F we get F=1/164000, or approximately 0.000006.

Question 2

a) Describe briefly the two-phase locking protocol and the timestamp protocol for concurrency control.
Consider the following two histories (time flows downward in the diagrams):

b) two transactions T1, T2 perform the following:

```
 T1          T2
read A
            write B
read B
```

Show that this history is feasible under the two-phase locking protocol, but not under the timestamp protocol.

c) three transactions T1, T2, T3 perform the following:

```
 T1          T2          T3
write A
            write A
                        write A
write B
            write B
```

Show that this history is feasible under the timestamp protocol, but not under the two-phase locking protocol.

Solution
a) From any textbook.

b) Under the two-phase locking protocol, it is possible for T2 to acquire a lock on B just before writing, and releasing it just after, and for T1 to acquire that lock on B before releasing the lock on A. Under the timestamp protocol this cannot happen, because the timestamp of T1 is lower than that of T2, but T2 attempts to access B (in an incompatible mode) before T1.

c) Under the timestamp protocol the interleaving is feasible because all accesses on any given variable are in timestamp order. This is not feasible under two-phase locking, as can be seen from the following argument: before T3 could lock A, T1 and T2 had to both lock and unlock A. but after that both T1 and T2 would have to lock B, so at least one of them had to acquire that lock after relinquishing A, which violates the two-phase condition.

CORE EXAMINATION
Department of Computer Science
New York University
January 23, 1998

This is the common examination for the M.S. program in CS. It covers core computer science topics: Languages and Compilers, Operating Systems, and Algorithms. The exam has two parts. The first part lasts three hours and covers the first two topics. The second part, given this afternoon, lasts one and one-half hours, and covers algorithms.

Attempt all of the questions. You are not required to take the algorithms section of the exam if you have passed the FOCS exam in the past.

Use the proper booklet for each question. Each booklet is marked with the Area and Question number, in the form PL&C1, PL&C2, PLC&C3, OS1, OS2, ALGS1, ALGS2, ALGS3. Use the appropriate booklet for each question. DO NOT put your name on the exam booklet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope. You can keep the exam.

Good luck!

Basic Algorithms

Question 1

Let G be a directed graph, with N vertices and E edges.

Show how each of the following problems can be solved in time O(N+E). (Hint: Depth-first search is a useful subroutine for each of these. You need not give the code for DFS; just explain, in English or pseudo-code, how you use it.)

- Given a vertex A, list all the vertices u such that there is a path through G from A to u.
- Given a vertex A, list all the vertices u such that there is a path through G from u to A.
- Given two vertices A and B, list all the vertices u such that there is a path through G from A to u, but no path through G from B to u.
- Determine whether the graph G is in fact a downward-pointing tree.
- Suppose you know the number of vertices N a priori, but do not know E. Can you solve problem 4 in time O(N), independent of E?

Solution

- Do a DFS starting at A and enumerate all the vertices encountered in the DFS.
- Construct the reverse of G; that is, the graph H that contains an edge v-¿u for each edge u-¿v in G. Do a DFS from A through H and return the list of all the vertices encountered.
- Do a DFS starting at B and mark each vertex encountered. Then do a DFS starting at A, but omit any vertices marked in the first DFS.
- Do repeated DFS from unvisited nodes, marking visited bodes (but not the root of each DSF) and stopping at a node already visited. (This is O(E). If more than one root has not been visited, return false. Otherwise to a DFS from the unique root and verify that each node is visited once.
- If we know a priori that the graph has n nodes, then:
  - Start counting the edges in G. If the count gets past n-1, then halt and return false. If the final count is less than n-1, then return false.

- For each edge u-¿v in G, mark v. Collect the unmarked vertices in G. These are the vertices with no in-arc. Unless there is exactly one such vertex, return false; else, let R (the root) be the one vertex with no in-arc.
- Do a DFS starting at R. If the DFS reaches every vertex in G then return true, else return false.

Question 2

Consider a heap with the minimum element on top.

- Characterize the structure of a heap, and give the running times of the 3 heap operations using the usual array based implementation: FINDMIN, DELETEMIN, and INSERT.

- Instead of using an array, suppose we implement the heap as a tree with both parent and child pointers. What happens to the performance of the 3 operations? Describe how they might be implemented, and whether or not they can be performed as efficiently.

Solution

- A heap is a complete binary tree with the ordering property that the key of the parent of node X is smaller than or equal to the key of node X itself. In the usual array based implementation, if a node is in position i, its children are in position 2i and 2i+1. With this implementation, a heap with N nodes supports the following operations in the specified run time:
  - FINDMIN - O(1). Root node is the minimum element, in position 1 of the array.
  - DELETEMIN - O(log N). Delete the root node, put the last element in the array in its place, and percolate it down until it is in the right spot. Since the heap is balanced, an N element heap has height O(log N).
  - INSERT - Add the new element to the first empty spot, and percolate it up the tree to the right location. Again, this takes O(log N).

- If the heap is represented by a tree, then:
  - FINDMIN - still takes O(1), assuming of course we keep a pointer to the root of the tree.
  - DELETEMIN - The difficulty here is in finding the "last" element in the tree. This can be done for example using a breadth first traversal (taking O(N)), or by keeping a pointer to the last node, in which case updating the pointer is difficult. (Note that sibling or level pointers were not part of this implementation.) Once the last node is located and moved to the root, heapifying it proceeds as usual in O(log N). Algorithms that delete the root, move up the smallest child node, and recursively continue, result in an unbalanced tree.
  - INSERT - The difficulty here is finding the first "empty" spot , or next leaf node. Again, once it is found, heapifying the new node takes O(log N).

Question 3

Below is a (really bad) way to solve the Maximum Subsequence problem: Given an array A[1..N], find the indices i and j so that the sum $\sum_{k=i}^{j} A[k]$ is a maximum.

```
   Procedure MaxSubSeq(i,j)
begin
    if (i = j) then
        MaxSubSeq = A[i]
    else begin
           ThisSum   =  ∑_{k=i}^{j} A[k]
           MaxSubSeq = MAX(ThisSum, MaxSubSeq(i+1,j), MaxSubSeq(i,j-1))
         end
end
```

It is initially called with

```
MaxSubSeq(1,N)
```

- Let $T(N)$ be the running time for this problem for an array of length N. Write down the recurrence relation and initial condition that describe $T(N)$.

- Solve the recurrence relation or give an estimate of the behavior of T(N) for large values of N.

- Give a sketch (only 2 sentences please) of a more efficient way to solve the problem, and state its complexity.

Solution

(a) The initial condition is $T(1) = \Theta(1)$, because when $N = 1$ (which corresponds to $i = j$) the procedure terminates. The recurrence equation, for $N > 1$, is

$$T(N) = 2T(N-1) + \Theta(N). \tag{1}$$

The term $2T(N-1)$ comes from the two recursive calls; the term $\Theta(N)$ is the time to compute the sum `ThisSum`.

(b) Define $S(N) = T(N)/2^N$. Dividing Equation (1) by 2^N gives (for $N > 1$)

$$S(N) = S(N-1) + \Theta(\frac{N}{2^N}).$$

By iterated substitution, we get

$$S(N) = \sum_{k=1}^{N} \Theta(\frac{k}{2^k}).$$

Because the sum $\sum_{k=1}^{\infty} \frac{k}{2^k}$ converges (to 2), we have $S(N) = \Theta(1)$. Hence $T(N) = \Theta(2^N)$.

(c) For each index pair i, j with $1 \leq i \leq j \leq N$, compute the sum $\sum_{k=i}^{j} A[k]$; return the sum with the largest value. There are $\Theta(n^2)$ index pairs, and computing one sum takes $O(n)$ time, so the total time is $O(n^3)$.

Though not required for this question, more efficient algorithms are known that solve the problem in $O(n)$ time.