SOLUTIONS TO CORE EXAMINATION
Department of Computer Science
New York University
February 3, 2006

Algorithms

Algorithms: Question 1

Hercules is fighting a herd of hydras. With each sword stroke, he removes one head from a hydra. When a k-headed hydra has a head cut off, it transforms itself into $k-1$ hydras, with $1, 2, \ldots, k-1$ heads respectively. If Hercules starts off fighting a single n-headed hydra, how many sword strokes are needed until no hydras remain?

Answer: Let $\mathrm{S}(n)$ be the number of sword strokes needed to kill an n-headed hydrea. We have then the recurrence

$\mathrm{S}(n) = 1 + \mathrm{S}(1) + \mathrm{S}(2) + \ldots \mathrm{S}(n-1)$

Thus,

$\mathrm{S}(n+1) = 1 + \mathrm{S}(1) + \mathrm{S}(2) + \ldots \mathrm{S}(n-1) + \mathrm{S}(n)$

Subtracting the first equation from the second gives

$\mathrm{S}(n+1) - \mathrm{S}(n) = \mathrm{S}(n)$; so

$\mathrm{S}(n+1) = 2\mathrm{S}(n)$.

Since $\mathrm{S}(1)=1$, we have the solution $\mathrm{S}(n)=2^{n-1}$.

Algorithms: Question 2

Let C be a collection of records in a (medical) database, in which each record $r=\langle r.\mathrm{wt}, r.\mathrm{ht}\rangle$ stores a person's height and weight. We say that person r is *at least as large* as person s if r is both at least as tall and at least as heavy than s; that is, $r.\mathrm{ht} \geq s.\mathrm{ht}$ and $r.\mathrm{wt} \geq s.\mathrm{wt}$. We also say that r and s are *related* if either r is at least as large as s or s is at least as large as r; and that they are *unrelated* if they are not related.

A. (1 point) If r and s are unrelated, and $r.\mathrm{ht} < s.\mathrm{ht}$, then what is the relation between $r.\mathrm{wt}$ and $s.\mathrm{wt}$?

Answer: $r.\mathrm{wt} > s.\mathrm{wt}$.

B. (3 points) Give an $\mathrm{O}(n \log n)$ time algorithm to test if the records in collection C are all unrelated.

Answer: First, sort the records by height. Second, for each consecutive pair of records r and s, check $r.\mathrm{ht} < s.\mathrm{ht}$ (rather than equal) and that $r.\mathrm{wt} > s.\mathrm{wt}$

C. (6 points) Give a data structure for storing a collection C of n unrelated records, which supports the following operations in time $O(\log n)$:

i. Test whether a new record q is unrelated with respect to all the records in C.

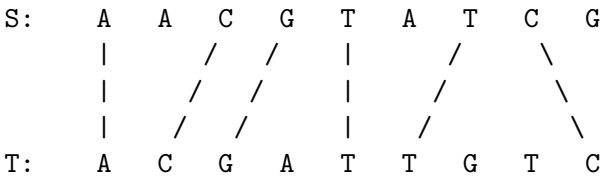
ii. If q is unrelated with respect to all the records in C, then add q to C.

Answer: Use a balanced tree (e.g. 2-3 tree; red-black tree; AVL tree) ordered by height. Given q find the records r and s that would precede and follow q in order by height. Check that r.ht $< q$.ht $< s$.ht and that r.wt $> q$.ht $> s$.ht. If the condition is satisfied, add q between r and s.

Algorithms: Question 3

Let S and T be two strings of symbols, each of length n. For i between 1 and n, we say that $S[i]$ *may correspond* to $T[j]$ if $S[i]$ and $T[j]$ are the same character and either $j = i - 1$, $j = i$, or $j = i + 1$. An *alignment* of S and T is a set of corresponding pairs with no overlap (i.e. each character in S is matched with at most one character of T and vice versa) and no crossing (i.e. the alignment does not both have $S[I]$ matched with $T[I + 1]$ and also $S[I + 1]$ matched with $T[I]$.)

For example, the diagram below shows an alignment of two strings of length 9. The alignment has size 6.



Given an efficient algorithm that finds the size of the maximum alignment between two input strings. Analyze the running time of your algorithm. Linear time is possible.

Hint: Use dynamic programming. You do not need any complex or sophisticated processing of the strings.

Answer: Consider any alignment of the first K characters in S with the first K characters in T. There are two possibilities:

1. S[K]=T[K]. In this case, the overall alignment combines an arc from S[K] to T[K] with an alignment of the first K−1 characters of S and T.

2. S[K]≠T[K]. In this case, the overall alignment is in fact either an alignment of the first K characters of S with the first K−1 of T or vice versa.

For an alignment of the first K characters in S with the first K-1 characters in T there are two possibilities.

1. S[K]=T[K-1]. In this case, the overall alignment combines an arc from S[K] to T[K-1] with an alignment of the first K-1 characters of S and the first K-2 of T.

2. S[K] $\neq$ T[K-1]. In that case, the overall alignment is in fact an alignment of the first K-1 characters of S and the first K-1 of T.

The case of an alignment of the first K-1 characters of S with the first K characters of T is the reverse.

Therefore: Let W[K] be the size of the best alignment of the first K characters of S and T. Let U[K] be the size of the best alignment of the first K characters of S with the first K-1 of T. Let V[K] be the size of the best alignment of the first K-1 characters of S with the first K of T. Then the following dynamic programming triply recursive routines computer W, U, and V in linear time:

```
function W(K)
  { if (K==0)  then return(0)
      else if (S[K] == T[K]) then return(W(K-1)+1)
      else return(max(U(K),V(K))
  }

function U(K)
  { if (K==1) then return(0)
      else if (S[K] == T[K-1]) then return(U(K-1)+1)
      else return(W(K-1))
  }

function V(K)
  { if (K==1) then return(0)
      else if (S[K-1] == T[K]) then return(V(K-1)+1)
      else return(W(K-1))
  }
```

PL&C: Question 1

A) (4 points) In Scheme, write a function "flatten" that, given an arbitrary nested list, produces the list of all its atoms. For example, given the list

```
(cond ((eq n 0) 1) (else (* n (fact (- n 1)))))
```

then the output of flatten on this list will be:

```
(cond eq n 0 1 else * n fact - n 1)
```

Answer:

```
(define (flatten L)
   (cond ((null L) L)
         ((atom L) (list L))
         (else (append (flatten (car L)) (flatten (cdr L))))))
```

B) (2 points) We want to determine whether two trees (represented as lists) have the same fringe, that is to say the same set of leaves, in the same order. Use the function flatten above to write the definition of a function Same_Fringe. For example, the function will return True when given the two lists:

```
((this is)  (an example))     and   (this (is (an (example))))
```

Answer:

```
(define (SameFringe L M) (equal (flatten L) (flatten M)))
```

C) (4 points) The function you wrote in (B)) is inefficient in some cases: if the trees differ on their first leaves, it is still necessary to compute their fringes in full before comparing them. How would the notion of iterator allow you to obtain a more efficient version of Same_Fringe?

An iterator is a programming abstraction that facilitates the traversal of a collection. In particular, an iterator can produce elements of the collection one by one, as needed. If we can write iterators for the two lists l1 and l2 mentioned in the previous question, we can write the SameFringe algorithm as follows (using some unspecified imperative language):

```
It1 := iterator (L1);
It2 := Iterator (L2);
while Has_More (It1) and then Has_More (It2) loop
   Get_Next (It1); Get_Next (It2);
   if Current (It1) /= Current (It2) then
```

```
      return False;
   end if;
end loop
...
```

And in this fashion the algorithm terminates as soon as two elements differ

PL&C: Question 2

Consider the following C++ code:

```
#include <iostream>

using namespace std;

class A {
protected:
  virtual void print() { cout << "A" << endl; }
  void print2() { cout << "AA" << endl; }
};

class B {
public:
  virtual void print() { cout << "B" << endl; }
  void print2() { cout << "BB" << endl; }
};

class C :public A, public B {
public:
  void print2() { cout << "CC" << endl; }
};
```

A) (3 points) Given the above code, what will happen when the following code is compiled and run?

```
int main()
{
  C c;
  c.print();
}
```

Answer: Compile error: "request for member 'print' is ambiguous"

B) (3 points) Alternatively, what will happen when the following code is compiled and run?

```
int main()
{
  C c;
  B* p = &c;
  p->print2();
}
```

Answer: BB.

C) (4 points) Rewrite class C (without changing classes A or B) so that the following code has the output shown. Do not use any iostream functions in the code you write.

```
int main()
{
  C c;
  c.a();
  c.b();
  c.aa();
  c.bb();
}

Output:
A
B
AA
BB
```

Answer: One solution:

```
class C :public A {
  B _b;
public:
  void a() { print(); }
  void b() { _b.print(); }
  void aa() { print2(); }
  void bb() { _b.print2(); }
};
```

PL&C: Question 3

Consider the alphabet $\Sigma_1 = \{0, 1, \vee, \wedge\}$, where we refer to the letters 0 and 1 as *operands* and the symbols $\vee$ and $\wedge$ as *operators*. A well-formed boolean expression is a string whose first and last symbols are operands and, within the string, operands and operators alternate.

A) (3 points) Write a regular expression that captures the language of all well-formed boolean expressions.

Answer: $D; \{op; D\}^*$, where D is an abbreviation for $\{0 + 1\}$ and op is an abbreviation for $\{\vee + \wedge\}$.

B) (3 points) Write a regular expression that characterizes the language of all boolean expressions which evaluate to 1. Note that $\wedge$ has higher priority than $\vee$. Thus, $0 \vee 1 \wedge 0$ should be interpreted as $0 \vee (1 \wedge 0)$ rather than $(0 \vee 1) \wedge 0$. In your regular expression you may use generalizations such as ϵ standing for the empty word, or brackets for an optional sub-expression.

Answer: $[\{D; op\}^*; D; \vee]\ 1\{\wedge 1\}^*\ [\vee; D; \{op; D\}^*]$.

C) (4 points) Extend the alphabet to $\Sigma_2 : \{0, 1, \vee, \wedge, (,)\}$. That is, allow parentheses within the boolean expressions. Write a context-free grammar that generates all boolean expressions over Σ_2 evaluating to 0.

Answer: Let DZ correspond to disjunctions with value 0; DU correspond to disjunction with value 1; CZ correspond to conjunctions with value 0; CU correspond to conjunctions with value 1; and EZ/EU represent either constant or parenthesized expressions with value 0/1 respectively.

$$
\begin{aligned}
&EZ \rightarrow 0 \mid (DZ) \\
&EU \rightarrow 1 \mid (DU) \\
&CZ \rightarrow EZ \mid EZ \wedge CZ \mid EZ \wedge CU \mid EU \wedge CZ \\
&CU \rightarrow EU \mid EU \wedge CU \\
&DZ \rightarrow CZ \mid CZ \vee DZ \\
&DU \rightarrow CU \mid CU \vee DU \mid CU \vee DZ \mid CZ \wedge DU
\end{aligned}
$$

Any expression evaluating to zero can be generated from start symbol DZ.

OS: Question 1

A. (4 points) Explain the difference between a *preemptive* and a *non-preemptive* process scheduling algorithm. Name one example of each. (You need only name the two algorithms; you do not have to describe them.)

Answer: In a preemptive scheduling algorithm, the running process may be suspended while in the middle of running, and put back on the ready queue. In a non-preemptive algorithm, the running process always continues until it either blocks or terminates. Round-robin is preemptive; FIFO is non-preemptive.

B. (3 points) In process scheduling, what is *starvation?* (Note: This is *not* the same as deadlock.) Give an example of a process scheduling algorithm that can suffer from starvation, even if no process ever goes into an infinite loop. Also give an example of a process scheduling algorithm that is guaranteed not to suffer from starvation. (Again, it is sufficient to name two algorithms.)

Answer: Starvation occurs when a process can wait in the ready state indefinitely, being never chosen when the scheduling algorithm picks a new process to run. Priority scheduling can suffer from starvation; a low priority process will never run as long as there are always higher priority processes ready that can be chosen instead. LIFO scheduling can suffer from starvation; a process may never be chosen to run as long as there are other processes that arrived in the system more recently. Round robin and FIFO do not suffer from starvation.

C. (3 points) In disk arm scheduling, what is starvation? Give an example of a disk arm scheduling algorithm that can suffer from starvation and one that cannot.

Answer: Starvation occurs when a request for a block is never met because the scheduling algorithm always chooses some other block to service first. Shortest seek first suffers from starvation; if the disk receives a continuous stream of requests for blocks near the disk arm, a request for a block far from the arm may never be met. The elevator algorithm does not suffer from the problem.

OS: Question 2

A. (2 points) Describe the *least recently used* (LRU) page replacement algorithm.

Answer: The algorithm chooses that page in memory whose most recent use was the furthest in the past.

B. (2 points) "Most processes ... exhibit a **locality of reference,** meaning that during any phase of execution, the process references only a relatively small fraction of its pages." (A. Tanenbaum, *Modern Operating Systems,* p. 222.) Explain how this principle justifies the use of the LRU algorithm.

Answer: If a process exhibits locality, then at each stage it will be working with a comparatively small number of pages in the working set. The objective of the paging algorithm is therefore to keep the pages in the working set in memory and to replace the pages no longer in the working set. The LRU strategy accomplishes this: If a page has not been used in a while, then it is presumably not in the working set, and therefore can be safely replaced.

C. (1 point) The exact LRU algorithm is not used in actual operating systems. Why not?

Answer: Recording an exact timestamp for the page(s) accessed at each machine cycle would demand too great an overhead.

D. (5 points) Most compilers allocate multi-dimensional arrays in row-major order. For example, in C, if the array A is declared as

```
int A[1024][1024]
```

then A[10][2] is allocated in the location immediately following A[10][1], and A[11][0] is allocated in the location immediately following A[10][1023].

Consider the following code fragment:

```
int A[1024][1024];

for (int j=0; j<1024; j++)
   for (int i=0; i < 1024; i++)
     A[i][j] =2;
```

Suppose that the computer architecture uses 32 bit (4 Byte) integers and 4 KByte pages; thus, the array A requires 1024 pages. Suppose that the OS can only allocate 1000 page frames to the array A in the process running this code. Assume that initially none of the pages for A are in memory. In the execution of the above code, how many page faults occur if the OS uses an LRU page replacement algorithm? How many page faults occur if the OS uses a LIFO (last in, first out) page replacement algorithm? (You may give an approximate answer — e.g. "About 5000" — to both of these.)

Answer: The process references the pages in cyclic order: 0, 1, 2, ... 1023, 0, 1, 2, ... 1023 etc. This fails the locality principle in the most extreme form; the next page reference is always to the

page least recently used. In the LRU strategy, therefore, *every* page reference is to a page that was replaced 24 cycles ago, and therefore involves a page fault, for a total of $1024 \cdot 1024 = 1,048,476$ page faults. In the LIFO strategy, pages $0 \ldots 998$ are loaded into memory initially, and remain there. The page in the final frame is therefore always the "Last In" chosen for replacement, and thus cycles through pages 99-1023 in each iteration of the outer loop. Thus, there are 1024 page faults in the first iteration of the outer loop, and 25 page faults in each of the subsequent 1023 iterations, for a total of 26599 page faults.