Dalhousie University Faculty of Computer Science
Design and Analysis of Algorithms
Assignment 1 CSCI 3110 Due: 24 Sept 2012

Problems 1 - 4 below are from the text (*Algorithms by Dasgupta, Papadimitriou, Vazirani* pages 8 - 10).

(1) (1 pt. each) Ex. 0.1 (c) - (o). For each part, briefly show the algebra justifying your answer.
(2) (1 pt. each) Ex. 0.2
(3) (3 pts each) Ex. 0.3
(4) (3 pts each) Ex. 0.4 (a) - (d). You may do part (e) for extra credit.
(5) Write a recursive procedure to perform Linear Search for a key in an array A. Prove your proc correct and estimate its running time.
(6) Prof. Ivan A. Ripof has invented a new search algorithm that he calls Trinary Search. He argues that it executes correctly and searches more efficiently than Binary Search.
Pre: A a sorted array
Post: An array index of key in A or -1 if key not found

```
TrinSearch(A, fst, lst, key)
 1  if (lst < fst)
 2        index = −1
 3  else
 4        thrd = ⌊(lst+fst)/3⌋
 5        twrd = 2∗⌊(lst+fst)/3⌋
 6        if (key == A[thrd])
 7              index = thrd
 8        else
 9              if (key < A[thrd])
10                    index = TrinSearch(A, fst, thrd − 1, key)
11              else
12                    if (key == A[twrd])
13                          index = twrd
14                    else
15                          if (key < A[twrd])
16                                index = TrinSearch(A, thrd + 1, twrd − 1, key)
17                          else
18                                index = TrinSearch(A, twrd + 1, lst, key)
19  return index
```

(a) Prove that $TrinSearch(A, fst, lst, key)$ works correctly.
(b) Support or dispute Prof. Ripof's claim about the efficiency of the code. Give clear reasons for your answer (no argument no credit).

CSCI 3110 Assignment 1 Solutions

October 10, 2012

1. In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$, or both (in which case f = $\Theta(g)$).

	$f(n)$	$g(n)$	Relation	Reason
(c)	$100n + \log n$	$n + (\log n)^2$	$f = \Theta(g)$	Both are $\Theta(n)$
(d)	$n \log n$	$10n \log 10n$	$f = \Theta(g)$	Both are $\Theta(n \log n)$
(e)	$\log 2n$	$\log 3n$	$f = \Theta(g)$	Both are $\Theta(\log n)$
(f)	$10 \log n$	$\log(n^2)$	$f = \Theta(g)$	Both are $\Theta(\log n)$
(g)	$n^{1.01}$	$n \log^2 n$	$f = \Omega(g)$	$n^{0.01} = \Omega(\log^2 n)$
(h)	$n^2 / \log n$	$n(\log n)^2$	$f = \Omega(g)$	$n = \Omega((\log n)^3)$
(i)	$n^{0.1}$	$(\log n)^{10}$	$f = \Omega(g)$	Any polynomial = $\Omega()$ any log
(j)	$(\log n)^{\log n}$	$n / \log n$	$f = \Omega(g)$	$2^{\log_2 n} = n$ so $(\log n)^{\log n + 1} = \Omega(n)$
(k)	$\sqrt{n}$	$(\log n)^3$	$f = \Omega(g)$	Any polynomial = $\Omega()$ any log
(l)	$n^{1/2}$	$5^{\log_2 n}$	$f = O(g)$	$5^{\log_2 n} = n^{\log_2 5}$ and $1/2 < \log_2 5$
(m)	$n2^n$	3^n	$f = O(g)$	$n = O((3/2)^n)$
(n)	2^n	2^{n+1}	$f = \Theta(g)$	$n = \Theta(n + 1)$
(o)	$n!$	2^n	$f = \Omega(g)$	$n! > 2^n$ for all $n \geq 4$

2. Show that, if c is a positive real number, then $g(n) = 1 + c + c^2 + \ldots + c^n$ is:

   (a) $\Theta(1)$ if $c < 1$.

   We take the limit

   $$\lim_{n\to\infty} \frac{g(n)}{1} = \lim_{n\to\infty} \frac{\sum_{i=0}^{n} c^i}{1} = \lim_{n\to\infty} \sum_{i=0}^{n} c^i = \frac{1}{1-c}$$

   for $c < 1$. This limit is constant, so $g = \Theta(1)$.

   (b) $\Theta(n)$ if $c = 1$.

   We expand

   $$g(n) = 1 + c + c^2 + \ldots + c^n = \sum_{i=0}^{n} c^i = n + 1 = \Theta(n).$$

   (c) $\Theta(c^n)$ if $c > 1$.

   Each term c^i is dominated by c^n for $i < n$, so $g(n) = \Theta(c^n)$.

   The moral: in big-Θ terms, the sum of a geometric series is simply the first term if the series is strictly decreasing, the last term if the series is strictly increasing, or the number of terms if the series is unchanging.

3. The Fibonacci numbers $F_0, F_1, F_2, \ldots$, are defined by the rule

   $$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}.$$

   In this problem we will confirm that this sequence grows exponentially fast and obtain some bounds on its growth.

(a) Use induction to prove that $F_n \geq 2^{0.5n}$ for $n \geq 6$.

We first establish the base cases $n = 6$ and $n = 7$.

$n = 6$ $F_6 = 8 \geq 2^{0.5 \cdot 3} = 8$.

$n = 7$ $F_7 = 13 \geq 2^{0.5 \cdot 3} \approx 11.3$.

Now, assume that the claim holds for all $n < k + 2$. Then

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &\geq 2^{0.5k} + 2^{0.5k+0.5} = \frac{(2^{0.5}+1)}{2} 2^{0.5k+1} = \frac{(2^{0.5}+1)}{2} 2^{0.5(k+2)} \\ &\geq 2^{0.5(k+2)}. \end{aligned}$$

Therefore, by induction, $F_n \geq 2^{0.5n}$ for $n \geq 6$.

(b) Find a constant $c < 1$ such that $F_n \leq 2^{cn}$ for all $n \geq 0$. Show that your answer is correct.

We use induction to prove that $c = 0.9$ works. We first establish the base cases $n = 0$ and $n = 1$.

$n = 0$ $F_0 = 0 \leq 2^{0.9 \cdot 0} = 1$.

$n = 1$ $F_1 = 1 \leq 2^{0.9 \cdot 1} \approx 1.86$.

Now, assume that the claim holds for all $n < k + 2$. Then

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &\leq 2^{0.9k} + 2^{0.9k+0.9} = 2^{0.9k}(1 + 2^{0.9}) = \frac{(1+2^{0.9})}{2^{1.8}} 2^{0.9(k+2)} \approx 0.82(2^{0.9(k+2)}) \\ &\leq 2^{0.9(k+2)}. \end{aligned}$$

Therefore, by induction, $F_n \leq 2^{0.9n}$ for $n \leq 6$.

(c) What is the largest c you can find for which $F_n = \Omega(2^{cn})$?

Let $b = 2^c$. Then we consider the relation $F_{n+2} = F_n + F_{n+1}$ and get $b^{n+2} = b^n + b^{n+1}$. This can be rearranged as $b^{n+2} - b^n - b^{n+1} = 0$. Solving for the roots of this equation gives $b = \frac{1 \pm 5^{0.5}}{2}$. Since b cannot be negative, $c = \log_2(b) = \log_2(\frac{1+5^{0.5}}{2}) \approx 0.69$.

4. (a) Show that two 2 x 2 matrices can be multiplied using 4 additions and 8 multiplications.

The formula to multiply two 2 x 2 matrices is:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix},$$

Which requires 4 additions and 8 multiplications.

(b) Show that $O(\log n)$ matrix multiplications suffice for computing X^n.

X^n can be computed with the following recurrence relation. This relation requires $O(\log n)$ matrix multiplications to compute X^n by making one recursive call to determine $X^{\lfloor n/2 \rfloor}$ and then applying a constant number of matrix multiplications.

$$\begin{aligned} X^n &= I \text{ if } n = 0 \\ &= X \text{ if } n = 1 \\ &= X^{n/2} \cdot X^{n/2} \text{ if } n \text{ is even} \\ &= X \cdot X^{\lfloor n/2 \rfloor} \cdot X^{\lfloor n/2 \rfloor} \text{ if } n \text{ is odd} \end{aligned}$$

(c) Show that all intermediate results of fib3 are $O(n)$ bits long.

We can use induction to prove the claim. Let

$$F = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Our base cases compute F^0 and F^1, which are $O(1)$ bits long. Now, assume that the claim holds for $1 \leq n < k$. If k is odd, then $F^k = F \cdot F^{\lfloor k/2 \rfloor} \cdot F^{\lfloor k/2 \rfloor}$. By the inductive hypothesis, computing this requires multiplying $O(1)$ bit numbers with $O(k/2)$ bit numbers and then with $O(k/2)$ bit numbers, resulting in $O(k)$ bit numbers. If k is even, then $F^k = F^{\lfloor k/2 \rfloor} \cdot F^{\lfloor k/2 \rfloor}$, which similarly has $O(k)$ bit numbers. Therefore, the claim holds by induction.

(d) Let $M(n)$ be the running time of an algorithm for multiplying n-bit numbers, and assume that $M(n) = O(n^2)$. Prove that the running time of fib3 is $O(M(n) \log n)$.

fib3 requires $O(\log n)$ multiplications, which each run in $O(M(n))$ time. Each of the $O(\log n)$ steps of the algorithm does $O(1)$ other work, so the algorithm requires $O(M(n) \log n)$ time.

(e) Can you prove that the running time of fib3 is $O(M(n))$?

We can prove this using induction. Our base cases are once again F^0 and F^1, which clearly require $O(M(n))$ time. Now, assume that the claim holds for $1 \leq n < k$. If k is odd, then $F^k = F \cdot F^{\lfloor k/2 \rfloor} \cdot F^{\lfloor k/2 \rfloor}$. By the inductive hypothesis, determining $F^{\lfloor k/2 \rfloor}$ requires $O(M(k/2) = O(M(k))$ time. Multiplying these arrays requires $O(M(k))$ time as well, so F^k can be found in $O(M(k))$ time. Similarly, F^k can be computed in $O(M(k)$ time if k is even. Therefore, the claim holds by induction.

5. Linear Search
PRE: An array
POST: An array index of key in A or -1 if not found
LINEARSEARCH(A, fst, lst, key) [1] $lst < fst$ $index = -1$ $key == A[fst]$ $index = fst$
LINEARSEARCH$(A, fst + 1, lst, key)$

Proof of Correctness
Inductive hypothesis: The proc `LinearSearch` correctly searches for the key for all arrays of size $lst - fst + 1 < n,\ \forall n > 0$.
Base Case: Let $n = 1$(array size) Then, if key matches the only element, algorithm will correctly return the only index $fst = 1$. If the key does not match, a recursive call is made with $fst = 2$, $lst = 1$, returning a -1, performing the search correctly *(This step could also be done taking* $n = 0 = lst - fst + 1 \rightarrow lst < fst$ *and the proc.* correctly *returns* -1 *on the empty array.)*
Inductive Step: For $n = fst - lst + 1$ The proc. checks if the first element matches the key. If it matches, the index of the first element is returned. If not, it makes a recursive call on an array of size $n - 1 < n$, which from the inductive assumption, works correctly. Hence the proc. works correctly.

Running time: Base Case check: $O(1)$; Check of 1st element with key $O(1)$; Recursive call: $T(n-1)$ Hence we have $T(n) = T(n-1) + c$ $(\exists c > 0)$. Unrolling this is simple: $T(n) = T(n-2) + 2c$ *etc etc* Proc terminates when $T(n) = T(0) + n \cdot c$, giving $T(n) \in O(n)$.

6. Trinary Search

(a) **Base Case:** $n = 0 = lst - fst + 1$, i.e, $lst < fst$ and the proc. returns -1.
Ind. Hyp.: TRINSEARCH works correctly for all i/p of size $n = k = lst - fst + 1$, $\forall k \geq 0$
Assume when $n = k + 1$, since $n > 0$, proc. jumps to line 4 and calculates *thrd* and *twrd*. If $key == A[thrd]$, then proc. return **index** and $fst \leq thrd \leq lst$, the proc. terminates correctly. **Else** if $key < A[thrd]$, since $thrd - 1 - fst + 1 = thrd - fst < k + 1$, TRINSEARCH can return correct index based on **Ind. Hyp.**. **Else** if $key == A[twrd]$, then proc. return **index** and $fst \leq twrd \leq lst$, the proc. terminates correctly. **Else** since $twrd - 1 - thrd - 1 + 1 = twrd - thrd - 1 < k + 1$ and $lst - twrd - 1 + 1 = lst - twrd < k + 1$, TRINSEARCH can return correct index based on **Ind. Hyp.**.

(b) Since proc. splits the arrays into 3 slices in each cycle, $O(\log_3(n))$ recursions are needed. Although $\log_3(n) < \log_2(n)$, it is in the equivalent class $O(\log(n))$ as $O(\log_2(n))$. TRINSEARCH can be thought of more efficient in terms of actual running time but the same efficient as BINARY SEARCH in terms of Big-O notation.

Dalhousie University Faculty of Computer Science
Design and Analysis of Algorithms I
Assignment 2 CSCI 3110 Due: 3 Oct 2012

(1) (5 pts) Ex. 1.4, 1.31

(2) (4 pts.) Ex. 1.8

(3) (5 pts) Ex. 1.14, 1.19. (*For 1.14 assume that multiplying n-bit numbers costs $M(n)$ like in problem 0.4.*)

(4) (4 pts) Ex 1.22, Ex. 1.24

(5) (4 pts) Ex 1.27

(6) (a) (8 pts) Write an algorithm to compute the gcd in the least number of steps that at each step makes a "greedy" choice for the remainder (for example $\gcd(21, 13)$, has the choice $21 = (1) \cdot 13 + 8$ or $21 = (2) \cdot 13 - 5$.. The "greedy" choice is the one that leaves a remainder closer to zero).

(b) Prove your algorithm correct

(c) Compute the time complexity of the algorithm.

CSCI 3110 Assignment 2 Solutions

October 13, 2012

1.4 (2 pts) Show that

$$\log(n!) = \Theta(n \log n).$$

To show the upper bound, we compare $n!$ with n^n. By definition, $n! = \prod_{i=1}^{n} i$ while $n^n = \prod_{i=1}^{n} n$ so $n! \leq n^n$ for all $n \geq 1$. This implies that $\log(n!) \leq \log(n^n) = n \log n$ for all $n \geq 1$ and, thus, $\log(n!) = O(n \log n)$.

Lower Bound (Method 1):
$n! = (1 \cdot 2 \cdot \ldots n) \geq (\lfloor n/2 \rfloor) \cdots n \geq (n/2) \cdots (n/2)$ ($n/2$ terms) $= (n/2)^{n/2}$
Hence: $\log(n!) \geq (n/2) \log(n/2) = (1/2) \cdot n(\log n - log2) = (1/2) \cdot (nlogn - 1)$ which is $\Omega(n \log n)$

Lower Bound (Method 2): Compare $n!$ with $(n/2)^{n/2}$. $n! = (1 \cdot 2 \cdot \ldots n) =$
$((1 \cdot n) \cdot (2 \cdot (n-1)) \cdot \ldots \cdot (\lceil n/2 \rceil)$ if n is odd and is times $\lfloor n/2 \rfloor$ if n is even, so $n! = \Omega((n/2)^{n/2}$, and, thus $\log(n!) = \Omega(\log((n/2)^{n/2}))$. Since $\log((n/2)^{n/2}) = (n/2) \log(n/2) = \Theta(n \log n)$, we have that $n! = \Omega(n \log n)$.

1.31 (3 pts) Consider the problem of computing $N! = 1 \cdot 2 \cdot 3 \ldots N$.

(a) If N is a n-bit number, how many bits long is $N!$, approximately in $\Theta(\cdot)$ form)?

$N!$ is $\Theta(\log_2(N!))$ bits long. From 1.4, we know that $\log_2(N!) = \Theta(N \log_2 N)$, so $N!$ is $\Theta(N \log_2 N)$ bits long. Since N is an n-bit number, this can also be written as $\Theta(n2^n)$ bits long.

(b) Give an algorithm to compute $N!$ and analyze its running time.

```
FACTORIAL(N)
1  if (N = 0)
2        Return 1
3  else
4        Return N · FACTORIAL(N − 1)
```

This algorithm directly computes $N!$ using its definition. The algorithm runs for $\Theta(N)$ iterations, and does a multiplication of two $O(N \log N)$-bit numbers and some $O(N \log N)$ time work at each iteration. Thus, the running time is $O(NM(N \log N))$, where $M(N \log N)$ is the time required to multiply two numbers with $O(N \log N)$ bits.

1.8 (4 pts) Justify the correctness of the recursive division algorithm given in page 15, and show that it takes time $O(n^2)$ on n-bit inputs. Proof by Induction on x.

Base case: $x = 0$, alg. returns $q = 0, r = 0$

Inductive hypothesis: The recursive division algorithm works correctly for $0 \text{`} x < X$,

i.e. alg. correctly returns (q, r) such that $\lfloor X/2 \rfloor = qy + r$

Inductive step: If X even; then $X = 2\lfloor X/2 \rfloor = 2qy + 2r$ and alg. returns $Q = 2q$ and $R = 2r$ if $R > y$ alg returns instead $Q = 2q + 1$ and $R = 2r - y$

If X odd; alg. returns $X = 2\lfloor X/2 \rfloor + 1 = 2qy + 2r + 1$ nd alg. returns $Q = 2q$ and $R = 2r + 1$ again if $R > y$ alg returns instead $Q = 2q + 1$ and $R = 2r + 1 - y$

The algorithm terminates after n recursive calls, because each call halves x, reducing the number of bits by one. Each recursive call requires a total of $O(n)$ bit operations, so the total time taken is $O(n^2)$

1.19 (3 pts) The *Fibonacci numbers* $F_0, F_1, \ldots$ are given by the recurrence $F_{n+1} = F_n + F_{n-1}, F_0 = 0, F_1 = 1$. Show that for any $n \geq 1$, $\gcd(F_n+1, F_n) = 1$

We will prove this by induction. Our base case is $n = 1$, where $\gcd(F_2, F_1) = \gcd(1, 1) = 1$. Now, assume that the claim holds for all $1 \leq n \leq k$. By the definition of F, $\gcd(F_{k+1}, F_k) = \gcd(F_k + F_{k-1}, F_k)$. By definition, this is the largest number d such that $d|(F_k + F_{k-1})$ and $d|F_k$. Then $xd = F_k + F_{k-1}$ and $yd = F_k$, for some integers $y > x$. Subtracting these two equations gives that $(x - y)d = F_{k-1}$, so we also have that $F_{k-1}|d$. Since $\gcd(F_k, F_{k-1}) = 1$, by the inductive hypothesis, it must be the case that $d = 1$.

1.27 (3 pts) Consider an RSA key set with $p = 17$, $q = 23$, $N = 391$, and $e = 3$ (as in Figure 1.9). What value of d should be used for the secret key? What is the encryption of the message $M = 41$?

The value of d should be the inverse of $e \mod (p-1)(q-1)$, calculated by the extended Euclid algorithm. Following the algorithm forward, we get:

$$\begin{aligned} \gcd(e, (p-1)(q-1)) &= ex + (p-1)(q-1)y \\ \gcd(3, 352) &= 3x + 352y \\ 352 &= 3(117) + 1 \\ 3 &= (1)(3) + 0 \end{aligned}$$

Now, substituting backwards to find d, we get:

$$1 = -117(3) + 352$$

So, tghe multiplicative inverse of 3 mod 352 is -117. Which is in the same equivalence class as -117 + and multiple of 352:

$$\begin{aligned} -117 + 352 &= 235 \\ 1 &= 3(235) - 2(352) \\ d &= 235 \end{aligned}$$

The encryption of the message $M = 41$ should be

$$\begin{aligned} y &= M^e \mod N \\ &= 41^3 \mod 391 \\ &= 68291 \mod 391 \\ &= 105. \end{aligned}$$

1. greedyGCD

   (a) Algorithm

   GREEDYGCD(a, b)
   1 **if** $(b == 0)$
   2 Return a
   3 $r = MIN(a \bmod b, b - a \bmod b)$
   4 Return greedyGCD(b, r)

2. Correctness
   Similar to textbook page 30. Note that $gcd(a, b)$ is the same as $gcd(a, -b)$

3. Running time
   (Refer to the Euclid Complexity hand-out - for details, I will simply highlight the differences here.)
   We have as before, $r < b$ and hence $2r < b + r \leq a$
   BUT we now have, due to the greedy choice that ensures: $r \leq b/2$
   Add r to each side: $\frac{3r}{2} \leq \frac{b+r}{2}$ using the original inequalities $\leq \frac{a}{2}$
   Hence $3r \leq a$. Multiply by b; giving $3rb \leq ab$ or $rb \leq \frac{ab}{3}$
   So, here we have the product of the arguments is a THIRD of the product of the arguments previous calls. The rest of the analysis follows that of the normal Euclid - except everything is $\log_3$.
   (A tighter argument seems possible. I think, it is possible to make everything work out such that $rb \leq \frac{ab}{4}$ but as the rabbit said: "I'm late! I'm late!")

Dalhousie University Faculty of Computer Science
Design and Analysis of Algorithms I
Assignment 3 CSCI 3110 Due: 12 Oct 2012

(1) Text 1.42
(2) Consider a tree T with vertices $V = \{a, b, c, d, e, f, g, h, i, j\}$ and, rooted at a with edges $E = \{(a, b), (b, d), (d, e), (a, c), (c, f), (c, g), (g, h), (g, i), (g, j)\}$.
   (a) Find the (undirected) connected graph with the maximum number of edges that has T as its DFS-tree (explain your answer).
   (b) Find the (undirected) connected graph with the maximum number of edges that has T as its BFS-tree (explain your answer).
(3) Text 3.2(b)
(4) Text 3.5
(5) Text 3.6

Dalhousie University Faculty of Computer Science
Design and Analysis of Algorithms I
Solution 3 CSCI 3110 Due: 12 Oct 2012

(1) (1.42)
This new cryptosystem is not secure because d can be computed in polynomial time give e, p. Since $\gcd(e, p-1) = 1$, d can be computed in $O(n^3)$ using Extended Euclid Algorithm. The running time is $O(n^3)$ Then we can recover the message by $(M^e)^d \mod p$. However, you need to prove that $(M^e)^d \equiv M \mod p$. Since $ed \equiv 1 \bmod (p-1)$, we have $ed = k(p-1)+1$. Then $M^{ed} = M^{k(p-1)+1} = MM^{k(p-1)} \bmod p$. Since p is prime, $M^{k(p-1)} \equiv 1 \bmod p$ based on Fermat's little theorem. Therefore $M^{ed} = M^{k(p-1)+1} = MM^{k(p-1)} \equiv M \bmod p$.

(2) Consider a tree T with vertices $V = \{a, b, c, d, e, f, g, h, i, j\}$ and, rooted at a with edges $E = \{(a,b), (b,d), (d,e), (a,c), (c,f), (c,g), (g,h), (g,i), (g,j)\}$.

Find the (undirected) connected graph with the maximum number of edges that has T as its DFS-tree (explain your answer).
DFS-Tree and BFS-Tree T

To find the graph G with the maximum number of edges that has T as its DFS-tree, we begin with $G = T$ and add edges that do not invalidate the DFS-tree. Assume that DFS visits the edges of T in the order they are given above. We can only add an edge (u, v) if u has a higher preorder number then v, or T would not be a valid DFS tree for G. Thus, we add an edge from the node with preorder number i to each node with a preorder number less than i which gives a graph with $9 + \binom{10}{2} = 9 + 45 = 54$ edges. We can also add the forward edges $(a,d), (a,e), (a,f), (a,g), (a,h), (a,i), (a,j), (b,e), (c,h), (c,i), (c,j)$ for a total of $54 + 11 = 65$ edges.
Note:: Figures are (for some reason) are below.

3.2(b) (3 pts) Perform depth-first search on each of the following graphs; whenever there's a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge, or cross edge, and give the pre and post number of each vertex

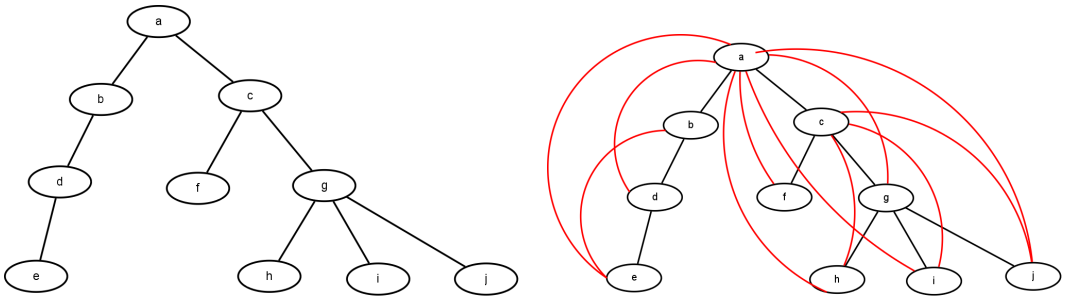
In the graph non-tree edges are shown as dashed lines and are labelled B,C, or F for back, cross, or forward edges. Each vertex is labelled pre:post with its preorder and postorder number.

3.3 (3 pts) Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.

(3) [(a)]

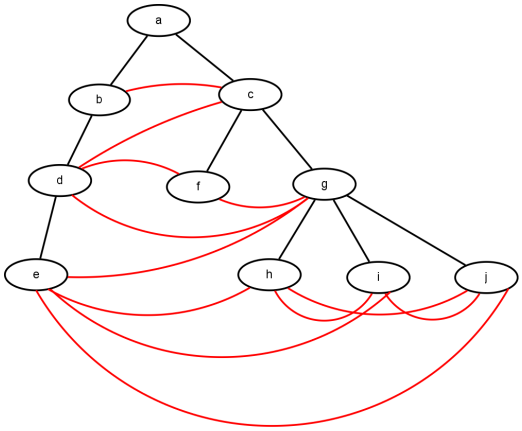
Indicate the pre and post numbers of the nodes.

These are shown in the graph as pre:post



(a) (Tree T)

(b) Q1.(a)



(c) Q1.(b)

FIGURE 1. Graphs for Question 2

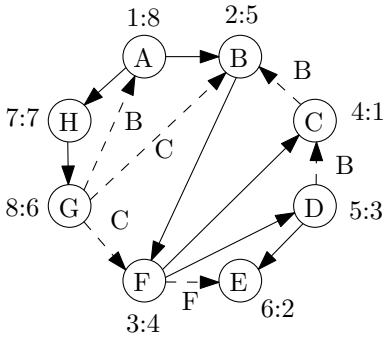


FIGURE 2. Graphs for Question 3

(b) What are the sources and sinks of the graph?

The sources are A and B. The sinks are G and H.

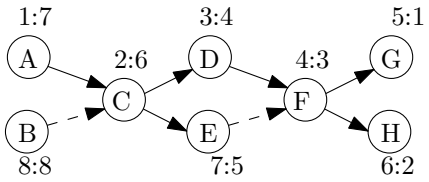


FIGURE 3. Graphs for Question 3

(c) What topological ordering is found by the algorithm?

The topological ordering is B, A, C, E, D, F, H, G.

(d) How many topological orderings does this graph have?

There are three pairs of interchangeable vertices in the ordering, (A, B), (D, E), and (G, H) so there are $2 \cdot 2 \cdot 2 = 8$ possible orderings.

(4) (3.6)

[(a)]Any edge $e = \{u, v\}$, it contributes twice to the degree , i.e., once to u and once to v, therefore, the sum of all degrees is equal to twice the number of edges, namely, $\sum_{u \in V} d(u) = 2|E|$.

Assume there are odd number of vertices whose degree is odd, then we have $\sum_{u \in V}$ is odd. Since $2|E|$ is an even number, $\sum_{u \in V} d(u) \neq 2|E|$, which is contradict to (a). Therefore, there must be even number of vertices whose degree is odd.

Alt Soln:

$$\sum_{u \in V} d(u) = \sum_{odddeg} d(odd - deg) + \sum_{evendeg} d(even - deg) = 2|E|$$

Now, the *R.H.S* is obviously even, as is the first term on the left (sum of evens is even). This means that the term $\sum_{odddeg} d(odd - deg)$ must be *even*. The only way this can happen is if there is an even number of odd vertices.

CSCI 3110 Assignment 3

October 12, 2012

Problems below are from the text *Algorithms by Dasgupta, Papadimitriou, Vazirani*

1. Text 3.3
2. Text 3.13
3. Text 3.15
4. Text 3.26
5. Text 4.4
6. Text 4.5
7. Text 4.8
8. Text 4.14

CSCI 3110 Solution 4

October 20, 2012

Problems below are from the text *Algorithms by Dasgupta, Papadimitriou, Vazirani*

1. Text 3.3 -See A3 solutions
2. Text 3.13

   (a) Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ whose removal leaves G connected. (Hint: Consider the DFS search tree for G.)
   Consider the DFS tree of G starting at any vertex. If we remove a leaf (say v) from this tree, we still get a tree which is a connected subgraph of the graph obtained by removing v. Hence, the graph remains connected on removing v.

   (b) Give an example of a strongly connected directed graph $G = (V, E)$ such that, for every $v \in V$, removing v from G leaves a directed graph that is not strongly connected.
   A directed cycle. Removing any vertex from a cycle leaves a path which is not strongly connected.

   (c) In an undirected graph with 2 connected components it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.
   A graph consisting of two disjoint cycles. Each cycle is individually a strongly connected component. However, adding just one edge is not enough as it (at most) allows us to go from one component to another but not back.

3. Text 3.15

3.15 (4 pts) The police department in the city of Computopia has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However, the city elections are coming up soon, and there is just enough time to run a linear-time algorithm.

(a) Formulate this problem graph-theoretically and explain why it can indeed be solved in linear time.

This problem can be represented by a directed graph $G = (V, E)$ where each intersection is a vertex v and there is an edge between intersections (u, v) if there is a road that goes directly from u to v. The criteria that there is a way to drive legally from any intersection to any other intersection is true if, and only if, there is path from each vertex of G to every other vertex. This is true if, and only if, the graph is Strongly connected, so this can be solved in linear time with DFS by determining if the graph is a single strongly connected component.

(b) Suppose it now turns out the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-wary streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.

We use the same graph G as in the first part of the question and label the town hall s. This property is true if, and only if, there is no path from s to a vertex v such that there is no path from v to s. Then there must be no path from s to a vertex outside of the connected component of s. To determine if the graph has this property we can label each vertex with the number of its connected component and then do a depth first search from s. If s can reach a vertex which is in a different connected component then the new claim is false.

4. Text 3.26

   (a) Show that an undirected graph has an Eulerian tour if and only if all its vertices have even degree. Conclude that there is no Eulerian tour of the Konigsberg bridges.
   Proof: (only if direction): Suppose an Eulerian tour exist. Consider a vertex that is visited (arrived at) k times via different edges as part of the tour (a cycle). Since the tour is Eulerian, each arrival edge must be paired with a distinct departure edge. Hence the degree of the vertex is even.
   (if direction) For the other direction we use induction on the number of vertices in the graph. First note that if $|V| = 2$, then trivially if the degree of both the vertices is even then the graph has an Eulerian tour. Let the statement be true for all graphs with $|V| = n$. We consider a graph G on $n + 1$ vertices such that all its vertices have even degrees. Let u be a vertex in this graph having neighbours $i_1, i_2, \ldots i_{2k}$. Consider a graph G_1 where we remove u and add edges $(i_1, i_2), (i_3, i_4), \ldots (i_{2k-1}, i_{2k})$ to G. Since G_1 has n vertices and the degree of each vertex is the same as in G (and thus even), G_1 must have an Eulerian tour. Replace every occurrence of the extra edges of the form (i_{t-1}, i_t) that we inserted, by (i_{t-1}, u) followed by (u, i_t). This gives an Eulerian tour of G.

   (b) To have an Eulerian path exactly two vertices (end points) must have odd degree and the rest must have even degree

   (c) In-degree for each vertex must be equal to Out-degree.

5. Text 4.4
   Work it out for yourself.

6. (4.5) Number of shortest paths
   We use an array to store the number of shortest paths to the vertices. The number of paths to a vertex v is the sum of the paths to its parents, through which the shortest path from the source s to v are the same and the shortest.
   We modify dijkstra algorithm to find the number of shortest paths. Since the edges have unit length, we can also modify BFS to achieve this goal.

```
DijkstraNumberOfPaths(G)
 1  for all u ∈ V
 2      dist(u) = ∞
 3      paths(u) = 0
 4  dist(s) = 0
 5  paths(s) = 1
 6  H = makequeue(V)
 7  while H is not empty
 8      u = deletemin(H)
 9      for all edges (u, v) ∈ E:
10          if dist(v) > dist(u) + l(u, v)
11              dist(v) = dist(u) + l(u, v)
12              paths(v) = paths(u)
13          else
14              if dist(v) == dist(u) + l(u, v)
15                  paths(v) = paths(v) + paths(u)
```

7. Text 4.8 - next assignment
8. Text 4.14 - next assignment

Dalhousie University Faculty of Computer Science
Design and Analysis of Algorithms
Assignment 5 CSCI 3110 Due: 2 Nov 2012

Problems are from the text (*Algorithms by Dasgupta, Papadimitriou, Vazirani p.120 - 126 and 148 - 155*).

1. **(6pts)** Let $G = (V, E)$ be a connected graph with n vertices and m edges with distinct edge costs. Let T be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with the greatest cost. A spanning tree G is a *minimum bottleneck spanning tree* if there is no other spanning tree of G with a cheaper bottleneck edge.

   (a) Is every min. bottleneck tree of G a MST (min span. tree) of G? Prove or give a counter example.

   (b) Is every MST of G a min bottleneck tree of G? Prove or give a counter example.

2. **(8pts)** For every edge of a connected (communication) graph G, you have a bandwidth b_e. For every pair $u,\ v \in V$ you want to select a $u - -v$ path$P$ on which path the vertices will communicate. The bottleneck rate $b(P)$ of the path is the min bandwidth of any edge it contains ($b(P) = min_{e \in P} b_e$). The best achievable bottleneck rate for the pair $u, v \in G$ is the maximum, over all $U - -V$ paths P in G of the value $b(P)$. It is difficult to keep track of a path for each pair, so maybe a spanning tree T of G could be found such that for every pair of nodes, the unique path between them in T, actually attains the best achievable bottleneck rate for $u, v \in G$.

   Show that such a tree exists, and give an efficient algorithm to find one. That is, find an algorithm constructing a spanning tree T in which, for each pair $u,\ v \in V$, the bottleneck rate of the $u - -v$ path in T id the best achievable.

3. **(2 pts)** Ex. 4.4

4. **(2 pts)** Ex. 4.8

5. **(4 pts)** Ex. 4.14

6. **(4 pts)** Ex. 4.20

7. **(2 pts)** Ex. 5.7

8. **(10 pts)** Ex. 5.9

CSCI 3110 Assignment 5 Solutions

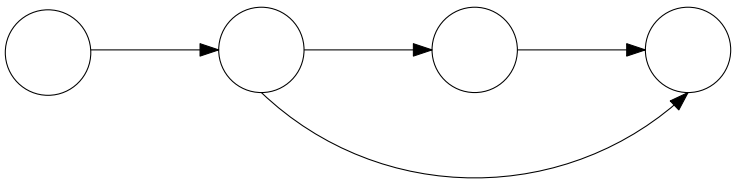
December 5, 2012

4.4 (2 pts) Here's a proposal for how to find the length of the shortest cycle in an undirected graph with unit edge lengths.

When a back edge, say (v, w) , is encountered during a depth-first search, it forms a cycle with the tree edges from w to v . The length of the cycle is level$[v]$ − level$[w] + 1$, where the level of a vertex is its distance in the DFS tree from the root vertex. This suggests the following algorithm:

- Do a depth-first search, keeping track of the level of each vertex.
- Each time a back edge is encountered, compute the cycle length and save it if it is smaller than the shortest one previously seen.

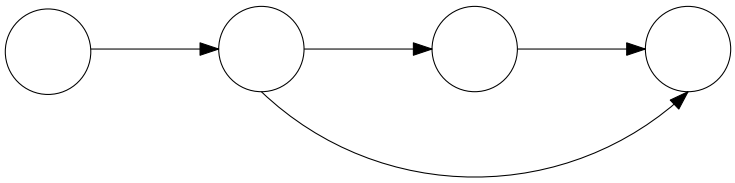
Show that this strategy does not always work by providing a counterexample as well as a brief (one or two sentence) explanation.



This figure shows a graph of 4 vertices a, b, c, d, e. A DFS tree is indicated by marking non-tree edges with dashed lines. The proposed algorithm will only find the length 4 cycle $a \to b \to c \to d \to a$, but will miss the length 3 cycle $a \to d \to e \to a$ because it consists of *two* back edges.

4.8 (2 pts) Professor F. Lake suggests the following algorithm for finding the shortest path from node s to node t in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra's algorithm starting at node s, and return the shortest path found to node t.

Is this a valid method? Either prove that it works correctly, or give a counterexample.



This figure shows a counterexample graph, where a is the constant that we add. When $a = 0$ (ie, the original graph), the shortest path from a to d is $a \to b \to c \to d$. However, when we add $a \geq 2$ to this graph, we penalize longer paths, so the shortest path from a to d is $a \to b \to d$.

4.14 (4 pts) You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between all pairs of nodes, with the one restriction that these paths must all pass through v_0.

Any shortest path from two vertices s to t must pass through v_0. Thus, any such path is composed of a path from s to v_0 and a path from v_0 to t. We first use Dijkstra's algorithm to find the shortest path length from v_0 to any other vertex, $B[\cdot]$. We then reverse the graph and find the shortest path length to v_0 from any vertex, $A[\cdot]$. The shortest path from s to t that passes through v_0 has length $A[s] + B[t]$. This takes $O(|V| + |E|) \log |V|)$ time. Note that we do not store all of the shortest path lengths directly, as that would require $O(n^2)$ time.

4.20 (4 pts) There is a network of roads $G = (V, E)$ connecting a set of cities V. Each road in E has an associated length l_e. There is a proposal to add one new road to this network, and there is a list E' of pairs of cities between which the new road can be built. Each such potential road $e' \in E'$ has an associated length. As a designer for the public works department you are asked to determine the road $e' \in E'$ whose addition to the existing network G would result in the maximum decrease in the driving distance between two fixed cities s and t in the network. Give an efficient algorithm for solving this problem.

A newly added edge $e' = (u, v)$ can only decrease the length of the shortest path from s to t if we follow e'. Thus, we would follow the paths $s \to u \to v \to t$. We can solve this problem by first using Dijkstra's algorithm to compute all shortest paths from s to any vertex, $S[\cdot]$ and to t from any vertex, $T[\cdot]$ (using G^R if the graph is directed). For each possible new edge $e' = (u, v)$, the new shortest path length from s to t is $S[u] + l_{e'} + T[v]$. We choose the edge e' that minimizes this (Or none, if $S[t]$ is never improved upon).

5.7 (2pts) Show how to find the maximum spanning tree of a graph, that is, the spanning tree of largest total weight.

Multiply all the edge weights by -1 and find the minimum spanning tree by any of the standard algorithms: Prim's, Kruskal's *etc*

5.9 (10 pts) The following statements may or may not be correct. In each case, either prove it (if it is correct) or give a counterexample (if it isn't correct). Always assume that the graph $G = (V, E)$ is undirected. Do not assume that edge weights are distinct unless this is specifically stated.

(a) If graph G has more than $|V| - 1$ edges, and there is a unique heaviest edge, then this edge cannot be part of a minimum spanning tree.

FALSE. Any unique heaviest edge that is not part of a cycle *must* be in the MST. A graph with one edge is a counterexample.

(b) If G has a cycle with a unique heaviest edge e, then e cannot be part of any MST.

TRUE. An MST has no cycles, so at least one edge of the cycle e' is not in an MST T. If $e' \neq e$ then we could swap e' for e in T and get a lighter spanning tree.

(c) Let e be any edge of minimum weight in G. Then e must be part of some MST.

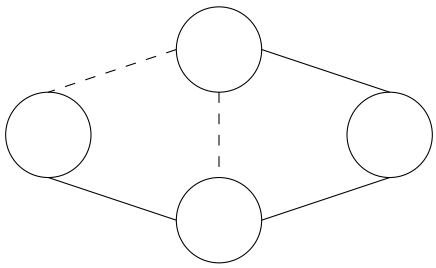
TRUE. An edge of minimum weight is trivially the minimum weight edge of some cut.

(d) If the lightest edge in a graph is unique, then it must be part of every MST.

TRUE. If the lightest edge is unique, then it is the lightest edge of any cut that separates its endpoints.

(e) If e is part of some MST of G, then it must be a lightest edge across some cut of G.

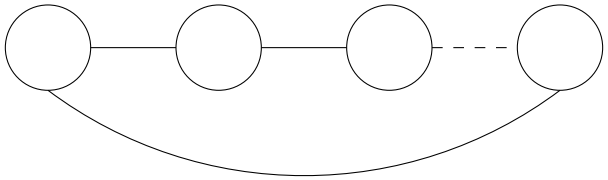
TRUE. If there were a lighter edge e' across some cut of G, then we could replace e with e' and obtain a smaller MST.



(f) If G has a cycle with a unique lightest edge e, then e must be part of every MST.

FALSE. The dashed edge with weight 5 is not part of the MST, but is the lightest edge in the left cycle.

(g) The shortest-path tree computed by Dijkstra's algorithm is necessarily an MST.



FALSE. Dijkstra's algorithm will use the heaviest edge of a cycle if it is on the shortest path from the start s to a node t.

(h) The shortest path between two nodes is necessarily part of some MST.

FALSE. The shortest path between s and t in (g) is not part of any MST.

(i) Prim's algorithm works correctly when there are negative edges.

TRUE. Prim's algorithm always adds the lightest edge between the visited vertices and the unvisited vertices, which is the lightest edge of this cut. Negative weights do not affect this.

(j) (For any $r > 0$, define an r-path to be a path whose edges all have weight $< r$.) If G contains an r-path from node s to t, then every MST of G must also contain an r-path from node s to node t.

TRUE. Suppose that a graph G contains an r-path from a node s to t but that there is an MST T of G that does not contain an r-path from s to t. Then T contains a path from s to t with an edge e of weight $w_e \geq r$. Consider the partition $(S|V-S)$ of vertices made by removing e from T. One vertex e' of the r-path must be along this cut, as the r-path connects s and t. Since $w_{e'} < r$, we can swap e' for e to get a spanning tree that is lighter than T, a contradiction.

5.10 (NOT ASSIGNED- FOR PRACTICE) Let T be an MST of graph G. Given a connected subgraph H of G, show that $T \cap H$ is contained in some MST of H.

Suppose not. Then there is an edge $e \in T \cap H$ across some cut $(S|V-S)$ of H such that another edge across the cut, e', is lighter. However, H is a subgraph of G, so e' is lighter than e across the cut $(S|V-S)$ of G. We could thus swap e' for e in T to get a lighter spanning tree, contradicting that T is an MST.

5.22 (NOT ASSIGNED - FOR PRACTICE) In this problem, we will develop a new algorithm for finding minimum spanning trees. It is based upon the following property:

Pick any cycle in the graph, and let e be the heaviest edge in that cycle. Then there is a minimum spanning tree that does not contain e.

(a) Prove this property carefully.

Suppose that the property is not true. Then there is an edge $e = (u, v)$ of a graph G such that e is the heaviest edge in a cycle and e is part of any MST T of G. Let $e' = (w, x)$ be the lightest edge of the cycle and consider a cut $(S|V - S)$ such that S contains the vertices of the cycle from v to w and $V - S$ contains the vertices of the cycle from u to x. Then the weight of e' is less than or equal to the weight of e across this cut and we can swap e' for e to get an MST T', a contradiction.

(b) Here is the new MST algorithm. The input is some undirected graph $G = (V, E)$ (in adjacency list format) with edge weights $\{w_e\}$.

```
1  sort the edges according to their weights
2  for each edge e ∈ E, in decreasing order of w_e :
3  if e is part of a cycle of G:
4        G = G − e (that is, remove e from G)
5  return G
```

Prove that this algorithm is correct.

We will prove this by induction on the number of edges that have been considered. The base case occurs when we consider the first edge. As this is the largest edge of the graph, it is not part of some spanning tree if its part of a cycle. If it is not part of a cycle, then it is part of any spanning tree.

Now, assume that we have considered k edges and have removed a subset X_k such that there is a spanning T with no edge in X_k. At edge $k + 1$, the algorithm keeps it if and only if it is not part of a cycle of $G \setminus X_k$. By the property, we keep edge $k + 1$ only if it is part of some MST T' of $G \setminus X_k$. By the inductive hypothesis, T' is also an MST of G, so the claim holds.

(c) On each iteration, the algorithm must check whether there is a cycle containing a specific edge e. Give a linear-time algorithm for this task, and justify its correctness.

Let $e = (u, v)$ be an edge of E. If there is a path from u to v that does not use e, then there is such a cycle. Therefore we let $G' = G \setminus e$ and do explore(G, u). If the algorithm visits v then there is such a cycle. This takes linear time.

(d) What is the overall time taken by this algorithm, in terms of $|E|$? Explain your answer.

The overall time taken by the algorithm is the time required to sort the edges, plus the time taken to determine if an edge is part of a cycle, for each edge. This is $O(|E| \log |E| + (|E| \cdot (|E| + |V|)) = O(|E|^2 + |E||V|)$ time.

Dalhousie University Faculty of Computer Science
Design and Analysis of Algorithms
Assignment 6 CSCI 3110 Due: 15 Nov 2010

1. (a) Ex. 2.4 p. 71
   (b) Ex. 2.5(f) and (i)
   (c) Ex. 2.12 p. 73

2. (a) Ex. 2.25
   (b) Ex. 2.29

3. (a) Ex. 2.16 p. 73
   (b) Consider an $m \times n$ array A of integers. Each row and colum of the array are in ascending order: $A[i,j] \leq A[i,j+1], \quad i = 1 \ldots n, \quad j = 1 \ldots m-1$ and $A[i,j] \leq A[i+1,j], \quad i = 1 \ldots n-1 \quad j = 1 \ldots m$.
   Design an algorithm that determines if a given integer x is in A, if so, return it's position otherwise return a -1. Explain your algorithm with words and diagram(s). Show that your alogrithm terminates and find its worst case complexity. *Hint: Start by comparing x with $A[1,m]$*

4. (a) Draw the recursion tree to guess a solution and prove your guess for
   $$T(n) = T(n-a) + T(a) + cn$$
   for constants $a \geq 1$ and $c > 0$. Assume $T(n) = \Theta(1)$ for $n \leq a$.
   (b) Draw the recursion tree to guess a solution and prove your guess for
   $$T(n) = T(\alpha n) + T((1-\alpha)n) + cn$$
   for constants $0 < \alpha < 1$ and $c > 0$.
   (c) Suppose that the splits at each level of quicksort are in the proportion $1-\alpha$ to α where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approx. $-\lg n / \lg \alpha$ and the maximum depth is approx $-\lg n / \lg(1-\alpha)$

5. (a) Give an algorithm that sorts an array of size n, only containing the elements 1,2 and 3, in linear time. Give a plausible argument that your algorithm sorts the array in linear time
   (b) Construct an example input for which quicksort will use $\Omega(n^2)$ comparisions when the pivot is chosen as the median of the first, last and middle elements of the sequence.

CSCI 3110 Assignment 6 Solutions

December 5, 2012

2.4 (4 pts) Suppose you are choosing between the following three algorithms:

1. Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
2. Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
3. Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which would you choose?

These are

1. $T(n) = 5T(n/2) + \Theta(n)$
   We have $a = 5$, $b = 2$, $f(n) = \Theta(n)$. $n^{\log_b a} = n^{\log_2 5} > n^2$, so $f(n) = O(n^{\log_b a - \epsilon})$. This is Case 1 of the Master Theorem, thus $T(n) = \Theta(n^{\log_2 5})$.
2. $T(n) = 2T(n-1) + O(1)$
   This recurrence does not fit the Master Theorem. However, observe that the number of subproblems doubles n times and each subproblem uses $O(1)$ time. Therefore $T(n) = \Theta(2^n)$.
3. $T(n) = 9T(n/3) + T(n^2)$
   We have $a = 9$, $b = 3$, $f(n) = \Theta(n^2)$. $n^{\log_b a} = n^{\log_3 9} = n^2$, so $f(n) = \Theta(n^{\log_b a})$. This is Case 2 of the Master Theorem. Thus, $T(n) = \Theta(n^2 \log n)$.

The algorithm which grows the slowest asymptotically is the 3rd algorithm with a running time of $\Theta(n^2 \log n)$, so we would choose it.

2.5 (2 pts) Solve the following recurrence relations and give a Θ bound for each of them.

(f) $T(n) = 49T(n/25) + n^{3/2} \log n$

We have $a = 49$, $b = 25$, $f(n) = \Theta(n^{3/2})$. $n^{\log_b a} = n^{\log_{25} 49}$, so $f(n) = \Omega(n^{\log_b a + \epsilon})$. This is Case 3 of the Master Theorem, so we check the regularity condition $af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large n. That is, $49((n/25)^{3/2} \log(n/25)) \leq cn^{3/2} \log n$. Since $49((n/25)^{3/2} \log(n/25)) = (49/25^{3/2})(n^{3/2} + (\log n - \log(25))$ the condition holds for $c = (49/25^{3/2}) = 49/125 < 1$. Thus, $T(n) = \Theta(n^{3/2} \log n)$.

(i) $T(n) = T(n-1) + c^n$, where $c > 1$ is some constant

$T(n) = T(n-1) + c^n = T(n-2) + c^{n-1} + c^n$. We can thus unroll this to $T(n) = 1 + c + c^2 + \ldots c^n$. As $c > 1$, this is $\Theta(c^n)$.

2.12 (2 pts) How many lines, as a function of n (in $\Theta(\cdot)$ form), does the following program print? Write a recurrence and solve it. You may assume n is a power of 2.

```
F(n)
1  if n > 1:
2      PRINT_LINE("still going")
3      F(n/2)
4      F(n/2)
```

This program prints one line and then splits into two subprograms with half of the input size each. The recurrence is $T(n) = 2T(n/2) + 1$. Using the Master Theorem, $n^{\log_b a} = n^{\log_2 2} = n$. Since $f(n) = 1$, we have that $f(n) = O(n^{\log_b a - \epsilon})$ so the solution is $\Theta(n)$ lines.

2.25 (4 pts) In Section 2.1 we described an algorithm that multiplies two n-bit binary integers x and y in time n^a, where $a = \log_2 3$. Call this procedure fastmultiply(x, y).

(a) We want to convert the decimal integer 10^n (a 1 followed by n zeros) into binary. Here is the algorithm (assume n is a power of 2):

```
PWR2BIN(n)
1  if n = 1
2      return 1010_2
3  else
4          z =???
5          return FASTMULTIPLY(z, z)
```

Fill in the missing details. Then give a recurrence relation for the running time of the algorithm, and solve the recurrence.

The missing details are $z =$ PWR2BIN$(n/2)$. The recurrence is $T(n) = T(n/2) + n^a$. Unrolling this, we get $\Sigma_{i=0}^{\log_2 n}(n/2^i)^a$ which is $\Theta(n^a)$.

(b) Next, we want to convert any decimal integer x with n digits (where n is a power of 2) into binary. The algorithm is the following:

```
DEC2BIN(x)
1  if n = 1
2      return binary[x]
3  else
4          split x into two decimal numbers x_L, x_R with n/2 digits each
5          return ???
```

Here binary$[\cdot]$ is a vector that contains the binary representation of all one-digit integers. That is, binary$[0] = 0_2$, binary$[1] = 1_2$, up to binary$[9] = 1001_2$. Assume that a lookup in binary takes $O(1)$ time. Fill in the missing details. Once again, give a recurrence for the running time of the algorithm, and solve it.

The missing details are **return** FASTMULTIPLY(PWR2BIN$(n/2)$, DEC2BIN(x_l)) + DEC2BIN(x_R). The recurrence is $T(n) = 2T(n/2) + n^a$, as there are two recursive calls with half of the input size and both FASTMULTIPLY and DEC2BIN use $\Theta(n^a)$ time.
Using the Master Theorem, we have $a = 2$, $b = 2$, and $f(n) = \Theta(n^a)$. Considering $n^{\log_b a} = n^{\log_2 2} = n$, we see that $f(n) = \Omega(n^{\log_b a + \epsilon})$ so we have Case 3. We check the regularity condition $af(n/b) \leq cf(n)$. This is $2(n/2)^a \leq cn^a$. As $2(n/2)^a = (2/2^a)n^a$, this holds for $c = (2/2^a) < 1$. Therefore, by the Master Theorem, the solution to the recurrence is $\Theta(n^a)$.

2.29 (4 pts) Suppose we want to evaluate the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ at point x.

(a) Show that the following simple routine, known as Horner's rule, does the job and leaves the answer in z.

```
1  z = a_n
2  for i = n − 1 downto 0
3        z = zx + a_i
```

Note that for $n = 0$ the loop is not executed and the alg. correctly returns a_0. Now consider the alg. is run for $n + 1$: In the first n iterations of the loop the algorithm computes: $z = a_1 + a_2x + \ldots + a_nx^{n-1}$. On the last iteration, the algorithm computes $a_0 + zx = a_0 + a_1x + \ldots + a_nx^n$. Which is the required computation. Hence, Horner's rule begins with $z = a_n$. It then multiplies z by x and adds a_{n-1}. It thus implements the factorization $p(x) = a_0 + x(a_1 + x(a_2 + x(\ldots)))$ from the inside out and is correct.

(b) How many additions and multiplications does this routine use, as a function of n? Can you find a polynomial for which an alternative method is substantially better?

This routine uses n additions and n multiplications as it does one addition and multiplication for each step and there are n steps. A polynomial with a single non-zero coefficient a_ix^i can be evaluated with a logarithmic number of multiplications and additions using the repeated squaring method of evaluating x^i.

2.16 (2 pts) You are given an infinite array $A[\cdot]$ in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are not given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\log n)$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message ∞ whenever elements $A[i]$ with $i > n$ are accessed.)

Iterate through the powers of 2, $i = 1, 2, 4, \ldots$ until we find the first location $A[i]$ such that $x \leq A[i]$. As $A[i/2] < x$, we know that $n \leq 2x$. We can then do a binary search from $A[0]$ to $A[i]$ to find x if it is in the array. Both steps take $\Theta(\log x) = \Theta(\log n)$ time.

3 (b) (2 pts) Consider an $m \times n$ array A of integers. Each row and colum of the array are in ascending order: $A[i, j] \leq A[i, j + 1]$, $i = 1 \ldots n$, $j = 1 \ldots m - 1$ and $A[i, j] \leq A[i + 1, j]$, $i = 1 \ldots n - 1$ $j = 1 \ldots m$.

Design an algorithm that determines if a given integer x is in A, if so, return it's position otherwise return a -1. Explain your algorithm with words and diagram(s). Show that your alogrithm terminates and find its worst case complexity. *Hint: Start by comparing x with $A[1, m]$*

A simple observation helps in finding the solution: Since each row and column of A is sorted, x is not in a row i if $x > A[i, m]$ and x is not in a column j if $x < A[1, j]$. Compare x with $A[1, m]$. If $x < A[1, m]$ then x can not be in the last column, so we compare x with $A[1, m - 1]$. If $x > A[1, m]$ then x can not be in the first row, so we compare x with $A[1, m]$. We can thus use the following algorithm:

```
SEARCH2DARRAY(A, x)
1  row = 1
2  col = m
3  while row ≤ n and col ≥ 1
4        if x < A[row, col]
5              col = col − 1
6        elseif x > A[row, col]
7              row = row + 1
8        else return (row, col)
9  return −1
```

Each iteration of the while loop either increases the row number by 1 or decreases the column number by 1, so the algorithm terminates in $\Theta(n+m)$ steps, each of which takes $\Theta(1)$ time. Thus, the algorithm terminates in $\Theta(n+m)$ time.

4. (6 pts) (a) Draw the recursion tree to guess a solution and prove your guess for $T(n) = T(n-a) + T(a) + cn$ for constants $a \geq 1$ and $c > 0$. Assume $T(n) = T(1)$ for $n < a$.

`See Fig ex_4_a and ex_4_b included alongside`

The solution for this recurrence is $T(n) = O(n^2)$. Take the worst case of $a = 1$. Then $T(n) = T(n-a) + T(a) + cn = T(n-1) + T(1) + cn$. This is $\Theta()$ of $T(n-1) + cn$, which can be unrolled to $c(n + (n-1) + (n-2) + \ldots + 1) = \Theta(n^2)$.

(b) Draw the recursion tree to guess a solution and prove your guess for $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$ for constants $0 < \alpha < 1$ and $c > 0$.

The solution for this recurrence is $T(n) = O(n \log n)$. We can prove this with substition by claiming that $T(n) \leq dn \log n$ for all n. Assume without loss of generality that $\alpha \geq 0.5$. The base case is $T(1)$ which is clearly $O(n \log n)$. Assume that the claim holds for all $T(i), 1 \leq i < n$. Now, consider

$$\begin{aligned}
T(i+1) &= T(\alpha(i+1)) + T((1-\alpha)(i+1) + c(i+1) \\
&\leq d\alpha(i+1)\log(\alpha(i+1)) + d(1-\alpha)(i+1)\log((1-\alpha)(i+1)) + c(i+1) \\
&\leq d\alpha(i+1)\log(\alpha(i+1)) + d(1-\alpha)(i+1)\log(\alpha(i+1)) + c(i+1) \\
&\leq d(i+1)\log(\alpha(i+1))) + c(i+1) \\
&\leq d(i+1)\log(i+1), \text{ for sufficient } d
\end{aligned}$$

(c) Suppose that the splits at each level of quicksort are in the proportion $1-a$ to a where $0 < a = 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approx. $-\log n / \log a$ and the maximum depth is approx $-\log n / \log(1-a)$.

The minimum depth is attained if we pick children of proportion α, and the maximum depth is obtained if we pick children of proportion $1-\alpha$. After k steps down the tree in the first case, the size of the child sub-array is $n \times \alpha^k$. If this sub-array is a leaf in the recursion tree, then its size is 1 and we get the equation $1 = n \times \alpha^k$. From here,

$$k = \log_\alpha 1/n = -\log n / \log \alpha.$$

The other case is symmetric. After k steps down the tree, the size of the child sub-array is $n \times (1-\alpha)^k$. If this sub-array is a leaf in the recursion tree, then its size is 1 and we get the equation $1 = n \times (1-\alpha)^k$. From here,

$$k = \log_{1-\alpha} 1/n = -\log n / \log(1-\alpha)$$

5. (4 pts) (a) Give an algorithm that sorts an array of size n, only containing the elements 1,2 and 3, in linear time. Give a plausible argument that your algorithm sorts the array in linear time

Since we only have 1, 2 or 3 as array elements:

- Call Partition specifying the pivot element to be the integer 2.
- Now we have: $A[1 \ldots q-1] \leq 2$ and $A[q \ldots n] > 2$; the latter sub-array only contains threes.
- Call `Pivot-Partition` again on the sub-array $A[1 \ldots q-1]$ with pivot element = 1.
- Now we have: $A[1 \ldots q'-1] \leq 1$ and $A[q' \ldots q] > 1$; the latter sub-array only contains twos and the former only contains ones.

Since Partition is a $\Theta(n)$ algorithm, and there are two calls to Partition with only constant overhead ($\Theta(1)$) for assigning the pivot, the algorithm described above is in $\Theta(n)$.

(b) Construct an example input for which quicksort will use $\Omega(n^2)$ comparisions when the pivot is chosen as the median of the first, last and middle elements of the sequence.

For the median-of-three pivot based Partition to cause quicksort to take $\Omega(n^2)$ time, the partition must be lop-sided on EVERY call. This will happen if two of the three elements examined (as potential pivots) are the two smallest or the two largest elements in the file. This must happen consistently through all the partitions. For example, consider an EIGHT element array:
(1, (the rest of the numbers must be larger than two), 2)
pivot = Median(1,2, something larger than 2) = 2
move 2 to end as pivot.
(1, (the rest of the numbers must be larger than two), 2)
PARTITION will swap $A[8]$ with $A[2]$ (the element next to 1). Since all other elements are larger than 2, no more swaps will occur and the partition will move to after 2 (now in $A[2]$). Quicksort will recursively sort from the third index to the last one. To continue the pattern of bad splits, the third smallest element must be in $A[3]$. and since the element originally in $A[8]$ was originally in $A[2]$, the element originally in $A[2]$ must be the fourth smallest. Continuing this pattern we see that the sixth smallest must be in $A[4]$ and fifth smallest in $A[5]$ So, the original array must be:

[1, 4, 3, 6, 5, 8, 7, 2]

Running quicksort:
pivot = 2, so array becomes ( )[denotes partition loc.):

[1, 2,][3, 6, 5, 8, 7, 4]

pivot = 4, so array becomes ( )[denotes partition loc.):

[1, 2, 3, 4][5, 8, 7, 6]

pivot = 6, so array becomes ( )[denotes partition loc.):

[1, 2, 3, 4, 5, 6][7, 8]

etc. etc.

Dalhousie University Faculty of Computer Science
Design and Analysis of Algorithms
Assignment 7 CSCI 3110 Practice Dec 2012

1. For the problem of multiplying matrices $A_1 \dots A_4$, draw the recursion tree. Identify the *overlapping* sub-problems.

2. (a) Ex. 6.1
   (b) Ex. 6.2

3. (a) Ex. 6.8
   (b) Ex. 6.11

4. Moving on a checkerboard:
   Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:
   1. the square immediately above,
   2. the square one up and one to the left (but only if checker is not already in leftmost column),
   3. the square one up and one to the right (but only if checker not already in the rightmost column),
   Each time you move from square x to square y, you receive $p(x, y)$ dollars. You are given $p(x, y)$ dollars for all pairs (x, y) for which a move from x to y is legal. Do not assume that $p(x, y)$ is positive.

Dalhousie University Faculty of Computer Science
Design and Analysis of Algorithms
Solutions to practice problems CSCI 3110 Practice Dec 2012

(1) **(a) 6.1:** A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S. For instance, if S is $5, 15, -30, 10, -5, 40, 10$, then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:
Input: A list of numbers, $a_1, a_2, \ldots, a_n$.
Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).
For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55 .
(Hint: For each $j \in \{1, 2, \ldots, n\}$, consider contiguous subsequences ending exactly at position j.)

Let $s[j]$ be the sum of the maximum contiguous sequence that ends at position j. Then for $s[j+1]$ we can either start a new contiguous sequence with score a_{j+1} or continue the contiguous sequence with score $s[j] + a_{j+1}$. The recurrence is:

$$s[j] = \begin{cases} a_j, & \text{if } j = 0 \\ \max\{s[j-1] + a_j, a_j\}, & \text{if } j > 0 \end{cases}$$

We can implement this recurrence in linear time by computing $s[0], s[1], \ldots, s[n]$. To recover the sequence, we first scan through $s[]$ to find the location $s[j]$ which is maximum. We scan backwards from j to find the negative element $s[i]$ at the beginning of the maximum contiguous sequence, or have $i = -1$ (we could also directly store these values during the search). Then the maximum contiguous sequence is $a_{i+1}, a_{i+2}, \ldots A_j$ with score $s[j]$. The backtracking takes linear time as well.

(b) 6.2 : You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.
You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty–that is, the sum, over all travel days, of the daily penalties.
Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Let $p[j]$ be the minimum penalty required to travel to hotel j. To get the minimum penalty required to travel to hotel $j+1$, we could drive from any of the previous hotels i to j and have a penalty of $p[i] + (200 - (a_j - a_i))^2$. The recurrence is:

$$p[j] = \begin{cases} a_j, & \text{if } j = 1 \\ \min_{1 \le i < j}\{p[i] + (200 - (a_j - a_i))^2\}, & \text{if } j > 0 \end{cases}$$

We can implement this recurrence in $O(n^2)$ time by computing $p[0], p[1], \ldots, p[n]$. To determine the hotels that should be stopped at, we store the hotel i that we should travel from to reach each j that got the penalty $p[j]$. Starting with n, we can follow these hotel links backwards to get the sequence of hotels. The backtracking takes linear time, so this takes $O(n^2)$ time in total.

(2) **(a) 6.8 :** Given two strings $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$, we wish to find the length of their longest common substring, that is, the largest k for which there are indices i and j with $x_i x_{i+1} \ldots x_{i+k-1} = y_j y_{j+1} \ldots y_{j+k-1}$. Show how to do this in time $O(mn)$.

Let $s[i][j]$ be the length of the longest common substring that ends with matching x_i and y_j. This is either 0, or, if $x_i = x_j$, then we can add x_i to the longest common substring that ends with matching x_{i-1} and y_{j-1}. The recurrence is:

$$s[i][j] = \begin{cases} 0, & \text{if } i = 0,\ j = 0, \text{ or } x_i \neq x_j \\ 1 + s[i-1][j-1], & \text{if } x_i = x_j \end{cases}$$

This can be implemented to run in $O(mn)$ time by filling the table in either row by row or column by column. To recover the longest common substring, we scan the table once to find the maximum value $s[i][j]$. The longest common substring is then $x_{i-s[i][j]-1} \ldots x_i$ (equivalently, $y_{j-s[i][j]-1} \ldots y_j$) or $\emptyset$ if the maximum is 0. Backtracking also takes $O(mn)$ time.

(b) 6.11 : Given two strings $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$, we wish to find the length of their longest common subsequence, that is, the largest k for which there are indices $i_1 < i_2 <$

$\ldots < i_k$ and $j_1 < j_2 < \ldots < j_k$ with $x_{i_1} x_{i_2} \ldots x_{i_k} = y_{j_1} y_{j_2} \ldots y_{j_k}$. Show how to do this in time $O(mn)$.

Let $s[i][j]$ be the length of the longest common subsequence of $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$. We have three options. The longest common subsequence of $s[i][j]$ could be the longest common subsequence of $s[i-1][j]$, or of $s[i][j-1]$, or, if $x_i = x_j$, then we could add x_i to the longest common subsequence of $s[i-1][j-1]$. The recurrence is:

$$s[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ \max\{s[i-1][j], s[i][j-1], s[i][j]+1\}, & \text{if } x_i = x_j \\ \max\{s[i-1][j], s[i][j-1], & \text{if } x_i \neq x_j \end{cases}$$

This can be implemented to run in $O(mn)$ time by filling the table in either row by row or column by column. Similar to question 6.2, we can store the locations of the last matched values a and b for each $s[i][j]$. The length of the longest common subsequence is $s[n][m]$ and we backtrack using the stored values to recover the actual sequence. Backtracking takes linear time, so this algorithm runs in $O(mn)$ time in total.

(3) **Moving on a checkerboard :**

Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:
(a) the square immediately above,
(b) the square one up and one to the left (but only if checker is not already in leftmost column),
(c) the square one up and one to the right (but only if checker not already in the rightmost column),
Each time you move from square x to square y, you receive $p(x, y)$ dollars. You are given $p(x, y)$ dollars for all pairs (x, y) for which a move from x to y is legal. Do not assume that $p(x, y)$ is positive.

Let $g[i, j]$ be the gain (in dollars) in the most profitable way to square (i, j), from row 1.For this problem - the sub-problems to consider are: going from some square in row 1 to a particular square in row i. To obtain most profitable way to sq. (i, j) - we need to most profitable way to get to get to sqs. $(i-1, j-1)$; $(i-1, j)$; $(i-1, j+1)$ (This shows the sub-problem optimality property) We want to calculate $g[1, n]$ (Please see dev. of recurrence notes posted on web-page)

$$\begin{aligned} g[i,j] &= max(g[i-1,j-1] + p([i-1,j-1], \nearrow) \\ &\quad , \quad g[i-1,j] + p([i-1,j], \uparrow) \\ &\quad , \quad g[i,j+1] + p([i-1,j+1], \nwarrow)) \\ g[i,j] &= 0, \ when \ i = 1, j = 1, 2 \ldots n \end{aligned}$$

Notice, to get the most profitable way to get to sqs. $(i+1, j-1)$ OR sq. $(i+1, j)$ OR sq. $(i+1, j)$ we will need to check sq. (i, j). Hence we have overlapping sub-problems. We can make a table, and search from bottom to up, storing how we get to sq (i, j) in $w[i, j]$ Below: `R` is $\nearrow$ etc..

```
CHECKER_PROFIT(n, p)
 1. for j = 1 to n
 2.        g[1,j] = 0
 3. for i = 2 to n
 4.     for j = 1 to n
 5.         g[i,j] = -infinity
 6.         if j > 1
 7.             g[i,j] = g[i-1,j-1] + p(i-1,j-1,"R")
 8.             w[i,j] = j-1
 9.         if g[i-1,j] +  p(i-1,j,"UP") > g[i,j]
10.             g[i,j] = g[i-1,j] +  p(i-1,j,"UP")
11.             w[i,j] = j
12.         if j < n && g[i-1,j+1] +  p(i-1,j+1,"L") > g[i,j]
13.             g[i,j] = g[i-1,j+1] +  p(i-1,j+1,"L")
14.             w[i,j] = j+1
15.  return g and w
```

Once the table is filled all can be looked up. The maximum profit is at some $g[n, j]$, where $1 \leq j \leq n$. We can then backtrack through the board by finding, for a square (i, j), a square $(i-1, j-1)$, $(i-1, j)$, or $(i-1, j+1)$ such that the moving from that square to (i, j) results in the penalty $g[i, j]$. First loop through $g[n, j]$ to find the maximum profit. Then, the following recursive procedure will output the visited squares using CHECKER_SQUARES(n, p, n, j):

```
CHECKER_SQUARES(n, p, i, j)
1. print (i,j)
2. if i == 1
3.     return
4. if j > 1 && g[i,j] == g[i-1,j-1] + p(i-1,j-1,"R")
5.     CHECKER_SQUARES(n, p, i-1, j-1)
6. if g[i,j] == g[i-1,j] + p(i-1,j,"UP")
7.     CHECKER_SQUARES(n, p, i-1, j)
8. if j < n && g[i,j] == g[i-1,j+1] + p(i-1,j+1,"L")
9.     CHECKER_SQUARES(n, p, i-1, j+1)
```

The running time of the algorithm is $O(n^2)$ -clear from pseudocode.

CSCI 3110 mini-Assignment 7 Solutions

December 5, 2012

Ex.1 2.18 Consider the task of searching a sorted array $A[1...n]$ for a given element x: a task we usually perform by binary search in time $O(\log n)$. Show that any algorithm that accesses the array only via comparisons (that is, by asking questions of the form is $A[i] \leq z$?), must take $\Omega(\log n)$ steps.

Consider the decision tree for the problem. At each internal node a comparision occurs $A[i] \leq x$, the result of which a decision is made for the next part of the search . The leaves of the decision tree represent the possible outputs of the algorithm: Here they are the n indices of the elements plus the possibility that the element x is not present in A. This means the decision tree must have at least $n+1$ leaves. A path from root to a leaf represents an execution of the algorithm. Since the tree has at least $(n+1)$ leaves, the decision tree must have height at least $\log(n+1)$, giving an execution time in $\Omega(\log(n+1))$.

Ex. 2. In the algorithm SELECT covered in class, the input is divided into groups of 5. Will the algorithm work in linear time if they are divided into (i) groups of 7? (ii) Give an argument that SELECT will not run in linear time if groups of 3 are used. (Bonus: if you can obtain a general expression, then you could answer both questions).

SOLUTION 1:
(i) Yes, the algorithm will work in linear time if they are divided into groups of 7. There are $n/7$ groups with at least $4 \cdot 1/2 \cdot (n/7)$ elements that are less than or equal to the median of the medians, and at least as many that are greater than or equal to the median of medians. Thus, the larger subset after partitioning has at most $n - 2n/7 = 5n/7$ elements. The running time is $T(n) = T(n/7) + T(5n/7) + \Theta(n)$ Solve using substitution:
Assume:

$$T(k) \leq c^* k \ for \ k < n$$

$$T(k) \leq c^*(n/7) + c^*(5n/7) + c_1 n = c^*(6n/7) + c_1 n = c^* n + (c_1 - c^*(1/7))n \leq c^* n$$

which is linear if $(c_1 - c^*(1/7)) \leq 0$ (for a more careful analysis, see class notes on SELECT (attached).

(ii) Using a similar analysis we get that there are only $2 \cdot 1/2 \cdot (n/3)$ elements that are guaranteed to be larger or smaller than the median of medians. The recurrence is $T(n) = T(n/3) + T(2n/3) + \Theta(n) = \Theta(n \log n)$. (solve using substitution)
SOLUTION 2:
If we use k elements as a group, the number of elements less than the median is: $\lceil k/2 \rceil \left(\lceil \frac{1}{2} \lceil \frac{n}{k} \rceil \rceil - 2 \right) \geq \frac{n}{4} - k$. In the worst case, we need to recursively call Select for $n - \left(\frac{n}{4} - k \right) = \frac{3n}{4} + k$ times. Thus we have:

$$T(n) = T\left(\lceil \frac{n}{k} \rceil \right) + T\left(\frac{3n}{4} + k \right) + O(n)$$

Using iterations, we have:

$$
\begin{aligned}
T(n) &\leq c\left(\lceil\frac{n}{k}\rceil\right) + c\left(\frac{3n}{4} + k\right) + O(n) \\
&\leq c\left(\frac{n}{k} + 1\right) + \frac{3cn}{4} + ck + O(n) \\
&= \frac{cn}{k} + \frac{3cn}{4} + c(k+1) + O(n) \\
&= cn(\frac{1}{k} + \frac{3}{4}) + c(k+1) + O(n) \\
&\leq cn
\end{aligned}
$$

where $\frac{1}{k} + \frac{3}{4} < 1 => k > 4$

Hence, the group size must be larger than 4 to make it linear.