CPSC 320 (Intermediate Algorithm Design and Analysis). Summer 2009.
Instructor: Dr. Lior Malka
Final Examination, July 24th, 2009

Student ID: _____

INSTRUCTIONS:

- There are 6 questions printed on pages $1 - 7$.
- Exam duration: 2 hours and 30 minutes.
- No aids are allowed, except for 1 letter size, double-sided cheat sheet.
- Start with easy questions and do not spend too much time on any section. The marks for each question appear in square bold brackets next to the question.
- Before you answer a question, read all of its sections and make sure that you fully understand them. You are always encouraged to ask for clarifications.
- When asked to give an algorithm, the intention is that you provide the asymptotically most efficient implementation and describe it concisely. Unless otherwise stated, no pseudocode is necessary.
- Unless otherwise stated, when asked to provide lower or upper bounds, the intention is that you provide the tightest bounds possible under worst case analysis.
- **Answers are to be written on examination paper only**. You can use another piece of paper as a draft. If you also use the examination paper as a draft, do not forget to cross your notes.
- Do not use red pen.

Good Luck!

Q1	
Q2	
Q3	
Q4	
Q5	
Q6	
Total	

1. **[10] General**. Let A be an unsorted array of n distinct integers. The algorithm $SortAndPrint(A, i, j)$ first sorts A and then prints all the elements between index i to index j in A. Describe an algorithm that outputs the same numbers, possibly in a different order, but in time $\Theta(n)$. The input to your algorithm is an unsorted array A of n distinct elements and indices i, j such that $1 \leq i, j \leq n$. Informally explain why your algorithm is correct and why it runs in time $\Theta(n)$.

   `Answer.` Let A' be the array A after sorting. The element $A'[i]$ is an element of A whose order is i. Thus, the required algorithm should output all the numbers in A whose order is between i to j. Intuitively, we achieve running time $O(n)$ by using the `Select` algorithm and then printing everything in between the end points. Formally, on input an array A and integer i, algorithm `Select` outputs the i-th order element of A in time $\Theta(n)$. We store $x =$`Select`(A, i) and $y =$`Select`(A, j), and then we scan A, printing each element $A[k]$ if $x \leq A[k] \leq y$. Two invocations of `Select` are $2 \cdot \Theta(n)$ and scanning A is $\Theta(n)$. The total is $\Theta(n)$.

2. **[15] Recurrences.**

   (a) **[5]** Using the *Master Theorem*, solve $T(n) = 9T(\frac{n}{3}) + n$. Provide the values you are using (e.g., a, b, c, ϵ, f).

   `Answer.` In this case we have $a = 9$, $b = 3$, $f(n) = n$, and $n^{\log_b(a)} = n^2$. For $\epsilon = \frac{1}{2}$ we get that $f(n) = O(n^{\log_b(a)-\epsilon})$, and therefore the first case of the master theorem applies. That is, $T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^2)$.

   (b) **[10]** Using the *Guess and Test Method*, prove an upper bound on $T(n) = 9T(\lfloor \frac{n}{3} \rfloor) + n$. Your bound should apply to any $n \geq 1$, and you can define T for the base case in any way.

   `Answer.` We already know from the previous section that $T(n) \leq O(n^2)$, so we try to prove the induction step first, assuming $T(k) \leq k^2$ for all $k < n$:

$$T(n) = 9 \cdot T(\frac{n}{3}) + n \leq 9\Big(\frac{n}{3}\Big)^2 + n = n^2 + n.$$

   Since we need to show that $T(n) \leq n^2$, the induction step does not work and we need to revise it. We try proving that $T(n) \leq n^2 - n$. The induction step is now

$$T(n) = 9 \cdot T(\frac{n}{3}) + n \leq 9\Big((\frac{n}{3})^2 - \frac{n}{3}\Big) + n = n^2 - 2n \leq n^2 - n,$$

   which coincides with the induction hypothesis. Turning our attention to the base case, we need to find an $n \in \mathbb{N}$ for which $T(n) = 9 \cdot T(\frac{n}{3}) + n \leq n^2 - n$ holds. It turns out that if we take $T(0) = 0$, then $T(0) = 9 \cdot T(0) + 0 \leq 0$ holds. We conclude that our guess $T(n) \leq n^2 - n$ can be formally proved using induction by defining $T(n) = 0$ if $n = 0$ and $T(n) = 9 \cdot T(\frac{n}{3}) + n$.

3. **[20] Dynamic Programming**. Let A be an array of numbers $a_1, a_2, \ldots, a_n \in \mathbb{N}$. The *longest increasing subsequence* of A is the largest ordered subset of $a_1, a_2, \ldots, a_n$ in which each element is strictly greater than the previous one. For example, the longest increasing subsequence of $5, 2, 8, 6, 3, 6, 9, 7$ is $2, 3, 6, 9$. The *length* of this subsequence is 4. We can compute the length of the longest increasing subsequence by creating an array M, initializing it with default values, and then computing new entries based on previous ones.

   (a) **[4]** Explain why the problem of computing the length of the longest increasing subsequence can be solved using dynamic programming.

   `Answer.` There are two reasons: *optimal substructure*, and *overlapping subproblems*. The optimal substructure is due to the fact that we can compute the length of the longest increasing subsequence for the numbers $a_1, \ldots, a_i$ by looking at the values of the longest increasing subsequence for the numbers $a_1, \ldots, a_{i-1}$. The overlapping subproblems are due to the fact that at stage i we recompute more than once the same solutions for subproblems from stages $i' < i$.

   (b) **[8]** Define the size of the array M, the initial default values of entries in M, and the recursive formula that computes new entries in M based on previous ones.

   `Answer.` We define $M[i, j]$ to be the length of a longest increasing subsequence in the subarray $A[1, \ldots, i]$ such that the corresponding longest increasing subsequence ends at index j.[1] The value of i ranges from 1 to n, and so is the value of j. Thus, M is of size n by n. Intuitively, the length of the longest increasing subsequence in $a_1, \ldots, a_i$ is computed by looking at the values $a_1, \ldots, a_{i-1}$. For each of these, we look at $A[j]$ and decide whether $A[i]$ could be appended to the sequence or not. Then, we take the maximum over these values. Formally,

$$M[i, j] = \begin{cases} null & \text{if } j > i \text{ or } j = 0 \\ 1 & \text{if } i = 1 \text{ and } j = 1 \\ \max_{1 \le k \le j}(\delta_k) & \text{otherwise,} \end{cases}$$

   where $\delta_k = M[i-1, k]$ if $A[k] \ge A[i]$ and $\delta_k = M[i-1, k] + 1$ otherwise.

   (c) **[8]** After computing M, what entry contains the *length* of the longest increasing subsequence in A? Prove it using induction.

   `Answer.` The index i for which $M[n, j]$ takes the maximal value (over all $1 \le j \le n$) is the length of the longest increasing subsequence in A. Proof: by induction on i, we prove that $M[i, j]$ is the length of a longest increasing subsequence in $A[1, \ldots, i]$ whose corresponding longest increasing subsequence ends at index j. The base case trivially holds when $i = 1$ because j is also 1. Assume the induction hypothesis for all numbers smaller than i. For the

[1] Notice that saying that the subsequence ends at index j is not the same as saying that the subsequence ends at the number $A[j]$.

induction step, assume towards contradiction that there is $j' \leq i$ and a longest increasing subsequence S in $A[1, \ldots, i]$ ending at index j' such that $M[i, j'] < |S|$. (Case 1:) If $A[j'] \geq A[i]$, then S is also a longest increasing subsequence in $A[1, \ldots, i-1]$ ending at index $j' \leq i-1$. By the definition of M, it follows that $M[i, j] = M[i-1, j'] < |S|$, and contradiction to the induction hypothesis for $i-1$. (Case 2:) If $A[j'] < A[i]$, then $S - \{A[i]\}$ is a longest increasing subsequence in $A[1, \ldots, i-1]$. The length of this sequence is $|S| - 1$, and it ends at some index $j' \leq i-1$. By the definition of M, it follows that $M[i-1, j] < |S| - 1$ for all j such that $A[j] < A[i]$. Thus, in particular $M[i-1, j'] < |S| - 1$, and contradiction to the induction hypothesis for $i-1$. The induction follows.

4. **[15] Amortized Analysis**. A *stack* is a data structure with the operation `push`(x), which pushes an element into the stack, and `pop`(), which pops an element out of the stack. Let $k \in \mathbb{N}$. A third operation called `debug` is added to the stack. This operation prints the content of the stack to the screen. It is triggered automatically by the stack after each sequence of k `push` or `pop` operations.

   (a) **[9]** Assuming the size of the stack never exceeds k, describe the amortized cost of each operation and use these costs to upper bound the total cost of n stack operations.

   `Answer.` In a sequence of n stack operations there are at most $\frac{n}{k}$ `debug` operations, each costing at most k steps, for a total of $\frac{n}{k} \cdot k = n$ steps. Thus, we assign \$ 2 to the `push` operation, \$ 0 to the `pop` operation, \$ 0 to the `debug` operation, and \$ n to the `stack` (alternatively, we can assign \$ n to the first `debug` operation and \$ 0 to the following `debug` operations, and specify that the remaining credit goes to the stack). Now, in each sequence of n of the 3 operations, every `push` and `pop` operation costs 1 step. Thus, these operations are covered by the \$ 2 assigned to the `push` operation. `Debug` operations are also covered by the stack because there are $\frac{n}{k}$ of them, each of cost at most k. Thus, the time complexity of a sequence of n operations is $2n + n = O(3n)$.

   (b) **[3]** Give a sequence of n stack operations that shows that your upper bound from Section (a) is also a lower bound. Assume again that the size of the stack never exceeds k.

   `Answer.` There are several sequences showing that the lower bound is $\Omega(n)$. For example, a sequence of alternating `push` and `pop` operations requires $2 \cdot \frac{n}{2} = n$ steps.

   (c) **[3]** Without limiting the size of the stack, give a lower bound on the cost of n stack operations.

   `Answer.` Consider a sequence of n `push` operations. The first `debug` costs k steps, the second costs $2k$ steps, and so on until the last `debug`, which costs $\frac{n}{k} \cdot k$. The total cost is therefore lower bounded by $k \sum_{i=1}^{n/k} i = \frac{n}{2}(\frac{n}{k} + 1) = \Omega(\frac{n^2}{k})$.

5. **[20] Graph algorithms**. Let $G = \langle V, E \rangle$ be an undirected connected graph with positive weights on the edges and let $s \in V$. Recall that after invoking `Dijkstra`(G, s), each vertex v has a distance value $d(v)$. We define a graph G' from G by adding exactly one arbitrary edge $\langle u, v \rangle$ of weight $w_{u,v} > 0$. That is, $G' = \langle V, E' \rangle$, where $E' = E \cup \{\langle u, v \rangle\}$ and $\langle u, v \rangle \notin E$. In the new graph G', we denote by $d'(v)$ the weight of a shortest path from any vertex $v \in V$ to s.

   (a) **[6]** Prove that the distance of at least one of u and v is unaffected by the new edge. Formally, prove that either $d'(u) = d(u)$ or $d'(v) = d(v)$.

   `Answer.` To prove this claim we show that if $d'(u) < d(u)$, then $d'(v) = d(v)$ (the other case is symmetric). If $d'(u) < d(u)$, then the shortest path from u to s must use the edge $\langle u, v \rangle$, or else contradiction to the choice of $d(u)$. Thus, the shortest path from u to s starts at u, goes to v on the edge $\langle u, v \rangle$, and then continues to s on the shortest path from v to s. The path cannot return to u because all the edges have positive weight. Since the shortest path from v to s in G' does not pass through u, the shortest path from v to s in G' does not use the edge $\langle u, v \rangle$. Hence, $d'(v) < d(v)$ contradicts the choice of $d(v)$ as the weight of a shortest path in G. Thus, $d'(v) = d(v)$.

   (b) **[6]** Prove that if $w_{u,v} \geq |d(u) - d(v)|$, then the distance of u and v is unaffected by the new edge (formally, $d'(u) = d(u)$ and $d'(v) = d(v)$).

   `Answer.` Assume towards contradiction that either $d'(u) < d(u)$ or $d'(v) < d(v)$. Suppose $d'(u) < d(u)$ (the other case is symmetric). As we saw in the previous section, the shortest path from u to v in G' starts at u, goes to v, and then continues to s on the shortest path from v to s in G. Thus, $d(u) > d'(u) = w_{u,v} + d'(v) = w_{u,v} + d(v)$. This leads to the inequality $w_{u,v} < d(u) - d(v)$, and contradiction.

   (c) **[8]** Prove that if $w_{u,v} \geq |d(u) - d(v)|$, then the distances of all vertices are unaffected (formally, $d(t) = d'(t)$ for any $t \in V$). *Hint: show that no shortest path in G' contains the edge $\langle u, v \rangle$.*

   `Answer.` Assume towards contradiction that there is $t \in V$ such $d'(t) < d(t)$, and consider the shortest path $p_{t,s}$ from t to s in G'. This path must use the edge $\langle u, v \rangle$, or else contradiction to the choice of $d'(t)$. Suppose the path starts at t, passes through u, then uses the edge $\langle u, v \rangle$, and continues on the shortest path $p_{v,s}$ from v to s. This implies that the weight of the shortest path from u to s in G' is $w_{u,v} + d'(v) = w_{u,v} + d(v) < d(u)$, which contradicts $w_{u,v} \geq |d(u) - d(v)|$

6. **[20] NP-Completeness**. Consider the efficient compiler from 3-SAT to CLIQUE. The input to the compiler is a 3-CNF formula φ, and the output is $\langle G(\varphi), k \rangle$, where $G(\varphi)$ is a graph and k is a number.

   (a) **[3]** We represent φ using an array A with 3 columns. The number of rows equals the number of clauses in φ. For each $j \in \{1, 2, 3\}$ and i, the value of $A[i, j]$ is t (respectively, -t) if the literal

x_t (respectively, $\overline{x}_t$) appears in the j-th position of the i$-th$ clause. Draw A for the formula

$$\varphi = (\overline{x}_1 \vee x_3 \vee x_4) \wedge (x_3 \vee x_5 \vee \overline{x}_5) \wedge (\overline{x}_2 \vee x_7 \vee x_9) \wedge (x_6 \vee x_2 \vee x_5) \wedge (\overline{x}_7 \vee \overline{x}_8 \vee x_9)$$

Answer.

-1	3	4
3	5	-5
-2	7	9
6	2	5
-7	-8	9

(b) **[10]** Give pseudocode for the efficient compiler from 3-SAT to CLIQUE. On input an array A representing a 3-CNF formula φ as in Section (a), the compiler outputs $\langle B, k\rangle$, where B is a matrix representation of the graph $G(\varphi)$ and k is a number. Briefly and informally explain what your algorithm does and why it is correct.

Answer. The algorithm numbers entry $\langle i, j\rangle$ in A as vertex $3(i-1)+j$ in B. It then connects all $3n$ vertices, unless the literals corresponding to the entries are complements, or they are in the same row. Correctness follows from the fact that in $G(\varphi)$ we do not connect complementing literals or literals in the same clause in φ. Thus, A (which represents φ) has a satisfying assignment if and only if B (which represents $G(\varphi)$) has a clique of size k.

```
Compiler(A)
    n = A.rows
    B is a 3n by 3n matrix
    For i = 1 to n
        For j = 1 to 3 // the number 3(i − 1) + j is a label for a vertex in G(φ)
            For i′ = 1 to n
                For j′ = 1 to 3 // the number 3(i′ − 1) + j′ is a label for a vertex in G(φ)
                    if i ≠ i′ and A[i, j] ≠ −A[i′, j′]
                        B[3(i − 1) + j, 3(i′ − 1) + j′] = 1
    return ⟨B, n⟩
```

(c) **[2]** What is the running time of your compiler?

Answer. On input a 3 by n matrix A of length $m = 3n$, the compiler scans the entire matrix A for each cell of A, resulting in time $3n \cdot 3n$, which is $\Theta(m^2)$.

(d) **[5]** Assume that your compiler from Section (b) is correct. Denote the length of the matrix A by m, and the length of $\langle B, k\rangle$ by n. Prove that if there is $c \in \mathbb{N}$ such that CLIQUE can be solved in time n^c, then there is $a \in \mathbb{N}$ such that 3-SAT can be solved in time $O(m^a)$. Prove that your

algorithm is correct and provide the value of a.

`Answer.` We define an algorithm that on input a 3-CNF φ (represented as a matrix A of size m) compiles A into $\langle G(\varphi), k\rangle$ (represented as a pair $\langle B, k\rangle$ of size $n = O(m^2)$) and then outputs the answer of the CLIQUE algorithm on $\langle G(\varphi), k\rangle$. The running time of the CLIQUE algorithm is $O(n^c) = O((m^2)^c) = O(m^{2c})$, which implies that the running time for our algorithm is $O(m^2) + O(m^{2c}) = O(m^{2c})$. Thus, $a = 2c$. To prove correctness, notice that if φ is satisfiable, then $G(\varphi)$ has a clique of size k, which implies that the CLIQUE algorithm will output 1. Conversely, if φ is not satisfiable, then $G(\varphi)$ does not have a clique of size k, which implies that the CLIQUE algorithm will output 0. Hence, our algorithm solves 3-SAT.