University of New Mexico
Department of Computer Science

# Final Examination

CS 362 Data Structures and Algorithms
Spring, 2008

| | |
|---|---|
| Name: | |
| Email: | |

---

- Print your name and email, *neatly* in the space provided above; print your name at the upper right corner of *every* page. Please print legibly.
- This is an *closed book* exam. You are permitted to use *only* two pages of "cheat sheets" that you have brought to the exam and a calculator. *Nothing else is permitted.*
- Do all the problems in this booklet. *Show your work!* You will not get partial credit if we cannot figure out how you arrived at your answer.
- Write your answers in the space provided for the corresponding problem. Let us know if you need more paper.
- Don't spend too much time on any single problem. The questions are weighted equally. If you get stuck, move on to something else and come back later.
- If any question is unclear, ask us for clarification.

---

| Question | Points | Score | Grader |
|---|---|---|---|
| 1 | 20 | | |
| 2 | 20 | | |
| 3 | 20 | | |
| 4 | 20 | | |
| 5 | 20 | | |
| Total | 100 | | |

1. **Short Answer**

   (a) Assume there is some function *foo* that has amortized cost of $O(\log n)$. If *foo* is called $n$ times in a row, what is the maximum cost of any single call to *foo* in this sequence of calls?

   *Solution: The maximum total cost of the $n$ calls is $O(n \log n)$. Thus the maximum cost of a single call is $O(n \log n)$*

   (b) Consider the following problem. You are given a connected graph $G$ with $n$ nodes and $m$ edges. You want to remove as many edges as possible and still keep the graph connected.

   - How many edges must remain in the graph in order to ensure that $G$ is connected? *Solution: $n - 1$ edges must remain in order to have a spanning tree of $G$*

   - Give an efficient algorithm to solve the problem *Solution: Find a spanning tree for the graph using your favorite algorithm (BFS, DFS, MST, etc). Then remove all edges that are not in this spanning tree*

(c) Assume you have a sorting algorithm that runs in linear time worst case (obviously this would have to be a non-comparison based sorting algorithm). If you use this new sorting algorithm, what will be the new asymptotic run time of Kruskal's on a graph with $n$ nodes and $m$ edges? Assume the amortized cost of all operations on the union-find data structure is $\log^* n$. Justify your answer. *Solution: The runtime will now be dominated by the costs of calls to the union-find data structure. Kruskal's calls Make-Set* $n$ *times, Find-Set* $2m$ *times and union* $n-1$ *times. The total run time will thus be dominated by calls to Find-Set and so will be* $O(m \log^* n)$.

(d) Assume you are given as input a directed graph without cycles (this is commonly called a directed acyclic graph or DAG for short), and a special source vertex $s$ that has no in edges. All other vertices in the graph have in-edges, and the edges on the graph have both positive and negative weights. How quickly can you find the *longest* path in this graph that starts at $s$? Justify your answer.

Hint: This is easy if you use an algorithm from class in the correct way.

*Solution: Flip the sign of all the edge weights and then use Bellman-Ford to find the shortest path from the source vertex in this new graph. Note that this graph will not have any negative cycles, since there are NO directed cycles. Thus, Bellman-Ford will find a shortest path in the negated edge-weight graph, which will be a longest path in the original graph. This takes* $O(|V||E|)$ *time.*

2. **Recurrences and Asymptotics**

(a) Give an asymptotic solution for the following recurrence: $T(n) = 2T(n/2) + n^2$ *Solution: $f(n) = n^2$ and $af(n/b) = 2(n/2)^2 = n^2/4$ so $f(n) \geq c * af(n/b)$ for $c > 1$. Thus the root node dominates and so $T(n) = \Theta(n^2)$*

(b) Solve the following recurrence $T(n) = 5T(n-1) - 6T(n-2)$. Give the solution in general form i.e. do not solve for the constants. *Solution: The annihilator is $\boldsymbol{L}^2 - 5\boldsymbol{L} + 6$ which factors to $(\boldsymbol{L} - 3)(\boldsymbol{L} - 2)$. This implies that the general solution is $T(n) = c_1(3)^n + c_2(2)^n$*

(c) Consider the recurrence $T(n) = 2^{\log T(n/2)} + T(n/2)$, where $T(1) = 1$

Use induction to prove that $T(n) = O(n)$. Don't forget to write down precisely what you are trying to prove and to include the base case, inductive hypothesis and inductive step.

*Solution: Goal: Show there exist constants $n_0$ and $c$ such that for $n \geq n_0$, $T(n) \leq cn$.*
*B.C.: $T(1) = 1 \leq cn$ for $c = 1$.*
*I.H.: For all $j < n$, $T(j) \leq cn$*
*I.S.*

$$\begin{aligned} T(n) &= 2^{\log T(n/2)} + T(n/2) \\ &\leq 2^{logcn/2} + c(n/2) \\ &= cn \end{aligned}$$

*Where the last step holds for $c = 1$, $n_0 = 1$, and the second step holds by the I.H.*

3. **PARTITION**

In the PARTITION problem, you are given a list of $n$ integers, $x_1, x_2, \ldots x_n$ all in the range 1 to $k$. You want to determine if there is some subset of the integers that sums to exactly $S/2$, where $S$ is the sum of all the integers in the list.

(a) Give an algorithm to solve this problem that has runtime polynomial in $n$ and $k$, and analyze your algorithm. Hint: Let $f(i, y) = 1$ if there is some subset of the first $i$ integers that sums exactly to $y$ and let it be 0 otherwise. *Solution: Let $f(i, y) = 1$ if there is some subset of the first $i$ integers that sums exactly to $y$ and let it be 0 otherwise. Then we can solve this using dynamic programming. The base case is $f(1, x_1) = 1$ and $f(1, 0) = 1$ and $f(1, y) = 0$ for all other values $y$. The recurrence is $f(i, y) = 1$ if and only if $f(i-1, y) = 1$ or $f(i-1, y-x_i) = 1$. At the end, we return true if $f(n, S/2) = 1$ and false otherwise. We can solve this recurrence by filling in a table top down from left to right. The runtime is $O(nS)$. Since $S$ is no more than $nk$, this runtime is $O(n^2k)$*

(b) The PARTITION problem is NP-Hard. Explain how this fact reconciles with your ability to devise an efficient algorithm above. Hint: This answer is no more than two sentences.
*Solution: The runtime of the above algorithm is exponential in the input size for large* $k$. *In particular, it takes only* $\log k$ *bits to represent an integer* $k$, *but the algorithm has runtime which is not polynomial in* $\log k$ *but is rather polynomial in* $k$.

4. **Minimum Spanning Trees**

Suppose you are given a connected graph $G = (V, E)$ with $n$ nodes and $m$ edges. Professor Bunyan claims that an edge $e = (u, v)$ is in a minimum spanning tree of $G$ if there is no path from $u$ to $v$ in $G$ that consists only of edges that have weight cheaper than $e$. Is Professor Bunyan correct? If so, prove it. If not, give a counter example.
Hint: Think about the safe-edge theorem.
*Solution: The statement is true. To prove it, consider the following cut. Let $S$ be the set of all nodes that can be reached from $u$ using edges that are cheaper than $e$. Consider the cut $(S, V - S)$. If we let $A = \{\}$, then this cut respects $A$ and moreover, by construction, $e$ is a light edge crossing this cut. Thus e is safe for $A$ and so e is in some MST.*

5. **Independent Set**

Recall that an Independent Set for a graph $G = (V, E)$ is a set of vertices $V'$ such that no pair of vertices in $V'$ have an edge between them in $G$. In the problem INDEPENDENT-SET, you are given a graph $G$, and an integer $k$ and must output TRUE if there is an independent set of size $k$ in $G$ and FALSE otherwise.

(a) Imagine you have a polynomial time algorithm that solves INDEPENDENT-SET. Show that you can use this algorithm to create a polynomial-time algorithm that takes as input a graph $G$ and integer $k$ and actually outputs an independent set of size $k$ if it exists.

Hint: You will need to make multiple calls to the algorithm that solves INDEPENDENT-SET, essentially one for each vertex in $G$.

*Solution: The key idea here is to note that all nodes that are not in the set $V'$ can be connected to all other nodes in $G$ and this will not effect the size of the independent set. If $k = 1$ then just output an arbitrary vertex in $G$ (any will do). Otherwise, let $n = |V|$. Iterate through each node $v$ in $G$ and do the following. Add edges between $v$ and all other nodes, check to see if there is an independent set of size $k$ in this new graph. If so, leave the new edges, if not, delete the new edges. Finally, return the set of all nodes with degree less than $n - 1$. This is an independent set of size at least $k$.*

Now you will describe an approximation algorithm for the independent set problem.

(b) If the minimum vertex cover of a graph $G$ is of size $x$, what is the size of the maximum independent set? Assume that $G$ has $n$ nodes and $m$ edges.

*Solution: n-x (as discussed in class)*

(c) Assume the smallest vertex cover of $G$ is less than or equal to $n/4$. Describe a good polynomial time approximation algorithm for finding the maximum independent set of $G$.

*Solution: The algorithm is simple: Use the 2-approximation for vertex cover that we described in class. Return all the nodes in $G$ that are* not *in this vertex cover as the independent set. Analysis: Let $x$ be the size of the minimum vertex cover. Then the approximation algorithm for vertex cover will find a vertex cover of size no more than $2x$. This means that the independent set returned will be of size at least $n - 2x$, whereas the largest independent set is of size $n - x$. The approximation ratio is thus $\frac{n-2x}{n-x}$. This ratio is minimized if $x$ is as large as possible, but $x \leq n/4$ by assumption. Thus, the approximation ratio is $2/3$. In other words, we can find an independent set of size $2/3$ of the largest independent set provided that the vertex cover of $G$ is no more than $n/4$. Moreover this algorithm takes polynomial time (in fact linear time in the number of edges and vertices).*

5. **Independent Set, continued.**