# Solutions to Final Exam

Algorithms and Their Complexity

February 10, 2000

It appears that solving the exam within the allotted time proved to be difficult for most people. (Some questions become easy only after you know the solution.) The final grades will take this into account and will be normalized upwards. Correct solutions to the final exam are sketched below. In addition, I comment on specific answers people handed in to some of the questions.

## 1 Finding the median

### 1.1 The question

Let $X = x_1, \ldots, x_n$ be a list of $n$ elements (where $n$ is odd). We have seen in class that *quicksort* sorts the list in expected $2n \ln n$ comparisons. Design a randomized algorithm that finds the median in $O(n)$ expected number of comparisons. Give both upper and lower bounds on the leading constant for the expected number of comparisons of your algorithm. In this exam it suffices that the bounds are reasonably close (e.g., within a factor of 2 of each other) rather than exact. (A remark that may be useful for the analysis of your algorithm: the integral of $1/x$ is $\ln x$ and the integral of $\ln x$ is $x \ln x - x$.)

### 1.2 Proposed solution

The following algorithm works in iterations, where after iteration $i$ we have a list $X_i \subset X$ in which the $k_i$th smallest element is the median of $X$. Initially $X_0 = X$ and $k_0 = (n+1)/2$. In iteration $i$ do the following.

1. Select a splitting element $x \in X_{i-1}$ at random.
2. Compare $x$ to every element of $X_{i-1}$. Let $k$ be the order of $x$ in $X_{i-1}$.
3. If $k = k_{i-1}$ then return $x$.
4. If $k > k_{i-1}$ then make $X_i$ the list of all elements smaller than $x$ in $X_{i-1}$, and $k_i = k_{i-1}$.
5. If $k < k_{i-1}$ then make $X_i$ the list of all elements larger than $x$ in $X_{i-1}$, and $k_i = k_{i-1} - k$.

To analyse the expected number of comparisons, we use the same approach that was used for quicksort in class. For notational convenience, we let $m$ denote $(n+1)/2$. For two elements whose true orders are $i$ and $j$ with $i < j$, the probability that they are compared by the algorithm is:

1. $2/(m-i+1)$ if $j \leq m$,

2. $2/(j-m+1)$ if $i \geq m$,

3. $2/(j-i+1)$ if $i < m < j$.

We now give an upper bound on the expected number of comparisons. The contribution of the first term above is

$$\sum_{i<m} \sum_{i<j\leq m} \frac{2}{m-i+1} = 2\sum_{i<m} \frac{(m-i)}{m-i+1} < 2(m-1) = n-1$$

By symmetry, the contribution of the second term above is also at most $n-1$. Hence it remains to compute the contribution of the third term. We shall change summations to integrals, and adjust the ranges of integration so that the value of the integral bounds from above the value of the respective sum.

$$\sum_{i<m} \sum_{j>m} \frac{2}{j-i+1} \leq 2\sum_{i<m} \int_m^n \frac{1}{j-i+1} dj = 2\sum_{i<m} (\ln(n-i+1) - \ln(m-i+1))$$

$$\leq 2\int_0^{m-1} \ln(n-i+1)di - 2\int_1^{m-1} \ln(m-i+1)di$$

$$\leq 2((n+1)\ln(n+1) - n - 1) - 2((m+1)\ln(m+1) - m - 1) - 2(m \ln m - m) + O(1)$$

$$\leq 2(n+1)\ln((n+1)/m) + O(1) \leq 2n \ln 2 + O(1)$$

Hence the expected number of comparisons is at most $2n(1+\ln 2) + O(1)$. Similar manipulations can bound the expected number from below. For the purpose of the exam it was sufficient to give rough bounds rather than exact bounds, allowing shortcuts in the above estimations.

**Remark:** One can choose a splitting element by first sorting a set of $k$ random elements. When $k$ is large enough, it is not hard to obtain an algorithm that has only $(1+o(1))3n/2$ expected number of comparisons.

### 1.3 Comments

Most people did give the algorithm presented above. Some also reached the stage where all that was left was to manipulate various sums. Manipulating the sums was in most cases sketchy at best (this was to be expected, given the time pressure). In the anlysis of some people, the expected running time of the algorithm was computed as a difference of two much larger values. These values were not computed exactly, but only approximated (e.g., sums approximated by integrals). The leading terms in these values canceled out, and then people were left without formally justified estimates on the order of magnitude of the difference.

I also accepted other less exact methods for bounding the expected running time (if explained reasonably well). For example – the expected number of iterations until the size of the list shrinks to 3/4 is at most 2 (because in each iteration the splitting element has probability at least 1/2 of being in the middle half), and hence the expected number of comparisons until the list shrinks to 3/4 is at most $2n$. (This line of argument is correct, based on properties of the expectation not discussed in class.) Thereafter, a linear upper bound on the running time is obtained as a geometric series.

# 2 Three dimensional matching

## 2.1 The question

We are given a set $S$ of $n$ elements and a list of $m$ *triples*, where each triple contains three elements from $S$. A *3-dimensional matching* is a collection of mutually disjoint triples. In class we remarked that finding a 3-dimensional matching of maximum cardinality is NP-hard.

1. Show that the problem can be solved in time $O(n^c 2^n)$, for some constant $c > 0$. (Namely, the running time can be bounded as exponential in $n$ rather than exponential in $m$.)
2. Give a (random) polynomial time algorithm for finding $\log n$ disjoint triples.

## 2.2 Proposed solution

The first part is a standard application of dynamic programming. There are only $2^n$ possible subsets of $S$. We shall consider all of them in order of increasing cardinality (in fact, it suffices to consider subsets whose cardinality is divisable by 3), and see which of them can be exactly covered by disjoint triples. Given a set $S'$, we check whether there is some triple $t$ whose elements are contained in $S'$ and whose removal gives a set $S''$ that has an exact cover. (This information is available to us as $|S''| < |S'|$.) Hence we make $m$ checks, and as we can assume that $m \leq \binom{n}{3}$, the running time is $O(n^c 2^n)$. The optimal solution can be obtained by backtracking from the largest set found to have an exact cover.

The second part can most easily be solved using the technique of *color coding* (from Lecture 3), though other approaches are also possible. To find $\log n$ disjoint triples (if they exist), color the elements of $S$ at random with $3 \log n$ colors. Now keep only triples containing three different colors, and merge triples that have the same set of colors. Use dynamic programming to find a 3-dimensional matching in the resulting smaller problem (now the running time is proportional to $2^{3 \log n} = n^3$ which is polynomial in the original problem size), and such a matching can easily be lifted back to a matching in the original problem. The algorithm succeeds if the random coloring colors each element of the $3 \log n$ elements used in a size $\log n$ matching by a different color, which happens with probability $(3 \log n)!/(3 \log n)^{3 \log n} \simeq e^{-3 \log n} \geq n^{-c}$ for some constant $c$. Repeating the randomized algorithm $\Omega(n^{-c})$ times the success probability can be made almost certain.

## 2.3 Comments

There is more than one correct way of using dynamic programming for this problem, which allows also for variations on how to use color coding. Almost everybody got the first part right. Almost nobody got the second part right. Some people correctly sensed that color coding may be relevant, but could not turn this intuition into an algorithm.

# 3 Approximating metric TSP

## 3.1 The question

Given a complete undirected graph with edge lengths obeying the triangle inequality ($w(i,k) \leq w(i,j) + w(j,k)$), the *metric TSP* problem asks for a minimum length cyclic tour visiting each vertex exactly once. We have seen in class an algorithm that approximates metric TSP within a ratio of 3/2. A natural greedy algorithm for metric TSP creates partial solutions and extends them as follows:

1. Sort the edges in order of increasing length.
2. For each edge in order, add it to the solution unless it contradicts edges already in the solution. Namely, the new edge is not allowed to be incident with a vertex that already has degree 2, or to close a cycle of length less than $n$.

Show that this algorithm approximates metric TSP within a factor of at most $H_n = \sum_{i=1}^{n} 1/i$.

## 3.2 Proposed solution

Let opt be the value of the optimal solution, and let $e_1, \ldots, e_n$ be the (sorted) list of edges chosen by the greedy algorithm. Similar to the analysis of the approximation ration of the greedy algorithm for set cover, it suffices to show that $w(e_i) \leq \text{opt}/(n-i+1)$. There is more than one way of showing this. Here is one solution that was handed in (and is somewhat simpler than the one I had in mind). The number of connected components in the graph containing the first $i-1$ edges is $n-i+1$. Each connected component has at least one vertex of degree less than 2. Take one such vertex from each connected component, and consider the minimum weight cycle going through each of these special vertices exactly once. It has weight at most opt, and by the triangle inequality, we may assume that it visits no other vertex (otherwise, shortcut the other vertex). Every individual edge does not contradict the first $i-1$ edges, hence we have a choice of $n-i+1$ edges of total weight at most opt. The least expensive of these edges costs at most $\text{opt}/(n-i+1)$, and the greedy algorithm would not pay more. (For the edge $e_n$ the above argument does not work as there is only one connected component. However, the triangle inequality implies that every edge in the graph is of weight at most opt, and in fact, at most opt/2.)

## 3.3 Comments

Most people realized that the $H_n = \sum_{i=1}^{n} 1/i$ ratio should come from paying at most $\text{opt}/(n-i+1)$ for the $i$th edge. However, they tried to argue that when the $i$th edge is chosen, $(n-i+1)$ edges of the optimal solution are still available. This is clearly false, as can easily be seen from toy examples (in particular, the last edge added by the greedy algorithm must be the unique edge that closes the cycle – there is no reason why the optimal solution should contain this edge). Moreover, this false attempt never uses the triangle inequalities, and hence cannot possibly work (see Lecture 11, section on TSP).

# 4 Very proper 3-CNF formulas

## 4.1 The question

A *proper 3-CNF formula* is the logical *and* of clauses, where each clause is the logical *or* of exactly 3 literals. For example:

$$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_4 \vee x_1 \vee x_3)$$

Finding a satisfying assignment for a proper 3-CNF formula is NP-hard. A *very proper 3-CNF formula* has the extra property that each variable appears exactly 3 times. Give a polynomial time algorithm for finding satisfying assignments for very proper 3-CNF formulas.

## 4.2 Proposed solution

Given a very proper 3-CNF formula $\phi$, construct the bipartite graph $G(V_1, V_2, E)$ in which $V_1$ is the set of clauses of $\phi$, $V_2$ is the set of variables, and edges connect between clauses and the variables that appear in them. Because $\phi$ is very proper, the bipartite graph $G$ is 3-regular. We have seen in Handout 7 (December 23, 1999) that every bipartite regular graph has a perfect matching. Let $M$ be a perfect matching in $G$. For each variable, set its value so that it satisfies the clause to which it is matched under $M$. This defines an assignment to the variables, and satisfies all the clauses.

## 4.3 Comments

Most people failed to notice the connection with matching. Several people thought that this question has something to do with the method of conditional expectations. (They were probably mislead in this direction by Homework 11, which also dealt with 3-SAT. Sorry, this was unintentional.) However, note that to apply the method of conditional expectations in this case one should first argue that a random assignment is expected to satisfy *all* clauses, meaning the the probability of satisfying every particular clause is 1. A random assignemt cannot possibly have this property (unless it fixes at least one variable per clause deterministically).

Several people reduced the problem to that of finding independent sets in graphs, and then tried to argue that the resulting graph is of bounded treewidth. However, this is not true. (If it were true, then it would also probably extend to cases where clauses contain at most three literals, rather than exactly three, and this problem is already NP-hard.)

# 5 Integer multiplication

## 5.1 The question

We have seen in class that two $n$-bit integers can be multiplied in time $O(n^{\log_2 3})$, rather than $O(n^2)$. Show a multiplication algorithm that works in $O(n(\log n)^c)$ time, for some constant $c > 0$. Here are some "hints" and assumptions to start you off (correct solutions not using these hints are also okay).

1. Observe that an $n$ bit number, whose bit representation is $a_{n-1} \dots a_1 a_0$ has value $\sum_{i=0}^{n-1} a_i x^i$ with $x = 2$.
2. You may assume that each of the two input numbers is at most $n/2$ bits long, and that $n$ is an exact power of 2.
3. The fraction of prime numbers is large enough to make it likely that for some $k < n$, the number $p = kn + 1$ is prime. Assume that you are given such a prime $p$, and also a generator $g$ for $Z_p^*$.

## 5.2 Proposed solution

The approach is via FFT. The numbers $a$ and $b$ and the solution $c = ab$ can formally be viewed as polynomials. For the coefficients of the polynomials, in suffices to work in $Z_p^*$, because the true final answers (the coefficients of $c$) are all smaller than $n$, and $n < p$. For FFT, a primitive $n$th root of unity is needed, and $g^k$ can serve for this purpose. Hence polynomial multiplication can be performed in this case while using scalar mutiplications that involve only $O(\log n)$ bit numbers, giving a $(n(\log n)^c)$ running time. Thereafter, the polynomial obtained for $c$ should be changed into an $n$-bit binary number by substituting $x = 2$. This requires another $O(n(\log n)^c)$ operations (e.g., using Horner's rule), because the coefficients of $c$ are $O(\log n)$ bits long, and because we substituted for $x$ a small number (moreover, multiplication by 2 is just a shift).

## 5.3 Comments

Most people realized the relevance of FFT. However, there was great variability on the level of discussions of issues such as the need for a root of unity, what can serve as the root of unity, how to get back from the polynomial to the actual number and how much does it cost.