Quiz 2 Solutions
Problem 1. Big Bully

The year is 2048. The world has been taken over by unruly orcs. Each orc is either a captain or a soldier. Roughly, an orc $X$ turns into a captain if any orc bigger than $X$ in $X$'s neighborhood is already the soldier of another captain, and $X$ is the biggest orc remaining in $X$'s neighborhood. On the other hand, $X$ ends up being a soldier if there is an orc in $X$'s neighborhood that is bigger than $X$ and has turned into a captain. Your goal is to find out which orcs are captains and which orcs are soldiers in a one-dimensional cave. More precisely:

Given: An array $A[1...n]$ of distinct integers (defining “orc size”), and an integer $k$ (defining “neighborhood size”).

Compute: $B[1...n]$, where each $B[i] \in \{\text{captain}, \text{soldier}\}$, such that, for every $i$,

- $B[i] = \text{captain}$ if, for every $j \in \{i-k, \ldots, i+k\}$, either $B[j] = \text{soldier}$ or $A[j] \leq A[i]$.
- $B[i] = \text{soldier}$ if there exists a $j \in \{i-k, \ldots, i+k\}$ such that $B[j] = \text{captain}$ and $A[j] > A[i]$.

(For this definition, assume that $A[i] = -\infty$ and $B[i] = \text{soldier}$ for $i < 1$ and $i > n$.)

Give an efficient algorithm for this problem. An ideal solution works well for all values of $k$ relative to $n$. For partial credit, solve the special cases when $k = O(1)$ and/or when $k = \Omega(n)$.

Solution: We give an $O(n)$ time algorithm for this problem. Initially every orc is a captain. The algorithm makes two passes over the array $A$. The first pass is left to right and the second pass is right to left. Informally, on the first pass we demote orcs that are dominated by their left neighbors and on the second pass we demote orcs that are dominated by their right neighbors.

We use a single auxiliary variable $p$ to keep a pointer to the orc that is currently dominating. Below is the pseudocode for our algorithm:

```python
1 for i ← 1 to n
2 B[i] ← captain
3 p ← 1
4 for i ← 1 to n
6 then B[i] ← soldier
7 else p ← i
8 p ← n
9 for i ← n to 1
11 then B[i] ← soldier
12 else p ← i
```

Let us argue that the output of our algorithm satisfies both constraints specified in the problem statement.
It is easy to see that the output of our algorithm can not contain two captains neighboring each other, since in the pass where we first observe the bigger captain we would demote the smaller.

It remains to argue that every soldier neighbors a bigger captain. Note that at the moment that an orc is demoted it is demoted by a bigger captain from its neighborhood. Let $i$ be a soldier. Assume that orc $i$ has been demoted by a bigger orc $j$. The only problem is that orc $j$ could have later been demoted by some other orc $k$. Note that this may only happen if $i$ was demoted by $j$ on the first pass, and $j$ was demoted by $k$ on the second pass. Thus orc $k$ is a captain in the neighborhood of $i$, and clearly we have $A[i] < A[j] < A[k]$.

There were many different solutions to this problem. The grade was assigned based on the running time (and correctness) of the algorithm.

Several solutions arise from the observation that the largest orc $i^*$ in the array must be a captain, with all other orcs in the neighborhood of $i^*$ as soldiers.

This observation suggests the following algorithm. First, start with all orcs unassigned, i.e., $B[i] = NULL$. Then, repeat the following loop until all orcs are assigned:

1. Scan all unassigned orcs in $A$, i.e., those with $B[i] = NULL$, and of those orcs find the orc $i^*$ with maximum size $A[i^*]$. If all orcs have been assigned, then terminate the loop.

2. Mark $i^*$ as a captain and mark any unassigned orcs $j \neq i^*$ in the interval $i^* - k \leq j \leq i^* + k$ as soldiers.

Finding the maximum in the first step takes $O(n)$. One can show that we can have at most $O(n/k)$ captains, giving us a runtime of $O(n^2/k)$. In the special case when $k = \Omega(n)$, this simple algorithm gives a linear-time solution.

For $k = \Theta(n/\lg n)$, the above algorithm can be improved by using some technique (i.e., sorting, building a heap, or a balanced search tree structure) to find the next maximum size unassigned orc more efficiently. If we store the orcs into a max-heap, then we can repeatedly extract the orcs in order of decreasing size. If we extract an unassigned orc $i$, then mark $i$ as a captain and all unassigned orcs in $i$'s neighborhood as soldiers. If we extract an assigned orc, then do nothing.

For each of the $O(n/k)$ orcs we mark as captains, we perform $O(k)$ work to check its neighborhood. For any element $i$ which is already a soldier when we extract it, we perform only $O(1)$ work in labeling. Thus, the cost of only labeling captains and orcs is only $O(n)$ time. Therefore, the runtime is dominated by the $O(n \lg n)$ cost to extract all elements from the heap in order of decreasing size.

Finally, if we make an additional assumption that the integer orc sizes are each at most $O(\lg n)$ bits, then we can use radix sort instead of a heap or other comparison-based sort and achieve an $O(n)$ algorithm.
Problem 2. The Gorehouse

Frustrated by his previous job experiences, Al Gorism has decided to make a career out of his best skill, developed by playing years of Tetris. He has taken up a job in a warehouse where his daily duty is to pack long rectangular boxes into an $L \times W$ warehouse, where $L$ is large, and $W$ is a small constant (think $W = 5$). Your goal is to design an algorithm to help him pack the boxes.

At the beginning of the day, Al starts with an empty $L \times W$ warehouse and a sequence of $n$ boxes lying on Conveyor Belt 1. He measures the boxes, and finds that the $i$th box $B_i$ has an integer length $\ell_i$ and an integer width $w_i$. Having measured all the boxes, his goal for the day is to pack them into the warehouse using three conveyor belts. Conveyor Belt 1 initially contains all boxes; it moves the boxes from east to west and deposits the boxes one at a time, in the order $B_1, B_2, \ldots, B_n$, onto Conveyor Belt 2. Conveyor Belt 2 can move the boxes north/south. Initially, the southern border of all the boxes are aligned with the southern wall of the warehouse. Thus, when box $B_i$ reaches Conveyor Belt 2, Al gets to choose how much north this box should be moved before it is transferred to Conveyor Belt 3. Conveyor Belt 3 then moves box $B_i$ west as far as possible (i.e., until its motion is obstructed by some box in the warehouse or by the western wall of the warehouse).

Give an efficient algorithm to compute how many units $u_i$ northward to move each box $B_i$, for $i \in \{1, 2, \ldots, n\}$, when it arrives on Conveyor Belt 2, so that all the boxes can be fit into the $L \times W$ warehouse. If there is no way to pack the boxes into the warehouse, then your algorithm should simply report this fact.

Solution:

Executive Summary: The solution uses dynamic programming to give an $O(n \cdot L^W)$ algorithm for this problem.
Details: We think of the warehouse as being tiled by $L \times W$ unit squares. The basic idea is to determine recursively if the first $i$ boxes can be fit into the profile consisting of the $L_1$ westmost tiles of the southmost row, the $L_2$ westmost tiles of the next row and so on up to $L_W$ westmost tiles of the top row. Let $P(L_1, \ldots, L_W, i)$ denote the indicator variable which is 1 if the first $i$ tiles can be packed into the profile $L_1, \ldots, L_W$, and 0 otherwise.

We note that $P(L_1, \ldots, L_W, i)$ satisfies the following conditions:

- If any of the $L_j$’s are negative, then $P(L_1, \ldots, L_W, i) = 0$.
- If all the $L_j$’s are non-negative and $i = 0$, then $P(L_1, \ldots, L_W, i) = 1$.
- For $i > 0$, $P(L_1, \ldots, L_W, i) = 1$ if and only if there exists some $u_i \in \{0, \ldots, W - w_i\}$ such that $P(L'_1, \ldots, L'_W, i - 1) = 1$ where $L'_j = L_j$ if $j \notin \{u_i + 1 \ldots u_i + w_i\}$ and $L'_j = \min_{k \in \{u_i + 1 \ldots u_i + w_i\}} \{L_k - \ell_i\}$ otherwise. (In English, $L'_1, \ldots, L'_W$ give the easternmost profile that would allow the $i$th box to be moved $u_i$ units northwards and still fit within the profile $L_1, \ldots, L_W$.

We can now compute the $P(\cdots)$ for all input arguments starting with $i = 1$ to $n$. In each iteration we enumerate the $L_1, \ldots, L_W$ in any order and compute $P(L_1, \ldots, L_W, i)$ using the recurrence above. Each $P(L_1, \ldots, L_W, i)$ takes $O(W^2)$ time to compute. There are $n \cdot L_W$ such values, thus taking a total of $O(W^2 \cdot n \cdot L_W)$ time to compute them all. We finally return $P(L, L, \ldots, L, n)$.

Notes on solutions: Alternately, one can also describe the solution in terms of the natural recursive algorithm with memoization based on the profile. The problem also has an exponential, $O(W^n)$, time algorithm. This solution received partial credit (it is slow but correct).

A fair number of solutions proposed a greedy algorithm minimizing the maximum of the $L_j$’s after inserting box $i$, for every $i$. This solution is obviously incorrect and received no credit.
Problem 3. Budget Shopping

You have just inherited a large toy store, Holidays ‘R’ Us. Throughout the day, customers come to you for advice of what to buy their relatives, given a hard budget of how much they can spend. To maximize your revenue, you always recommend the most expensive item that they can afford. The store has \( n \) items for sale, of distinct positive prices \( p_1 < p_2 < \cdots < p_n \), and can sell arbitrarily many copies of each item. Over the holiday season, your \( m \) customers come in arbitrary order with positive budgets \( b_1, b_2, \ldots, b_m \), respectively. You convince each customer \( i \) to purchase the predecessor product, i.e., the most expensive product \( p_j < b_i \).

Although you cannot predict the budgets of customers, you observe that products tend to be sold in “waves”. To model this feature, when you sell product \( p_j \) to customer \( i \) (the \( i \)th sale), you define the recentness \( r_i \) of this sale to be the number of customers since that product \( p_j \) was last purchased, i.e., \( r_i = i - i' \) where \( i' \) is the largest index < \( i \) such that customer \( i' \) purchased product \( p_j \). As a special case, if customer \( i \) is the first to have purchased the product \( p_j \), then the recentness of the \( i \)th sale is defined to be \( n \). You observe that, in practice, the recentness of most purchases is much smaller than \( n \).

To speed your customer advice, your goal is to build a data structure storing the set of prices that supports predecessor queries as fast as possible. The \( i \)th query \( b_i \) should run faster when the recentness \( r_i \) is small.

Solution: The optimal data structure for this problem supports the \( i \)th query \( b_i \) in \( O(\lg r_i) \) time. This bound is not easy to achieve—only one student did so—but there are many simpler structures whose performance adapts to smaller values of \( r_i \). We will describe some of the most common such partial solutions, and try to show the thought process for successive improvements.

The first solution that should come to your mind for solving the predecessor problem is a bound of \( O(\lg n) \), either by binary search in a sorted array or by searching in a balanced binary search tree. This bound is already a good start. Most students got at least this far.

A simple idea for adapting to \( r_i \) is to obtain a bound of \( \Theta(r_i) \). The easiest way to achieve this bound is to maintain the products in a linked list, ordered by the time they were last accessed, with most recent items at the front. (Initially the list can be ordered arbitrarily.) Each node of the list needs to store both the price \( p_j \) of the item and the price \( p_{j+1} \) of the next item, so that the query can tell directly whether this product will be purchased by customer \( b_i \) (when \( p_j < b_i \leq p_{j+1} \)). A query walks through the linked list, and when it finds the matching product, moves it to the front of the list, thereby keeping the desired list ordering. Many students found this solution.

Of course, \( r_i \) could be larger than \( \lg n \), so a query should really only look at the first \( O(\lg n) \) products of the list, after which it should fall back to the sorted array or binary search tree. This leads to a bound of \( O(\min\{r_i, \lg n\}) \). To achieve this bound, we add pointers between corresponding nodes in the linked list and in the sorted array or binary search tree. Then a query walks through the first \( \lg n \) nodes of the list; if it doesn’t find the answer there, it does a binary search or tree walk to find the predecessor, and then follows the pointer to the corresponding node in the list. In either case, the query then moves the list node to the front. Many students found this solution.
But if we are using only the first \( \lg n \) products of the list, surely we can store them in a more efficient way? Specifically, if we store the most recent \( \lg n \) products in a balanced binary search tree (e.g., red-black tree), then a query costs \( O(\lg \lg n) \) time if \( r_i \leq \lg n \) and \( O(\lg n) \) time otherwise. This is a big improvement if \( r_i \) is between \( \lg \lg n \) and \( \lg n \). To actually maintain this “cache” of \( \lg n \) products, we also need to store a linked list of the most recent \( \lg n \) products, ordered by when they were last accessed, and store pointers between corresponding nodes in the linked list and in the binary search tree. When a query finds the answer in the cache tree, we can move the corresponding list node to the front of the list. When a query has to go to the main tree of all products, we kick out the oldest item from the cache (the last product in the list), and replace it with the newly queried item (putting it at the front of the list). Several students found this solution.

If we push this caching idea a little further, we can get a bound of \( O(\lg n) \). The idea is to store caches of exponentially increasing size: \( 1, 2, 4, 8, \ldots, 2^{\lg n} \). The first cache stores the most recent product; the next cache stores the next two most recent products; etc. (The last cache might not be full.) Each product is in exactly one cache. Each cache consists of a balanced binary search tree, keyed on price, and a linked list, ordered most-recent-product-first. A query searches for the budget in the first cache, then the second cache, etc. If the product has recentness \( r_i \), then we will find it in the cache of size \( 2^k \) for some \( k \leq \lfloor \lg r_i \rfloor \). Then we move this element to the size-1 cache, kick out the oldest element from that cache and put it into the size-2 cache, kick out the oldest element from that cache and put it into the size-4 cache, etc., until the size-\( 2^k \) cache which has room for one element. Thus we perform one search, one delete, and one insert in each of the caches from 1 up to \( 2^k \). The total cost is thus \( O(\lg 1 + \lg 2 + \lg 4 + \cdots + \lg 2^k) = O(\sum_{i=0}^{\lg n} \lg 2^i) = O(\sum_{i=0}^{\lg n} i) = O(k^2) = O((\lg r_i)^2) \). One student found this solution.

Now, finally, we describe an optimal structure achieving a bound of \( O(\lg r_i) \). The trouble with exponentially increasing caches is that the running time ends up being an arithmetic sum. To make it a geometric sum, we build caches of doubly exponentially increasing size: \( 2, 4, 16, 256, \ldots, 2^{2^{\lfloor \lg \lg n \rfloor}} \). The first cache stores the \( 2^{2^0} \) most recently purchased products; the second cache stores the next \( 2^{2^1} \) most recently purchased products; etc. (The last cache might not be full.) We represent the size-\( 2^{2^i} \) cache in two different ways: a balanced search tree (e.g., a red-black tree), keyed on price, and a linked list, ordered most-recent-product-first. Every product appears in exactly one cache, in both the tree and the list forms. We store pointers between corresponding nodes in the tree and list forms. A query searches for the budget in the first cache, then the second cache, etc. If the product has recentness \( r_i \), then we will find it in the cache of size \( 2^{2^k} \) for some \( k \leq \lfloor \lg \lg r_i \rfloor \). Then we move this element to the size-2 cache, kick out the oldest element from that cache and put it into the size-4 cache, kick out the oldest element from that cache and put it into the size-16 cache, etc., until the size-\( 2^{2^k} \) cache which has room for one element. Thus we perform one search, one delete, and one insert in each of the caches from 2 up to \( 2^{2^k} \). The total cost is thus \( O(\lg 2 + \lg 4 + \lg 16 + \cdots + \lg 2^{2^k}) = O(\sum_{i=0}^{\lg n} \lg 2^{2^i}) = O(\sum_{i=0}^{\lg n} 2^i) = O(2^k) = O(2^{\lg \lg r_i}) = O(\lg r_i) \). One student found this solution.

In case you are interested, the \( O(\lg r_i) \) bound is called the working-set property in the literature, indicating that working with a small subset of elements is faster; it plays an important role in the study of optimal search trees.
Problem 4. Drowsy Shortest Paths

On the morning of November 27, 2006, you decide to attend the mandatory Lecture 21. Your task is complicated by the fact that lecture is early in the morning and you are still sleepy when trying to figure out the shortest path to get there. Unlike shortest paths when you are fully awake, this drowsy shortest paths problem is somewhat different. As before, every edge takes some fixed time to traverse. But when you are sleepy, every time you reach a vertex it takes some time to figure out which outgoing edge to take. Furthermore, the time it takes to resolve the confusion depends on how awake you are. So it might take less time to get to class if you start later (because you are more awake), but you might end up missing the beginning of lecture if you start too late! Your goal is to determine how late you can wake up and still get to lecture in time.

The map of MIT is described by a directed graph \( G = (V, E) \). Your starting point is a vertex \( s \) and your goal is to reach the 6.046 lecture at vertex \( z \). If you wake up \( t_0 \) units of time before start of lecture, then the time to visit a vertex \( v \) is \( \alpha \cdot t_0 \) minutes, where \( \alpha < 1 \) is a global nonnegative constant. (So the time to visit a vertex \( v \) is independent of the vertex \( v \), but it depends on how much sleep you get.) Each edge \( e \) takes some nonnegative real number \( w(e) \) minutes to traverse, independent of when you wake up. Give an efficient algorithm to compute the least amount of time \( t_0 \) before lecture that you could wake up and still get to lecture in time.

Solution:

Summary The best solution we know for this problem uses a modification of the Bellman-Ford algorithm and computes the optimal value of \( t_0 \) in \( O(E \cdot \min(V, \frac{1}{\alpha})) \) time. Several other solutions with slower run-times are possible. We describe the Bellman-Ford solution in some detail and briefly discuss some of the other common solutions.

Main idea The first thing to note is that for any path \( p \) from \( s \) to \( z \), the drowsy time \( t_0 \) to traverse the path satisfies

\[
t_0 = m \cdot \alpha \cdot t_0 + \sum_{e \in p} w(e),
\]

where \( \sum_{e \in p} w(e) \) is the regular (non-drowsy) time to traverse \( p \), and \( m \) is the number of edges in \( p \) (since \( \alpha \cdot t_0 \) time is added at each vertex other than \( z \)). This gives

\[
t_0 = \frac{\sum_{e \in p} w(e)}{1 - m\alpha}.
\]

The crucial thing to observe now is that if we consider all the paths from \( s \) to \( z \) with a fixed number of edges \( m \), then the best (smallest) value of \( t_0 \) among these \( m \)-edge paths is achieved with a path that has the shortest regular (non-drowsy) time among them. Thus if we let \( \delta^{(m)}(s, z) \) denote the shortest regular time among paths with exactly \( m \) edges, then the smallest drowsy time that can be achieved with a path of \( m \) edges is given by

\[
t_0^{(m)} = \frac{\delta^{(m)}(s, z)}{1 - m\alpha}.
\]
The best value of $t_0$ overall is then the smallest among all $t_0^{(m)}$ (note that we only need to consider values of $m$ smaller than $\frac{1}{\alpha}$, since by Eq. (1), $m > \frac{1}{\alpha}$ would imply it is not possible to get to lecture in time with any path of $m$ edges):

$$t^*_0 = \min_{1 \leq m \leq \min(|V| - 1, \frac{1}{\alpha})} \delta^{(m)}(s, z).$$

Thus all we need is an algorithm that can compute $\delta^{(m)}(s, z)$, the shortest regular time that can be achieved with a path of $m$ edges, for each $m$; we can then compute the optimal $t^*_0$ as above.

Bellman-Ford based algorithm  We can make a simple modification to the Bellman-Ford algorithm to compute the values $\delta^{(m)}(s, z)$, and keep track of the smallest $t_0^{(m)}$ as we vary $m$ from 1 to $\min(|V| - 1, \frac{1}{\alpha})$. The modification ensures that in the $m$th outer loop we separate the newly computed $d$ values from the ones computed in the previous iteration.

```plaintext
DROWSY-SHORTEST-PATHS(G, s, z)
1    for each vertex $v \in V$
2        do  $d[v] \leftarrow \infty$
3    $d[s] \leftarrow 0$
4    $t_0 \leftarrow d[z]$
5    for $m \leftarrow 1$ to $\min(|V| - 1, \frac{1}{\alpha})$
6        do  for each vertex $v \in V$
7                do  $d_{old}[v] \leftarrow d[v]$
8                $d[v] \leftarrow \infty$
9        for each edge $(u, v) \in E$
10            do  if $d[v] > d_{old}[u] + w(u, v)$
11                then  $d[v] \leftarrow d_{old}[u] + w(u, v)$
12            if $t_0 > \frac{d[z]}{1 - \alpha \cdot m}$
13                then  $t_0 \leftarrow \frac{d[z]}{1 - \alpha \cdot m}$
14    return $t_0$
```

Correctness  From the analysis of Bellman-Ford it follows that for all $m, v$, at the end of the $m$th iteration, the quantity $d[v]$ contains the shortest regular time that can be achieved with a path from $s$ to $v$ using exactly $m$ edges. Therefore, at the end of the $m$th iteration, the quantity $d[z]$ is equal to $\delta^{(m)}(s, z)$ as discussed above. Correctness thus follows from the above discussion.

Run-time analysis  The running time of the above algorithm is $O(E \cdot \min(V, \frac{1}{\alpha}))$; the space it takes is $O(V)$. 
Other algorithms Several students came up with the above Bellman-Ford based algorithm. Some other slower algorithms that were also found among the solutions are as follows:

Repeated runs of Dijkstra
There are at least two (correct) solutions that use Dijkstra; both involve \( O(\min(V, \frac{1}{\alpha})) \) runs of Dijkstra and therefore take \( O((V \lg V + E) \cdot \min(V, \frac{1}{\alpha})) \) time. One solution simply uses one run of Dijkstra for each \( m \) to compute \( \tilde{\delta}^{(m)}(s, z) \); this can be done by keeping track of the lengths of paths found from \( s \) and not relaxing any edges going out of a vertex to which a path of length \( m \) has been found. The other solution starts by running Dijkstra using the original weights \( w(e) \) to find the shortest (regular-time) path overall, and computing the drowsy time \( t_0 \) corresponding to this path (using Eq. (1)); the next run of Dijkstra then uses the new weights \( w(e) + \alpha \cdot t_0 \), finds a corresponding shortest path, and updates \( t_0 \) based on the new path; this process is repeated until no improvement is obtained (it can be shown that the number of edges in the path found goes down on each run of Dijkstra, and therefore in this case too, at most \( O(\min(V, \frac{1}{\alpha})) \) runs of Dijkstra are required).

Dynamic programming
The dynamic programming method used to find all-pairs shortest paths (CLRS, Section 25.1) can be used to give another (albeit slower) algorithm for this problem. The matrices \( L^{(m)} \) computed in that method contain the weights of shortest paths of at most \( m \) edges. It is easy to show that if \( \tilde{\delta}^{(m)}(s, z) \) denotes the weight of a shortest (regular-time) path from \( s \) to \( z \) among paths with at most \( m \) edges, we can still compute \( t_0^* \) as

\[
  t_0^* = \min_{1 \leq m \leq \min(|V| - 1, \frac{1}{\alpha})} \tilde{\delta}^{(m)}(s, z) - m\alpha.
\]

Since we can obtain the required \( \tilde{\delta}^{(m)}(s, z) \) by computing the matrices \( L^{(m)} \) for each \( m \) ranging from 1 to \( \min(|V| - 1, \frac{1}{\alpha}) \), from the results of Section 25.1 in CLRS, this gives us an \( O(V^3 \cdot \min(V, \frac{1}{\alpha})) \) time algorithm for finding \( t_0^* \). In fact, this algorithm is redundant in the sense that we really only need to compute one row of each of these matrices, namely the row corresponding to the source vertex \( s \). This observation gives a quick improvement with this approach, leading to an \( O(V^2 \cdot \min(V, \frac{1}{\alpha})) \) time algorithm.

Brute force
Exhaustively evaluate all paths from \( s \) to \( z \). Exponential time, but correct.

Search for \( t_0^* \) using doubling trick with Dijkstra
A completely different approach can be used if one assumes the weights \( w(e) \) and \( t_0 \) are all integers. In this case, one can search for the optimal value \( t_0^* \) using a doubling trick as follows. Start with \( t_0 = 1 \); run Dijkstra with weights \( w(e) + \alpha \cdot t_0 \). If the shortest path returned has (drowsy) traversal time smaller than \( t_0 \), double \( t_0 \) and repeat. If the shortest path returned has (drowsy) traversal time greater than \( t_0 \), use binary search between \( t_0/2 \) and \( t_0 \) to find the right value. If the traversal time is equal to \( t_0 \), return that value of \( t_0 \). This algorithm takes \( O((\lg t_0^*)(V \lg V + E)) \) time. The integer assumption for this algorithm is a strong assumption (although it can be shown to also give some useful approximation in the real case).