COMP/MATH 3804 – Design and Analysis of Algorithms I

Fall 2015

FINAL EXAM SOLUTIONS

Question 1 (12%). Modify Euclid's algorithm as follows.

```
function Newclid(a,b)
  if a<b: swap a and b
  if b=0: return a
  return Newclid(a-b,b)
```

Does `Newclid(a,b)` still compute $\gcd(a,b)$? Briefly justify your answer.
Show that `Newclid(a,b)` is exponentially slower than `Euclid(a,b)` on certain inputs.

Solution. `Newclid(a,b)` computes $\gcd(a,b)$: the proof of Euclid's rule is based on the fact that $\gcd(a,b) = \gcd(a-b,b)$, provided that $a \geq b$. This is exactly what `Newclid` does. Moreover, `Newclid` terminates because the value of its input decreases at each recursive call.

If $b = 1$, then `Newclid(a,b)` performs $\Theta(a)$ recursive steps, at least half of which take time $\Theta(\log a)$, while `Euclid(a,b)` performs just one $O((\log a)^2)$ step. Therefore `Newclid(a,1)` is an exponential-time algorithm, while `Euclid(a,1)` runs in polynomial time.

Question 2 (12%). Suppose you have to choose among three algorithms to solve a problem:

- Algorithm A solves an instance of size n by recursively solving eight instances of size $n/2$, and then combining their solutions in time $O(n^3)$.
- Algorithm B solves an instance of size n by recursively solving twenty instances of size $n/3$, and then combining their solutions in time $O(n^2)$.
- Algorithm C solves an instance of size n by recursively solving two instances of size $2n$, and then combining their solutions in time $O(n)$.

Which one is preferable, and why?

Solution. Algorithm C does not even terminate, because it recursively calls itself on instances of increasing size. Hence it does not have a running time (i.e., its running time is "infinite"). By the Master theorem, Algorithm A's running time is $O(n^3 \log n)$, and Algorithm B's running time is $O(n^{\log_3 20})$. Since $\log_3 20 < 3$, Algorithm B is preferable.

Question 3 (12%). You are given an unsorted array of n distinct integers, along with m *queries*. Each query is an integer that you have to search in the array, reporting "found" or "not found". Queries have to be processed in the order they are given. Assuming that $m = \lfloor \sqrt{n} \rfloor$, would you answer each query via a linear search of the unsorted array, or would you rather preliminarily sort the array to speed up your searches? What if $m = \lfloor\sqrt{\log n}\rfloor$ instead? Briefly justify your answers.

Solution. Doing m linear searches requires $\Theta(mn)$ time. On the other hand, sorting the array takes $\Theta(n \log n)$ time, and doing m binary searches on the sorted array requires $\Theta(m \log n)$ time. Hence sorting is convenient (or equivalent to not sorting) if and only if $m = \Omega(\log n)$. Therefore, if $m = \lfloor\sqrt{n}\rfloor = \Omega(\log n)$, then sorting is convenient. If $m = \lfloor\sqrt{\log n}\rfloor \neq \Omega(\log n)$, then sorting is not convenient.

Question 4 (15%). In the fast Fourier transform algorithm, why do we choose ω to be a *primitive* n-th root of unity (as opposed to any other n-th root of unity)?

Solution. Because, to compute the value representation of the input polynomial, we want to evaluate it at n distinct points. If ω is not a primitive n-th root of unity, then its powers do not generate all of the n-th roots of unity, but only a subset of them. This causes some repetitions among the first n powers of ω, which means that the polynomial gets evaluated at fewer than n points.

Question 5 (12%). Recall that, in a depth-first search of a directed graph, the vertex that receives the *highest* `post` number must lie in a *source* strongly connected component. Is it true that the vertex that receives the *lowest* `post` number lies in a *sink* strongly connected component? Give a short proof or provide a counterexample.

Solution. No. This graph is a counterexample: $(\{a, b, c\}, \{(a, b), (a, c), (b, a)\})$. If the depth-first search picks vertices in alphabetical order, then `post(a)` $= 6$, `post(b)` $= 3$, and `post(c)` $= 5$. Hence b has the lowest `post` number, but it does not lie in a sink strongly connected component.

Question 6 (10%). Recall that, in a depth-first search of a directed graph, for an edge (u, v) we distinguish three cases:

- if `pre(u)<pre(v)<post(v)<post(u)`, then (u, v) is a *forward* edge;
- if `pre(v)<pre(u)<post(u)<post(v)`, then (u, v) is a *back* edge;
- if `pre(v)<post(v)<pre(u)<post(u)`, then (u, v) is a *cross* edge.

Why is the ordering `pre(u)<post(u)<pre(v)<post(v)` not possible for edge (u, v)?

Solution. The fact that `pre(u)` $<$ `pre(v)` means that u is discovered before v. But `post(u)` $<$ `pre(v)` implies that u is left for the last time before v is ever visited. But this contradicts the depth-first search algorithm, which visits every unvisited neighbor of u (hence also v) before abandoning u for the last time.

Question 7 (10%). In Dijkstra's algorithm, why do we use a priority queue, as opposed to a regular first-in-first-out queue? Give an example of a graph with positive lengths on edges on which Dijkstra's algorithm fails to compute shortest paths if a first-in-first-out queue is used.

Solution. Consider the complete undirected graph on $\{s, a, b\}$, with lengths $\ell(s, a) = 3$ and $\ell(s, b) = \ell(a, b) = 1$. Suppose the starting vertex is s, and a is inserted in the queue before b, with `dist(a)` $= 3$ and `dist(b)` $= 1$. If the queue is first-in-first-out, then a is ejected first, leaving its `dist` value permanently at 3. This is incorrect, because the true distance between s and a is 2.

Question 8 (12%). Recall that Kruskal's algorithm computes a *minimum*-weight spanning tree of a graph by sorting its edges by *increasing* weight. Is it true that the same algorithm would return a *maximum*-weight spanning tree if it sorted the edges by *decreasing* weight? Give a short proof or provide a counterexample.

Solution. It is true. Too see why, consider a weighted graph G, and negate the weights of all its edges, obtaining a new weighted graph G'. Run the standard Kruskal's algorithm on G' and the modified Kruskal's algorithm on G. By construction, the two algorithms pick the same edges, and therefore return the same tree T. By the correctness of the standard Kruskal's algorithm, T has minimum weight in G', which means that it has maximum weight in G.

Question 9 (10%). Explain, in general terms, the main differences between the divide-and-conquer technique and dynamic programming.

Solution. Divide and conquer is an inherently recursive technique: it divides the input problem into smaller sub-problems, and then it recurses on each sub-problem, combining their solutions afterwards. Nothing is done to prevent the same sup-problem from being solved several times.

Dynamic programming is not an inherently recursive technique: in its basic form, it builds several sub-problems and solves them bottom-up, storing their solutions in a table for future reference. This prevents the same sub-problem from being solved multiple times.

The efficiency of a divide-and-conquer algorithm typically comes from the fact that each problem is broken down into substantially smaller sub-problems, which prevents the recursion tree from getting too large. On the other hand, the efficiency of dynamic programming typically lies in the fact that the number of sub-problems and their interdependencies is relatively small (e.g., polynomial), and no sub-problem is solved more than once.

Question 10 (15%). Recall that there exists a dynamic-programming algorithm for the Knapsack problem that has a running time of $O(nW)$, where n is the number of items, and W is the knapsack's capacity. However, the Knapsack problem is also known to be NP-hard. Therefore, given any problem in NP, we can reduce it to Knapsack in polynomial time, and then solve the Knapsack instance with the dynamic-programming algorithm. This would give us a polynomial-time algorithm for every problem in NP, and it would therefore imply that P=NP. What is the fallacy in this argument?

Solution. The mistake is assuming that the dynamic programming algorithm runs in polynomial time. The input of Knapsack is an array of n pairs of integers, plus an additional integer W. Consider the class of instances in which W is the largest of such integers and n is a constant. The size of the input, restricted to these instances, is $O(\log W)$, and the running time of the algorithm is $O(W)$, which is exponentially larger.

Question 11 (15%). Recall that, in the Set Cover problem, we are given a universe set U of size n, along with m subsets $S_1, S_2, \cdots, S_m$ of U. The goal is to select the smallest number of these subsets, such that their union is all of U. Model Set Cover as an integer linear programming problem (i.e., a linear programming problem in which all variables are integers).

Solution. Let $U = \{1, \cdots, n\}$. The linear programming problem has m integer variables $x_1, \cdots, x_m$, where x_i represents the subset S_i: $x_i = 1$ if S_i is selected, and $x_i = 0$ otherwise. The following linear programming specification is a direct translation of Set Cover:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{m} x_i \\ \text{subject to} & \sum_{i:1\in S_i} x_i \geq 1 \\ & \sum_{i:2\in S_i} x_i \geq 1 \\ & \qquad \vdots \\ & \sum_{i:n\in S_i} x_i \geq 1 \\ & 0 \leq x_1 \leq 1 \\ & 0 \leq x_2 \leq 1 \\ & \qquad \vdots \\ & 0 \leq x_m \leq 1 \end{array}$$

The objective function counts how many subsets have been selected, and the goal is to minimize it. The first n constraints express the fact that each element of U has to appear in at least one of the selected subsets. The last m inequalities set the bounds for the variables. This, paired with the assumption that the variables are integers, forces their values to be either 0 or 1.

Question 12 (12%). If we could solve an NP-complete problem in polynomial time, would all other problems in NP necessarily be solvable in polynomial time? Briefly justify your answer.
If we could solve an NP-complete problem in time $O(n^{2015})$, would all other problems in NP necessarily be solvable in time $O(n^{2015})$? Briefly justify your answer.

Solution. First question: yes. Suppose that the NP-complete problem A is solvable in polynomial time, and let an NP problem B be given. Since A is NP-complete, any instance of B can be translated into an "equivalent" instance of A in polynomial time. Then the instance of A can be solved in polynomial time, by assumption. Composing these two algorithms yields a polynomial-time algorithm for B.

Second question: no. Even if A is solvable in time $O(n^{2015})$, we still do not know anything about the running time of the reduction from B to A, except that it is polynomial. If such a polynomial is not $O(n^{2015})$, then composing the reduction and the algorithm for A does not yield an $O(n^{2015})$ algorithm. Hence, without making further assumptions, we cannot deduce that B can be solved in time $O(n^{2015})$.

Question 13 (18%). Let MAX SAT be the maximization problem of finding an assignment to the variables of a given Boolean formula (expressed in conjunctive normal form) such that the highest number of clauses evaluate to true. Consider the following approximation algorithm for MAX SAT:

```
function ApproxSAT(F)
  input: a Boolean formula F in conjunctive normal form
  output: an assignment to the variables of F

  A = assignment where every variable of F is true
  B = assignment where every variable of F is false
  if A satisfies more clauses of F than B does: return A
  else: return B
```

Prove that the approximation ratio of `ApproxSAT` is *exactly* 2.

Solution. Recall that the approximation ratio of an algorithm $\mathcal{A}$ for a *maximization* problem is defined as $\sup_I \frac{\mathrm{OPT}(I)}{\mathcal{A}(I)}$, where I is an instance of the problem.

Referring to the `ApproxSAT` algorithm, observe that, if the assignment A does not make a clause true, then all the literals in the clause are negative. But in this case, B makes the clause true. Hence, each clause is satisfied by either A or B, implying that one of the two assignments satisfies at least half the clauses of the formula F. In particular, it satisfies at least half the maximum number of clauses of F that are simultaneously satisfiable, implying that `ApproxSAT` is at least a 2-approximation algorithm.

To see why the approximation ratio of `ApproxSAT` is exactly 2, consider the Boolean formula $(x) \wedge (\bar{y})$, which has two clauses, both of which are satisfied by the assignment $x =$ true and $y =$ false. The optimum here is 2, but A and B satisfy only one clause each.

Problem 1. Let an array of integers $A = [a_1, a_2, \cdots, a_n]$ be given. Suppose that there exists an (unknown) index k such that the subarray $[a_1, \cdots, a_k]$ is sorted in *strictly increasing* order, and the subarray $[a_k, \cdots, a_n]$ is sorted in *strictly decreasing* order (i.e., if $1 \leq i < j \leq k$ then $a_i < a_j$, and if $k \leq i < j \leq n$ then $a_i > a_j$). Your goal is to determine k.

> **Part 1 (15%).** Prove that any comparison-based algorithm that solves this problem has a running time of $\Omega(\log n)$. [*Hint: count the leaves of the comparison tree.*]
>
> **Part 2 (25%).** Describe in English (no pseudo-code) an optimum algorithm to solve this problem, and analyze its running time. [*Hint: use divide and conquer.*]

Solution.

Part 1. Any such algorithm must return an index between 1 and n. Since a_k can be located anywhere in the array, all n possibilities can actually occur. Therefore, the choice tree of any comparison-based algorithm that solves this problem must have at least n leaves, implying that its height must be $\Omega(\log n)$. The height of the tree is the number of comparisons that the algorithm performs in the worst case, which in turn is a lower bound to the total number of operations, and hence to the running time. It follows that the running time of such an algorithm must be $\Omega(\log n)$.

Part 2. We modify the binary search algorithm: intuitively, we find the maximum element a_k by bisecting the array, looking at the "gradient" at the midpoint, and going "uphill".

If $n \leq 2$, we trivially find k in constant time. If $n \geq 3$, we set $m = \lceil n/2 \rceil$, and we inspect a_{m-1}, a_m, and a_{m+1}. If $a_{m-1} < a_m > a_{m+1}$, return m; if $a_{m-1} > a_m > a_{m+1}$, recurse on $[a_1, \cdots, a_{m-1}]$; if $a_{m-1} < a_m < a_{m+1}$, recurse on $[a_{m+1}, \cdots, a_n]$.

The algorithm is correct because, by the properties of A, if $a_{m-1} < a_m > a_{m+1}$, then necessarily $m = k$. Also, if a_{m-1}, a_m, and a_{m+1} are in descending order, then $k < m$ (again by the properties of

A); similarly, if they are in ascending order, then $k > m$. The other possibility, $a_{m-1} > a_m < a_{m+1}$, clearly cannot occur in A.

The running time satisfies the relation $T(n) = T(\lfloor n/2 \rfloor) + O(1)$, because at most two comparisons are performed at each step (hence the term $O(1)$), and then the algorithm either terminates or is executed again on one half of the array. By the Master theorem, $T(n) = O(\log n)$, which is optimal due to Part 1.

Problem 2. Let an array of (positive and negative) integers $A = [a_1, a_2, \cdots, a_n]$ be given. Your goal is to find the contiguous subsequence of A with maximum sum (i.e., two indices $1 \le i \le j \le n$ that maximize the sum $a_i + a_{i+1} + \cdots + a_{j-1} + a_j$).

Choose only one of the following options.

> **Option 1 (5%).** Describe in English (no pseudo-code) an $O(n^3)$ algorithm to solve this problem.
>
> **Option 2 (15%).** Describe in English (no pseudo-code) an $O(n^2)$ algorithm to solve this problem.
>
> **Option 3 (35%).** Describe in English (no pseudo-code) an $O(n)$ algorithm to solve the problem. [*Hint: use dynamic programming, similarly to the longest increasing subsequence problem.*]

Solution (Option 3). We identify n sub-problems $P(1), \cdots, P(n)$, where $P(k)$ consists in finding the contiguous subsequence S_k of A with maximum sum that *ends* in a_k. Specifically, $P(k) = (s_k, f_k)$, where s_k is the sum of the elements of S_k, and f_k is the index of the first element of S_k.

We can immediately compute $P(1) = (a_1, 1)$. Then, for all indices $k > 1$ in increasing order, we can compute $P(k)$ as follows: look up $P(k-1) = (s_{k-1}, f_{k-1})$; if $s_{k-1} \le 0$, then $P(k) = (a_k, k)$; otherwise, $P(k) = (s_{k-1} + a_k, f_{k-1})$. While we do this, we also store the index k^* corresponding to the largest sum s_{k^*} found so far. When all the subproblems have been solved, we know the index k^* where the maximum-sum contiguous subsequence ends, and we also know its starting index f_{k^*}. Hence we return the subsequence $[a_{f_{k^*}}, \cdots, a_{k^*}]$, or simply the two indices f_{k^*} and k^*.

The correctness of the algorithm can be easily proven by induction on k. Indeed, S_k is either the single entry $[a_k]$, or it is formed by appending a_k to S_{k-1}. The second option is better if and only if the sum of the elements of S_{k-1} is negative. According to this analysis, and assuming by inductive hypothesis that $P(k-1)$ has been correctly computed, the above algorithm is correct.

The running time is clearly linear, because there are n sub-problems, each of which is solved in constant time. The total amount of work needed to store and update k^* is also linear.

Problem 3. Let $A = [a_1, a_2, \cdots, a_n]$ be an array of distinct integers, and let k be a given integer. Your goal is to find two distinct elements of A whose sum is exactly k, or report that no such elements exist.

Choose only one of the following options.

> **Option 1 (5%).** Describe in English (no pseudo-code) an $O(n^2)$ algorithm to solve this problem.
>
> **Option 2 (30%).** Describe in English (no pseudo-code) an $O(n \log n)$ algorithm to solve this problem. [*Hint: a well-known $O(n \log n)$ algorithm may be a useful tool.*]

Solution (Option 2). Preliminarily sort the array A in increasing order, in $O(n \log n)$ time. Then keep two indices i and j, initially pointing at the first and last element of the sorted array A, respectively. If $a_i + a_j = k$, return a_i and a_j; if $a_i + a_j < k$, increase i by 1; if $a_i + a_j > k$, decrease j by 1. Repeat this step until two suitable elements are found, or until $i = j$. In the latter case, report that no two elements of A have sum k.

Let us prove that the algorithm is correct. If no two elements of A have sum k, then obviously the algorithm will never find two such elements. Hence the difference between i and j will keep decreasing by 1 at every step, until $i = j$. When this happens, the algorithm correctly reports that the two desired elements do not exist. Suppose now that they do exist: $a_x + a_y = k$, with $x < y$. Suppose by contradiction that the algorithm never evaluates the sum $a_x + a_y$, and erroneously reports that no such elements exist. This only happens when i and j become equal, which means that i and j, collectively, have scanned the whole array. Therefore, at some point, either i has been equal to x, or j has been equal to y (because $i \le j$ and $x < y$). Suppose without loss of generality that $i = x$ occurs before $j = y$ (the other case is symmetric, and note that they cannot occur both at the same time, because only one index changes at each step). At that point, since j has not been equal to y yet, it must be greater than y. But the array is sorted in increasing order, so the sum $a_i + a_j$ must be greater than $a_x + a_y = k$ (because $a_i = a_x$ and $a_j > a_y$). Hence the algorithm decreases j by 1. This is repeated until $j = y$. Meanwhile i has not changed, implying that now $a_i = a_x$ and $a_j = a_y$. So the sum $a_i + a_j$ evaluates to k, and the two correct elements are returned, contradicting the assumption that they are never found.

The running time of the algorithm is $O(n \log n)$ for the initial sorting phase, and $O(n)$ for the final scan (because $j - i$ decreases by 1 at each step, and is initially $n - 1$), yielding an overall running time of $O(n \log n)$.

Problem 4. We say that an undirected graph $G = (V, E)$ is *c-colorable* if its vertices can be colored using a palette of c colors, in such a way that adjacent vertices get different colors. Formally, G is c-colorable if there exists a function $f : V \to \{1, 2, \cdots, c\}$, called *c-coloring*, such that, for every $\{u, v\} \in E$, $f(u) \neq f(v)$.

> **Part 1 (20%).** Describe in English (no pseudo-code) a linear-time algorithm that, given a connected graph $G = (V, E)$ represented as an adjacency list, greedily colors its vertices in such a way that adjacent vertices get different colors. Moreover, if G is 2-colorable, your algorithm should not use more than two colors. Explain in intuitive terms why your algorithm is correct (i.e., why adjacent vertices get different colors, and why it returns a 2-coloring if the graph is 2-colorable), and show that it runs in time $O(|V| + |E|)$. [*Hint: do a breadth-first search.*]
>
> **Part 2 (15%).** Exhibit infinitely many 3-colorable graphs for which the algorithm you gave in Part 1 does <u>not</u> return a 3-coloring.
>
> **Part 3 (30%).** Prove that deciding if a given graph is 3-colorable is NP-complete. [*Hint: give a reduction from 3-SAT. Transform a Boolean formula φ into a graph G such that φ is satisfiable if and only if G is 3-colorable. Start by constructing a triangle, and label its vertices "True", "False", and "None". This is the "palette triangle". Then represent a variable of φ as a pair of adjacent vertices, and represent an OR gate as a triangle. Appropriately connect the palette triangle with the variable vertices and the OR gates in such a way that 3-coloring the resulting graph is equivalent to finding a satisfying assignment to φ.*]

Solution.

Part 1. The algorithm is based on a breadth-first traversal, using a first-in-first-out queue. Intuitively, each time a vertex is visited, it is assigned a color that has not been assigned to any of its neighbors, trying to assign color 1 whenever possible (recall that colors are encoded as integers). The details are as follows.

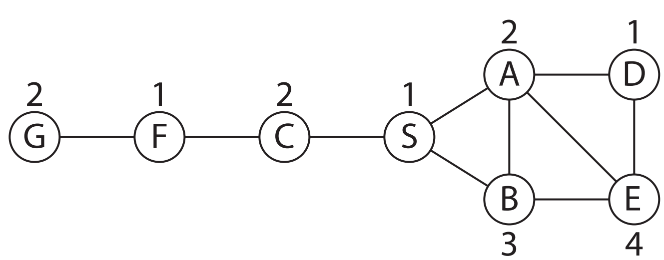
First, mark every vertex of G as unvisited and uncolored. Initialize the queue by inserting any vertex, and mark the vertex as visited. At each step, eject the first vertex v from the queue, and scan all its neighbors. When an unvisited neighbor of v is found, insert it into the queue and mark it as visited. Meanwhile, remember the largest color encountered so far among the neighbors of v, and also remember if color 1 has been encountered (ignoring the uncolored neighbors). At the end of the scan, if color 1 has not been found, then assign color 1 to v. If color 1 has been found, then assign v the largest color encountered, incremented by 1. Repeat until the queue is empty.

This algorithm visits every vertex of G, due to the properties of breadth-first traversals, and because G is connected. Each time a vertex is processed, it is assigned a color that has not been assigned to any of its neighbors, yet. Therefore, eventually all vertices are colored, and no two neighbors have the same color.

Suppose that G is 2-colorable, fix a 2-coloring f, and fix a starting vertex s. Without loss of generality, we may assume that $f(s) = 1$ (if not, invert all colors). It is easy to prove by induction that $f(v) = 1$ if and only if the distance between the vertices v and s is even. Again by induction, we can prove that our greedy algorithm, starting from s, computes exactly f, and therefore uses at most two colors. This is true for s, because none of the neighbors of s has a color when s is processed, and so by default s gets color 1. Now assume that our claim is true for all vertices at distance d from s, and let v be a vertex at distance $d+1$. By the properties of the breadth-first search, when v is ejected from the queue, all the vertices at distance d from s have already been traversed, and colored. By the inductive hypothesis, all neighbors of v are either uncolored or have the same color, either 1 or 2. The greedy algorithm then assigns v the other color, which once again agrees with f. This holds for all vertices at distance $d + 1$ from s, which concludes our proof by induction.

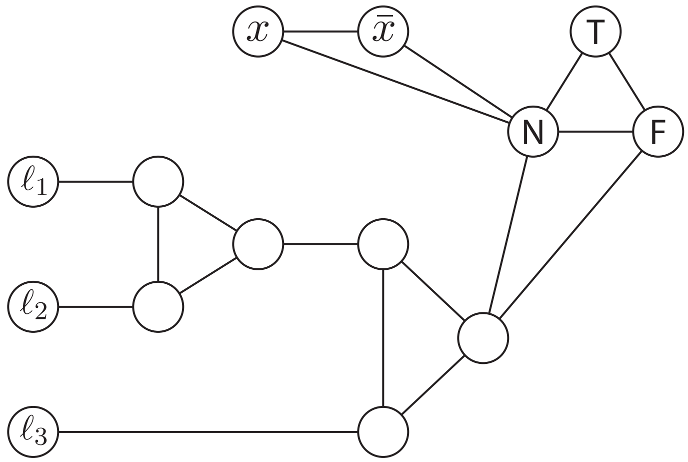
The running time of the algorithm is the same as the breadth-first search. Indeed, each time a neighbor of a vertex is examined, only a constant number of operations are performed (e.g., insert it into the queue, look up its color, check if it is 1, compare it with the maximum color seen so far, etc.). Similarly, the number of operations performed on each vertex (other than the scan of its neighbors) is constant (e.g., eject it from the queue, decide whether its color should be 1, etc.). Hence the running time is $O(|V| + |E|)$.

Part 2. The graph below is an example. Suppose that the greedy algorithm starts from vertex S, and then processes vertices in alphabetical order. The numbers are the colors that are assigned to each vertex. Despite the fact that the graph is 3-colorable, the greedy algorithm assigns color 4 to vertex E.



To construct infinitely many such graphs, extend the chain on the left.

Part 3. Following the hint, construct the palette triangle, a pair of adjacent vertices for each variable, and two OR gates for each clause. Their connections are illustrated in the following figure.



The triangle on the right is the palette triangle, whose vertices are labeled T (True), F (False), and N (None). The two vertices corresponding to variable x are labeled as the two literals x and $\bar{x}$. The two unlabeled triangles are OR gates, which together constitute the clause $(\ell_1 \vee \ell_2 \vee \ell_3)$, where each ℓ_i is a positive or negative literal. The OR gates are connected to the appropriate variable vertices (those labeled ℓ_i in the figure). The first OR gate's output is connected to one input of the second OR gate, and the second OR gate's output is connected to both vertices F and N in the palette triangle.

Let us prove that the Boolean formula φ is satisfiable if and only if the corresponding graph is 3-colorable. Suppose that only three colors are available: T, F, and N. Then, the palette triangle must have all three colors, which, without loss of generality, are arranged as in the figure (if they are not, rename the colors appropriately). All variable vertices are connected to N, and hence they can only take colors T and F. Moreover, the two vertices corresponding to the same variable are connected, forcing their colors to be different. Hence, assigning a color to them is equivalent to assigning a value to that variable. Also, because the output vertex of each clause is connected to both F and N, its color must be T. This represents the fact that all clauses must evaluate to true in order for the formula to be satisfied. Now, it is straightforward to see that the output vertex can take color T if and only if at least one of the three vertices labeled ℓ_i has color T (assuming its color can be only T or F, and not N, since they are variable vertices). That is, if one ℓ_i has color T, then there is a 3-coloring of the two OR gates; if all the ℓ_i's have color F, then no such 3-coloring exists. This concludes the proof that φ is satisfiable if and only if the graph is 3-colorable.

Problem 5. In this problem we *analyze* the running time of some variants of the Mergesort algorithm. Instead of partitioning the input array into two roughly balanced parts, we will partition it into m not necessarily balanced parts (m may be a constant, or it may depend on the size of the array, n). Then we will recursively run the modified Mergesort algorithm on each part, and we will merge the resulting m sorted arrays. Your task is to compute the *running time* of each of the following variants of the algorithm. (You do <u>not</u> have to *design* any algorithm or write any pseudo-code in this problem.)

Part 1 (10%). Let $m \geq 2$ be a constant, and let the m subarrays have the same size (for simplicity, suppose that n is a power of m). The sorted subarrays are merged in any order. Prove that the running time of this variant is $O(n \log n)$.

Part 2 (25%). Let $m = \sqrt{n}$, and let the m subarrays have the same size (for simplicity, assume that m is an integer, at all levels of the recursion tree). The sorted subarrays are merged using a priority queue implemented as a binary heap (hence, insertions and deletions are performed in logarithmic time, with respect to the size of the queue). Prove that the running time of this variant is $O(n \cdot \log n \cdot \log\log n)$. [*Hint: show that the recursion tree has* $O(\log\log n)$ *levels.*]

Part 3 (30%). Let $m = 2$, and let the two subarrays have sizes $n/3$ and $2n/3$, respectively (for simplicity, assume these are always integers). The sorted subarrays are merged as usual. Prove that the running time of this variant is $O(n \log n)$. [*Hint: use induction to show that* $T(n) \le C \cdot n \log n$*, for some constant* C*.*]

Solution.

Part 1. All the subarrays have size n/m. No matter in what order they are merged, $m-1$ merges are performed between arrays of size less than n. Hence this process takes time $O(mn)$, which is $O(n)$, because m is a constant. Therefore the running time is $T(n) = m \cdot T(n/m) + O(n)$. By the Master theorem, $T(n) = O(n \log n)$.

Part 2. At the k-th level of the recursion, the subarrays have size $n^{1/2^k}$. Suppose that the base case has constant size $c > 1$. Then, the base level is reached when k satisfies the equation $n^{1/2^k} = c$. Solving for k yields $k = \log_2 \log_c n = \log_2(\log_c 2 \cdot \log_2 n) = \log_2 \log_c 2 + \log_2 \log_2 n = O(\log\log n)$, because c is a constant. Hence the recursion tree has $O(\log\log n)$ levels.

At each level, each sub-problem is solved by merging some subarrays that are stored in a priority queue. At each step of the merging process, the subarray with the smallest first element is ejected from the priority queue, its first element is extracted, and the reduced sub-array is re-inserted into the queue, with a new priority given by its new first element. Since the number of sub-arrays cannot exceed n, each of these operations is done in time $O(\log n)$.

Now observe that the sizes of all the sub-problems at each given level have sum exactly n. No matter how the n elements are distributed among the sub-problems, each element is first inserted into a queue and then extracted when its turn comes. So, processing each element takes time $O(\log n)$, and no element is processed twice in the same level. Therefore, the total amount of work that is done at each level is $O(n \log n)$ (this also includes splitting the sub-problems into smaller sub-problems, which collectively takes time $O(n)$, at each level).

In conclusion, there are $O(\log\log n)$ levels, and the time spent at each level is $O(n \log n)$, implying that the total running time is $O(n \cdot \log n \cdot \log\log n)$.

Part 3. Merging the two sub-arrays with the usual Merge procedure takes time $O(n)$, regardless of their sizes. Hence the running time satisfies the recursion $T(n) = T(n/3) + T(2n/3) + O(n)$. Let us prove by induction that $T(n) \le C \cdot n \log_2 n$, for some constant C, and for large-enough values of n. Note that, for all n smaller than any given constant, we have $T(n) \le C'$, where C' is a suitable constant. This implies that setting $C \ge C'$ works for all small-enough values of n.

Let us now prove that $T(n) \le C \cdot n \log_2 n$ holds for a given large-enough n, assuming that it holds for all the smaller values of n. n is taken to be sufficiently large, so that the term $O(n)$ in the recursion is at most $D \cdot n$, for some constant D (and no additive constant). Therefore, the recursion becomes

$T(n) \le T(n/3) + T(2n/3) + Dn$. Applying the inductive hypothesis to $T(n/3)$ and $T(2n/3)$, we get

$$\begin{aligned}
T(n) &\le \frac{1}{3}\,C \cdot n\log_2(n/3) + \frac{2}{3}\,C \cdot n\log_2(2n/3) + Dn \\
&= \left(\frac{1}{3}\,C \cdot n\log_2 n - \frac{1}{3}\,C \cdot n\log_2 3\right) + \left(\frac{2}{3}\,C \cdot n\log_2 n - \frac{2}{3}\,C \cdot n\log_2 \frac{3}{2}\right) + Dn \\
&= C \cdot n\log_2 n + \left(-\frac{1}{3}\,C \cdot \log_2 3 - \frac{2}{3}\,C \cdot \log_2 3 + \frac{2}{3}\,C + D\right) \cdot n \\
&= C \cdot n\log_2 n + \left(-C \cdot \log_2 3 + \frac{2}{3}\,C + D\right) \cdot n.
\end{aligned}$$

If we want $T(n)$ to be at most $C \cdot n\log_2 n$, we need the right-hand-side term to be negative. This condition can be expressed as

$$D - C \cdot \log_2 3 + \frac{2}{3}\,C \le 0,$$

which is equivalent to

$$C \ge \frac{D}{\log_2 3 - 2/3}.$$

Setting $C = 2D$ abundantly suffices.

Since we must also have $C \ge C'$, we may choose $C = \max\{C', 2D\}$, which allows us to conclude that $T(n) = O(n\log n)$.