Algorithms Tutorial Problems 3
Greedy Strategy
Solutions

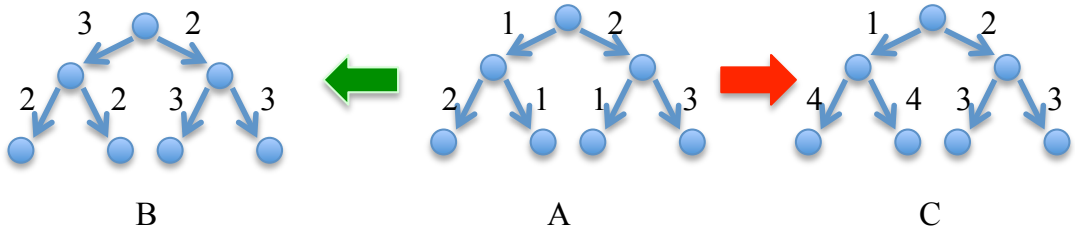
1. There are N robbers who have stolen N items. You would like to distribute the items among the robbers (one item per robber). You know the precise value of each item. Each robber has a particular range of values they would like their item to be worth (too cheap and they will not have made money, too expensive and they will draw a lot of attention). Devise an algorithm that either distributes the items so that each robber is happy or determines that there is no such distribution.

   Solution: Order the items so that their values are increasing. Take the lowest value item v_1 and consider all thieves such that v_1 is in their range of acceptable values. Pick the one with the smallest upper limit and give the first item to that thief. Continue in this manner considering the item with the next smallest value v_2 and so forth.

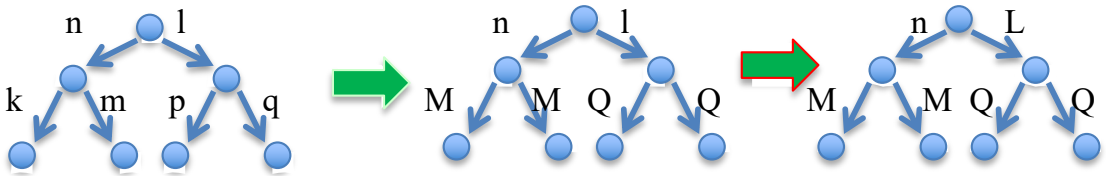
   We now need to prove that this method is optimal. For each thief i let L_i be their lowest acceptable value and let U_i be their highest acceptable value. Assume that there is an assignment of items to thieves which satisfies all thieves, but which is different to that obtained by our greedy strategy. This means there is at least one item assignment which violates our greedy assignment policy. Let item k with value v_k be the least valuable such item, and suppose that this item was assigned to thief i (so $v_k \in [L_i, U_i]$). Since this item assignment violated our greedy policy, there must be another thief j (with range $[L_j, U_j]$) who would have been happy with item k, but whose highest acceptable value is lower than thief i's, so $v_k \in [L_j, U_j]$ and $U_j < U_i$. Now suppose this thief was assigned an item m with value v_m (so $v_m \in [L_j, U_j]$). Since item k is the least value item for which our greedy strategy was violated, $v_k < v_m$. Hence, $v_m \in [L_i, U_i]$, which means that thief i would be happy with item m, and we

can therefore swap the assignments for the two thieves while keeping them both happy. By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy. Therefore, our method is optimal.

2. (Timing Problem in VLSI chips) Consider a complete binary tree with $n = 2^k$ leaves. Each edge has an associated positive number that we call the length of the edge (see figure below). The distance from the root to a leaf is the sum of the lengths of all edges from the root to the leaf. The root emits a clock signal and the signal propagates along all edges and reaches each leaf in time proportional to the distance from the root to that leaf. Design an algorithm which increases the lengths of some of the edges in the tree in a way that ensures that the signal reaches all the leaves at the same time while the total sum of the lengths of all edges is minimal. (For example, in the picture below if the tree A is transformed into trees B and C all leaves of B and C have a distance of 5 from the root and thus receive the clock signal at the same time, but the sum of the lengths of the edges in C is 17 while sum of the lengths of the edges in B is only 15.)

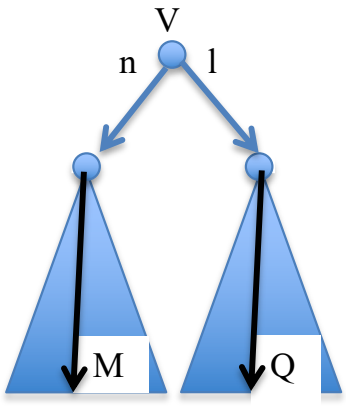


Solution: We proceed by recursion, working from the leaves towards the root. Consult the figure below.



$M = \max\{k,m\}$; $Q = \max\{p,q\}$; If $M + n > Q + l$ $L = M+n-Q$

We start with the edges connecting two adjacent leaves. Clearly, they have to be of the same length. Thus, referring to the figure, we let $M = \max\{k, m\}$ and $Q = \max\{p, q\}$ and change the shorter length of the two to the value of the longer length. Moving one level up, we consider $M + n$ and $Q + l$. If $M+n > Q+l$, then we increase l to $L = M+n-Q$, so that $M+n = Q+L$. If $Q+l > M+n$, then we increase n to $N = Q+L-M$ so that $M+N = Q+l$.



We continue in this manner; assuming that we have made the lengths of all paths to all leaves of the subtree T_1 equal to M and the lengths of all paths to all leaves of the subtree T_2 equal to Q, we consider the edges connecting a vertex V to the roots of these subtrees. Assuming that the edge connecting V to the root of T_1 is of length n and that the edge connecting V to the root of T_2 is of length l; if $M+n > Q+l$ we replace the edge of length l with an edge of length $M+n-Q$; if $Q+l > M+n$ we replace the edge of length n with an edge of length $Q+l-M$. Clearly, the produced tree is of minimal total length with equal distances from the root to all leaves.

3. Assume that you are given a complete graph G with weighted edges such that all weights are distinct. We now obtain another complete weighted graph G' by replacing all weights $w(i,j)$ of edges $e(i,j)$ with new weights $w(i,j)^2$.

   (a) Assume that T is the minimal spanning tree of G. Does T necessarily remain the minimal spanning tree for the new graph G'?

   (b) Assume that p is the shortest path from a vertex u to a vertex v in G. Does p necessarily remain the shortest path from u to v in the new graph G'?

Solution:

(a) Yes; just repeat Kruskal's algorithm with the new weights. The ordering of the edges does not change so the same minimum spanning tree will be produced.

(b) No; consider for example a graph G with vertices A, B, C and D and edges $AB = 1$, $BD = 5$, $AC = 3$, $CD = 4$, $AD = 7$ and $BC = 8$. Then the shortest path from A to D in G is ABD with length 6, while the path ACD is longer, with length 7. However, after squaring all the weights, the length of ABD becomes $1 + 25 = 26$, while the length of ABD becomes $9 + 16 = 25$ which is shorter.

4. Assume that you are given a complete weighted graph G with n vertices $v_1, \ldots, v_n$ and with the weights of all edges distinct and positive. Assume that you are also given the minimum spanning tree T for G. You are now given a new vertex v_{n+1} and the weights $w(n+1, j)$ of all new edges $e(n+1, j)$ between the new vertex v_{n+1} and all old vertices $v_j \in G$, $1 \leq j \leq n$. Design an algorithm which produces a minimum spanning tree T' for the new graph containing the additional vertex v_{n+1} and which runs in time $O(n \log n)$.

   Solution: It should be clear that only the new edges and the edges of the old minimum spanning tree have a chance of being included in the new minimum spanning tree. Thus, to obtain a new spanning tree just run Kruskal's algorithm on the $n-1$ edges of the old spanning tree plus the n new edges. The runtime of the algorithm will be $O(n \log n)$.

5. You have n items for sale, numbered from 1 to n. Alice is willing to pay $a[i] > 0$ dollars for item i, and Bob is willing to pay $b[i] > 0$ dollars for item i. Alice is willing to buy no more than A of your items, and Bob is willing to buy no more than B of your items. Additionally, you must sell each item to either Alice or Bob, but not both, so you may assume that $n \leq A+B$. Given $n, A, B, a[1 \ldots n]$ and $b[1 \ldots n]$, you have to determine the **maximum** total amount of money you can earn in $O(n \log n)$ time.

   Solution: Let $d[i] = |a[i] - b[i]|$; sort all $d[i]$ in a decreasing order and re-index all the items such that $|a[1] - b[1]|$ is the largest difference and so on, the i^{th} item being such that $d[i] = |a[i] - b[i]|$ is the i^{th} difference in size. We now go through the list giving the i^{th} item to Alice if $a[i] > b[i]$ and the total number of items given to Alice thus far is at most A and giving instead this item to Bob if $b[i] > a[i]$ and the total number of items given to Bob thus far is at

most B. If at certain stage A is reached we give all the remaining items to B and similarly if B is reached we give all the remaining items to A. If neither A nor B are reached and there are leftover items for which $d[i] = 0$, i.e., such that $a[i] = b[i]$ we give them to either Alice or Bob making sure neither A nor B is exceeded.

To see that this algorithm is optimal, let $m[i] = \min(a[i], b[i])$. Then regardless of who gets item i you get at least the amount $m[i]$, plus you get the amount $d[i]$ if item i is given to the higher bidder. Our algorithm tries to get as many $d[i]$'s as possible, preferentially taking as large $d[i]$'s as possible, so the algorithm is clearly optimal.

Computing all the differences $d[i] = |a[i] - b[i]|$ takes $O(n)$ time; sorting $d[i]$'s takes $O(n \log n)$ time and going through the list takes $O(n)$ time. Thus the whole algorithm runs in time $O(n \log n)$.

6. Assume that you have an unlimited number of \$2, \$1, 50c, 20c, 10c and 5c coins to pay for your lunch. Design an algorithm that, given the cost that is a multiple of 5c, makes that amount using a minimal number of coins.

Solution: Keep giving the coin with the largest denomination that is less than or equal to the amount remaining, until the desired amount is reached. To prove that this results in the smallest possible number of coins for any amount to be paid, assume the opposite - that is, suppose that for a certain amount M, there is an optimal way of payment which is more efficient than the one described by the greedy algorithm. Clearly, since the ordering in which the coins are given does not matter, we can assume that such payment proceeds from the largest to the smallest denomination. Consider the instance of the greedy policy being violated. This can happen, for example, if the remaining amount to be paid is at least \$2, but a \$2 dollar coin was not used. However, if this is the case, notice that at most one \$1 coin could have been used, as otherwise the payment would not be efficient because two \$1 coins can be replaced by a single \$2 dollar coin. Thus, after the option of giving a \$1 dollar coin has been exhausted we are left with at least \$1 to be given without using any \$1 dollar coins. For the same reason at most one 50 cent coin can be given and we are left with an amount of at least 50 cents to be given without using any 50 cent coins. Note that at most two 20 cent coins can be used because three 20 cent coins can be replaced with a 50 cent and a 10 cent coins. Also note that if two 20 cent coins are used, no 10 cent coins can be used because two 20 cent coins and a 10 cent coin can be replaced with a single 50 cent coin. Thus, if two 20 cent coins are used, only 5 cent coins can be used to give the remaining amount

of at least 10 cents, which would require two 5 cent coins, but these could be replaced by a single 10 cent coin, contradicting optimality. If, on the other hand, only one 20 cent coin is used to give an amount of at least 50 cent we are left with at least 30 cents to be given using 10 cent and 5 cent coins. Note that in such a case only one 10 cent coin can be used because two 10 cent coins can be replaced by a 20 cent coin. So we are left an amount of at least 20 cents to be given with 5 cent coins only. However only one 5 cent coin can be used because two 5 cent coins can be replaced by one ten cent coin contradicting the optimality of the solution. If the greedy strategy was violated for the first time when smaller amounts are due, the analysis is a subset of the analysis above. Thus, the greedy strategy provides an optimal solution. Note that in all countries the notes and coins denominations are chosen so that the greedy strategy is optimal.

7. Assume that the denominations of your $n+1$ coins are $1, c, c^2, c^3, \ldots, c^n$ for some integer $c > 1$. Design a greedy algorithm which, given any amount, makes that amount using a minimal number of coins.

   Solution: As in the previous problem, keep giving the coin with the largest denomination that is less than or equal to the amount remaining. To prove that this is optimal, we once again assume there is an amount for which there is a payment strategy that is more efficient than the greedy strategy, and assume that the coins are given in order of decreasing denomination. At some point during the payment, the greedy strategy will be violated, which means that the remaining amount is at least c^j for some j but the strategy chooses not to give a coin of c^j cents. So the next-largest denomination that can be used is c^{j-1}. However, note that the strategy can give at most $c-1$ coins of denomination c^{j-1}, because c many coins of denomination c^{j-1} can be replaced with a single coin of denomination c^j. Thus, after giving fewer than c many coins of denomination c^{j-1} we are left with at least the amount $c^j - (c-1)c^{j-1} = c^{j-1}$ to be given using only coins of denomination c^{j-2}. Continuing in this manner, we eventually end up having to give at least c cents using only 1 cent coins which contradicts the optimality of the method.

8. Give an example of a set of denominations containing the single cent coin for which the greedy algorithm does not always produce an optimal solution.

   Solution: Consider the denominations 4c, 3c and 1c and an amount to be paid of 6 cents. The greedy algorithm would first give one 4c coin and would then be forced to give 2 cents using two 1c coins. However, giving two 3c coins

is more optimal.

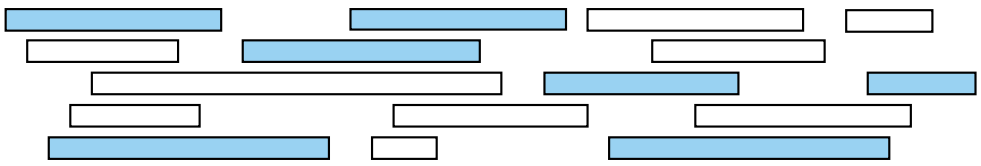
9. Given two sequences of letters A and B, find if B is a subsequence of A in the sense that one can delete some letters from A and obtain the sequence B.

   Solution: Use a simple greedy strategy. First, find and mark the earliest occurrence of the first letter of B in A. Then, for each subsequent letter of B, find and mark the earliest occurrence of that letter in A which is after the last marked letter. If you reach the end of B before or at the same time as you reach the end of A, then B is a subsequence of A.

10. There is a line of 111 stalls, some of which lack a roof and need to be covered with boards. You can use up to 11 boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible.

    Solution: First, cover all roofless stalls with a single board that stretches from the first roofless stall to the last roofless stall. Then, repeat the following up to 10 times: Find the longest line of consecutive roofed stalls that are covered by a board, and then excise that part of the board, replacing it with two boards. Stop when there is no stall with a roof that is covered by a board.

11. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a tiling path if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The size of a tiling cover is just the number of intervals. Describe and analyse an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.



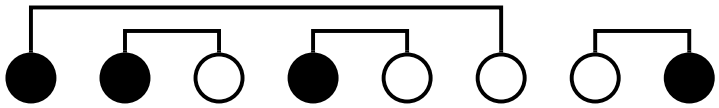
A set of intervals. The seven shaded intervals form a tiling path.

Solution: Sort the intervals in increasing order of their left endpoints. Start with the interval with the smallest left endpoint; if there are several intervals

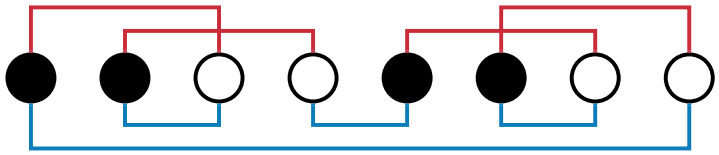
with the same such smallest left endpoint choose the one with the largest right endpoint. Then, consider all intervals whose left endpoints are in the last chosen interval, and pick the one with the largest right endpoint. Continue in this manner until an interval with the absolute largest right endpoint is chosen.

The algorithm involves sorting the intervals, and then performing by a single pass through all of the intervals. Hence, the algorithm runs in $O(n \log n)$ time.

12. Assume that you are given n white and n black dots lying in a random configuration on a straight line, equally spaced. Design a greedy algorithm which connects each black dot with a (different) white dot, so that the total length of wires used to form such connected pairs is minimal. The length of wire used to connect two dots is equal to the straight-line distance between them.



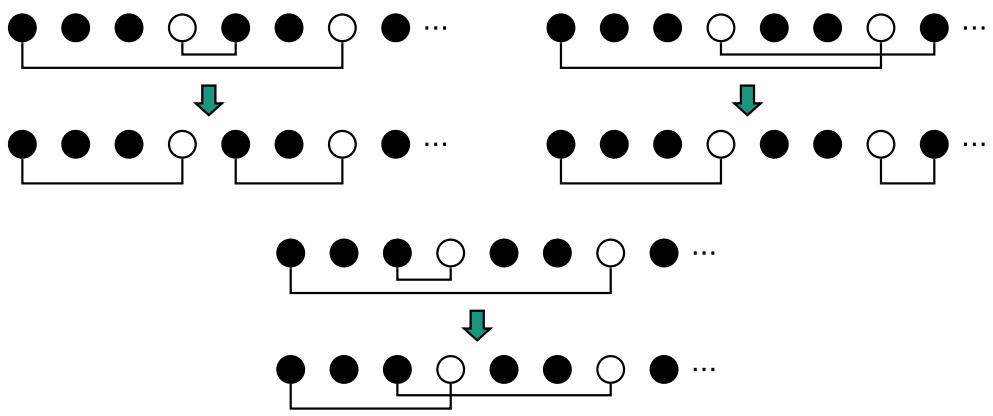
Solution: One should be careful about what kind of greedy strategy one uses. For example, connecting the closest pairs of equally coloured dots produces suboptimal solution as the following example shows:



Connecting the closest pairs (blue lines) uses $3 + 7 = 10$ units of length while the connections in red use only $4 \times 2 = 8$ units of length.

The correct approach is to go from left to right and connect the leftmost dot with the leftmost dot of the opposite colour and then continue in this way. To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots. Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost available dot of the opposite colour. We look at the leftmost dot for which the greedy strategy is violated. There are three types of configurations to consider (two are shown

below) - but for each configuration, the strategy uses either the same amount or more wire than the greedy strategy. Hence, the hypothetical strategy is no better than the greedy strategy, contradicting our original assumption, and so the greedy strategy is optimal.

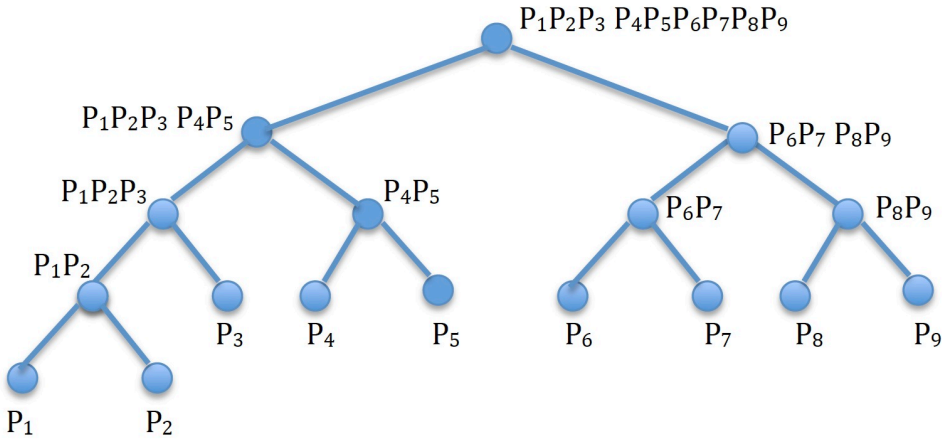


13. Assume you are given n sorted arrays P_i, $1 \leq i \leq n$, of different sizes e_i. You are allowed to merge any two arrays into a single new sorted array and proceed in this manner until only one array is left. Design an algorithm that achieves this task and uses minimal total number of moves of elements of the arrays. Give an informal justification for why your algorithm is optimal.

    Solution: Clearly, the elements of the arrays that are merged earlier will be moved more times than the elements of the arrays that are merged later. (Note that this problem is almost identical to the problem of Huffman encoding with the normalised numbers of elements $e_i / \sum_{j=1}^{n} e_j$ taken as frequencies.) Thus, we take the two shortest arrays and merge them, and continue with the resulting set of arrays in the same manner until there is only one array remaining.

    In the figure above, the shortest arrays were P_1 and P_2 so they were merged first; then the shortest arrays were the merged array P_1P_2 and P_3, then the pair P_4 and P_5, then P_6 and P_7, then P_8 and P_9, then $P_1P_2P_3$ and P_4P_5, and so on. The proof of optimality is the same as for the Huffman codes, see the textbook.

14. A photocopying service with a single large photocopying machine faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to schedule the jobs on their single machine in an order



that keeps their customers the happiest. Customer i's job will take t_i time to complete. Given a schedule (i.e., an ordering of the jobs), let C_i denote the finishing time of job i. For example, if job i is the first to be done we would have $C_i = t_i$, and if job j is done right after job i, we would have $C_j = C_i + t_j$. Each customer i also has a given weight w_i which represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of their job. So the company decides that they want to order the jobs to minimise the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$. Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for job i. You want to order the jobs so as to minimise the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Solution: Schedule the jobs in decreasing order of w_i/t_i. This problem is very similar to the Tape Storage problem, which was covered in the lectures. To prove that this is optimal, we first introduce the concept of an inversion. We say that two jobs i and j form an inversion if job i is scheduled before job j, but $w_i/t_i < w_j/t_j$. Now consider any schedule which violates our greedy scheduling. Such a schedule will contain an inversion of at least one pair of consecutive jobs. If we repeatedly fix all such inversions between two consecutive jobs (by swapping them in the schedule) without increasing the value of $S = \sum_{i=1}^{n} w_i C_i$ then, by the "bubble sort algorithm" argument, all inversions will eventually disappear and we will have shown that the greedy solution is no worse than the solution considered. So let us prove that swapping two consecutive inverted jobs can only reduce the value of $\sum_{i=1}^{n} w_i C_i$. Assume that we have re-enumerated the jobs so that they are numbered according to their place in the schedule, so that the two inverted jobs are numbered i and $i+1$. Now let T be the

time just before job i starts. Note that by swapping two successive jobs only two terms of the sum $\sum_{i=1}^{n} w_i C_i$ change: before the swap this sum contains $w_i(T+t_i)+w_j(T+t_i+t_{i+1})$, while after the swap the new sum S' will contain $w_{i+1}(T+t_{i+1})+w_i(T+t_{i+1}+t_i)$. Thus,

$$\begin{aligned} S - S' &= w_i(T+t_i) + w_{i+1}(T+t_i+t_{i+1}) - w_{i+1}(T+t_{i+1}) + w_i(T+t_{i+1}+t_i) \\ &= w_iT + w_it_i + w_{i+1}T + w_{i+1}t_i + w_{i+1}t_{i+1} - w_{i+1}T - w_{i+1}t_{i+1} - w_iT - w_it_{i+1} - w_it_i \\ &= w_{i+1}t_i - w_it_{i+1}. \end{aligned}$$

We now note that if $w_i/t_i < w_{i+1}/t_{i+1}$ then $w_it_{i+1} < w_{i+1}t_i$ which implies $w_{i+1}t_i - w_it_{i+1} > 0$ and thus $S - S' = w_{i+1}t_i - w_it_{i+1} > 0$, i.e., $S > S'$ and consequently the total sum has decreased. This completes the proof of optimality of our schedule.

15. You are running a small manufacturing shop with plenty of workers, but just a single milling machine. You have to produce n items; item i requires m_i machining time first, then p_i polishing time by hand; finally it is packaged by hand and this takes c_i amount of time. The machine can mill only one item at a time, but your workers can polish and package in parallel as many objects as you wish. You have to determine the order in which the objects should be machined so that the whole production is finished as quickly as possible. Prove that your solution is optimal.

    Solution: Ignore the machining time and sort the jobs in decreasing order with respect to the sum $p_i + c_i$. The proof of optimality is straightforward. Just as in the previous example, it is enough to show that if $p_i + c_i < p_{i+1} + c_{i+1}$ and yet job i has been scheduled before job $i+1$, then the completion time cannot increase if we swap jobs i and $i+1$.

16. You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if a job is accepted, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the activity selection problem.) Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.

Solution: This problem is similar to the Activity Selection problem, except that time is now a 24hr circle, rather than an interval, because there might be jobs whose start time is before the midnight and finishing time after midnight. However, we can reduce it to several instances of the interval case.

Let $S = \{J_1, J_2, \ldots, J_k\}$ be the set of all jobs whose start time is before midnight and finishing time is after midnight. We now first exclude all of these jobs and sort the remaining jobs by their finishing time. We now solve in the usual manner the activity selection problem with the remaining jobs only, by always picking a non-conflicting job with the earliest finishing time. We record the number of accepted jobs obtained in this manner as a_0. We now start over, but this time we take J_1 as our first job, and we record the number of accepted jobs obtained as a_1. We repeat this process with the rest of the jobs in S, recording the number of accepted jobs as $a_2, \ldots, a_k$. Finally we pick the selection of jobs for which the corresponding a_m is the largest, $0 \leq m \leq k$.

To prove optimality, we note that the optimal solution either does not contain any of the jobs from the set S or contains just one of them. If it does not contain any of the jobs from S, then it would have been constructed when the problem was solved for all jobs excluding S, by the same argument as was given for the Activity Selection problem; if it does contain a job J_m from S, then it would have been constructed during the round when we started with J_m.

Time complexity: Sorting all jobs which are not in $S = \{J_1, J_2, \ldots, J_k\}$ takes at most $O(n \log n)$ time. Each of the $k + 1$ procedures run in linear time (because we go through the list of all jobs by their finishing time only once, checking if their start time is before or after the finishing time of the last chosen activity). The number of rounds is at most n, so the time complexity is $O(n \log n + n^2) = O(n^2)$.

17. Assume that you got a fabulous job and you wish to repay your student loan as quickly as possible. Unfortunately, the bank *Western Road Robbery* which gave you the loan has the condition that you must start your monthly repayments by paying off \$1 and then each subsequent month you must pay either double the amount you paid the previous month, the same amount as the previous month or half the amount you paid the previous month. On top of these conditions, your schedule must be such that the last payment is \$1. Design an algorithm which, given the size of your loan, produces a payment schedule which minimises the number of months it will take you to repay your loan while satisfying all of the bank's requirements. If the optimality of your solution is

obvious from the algorithm description, you do not have to provide any further correctness proof.

Solution: To repay your loan as quickly as possible, you will want to double your repayment as much as possible while still ensuring that you can finish your repayment with a payment of \$1 at the end. So assuming that the value of your loan is S; we first find the largest n such that $2(1+2+2^2+\ldots+2^n) \leq S$. This amounts to finding the largest n such that $2(2^{n+1}-1) \leq S$. Let $R = S - 2(2^{n+1}-1)$. Since $2(2^{n+2}-1) > S$, we get

$$R = S - 2(2^{n+1}-1) < 2(2^{n+2}-1) - 2(2^{n+1}-1) = 2^{n+3} - 2^{n+2} = 2^{n+2}$$

i.e.,

$$R \leq 2^{n+2} - 1 = 1 + 2 + 2^2 + \ldots + 2^{n+1}$$

You can now return your loan as follows: start with \$1 and keep doubling until you reach 2^n. If $R \geq 2^{n+1}$ double once again; if this is the case let $P = R - 2^{n+1}$; otherwise let $P = R$. Represent P in binary; you now start halving the amount you pay, but you repeat once the amount 2^k whenever the k^{th} bit of P is one.

It should be clear that this produces an optimal solution; for those who like rigorous proofs, one can prove optimality in a very similar way to the proof of the *Giving Change* problem with coins $1, c, c^2, c^3, \ldots, c^n$.

18. Alice wants to throw a party and is deciding whom to invite. She has n people to choose from, and she has created a list consisting of pairs of people who know each other. She wants to invite as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and at least five other people whom they do not know. Give an efficient algorithm that takes as input the list of n people and the list of all pairs who know each other and outputs a subset of these n people which satisfies the constraints and which has the largest number of invitees. Argue that your algorithm indeed produces a subset with the largest possible number of invitees.

Solution: For each person, count the number of people they know and the number of people they do not know, recording this in a table. Now eliminate all people who know fewer than five other people or don't know fewer than five people among the set of people currently being considered. Then, update the count of known/unknown people for each affected person. Continue in this manner until everyone left satisfies the condition.

This strategy is clearly optimal, as we begin with the set of all people, and only eliminate people when they do not satisfy the condition.

19. In Elbonia cities are connected with one way roads and it takes one whole day to travel between any two cities. Thus, if you need to reach a city and there is no direct road, you have to spend a night in a hotel in all intermediate cities. You are given a map of Elbonia with toll charges for all roads and the prices of the cheapest hotels in each city. You have to travel from the capital city C to a resort city R. Design an algorithm which produces the cheapest route to get from C to R.

    Solution: This is almost the shortest path problem, except that now going through a vertex (i.e., city) also has a weight associated with it. To reduce this problem to the shortest path problem, split each vertex (representing a city) into two new vertices. The first of these vertices will represent arrivals at that city, and will receive all the incoming edges of the original vertex, while the other will represent departures from that city, and will be the starting point of all outgoing edges from the original vertex. Connect the two new vertices with an edge whose weight is the price of spending the night in that city. Now use Dijkstra's algorithm with the vertex representing departures from C as the source and the vertex representing arrivals at R as the destination.

20. You are given a set S of n overlapping arcs of the unit circle. The arcs can be of different lengths. Find a largest subset P of these arcs such that no two arcs in P overlap (largest in terms of the total number of arcs, not in terms of the total length of these arcs). Prove that your solution is optimal.

    Hint: compare with problem 16.

    Solution: This problem is equivalent to problem 16. Convert each arc into a time period (by, for example, superimposing a 24-hour clock onto the unit circle) and proceed as in problem 16.

21. You are given a set S of n overlapping arcs of the unit circle. The arcs can be of different lengths. You have to stab these arcs with a minimal number of needles so that every arc is stabbed at least once. In other words, you have to find a set of as few points on the unit circle as possible so that every arc contains at least one point. Prove that your solution is optimal.

    Hint: compare with problem 16.

    Solution: To avoid confusion, we will imagine that the arcs are laid out on a straight line (with some arcs potentially wrapping around the ends). Let A_1 be

the arc whose right endpoint occurs earliest on the line, A_2 be the arc whose right endpoint occurs next-earliest, and so on.

Stab A_1 at its right endpoint. Then, consider the arcs that have not yet been stabbed, and pick the arc whose right endpoint occurs the earliest (with respect to the last stabbed point) and stab it at its right endpoint. Repeat in this manner until all arcs have been stabbed, and record the number of needles used as n_1. Now start over, but this time begin by stabbing A_2 at its right endpoint, and then continue in the above manner, wrapping around the end of the line if necessary. Record the number of needles used as n_2. Do this for each arc, and then take the arrangement for which the number of needles used is minimal.

22. You are given a connected graph with weighted edges. Find a spanning tree such that the largest weight of all of its edges is as small as possible.

Solution: Finding the minimum spanning tree solves this problem.

To prove that this is optimal, we first introduce the notion of a cut. A cut is a partition of the vertices of a graph into two disjoint subsets. An edge that connects a vertex in one of the subsets to a vertex in the other subset is said to cross the cut.

Suppose that T is a minimum spanning tree of G and T' is an optimal solution of the problem. Let e be the largest weight edge in T (with weight $c(e)$), and consider the cut in T defined by e. Since T is a minimum spanning tree, every edge that crosses the cut must have a weight of at least $c(e)$ (otherwise e would not be in the minimum spanning tree). Now consider the edges of T' which cross the cut. Since T' is connected, such crossing edges must exist. By the above statement these edges must also have a weight of at least $c(e)$. But if the weight of any of these edges is larger than $c(e)$, T would be more optimal than T', which would contradict our original assumption. Hence, these edges must have a weight of exactly $c(e)$.

Hence, the maximum edge weight of a minimum spanning tree is equal to the maximum edge weight of the optimal solution, and therefore, a minimum spanning tree is optimal.

23. You need to write a very long paper titled "The Meaning of Life". You have compiled a sequence of books in the order that you will need them, some of them multiple times. Such a sequence might look something like this:

$$B_1, B_2, B_1, B_3, B_4, B_5, B_2, B_6, B_4, B_1, B_7, \ldots$$

Unfortunately, the library only lets you borrow ten books at a time, so every now and then you have to make a trip to the library to exchange books. On each trip you can exchange any number of books. Design an algorithm which decides which books to exchange on each library trip so that the total number of trips which you will have to make to the library is as small as possible.

Solution: A simple greedy algorithm will suffice. Borrow the first ten books that you need according to your sequence, then when you need a book which you do not have, return to the library and exchange the books which you currently have for the ten books that you will need next, according to your sequence (which may involve keeping some books that you are currently borrowing). Repeat in this manner until you have finished your paper!

This algorithm is clearly optimal as it delays making a trip to the library for as long as possible.

24. Assume that you are given a set X of n disjoint intervals $I_1, I_2, \ldots, I_n$ on the real line and a number $k \leq n$. You have to design an algorithm which produces a set Y of at most k intervals $J_1, J_2, \ldots, J_k$ which cover all intervals from X (i.e. $\bigcup_{i=1}^{n} I_i \subseteq \bigcup_{j=1}^{k} J_j$) and such that the total length of all intervals in Y is minimal. (Note that we do not require that $Y \subseteq X$, i.e., intervals J_j can be new. For example, if your set X consists of intervals $[1, 2]$, $[3, 4]$, $[8, 9]$, and $[10, 12]$ and if $k = 2$, then you should choose intervals $[1, 4]$ and $[8, 12]$ of total length $3 + 4 = 7$.)

    Solution: This problem is similar to problem 10. First, create a single large interval that covers all the intervals in X, and add this interval to Y. Then, until there are k intervals in Y, select the interval J_j in Y which contains the largest gap, where a gap is an interval between two consecutive intervals in X, and cut out this gap from J_j, producing two new (smaller) intervals.

25. Assume that you are given a complete undirected graph $G = (V, E)$ with edge weights $\{w(e) : e \in E\}$ and a proper subset of vertices $U \subset V$, $U \neq V$. Design an algorithm which produces the lightest spanning tree of G in which all nodes of U are leaves (there might be other leaves in this tree as well).

    Solution: Construct the minimum spanning tree of the graph consisting of all of the vertices in the original graph G except for those in U (i.e., $V - U$). Let this spanning tree be T. Then, for each vertex v in U, select the lowest cost edge that connects it to a vertex in T, and add this to the spanning tree.

26. You are given a connected graph with weighted edges with all weights distinct. Prove that such a graph has a unique minimum spanning tree.

Solution: Consider running Kruskal's algorithm on the graph. When all edge weights are distinct, Kruskal's algorithm never needs to make a choice between two edges, and therefore there is only one possible result. Hence, the graph has a unique minimum spanning tree.

27. You have N students with varying skill levels and N jobs with varying skill requirements. You want to assign a different job to each student, but only if the student meets the job's skill requirement. Design an algorithm to determine the maximum number of jobs that you can successfully assign.

Solution: Going from the least skilled to the most skilled student, assign each student the job with the highest skill requirement that they meet the requirements for and that hasn't already been assigned, if one exists.

To prove that this is optimal, we first introduce some notation. Let an assignment of a student to a job be given by a pair (s, j), where s is the student, and j is the job. Let $L(s)$ be the skill level of student s, and let $R(j)$ be the skill requirement of job j.

Now, assume there is an alternative strategy that also produces an optimal assignment. We order the assignments produced by each strategy in increasing order of the student's skill level, and consider the *first* violation of the greedy policy by the alternative strategy. There are two ways a violation could occur:

(a) *The greedy strategy assigns student s_1 the job j_1, but the alternative strategy assigns the same student the job j_2.*

Since the greedy strategy assigned student s_1 the job j_1, we have $L(s_1) \geq R(j_1)$. Also, since the greedy strategy assigns each student the job with the highest skill requirement that they meet the requirements for (that hasn't already been assigned), we necessarily have $R(j_1) \geq R(j_2)$, otherwise the greedy strategy would not have assigned j_1 to s_1. If the alternative strategy assigned j_1 to a different student, say s_2, this student must also meet the skill requirement for j_2 (since $R(j_1) \geq R(j_2)$). Hence, we can modify the assignment made by the alternative strategy to adhere to the greedy policy by swapping the assignments of the two students.

(b) *The greedy strategy assigns student s_1 the job j_1, but the alternative strategy does not assign s_1 any job.*

Note that in this case, the alternative strategy *must* have assigned j_1 to some student, otherwise the assignment would not be optimal, as we would be able to add the assignment (s_1, j_1). Hence, suppose that the alternative strategy assigned j_1 to a student s_2. Since the greedy strategy considers students in increasing order of skill level, we necessarily have $L(s_1) \leq L(s_2)$. (If $L(s_1) \geq L(s_2)$ then the greedy strategy would have assigned s_2 a job first.) But since the greedy strategy assigned j_1 to s_1, $L(s_1) \geq R(j_1)$. Hence, we can modify the assignment made by the alternative strategy to adhere to the greedy policy by assigning j_1 to s_1 rather than assigning it to s_2.

We have shown that for each possible violation, a modification can be made to the assignment to make it adhere to the greedy policy. Hence, if an assignment contains multiple violations, we can apply these modifications one by one, transforming the assignment into one that would be produced by the greedy strategy. Thus, any optimal assignment can be transformed into an assignment that adheres to the greedy policy, and therefore the greedy strategy is optimal.

28. There are N towns situated along a straight line. The location of the i'th town is given by the distance of that town from the westernmost town, d_i (so the location of the westernmost town is 0). Police stations are to be built along the line such that the i'th town has a police station within A_i kilometers of it. Design an algorithm to determine the minimum number of police stations that would need to be built.

    Solution: This problem is similar to the problem on interval stabbing. For each town i, create the corresponding interval $[d_i - A_i, d_i + A_i]$, and then proceed as in the interval stabbing problem.

29. There are N people that you need to guide through a difficult mountain pass. Each person has a speed in days that it will take to guide them through the pass. You will guide people in groups of up to K people, but you can only move as fast as the slowest person in each group. Design an algorithm to determine the fewest number of days you will need to guide everyone through the pass.

    Solution: Create a group consisting of the K slowest people, another group consisting of the next K slowest people, and so on. Note that the final group consisting of the fastest people may contain fewer than K people. Then guide these groups one by one through the pass (the order in which you guide the groups does not matter).

The optimality of this strategy should be obvious. Consider the slowest person. This person will need to be guided through the mountain pass eventually, and since you can only move as fast as the slowest person in each group, the speed of the other people in the group will not affect the speed of the group. So to save as much time as possible, you should send them with as many of the other slowest people as possible. The same argument applies to the remaining people after this group has been sent.

30. There are N courses that you can take. Each course has a minimum required IQ, and will raise your IQ by a certain amount when you take it. Design an algorithm to find the minimum number of courses you will need to take to get an IQ of at least K.

    Solution: Out of all the courses remaining that you can take, take the course that will raise your IQ the most. Repeat until your IQ is K or higher.

    The optimality of this strategy should be obvious. Taking the course that will raise your IQ the most will make the most courses available to you, thus giving you more options.