Algorithms: COMP3121/9101

Aleks Ignjatović, ignjat@cse.unsw.edu.au
office: 504 (CSE building K 17)
Admin: Song Fang, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
University of New South Wales Sydney

More Greedy Practice Problems

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 U_i and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value v_2 and the remaining available thieves choosing for v_2 the thief among them with the smallest upper bound U_j for which $v_2 \in [L_j, U_j]$, and so fourth.

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 U_i and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value v_2 and the remaining available thieves choosing for v_2 the thief among them with the smallest upper bound U_j for which $v_2 \in [L_j, U_j]$, and so fourth.

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 U_i and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value v_2 and the remaining available thieves choosing for v_2 the thief among them with the smallest upper bound U_j for which $v_2 \in [L_j, U_j]$, and so fourth.

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 U_i and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value v_2 and the remaining available thieves choosing for v_2 the thief among them with the smallest upper bound U_j for which $v_2 \in [L_j, U_j]$, and so fourth.

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 U_i and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value v_2 and the remaining available thieves choosing for v_2 the thief among them with the smallest upper bound U_j for which $v_2 \in [L_j, U_j]$, and so fourth.

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 $U_j < U_i$.

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 $U_j < U_i$.

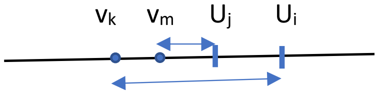
We now need to prove that this method is optimal.

- 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 $U_j < U_i$.

We now need to prove that this method is optimal.

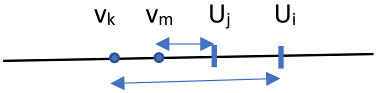
- For each thief i let L_i be their lowest acceptable value and let U_i be their highest acceptable value. Assume that there is an assignment of items to thieves which satisfies all thieves, but which is different to that obtained by our greedy strategy.
- This means there is at least one item assignment 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.



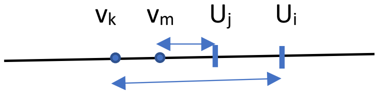
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in $O(n \log n)$; for each item we search through the list to thieves to find all tieves for whom that item is in their range which is done in $O(n)$ for each item. Thus in total this takes $O(n^2)$.
- Thus the whole algorithm runs in time $O(n^2)$.

- 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.



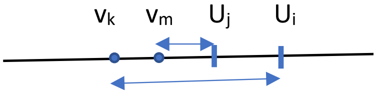
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in $O(n \log n)$; for each item we search through the list to thieves to find all tieves for whom that item is in their range which is done in $O(n)$ for each item. Thus in total this takes $O(n^2)$.
- Thus the whole algorithm runs in time $O(n^2)$.

- 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.



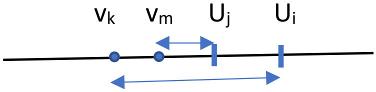
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in $O(n \log n)$; for each item we search through the list to thieves to find all tieves for whom that item is in their range which is done in $O(n)$ for each item. Thus in total this takes $O(n^2)$.
- Thus the whole algorithm runs in time $O(n^2)$.

- 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.



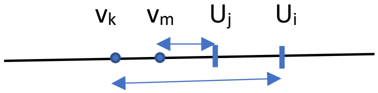
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in $O(n \log n)$; for each item we search through the list to thieves to find all tieves for whom that item is in their range which is done in $O(n)$ for each item. Thus in total this takes $O(n^2)$.
- Thus the whole algorithm runs in time $O(n^2)$.

- 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.



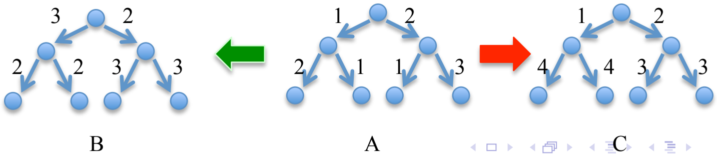
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in $O(n \log n)$; for each item we search through the list to thieves to find all tieves for whom that item is in their range which is done in $O(n)$ for each item. Thus in total this takes $O(n^2)$.
- Thus the whole algorithm runs in time $O(n^2)$.

- 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.

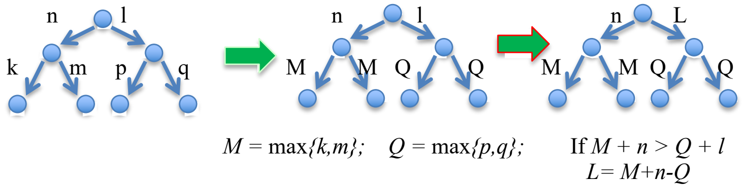


- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in $O(n \log n)$; for each item we search through the list to thieves to find all tieves for whom that item is in their range which is done in $O(n)$ for each item. Thus in total this takes $O(n^2)$.
- Thus the whole algorithm runs in time $O(n^2)$.

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.)

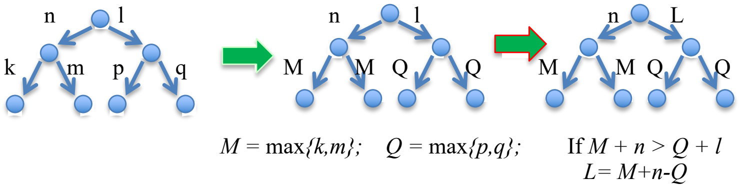


Solution: We proceed by recursion, working from the leaves towards the root. Consult the figure below.



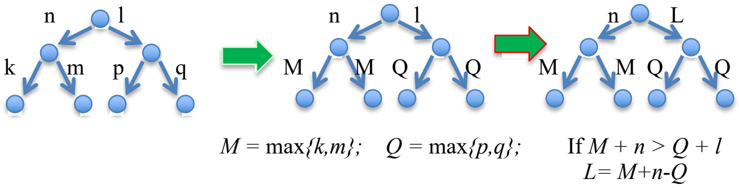
- We start with the edges connecting two adjacent leaves. Clearly, they have to be of the same length.
- Thus, referring to the figure, we let $M = \max\{k, m\}$ and $Q = \max\{p, q\}$ and change the shorter length of the two to the value of the longer length.
- Moving one level up, we consider $M +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

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

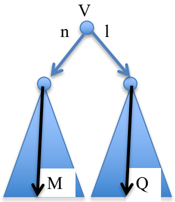


- We start with the edges connecting two adjacent leaves. Clearly, they have to be of the same length.
- Thus, referring to the figure, we let $M = \max\{k, m\}$ and $Q = \max\{p, q\}$ and change the shorter length of the two to the value of the longer length.
- Moving one level up, we consider $M +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

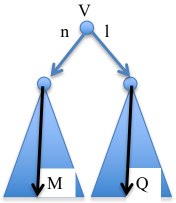
Solution: We proceed by recursion, working from the leaves towards the root. Consult the figure below.



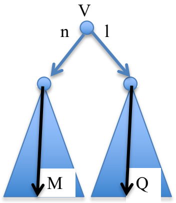
- We start with the edges connecting two adjacent leaves. Clearly, they have to be of the same length.
- Thus, referring to the figure, we let $M = \max\{k, m\}$ and $Q = \max\{p, q\}$ and change the shorter length of the two to the value of the longer length.
- Moving one level up, we consider $M +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



- 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.



- 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. 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.

3. 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.

4. 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.

4. 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.

4. 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.

4. 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.

4. 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.

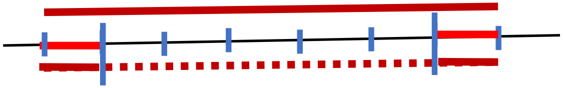
Solution:

- 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.

5. 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:

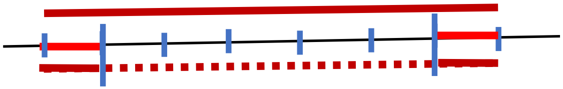
- First, cover all roofless stalls with a single board that stretches from the first roofless stall to the last roofless stall.
- Then, repeat the following up to 10 times: Find the longest line of consecutive roofed stalls that are covered by a board, and then excise that part of the board, replacing it with two boards.



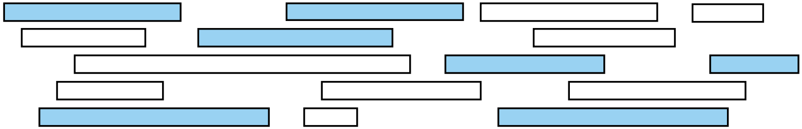
5. There is a line of 111 stalls, some of which lack a roof and need to be covered with boards. You can use up to 11 boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible.

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.



6. 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.



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.

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.

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.

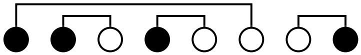
Solution:

- 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.

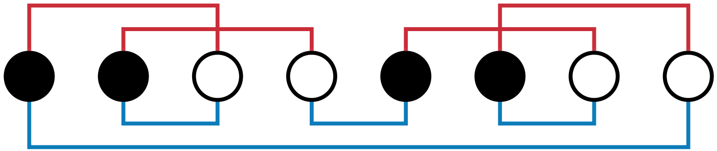
Solution:

- 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.

7. 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.

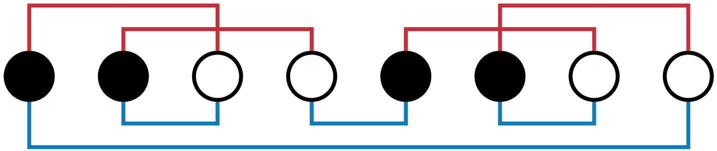


Solution: One should be careful about what kind of greedy strategy one uses. For example, connecting the closest pairs of equally coloured dots produces suboptimal solution as the following example shows:



- 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.

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.

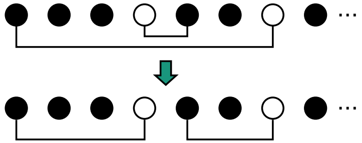
- To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots.
- Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost 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 (shown on the next slide) - 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.

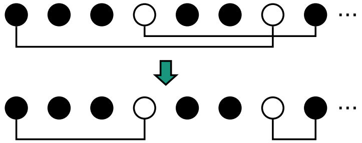
- To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots.
- Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost 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 (shown on the next slide) - 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.

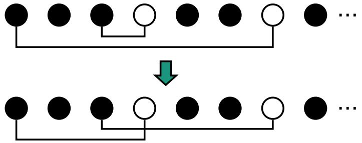
- To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots.
- Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost 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 (shown on the next slide) - 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.

- 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 (shown on the next slide) - 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.

- 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 (shown on the next slide) - 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.







8. 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. 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 $i + 1$.

8. 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. 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 $i + 1$.

9. 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, $0 \leq m \leq k$.

- 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, $0 \leq m \leq k$.

- 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, $0 \leq m \leq k$.

- 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, $0 \leq m \leq k$.

- 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, $0 \leq m \leq k$.

- 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 $O(n \log n + n^2) = O(n^2)$.

- 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 $S = \{J_1, J_2, \dots, J_k\}$ 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 $O(n \log n + n^2) = O(n^2)$.

- 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 $O(n \log n + n^2) = O(n^2)$.

- 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 $O(n \log n + n^2) = O(n^2)$.

- 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 $S = \{J_1, J_2, \dots, J_k\}$ 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 $O(n \log n + n^2) = O(n^2)$.

- 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 $O(n \log n + n^2) = O(n^2)$.

10. 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.

10. 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

11. 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.

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.

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.

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.

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.

12. 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.

12. 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.

12. 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.

12. 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.

12. 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.

13. 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.

Solution: This problem is equivalent to problem 9 about the processor. Convert each arc into a time period (by, for example, superimposing a 24-hour clock onto the unit circle) and proceed as in problem 9.

13. 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.

Solution: This problem is equivalent to problem 9 about the processor. Convert each arc into a time period (by, for example, superimposing a 24-hour clock onto the unit circle) and proceed as in problem 9.

14. 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.

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.

14. 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.

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.

14. 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.

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.

14. 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.

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.

- 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.

- 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.

15. 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 kilometres of it. Design an algorithm to determine the minimum number of police stations that would need to be built.

Solution: This problem is essentially identical to the problem of placing minimal numbers of base station towers which we did in class.

15. 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 kilometres of it. Design an algorithm to determine the minimum number of police stations that would need to be built.

Solution: This problem is essentially identical to the problem of placing minimal numbers of base station towers which we did in class.

16. 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.

16. 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.

16. 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.

16. 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.

16. 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.

17. 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.

17. 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.

17. 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.