COMP3121/9101: Algorithm Design and Analysis

Problem Set 3 – The Greedy Method

[K] – key questions **[H]** – harder questions **[E]** – extended questions

☆ – contains sample solutions

Contents

1. Section One: Ordering

☆ **[K] Exercise 1.**

(a) 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.

(b) Justify the correctness of your algorithm.

[K] Exercise 2. Given two sequences of letters A and B, find if B is a sub-sequence of A in the sense that one can delete some letters from A and obtain the sequence B. Then prove that your algorithm is correct.

[K] Exercise 3. After the success of your latest research project in mythical DNA, you have gained the attention of a most diabolical creature: Medusa. Medusa has snakes instead of hair. Each of her snakes' DNA is represented by an uppercase string of letters. Each letter is one of S, N, A, K or E. Your extensive research shows that a snake's venom level depends on its DNA. A snake has venom level x if its DNA:

- has exactly $5x$ letters
- begins with x copies of S
- then has x copies of N
- then has x copies of A
- then has x copies of K
- ends with x copies of E.

For example, a snake with venom level 1 has DNA SNAKE, while a snake that has venom level 3 has DNA SSSNNNAAAKKKEEE. If a snake's DNA does not fit the format described above, it has a venom level of 0. Medusa would like your help making her snakes venomous, by deleting zero or more letters from their DNA. Given a snake's DNA of length n, design an $O(n \log n)$ algorithm to determine the maximum venom level the snake could have.

Hint. *Combine greedy with binary search.*

2. Section Two: Selection

☆ **[K] Exercise 4.** Consider the following problem.

Given a set of denominations and a value V, find the minimum number of coins that add to give V.

Consider the following greedy algorithm.

Algorithm: Pick the largest denomination coin which is not greater than the remaining amount. Since we always choose the largest amount on each iteration, we minimise the number of coins required.

Provide a counter example to the algorithm to show that greedy does not always yield the best solution.

☆ **[K] Exercise 5.**

(a) 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 $O(n^2)$ algorithm to find the minimum number of courses you will need to take to get an IQ of at least K.

(b) Justify the correctness of your algorithm.

[K] Exercise 6.

(a) 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 $n, A, B, a[1..n]$ and $b[1..n]$, you have to determine the **maximum** total amount of money you can earn in $O(n \log n)$ time.

(b) Justify the correctness of your algorithm using a greedy stays ahead argument.

[K] Exercise 7. 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.

[K] Exercise 8. 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 that runs in linear time complexity for which, given any amount, makes that amount using a minimal number of coins.

☆ **[E] Exercise 9.** Let A be a $2 \times n$ array of positive real numbers such that the elements in each column sum to 1. Design an $O(n \log n)$ algorithm to pick one number in each column such that the sum of the numbers chosen in each row never exceeds $\frac{n+1}{4}$.

For example, consider the following 2×8 array.

0.4	0.7	0.9	0.2	0.6	0.4	0.3	0.1
0.6	0.3	0.1	0.8	0.4	0.6	0.7	0.9

We can choose the following configuration (the configuration is highlighted in red).

0.4	0.7	0.9	**0.2**	**0.6**	**0.4**	**0.3**	**0.1**
0.6	**0.3**	**0.1**	0.8	0.4	0.6	0.7	0.9

The chosen numbers on the first row give us

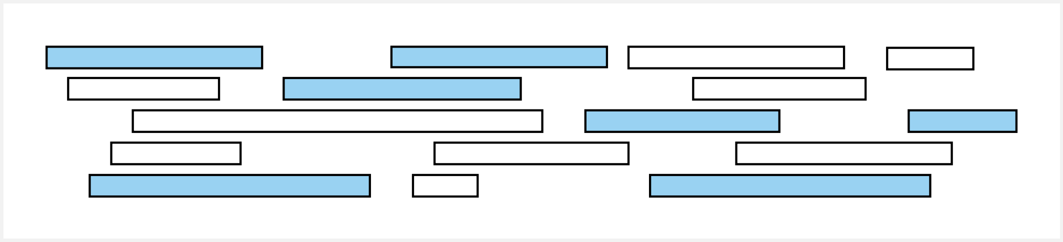
$$0.4 + 0.2 + 0.6 + 0.4 + 0.3 + 0.1 = 2 < 2.25 = \frac{9}{4},$$

while the chosen numbers on the second row give us

$$0.3 + 0.1 = 0.4 < 2.25 = \frac{9}{4}.$$

3. Section Three: Scheduling and Graphs

☆ [**K**] **Exercise 10**. 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 in $O(n^2)$ time. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, representing the left and right endpoints of the intervals in X.



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

[**K**] **Exercise 11**. 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, $\sum_{i=0}^{n} w_i C_i$. Design an $O(n \log n)$ 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, $\sum_{i=0}^{n} w_i C_i$.

[**K**] **Exercise 12**. 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, and state the time complexity of the algorithm.

[**K**] **Exercise 13**. 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.)

(a) 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. Design an $O(n^2)$ 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.

(b) Suppose that now for each of the n jobs given, an inspector is required to check the operation of the processor at any point during its time of operation at least once, however, you do not know which subset of the jobs will be chosen. Therefore, design an algorithm that finds the minimal and valid list of time stamps for the inspector to come and check the operation. You may assume that each inspection can be done instantaneously.

☆ **[K] Exercise 14**. 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 $O(n \log n)$.

[K] Exercise 15. You are given a connected graph with weighted edges with all weights distinct. Prove that such a graph has a unique minimum spanning tree.

[H] Exercise 16. There are n stepping stones and each stepping stone has an associated positive integer, which represents the maximum number of stones you can skip past. You initially start at the first stone.

(a) Show that it is *always* possible to arrive at the last stone.

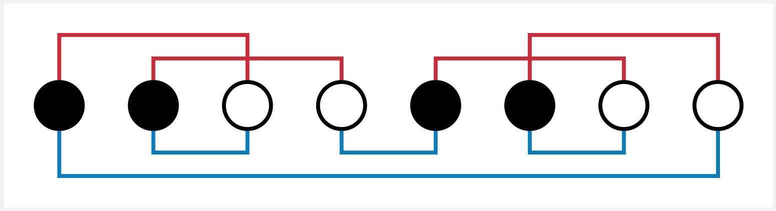
(b) You now want to arrive at the last stone with the smallest number of jumps. Construct a greedy algorithm to obtain the sequence of jumps that minimises the number of jumps, clearly outlining what the greedy heuristic is.

(c) Justify the correctness of your algorithm.

4. Selected Solutions

Exercise 1.

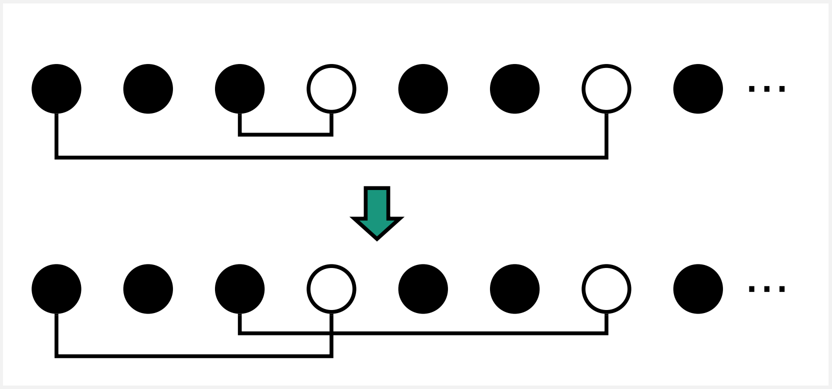
(a) One should be careful about what kind of greedy strategy one uses. For example, connecting the closest pairs of equally coloured dots produces suboptimal solution as the following example shows:



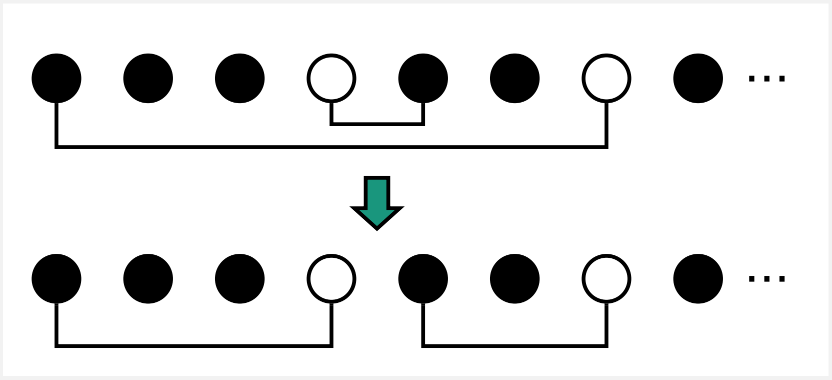
Connecting the closest pairs (blue lines) uses 3 + 7 = 10 units of length while the connections in red use only 4×2 = 8 units of length.

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. By storing pointers to the leftmost dots and updating it as we go, the overall complexity of our algorithm is $O(n)$.

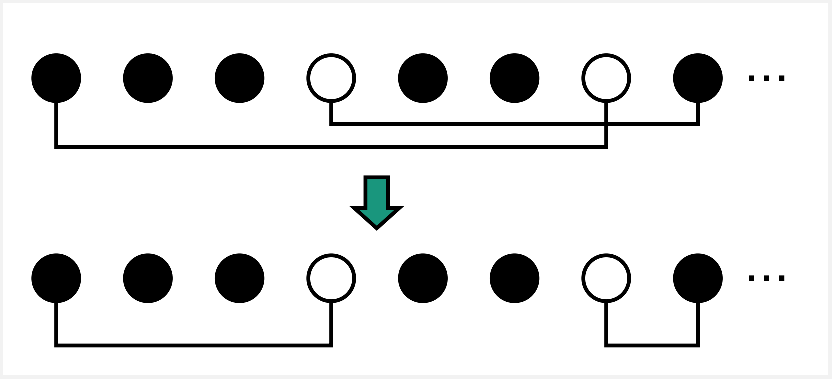
(b) To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots. Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost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



Configuration I: W_1's partner is before W_1. Total wire is unchanged.



Configuration II: W_1's partner is between W_1 and B_1's partner. Total wire decreases, as we avoid the doubled wire between W_1 and its partner.



Configuration III: W_1's partner is after B_1's partner. Total wire decreases, as we avoid the doubled wire between W_1 and B_1's partner.

FIGURE 1. Three configurations in which the first dot B_1 is not paired with the first white dot W_1.

Exercise 2. 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 sub-sequence of A.

To show that this method is optimal, suppose for cases where B is a sub-sequence of A with $|B| = m \leq |A| = n$. Then let $S \subset \mathbb{Z}_n$ with $|S| = m$ such that $S_i = j$ indicating that $A[j] = B[i]$ which represents a specific selection of marks. Suppose there exists some S that disagrees with the generated marks of our greedy solution (denoted S'), then let j be the first index such that S and S' differs. This indicates that there must exist some $k < j$ such that we could have picked $S_i = k$ while still keeping S valid. Then by continuing with this manner, we can eventually change S to adhere to S'.

As we only linear scan through A or B once, the total complexity of our solution is effectively $O(\max(n, m))$.

Exercise 3. Given a DNA string of length n, our goal is to find the biggest subsequence of 'SNAKE's that occur in our original string in time complexity $O(n \log n)$. The key insight behind this problem is the way we apply a binary search, which we will also prove optimality with.

To begin, consider the length n string that encompasses the DNA and define L to be $\min\{n_S, n_N, n_A, n_K, n_E\}$ where n_i is the number of letter occurrences in the DNA string. Then define an array of snake variations such that

$$\text{snake} = \left[SNAKE, SSNNAAKKEE, \ldots, \underbrace{S \ldots S}_{L} \underbrace{N \ldots N}_{L} \underbrace{A \ldots A}_{L} \underbrace{K \ldots K}_{L} \underbrace{E \ldots E}_{L} \right].$$

Notice that if a snake pattern with $L = k$ **fails**, then so does $L = k + m$ for all $m \geq 0$. Conversely, if a snake pattern with $L = \ell$ succeeds, then it is redundant to check all venom levels $L = \ell - m$ for all $m \geq 0$ since venom level is *at least* size ℓ. Combining these two results gives us a way to apply a binary search to make the algorithm more efficient.

Therefore, the algorithm works as follows: find the median term and perform the greedy check by checking for a subsequence of venom level $L/2$. If the check is successful, then we proceed with a binary search on the upper half of the array. If the check is unsuccessful, then we proceed with a binary search on the lower half of the array and we repeat this process.

The binary search takes $\log n$ many iterations since, in each iteration, the array gets divided into two parts. Additionally, in each check, the greedy subsequence check takes n many steps since it has to step through the entire DNA string.

We shall proceed with a "Greedy stays ahead" (inductive) proof. Consider the base case of an empty string. It is clear that the size of such a string is zero and hence, the maximum possible venom level must also be zero. This is clear in that the binary search will return an empty array in which case, it will return the maximum venom level stored which is zero.

Now suppose that our solution is *as good* as the optimal solution at some point k; that is, for all iterations $n < k$, the algorithm produces an optimal solution. Consider the next iteration $(k + 1)$. One of two events can occur.

- Either the new letter appended to the string changes the maximum venom level, or
- The new letter appended to the string does not affect the maximum venom level.

We will show that, regardless of which case, the binary and greedy search will produce the correct venom level.

Claim. Let T be a string of k letters. For some integers $a \leq b \leq c \leq d \leq e < \lfloor k/5 \rfloor$, suppose there exist a S's, followed by b N's, followed by c A's, followed by d K's and followed by e E's in T. Then the addition of a new letter will return the same venom level $\min\{a, b, c, d, e\} = a$.

Proof. Here, we assume that the string can consist of any letters in between any two S's, N's, A's, K's or E's. For example, the string `NNSNAAKESE` satisfies T since there exist 1 `S`, followed by 1 `N`, followed by 1 `K` and finally 1 `E`. Further, the binary search will guarantee that if the greedy check at some instance a fails, then so does $a + m$ for any $m > 0$.

Now suppose that a letter is appended to the string T. Since the minimum number of S's precedes all other letters, then we cannot obtain any subsequence of the form

$$(\text{for any } m > 1) \qquad \underbrace{\texttt{S...S}}_{a+m}\underbrace{\texttt{N...N}}_{a+m}\underbrace{\texttt{A...A}}_{a+m}\underbrace{\texttt{K ...K}}_{a+m}\underbrace{\texttt{E ...E}}_{a+m}$$

Since there are a minimum of a `S`'s, then any additional letter appended to the string will return the same venom level. □

Claim. Let T be a string of k letters. For some integers $\lfloor k/5 \rfloor \geq a \geq b \geq c \geq d > e$, suppose there exist a S's, followed by b N's, followed by c A's, followed by d K's and followed by $(e-1)$ E's in T. Then the addition of a new letter E will return a new maximum venom level $\min\{a, b, c, d, e\} = e$. Otherwise, it will return the same venom level.

Proof. In a similar light to the previous lemma, we know that the letter `E` is at a minimum. We make a similar assumption to the previous claim in that there can be finitely many letters between any two consecutive letters. For example, the following string `NNSSSSNANNAAAKKE` is a valid string in T since there are 4 `S`'s followed by 3 `N`'s, followed by 3 `A`'s, followed by 2 `K`'s and finally 1 `E`. Further, the binary search guarantees that, if the greedy search is successful at some instance a, then it will also work for all instances $a - m$ for all $m > 0$. And so, checking strings with $a - m$ `S`'s, `N`'s, etc. won't affect the venom level.

Then the addition of a new letter will either be an `E` in which case applying the binary search will yield a new maximum since we attain a new minimum of e. Since the greedy search will return true for e `S`'s, `N`'s, etc. then it will also work for all instances before it. Any other letter appended will return the same venom level since we still attain $(e-1)$ `E`'s. □

From these two claims, one can continuously construct strings of letters and the algorithm will continuously produce the maximum venom level by continually applying a binary search on the number of `S`'s, `N`'s, `A`'s, `K`'s and `E`'s, and then applying greedy search to check whether this is successful or not. Since we apply a binary search on the index, then the binary search will have a depth of $\log(n/5) = \log n - \log 5 = O(\log n)$, with each depth being a $O(n)$ greedy search. Hence the entire algorithm will run in $O(\underbrace{n + n + \cdots + n}_{\log n}) = O(n \log n)$.

Exercise 4. Consider the following set of denominations $S = \{1, 2, 6, 8\}$ and $V = 12$. The greedy solution would choose the following: $(8, 2, 1, 1)$ whereas the optimal solution would choose $(6, 6)$. As an aside, this problem is a variant of the *knapsack* problem which will be introduced in the dynamic programming section of the course.

Exercise 5.

(a) The problem can be solved by considering all the courses remaining that you can take, then take the course that will raise your IQ the most. Repeat until your IQ is K or higher.

We can then sort the courses based on the required IQ in $O(n \log n)$. The main selection process will require a linear search of $O(n)$ for each selected course; hence, the total complexity of the algorithm yields $O(n^2)$.

(b) Since our greedy solution considers all of the courses that can be taken, it is clear that the greedy solution chooses the course that raises the IQ the most which implies that our greedy solution is correct for the first course.

Now, let $\mathcal{G} = (g_1, \ldots, g_k)$ be the first k courses that our greedy solution chooses, and let $\mathcal{O} = (o_1, \ldots, o_k)$ be the first k courses that any solution chooses. We assume that our greedy solution has chosen the k courses in a way such that the IQ is at least as high as the IQ chosen by $\mathcal{O}$. Consider all of the courses that are considered by $\mathcal{O}$. Since our greedy solution chooses courses such that the first k courses form an IQ at least as high as the courses chosen by $\mathcal{O}$, the courses considered by $\mathcal{G}$ also consider all of the courses chosen by $\mathcal{O}$. Since our greedy solution always picks the course that maximises the IQ, it follows that the next course raises our IQ as high as any choice that $\mathcal{O}$ makes. This argument can be repeated at each iteration, which implies that our greedy solution is also optimal.

Exercise 6.

(a) Let $d[i] = |a[i] - b[i]|$. Using MERGESORT, sort all $d[i]$ in decreasing order and re-index all items such that $|a[1]-b[1]|$ is the largest difference (i.e. the ith item such that $d[i] = |a[i]-b[i]|$ is the ith difference in size).

We now go through the list giving the ith 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.

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 lastly takes $O(n)$ time. Thus the total run time complexity is $O(n \log n)$.

(b) Clearly, the algorithm is optimal for the first item. Let $\mathcal{G} = (g_1, \dots, g_k)$ be the choices of selling item the first k items to either Alice or Bob made by our greedy algorithm and let $\mathcal{O} = (o_1, \dots, o_k)$ be the choices made by any arbitrary solution. Furthermore, assume that our greedy solution makes just as much as profit as $\mathcal{O}$.

Consider the $(k+1)$th item. If $o_{k+1} = g_{k+1}$, then there is nothing left to prove and our greedy solution is just as good as any solution. Therefore, we may assume that $o_{k+1} \neq g_{k+1}$. Since our greedy solution picks the person who would pay a higher price for the $(k+1)$th item, any deviation from our strategy results in a suboptimal profit unless they are unavailable to buy the item. Therefore, if there are no options available, the greedy solution will resort to picking the person who pays the minimum. If our greedy solution chose the person who would pay the smaller amount and $\mathcal{O}$ picked the person paying the higher amount, this implies that there was a previous iteration where the two solutions did not agree. However, since our greedy solution always chooses the higher bidder, the other solution must have been suboptimal by the way that we've sorted each item. This is because any future iteration would result in a smaller difference in profit by choosing Alice over Bob and vice versa; therefore, such a solution must have played suboptimally. In all cases, our solution is always just as optimal as any solution at the $(k+1)$th iteration. Repeating this argument at every iteration shows that our greedy solution is optimal.

Exercise 7. To solve this problem, we proceed to 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, we assume the opposite - that is, suppose that for a certain amount M, there is an optimal way of payment that is more efficient than the one described by the greedy algorithm. 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 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 coin. Thus, after the option of giving a \$1 coin has been exhausted we are left with at least \$1 to be given without using any \$1 coins. For the same reason at most one 50c coin can be given and we are left with at least 50 cents to be given without using any 50c coins. Note that at most two 20c coins can be used because three 20c coins can be replaced with a 50c and 10c coin. Also note that if two 20c coins are used, no 10c coins can be used because two 20c coins and a 10c coin can be replaced with a single 50c coin.

Thus, if two 20c coins are used, only 5c coins can be used to give the remaining amount of at least 10 cents, which would require two 5c coins, but these could be replaced by a single 10c coin, contradicting optimality. If, on the other hand, only one 20c coin is used to give an amount of at least 50 cents we are left with at least 30 cents to be given using 10c and 5c coins. Note that in such a case only one 10c coin can be used because two 10c coins can be replaced by a 20c coin. So we are left an amount of at least 20 cents to be given with 5c coins only. However, only one 5c coin can be used because two 5c coins can be replaced by one 10c 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.

In terms of time complexity, note that our algorithm at each step only needs to determine the largest $k \in \mathbb{Z}^+$ s.t. $kd \leq M$ with d being the value of the current denomination. This is achievable in $O(1)$ as we just need to compute $\lfloor M/d \rfloor$ and then move on to the next denomination. Therefore, our algorithm runs in $O(m)$ time for m is the number of denominations we have.

Exercise 8. 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.

For each denomination, the most we can use is all of the n coins with $O(\log n)$ to search for each denomination value. Therefore the total complexity is $O(n \log n)$.

Exercise 9. The algorithm itself is actually quite natural; the difficulty is in the proof of correctness. One such trivial algorithm might be to always choose the smallest of the two numbers in each column. But this won't work for arrays where all of the elements in one row is 0.4. So we need to be slightly smarter.

Denote the element in the first row and ith column by a_i. Similarly, denote the element in the second row and ith column by $b_i = 1 - a_i$. We can apply merge sort to sort one of the rows. Without the loss of generality, sort the first row in $O(n \log n)$ time. Hence, we can assume that $a_1 \leq a_2 \leq \cdots \leq a_n$ is a non-decreasing sequence of numbers. But this implies that the sequence in the second row is non-increasing. We can show this very easily by observing that, if $a_i \leq a_j$, then $-a_i \geq -a_j$ and $b_i = 1 - a_i \geq 1 - a_j = b_j$. Hence, $b_1 \geq b_2 \geq \cdots \geq b_n$. At every iteration, we take a_i until the sum exceeds $\frac{n+1}{4}$. In other words, we choose $\{a_1, a_2, \ldots, a_k\}$ so that

$$a_1 + a_2 + \cdots + a_k \leq \frac{n+1}{4},$$

but

$$a_1 + a_2 + \cdots + a_k + a_{k+1} > \frac{n+1}{4}.$$

For the remaining columns, the algorithm chooses $\{b_{k+1}, \ldots, b_n\}$.

We claim that this provides the correct output on any $2 \times n$ array. Half of the proof is trivial since, by construction, the sum of the values in $\{a_1, a_2, \ldots, a_k\}$ should never exceed $\frac{n+1}{4}$. It, therefore, suffices to show that the choices in the second row also never exceed $\frac{n+1}{4}$.

To this end, recall that $\{b_m\}$ is a sequence of non-increasing numbers. Thus, we always have that

$$b_{k+1} + b_{k+2} + \cdots + b_n \le \underbrace{b_{k+1} + b_{k+1} + b_{k+1} + \cdots + b_{k+1}}_{n-k \text{ terms}} = (n-k)b_{k+1}.$$

We now estimate b_{k+1}. Observe that a_k is the largest value chosen in the top row. Thus, a_{k+1} is *at least* the average of $a_1, a_2, \ldots, a_k, a_{k+1}$. Thus, we have

$$a_{k+1} \ge \frac{a_1 + a_2 + \cdots + a_{k+1}}{k+1} > \frac{n+1}{4(k+1)}$$

since $a_1 + \cdots + a_{k+1} > \dfrac{n+1}{4}$. This gives us an upper bound on b_{k+1}, namely

$$b_{k+1} = 1 - a_{k+1} < 1 - \frac{n+1}{4(k+1)}.$$

So the sum of the chosen numbers can be loosely bounded above by the following:

$$b_{k+1} + b_{k+2} + \cdots + b_n \le (n-k)b_{k+1} = (n-k)\left(1 - \frac{n+1}{4(k+1)}\right).$$

Let $f_n(k) = (n-k)\left(1 - \dfrac{n+1}{4(k+1)}\right)$ where n is some fixed integer. After some computation, we see that

$$\frac{d}{dk}(f_n(k)) = -1 + \frac{(n+1)^2}{4(k+1)^2},$$

which implies that the critical point occurs at $k_{\max} = \dfrac{n-1}{2} > 0$. One can show that this is indeed the global maximum. Substituting $k_{\max}$ into $f_n(k)$, one can verify that

$$f_n(k_{\max}) = \frac{n+1}{4}.$$

Hence, for any fixed n, it follows that regardless of what k is, we have

$$b_{k+1} + b_{k+2} + \cdots + b_n \le f_n(k_{\max}) = \frac{n+1}{4},$$

which completes the proof.

Exercise 10. 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 (as long as this right endpoint is beyond the right endpoint of the last chosen interval). If there are no such intervals, move on to the next smallest left endpoint and pick again as in the first step. Continue in this manner until an interval with the absolute largest right endpoint is chosen. Note that the right endpoints of the chosen intervals also form an increasing subsequence of R.

For the optimality of the algorithm, let $G = [g_1, g_2, \ldots, g_m]$ be the tiling cover generated by our greedy strategy and let $A = [a_1, a_2, \ldots, a_n]$ be an alternative tiling cover, both listed in increasing order of left endpoint. Suppose index i is the first point of difference, i.e. $A = [g_1, g_2, \ldots, g_{i-1}, a_i, a_{i+1}, \ldots, a_n]$ where $g_i \ne a_i$.

- $L(g_i)$ must be greater than or equal to $L(g_{i-1})$ by the sort order, and furthermore they must be unequal as otherwise the greedy algorithm would only have taken one of these two intervals. Therefore $L(g_i) > L(g_{i-1})$.

- If $L(g_i) \leq R(g_{i-1})$, i.e. interval g_i has left endpoint within interval g_{i-1}, then $R(g_i) > R(g_{i-1})$, as otherwise the greedy algorithm would not have taken interval g_i.

  Now, if $L(a_i) > R(g_{i-1})$, i.e. interval a_i does not overlap with interval g_{i-1}, then we can choose a small ϵ so that $x = R(g_{i-1}) + \epsilon$ belongs to interval g_i but not to any interval in A (since g_{i-1} has the greatest right endpoint of all prior chosen intervals, and a_i has the least left endpoint of all subsequently chosen intervals). This means that A is not a tiling cover, which is a contradiction.

  Therefore interval a_i must overlap with interval g_{i-1}. Of all such intervals, g_i has the largest right endpoint, so swapping a_i out for g_i gives a tiling cover $A' = [g_1, g_2, \ldots, g_{i-1}, g_i, a_{i+1}, \ldots, a_n]$ which uses the same number of intervals as A.

- On the other hand, if $L(g_i) > R(g_{i-1}$, i.e. interval g_i does not overlap with interval g_{i-1}, then by the construction used in the greedy algorithm, we know that no intervals with left endpoint within g_{i-1} have right endpoint beyond $R(g_{i-1})$, and that of all subsequent intervals, g_i starts earliest, and finishes latest out of all such intervals.

  Now, if $L(a_i) > L(g_i)$, then $x = L(g_i)$ would not be covered by any interval in A, so A would not be a tiling cover (a contradiction). Therefore $L(a_i) = L(g_i)$. It follows that again $A' = [g_1, g_2, \ldots, g_{i-1}, g_i, a_{i+1}, \ldots, a_n]$ is a tiling cover which uses the same number of intervals as A.

Continuing thus, we can resolve every difference between the greedy selection G and the alternative selection A, leaving only potentially several unnecessary intervals in A. Thus the greedy strategy uses as few intervals as possible, i.e. it is optimal.

The algorithm involves sorting the intervals in $O(n \log n)$ and then linearly searching for the correct interval to pick next resulting in a total worst-case complexity of $O(n^2)$.

Exercise 11. 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 $S = \sum_{i=0}^{n} w_i C_i$ 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 $\sum_{i=0}^{n} w_i C_i$. 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 $\sum_{i=0}^{n} w_i C_i$ change: before the swap this sum contains $w_i(T + t_i) + w_{i+1}(T + t_i + t_{i+1})$, 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,

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

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

In terms of time complexity, we can compute w_i/t_i for all n elements in $O(n)$ time and then sort all those values in $O(n \log n)$ using MergeSort to produce the final schedule. Hence the total complexity is $O(n \log n)$.

Exercise 12. We proceed with the problem by first sorting both the students and job requirements in skills and level of skill requirement. Then we assign each student the job with the highest matching skill requirement that hasn't been already assigned. Then the number of assigned jobs will be the maximum number of jobs that can be successfully assigned.

To prove that this is optimal, let us introduce some notations. 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:

Claim. 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.

Claim. The greedy strategy assigns student s_1 the job j_1, but the alternative strategy does not assign s_1 any job.

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.

Hence, 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.

In terms of time complexity, we sort both jobs and students in $O(n \log n)$, then the greedy selection process can be done in $O(\log n)$ time via a sorted stack. Therefore the worst-case complexity is $O(n \log n)$ in total.

Exercise 13.

(a) This part 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 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.

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

(b) Note that as we do not know which jobs will be selected to execute, we must produce a schedule such that the time range of every job contains at least one time stamp. To solve this problem, we again visualize the problem of intervals on a 24hr clock. To avoid confusion, we will imagine that the jobs are laid out in a straight line. Let $S = \{J_1, J_2, \ldots, J_n\}$ be the set of jobs sorted by their finish time on the line. From now we refer to a job J_i being stabbed if there exists a picked timestamp within its scheduled duration.

We start by picking the time stamp right at the finish time of J_1. Then, we consider the jobs that have not been stabbed and pick the job whose starting time occurs the earliest (with respect to the last picked time stamp) and stab it at its right endpoint. Repeat in this manner until all jobs have been stabbed, and then record the number of needles used as n_1. Now start over, but this time begin by stabbing J_2 at its right endpoint, and then continue in the above manner while wrapping around the end of the line and recording the number of needles used as n_2. Do this for each J_i, and then take the arrangement for which the number of needles used is minimal for the final solution.

The optimality of this problem again follows a very similar logic from part (a) and the optimality of the *Interval Stabbing Problem.*

Sorting all jobs in their finish time takes $O(n \log n)$ and then for each J_i it takes $O(n)$ to compute n_i (going through each job and picking the position of the timestamp), therefore the total complexity will be $O(n^2)$.

Exercise 14. 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 $O(n \log n)$.

The proof of correctness is constructed analogously to the proof of correctness for Kruskal's algorithm (via the exchange argument). The result of Kruskal's algorithm ensures that the resulting tree is minimal. By performing Kruskal's algorithm on the selected $n-1$ edges of the old spanning tree in addition to the new edges, the construction of such a tree ensures that contains the additional vertex.

Exercise 15. 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.

Exercise 16.

(a) At each stone, we can always step to the next adjacent stone since each stepping stone has value ≥ 1. This shows that it is always possible to reach the last stone.

(b) The intuition is that we never want to miss out on a potential stone that gives greater distance. Therefore, a greedy heuristic of jumping as far as you can doesn't work because you can miss out on stones which can give you more options in the long run. Come up with a counterexample to show that this is indeed a suboptimal approach in the general case.

We need to think of a smarter greedy heuristic. In particular, if I'm at index i, then I want to have arrived at the ith stone with the smallest number of jumps possible. If we can generate this, then we are able to compute the smallest number of jumps that reaches the nth stone. Therefore, our greedy approach would be to reach each stone with the smallest number of jumps; we ignore any future updates. Note that this is very closely related to the concept of *dynamic programming*. Our solution is to look at all possible stones that can be jumped at the current stone and pick the stone that gives the farthest end point.

(c) We prove that this is correct with the greedy stays ahead approach. At the first jump, we scan through all possible stones that we can and jump to the furthest stone which trivially gives us the furthest distance.

Now, let our greedy solution be $\mathcal{G} = (g_1, \dots, g_k)$, where g_i defines the stone we jump to on the ith jump. Similarly, let $\mathcal{O} = (o_1, \dots, o_k)$ be any arbitrary solution at the kth jump. After the kth jump, we may assume that we have covered as much distance as $\mathcal{O}$. Now, consider all of the options that is considered by $\mathcal{O}$. If the option $\mathcal{O}$ chooses at the current iteration exceeds all of the choices that $\mathcal{G}$ has at the current iteration, then this implies that $\mathcal{G}$ did not choose the stone that maximises the end point in a previous iteration which is a contradiction to the greedy solution. Therefore, any choice that $\mathcal{O}$ makes in the current iteration must result in an endpoint that is at most the distance made by $\mathcal{G}$; in other words, $\mathcal{G}$ will always make a choice that is at least as far as $\mathcal{O}$ at the $(k+1)$th jump. Repeating the same argument at each iteration shows that our greedy solution is optimal.