COMP3121/9101
Algorithm Design

Practice Problem Set 3 – Greedy Algorithms

[**K**] – key questions [**H**] – harder questions [**E**] – extended questions [**X**] – beyond the scope of this course

Contents

§ SECTION ONE: ORDERING

[K] Exercise 1.

(a) 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 runs in $O(n^2)$ which either produces the best distribution such that every robber is happy or determines that there is no such distribution.

(b) Using an exchange argument, justify the correctness of your algorithm.

- Start by supposing that there is some other solution that assigns the items to the robbers that makes all robbers happy, which is different to your greedy solution.
- By choosing suitable items, how do you justify that swapping the items between your solution and the chosen solution will not worsen your solution?
- Argue that, by repeatedly swapping your items, you can transform any arbitrarily correct solution into yours.

Solution.

(a) By applying a sorting algorithm like MergeSort, order the items in ascending order based on their values (v_i). Take the lowest value item v_1 and consider all robbers such that v_1 is in their range of acceptable values. Pick the one with the lowest upper limit and give the first item to that robber. Continue in this manner considering the item with the next smallest value v_2 and so forth. If at any point no such assignment is possible, we can determine that there is no such distribution to keep all the robbers happy.

Note that we require $O(n \log n)$ from sorting the item's value. Then for each item, we linearly search the number of robbers that satisfies the condition, resulting in a run-time complexity of $O(n^2)$. Therefore, the final complexity is $O(n^2)$ for our algorithm.

(b) We now prove that this method is optimal. For each robber 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 robber which satisfies all robbers but is different from the result produced by our greedy strategy. This means there is at least one item assignment that 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 robber i (so $v_k \in [L_i, U_i]$). Since this item assignment violated our greedy policy, there must be another robber j (with range $[L_j, U_j]$) who would have been happy with item k, but whose highest acceptable value is lower than robber i's, so $v_k \in [L_j, U_j]$ and $U_j < U_i$.

Now suppose this robber 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 robber i would be happy with item m, and we can therefore swap the assignments for the two robbers 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.

□

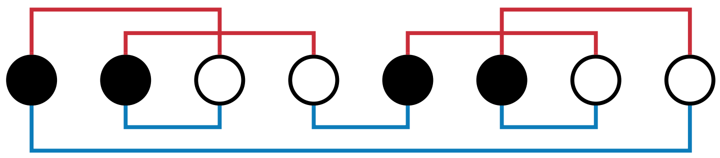
[K] Exercise 2.

(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 by using an exchange argument.

Solution.

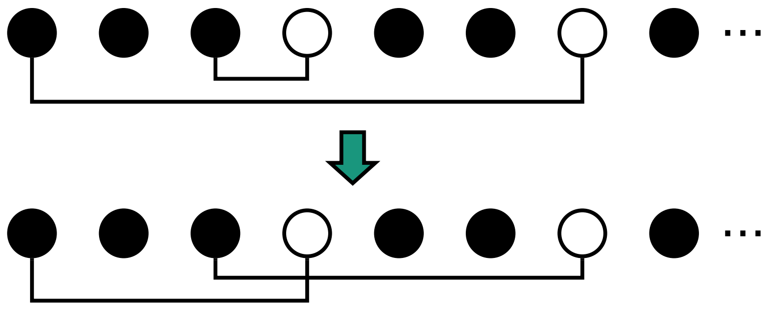
(a) One should be careful about what kind of greedy strategy one uses. For example, connecting the closest pairs of equally coloured dots produces suboptimal solution as the following example shows:



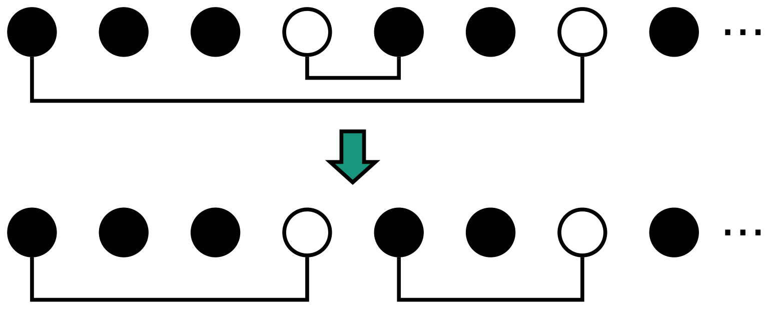
Connecting the closest pairs (blue lines) uses $3 + 7 = 10$ units of length while the connections in red use only $4 \times 2 = 8$ units of length.

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

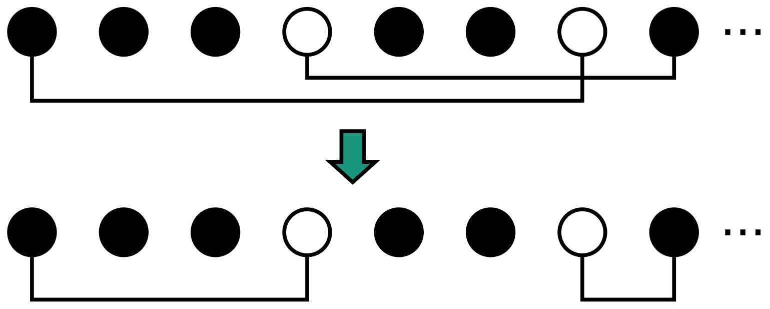
(b) To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots. Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost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

$\square$

[K] Exercise 3. 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 last payment must be \$1.

Design a $O(\log S)$ algorithm that, given a positive integer S as your loan amount, produces a payment strategy which minimises the number of months it will take you to repay your loan.

Solution. We first narrow down the structure of our payment schedule.

Claim: There is a shortest valid schedule which pays ascending amounts up to some point, then descending amounts thereafter.

Proof: Suppose we have a shortest valid schedule. We can reorder it to fit the criterion above by simply paying each different amount in the schedule $1, 2, 4, \ldots$ once each, then pay the remaining sums of each amount in decreasing order. This is valid because any valid schedule must pay each amount at least twice, with the possible exception of the largest amount. The total is the same, as is the number of months taken, so this is an equally correct solution.

Claim: We should never pay the same amount three times in a row, except potentially the largest amount.

Proof: Suppose we pay the same amount 2^i three times in the ascending portion of the sequence. We could replace the last two of these payments with a single payment of the next amount 2^{i+1}, thus saving a month. Likewise, the first two of three payments of 2^i in the descending portion could be replaced by a single payment of the previous amount 2^{i+1}.

Claim: We should never pay the largest amount four times in a row.

Proof: Suppose we pay the largest amount 2^i four times in a row. We could replace the second and third of these payments with a single payment of 2^{i+1}, thus saving a month.

Claim: We should never pay the same amount twice in the ascending portion and twice in the descending portion.

Proof: Suppose we pay $\ldots, 2^i, 2^i, 2^{i+1}, \ldots, 2^{i+1}, 2^i, 2^i, \ldots$. We can replace both red terms with a single payment of 2^{i+1} to get the schedule $\ldots, 2^i, 2^{i+1}, 2^{i+1}, \ldots, 2^{i+1}, 2^i, \ldots$, thus saving a month.

Therefore, we restrict ourselves to schedules of the form

$$1, 2, 4, \ldots, 2^{k-1}, 2^k(, 2^k, 2^k), 2^{k-1}(, 2^{k-1}), \ldots, 4(, 4), 2(, 2), 1(, 1),$$

where bracketed terms are optional.

Claim: Given S, the largest amount k in such a schedule is unique.

Proof: In a schedule with largest payment 2^{k-1}, the total is up to

$$\begin{aligned}&1 + 2 + 4 + \ldots + 2^{k-1} + 2^{k-1} + 2^{k-1} + 2^{k-1} + \ldots + 4 + 4 + 2 + 2 + 1 + 1\\ &= 3(1 + 2 + 4 + \ldots + 2^{k-1})\\ &= 3(2^k - 1)\\ &= 3 \times 2^k - 3,\end{aligned}$$

whereas in a schedule with largest payment 2^k, the total is at least

$$\begin{aligned}&1 + 2 + 4 + \ldots + 2^{k-1} + 2^k + 2^{k-1} + \ldots + 4 + 2 + 1\\ &= 2(1 + 2 + 4 + \ldots + 2^{k-1}) + 2^k\\ &= 2(2^k - 1) + 2^k\\ &= 3 \times 2^k - 2,\end{aligned}$$

so there is no overlap between the amounts that can be made, completing the proof.

Now we can present the algorithm.

- Find the largest k such that $3 \times 2^k - 2 \le S$, i.e. $k = \left\lfloor \log_2 \frac{S+2}{3} \right\rfloor$. The largest payment in the schedule will be 2^k.
- Now, each amount from 1 to 2^{k-1} will be paid at least twice each, and 2^k will be paid at least once. Make an array $Q[0..k] = [2, \dots, 2, 1]$ representing the number of payments of each power of two. This leaves $S - (3 \times 2^k - 2)$ unaccounted for, which we will call R.
- If R is odd, we must make a third payment of 1. Subtract 1 from R and update $Q[0]$ to 3. Now divide R by 2, and repeat to decide whether third payments of each amount $2, 4, \dots, 2^{k-1}$ are required, updating Q accordingly.
- The remaining value R will be either zero, one or two, representing the number of additional payments of 2^k required, and hence the amount by which to increment $Q[k]$.
- Finally, write out the schedule in the form above, deciding whether to include the optional bracketed terms depending on the values in Q.

The first step takes $O(1)$ using real arithmetic, but even a binary search only takes $O(\log S)$. Each subsequent step takes $O(k)$ time. Since $k = O(\log S)$, the time complexity is $O(\log S)$. □

[H] Exercise 4. 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.

Combine greedy with binary search.

Solution. Note that this is very similar to a problem from Tutorial 1.

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

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

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 $\min\{a, b, c, d, e\} = a$.

Solution. 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

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

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.

Solution. 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 $O(\underbrace{n + n + \cdots + n}_{\log n}) = O(n \log n)$. $\square$

§ SECTION TWO: SELECTION

[K] Exercise 5. 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.

Solution. 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. □

[K] Exercise 6.

(a) 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 $O(n \log n)$ algorithm to determine the fewest number of days you will need to guide everyone through the pass.

(b) Justify the correctness of your argument using an exchange argument.

Solution.

(a) 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). As we only need to sort the array of people and then go through them once, the total complexity is $O(n \log n)$.

(b) The optimality of this strategy should be obvious. Consider the slowest person. This person will need to be guided through the mountain pass eventually, and since you can only move as fast as the slowest person in each group, the speed of the other people in the group will not affect the speed of the group. So to save as much time as possible, you should send them with as many of the other slowest people as possible. The same argument applies to the remaining people after this group has been sent. Formally, suppose that some optimal solution O that disagrees with our greedy solution G (here we assume that Groups of O is sorted by the speed of the slowest person), we may look at the first instance where a person belongs to a different group in O (p_i) than G (p_j). Note that it cannot be the case that this person is the slowest person in the group else it violates the assumption that O is optimal. Then, we can choose to again swap the spots of p_i with p_j in O without losing optimality ($p_i > p_j$ in terms of speed). Therefore, our exchange argument is complete.

□

[K] Exercise 7.

(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 using a greedy stays ahead argument.

- The inductive proof is based on the IQ quantity; at each step, you always want to argue that the IQ from the selection of courses that your greedy algorithm takes is at least as high as any arbitrary solution.
- Start by looking at your base case and verifying that your algorithm solves the problem with 1 course.
- Define your greedy solution after k selection of courses, and define any arbitrary solution after k selection of courses too. Suppose that your greedy algorithm has chosen the k courses in such a way that the IQ is at least as high as the other solution's selection of courses.
- Show that, at the $(k+1)$th course, your greedy algorithm is also at least as good as the chosen solution.
- Conclude that your greedy algorithm is as good as any arbitrary solution, which justifies the correctness of your algorithm.

Solution.

(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.

□

[K] Exercise 8.

(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.

Solution.

(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 $\mathcal{G} = (g_1, \ldots, g_k)$ be the choices of selling item the first k items to either Alice or Bob made by our greedy algorithm and let $\mathcal{O} = (o_1, \ldots, o_k)$ 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.

□

[K] Exercise 9. Assume that you have an unlimited number of \$2, \$1, 50c, 20c, 10c and 5c coins to pay for your lunch. Design an algorithm that, given the cost that is a multiple of 5c, makes that amount using a minimal number of coins.

Solution. 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. □

[K] Exercise 10. 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.

How does this problem differ to Exercise 21?

Solution. As in the previous problem, keep giving the coin with the largest denomination that is less than or equal to the amount remaining.

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

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 $O(n \log n)$. □

[K] Exercise 11. 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 by either using the greedy stays ahead or exchange argument.

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 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 $O(\max(n, m))$. □

[K] Exercise 12. There are N towns situated along a straight line. The location of the i'th town is given by the distance of that town from the westernmost town, d_i (so the location of the westernmost town is 0). Police stations are to be built along the line such that the i'th town has a police station within A_i kilometers of it. Design an algorithm to determine the minimum number of police stations that would need to be built.

How is this similar to Interval Stabbing?

Solution. For each town i, create the corresponding interval $[d_i - A_i, d_i + A_i]$, and then proceed as in the interval stabbing problem. The proof of correctness and time complexity come directly from the interval stabbing problem from lectures. □

[H] Exercise 13. Given a sorted integer array A and an integer n, add elements to the array such that any number between $[1, n]$ (inclusive) can be formed by the sum of some elements in the array. Design an $O(n)$ algorithm to determine the minimum number of elements to add to A.

Solution. We repeat this process until every integer in the range $[1, n]$ can be written as a sum of elements in A. The algorithm then returns the number of elements that were added into A. To prove that this algorithm is correct, we need to prove a few results.

Claim. If k is the smallest element that could not be formed by a sum of elements in A, then adding k covers the range $[1, 2k)$.

Solution. If $k > 1$ is the smallest element that could not be formed by a sum of elements in A, then clearly $[1, k)$ is completely covered. Adding k will ensure that $[1, k]$ is covered as well. This also ensures that $k + j$ for $j < k$ will also be covered by simply appending k to the sum. Thus, $k + (k - 1) = 2k - 1$ is covered as well as any integer between $k + 1$ and $2k - 1$. Thus, adding k to A ensures that $[1, 2k)$ is also covered. □

Claim. Let k be the smallest element that could not be formed by a sum of elements in A. Let $m > k$. Choosing to append m instead of k requires more integers to be appended.

Note: This claim asserts that choosing any other value other than the smallest element will result in a suboptimal algorithm – one that requires more selections of integers to append to A.

Solution. Since k cannot be formed by summing elements in A, it is easy to see that not every integer in the range $[1, k]$ can be written as a sum of elements in A. Thus, choosing any $m > k$ also ensures that k cannot be written as a sum of elements in A because $m + \ell > k$, where ℓ is the sum of any selection of elements in A. Therefore, to ensure that we also consider k, we need to include k.

From the previous claim, this covers the range $[1, 2k)$.

- If $m < 2k$, then the inclusion of m is suboptimal which proves our claim since choosing k will yield one less element to include.
- If $m = 2k$, then $[1, 2k]$ is covered with two inclusions. However, employing the strategy by the greedy algorithm will cover more ground since, for $p > k$ being the smallest element not yet covered, we can cover $[1, 2p)$ with two elements, where $2p > 2k$. This again yields a suboptimal solution.
- If $m > 2k$, then $[1, 2k) \cup \{m\}$ is covered. However, the interval is disjoint and therefore, does not cover any more than $[1, 2k)$, which yields a suboptimal solution.

In all, choosing any $m > k$ will either be no different to choosing k or worsen the overall solution, which proves the claim. □

This proves that the algorithm is optimal. To show that the algorithm runs in $O(n)$ time, observe that we traverse through the interval $[1, n]$ in $O(n)$ time in the worst case. Each operation in the query is done in $O(1)$ time since performing small summations takes constant time. Thus, the overall algorithm takes $O(n)$ time. □

[E] Exercise 14. 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

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

Solution. 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 $a_1 \le a_2 \le \cdots \le a_n$ 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 $a_i \le a_j$, then $-a_i \ge -a_j$ and $b_i = 1 - a_i \ge 1 - a_j = b_j$. Hence, $b_1 \ge b_2 \ge \cdots \ge b_n$. 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

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

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:

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

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

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

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

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

which completes the proof. □

§ SECTION THREE: SCHEDULING

[K] Exercise 15. 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.

`Problem Set 3/Section 3/tiling_path.pdf`

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

Solution. Sort the intervals in increasing order of their left endpoints. Start with the interval with the smallest left endpoint; if there are several intervals with the same such smallest left endpoint choose the one with the largest right endpoint. Then, consider all intervals whose left endpoints are in the last chosen interval, and pick the one with the largest right endpoint (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 $G = [g_1, g_2, \dots, g_m]$ be the tiling cover generated by our greedy strategy and let $A = [a_1, a_2, \dots, a_n]$ be an alternative tiling cover, both listed in increasing order of left endpoint. Suppose index i is the first point of difference, i.e. $A = [g_1, g_2, \dots, g_{i-1}, a_i, a_{i+1}, \dots, a_n]$ where $g_i \neq a_i$.

- $L(g_i)$ 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 $A' = [g_1, g_2, \dots, g_{i-1}, g_i, a_{i+1}, \dots, a_n]$ 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 $A' = [g_1, g_2, \dots, g_{i-1}, g_i, a_{i+1}, \dots, a_n]$ 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 $O(n^2)$. □

[K] Exercise 16. 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, $\sum_{i=0}^{n} w_i C_i$.

Solution. Schedule the jobs in decreasing order of w_i/t_i. This problem is very similar to the Tape Storage problem, which was covered in the lectures. To prove that this is optimal, we first introduce the concept of an inversion. We say that two jobs i and j form an inversion if job i is scheduled before job j, but $w_i/t_i < w_j/t_j$. Now consider any schedule which violates our greedy scheduling. Such a schedule will contain an inversion of at least one pair of consecutive jobs. If we repeatedly fix all such inversions between two consecutive jobs (by swapping them in the schedule) without increasing the value of $S = \sum_{i=0}^{n} w_i C_i$ 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,

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

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

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 $O(n \log n)$. □

[K] Exercise 17. 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.

Solution. 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. □

[K] Exercise 18. 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.

Solution. Those problems are closely related to example problems we have given in the lectures.

(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 $O(n^2)$.

□

[K] Exercise 19. You are given a set of n jobs where each job i has a deadline $d_i \geq 1$ and profit $p_i > 0$. Only one job can be scheduled at a time. Each job takes 1 unit of time to complete. We earn the profit if and only if the job is completed by its deadline. Design an $O(n^2)$ algorithm to find the subset of jobs that maximises the total profit.

Solution. Given a set of jobs, deadlines and profits, our goal is to construct an algorithm that produces the greatest profit in $O(n^2)$. Since we're attempting to maximise profit, sort the set of jobs by profit keeping note of its deadlines and construct a deadline array that marks all of the times in which we will perform each job. Let a particular job with deadline d_i and profit p_i be denoted as i. Since the job can be completed no later than d_i, we shall look for index i in our deadline array and see if there is already a job there. One of two events may occur:

- **Case 1**: There is a vacant spot. If there is a vacant spot, then we shall simply place the job there.
- **Case 2**: There is already a job at there. If there is already a job at index i, then we shall look for the next best option which is the index $i - 1$. If there is already a job there, then we shall keep looking until we reach the end of the array or a vacant spot. If we reach the end of the array, then we simply ignore the job and continue along the set of jobs.

In each iteration, we check *at most* $(n - 1)$ spots in our deadline array. As such, we'd have to run the deadline check at most $1 + 2 + 3 + \cdots + (n - 1) = \dfrac{n(n-1)}{2} = O\left(n^2\right)$ times. As a result, the algorithm takes $O(n^2)$.

We shall prove that this strategy is optimal with a proof by contradiction. That is, assume that there *is* a strategy O that produces more profit than our strategy X does. Further, define jobs i and j with deadlines d_i, d_j and profits $p_i < p_j$ such that job i appears in X and job j appears in O at some point k.

Since j has a greater profit, then it must have been chosen at some point before i. However, assuming that it did not appear in X, then its deadline must not have been successfully found in the deadline array. We consider the three different scenarios between i and j.

- **Case 1**: $d_j < d_i$. Since the deadline of j is less than i and the fact that j does not appear in, this is an immediate contradiction since if job i is chosen by X and job j is discarded by X, that implies that job i must have been considered before j. However, this is not possible since $p_i < p_j$.
- **Case 2**: $d_j = d_i$. Again, we arise at a contradiction since, if the two deadlines are equal, then j is searched first before i. But since j does not appear in X and i appears in X, this, again, implies that X searched for a vacancy for job i first which is not possible.
- **Case 3**: $d_j > d_i$. Since j is searched before i and j does not appear in X, then more profitable jobs have already been placed before j which could have been done before j. Thus, we keep traversing down the array until we arrive at the case where $d_j = d_i$. Since $d_i < d_j$ and i is chosen by X, then this means that there were vacant positions in positions $\leq d_i$. But since job j was searched for first, then strategy X must have found a position for job j. However, since job j is discarded by strategy X, we arise at a contradiction.

Hence, strategy O is *only* as good as strategy X and as such, our strategy is optimal which completes the proof. □

[H] Exercise 20. You are given a set X of n disjoint intervals $I_1, I_2, \ldots, I_n$ on the real line and a number $k \leq n$. Design an algorithm that produces a set Y of at most k intervals $J_1, J_2, \ldots, J_k$ which cover all intervals from X; hence

$$\bigcup_{i=1}^{n} I_i \subseteq \bigcup_{j=1}^{k} J_j$$

and such that the total length of all intervals in Y is minimal. Note that we do not require that $Y \subseteq X$, i.e., intervals J_j can be new. For example, if your set X consists of intervals $[1, 2]$, $[3, 4]$, $[8, 9]$, and $[10, 12]$ and if $k = 2$, then you should choose intervals $[1, 4]$ and $[8, 12]$ of total length $3 + 4 = 7$.

Solution. Here we borrow the idea from Exercise 17 which we start by creating a single large interval that covers all the intervals in X, and add this interval to Y. Then, until there are k intervals in Y, select the interval J_j in Y which contains the largest gap, where a gap is an interval between two consecutive intervals in X, and cut out this gap from J_j, producing two new (smaller) intervals.

To show the optimality of this algorithm, we will construct an argument with induction. We start by realizing that at $k = n$, the set $Y = X$ completely, therefore our problem is actually finding a disjoint (why disjoint?) set Y such that it covers X with minimal length. Therefore, we can show via induction that at each step of $l < k \leq n$ we make the most optimal choice to modify Y.

Starting with our base case of $l = 1$, the solution is trivial as we simply select the biggest continuous interval that covers X. Now suppose that for $l < k$ set $|Y| = l$ is optimal. Then since $l < n$, there must exist some $J_i \in Y$ that covers more than 2 intervals in X via the Pigeonhole principle and hence it must have covered all gaps in between those intervals. The most optimal and valid selection is to pick the largest gap to split the continuous intervals into 2 which proves the case for $l + 1$. Equally as valid, one may produce an exchange argument from this framework (*left as an exercise*).

One may use $O(n)$ many computations to find all the gaps and $O(n \log n)$ computation to sort them in the order of their sizes. Then the process of selection will take $O(n)$ in total to split the stored intervals, resulting in a total complexity of $O(n \log n)$. □

§ SECTION FOUR: GAMES AND GRAPHS

[K] Exercise 21. Assume that you are given a complete graph G with non-negatively weighted edges such that all weights are distinct. We now obtain another complete weighted graph G' by replacing all weights $w(i,j)$ of edges $e(i,j)$ with new weights $w(i,j)^2$.

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

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

Solution. To answer those questions we consider some properties of MST and the shortest path

(a) By considering how Kruskal's algorithm functions, it sorts the edges of the graph by weight (in ascending order) and then greedily selects and joins them until an MST is formed. Therefore, as long as the order of the edges is consistent between G and G', Kruskal will produce the same result. This is precisely the case with $w(i,j)^2$ as it is monotone over $\mathbb{N}$.

(b) This is not the case, consider for example a graph G with vertices A, B, C and D and edges $AB = 1$, $BD = 5$, $AC = 3$, $CD = 4$, $AD = 7$ and $BC = 8$. Then the shortest path from A to D in G is ABD with length 6, while the path ACD is longer, with length 7. However, after squaring all the weights, the length of ABD becomes $1 + 25 = 26$, while the length of ACD becomes $9 + 16 = 25$ which is shorter. In a more abstract notion, there is **no guarantee** that for 2 paths with respective weights of $\left\{w_1^{(1)}, w_2^{(1)}, \ldots, w_k^{(1)}\right\}$ and $\left\{w_1^{(2)}, w_2^{(2)}, \ldots, w_n^{(2)}\right\}$, we have that

$$\sum_{i=1}^{k} w_i^{(1)} < \sum_{i=1}^{n} w_i^{(2)} \implies \sum_{i=1}^{k} \left(w_i^{(1)}\right)^2 < \sum_{i=1}^{n} \left(w_i^{(2)}\right)^2$$

as the gradient of x^2 grows for bigger values.

□

[K] Exercise 22. 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 $O(n)$ 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.

Problem Set 3/Section 4/VLSI1.pdf

Solution. We proceed recursively as follows:

- Beginning with the edges that connect two adjacent leaves, these lengths have to be of the same length. Thus, we update both edges to be the maximum of the edge weights.
- Moving one level up, we then update both of the edges by incrementing the edge weight with

To understand the algorithm, consider the following example.

From the original binary tree, we compare the edges of weight k and m, and update the smaller weighted edge to $M = \max\{k, m\}$. Similarly, we compare the edges of weight p and q, and update the smaller weighted edge to $Q = \max\{p, q\}$. When we move one level up, we now consider the edge weights $n + M$ and $l + Q$. Repeating the same step, we update the smaller of the two branches and repeat this process until we hit the root of the tree.

We now prove the correctness of the algorithm. We prove two claims.

`Problem Set 3/Section 4/VLSI2.pdf`

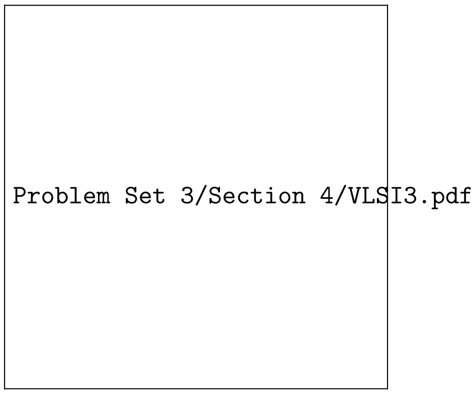
Claim. The signal reaches all of the leaves at the same time.

Solution. We prove this by induction on k. For the case when $k = 1$, we update the smaller edge to have the same weight as the larger edge. Thus, from the root node, the signal reaches all of the leaves at the same time and the claim is proved for $k = 1$. Now assume that the claim holds for k (i.e. where we have 2^k leaves).

Consider a complete binary tree with $n = 2^{k+1}$ leaves. We note that such a tree consists of two subtrees with 2^k leaves. By our inductive hypothesis, these subtrees satisfy the claim. Refer to the diagram below, where the length of the left subtree is M and the length of the right subtree is Q.

The algorithm then updates the smaller edge weight as follows:

- If $n + M > l + Q$, then update l to $n + M - Q$. The signal, therefore, utilises length $n + M$ to travel to all of the leaves on the left subtree. The signal utilises length $(n + M - Q) + Q = n + M$ to travel to all of the leaves on the right subtree. Therefore, in this case, the signal reaches all of the leaves at the same time.



- If $n + M < l + Q$, then update n to $l + Q - M$. The signal, therefore, utilises length $(l + Q - M) + M$ to travel to all of the leaves on the left subtree. The signal utilises length $l + Q$ to travel to all of the leaves on the right subtree. Therefore, in this case, the signal also reaches all of the leaves at the same time.
- If $n + M = l + Q$, we do nothing and the signal trivially reaches all of the leaves at the same time.

In all of these cases, moving up a level also ensures that the signal reaches all of the leaves at the same time which completes the proof in the $k + 1$ case. Therefore, by induction, the algorithm produces a feasible solution which completes the proof. □

We've shown that the algorithm produces *a* solution. We now prove that the solution is optimal (i.e. the length is minimal).

Claim. The total sum of lengths of all edges is minimised.

Solution. We prove this using a *greedy stays ahead* approach. We will inductively prove that our algorithm always stays ahead of the optimal solution. To make the arguments clean and concise, we will give some commentary regarding how you should reason about your arguments.

1. **Label your algorithm's *partial* solutions**. Let x be the length of the tree from the leaves to the root of the subtree at height i generated by our algorithm A. Let y be the length of the tree at height i generated by any optimal solution O.

   Note. Here, we know *exactly* how our solution is constructed but we don't know how the arbitrary algorithm produces their output.

2. **Define your measure**. In this case, the measure is pretty clear – let $f(a_i)$ be the sum of the lengths of the subtree from the node at height i using our greedy algorithm A. Similarly, let $f(o_i)$ be the sum of the lengths of the subtree from the node at height i using an optimal algorithm O.

   Note. In other cases, the measure may not be so obvious. Your function (or measure) should be well defined.

3. **Prove that greedy stays ahead**. Using the measure we defined, we can now inductively prove that the greedy algorithm we proposed is always no worse than any other algorithm. We proceed by induction. The base case holds almost immediately (see the proof for feasibility).

   Now suppose that, for all $j \leq i$, our greedy solution is optimal – that is, for every $j \leq i$, we have that $f(a_j) = f(o_j)$. Now consider the choices made by any algorithm at height $i + 1$. We show that $f(a_{i+1}) \leq f(o_{j+1})$. At height $i + 1$, algorithm O can choose to increase the sum of the length in three ways relative to algorithm A. Let x and y be the increase of the sum of the length given by algorithm O and A respectively.

   - If $x < y$, then algorithm O chooses to increase any of the branches by a value smaller than the maximum of the two branches of concern. However, this yields an infeasible solution since either one of the branches need to reduce in length or the subsequent signal can't reach the leaves at the same time. Hence, this case yields an invalid solution.

- If $x = y$, then algorithm A and algorithm O do not differ which shows that A is just as optimal as any optimal solution. In other words, $f(a_{i+1}) = f(o_{i+1})$.
- If $x > y$, then we increase the length by something greater than the maximum of two branches. Thus, $f(a_{i+1}) < f(o_{i+1})$ which suggests that O is no longer optimal since A is produces a more optimal solution.

By inducting on the height, we can see that our algorithm produces the correct optimal choice on every level which shows that A is optimal. □

We finally show that this algorithm runs in $O(n)$ time. In the first level, we solve each pair of 'sibling' leaves (i.e. those with the same parent) in constant time, and therefore the first level takes $\Theta(n)$. Each subsequent level has half as many leaves, so we have the recurrence

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n).$$

This satisfies case 3 of the Master Theorem, so the solution is $T(n) = \Theta(n)$. □

[K] Exercise 23. Assume that you are given a complete weighted graph G with n vertices $v_1, \ldots, v_n$ and with the weights of all edges distinct and positive. Assume that you are also given the minimum spanning tree T for G. You are now given a new vertex v_{n+1} and the weights $w(n+1, j)$ of all new edges $e(n+1, j)$ between the new vertex v_{n+1} and all old vertices $v_j \in G$, $1 \leq j \leq n$. Design an algorithm which produces a minimum spanning tree T' for the new graph containing the additional vertex v_{n+1} and which runs in time $O(n \log n)$.

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

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. □

[K] Exercise 24. Assume that you are given a complete undirected graph $G = (V, E)$ with edge weights $\{w(e) : e \in E\}$ and a proper subset of vertices $U \subset V$. Design an algorithm that produces the lightest spanning tree of G in which all nodes of U are leaves (there might be other leaves in this tree as well).

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

The validity of the algorithm is rather clear as picking a single direct edge to connect to U will make all nodes in U a leaf. The optimality of this approach follows the optimality of Kruskal's algorithm, once we are done running Kruskal, it is guaranteed that we will always receive an MST in $(V - U)$ with the minimal weight K. Thereby, selecting any edge that violates the ascending order of weights will disqualify the optimality of a set of selections as the total weight of any extra selected edges e will simply be $K + w(e)$.

The overall complexity of Kruskal in the initial step runs in $O((|V| - |U|) \log(|V| - |U|)$, then sorting the relevant edges connecting to U and selecting will take $O(|U| \log(|U|)$. Therefore the overall complexity of the algorithm $O(|V| \log(|V|))$. □

[K] Exercise 25. You are given a connected graph with weighted edges with all weights distinct. Prove that such a graph has a unique minimum spanning tree.

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

[K] Exercise 26. Define the Fibonacci numbers as

$$F_1 = 1,\ F_2 = 1, \text{ and } F_n = F_{n-1} + F_{n-2} \text{ for all } n \geq 3.$$

(a) Design an algorithm to express a given positive integer n as a sum of non-consecutive Fibonacci numbers.

(b) Justify the correctness of your algorithm.

> You will need to argue that your algorithm finds Fibonacci numbers that sum to n and each of the Fibonacci numbers are non-consecutive.

(c) Show that the solution obtained in part **(a)** is unique.

Solution.

(a) Find the largest Fibonacci number less than or equal to n, and subtract it from n. Repeat until there is no remainder.

(b) Let F_k be the largest Fibonacci number less than or equal to n. Then $n < F_{k+1} = F_k + F_{k-1}$, and therefore $n - F_k < F_{k-1}$. It follows that the largest Fibonacci number less than or equal to the remainder is at most F_{k-2}, which is not consecutive with F_k in the Fibonacci sequence.

(c) Suppose for a contradiction that for some positive integer n such that $F_k \leq n < F_{k+1}$, n can be written as a sum of non-consecutive Fibonacci numbers without using F_k. The sum is then at most $F_{k-1} + F_{k-3} + F_{k-5} + \dots$.

Let $P(m)$ be the predicate that $F_m > F_{m-1} + F_{m-3} + F_{m-5} + \dots$.

- The base case $P(1)$ is true, as the sum on the right side is empty.
- The base case $P(2)$ is clearly true.
- Assume that $P(m)$ is true for some $m \geq 1$. We now prove that $P(m+2)$ is also true. We see that

$$F_{m+1} + (F_{m-1} + F_{m-3} + F_{m-5} + \dots) < F_{m+1} + F_m = F_{m+2},$$

as required.

Therefore, any sum of non-consecutive Fibonacci numbers excluding F_k is less than F_k, and so it cannot add to n. It follows that F_k must be chosen, proving the uniqueness of our solution.

□

[H] Exercise 27. 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, using the "greedy stays ahead" method. To assist with your argument, you may want to use the following scaffold to ensure that your argument is on the right track.

- The inductive proof is based on the number of jumps. You should clearly outline why the base case (one stepping stone) follows from your greedy algorithm.
- Clearly define the output of your greedy algorithm and the output of an arbitrarily chosen solution. Suppose that your greedy algorithm is at least *as far* as the chosen solution at jump k.
- Show that, in jump $k+1$, your greedy algorithm is also at least *as far* as the chosen solution.
- Conclude that your greedy algorithm is as good as any arbitrary solution, which justifies the correctness of your algorithm.

Solution.

(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.

□

[H] Exercise 28. You are given an $n \times n$ grid of positive integers. Your task is to find a path from the left top square to the bottom right square, moving one square down or to the right at each step, minimising the sum of numbers traversed along the path.

(a) Consider the following greedy algorithm:

At each step, if there are two directions in which we can step, pick the direction that yields a smaller number in the next square.

Show that this algorithm does *not* always produce the correct solution, using a counterexample.

(b) Design an algorithm that solves the problem correctly. What is the time complexity of the algorithm?

> Convert the grid into a graph. Are there algorithms that we can use to solve the graph instance?

Solution.

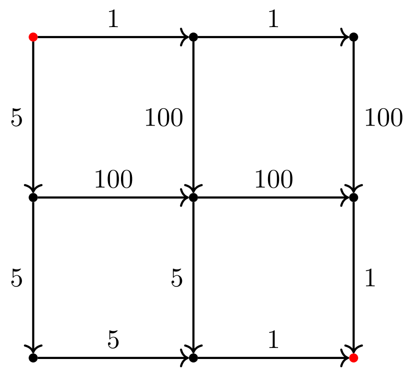
(a) Consider the following 3×3 grid.

1	1	1
5	100	100
5	5	1

Then the greedy solution will choose to go right, right, down, down with a total score of $1 + 1 + 100 + 1 = 103$, while a more optimal solution is to go down, down, right, right for a score of $5 + 5 + 5 + 1 = 16$.

(b) We construct a graph out of the grid. Each square forms a vertex. Therefore, there are n^2 vertices in our graph. Now, u is connected to v if we can travel from u to v by either walking down or to the right. The weight of the edge is the cost of travelling from u to v.

For example, the above example can be constructed as the following graph.



Then the problem reduces to computing the shortest path between the top left vertex and the bottom right vertex. Since each of the weights are positive, we can use Dijkstra's algorithm to compute the path and the weight of the path. There are n^2 many vertices. There are $[n + (n - 1)] - 1 = 2n - 2$ many vertices that have only one edge. The remaining $n^2 - 2n + 2$ vertices have two edges. Therefore, there are $(2n - 2) + 2(n^2 - 2n + 2) = O(n^2)$ edges in such a graph. Therefore, since Dijkstra's time complexity is $O(|E| \log|V|)$, our time complexity is $O(n^2 \log n)$.

□

[H] Exercise 29. Let $G = (V, E)$ be a directed and unweighted graph. We may further assume that G is an acyclic graph.

(a) Design an $O(|V|)$ algorithm that finds the minimum number of vertices such that all vertices in G are reachable from the set of vertices chosen.

(b) Now suppose that we relax the constraint that G is acyclic. Design an $O(|V| + |E|)$ algorithm that finds the minimum number of vertices such that all vertices in G are reachable from the set of vertices chosen.

> How should we reduce the graph G into an acyclic graph?

[H] Exercise 30. Let $G = (V, E)$ be a directed graph. For each vertex $v \in V$, there is some amount of money $c(v)$ sitting on vertex v. Given a start vertex $s \in V$ and an end vertex $t \in V$, your goal is to determine the maximum amount of money you can obtain by following any path from s to t.

(a) Assuming that G is *acyclic*, design an algorithm that determines the maximum amount of money you can obtain by following a path from s to t.

(b) We now relax the constraint that G is acyclic. Design an $O(|V| + |E|)$ algorithm that determines the maximum amount of money you can obtain by following any path from s to t.

[E] Exercise 31. You are given a rooted tree with n vertices. Each node i has an integer weight w_i. The tree is given to you as an array of vertex weights W and an array of parents P (i.e. the ith vertex has weight $W[i]$ and parent $P[i]$).

We want to colour all of the vertices of the tree as follows:

1. The root vertex must be coloured first.
2. For all other vertices, its parent vertex must be coloured before it is coloured.

However, colouring a vertex has a cost, measured by when it is coloured and its associated weight. In other words, the kth vertex – say vertex a – that is coloured has cost $k \times w_a$. For example, the cost of colouring the root node r is $1 \times w_r$ since the root of the tree always has to be coloured first.

(a) A basic greedy approach to this problem is to focus on the current vertex and colour all its children in descending order of weight. Show that this doesn't always yield a correct solution by providing a counterexample.

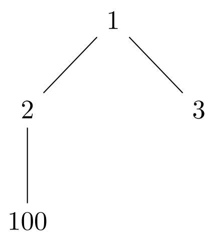
(b) Design an $O(n^3)$ algorithm that finds the minimum total cost of colouring the whole tree. Justify the correctness of your algorithm.

(c) (**Extended students only**) Design an algorithm that solves the previous problem in $O(n \log n)$.

> Use a disjoint set data structure to reduce the time complexity.

Solution.

(a) Consider the following counter example.



The greedy solution would proceed $1 \to 3 \to 2 \to 100$ for a cost of 413. However, colouring the nodes in the order $1 \to 2 \to 100 \to 3$ has a more optimal cost of 317, which shows that the greedy solution does not always produce the minimum total cost.

(b) Suppose we have an arbitrarily weighted tree and an ordering in which the nodes are coloured. Let the node with the greatest weight be node x and the ith coloured node, and the node coloured before it be node y. If, after the first $i-2$ nodes were coloured, we could already colour x, then switching the order in which we paint x and y would change our total cost by

$$((i-1)\cdot W[x]+i\cdot W[y])-((i-1)\cdot W[y]+i\cdot W[x])=W[y]-W[x]\leq 0,$$

since x has the greatest weight. As such, we conclude that we would always want to colour the maximum weight node as soon as possible after its parent has been coloured.

Extending this, suppose we have a sequence of nodes with weights $a_1,\dots,a_p$ that we decide should be coloured one after another with no other nodes in between, and a sequence with weights $b_1,\dots,a_q$ that have the same condition. Then, there are two orderings of these nodes:

- $a_1,\dots,a_p,b_1,\dots,b_q$ with total cost:
$$C_1=\sum_{i=1}^{p} ia_i+\sum_{i=1}^{q}(i+p)b_i;$$
- $b_1,\dots,b_q,a_1,\dots,a_p$ with total cost:
$$C_2=\sum_{i=1}^{q} ib_i+\sum_{i=1}^{p}(i+q)a_i.$$

Then ordering 1 is better if

$$\begin{aligned}C_2-C_1&\geq 0,\\ \sum_{i=1}^{p} qa_i-\sum_{i=1}^{q} pb_i&\geq 0,\\ \frac{1}{p}\sum_{i=1}^{p} a_i&\geq \frac{1}{q}\sum_{i=1}^{q} b_i.\end{aligned}$$

This means that we can replace x and $P[x]$ with a composite node of weight $\frac{1}{2}(W[x]+W[P[x]])$ that represents the sequence $x\to P[x]$. We can then find the new maximum node in this modified tree and combine it with its parent, and repeat this until we have combined everything into a single node and have the final ordering of the tree.

In an implementation of this algorithm, we can merge the child node into its parent node and adjust all children of node x to have $P[x]$ as their parent. We will also need arrays:

- $A[n]$, storing what node will come after each node in our final ordering.
- $L[n]$, storing the last node in the sequence represented by each composite node. This will be initialised with $L[i]=i$, as all nodes start off as 1-node sequences.
- $S[n]$, storing the number of nodes in the ith node's sequence. All elements of this array are initially 1.

We will reuse the input array W to represent the total weight of all nodes in a node's sequence. We will also assume arrays are 1-indexed, with $P[i]=0$ if i has no parent or has been combined into another node – initially only the root node will have $P[i]=0$.

Then, we will repeatedly find a node x with the maximum value of $W[i]/S[i]$ among all nodes where $P[i]\neq 0$ – if none exist then we finished combining nodes and can output the sequence. Once this node has been found, we will apply the following adjustments onto our arrays:

- For all node i where $P[i] = x$, set $P[i]$ to $P[x]$ to move all of x's children to be children of $P[x]$.
- Increase $S[P[x]]$ by $S[x]$ and $W[P[x]]$ by $W[x]$ to update $P[x]$'s effective weight.
- Let y be the $L[P[x]]$, the last node of x's parent's sequence. This will need to be updated to $L[x]$ and $A[y]$ needs to be set to x before that.
- Finally, set $P[x]$ to 0 to indicate that x is no longer the first node in any sequence.

Finally, by following A, we can calculate the overall cost as well as output the sequence in which nodes are to be coloured.

Each iteration of this is dominated by finding the maximum node and updating nodes' parents, both of which take $O(n)$ time. This procedure will need to run $\Theta(n)$ times, as each step reduces the number of node sequences by 1, giving us a total running time of $O(n^2)$.

Arrays L and S, and adjusting array W, can be made redundant by iterating over A to determine the weight and the last element in a sequence. This does not increase the overall complexity of the running time.

(c) We can adjust the previous algorithm to be able to run each iteration in $O(\log n)$ time.

Instead of iterating over all nodes to find one with the maximum weight, we can instead use a priority queue (also known as a max heap) that holds (equivalent cost, node number) pairs. This is initially populated with $(W[i], i)$. Each iteration, we pop items from this until we find a node with the correct weight and $P[i] \neq 0$, and add $(W[P[x]]/S[P[x]], P[x])$ to the priority queue at the end of each iteration.

To reduce the running time of updating parents, we can use a disjoint-set (also called union find) data structure. Instead of the $O(n)$ reparenting step, we simply merge x into $P[x]$ so that all queries to elements of x's and $P[x]$'s sets will give $P[x]$.

Operations on the priority queue and the disjoint-set data structures all take $O(\log n)$ time, meaning that the overall running time is reduced to $O(n \log n)$.

□

[X] Exercise 32. A set of edges $M \subseteq E$ for a set $S \subseteq V$ of terminal vertices is a *multiway cut* if the removal of these edges completely separate all of the terminal vertices.

Multiway Cut

Instance: A connected graph $G = (V, E)$ and a set of terminal vertices $S = \{s_1, \dots, s_k\}$. Consider the optimisation problem.

Task: Return the size of the smallest multiway cut.

It is known that this problem does not have a polynomial-time algorithm. We, however, will present a polynomial-time greedy algorithm that does not do so badly.

[E] Exercise 33.