Midterm Review Problems – Sample Solutions

Semester 1, 2019

by Joshua Lau and Aleks (for possible mistakes blame only Aleks)

A helpful note on ordering-style problems

Here's an approach that works well for many ordering-style problems. These are the ones where you have to choose an order among n elements (each with some properties, perhaps weights / deadlines / durations / costs and so on) that minimises or maximises some function of their order.

First, just consider the case when there are two elements. Call them A and B, say. Then, we know that there are only two possible orders: either A then B, or B then A. For both of these orders, try writing down mathematical expressions for the cost / value you get from each order. For instance, for Question 21 this might be $w_A t_A + w_B(t_A + t_B)$ (and swapping A and B for the reverse order). Then, decide *when* it's better to take the order A then B. In this case, this is when

$$w_A t_A + w_B(t_A + t_B) < w_B t_B + w_A(t_B + t_A).$$

Next, cancel out common terms (there almost certainly will be some):

$$w_B t_A < w_A t_B.$$

Finally, rearrange the inequality so all A terms are on one side, and all B terms are on the other:

$$\frac{t_A}{w_A} > \frac{t_B}{w_B}$$

This gives us a sorting **key**: a numerical value we can derive exclusively as a function of a single element. For many problems, using this key directly is enough to solve the entire problem although this approach **only works** if a small, local change (swapping two adjacent elements) **does not affect** the previous or subsequent elements in the order. As long as this is the case, this approach will typically work.

Finally, when writing up your answer, you should begin by stating what you are sorting by. If required, you can extend this into a full *proof of correctness*. One way to do this is to assume that the elements are in some order that is not sorted. This means there must be a pair of *adjacent* elements that are out of order. By extending the same logic as the two element case, we show that we can swap them, and mathematically describe the impact this has on the final cost / value / sum etc. Using our inequalities, we can argue that swapping makes the solution no worse (or even strictly better), and by repeatedly doing so, we get a sorted order (this is a property of Bubble Sort). This implies that the sorted order is no worse than any other order, so it is optimal.

Solutions

1. Assume you are given two arrays A and B, each containing n distinct integers and equation $x^8 - x^4y^4 = y^6 + x^2y^2 + 10$. Design an algorithm which runs in time $O(n \log n)$ which finds if A contains a value for x and B contains a value for y that satisfy the equation.

   Solution: First, observe that all the terms are even powers, so we can ignore the sign of the numbers in our arrays. Then, we can rewrite the equation as $y^6 + x^4y^4 + x^2y^2 = x^8 - 10$. Now, for a fixed value of x, the left-hand side is non-decreasing with respect to y (for non-negative y) and the right-hand side is constant.

   This gives rise to a familiar solution: we sort B, say, into non-decreasing order. We iterate through A, giving us our possible values for x. For a fixed value of x, we binary search over B to check if a feasible y exists: to determine if we want smaller or larger values for y, we compare $y^6 + x^4y^4 + x^2y^2$ with $x^8 - 10$, and choose the correct half of our current binary search's range to continue with.

2. Assume that you are given an array A containing $2n$ numbers. The only operation that you can perform is make a query if element $A[i]$ is equal to element $A[j]$, $1 \leq i, j \leq 2n$. Your task is to determine if there is a number which appears in A at least n times using an algorithm which runs in linear time. (Warning: a tricky one. The reasoning resembles a little bit the reasoning used in the celebrity problem: try comparing them in pairs and first find one or at most two possible candidates and then count how many times they appear.)

   Solution: This is sometimes called the Majority Voting Problem. Our solution proceeds in two stages. First, we identify one or at most two candidate values for x. Then, in $O(n)$ we can count the number of times each of this candidates occurs in A, and check if it is at least n. Thus, it remains to find such candidates.

   Assume that an element a appears in A at least n times. We will put elements into three piles, X, Y and Z. We split these $2n$ elements into n pairs. Pick any such pair of elements from A and compare them.

   - If the two elements are different toss them both into pile Z. Note that in this way we could have removed at most one copy of a, so after removing these two elements we are left with $2n - 2$ elements in A such that at least $n - 1$, i.e., at least a half of them are equal to a.
   - We continue comparing elements in these pairs and if all pairs consist of distinct elements, we check for the very last pair by direct counting if either of the two elements appears in A at least n times.
   - If we encounter a pair of equal elements, we put one on pile X and the other on pile Y.

- Continuing in this way through all of n pairs, if a appeared at least n times in A, then at least half of elements of $X \cup Y$ will be equal to a. But the two piles X ands Y are identical, so each one of them must contain at least half elements equal to a.
- We now apply the same procedure to pile X only. Note that X has at most half as many elements as A, so the total process involves less than $n + n/2 + n/4 + n/8 + ... = 2n$ many comparisons. At the end we will end up with at most two distinct elements and we simply count for each of them how many times they appear in A.

3. Let M be an $n \times n$ matrix of distinct integers $M(i, j)$, $1 \leq i \leq n$, $0 \leq j \leq n$. Each row and each column of the matrix is sorted in the increasing order, so that for each row i, $1 \leq i \leq n$,

$$M(i, 1) < M(i, 2) < \ldots < M(i, n)$$

and for each column j, $1 \leq j \leq n$,

$$M(1, j) < M(2, j) < \ldots < M(n, j)$$

You need to determine whether M contains a given integer x in $O(n)$ time.

Solution: We start at $(n, 1)$. Now suppose we are at (i, j). If $M(i, j) = x$, we are done. Otherwise, if $M(i, j) > x$, we know that $M(i, j+1) < M(i, j+2) < \ldots M(i, n)$ are all greater than x, so x cannot be in row i we move to $(i-1, j)$. Similarly, if $M(i, j) < x$, we know that $M(1, j), M(2, j), \cdots < M(i, j)$ are all less than x, so x cannot be in column j. Hence, we move to $(i, j + 1)$. This procedure looks at at most $2n - 1$ cells of the matrix, since we always move up or right.

4. Assume you have an array of $2n$ distinct integers. Find the largest and the smallest number using $3n - 2$ comparisons only.

Solution: We split all numbers in $\frac{2n}{2} = n$ pairs and compare the two numbers in each pair. we put all the smaller ones on pile S and all the larger ones in pile L. Clearly the smallest among all numbers cannot be in pile L and the largest among all numbers cannot be in pile S. Thus we now find the smallest number in S using $n-1$ additional comparisons and the largest number in M also with $n - 1$ comparisons, in total using only $3n - 2$ comparisons.

5. Assume that you have an array of 2^n distinct integers. Find the largest and the second largest number using only $2^n + n - 2$ comparisons.

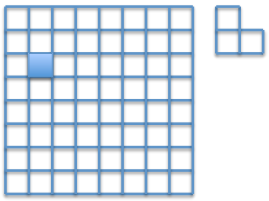
Solution: We can think of forming a knock-out tournament between the 2^n integers. The tournament "draw" takes the shape of a perfect binary tree. Each "match" is a comparison between two integers, with the greater one winning and proceeding to the next round. Since we know a binary tree with X leaves has $X - 1$ internal vertices, our tournament has $2^n - 1$ matches to

determine the winner (largest element of the array). Additionally, since the binary tree is perfect, we know that the winner played precisely n matches, winning all of them.

Now who could possibly have been second-best? The second-best player (and thus second-largest number) *must have lost to the winner!* This is because they would have beaten anyone else. Thus, we only have n candidates for the second largest element of the array, and we can determine the largest of these in $n-1$ comparisons, giving $2^n + n - 2$ comparisons, as required.

You should convince yourself that equal elements make no difference to the design or implementation of our algorithm. As a general rule, it is often convenient to assume elements / values are distinct when first solving a problem, and then returning later to iron out the cases once a solution is found.

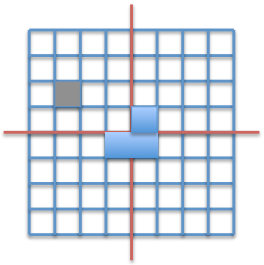
6. You are given a $2^n \times 2^n$ board with one of its cells missing (i.e., the board has a hole); the position of the missing cell can be arbitrary. You are also given a supply of "dominoes" each containing 3 such squares; see the figure below. Your task is to design an algorithm for covering the entire board with such



"dominoes", except for the hole which should remain uncovered.

Solution: We can do this recursively / inductively. This problem is easy for $n = 1$: we are given a 2×2 board with 1 square removed, so we can just fit our "domino" into the remaining 3 squares.

Now suppose $k \geq 1$ is an integer, and we have a solution for *all* such $2^k \times 2^k$ boards that are missing exactly one square. If we are given a $2^{k+1} \times 2^{k+1}$ board, we can split it into its four quadrants, giving four $2^k \times 2^k$ boards:



Our missing square necessarily occurs in one of these quadrants, so we tile that recursively using our algorithm. For the remaining three quadrants, we

place a "domino" on the three squares that are touching the centre (the place where we split our boards), and treat those squares as removed. We can then apply our algorithm recursively into these three quadrants as well, as they too are now missing a square.

7. Multiply the following pairs of polynomials using at most the prescribed number of multiplications where both numbers multiplied are large (large numbers are those which depend on the coefficients and thus can be arbitrarily large).

   (a) $P(x) = a_0 + a_2x^2 + a_4x^4 + a_6x^6$ and $Q(x) = b_0 + b_2x^2 + b_4x^4 + b_6x^6 + b_8x^8$ using at most 8 multiplications of large numbers;

   (b) $P(x) = a_0 + a_{100}x^{100}$ and $Q(x) = b_0 + b_{100}x^{100}$ with at most 3 multiplications of large numbers.

   Solution:

   (a) First, observe that it is enough to be able to multiply any degree 3 polynomial by a degree 4 polynomial using 8 such multiplications: both P and Q are really polynomials with those respective degrees in x^2.

   Since the result is a polynomial of degree 7, we can uniquely determine it by determining its values at 8 points. For this we can choose $\{-4, -3, \ldots, 3\}$ or alternatively, the 8th roots of unity. We evaluate P at these 8 points and Q at these 8 points: these are only multiplications of a large number my a (constant size) scalar, so these operations are cheap.

   We then multiply the results pointwise: these require precisely 8 large number multiplications. We can then determine the coefficients from these values by setting up a system of linear equations. Solving this is done by inverting a constant matrix (as described in the course), so this inversion can even be done by hand, offline and requires no computation. We then multiply the matrix by the vector formed by the pointwise multiplications, which again only multiplies these results by scalars, to give the final polynomial.

   (b) We have that $PQ(x) = a_0b_0 + (a_0b_{100} + a_{100}b_0)x^{101} + (a_{100}b_{100})x^{200}$. Remember that addition is cheap, but multiplication is expensive. By observing that $(a_0 + a_{100})(b_0 + b_{100}) = a_0b_0 + a_0b_{100} + a_{100}b_0 + a_{100}b_{100}$, we can perform this multiplication, as well as a_0b_0 and $a_{100}b_{100}$. The latter two give the coefficients of x^0 and x^{200}, and subtracting these from the first gives the coefficient of x^{101}. Thus, we only use 3 multiplications.

8. Describe all k which satisfy $i\,\omega_{64}^{13}\omega_{32}^{11} = \omega_{64}^{k}$ (i is the imaginary unit).

Solution: We have

$$\begin{aligned}\omega_{64}^{k} &= i\omega_{64}^{13}\omega_{32}^{11} \\ &= \omega_{64}^{16}\omega_{64}^{13}\omega_{64}^{22} \\ &= \omega_{64}^{16+13+22} \\ &= \omega_{64}^{51}\end{aligned}$$

Since the roots of unity are periodic, this means $k = 51 + 64n$ for all integers $n \in \mathbb{Z}$.

9. Consider the polynomial

$$P(x) = (x - \omega_{64}^{0})(x - \omega_{64}^{1})(x - \omega_{64}^{2}) \dots (x - \omega_{64}^{63})$$

   (a) Compute $P(0)$;

   (b) What is the degree of $P(x)$? What is its coefficient of the highest degree of x present in $P(x)$?

   (c) What are the values of $P(x)$ at the roots of unity of order 64?

   (d) Can you represent $P(x)$ in the coefficient form without any computation?

10. Describe how you would compute all elements of the sequence $F(0), F(1), F(2), \dots, F(2n)$ where

$$F(m) = \sum_{\substack{i+j=m \\ 0 \le i,j \le n}} \log(j+2)^{i+1}$$

in time $O(n \log n)$.

Solution:

(a)

$$\begin{aligned}P(0) &= (0 - \omega_{64}^{0})(0 - \omega_{64}^{1}) \dots (0 - \omega_{64}^{63}) \\ &= (-1)^{64}\omega_{64}^{0+1+\dots+63} \\ &= \omega_{64}^{\frac{64\times 63}{2}} \\ &= \omega_{64}^{32*63} \\ &= \left(\omega_{64}^{32}\right)^{63} \\ &= (-1)^{63} \\ &= -1\end{aligned}$$

   (b) The degree of $P(x)$ is 64. The coefficient of the highest degree of x is 1 (there is only one term in the product of degree 64, which is all the x's multiplied together).

   (c) The values at all roots of unity are 0.

(d) Since $P(x)$ is precisely the polynomial whose roots are at the 64th roots of unity and the leading coefficient is one, $P(x) = x^{64} - 1$, by definition. Note that this is clearly an easier way to solve part (a)!

11. Describe how you would compute all elements of the sequence $F(0), F(1), F(2), \ldots, F(2n)$ where

$$F(m) = \sum_{\substack{i+j=m \\ 0 \le i,j \le n}} \log(j+2)^{i+1}$$

in time $O(n \log n)$.

Solution: We have that

$$\begin{aligned} F(m) &= \sum_{\substack{i+j=m \\ 0 \le i,j \le n}} \log\left((j+2)^{i+1}\right) \\ &= \sum_{\substack{i+j=m \\ 0 \le i,j \le n}} (i+1)\log(j+2) \end{aligned}$$

Now this is just a convolution of two sequences. We can set up the following polynomials

$$\begin{aligned} P(x) &= \sum_{i=0}^{n} (i+1)x^i \\ Q(x) &= \sum_{j=0}^{n} (\log(j+2))x^j \end{aligned}$$

and calculate their product, $(PQ)(x)$ in $O(n \log n)$ time using the Fast Fourier Transform (FFT). Then $F(m)$ is just the coefficient of x^m in $PQ(x)$.

12. In Elbonia coin denominations are 81c, 27c, 9c, 3c and 1c. Design an algorithm that, given the amount that is a multiple of 1c, pays it with a minimal number of coins. Argue that your algorithm is optimal.

Solution: We use the greedy strategy, always giving the largest number of the highest possible denomination. Thus we first give $\lfloor x/81 \rfloor$ many 81c coins and continue with the $27c$ coins etc. To prove that this results in the fewest possible number of coins assume the opposite, that there is a combination with fewer coins for the same amount and among such combinations pick the one using the fewest number of coins. Such combination of coins clearly is not the one produced by our strategy. Without loss of generality, we can assume that such a combination violates the greedy strategy already with the number of $81c$ coins used. Thus, after subtracting the amount given with $81c$ coins, we are left with an amount of at least $81c$. But such a combination can contain at most two 27c coins, otherwise it would not be optimal: if three or more 27c coins were present we could replace three of them with a single 81c coin,

thus violating our assumption that such a combination employs the minimal possible number of coins. However, after subtracting at most $2 \times 27c$ we end up with an amount of at least $27c$ to be given using only 9,3 and 1c coins. Again, at most 2 coins of 9c could be used and we end up with at least 9c to be given using only 3 and 1 cent coins. At most two 3 cent coins can be used and we end up with 3c to be given using only 1c coins, which clearly violates the assumed optimality of such a combination.

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

    Solution: Consider the set of coins $\{1c, 3c, 4c\}$. If we wish to make $6c$ of change, the greedy algorithm would first choose 4c, then be forced to take two 1c coins for three coins in total. The optimal solution is to choose two 3c coins.

14. Assume you are given n tasks each of which takes the same, unit amount of time to complete. Each task i has an integer deadline d_i and penalty p_i associated with it which you pay if you do not complete the task in time. Design an algorithm that schedules the tasks so that the total penalty you have to pay is minimized.

    Solution: Sort the tasks in non-increasing order of penalty p_i. Since all the tasks take the same amount of time, we should prioritise those that have the largest penalty first. For each task in sorted order, we will assign to start it at the latest time that still allows us to complete it before the deadline: for instance, if our deadline is $d_i = 7$, we will try to start at time 6. However, suppose there are other tasks scheduled to start at 4, 5, 6, then we assign to start it at time 3, i.e., the latest available instant preceding 6. If no such time exists, we can't complete this on time and have to pay the penalty for this task and we can schedule them after all successfully scheduled task have been completed.

    To see this is optimal, suppose that there is a better schedule S. We go through all jobs in order of decreasing penalty and find the highest penalty job which in S violates the greedy choice. Thus, such a job was not scheduled at the latest free slot preceding its deadline. This job is either not successfully scheduled at all or it was scheduled before its position in the greedy schedule, because all slots after its position in the greedy schedule and before the deadline are occupied by higher penalty jobs (we assumed S agrees with the greedy schedule for all higher penalty jobs). If it was not scheduled at all, evict the lower penalty job placed at the slot which would be chosen by the greedy schedule and place it there to obtain a better schedule, which is a contradiction with the optimality of S. If it was scheduled before its slot in the greedy schedule just swap it with the job at its position in the greedy schedule to obtain a correct equally good schedule complying with the greedy schedule for that job too (note that the job we are swapping with is moved to an earlier time and thus will be completed before its deadline). Thus, proceeding in this way we can

morph S into the schedule obtained by the greedy choices, which contradict the assumption that S is strictly better than the schedule obtained by the greedy choices.

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

    Solution: Let us call the stalls which need to be covered *the necessary stalls*. Cover all stalls between (and including) the first and the last necessary stall with a single board. Look for the longest contiguous sequence of stalls which are not necessary and excise the corresponding slice of the board. Continue in this manner until you obtain 11 pieces. Optimality is easy to prove as before by assuming there is a better way of covering the necessary stalls, take the best possible such and look for the longest stretch of stalls which are not necessary but remain covered in this "better solution". Remove this slice but join two pieces with a smaller gap to obtain an even better solution and achieve a contradiction.

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

    Solution: Observe that our solution is uniquely determined by the order which we choose to mill the items, and that no matter which order we choose, the total milling time is constant. Specifically, the production is finished when the last item has finished polishing. We sort our items in non-increasing order of p_i and mill them in this order. We now prove that this is optimal.

    Suppose, without loss of generality, that our items are in order $1, 2, \ldots, n$. Then define the finishing time of item k as

    $$f(k) = p_i + \sum_{j=1}^{i} m_i$$

    so our overall finishing time is the maximum among all $f(k)$. Further suppose there are two adjacent items i and $i+1$ in our order such that $p_i < p_{i+1}$: if there aren't, then the items are already in sorted order. Then if we process our items in the modified order $1, 2, \ldots, i-1, i+1, i, i+2, i+3, \ldots, n$ and let $g(k)$ be the finishing time for item k under this order. Then, $g(k) = f(k)$ whenever $k \notin \{i, i+1\}$. Now let $S = m_1 + m_2 + \cdots + m_{i-1}$. Then we have

that

$$\begin{aligned} f(i+1) &= S + m_i + m_{i+1} + p_{i+1} \\ g(i) &= S + m_{i+1} + m_i + p_i \\ g(i+1) &= S + m_{i+1} + p_{i+1} \end{aligned}$$

So since $p_{i+1} > p_i$, we have $f(i+1) > g(i)$ and since all milling times are positive, we also have that $f(i+1) > g(i+1)$. Hence, swapping i and $i+1$ gives a modified order that is never worse than our original order, which never increases $\max_i\{f(i)\}$. We can repeat this argument until we get sorted order: to show that this indeed occurs, observe that this procedure is the same as bubble sort, for which we know from earlier courses that always terminates.

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

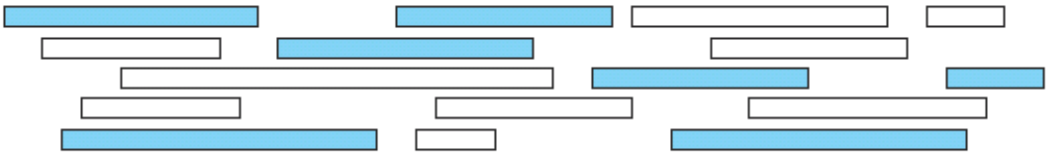
    Solution: Recall that this problem is an easy greedy on a line (pick the one that ends first, repeat). For each arc, we separately assume it is in the solution, then repeat clockwise along the arc just as in the line. Then, take the maximum among all these solutions. Note that this solution runs in $O(n^2)$ time.

    To prove that this solution is correct, we note that we clearly must choose some arc to be a part of our set. Once this arc is chosen, we cannot take any arc intersecting with this one, so we effectively "delete" the arc from the circle. This leaves a line, and we can appeal to the proof of correctness for our "line-algorithm" in this case.

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

    Solution: Recall that this problem is also an easy greedy on a line (pick the first ending point, place a needle there, remove all stabbed intervals, repeat). Again, we can reduce this problem to a line as follows: pick a starting interval and stab it at its end. Remove all stabbed intervals and pick among the rest an interval with the "earliest" end and stab it at the very end. Continue in this way until all intervals have been stabbed or eliminated and record the number of pins used. Repeat this procedure for all intervals as the starting interval. Among all thus obtained solutions pick the one which uses the fewest number of needles. To prove that this solution is correct, take an optimal solution and move all needles to the right till they stab one of the stabbed intervals at its right end. Pick any such interval stabbed at its right end; then such solution would have been obtained by our method while stating with that interval.

19. 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. Design and estimate the time complexity of an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.



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

Solution: Sort the intervals in non-decreasing order of left point and let S be the set containing all X_L and all X_R values. Initially, we need to cover the minimum left point $x_{\min}$. Observe that we must take one of the intervals that have $x_L \leq x_{\min}$: we may as well take the one that has the largest x_R. Repeat the process, updating $x_{\min}$ with the smallest value larger than this largest x_R in S. Proof of optimality just as before.

20. Suppose you have n video streams that need to be sent, one after another, over a communication link. Stream i consists of a total of b_i bits that need to be sent, at a constant rate, over a period of t_i seconds. You cannot send two streams at the same time, so you need to determine a schedule for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time 0 (and therefore ends at time $\sum_{i=1}^{n} t_i$, whichever order you choose). We assume that all the values b_i and t_i are positive integers. Now, because youre just one user, the link does not want you taking up too much bandwidth, so it imposes the following constraint, using a fixed parameter r:

    For each natural number $t > 0$, the total number of bits you send over the time interval from 0 to t cannot exceed $r\,t$.

    Note that this constraint is only imposed for time intervals that start at 0, not for time intervals that start at any other value. We say that a schedule is valid if it satisfies the constraint.

    (a) Design an $O(n \log n)$ algorithm which outputs a valid schedule if there

is one and outputs the message "no valid schedule" otherwise. *Hint: Consider the bit rate b_i/t_i of each stream.*

(b) Design an $O(n)$ such algorithm. *Hint: for each bitstream consider the quantity $s_i = rt_i - b_i$. Which streams should go first depending on the sign of s_i? In other words which streams buy "extra time" for sending other streams?*

Solution:

(a) In $O(n \log n)$ time we sort (e.g. using Merge Sort) the streams so that their bit rates b_i/t_i are in non-decreasing order. Thus, the streams with the smaller bit rates occur first. Given this order, we can check in $O(n)$ (just by iterating and summing) if the order is valid. If it is, then we output it, otherwise we output "no valid schedule", since if some schedule exists, this sorted order must be valid.

(b) Our requirement can be summarised as follows: for all $1 \leq k \leq n$ we have that

$$\begin{aligned}\sum_{i=1}^{k} b_i &\leq r \sum_{i=1}^{k} t_i \\ &= \sum_{i=1}^{k} rt_i.\end{aligned}$$

By rearranging, we obtain

$$\sum_{i=1}^{k} (rt_i - b_i) \geq 0.$$

Hence, by mapping each item i to the value $rt_i - b_i$ we can reduce our problem to the following: given an array of size n of values, arrange them in some order so that the sum of every prefix is non-negative (or determine if this is not possible).

Clearly, if the sum of all these numbers is negative, then this is impossible, since no matter how we arrange them, our constraint will be violated at the end: we can check if this is the case in $O(n)$. Now suppose this is not the case, this necessitates that the sum of the positive values is greater than the sum of the negative values. Hence, it suffices to place all the positive numbers before all the negative numbers. We can do this is $O(n)$ in various ways, for instance, performing a "bucket sort" with two buckets: one for positive numbers and one for negative numbers.

21. 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 do the jobs on their single machine in an order that keeps their customers 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 j is the first to be done we would have $C_i = t_j$, 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 is job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times, $\sum_{i=0}^{n} w_i C_i$. Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for job i. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=0}^{n} w_i C_i$.

Solution: (Note: this problem is essentially the same as the tape storage problem done in class, when each file to be stored has a length t_i and a probability p_i that it will be needed.) We choose to perform jobs in non-increasing order of $\frac{w_i}{t_i}$ At first glance this makes some intuitive sense: if all the jobs have equal weight then this is exactly sorting in non-decreasing order by duration, so we do the faster jobs first (as we have seen in other greedy problems). Similarly, if all jobs take the same amount of time, then we prioritise the jobs that have heavier weight.

To prove this is the case, suppose, without loss of generality, that our jobs are numbered $0, 1, \ldots, n$ and that there is an adjacent pair "out of order" meaning there is some i such that $\frac{w_i}{t_i} < \frac{w_{i+1}}{t_{i+1}}$ or equivalently, $w_i t_{i+1} < w_{i+1} t_i$. Now suppose we consider the modified order $0, 1, \ldots, i-1, i+1, i, i+2, i+3, \ldots, n$. Then note that the completion time of all jobs besides i and $i+1$ is not affected, so we need only to consider the effect on i and $i+1$. Let $S = \sum_{i=0}^{n} t_i$ be the time it takes to complete all prior jobs. In the original order the weighted contribution of these jobs was

$$\begin{aligned} A &= w_i(S + t_i) + w_{i+1}(S + t_i + t_{i+1}) \\ &= S(w_i + w_{i+1}) + (w_i t_i) + (w_{i+1} t_{i+1}) + w_{i+1} t_i \end{aligned}$$

whereas in the modified order it is

$$\begin{aligned} B &= w_{i+1}(S + t_{i+1}) + w_i(S + t_{i+1} + t_i) \\ &= S(w_i + w_{i+1}) + (w_i t_i) + (w_{i+1} t_{i+1}) + w_i t_{i+1} \end{aligned}$$

so

$$\begin{aligned} B - A &= w_i t_{i+1} - w_{i+1} t_i \\ &< 0 \end{aligned}$$

since i and $i+1$ are out of order. Hence, we can always swap a pair of elements out of order (according to our defined sort order) and obtain a strictly smaller weighted sum of completion times. As we can always do this whenever the elements are not in a sorted order, and, starting at any sequence we can do this only finitely many times before arriving at a sorted order (by bubble sort), it shows that our solution is optimal.

22. You are given n points x_i $(1 \leq i \leq n)$ on the real line and n intervals $I_j = [l_j, r_j]$, $(1 \leq j \leq n)$. Design an algorithm which runs in time $O(n^2)$ and determines if each point x_i can be assigned to a distinct interval I_j so that $x_i \in I_j$.

    Solution: Sort the points in non-decreasing order. For each one in order, we want to assign it to some interval $[l_j, r_j]$ with $l_j \leq x_i \leq r_j$. Now among all those with $l_j \leq x_i$, we want r_j to be as small as possible. If the smallest such r_j is strictly less than x_i, our order dictates that this interval can't be matched to any other point either, so we can return false. Otherwise, we should assign this interval to this point and continue.

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

    Solution: Show that Kruskal's algorithm produces a spanning tree which is minimal in this sense as well. The proof is identical to the optimality proof for the original Minimum Spanning Tree algorithm.

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

    Solution: Assuming all tolls and hotel costs are non-negative, we can use Dijkstra's algorithm for this problem: there are many ways to see this, we provide just one. We can split any city u into two nodes, u_{in} and u_{out}. All roads $u \to v$ will be $u_{\text{out}} \to v_{\text{in}}$ in our new graph, and vice-versa. The weight of each such edge is equal to the toll cost of that road. Additionally, for all u we have an edge $u_{\text{in}} \to u_{\text{out}}$ with weight equal to the cost of the cheapest hotel in u. Then, the shortest route from C to R is given by the shortest route from C_{out} to R_{in} in our modified graph.

25. You need to write a very long paper "The meaning of life". You compiled a sequence of books in the order you will need them, some of them multiple times. Such a sequence might look something like this:

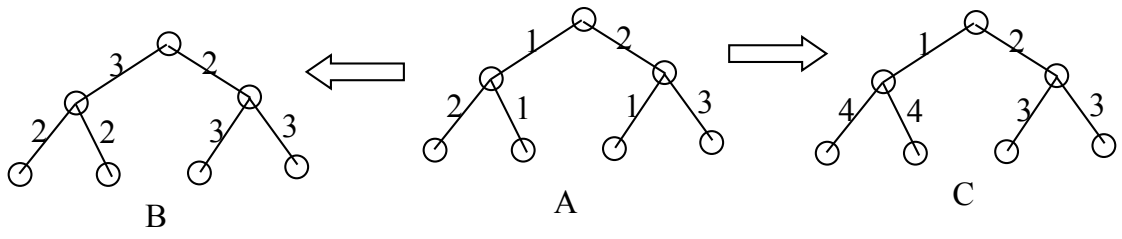
    $$B_1, B_2, B_1, B_3, B_4, B_5, B_2, B_6, B_4, B_1, B_7, \ldots$$

    Unfortunately, the library lets you keep at most 10 books at home at any moment, so every now and then you have to make a trip to the library to exchange books. On each trip you can exchange any number of books (of course, between 1 and all of 10 books you can keep at home). Design an algorithm which decides which books to exchange on each library trip so that

the total number of trips which you will have to make to the library is as small as possible.

Solution: Note that we only care about the number of trips we make to the library: we do not care how many times we have to carry books / how many books we carry in each trip. Thus, on our first visit, we consider our sequence of books, and find the first 10 distinct books the appear in our sequence. We take them home, and use them which will remove some prefix of our sequence: our progress is only halted once we encounter a new distinct book. This requires a trip to the library, so we may as well bring all 10 books back to the library, and then return home with the first 10 distinct books of our new sequence (the sequence with the completed prefix removed). Note that this next batch of books may contain some books that we brought back to library: this is fine, since we only care about the number of trips we took to the library! Of course, we can make the solution more natural if we bring for exchange only those books which are not among the next 10 needed books.

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



Solution: We solve this problem recursively. If $k = 0$, we are done. Otherwise, suppose our root has two children L and R that are x_L and x_R from our root, respectively.

We recurse on either side to solve two instances of the problem with 2^{k-1} leaves: after this is done, all leaves in the left subtree have the same distance d_L to L, and all the leaves in the right subtree have the same distance d_R to

R. Then, without loss of generality suppose $d_L + x_L \geq d_R + x_R$. In this case, we simply add $d_L + x_L - (d_R + x_R)$ to d_L. The reverse case is handled in the same way.

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

    Solution: Suppose there is someone who does not have five people they know. Then this will be the case no matter what subset of people we invite, so we should remove them from consideration entirely. Similarly, if there is some person who does not have five people they don't know, we should also remove them from consideration entirely. At each iteration, we must remove at least one person, so this algorithm terminates. When this algorithm terminates, by definition the subset of invitees is valid. Since we only remove people we have deduced could not possibly be invited, we always produce the largest possible number of invitees.

28. Assume that you got a fabulous job and you wish to repay your student loan as quickly as possible. Unfortunately, the bank "National Road Robbery" which gave you the loan has the condition that you must start 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 a half of the amount you paid the previous month. On top of these conditions, your schedule must be such that the last payment is \$1. Design an algorithm which, given the size of your loan, produces a payment schedule which minimizes the number of months it will take you to repay your loan while satisfying all of the banks requirements.

    Solution: Suppose the loan is N dollars. We can view the powers of two that we choose in the following way. Let 2^k be the largest amount we pay in any single month. Then, we necessarily must have paid $2^0+2^1+\cdots+2^k$ "on the way up" and then an extra $2^{k-1} + 2^{k-2} + \cdots + 2^0$ "on the way down". This means we pay a minimum of $2(2^0 + 2^1 + \cdots + 2^{k-1}) + 2^k = 2(2^k - 1) + 2^k = 3 \cdot 2^k - 2$ so necessarily $N \geq 3 \cdot 2^k - 2$. We choose the largest k that satisfies such an inequality. Note that this inequality is equivalent to $(N + 2)/3 > 2^k$, which means that $k = \lfloor \log_2((N + 2)/3) \rfloor$. Note also that, by maximality of k we also have $N < 3 \cdot 2^{k+1} - 2$ which means that $N < 3 \cdot 2^k - 2 + 3 \cdot 2^k$, i.e., $N - (3 \cdot 2^k - 2) < 3 \cdot 2^k$. Thus, the additional amount A you have to pay

satisfies $A < 3 \cdot 2^k$.

- If $A = 0$ you are done; if $1 \leq A < 2^k$ you represent A in binary and see what powers of 2 you have to pay twice, either on your way up to 2^k or on your way down.
- If $2^k \leq A < 2^{k+1} = 2 \cdot 2^k$ you pay the amount 2^k twice (in total) and then you are left with an additional amount $0 \leq B \leq 2^k - 1$ which you pay as in the previous case.
- If $2^{k+1} = 2 \cdot 2^k \leq A < 3 \cdot 2^k$ then you pay 2^k three times in total and you are left with an additional amount to pay $0 \leq C < 2^k$ which you again represent in binary and see which of the amounts from $1 = 2^0$ to 2^{k-1} you have to pay twice.

Proof of the correctness: well it should be obvious from the algorithm description, but you can try to give a more rigorous proof, similar to the proof of optimality of the greedy giving change in coins algorithm.