2 Bounding Sums

Solution. *There are many possible solutions.*

$$f_1(i) = 2^i : \sum_{i=1}^{n} 2^i = 2^{n+1} - 2 \in \Theta\left(2^n\right)$$
$$f_2(i) = i : \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta\left(n^2\right) \neq \Theta(n).$$

3 True and False Practice

(a) True. The cut property holds even with negative edges. We can add a constant to each edge to make all positives and run Kruskal's if we want positive edges.

(b) False. Longer paths are penalized in G'. This means that a shortest path with many edges in G is no longer the shortest path in G'. We consider such graph below:

$$\begin{tikzcd} L \arrow[r, "3"'] \arrow[rr, bend left, "1"] & A \arrow[r, "-4"'] & Y \end{tikzcd}$$

the shortest path is $L \longrightarrow A \longrightarrow Y$, then we add a large positive constant, i.e. 5, to each edge, making all edges positive:

$$\begin{tikzcd} L \arrow[r, "8"'] \arrow[rr, bend left, "6"] & A \arrow[r, "1"'] & Y \end{tikzcd}$$

the shortest path is replaced with $L \longrightarrow Y$.

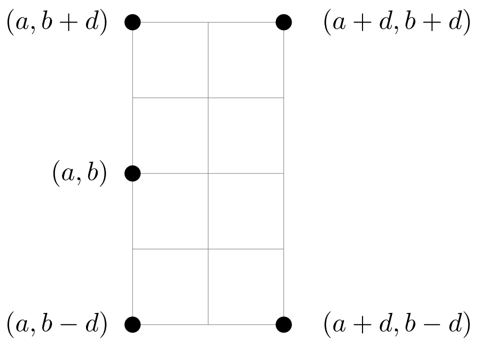
(c) False. Consider the graph below: a topological sort of the vertices would be A, B, C, but using the method in this question, we get A, C, B.

$$\begin{tikzcd} A \arrow[r, "5"'] \arrow[rr, bend left, "1"] & B \arrow[r, "2"'] & C \end{tikzcd}$$

(d) False. Consider the case when the new vertex u in G' has edges to all other vertices in G'. If each of these edges from u are lighter than any edge of G, then the MST of G' is the set of all edges from u.

4 Agent Meetup

(a) We can fit the 8 agents in this rectangle (visualized below) without two of these agents being Manhattan distance d or less apart as follows:



At most one agent can be in each square, since the distance between every pair of points within each square is at most d. Since we assume all locations are integer, a better bound is possible but likely not as straightforward.

(b) We give an algorithm whose runtime is $\mathcal{O}(n \log^2 n)$. It is possible to improve it to $\mathcal{O}(n \log n)$ by optimizing parts of the algorithm, to simplify the solution we won't bother to do so here.

Solution. *The high level idea is similar to maximum subarray sum. Define L and R to be the left and right half of the agents when sorted by x-coordinate; why this is the right way to split the agents will become clear later.*
The closest pair of agents are either (1) both in L, (2) both in R, or (3) one is in L and one is in R.
We can handle cases (1) and (2) recursively. It might seem like to handle case (3) we have to find the closest distance between an agent in L and an agent in R. The key idea is that letting the smallest distance we found in cases (1) and (2) be $d+1$, we only need to consider pairs of agents in case (3) that are distance at most d apart (recall all distances are integer). This idea will allow us ignore many pairs of agents in L and R.
If m_x is the median x-coordinate of all agents, note that any agent in R with $x_i > m_x + d$ can't be distance d or less from any agent in L. So let R_{close} be all agents in R for which $x_i \leq m_x + d$; we only need to consider agents in R_{close} in case (3). Now consider any agent i in L at position (x_i, y_i). We can skip computing the distance between agent i and any agent in R_{close} whose y-coordinate does not lie in the range $[y_i - d, y_i + d]$. If we sort R_{close} by y-coordinate beforehand, we can quickly identify agents in R_{close} in this range by binary searching.
So our non-recursive work is: for each agent i in L, identify the agents in R_{close} in the y-coordinate range $[y_i - d, y_i + d]$ and compute the distance between i and these agents. We then output the smallest distance we found either between the recursive calls and this non-recursive step.
As a base case, if there are only two agents, we can just report the distance between them.

Proof. If there are only two agents, our algorithm is of course correct. Otherwise, we proceed by induction.
If the closest pair of agents are distance $d+1$ apart and both in L or both in R, one of the recursive calls returns the right answer by our inductive hypothesis. Otherwise, the closest pair of agents is distance at most d apart and split between L and R. By part (a) and the definition of R_{close}, every pair of agents in $L \times R$ at distance at most d has their distance computed by the algorithm, in which case the final output will be correct. □

Runtime Analysis. *The non-recursive work we do is as follows.*

- *Sorting the list of all agents by x-coordinate, which takes $\mathcal{O}(n \log n)$ time.*
- *Sorting R_{close} by y-coordinate, which takes $\mathcal{O}(n \log n)$ time.*
- *Locating the agents in R_{close} to compare agents in L. Since this takes $\mathcal{O}(\log n)$ time per agent, this overall takes $\mathcal{O}(n \log n)$ time.*
- *By part (a), for each agent $i \in L$, there are $\mathcal{O}(1)$ agents j for which $m_x \leq x_j \leq m_x + d$, $y_i - d \leq y_j \leq y_i + d$. So we do $\mathcal{O}(n)$ non-recursive distance computations in $\mathcal{O}(n)$ time.*

So the recurrence relation is $\mathcal{T}(n) = 2\mathcal{T}(n/2) + \mathcal{O}(n \log n)$. We can't use the master theorem, but drawing the tree of subproblems, we can see the i th level of recursion has 2^i subproblems doing $\mathcal{O}(n/2^i \log(n/2^i))$ work. So the work per level is $\mathcal{O}(n \log n)$, i.e. the total work is $\mathcal{O}(n \log^2 n)$. (We can also show a lower bound of $\Omega(n \log^2 n)$ since levels 0 to $\frac{1}{2}\log n$ do $\Omega(n \log n)$ work.)

5 Money Changing

(a) A can be expressed as a linear combination of the x_i if and only if $x_i = 1$ for some i. If one of your denominations x_i is 1, you will certainly be able to express every integer A as $\sum_{i=1}^{n} a_i x_i$ for some non-negative integers $a_1, \ldots, a_n$. Conversely, in order to express $A = 1$ as a linear combination, you must have $x_i = 1$ for some i.

(b) Order your denominations such that $x_1 > x_2 > \cdots > x_n$. Then the greedy algorithm for this problem would be: Given A, let a1 be the largest integer such that $a_1 x_1 \leq A$. If $A - a_1 x_1 > 0$, let a_2 be the largest integer such that $a_2 x_2 \leq A - a_1 x_1$. If you have nothing left over after doing this for $i = 1, \ldots, n$, then $A = \sum_{i=1}^{n} a_i x_i$.

(c) Since 1 divides 5 and 5 divides 10, it is clear that if we have a case in which the greedy algorithm would not find the optimal solution, it must involve 25, i.e. A must be greater than 25.
Note that $x_4 = 1$ cent, $x_3 = 5$ cent, and so on.
Assume the greedy algorithm does not find the optimal solution for A, $A > 25$. Then

$$A = \sum_{i=1}^{4} a_i x_i = \sum_{i=1}^{4} b_i x_i$$

and

$$\sum_{i=1}^{4} a_i > \sum_{i=1}^{4} b_i$$

where the a_i were determined by the greedy algorithm and the b_i are optimal in that $\sum_{i=1}^{4} b_i$ is minimal.

Obviously, $a_4 = b_4$ [since $a_4 \leq 4$, any change of the number of 1 cent coins must occur in 5 unit steps to give the same sum-this is obviously worse than changing b_3]. Also, since the other denominations are 5, 10, 25, the number of 1 cent coins that the optimal algorithm takes must be $A \mod 5$, which is the number of 1 cent coins our greedy algorithm takes too. In addition to that note that $a_3 \leq 1$.
By the above considerations we must have $a_1 > b_1$. Why? Because our greedy algorithm can certainly not pick less 25-cent coins than the optimal algorithm. The first thing our greedy algorithm does is pick as many 25-cent coins as possible! Also, a_1 is not equal to b_1, because if it were, then we know that our greedy algorithm correctly picks the optimal set of coins until $A = 24$ anyway (since 1 divides 5 and 5 divides 10.)
So, let $x := a_1 b_1$. Note that x is a positive number.
For a_2, b_2 have three cases to consider: $a_2 = b_2$, $a_2 > b_2$ and $a_2 < b_2$.
Let's set $y := a_2 b_2$.
Now, remember that

$$\sum_{i=1}^{4} a_i x_i = \sum_{i=1}^{4} b_i x_i$$

We can rewrite this as

$$b_3 = 5x + 2y + a_3$$

using the actual values of x_i, the fact that $a_4 = b_4$, and our definitions of x and y.
Thus the number of coins changes by

$$\sum_{i=1}^{4} b_i - \sum_{i=1}^{4} a_i = 4x + y$$

If we can show that this number is positive, this is a contradiction and we are done.
since we expected

$$\sum_{i=1}^{4} a_i > \sum_{i=1}^{4} b_i$$

In cases 1 and 2, x and y are ≥ 0. Therefore $4x + y$ is clearly positive.
In case 3, y is negative. But, as we have to ensure that $b_3 = 5x + 2y + a_3 \leq 0$ and we know that a_3 is at most 1, we have

$$y \geq \frac{-5}{2}x - \frac{1}{2}$$

Hence

$$4x + y \geq \frac{3}{2}x - \frac{1}{2}$$

and it is again positive.

(d) A couple of real world examples:

- The United States of America 1875 − 1878 had 25 cent, 20 cent, 10 cent and 5 cent coins (and no 40 cent coins). To get 40 cents, the greedy algorithm gives 25 — 10 — 5, i.e. three coins, whereas the minimum is two coins (2020).
- Cyprus in 1901 had 18 Piastres, 9 Piastres, 4.5 Piastres and 3 Piastres Silver coins and 1 Piastre, 0.5 Piastre and 0.25 Piastre Bronze coins.
  To get 6 Piastres, the greedy algorithm would take 4.5, 1 and 0.5 Piastre coins (three coins), whereas the minimum would be two 3 Piastre coins.

6 Box Union

At any given time, let

- $u(s)$: the number of boxes under box s.
- $size(s)$: the total number of boxes in a given stack, which we additionally store in s if s is a root.
- $z(s)$: the augmented field stored at node s.
  when s is a root node, $z(s) = u(s)$,
  and otherwise $z(s) = u(s) - u(p(s))$ where $p(s)$ denotes the parent of s in the disjoint forest data structure.

In the disjoint forest union, we perform a $find(s)$ operation, and output the sum of $z(s')$ for s' in the path between s and the root r; this sum can be verified to equal $u(s)$.
When the data structure is initialized, setting all $z(s)$ to 0, along with setting $size(s)$ to 1 maintains the invariant.

Whenever a union operation is performed, one root s is made a child of another root s', in this case, we

1. if s' is in the "upper" stack, update

$$z_{new}(s') := u_{new}(s') = u_{old}(s') + size(s) = z_{old}(s') + size(s)$$

   and

$$z_{new}(s) := u_{new}(s) - u_{new}(s') = z_{old}(s) - z_{new}(s') = z_{old}(s) - z_{old}(s') - size(s)$$

   otherwise if s' is in the "lower" stack, keep $z(s')$ unchanged and update

$$z_{new}(s) := u_{new}(s) - u_{new}(s') = z_{old}(s) + size(s') - z_{new}(s') = z_{old}(s) + size(s') - z_{old}(s')$$

2. replace $size(s')$ with $size(s) + size(s')$.

Before a path compression operation is performed, we can compute the value of $u(s)$ for all s whose parent is updated to root r and replace each $z(s)$ with $u(s) - z(r)$.